Open Geospatial Consortium, Inc.

OpenGIS® Simple Features Specification

For CORBA

Revision 1.1

Date: June 2, 1999

License Agreement

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD.

THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER'S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR's sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable

Table of Contents

0 Preface

0.1 Submitting Companies

The following companies submitted this implementation specification in response to the OGC Request 1, Open Geodata Model Working Group, A Request for Proposals: OpenGIS Features (OpenGIS Project Document Number 96-021): Item 3 Simple Feature API for CORBA:

Bentley Systems, Inc.

Environmental Systems Research Institute (ESRI)

Genasys II, Inc.

Oracle Corporation

Sun Microsystems, Inc

University of California at Los Angeles (UCLA)

0.2 Submission Contact Points

All questions about this submission should be directed to:

Brian Gottier
Bentley Systems, Inc.
690 Pennsylvania Drive
Exton, PA 19341, USA
phone: +1 610 458 5000
fax: +1 610 458 1480
email: brian.gottier@bentley.com

David Beddoe
ESRI National Accounts
2070 Chain Bridge Rd, Suite 180
Vienna, VA 22182-2536, USA
phone: +1 703 506 9515
fax: +1 703 506 9514
email: dbeddoe@esri.com

John Davidson
Genasys II, Inc.
107 C. West Federal St. Suite 12

Middleburg, VA 22117, USA
phone: +1 703 648 2490
fax: +1 703 648 2492
email: johnd@genasys.com

Dr John R Herring
Oracle Corporation
196 VanBuren Street
Herndon, VA 22070, USA
phone: +1 703 736 8124
fax: +1 703 708 7233
email: jrherring us.oracle.com

Michael Cosentino
Sun Microsystems, Inc.
901 San Antonio Rd
Mail Stop MPK 15-210
Palo Alto, CA 94303, USA
phone: +1 650 786 4847
email: michael.costentino@eng.sun.com

Prof. Richard Muntz
UCLA Data Mining Laboratory
3277 Boelter Hall
Los Angeles, CA 90025, USA
phone: +1 310 825 3546
fax: +1 310 825 2273
email: muntz@cs.ucla.edu

0.3 Document Conventions

Two different fonts have been employed to disambiguate between CORBA IDL interfaces and CORBA object instances. CORBA IDL is indicated using `this font`. CORBA object instances are indicated using this font.

0.4 Revision History

Revision 1.0 includes the following changes from Revision 0:

- Inserted sections 3.2.6.2 and 3.2.6.3. The source for this change was proposal #2 and #3 from Revision Request 97-400.

- Many minor cleanups made. The source for these changes was Revision Request 97-404.

Revision 1.1 includes the following changes from Revision 1.0:

- The `query evaluator` interface has been simplified. `QueryableFeatureCollections` only allow queries on the contained features, and always return a feature collection.

- The interfaces of `Feature` and `FeatureIterator` have been aligned.

0.5 Editorial Notes

1 Overview

The purpose of this specification is to provide interfaces to allow GIS software engineers to develop applications that expose functionality required to access and manipulate geospatial information comprising features with 'simple' geometry using OMG's CORBA technology.

CORBA (Common Object Request Broker Architecture) provides a specification for the object-oriented distributed systems in a language, operating system, platform and vendor independent way.

CORBA consists of a core implemented by the various commercially available ORBs (Object Request Brokers) and a number of specified object services and application facilities (CORBAservices and CORBAfacilities). Facilities may be horizontal (provide services of a general nature to applications such as a graphic user interface or task management) or vertical (provide services targeted to a particular industry or domain). It is envisaged that this specification will become a candidate for inclusion in the OMG's work as a vertical CORBAfacility covering geospatial information management (see Figure 1.1). More information on the OMG and CORBA may be found in [3].

In the design of this specification, the approach has been to use, where possible, existing CORBA specifications to allow leveraging of the past and present efforts of OMG and vendors of other CORBA compliant products and specifications. Where it has been deemed inappropriate, alternative specifications have been developed that follow as closely as possible existing CORBA specifications.

To use CORBA, it has been necessary to describe the abstract architecture defined in the Abstract Specification [1] in the language of CORBA's OMA (Object Management Architecture). This has been achieved through the creation of interfaces defined in CORBA-IDL (Interface Definition Language) to represent the various OpenGIS constructs. These interfaces form two sub-systems or modules: the Feature Module and the Geometry Module.

The Feature and Geometry Models, which form the basic architecture of these modules, are described in Chapter 2. Chapter 3 describes each of the interfaces and structures in CORBA IDL and natural language. Chapters 4 and 5 elucidate some of the design choices and assumptions that were made during the specification process. Chapter 4 addresses the issue of Feature Identity. Chapter 5 addresses the issue of Feature Type. References are provided in Chapter 6. A full IDL specification is repeated in Chapter 7 for the convenience of developers.

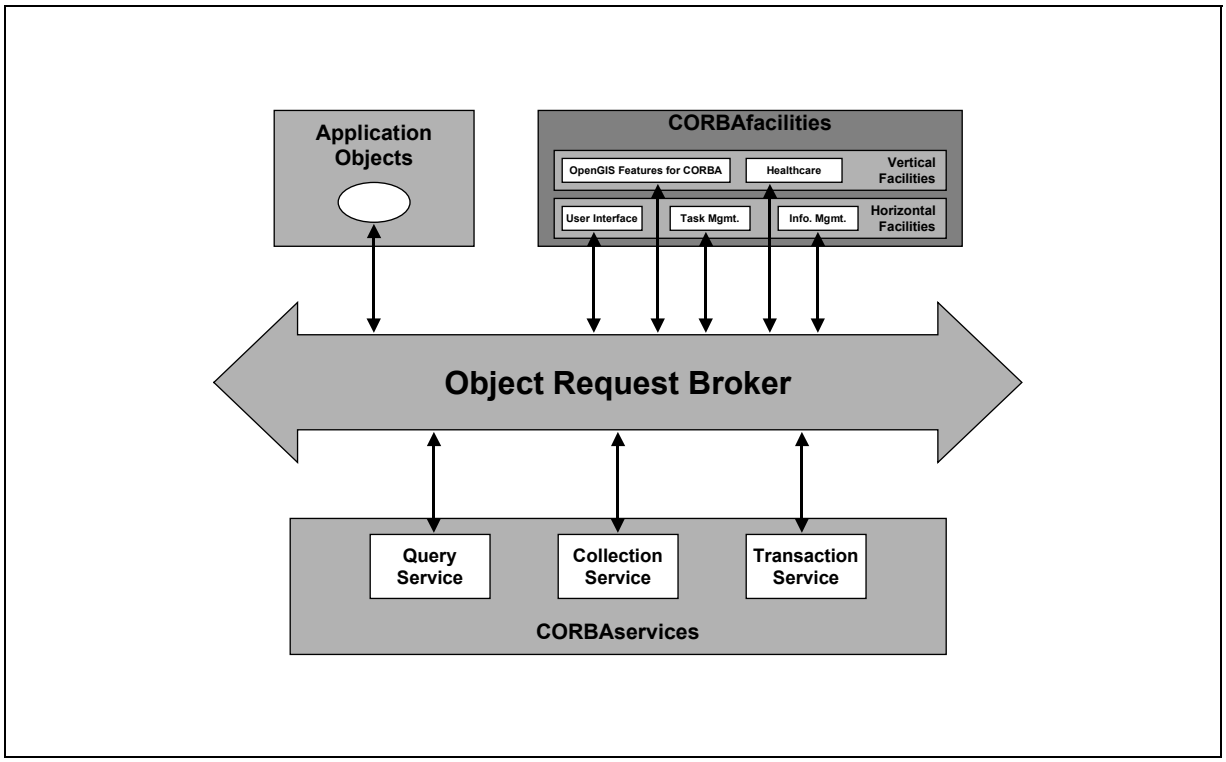

Figure 1.1— The Architecture of OMG's CORBA and the possible role of an OpenGIS Features for CORBA Specification.

The specification is broad enough to allow maximal flexibility in implementation. In particular, it has been designed with two implementation models is mind:

the exposure of existing (legacy) geospatial data and applications whether they be RDBMS or proprietary file repositories through some form of object 'wrapping'.

the development of new distributed object-oriented GIS applications.

This was undertaken to ensure that the OpenGIS Interoperability specification provides a low cost entry point for existing players in the GIS marketplace while allowing a natural progression towards implementations based on the increasingly popular and powerful distributed and object-oriented technologies such as Java and the Internet. In particular, care was taken to ensure that the powerful aspects of the O-O programming paradigm were exploitable through this specification.

To promote cross-platform interoperability a cooperative effort was made with the authors of the OLE/COM and SQL item submissions to agree on the semantics of various constructs and operations not adequately defined in the Abstract Specification. The Geometry Model defined in this specification is semantically identical to those included in all submissions. Interoperability at the Geometry level may therefore be achieved by the development of a syntactic bridge between the platforms. Such bridges would be a straight-forward development task using software currently available which is based on the OMG COM-CORBA Interworking specification. Additionally, semantic concordance between the specifications will facilitate the development of applications compliant with multiple OpenGIS Interoperability standards.

2 Architecture

2.1 Feature Model Architecture:

This specification employs the following model for creating, accessing and querying simple geospatial *features* and *feature collections*.

Real world entities such as "Roads" are typically represented as features comprising a set of spatial and non-spatial attribute values (e.g., a geometry such as a line string representing the road's spatial extent, a string representing its name, etc.). Features may have an associated set of operations or behavior. Features are also referred to as *feature instances*.

System engineers categorize representations of these real world entities as *feature type*s. A feature type defines the set of properties (and possibly behavior) that characterize features of that type. A *property* has a name and type. Properties may be of any (IDL) type, including simple types (shorts, longs, floats, strings etc.), constructed types (structs, unions, sequences) and object references (including references to other features). Every feature is of primarily one type (systems using type inheritance may allow features to be of multiple types – the use of inheritance, whether single or multiple, is an implementation issue).

Feature collections are groups of features constructed for various purposes. Feature collections come in two fundamental flavors. Feature collections supporting the concept of "containment" *own* their constituent features i.e. the persistence of the member features is effected through the collection. If a feature is removed from its containing feature collection (without being moved to another) it ceases to exist. Other feature collections provide support for organizing and managing *existing* features without owning them i.e. features that are contained in other feature collections but which are grouped in some way towards a particular end: e.g. to scope a query. Features are contained in one and only one container feature collection, although they may be, by reference, members of other, non-containing, feature collections. A client's primary access to a given feature will typically be through a feature collection.

All feature collections may also be considered to be features in their own right and may have various associated properties and associated types. Thus, Features, may be simple or composite (Features constructed from sub-Features). This specification only covers simple (atomic) Features.

A containing feature collection must also provide clients with the various feature types it may contain on demand. This collection of feature types is the *feature schema* of the collection.

An *OpenGIS Server* or implementation provides the software that exposes these constructs to outside clients through specified interfaces. Figure 2.1 provides a diagrammatic representation of how the various interfaces relate to each other. A typical server application will expose a `ContainerFeature-Collection` interface to clients, either through a static (hard-coded) binding or through a dynamic binding mechanism such

as the CORBA Naming and Trader Services (the 'White Pages' and 'Yellow Pages', respectively, of a CORBA system). A ContainerFeatureCollection is notionally equivalent to a 'GIS database'. It contains or 'owns' a set of Features each which may be exposed to clients through the `Feature` interface. Features have a set of properties, which may be iterated over by a FeaturePropertyIterator. A FeatureType object defines the properties within this set, which is available through the `Feature` interface. The FeatureType object groups `PropertyDef` structures defining each property. These may be accessed through a PropertyDefIterator manufactured on demand by the FeatureType object.

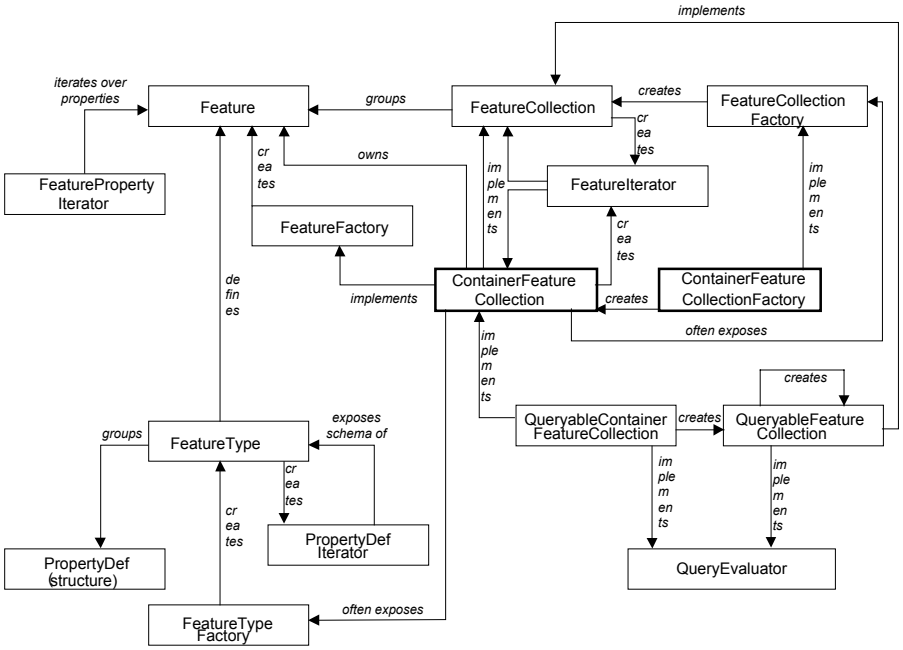

Figure 2.1—The interfaces exposing the OpenGIS Feature Model. Those in bold will typically be advertised in a naming or trader service.

Systems may enable client applications to create new Feature objects through a FeatureFactory object. Usually the ContainerFeatureCollection will assume this role. Some ContainerFeatureCollections may also expose a FeatureTypeFactory to allow client applications to create new FeatureTypes.

FeatureCollection objects may group features. ContainerFeatureCollections are a special type of FeatureCollection: those that own their member Features. In general FeatureCollections only refer to their member Features. Such referential collections may group a sub-set of Features within a single ContainerFeatureCollection (which may be created by a client application through a FeatureCollectionFactory object exposed by the containing collection) or they may group Features from a number of containers. Both referential FeatureCollections and ContainerFeatureCollections may provide query functionality to clients by implementing the `QueryEvaluator` interface (i.e. by exposing the `QueryableFeatureCollection` or `QueryableContainerFeatureCollection` interfaces). Clients may issue SQL or OQL queries against these collections. The results are returned through a `QueryableFeatureCollection`.

ContainerFeatureCollections will most often be built by system designers using the native tools of a commercial GIS implementation. Some GIS software vendors may provide the functionality to create new

ContainerFeatureCollections through a client application using a ContainerFeatureCollectionFactory. Such objects would need to be bound to either statically or through a Naming or Trader Service.

2.2 Geometry Model Architecture

Interoperable geoprocessing requires the unambiguous exposure of geometric entities. The set of interfaces included in this proposal provides a means through which various geoprocessors may expose geometric entities to each other. The interfaces are based on the following abstract model.

All geometric entities belong to an abstract class of 'geometries'. All have a number of common characteristics e.g. all have spatial extent, all use some form of spatial reference system, etc.

Geometries are categorized by their dimension as zero-dimensional geometries (points), one-dimensional geometries (curves) and two-dimensional geometries (surfaces). This model can be extended to three dimensions (solids), four dimensions (hyper-solids) and higher dimensions if necessary.

These dimension-based categories are also abstract (with the exception of points). They may be further sub-divided into more specific categories with various properties which are sufficiently well constrained to be realized unambiguously by software implementations (i.e. are concrete). For example, a specific sub-set of curves called line-strings are defined by an ordered series of points with linear interpretation between points. These may be implemented in various ways by software developers and are therefore concrete.

Additionally, individual ('simple') geometries may be aggregated to form composite geometries or geometry collections. These geometry collections form a separate category of geometries. In general, geometry collections may be composed from geometries of different dimensionality (heterogeneous collections). Geometry collections, which comprise only geometries of a single dimension (homogenous collections), are specializations of this general type. Geometry collections may, of course, be further restricted by various implementations.

All Geometries are capable of exposing their underlying coordinate geometries in the form of Well-known Structures (WKSs). The semantics of the coordinates (i.e. the mapping between coordinates in coordinate space and real world locations) is provided by a Spatial Reference System (SRS). This Specification embraces the European Petroleum Survey Group (EPSG) work in this area which has also been included in the GeoTIFF specification and the Epicentre Model of the Petrotechnical Open Software Consortium (POSC) (see [6]).

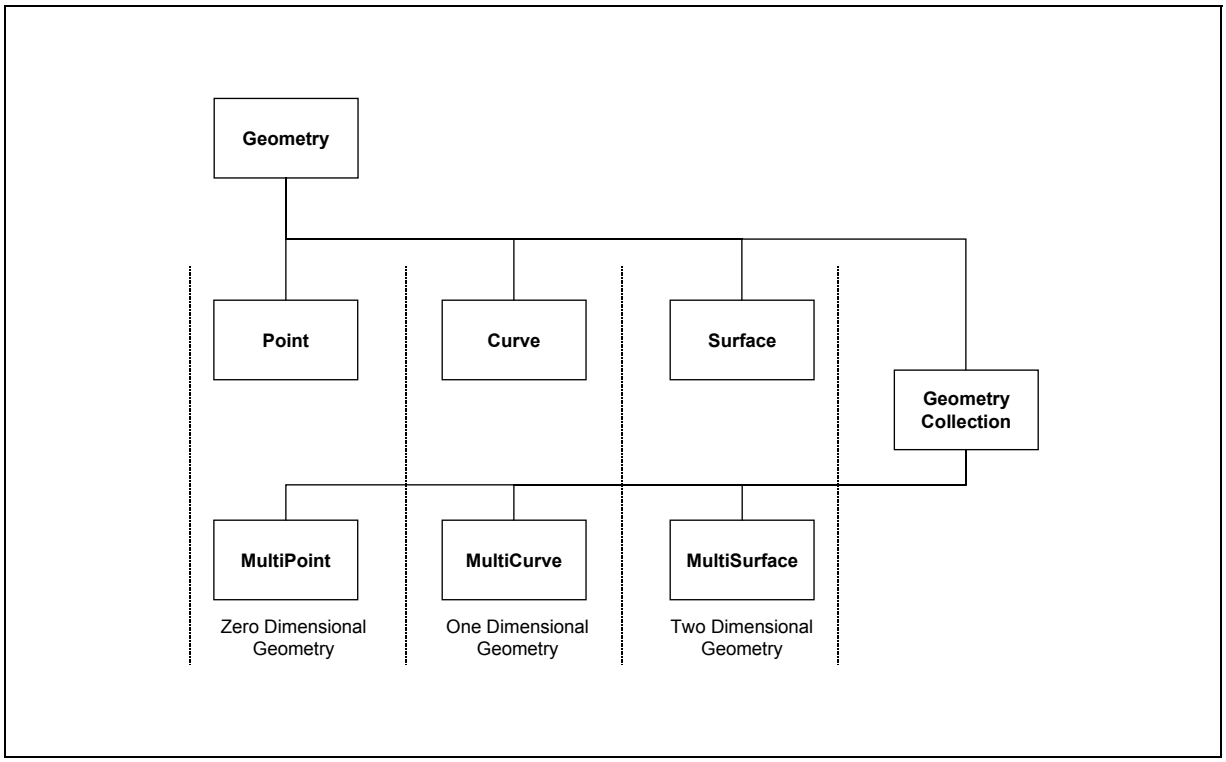

Figure 2.2—Abstract Model of OpenGIS geometry Interfaces

A Spatial Reference System may be composed of a number of components which vary in number and type depending on the type of the spatial reference system. Most SRSs abstract the shape of the earth to an ellipsoid. The mapping between this mathematical surface and the earth's surface is defined by a horizontal datum. The ellipsoidal or geographic coordinates may be used to define geometries directly. Such an SRS is a Geographic Coordinate System. These coordinates may also be mapped from the ellipsoid to a plane using some form of projection. This mapping forms a Projected Coordinate System. Projected and Geographic Coordinate Systems are grouped in this specification as GeodeticSpatialReferenceSystems to distinguish them from other forms of SRSs described in the Abstract Specification.

The other components of an SRS (unit definitions, prime meridians, etc.) described in the Epicentre model are exposed as interfaces in this specification.

3 Component Specifications

This specification has two major components or modules: the feature module and the geometry module.

3.1 Feature Module

3.1.1 Feature Related Interfaces

3.1.1.1 Feature Interface

3.1.1.1.1 Purpose

The `Feature` interface is intended to give clients access to a Feature object's instance data. It is generic: every Feature object regardless of its type conforms to this interface.

3.1.1.1.2 IDL Specification

```
interface PropertyDefIterator; // forward declaration
interface Geometry;
interface FeaturePropertySetIterator;
interface FeatureType;           // forward declaration

typedef sequence<FeatureType> FeatureTypeSeq;

typedef string Istring;
typedef sequence<Istring> IStringSeq;

// Structure to describe name-value pairs
struct NVPair {
     Istring    name;   // name is a string
     any        value;  // value is an 'any' type
};

typedef sequence <NVPair> NVPairSeq;

struct FeatureData {
     FeatureType   type;
     NVPairSeq     props;
};
typedef sequence<FeatureData> FeatureDataSeq;

struct PropertyDef {
     Istring       name;
     TypeCode      type;
```

```
    boolean       required;
};

typedef sequence<PropertyDef> PropertyDefSeq;

struct WKSPoint {
    double    x;
    double    y;
};

typedef  sequence<WKSPoint>     WKSPointSeq;
typedef  sequence<WKSPoint>     WKSLineString;
typedef  sequence<WKSLineString>  WKSLineStringSeq;
typedef  sequence<WKSPoint>     WKSLinearRing;
typedef  sequence<WKSLinearRing> WKSLinearRingSeq;

struct WKSLinearPolygon {
    WKSLinearRing      externalBoundary;
    WKSLinearRingSeq   internalBoundaries;
};

typedef sequence <WKSLinearPolygon> WKSLinearPolygonSeq;

enum WKSType {
    WKSPointType,WKSMultiPointType, WKSLineStringType,WKSMultiLineStringType,
    WKSLinearRingType, WKSLinearPolygonType, WKSMultiLinearPolygonType,
    WKSCollectionType
};

union WKSGeometry       // near-equivalent to the 'CoordinateGeometry of the spec'

    switch (WKSType) {

        case WKSPointType:
            WKSPoint      point;

        case WKSMultiPointType:
            WKSPointSeq          multi_point;

        case WKSLineStringType:
            WKSLineString line_string;

        case WKSMultiLineStringType:
            WKSLineStringSeq     multi_line_string;

        case WKSLinearRingType:
            WKSLinearRing linear_ring;

        case WKSLinearPolygonType:
            WKSLinearPolygon   linear_polygon;

        case WKSMultiLinearPolygonType:
            WKSLinearPolygonSeq        multi_linear_polygon;

        case WKSCollectionType:
            sequence<WKSGeometry>   collection;
    };


typedef sequence<octet> OctetSeq;

struct Decimal {
        long      precision;
        long      scale;
        OctetSeq  value;
};

interface Feature {
    exception InvalidParams {string why;};

    exception PropertyNotSet {}; // Property does not exist.
```

```
    exception InvalidProperty {}; // Not a valid property for the given feature.
    exception InvalidValue {}; // value is not valid for property
    exception InvalidConversion {};
    exception RequiredProperty {}; // property is required for the given feature

    // feature type
    readonly attribute     FeatureType   feature_type;

    // geometry
    Geometry       get_geometry (in NVPairSeq geometry_context) raises (InvalidParams);

    // generic property methods to get/set property values
    boolean property_exists(in Istring name) raises(InvalidProperty);

    any get_property(in Istring name) raises(PropertyNotSet,InvalidProperty);

    void set_property(in Istring name, in any value)
                              raises(InvalidProperty, InvalidValue);

    void delete_property(in Istring name) raises(PropertyNotSet, InvalidProperty,
                                                 RequiredProperty);

   // accessing property values by property names
    string        get_string_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    float         get_float_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    double        get_double_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    long          get_long_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    short         get_short_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    boolean       get_boolean_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    Decimal       get_decimal_by_name(in Istring propertyName)
                  raises (InvalidConversion, InvalidProperty);

    OctetSeq      get_byte_stream_by_name(in Istring propertyName)
                  raises (InvalidConversion, InvalidProperty);

    Geometry      get_geometry_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    WKSGeometry   get_wksgeometry_by_name(in Istring propertyName)
                  raises (InvalidConversion, InvalidProperty);

    OctetSeq      get_wkbgeometry_by_name(in Istring propertyName)
                  raises (InvalidConversion, InvalidProperty);


    NVPairSeq get_property_sequence(in unsigned long n);

    FeaturePropertySetIterator get_property_iterator();

    void destroy();

};

typedef sequence<Feature> FeatureSeq;
```

3.1.1.1.3 Interface Description

`feature_type`—According to the abstract specification [**1**] each feature has type. This implementation specification proposes to represent type through a separate interface: `FeatureType`. This read only attribute gives clients access to the feature's type information including property names and types.

`get_geometry`—aids in the discovery of geometric attributes. The Abstract Specification states that features have properties, a subset of which, are geometric. A client may gain access to these geometric properties though the `get_property`() operation but this would require the client to either know the name of particular geometric properties or discover them through the feature's associated `FeatureType` interface. Even with this knowledge a client would need to make a choice as to which of potentially several geometric attributes it requires. The `get_geometry()` operation makes this process easier for clients, by allowing the server to make many of these choices on the basis of optional context information supplied to it by the client. Examples of contextual information include scale (a suitable geometric representation of a feature at 1:500 could well be different from that at 1:5000000), theme (e.g. hydrological vs. topographic) and location (e.g. a clipping polygon). The form of this context information is a name-value list (`NVPairSeq`). A server instance may extract and process any of the supplied context fields; the server is not required to use this information in any way. In due course, context names and semantics may need to be specified by the OGC or various GICs. A client can also use `get_geometry`() to retrieve a default geometric representation of a feature in a context free manner (the `getDefaultGeometry()` of Bentley's first submission). `get_geometry`() returns a reference to a Geometry object which may be a point, curve, surface or compound geometry (collection).

`property_exists`—checks to see if a property is set in this list. The exception `InvalidProperty` is thrown if the property is not a part of the FeatureType supported by the feature.

`get_property`–gets the value associated with a property name. This operation raises a `PropertyNotSet` exception if the named property does not exist. An `InvalidProperty` exception is thrown if the property is not a part of the FeatureType supported by the feature.

`set_property`—sets the named property to the supplied value. If the property is not defined (does not exist), but is a part of the feature's property schema, then this operation creates the property. An `InvalidProperty` exception is raised if the property is invalid for a given feature and an `InvalidValue` exception is raised if the value is not within the valid domain of the named property.

`delete_property`—deletes the value of the property with the given name and sets that property to null. Raises `PropertyNotSet` if specified property is not defined. Raises `RequiredProperty` if the property is required. This operation raises an `InvalidProperty` exception if the property is invalid for a given feature.

`Accessors`—The methods of the form `get_`*`X`*`_by_name()` retrieve the data from the specified property (by index or name) as the datatype *`X`*. An `InvalidProperty` is raised if the property name is. A `PropertyNotSet` is raised if the property is not set for the feature. The purpose of these accessors is to allow efficient retrieval of properties when the type is known, without the need to marshall the data into and out of anys.

`get_property_sequence`—returns at most `n` name-value pairs from the property set of the feature.

`get_property_iterator`—returns an iterator over the property set.

`destroy`—destroys the Feature instance.

3.1.1.1.2 Type Specific Feature Interfaces

No subclasses of `Feature` are necessary but they are not precluded. An OpenGIS server may, for example, define a `Road` interface that inherits from `Feature`, and expose it to clients (through IDL or the CORBA Interface Repository). Some clients may be equipped to take advantage of the additional services the `Road` interface provides. Such an interface *must* inherit from `Feature` to ensure generic clients (those unable to use such specialized interfaces) may still access feature data through the generic operations described above.

3.1.1.1.3 Feature Identity

This proposal specifies that the identity of a Feature is encapsulated into its object reference. This policy is provided for in CORBA by the upcoming Portable Object Adapter specification [2]. For a discussion of these issues see Section 4.

3.1.1.1.4 Typical Server Implementations

Typically in object-oriented environments GIS features are implemented as objects. The `Feature` interface will be a thin wrapper over these feature objects. The `get_property`() and `set_property`() operations will retrieve and modify these objects' internal properties. The implementation of the `FeatureType` attribute will typically be a class object. The `get_geometry()` method scans the Feature's property list for all properties which have "Geometry" `TypeCode`.

In RDBMSs, GIS features are typically represented as rows in various relational tables. The feature interface will allow access to the tuples within a particular row. The `get_property`() and `set_property`() operations will probably be effected by the submission of appropriate SQL calls. The underlying implementation of the FeatureType attribute will typically be a row in a system table. The table name and primary key of the row will be encoded by the server into the Feature's object reference.

3.1.1.1.5 Client Scenarios

Assume that a generic 'feature browser' client retrieved a Feature interface reference from a `FeatureIterator` (see Section 3.1.3.3). It could use the following algorithm to display the features geometric and non-geometric properties in some sort of form display:

```
// generic client pseudocode

Feature            featureRef;
FeatureType        featureTypeRef;
CORBA_Any          propertyValue;
PropertyDefSeq     propertySet;
...
// get FeatureType reference
featureTypeRef = featureRef.get_feature_type();

// get Property Names from feature type reference
propertySet = featureTypeRef.get_property_defs();

for ( i=0; i< propertySet.length; i++) { // for each property ...
     try {
          propertyValue = featureRef.get_property((propertySet.get(i)).name);

          if (isGeometry(propertyValue.type)
               drawInGraphicsWindow (propertyValue.value);
          else
               writeInTextWindow (propertyValue.value);

     } catch (InvalidPropertyName) {
```

```
     // ... exception handling
     } catch (PropertyNotSet) {
     // ... exception handling
     }
}
```

The details of `get_property_defs`() function is explained under `FeatureType` (Section 3.1.2). Note that this client uses CORBA any types, which contain type information. The client can test what type of information is contained in an any and act in an appropriate manner: in this case deciding between graphic and non-graphic output. A more sophisticated client may use this type information to determine the layout of a form-based dialog. This type information is also available from the `FeatureType` reference so the client may get an array of property types with the array of property names and directly cast the any to a variable of the appropriate type.

A dedicated client (one that is familiar with the server's type system) may also use the generic interface to retrieve data.

```
// dedicated client (generic interface) pseudocode

Feature            featureRef;
FeatureType        featureTypeRef;

CORBA_Any          propertyValue
String             roadName;
Geometry           roadCentreLine;
...
// get FeatureType reference
if (strcmp(featureRef.get_feature_type().name(),"RoadType") == 0)
     try {
          // get Road name
          propertyValue = featureRef.get_property("road_name");
          roadName = (String)propertyValue.value;

          // get Road centreLine
          propertyValue = featureRef. get_property("centre_line");
          roadCentreLine = (Geometry)propertyValue.value;
     } catch (InvalidPropertyName) {
     // ... exception handling
     }
}
```

This code only differs from the above in that the client already knows what properties a 'Road' feature will have. (There are many cases where clients can be expected to have this sort of knowledge: it may use a GIC standard; use a published IDL type schema or be an in-house solution based on commercial OpenGIS compliant software.) Such a client is, however, not restricted to using this generic interface:

```
// dedicated client (type-specific interface) pseudocode

Feature            featureRef;
FeatureType        FeatureTypeRef;

Road               roadRef;
String             roadName;
Geometry           roadCentreLine;
...
// get FeatureType reference
if (strcmp(featureRef.get_feature_type().name(),"RoadType")==0)
     roadRef = (Road)featureRef;

// get Road name
roadName = roadRef.get_road_name();

// get CentreLine
roadCentreLine = roadRef.get_centre_line();
}
```

This code will provide more performance because the switching into and out of any types has been eliminated. This is achieved at the cost of generality.

3.1.1.1.6 Rejected Approaches

Requiring the representation of all features as type-specific interfaces and its attendant requirement to insert feature type details into the Interface Repository as proposed in Bentley's first submission has been rejected on the grounds of placing unreasonable obligations on both server and client implementations. See the discussion under FeatureType interface for further details.

The UCLA proposal to only provide feature data through a 'GIS_Iterator' without persistence or identity was also rejected on the grounds of having insufficient power to fulfill the functional requirements of RFP 1.

3.1.1.2 FeaturePropertySetIterator Interface

3.1.1.2.1 Purpose

The FeaturePropertyIterator interface provides support for iterating through a given Feature object's property set. The factory of a FeaturePropertyIterator instance is a Feature object.

3.1.1.2.2 IDL Specification

```
interface FeaturePropertySetIterator {

     exception IteratorInvalid {};


     // Get next NVPair structure
     boolean  next(out NVPair the_pair)
                  raises (IteratorInvalid);

     // Get next "n" NVPair structures.
     boolean  next_n(in unsigned long n,
                  out NVPairSeq n_pairs)
                  raises (IteratorInvalid);

     // Discard the iterator
     void     destroy();

     // reset the iterator
     void     reset() raises (IteratorInvalid);

};
```

3.1.1.2.3 Interface Description

`next`—retrieves the next property name-value pair. The order of retrieved properties is implementation dependent. This operation returns true if a valid property was placed in `the_pair` and false if there are no more properties to retrieve. An `IteratorInvalid` exception is raised if the iterator has become invalid.

`next_n`—retrieves the next `n` property name-value pairs. If less than `n` properties are available, then all remaining pairs are retrieved. This operation returns true if a sequence of properties was placed in `n_pairs` and false if there are no more properties to retrieve. An `IteratorInvalid` exception is raised if the iterator has become invalid.

`destroy`—destroys the iterator object.

`reset`—resets the iterator to the first property of the set. An `IteratorInvalid` exception is raised if the iterator has become invalid.

3.1.1.3 FeatureFactory Interface

3.1.1.3.1 Purpose

The `FeatureFactory` provides support for the creation of a Feature instance of a given FeatureType.

3.1.1.3.2 IDL Specification

```
interface FeatureFactory {

     exception     FeatureTypeInvalid {string why;};
     exception     PropertiesInvalid {string why;};

     Feature       create_feature(in FeatureType type, in NVPairSeq properties)
                              raises (FeatureTypeInvalid, PropertiesInvalid);

     FeatureSeq    create_features(in FeatureDataSeq features)
                              raises (FeatureTypeInvalid, PropertiesInvalid);

};
```

3.1.1.3.3 Interface Description

`create_feature`—creates a new Feature instance of the desired type (i.e., the `FeatureType` argument). A `FeatureTypeInvalid` exception is thrown if the supplied FeatureType is not supported by the FeatureFactory instance. A `PropertiesInvalid` exception is thrown if the arguments supplied for the Feature object of the given FeatureType are incorrect or insufficient (i.e., the required set of parameters is not supplied).

`create_features`—creates a set of new Feature instances of the desired type (i.e., the `FeatureType` argument). A `FeatureTypeInvalid` exception is thrown if the supplied FeatureType is not supported by the FeatureFactory instance. A `PropertiesInvalid` exception is thrown if the arguments supplied for a given Feature object of the given FeatureType are incorrect or insufficient (i.e., the required set of parameters is not supplied).

3.1.2 Feature Type Related Interfaces

3.1.2.1 FeatureType Interface

3.1.2.1.1 Purpose

The `FeatureType` interface is intended to provide details of the type of a Feature that are described as 'Feature Schema' in the Abstract Specification's Essential Model, specifically the names and types of the properties associated with each instance of a Feature of the given FeatureType.

3.1.2.1.2 IDL Specification

```
interface FeatureType {
     exception     InheritanceUnsupported {};
```

```
    exception     PropertyDefInvalid {};

    // feature type name
    readonly attribute Istring   name;

    // feature type parents
    FeatureTypeSeq     get_parents() raises (InheritanceUnsupported);

    // feature type children
    FeatureTypeSeq     get_children()raises (InheritanceUnsupported);

    // definition of properties for this feature type
    boolean            property_def_exists(in Istring name);
    PropertyDef        get_property_def(in Istring name) raises(PropertyDefInvalid);
    PropertyDefSeq     get_property_def_sequence(in long levels, in unsigned long n);
    PropertyDefIterator    get_property_def_iterator (in long levels);
    void               destroy();
};

typedef sequence<FeatureType> FeatureTypeSeq;
```

3.1.2.1.3 Interface Description

`name`—a string naming the feature type. This will typically correspond to a table name in an RDBMS and a class name in an OO environment.

`get_parents`—returns a sequence of FeatureTypes which represent the direct ancestors of the FeatureType. The return sequence will be empty if the implementation the FeatureType is a root type. The sequence will have no more than one member if the implementation supports only single inheritance. Implementations supporting multiple inheritance will return an arbitrary length sequence. Implementations that do not support inheritance will raise an `InheritanceUnsupported` exception.

`get_children`—returns a sequence of FeatureTypes which represent the children of the FeatureType. The return sequence will be empty if the FeatureType has no children. Implementations that do not support inheritance will raise an `InheritanceUnsupported` exception.

`property_def_exists`—returns true if a property named `name` is defined.

`get_property_def`– returns the `PropertyDef` structure associated with the property `name`.

`get_property_def_sequence`–returns a sequence of at most `n` property definitions. The `levels` parameter indicates how many levels of the type hierarchy should be searched for inherited properties. A value of 0 indicates only properties defined by this FeatureType explicitly should be returned. A value of 1 indicates properties from this FeatureType and its immediate parent(s) should be returned. A value of –1 indicates that the all properties should be returned regardless of the depth of the inheritance tree. This value is ignored by implementations that do not support inheritance.

`get_property_def_iterator`—returns an iterator over the schema of the FeatureType.

`destroy`—destroys the FeatureType instance.

3.1.2.1.4 Conceptual framework

For a justification and discussion of the conceptual framework of the `FeatureType` interface, see Section 5.

3.1.2.1.5 Feature Polymorphism

A Feature is of one and only one principal type. The FeatureType referred by the Feature's `feature_type` property is its principal type. If the implementation supports inheritance (either single or multiple), a Feature may inherit properties (and behavior) from parent types.

3.1.2.1.6 Typical Server Implementations

Typically in object-oriented environments a class or factory object embodies object type. These objects are obvious candidates for the implementation of the `FeatureType` interface. They have identity, property (and method) signatures, support polymorphism and are used by the memory management system when creating new objects.

In RDBMSs, the FeatureType corresponds most closely with the data definition (schema) of a relational table. Tables also have identity (based on table name) and a set of properties or columns (retrievable from a row in a system table or data dictionary). The table name is also used in the SQL insert statement when creating a new row in the table (a new feature instance). Tables have no standard way of addressing polymorphism or feature behavior but these are not required for OpenGIS compliance.

3.1.2.1.7 Rejected Approaches

Including methods to retrieve feature type from the Feature interface directly was considered. This may have taken the form of a `get_schema()` operation. This was rejected on the grounds that such a identity-free approach to feature type would significantly diminish the power of the specification.

3.1.2.1.8 Feature Behavior & the Interface Repository

Discussions at various OpenGIS fora have indicated that the Abstract Specification's notion of a "property set" includes a set of operations defining the behavior of features of a particular type. This proposal excludes operations from the schema provided through the `FeatureType` interface. This is a pragmatic policy: CORBA already has a well-defined mechanism for the dynamic discovery of behavior: the Interface Repository (IR). In fact, the IR could be used in place of the `FeatureType` interface to expose all properties as well as behavior, but it was considered overly onerous to require all OpenGIS servers to provide IR definitions for all `FeatureTypes`.

It should be noted, however, that it is likely to become easier for server developers to offer such services as the vendor community provides increasingly comprehensive development tools in the CORBA environment. For example, a tool to automatically generate IR entries from data dictionaries in an RDBMS would remove most of the development effort of the IR approach for server implementers exposing RDBMS data and metadata. If such tools become sufficiently powerful, the `FeatureType` interface may be deprecated.

Server developers prepared to offer feature behavior might include IR entries for each type-specific Feature interface to allow dynamic discovery of behavior. A client able to dynamically discover and utilize feature behavior will use the IR to obtain details of available functionality (using the `get_interface()` operation which the `Feature` interface inherits from `CORBA_Object`). If the response to `get_interface()` is the `Feature` interface the client can assume that no feature behavior is available and that property set information is only obtainable through the `FeatureType` interface. Clients unable to dynamically discover feature behavior will always use the `FeatureType` interface. To ensure such clients can always access property information, all OpenGIS compliant servers must provide a FeatureType object for all features whether or not they also provide feature type specific interface information through the IR.

3.1.2.2 FeatureTypeFactory Interface

3.1.2.2.1 Purpose

The `FeatureTypeFactory` interface provides support for the creation of FeatureType object instances.

3.1.2.2.2 IDL Specification

```
interface FeatureTypeFactory {

     exception InvalidParams {string why;};

     FeatureType   create(in string name, in PropertyDefSeq schema,
                          in FeatureTypeSeq parents)
                          raises(InvalidParams);
};
```

3.1.2.2.3 Interface Description

`create`—creates a FeatureType instance given a type name, attribute schema, and lists of parents of the FeatureType. Raises an `InvalidParams` exception if any of the supplied arguments are invalid.

3.1.2.3 PropertyDefIterator Interface

3.1.2.3.1 Purpose

The `PropertyDefIterator` interface provides support for iterating over a FeatureType's property definitions or schema. The factory of a PropertyDefIterator instance is a FeatureType object.

3.1.2.3.2 IDL Specification

```
interface PropertyDefIterator {

     exception IteratorInvalid {};


     // Get next PropertyDef structure
     boolean     next(out PropertyDef schema_property)
                       raises (IteratorInvalid);

     // Get next "n" PropertyDef structures
     boolean     next_n(in unsigned long n,
                       out PropertyDefSeq schema_properties)
                       raises (IteratorInvalid);

     // Discard the iterator
     void        destroy();

     // reset the iterator
     void          reset() raises (IteratorInvalid);

};
```

3.1.2.3.3 Interface Description

`next`—retrieves the next property definition. This operation returns true if a property was placed in `schema_property` and false if there are no more property definitions to retrieve. An `IteratorInvalid` exception is raised if the iterator has become invalid.

`next_n`—retrieves the next `n` schema attributes. If less than `n` attributes are available, then retrieves all those remaining. This operation returns true if a sequence of property definitions was placed in `schema_properties` and false if there are no more property definitions to retrieve. An `IteratorInvalid` exception is raised if the iterator has become invalid.

`destroy`—deletes the iterator object.

`reset`—resets the iterator to the first schema attribute of the set. An `IteratorInvalid` exception is raised if the iterator has become invalid.

3.1.3 Feature Collection Related Interfaces

3.1.3.1 Feature Collection Interface

3.1.3.1.1 Purpose

This interface provides services for the management of groups of OpenGIS features. These groups can come into being for a number of reasons: e.g. a project as a whole, for the scope of a query, as the result of a query or arbitrarily selected by a user for some common manipulation. A feature's membership of a particular FeatureCollection does not *necessarily* imply any relationship with other member features. Composite or compound features which 'own' constituent member Features (e.g. an Airport composed of Terminals, Runways, Aprons, Hangars, etc) may also support the `FeatureCollection` interface to provide a generic means for clients to access constituent members without needing to be aware of the internal implementation details of the compound feature. Compound features are not specified in this proposal.

3.1.3.1.2 IDL Specification

```
interface FeatureIterator;       // forward declaration

interface FeatureCollection : Feature {

     exception     IteratorInvalid {};
     exception     PositionInvalid {};
     exception     FeatureInvalid {string why;};
     exception     PropertiesInvalid {string why;};


     readonly attribute     long          number_features;
     FeatureTypeSeq  supported_feature_types();

     void      add_element (in Feature element) raises (FeatureInvalid);
     void      merge (in FeatureCollection elements) raises (FeatureInvalid);

     void      insert_element_at (in Feature element, in FeatureIterator where)
                   raises (FeatureInvalid, IteratorInvalid);
     void      replace_element_at (in Feature element, in FeatureIterator where)
                   raises (FeatureInvalid, IteratorInvalid, PositionInvalid);

     void      remove_element_at (in FeatureIterator where)
                   raises (IteratorInvalid, PositionInvalid);
     void      remove_all_elements ();

     Feature   retrieve_element_at (in FeatureIterator where)
                   raises (IteratorInvalid, PositionInvalid);

     FeatureIterator    create_iterator();

};
```

3.1.3.1.3 Interface Description

Feature Interface inheritance—This interface inherits from the `Feature` interface. This approach is in line with the Abstract Specification that allows feature collections to be defined as features [1, paragraph 3.13.1.4]. FeatureCollections, thus, have persistence and identity mechanisms identical to those of Features. In the FeatureCollection context, the property set is constituted by attributes of the collection as a whole rather than single member Features. The specification of property types and names will, as with features, be the province of system implementers and GICs. These details will be available to clients through the FeatureCollection's associated FeatureType object. The implementation of the `get_geometry`() method will also be server dependent: it may return the collection of all features' geometries or a generalized representation of the feature collection as a whole. This allows clients which simply require a geometric representation of the collection for review (e.g. a topographic reference for thematic maps) to retrieve it directly from the collection without dealing with individual features. As part of a schema definition for a FeatureCollection instance, a user may have a property like "SupportedFeatureTypes", which the user can query to find out what FeatureTypes the given collection supports. The `destroy()` operation destroys the FeatureCollection instance. In general, the FeatureCollection's elements are *not* destroyed by this operation.

`number_features`—this attribute identifies the number of Features within the FeatureCollection.

`supported_feature_types`—this method returns the type description of all feature types supported by the FeatureCollection.

`add_element`—adds the Feature `element` to the FeatureCollection. If `element` does not adhere to all membership restrictions of the FeatureCollection a `FeatureInvalid` exception is raised.

`merge`—merges the FeatureCollection `elements` into the FeatureCollection. If a component of `elements` does not adhere to the membership restrictions of the FeatureCollection a `FeatureInvalid` exception is raised.

`insert_element_at`—inserts the Feature `element` at the position in the FeatureCollection indicated by the FeatureIterator `where`. A `FeatureInvalid` exception is raised if `element` does not conform to the membership requirements of the FeatureCollection. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the FeatureCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Feature.

`replace_element_at`—replaces the Feature `element` at the position in the FeatureCollection indicated by the FeatureIterator `where`. A `FeatureInvalid` exception is raised if `element` does not conform to the membership requirements of the FeatureCollection. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the FeatureCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Feature.

`remove_element_at`—removes the Feature at the position in the FeatureCollection indicated by the FeatureIterator `where`. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the FeatureCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Feature.

`remove_all_elements`—empties the FeatureCollection.

`retrieve_element_at`—returns the Feature at the position in the FeatureCollection indicated by the FeatureIterator `where`. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the FeatureCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Feature.

`create_iterator`—returns a FeatureIterator which may be used to access Feature objects and data.

3.1.3.1.4 Typical Server Implementations

In an OO environment there are various numbers of possible implementations of FeatureCollections which group object references together. These include arrays, linked lists, binary or B-trees, hash tables, etc. The separation of `FeatureCollection` from `FeatureIterator` hides these implementation details from clients.

In an RDBMS, a FeatureCollection will typically be implemented as a table or view.

3.1.3.2 FeatureCollectionFactory Interface

3.1.3.2.1 Purpose

The `FeatureCollectionFactory` interface provides support for the creation of FeatureCollection object instances.

3.1.3.2.2 IDL Specification

```
interface FeatureCollectionFactory {

     exception FeatureTypeInvalid {string why;};
     exception PropertyInvalid {string why;};
     exception FeatureInvalid {string why;};

     FeatureCollection  create(in FeatureType collection_type,
                           in NVPairSeq collection_properties,
                    in FeatureTypeSeq supported_feature_types)
                    raises (FeatureTypeInvalid, PropertyInvalid);

     FeatureCollection createFromCollection(in FeatureType collection_type,
                    in NVPairSeq collection_properties,
                    in FeatureTypeSeq supported_feature_types,
                    in FeatureCollection collection)
                    raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

     FeatureCollection createFromSequence(in FeatureType collection_type,
                    in NVPairSeq collection_properties,
                    in FeatureTypeSeq supported_feature_types,
                    in FeatureSeq list)
                    raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);
};
```

3.1.3.2.3 Interface Description

`create`—creates a FeatureCollection instance given the FeatureType of the collection and a set of collection properties, and a list of valid feature types that are supported by the collection. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied.

`createFromCollection`—create a FeatureCollection instance given the FeatureType of the collection, a set of collection properties, a list of valid feature types that are supported by the collection, and a FeatureCollection instance from which to copy Feature instance references. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A

`FeatureInvalid` exception is raised if the supplied Feature instances do not adhere to the collection's membership restrictions, if any.

`createFromSequence`—create a FeatureCollection instance given the FeatureType of the collection, a set of collection properties, a list of valid feature types that are supported by the collection, and a sequence of Feature instances. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied Feature instances do not adhere to the collection's membership restrictions, if any.

3.1.3.3 FeatureIterator Interface

3.1.3.3.1 Purpose

This interface provides clients with the ability to access feature data through a FeatureCollection object. FeatureCollections act as factories for FeatureIterators. This interface allows OpenGIS servers to expose Feature data without requiring the exposure of persistent Feature objects to CORBA clients. The process of CORBA Object creation may be expensive: this iterator allows the service to provide data for multiple features through a single CORBA Object.

3.1.3.3.2 IDL Specification

```
interface FeatureIterator  {

     exception IteratorInvalid {};
     exception PositionInvalid {};
     exception FeatureNotAvailable {};

     exception InvalidConversion {};
     exception InvalidProperty {};
     exception PropertyNotSet {};
     exception InvalidParameters {};

     // iterating over features
     boolean     next (out Feature the_feature)
          raises (IteratorInvalid, PositionInvalid,
          FeatureNotAvailable);

     boolean   next_n (in short n, out FeatureSeq the_features)
          raises (IteratorInvalid, PositionInvalid,
          FeatureNotAvailable);

     void advance ()
          raises (IteratorInvalid, PositionInvalid);

     Feature  current ()
          raises (IteratorInvalid, PositionInvalid,
          FeatureNotAvailable);

     // accessing current feature via 'Feature'-like methods
     FeatureType   get_feature_type();

     Geometry      get_geometry(in NVPairSeq geometry_context) raises (InvalidParameters);

     boolean       property_exists(in Istring name) raises(InvalidProperty);
     any           get_property(in Istring name) raises (PropertyNotSet,
                                                         InvalidProperty,);

     string        get_string_by_name(in Istring propertyName)
                   raises (PropertyNotSet, InvalidProperty);

     float         get_float_by_name(in Istring propertyName)
                   raises (PropertyNotSet, InvalidProperty);
```

```
    double          get_double_by_name(in Istring propertyName)
                    raises (PropertyNotSet, InvalidProperty);

    long            get_long_by_name(in Istring propertyName)
                    raises (PropertyNotSet, InvalidProperty);

    short           get_short_by_name(in Istring propertyName)
                    raises (PropertyNotSet, InvalidProperty);

    boolean         get_boolean_by_name(in Istring propertyName)
                    raises (PropertyNotSet, InvalidProperty);

    Decimal         get_decimal_by_name(in Istring propertyName)
                    raises (InvalidConversion, InvalidProperty);

    OctetSeq        get_byte_stream_by_name(in Istring propertyName)
                    raises (InvalidConversion, InvalidProperty);

    Geometry        get_geometry_by_name(in Istring propertyName)
                    raises (PropertyNotSet, InvalidProperty);

    WKSGeometry     get_wksgeometry_by_name(in Istring propertyName)
                    raises (InvalidConversion, InvalidProperty);

    OctetSeq        get_wkbgeometry_by_name(in Istring propertyName)
                    raises (InvalidConversion, InvalidProperty);


    NVPairSeq get_property_sequence(in unsigned long n);
    FeaturePropertySetIterator get_property_iterator();

    void      reset() raises (IteratorInvalid);

    boolean   more();

    void      destroy();
};
```

3.1.3.3.3 Interface Description

`current`—returns a reference to a persistent CORBA object conforming to the `Feature` interface and representing the Feature at the iterator's current position. This operation raises a `PositionInvalid` exception if the iterator is not pointing to a valid Feature. It raises a `FeatureNotAvailable` exception if the server does not support the provision of persistent CORBA object references to its Features. The operation raises a `IteratorInvalid` exception if the iterator is no longer valid: this will probably be due to operations on its underlying collection since the iterator was created (eg. additions and deletions to and from the FeatureCollection).

`next`—moves the iterator's internal pointer to the next Feature and retrieves a reference to a persistent CORBA object conforming to the `Feature` interface and representing that Feature. This operation returns true if a valid Feature reference was placed in `the_feature` and false if there are no more Features to retrieve. An `IteratorInvalid` exception is raised if the iterator has become invalid. This operation is functionally equivalent to an `advance()` invocation followed by a `current()` invocation. `next()` raises the same exceptions as `current()`. The order in which features are returned is implementation dependent. Consecutive calls to `next()` will retrieve each Feature within in the collection exactly once.

`next_n`—retrieves the next `n` Features as a sequence of CORBA object references and moves the iterator's internal pointer to the next Feature after those retrieved, if any. This operation returns true if a valid sequence of Feature references was placed in `the_feature` and false if there are no more Features to retrieve. An `IteratorInvalid` exception is raised if the iterator has become invalid.

`advance`—moves the iterator's internal pointer to the next Feature. Subsequent calls to retrieve properties will operate on the Feature at the current position.

Feature Property Access—the `FeatureIterator` interface provides the user with the ability to retrieve properties of the Feature at the current iterator position. Generic operations include `get_feature_type`, `get_property_by_name`, and `get_geometry`. `get_property_by_name`—returns the named property's data. The `get_property_sequence` and `get_property_iterator` operations provide clients with the ability to request the Feature's properties as a sequence or to get an iterator over the property set. These operations are functionally identical to their equivalents in the `Feature` interface and raise exceptions if the same manner.

`reset`—this method resets the iterator's internal pointer to the first Feature in the collection. This method will also revalidate an invalid iterator. Which Feature is deemed to be first is implementation dependent. an `IteratorInvalid` exception is raised if the iterator has become invalid.

`more`—returns TRUE if there are more features after the current Feature and FALSE otherwise.

`destroy`—deletes the FeatureIterator instance.

3.1.3.3.2 Typical Server Implementations

The separation of the `FeatureIterator` operations from the `FeatureCollection` interface allows client applications to deal with various implementations of `FeatureCollections` in a generic way. `FeatureIterators` will, in general be closely coupled with their associated FeatureCollections which act as factories for iterators. For example, in an OO environment where a `FeatureCollection` is implemented as a linked list, the `FeatureIterator` would be a wrapper around a pointer to the 'current' feature.

In an RDBMS, where a `FeatureCollection` is often implemented as a table or view, the `FeatureIterator` will typically be a cursor.

3.1.4 Container Feature Collection Interfaces

3.1.4.1 ContainerFeatureCollection Interface

3.1.4.1.1 Purpose

Typically a FeatureCollection groups its member Features by maintaining a list of references. A ContainerFeatureCollection's elements are *entirely created and owned* by it. The Features belong logically to the ContainerFeatureCollection, not just referred to by it. Therefore, a ContainerFeatureCollection acts as a *factory* for FeatureTypes that it supports. In addition, a feature added via methods like `add_element()` is *deep-copied* i.e., a new feature of the given type is created by the ContainerFeatureCollection.

3.1.4.1.2 IDL Specification

```
interface ContainerFeatureCollection : FeatureCollection, FeatureFactory {
};
```

3.1.4.1.3 Interface Description

FeatureCollection Interface inheritance—The `ContainerFeatureCollection` interface also inherits the `FeatureCollection` interface for membership access and manipulation. However, it specializes the implementation of some of these methods with "ownership" semantics rather than referential semantics (see below). The `destroy()` method destroys the ContainerFeatureCollection instance. All of the ContainerFeatureCollection's elements *are* destroyed by this operation.

`add_element`—adds the feature `element` to the ContainerFeatureCollection. If `element` does not adhere to all membership restrictions a `FeatureInvalid` exception is raised. Otherwise, a deep copy of the `Feature` is performed, and a new Feature object is created.

`merge`—merges the FeatureCollection's `elements` into the ContainerFeatureCollection. If a component of elements does not adhere to the membership restrictions of the ContainerFeature-Collection a `FeatureInvalid` exception is raised. Otherwise, a deep copy of each element Feature of the FeatureCollection is performed, and a new Feature object is created for every element.

`insert_element_at`—inserts the Feature `element` at the position in the ContainerFeatureCollection indicated by the FeatureIterator `where`. A `FeatureInvalid` exception is raised if `element` does not conform to the membership requirements of the ContainerFeatureCollection. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the ContainerFeatureCollection. Otherwise, a deep copy of the Feature is performed, and a new Feature object is created.

`replace_element_at`—replaces the Feature `element` at the position in the ContainerFeatureCollection indicated by the FeatureIterator `where`. A deep copy of the Feature is performed, and a new Feature object is created to replace the Feature located at the position indicated by the iterator. The old Feature is destroyed. A `FeatureInvalid` exception is raised if `element` does not conform to the membership requirements of the FeatureCollection. An `IteratorInvalid` exception is raised if where is not a valid iterator of the FeatureCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Feature.

`remove_element_at`—removes the Feature at the position in the ContainerFeatureCollection indicated by the FeatureIterator `where`. A successful invocation of this operation will result in the destruction of the Feature object. An `IteratorInvalid` exception is raised if where is not a valid iterator of the ContainerFeatureCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Feature.

`remove_all_elements`—empties the ContainerFeatureCollection and deletes all member Features.

`retrieve_element_at`—returns the Feature at the position in the ContainerFeatureCollection indicated by the FeatureIterator `where`. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the ContainerFeatureCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Feature.

FeatureFactory inheritance– `create_feature()` and `create_features()` create new Feature instances of the desired type (i.e., the `FeatureType` argument). A `FeatureTypeInvalid` exception is thrown if the supplied FeatureType is not a member of the collection's supported FeatureTypes. A `PropertiesInvalid` exception is thrown if the arguments supplied for the Feature object(s) of the given FeatureType are incorrect or insufficient (i.e., the required set of parameters is not supplied). The ContainerFeatureCollection wholly owns the resultant features.

3.1.4.1.4 Typical Server Implementations

The ContainerFeatureCollection, as the interface exposing creation and deletion of features will typically be implemented by the module of an application responsible for maintaining the resources required to store feature data. In an OO system this may be some form of persistent object storage (such as a file system or an integration mechanism responsible for pulling data into and out of an underlying RDBMS server). In an RDBMS implementation the ContainerFeatureCollection will typically be a wrapper over a table or a series of tables.

3.1.4.2 ContainerFeatureCollectionFactory Interface

3.1.4.2.1 Purpose

The `ContainerFeatureCollectionFactory` interface provides support for the creation of ContainerFeatureCollection object instances. Clients will typically gain access to such factories through a naming or trading service.

3.1.4.2.2 IDL Specification

```
interface ContainerFeatureCollectionFactory {

     exception     FeatureTypeInvalid {string why;};
     exception     PropertyInvalid {string why;};
     exception     FeatureInvalid {string why;};

     ContainerFeatureCollection create(in FeatureType collection_type,
                       in NVPairSeq collection_properties)
                       raises (FeatureTypeInvalid, PropertyInvalid);

     ContainerFeatureCollection createFromCollection(in FeatureType collection_type,
                       in NVPairSeq collection_properties,
                       in FeatureCollection collection)
                       raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

     ContainerFeatureCollection createFromSequence(in FeatureType collection_type,
                       in NVPairSeq collection_properties,
                       in FeatureSeq list)
                       raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);


     ContainerFeatureCollection createFromFeatureData(in FeatureType collection_type,
                       in NVPairSeq collection_properties, in FeatureDataSeq list)
                       raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

};
```

3.1.4.2.3 Interface Description

`create`—creates a ContainerFeatureCollection instance given the FeatureType of the collection and a set of collection properties. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied.

`createFromCollection`—creates a ContainerFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a ContainerFeatureCollection instance from which to copy Feature instance references. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType is unsupported by this factory. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied Feature instances do not adhere to the collection's membership restrictions, if any.

`createFromSequence`—creates a ContainerFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a sequence of Feature instances. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is unsupported by this factory. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied Feature instances do not adhere to the collection's membership restrictions, if any.

`createFromFeatureData`—creates a ContainerFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a sequence of features described by value structures (`FeatureData`). Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is unsupported by this factory. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied feature values do not adhere to the collection's membership restrictions, if any.

3.1.5 Queryable Interfaces

Implementations wishing to expose their Features to OQS querying may do so in two ways. Firstly, they may directly evaluate SQL or OQL queries sent to them by exposing a FeatureCollection supporting the OQS `QueryEvaluator` interface (i.e. exposing a `QueryableFeatureCollection`). RDBMS- & ODBMS-based servers that have internal querying functionality will typically take this course.

Alternatively, the implementation may expose its collection of Features in the form of an OQS QueryableCollection in which each element is a CORBA Object supporting the `Feature` interface. This would allow a commercial implementation of the OQS to evaluate queries against the feature collection from outside the native implementation of the FeatureCollection. This approach would typically be used by implementations without a native querying capacity.

The Collection interface of the CORBA Object Query Service (OQS) is intended for use in both scoping queries and packaging responses to them. An OpenGIS FeatureCollection may be used to scope a query but it cannot be used to return results of a query. Queries returning a collection of Feature references will return them as a QueryResultSet and not an OpenGIS FeatureCollection.

The premises of the query largely determine the return type of queries. For example:

```
SELECT * FROM Roads WHERE name = "Route 66"
```

will return a QueryResultSet of records (the record contains the desired set of Road Feature properties).

Clients will be able to access the members of these return collections in many ways – as singleton property values or as a record. Property values may be of any CORBA type. Convenience methods are provided to retrieve a property as a well-known value type in the situation where the client "knows" what the property type is. Otherwise, the client may retrieve the data as a Value structure that encodes type information. Alternatively, one can add direct support value retrieval based on CORBA any or the DynAny object that allows access to objects of type any without static knowledge of the structure. (The DynAny object is part of the current OMG effort to improve ORB Portability. For more information see [**2**]);

3.1.5.1 Query Example

```
 OGIS::QueryableContainerFeatureCollectionRef countries;

// Create the geometric constraints for the query
OGIS::GeomConstraintSeq geom_constraints(0);
geom_constraints.length(0);

// the where clause to execute
char* whereClause = "continent = 'Europe'";
```

```
// Figure out the supported query languages, verify support for SQL92
// Verification is left out from code for brevity
 OGIS::QueryEvaluator::QLType qlType = OGIS::QueryEvaluator::QLType::SQL_92Query;

OGIS::QueryableFeatureCollection results;

// Execute the query
try {
  results = countries->query(whereClause, qlType, geom_constraints);
} catch (CORBA::Exception& e) {
  cout << "Error evaluating the query: " << ODF::exception_to_string(e) << endl;
  exit(1);
}

OGIS::FeatureIterator iterator;
String country_name;

// Get an iterator over the result collection
try {
  iterator = results->create_iterator();
} catch (CORBA::Exception& e) {
  cout << "Error retrieving an iterator: " << ODF::exception_to_string(e) << endl;
  exit(1);
}

    // Iterate over the result set
try {
      while ((more = iterator->advance()) == B_TRUE) {
         try {
              country_name = get_property_by_name("name");
         } catch (FeatureIterator::InvalidConversion) {
              cout << "error converting the property to a string !"
                   << ODF::exception_to_string(exc);
         } catch (FeatureIterator::InvalidProperty) {
              cout << "invalid property name!"
                   << ODF::exception_to_string(exc);
         }
      }
} catch( CORBA::Exception& exc) {
      cout << "error advancing the iterator!"<<ODF::exception_to_string(exc);
}

try {
     iterator->destroy();
     results->destroy();
} catch (CORBA::Exception exc) {
     cout << "error destroying!"<<ODF::exception_to_string(exc);
}
```

3.1.5.2 QueryEvaluator Interface

3.1.5.2.1 Purpose

This interface provides support for expressing and evaluating a query statement. The `QueryEvaluator` interface provides support to select a query language (from a supported set), define a 'where'-clause, define geometric query constraints and to pose a query to a queryable feature collection. The `QueryEvaluator` is responsible for the synchronous execution of the query and for producing a result set (if appropriate). By default, the `QueryEvaluator` executes queries as atomic operations – no transactional semantics are supported in this version.

The `QueryEvaluator` provides a generic framework for expressing queries against a collection of Features. It is neutral with respect to the query language employed to express the query. The principal query language dialects are SQL Query and OQL. The QLType enumeration provides a set of well-known query languages along with their derivatives.

The results of a query executed by a `QueryEvaluator` are returned as a `QueryableFeature Collection`. This allows the results of a query to be further refined by another query, and to be manipulated in all of the ways in which other feature collections can be manipulated.

SQLQuery represents a generic term denoting the evolution of the SQL standard process (i.e., SQL89, SQL92, SQL9x). It is envisioned that SQL92Query, the current standard, will evolve into the SQL9x.

OQL represents a generic term denoting the evolution of the OQL standards process (i.e., OQL-93). It is envisioned that OQL-93, the current standard, will evolve into OQL-9x.

OQL-93Basic represents the marriage of SQL92Query and OQL-93Query. For more information, the reader is referred to the CORBA Query Service Specification [**5**].

3.1.5.2.2 IDL Specification

```
interface QueryableFeatureCollection;     // forward declaration

interface QueryEvaluator {
     exception     QueryLanguageTypeNotSupported {};
     exception     InvalidQuery {string why;};
     exception     QueryProcessingError {string why;};
     exception     InvalidGeometry {string why;};
     exception     WKBNotImplemented {};
     exception     InvalidSpatialOperator {};

     enum QLType {
          SQLQuery, SQL_92Query, OQL, OQLBasic, OQL_93, OQL_93Basic
     };

     typedef sequence<QLType> QLTypeSeq;

     enum SpatialOperatorType {
          TouchOp, ContainsOp, WithinOp, DisjointOp, CrossesOp, OverlapsOp, IntersectsOp
     };

     readonly attribute QLTypeSeq ql_types;
     readonly attribute QLType default_ql_type;

     enum GeomSwitch { GeomType, WKSGeomType };
     union QueryGeom
          switch ( GeomSwitch ) {
               case GeomType:     Geometry      geom;

               case WKSGeomType:  WKSGeometry   wks_geom;
          };

     struct GeomConstraint {
          Istring             geom_name;
          SpatialOperatorType spatial_op;
          QueryGeom           geo;
     };

     typedef sequence<GeomConstraint> GeomConstraintSeq;

     QueryableFeatureCollection query(
                                 in string where_clause, in QLType ql_type,
                                 in GeomConstraintSeq geom_constraints )
                                 raises(QueryLanguageTypeNotSupported, InvalidQuery,
                                      InvalidGeometry, QueryProcessingError,
                                      InvalidSpatialOperator);
};
```

3.1.5.2.3 Interface Description

query—this method enables a subset of a collection of features to be found which satisfy both geometric and non-geometric constraints. The where_clause contains a syntactically correct "where clause" expressed in the query language indicated by the query language type (ql_type) parameter. The geom_constraints contains a list of geometric constraints which are logicalled ANDed with each other and with the where_clause to return a collection of features which may subsequently have further queries made on them. The geometric constraints are expressed in the form "geom_name spatial_op some_geometry", allowing different geometric properties to be selected for interaction with different types of geometry (either live object geometry or a well known structure). If a where_clause is not specified, it is assumed that a spatial search query is being requested. If geom_constraints is not supplied, it is assumed that a non-spatial query is to be evaluated. If both arguments are supplied, then a spatial and non-spatial attribute query is evaluated. Only Features satisfying both search criteria are returned. If neither geom_constraints nor where_clause are specified, all features are returned.

3.1.5.3 QueryableFeatureCollection Interfaces

3.1.5.3.1 Purpose

The QueryableFeatureCollection interface provides support for expressing and evaluating queries against a collection's contents. The query languages supported, as well as the iteration over the query result set, are inherited from the QueryEvaluator interface. It is an implementation detail as to how a QueryableFeatureCollection object evaluates a query: directly, or via delegation to its constituent members.

The QueryableFeatureCollectionFactory interface provides support for the creation of QueryableFeatureCollection instances. The interface essentially duplicates the FeatureCollectionFactory interface, except for returning a specialized QueryableFeatureCollection instance.

3.1.5.3.2 IDL Specification

```
interface QueryableFeatureCollection : FeatureCollection, QueryEvaluator{
};

interface QueryableFeatureCollectionFactory {

     exception     FeatureTypeInvalid {string why;};
     exception     PropertyInvalid {string why;};
     exception     FeatureInvalid {string why;};

     QueryableFeatureCollection create(in FeatureType collection_type,
                       in NVPairSeq collection_properties)
                       raises (FeatureTypeInvalid, PropertyInvalid);

     QueryableFeatureCollection createFromCollection(in FeatureType collection_type,
                       in NVPairSeq collection_properties,
                       in FeatureCollection collection)
                       raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

     QueryableFeatureCollection createFromSequence(in FeatureType collection_type,
                       in NVPairSeq collection_properties,
                       in FeatureSeq list)
                       raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

     QueryableFeatureCollection createFromFeatureData(in FeatureType collection_type,
                       in NVPairSeq collection_properties, in FeatureDataSeq list)
                       raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);
```

```
};
```

3.1.5.3.3 Interface Description

`create`—create a QueryableFeatureCollection instance given the FeatureType of the collection and a set of collection properties. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied.

`createFromCollection`—creates a QueryableFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a QueryableFeatureCollection instance from which to copy Feature instance references. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied Feature instances do not adhere to the collection's membership restrictions, if any.

`createFromSequence`—creates a QueryableFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a sequence of Feature instances. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied Feature instances do not adhere to the collection's membership restrictions, if any.

`createFromFeatureData`—creates a QueryableFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a sequence of features described by value structures (`FeatureData`). Raises a `FeatureTypeInvalid` exception if the supplied `FeatureType` instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied feature values do not adhere to the collection's membership restrictions, if any.

3.1.5.4 QueryableContainerFeatureCollection Interfaces

3.1.5.4.1 Purpose

The `QueryableContainerFeatureCollection` interface provides support for expressing and evaluating queries against a container collection's contents. The query languages supported, as well as the iteration over the query result set, are inherited from the `QueryEvaluator` interface. It is an implementation detail as to how a QueryableContainerFeatureCollection object evaluates a query: directly, or via delegation to its constituent members.

The `QueryableContainerFeatureCollectionFactory` interface provides support for the creation of QueryableContainerFeatureCollection instances. The interface essentially duplicates the `ContainerFeatureCollectionFactory` interface, except for returning a specialized QueryableContainerFeatureCollection instance.

3.1.5.4.2 IDL Specification

```
interface QueryableContainerFeatureCollection: ContainerFeatureCollection, QueryEvaluator
{
};

interface QueryableContainerFeatureCollectionFactory {
     exception     FeatureTypeInvalid {string why;};
     exception     PropertyInvalid {string why;};
     exception     FeatureInvalid {string why;};
```

```
    QueryableContainerFeatureCollection create(in FeatureType collection_type,
                      in NVPairSeq collection_properties)
                      raises (FeatureTypeInvalid, PropertyInvalid);

    QueryableContainerFeatureCollection createFromCollection(
                      in FeatureType collection_type,
                      in NVPairSeq collection_properties,
                      in FeatureCollection collection)
                      raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

    QueryableContainerFeatureCollection createFromSequence(
                      in FeatureType collection_type,
                      in NVPairSeq collection_properties,
                      in FeatureSeq list)
                      raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

    QueryableContainerFeatureCollection createFromFeatureData
                      (in FeatureType collection_type,
                      in NVPairSeq collection_properties,
                      in FeatureDataSeq list)
                      raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

};
```

3.1.5.4.3 Interface Description

`create`—creates a QueryableContainerFeatureCollection instance given the FeatureType of the collection and a set of collection properties. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied.

`createFromCollection`—creates a QueryableContainerFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a ContainerFeatureCollection instance from which to copy Feature instance references. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied Feature instances do not adhere to the collection's membership restrictions, if any.

`createFromSequence`—creates a QueryableContainerFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a sequence of Feature instances. Raises a `FeatureTypeInvalid` exception if the supplied FeatureType instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied Feature instances do not adhere to the collection's membership restrictions, if any.

`createFromFeatureData`—creates a QueryableContainerFeatureCollection instance given the FeatureType of the collection, a set of collection properties, and a sequence of features described by value structures (`FeatureData`). Raises a `FeatureTypeInvalid` exception if the supplied `FeatureType` instance is invalid. A `PropertyInvalid` exception is raised if the required collection parameters are not supplied. A `FeatureInvalid` exception is raised if the supplied feature values do not adhere to the collection's membership restrictions, if any.

3.2 Geometry Module

3.2.1 Spatial Reference System Interfaces

3.2.1.1 SpatialReferenceInfo Interface

3.2.1.1.1 Purpose

The `SpatialReferenceInfo` interface exposes a number of attributes common to all entities with the EPSG/POSC system. Most of the SRS interfaces inherit directly or indirectly from it.

3.2.1.1.2 IDL Specification

```
interface SpatialReferenceInfo {

     attribute string       name;
     attribute string       authority;
     attribute long         code;
     attribute string       alias;
     attribute string       abbreviation;
     attribute string       remarks;

     readonly attribute string    well_known_text;   // UGH!!!!!

};
```

3.2.1.1.3 Interface Description

`name`—is the EPSG assigned name of the Spatial Reference System component. This name is unique.

`authority`—is the organization, body or person who created the entity. For EPSG supplied reference data the `authority` is "EPSG".

`code`—is an EPSG assigned unique code (integer) of the entity. EPSG reserves the integer range 0 to 32767. Non EPSG standard entities will use codes greater than 32767.

`alias`—is the EPSG assigned alias of the Spatial Reference System component.

`abbreviation`—is an abbreviation of the Spatial Reference System component.

`remarks`—is a natural language description of the entity.

`well_known_text`—is a comma delimited textual representation of the parameters on the Spatial Reference System component.

3.2.1.2 Unit Interface

3.2.1.2.1 Purpose

The `Unit` interface abstracts various coordinate units used by spatial reference systems.

3.2.1.2.2 IDL Specification

```
interface Unit : SpatialReferenceInfo {
};
```

3.2.1.3 AngularUnit Interface

3.2.1.3.1 Purpose

The `AngularUnit` interface exposes the definition of the units used by the SRS entity to define angles. The angular units are defined with respect to radians. Name and identity information is inherited from the `SpatialReferenceInfo` interface.

3.2.1.3.2 IDL Specification:

```
interface AngularUnit : Unit {

     attribute double   radians_per_unit;

};
```

3.2.1.3.3 Interface Description:

`radians_per_unit`—defines the number of radians per unit.

3.2.1.4 LinearUnit Interface

3.2.1.4.1 Purpose

The `LinearUnit` interface exposes the definition of the linear units used by an SRS entity, allowing the use of various standard and non-standard linear units. Name and identity information is inherited from the `SpatialReferenceInfo` interface.

3.2.1.4.2 IDL Specification

```
interface LinearUnit : Unit {

     attribute double   metres_per_unit;

};
```

3.2.1.4.3 Interface Description

`metres_per_unit`—defines the value of a unit in metres.

3.2.1.5 Ellipsoid Interface

3.2.1.5.1 Purpose

Most Spatial Reference Systems use a mathematical abstraction of the earth's shape on which to base a coordinate system. Typically this surface is an ellipsoid of revolution. The ellipsoid interface exposes the defining parameters of an ellipsoid. Identity parameters are inherited from the `SpatialReferenceInfo` interface.

3.2.1.5.2 IDL Specification

```
interface Ellipsoid : SpatialReferenceInfo {

     attribute double       semi_major_axis;
     attribute double       semi_minor_axis;
```

```
    attribute double       inverse_flattening;
    attribute LinearUnit   axis_unit;

};
```

3.2.1.5.3 Interface Description:

`semi_major_axis`—is the semi-major axis of the ellipsoid in `axis_units`.

`semi_minor_axis`—is the semi-minor axis of the ellipsoid in `axis_units`.

`inverse_flattening`—is the inverse flattening the ellipsoid (`inverse_flattening = semi_major_axis /(semi_major_axis - semi_minor_axis))`.

`axis_unit`—indicates the units used to define the axes of the ellipsoid.

3.2.1.6 HorizontalDatum Interface

3.2.1.6.1 Purpose

In the EPSG standards, a horizontal datum provides a mapping between the surface of the earth and the surface of a base ellipsoid. This datum is exposed through the `HorizontalDatum` interface.

3.2.1.6.2 IDL Specification

```
interface HorizontalDatum : SpatialReferenceInfo {

    attribute Ellipsoid    base_ellipsoid;

};
```

3.2.1.6.3 Interface Description

`base_ellipsoid`—is the ellipsoid on which the HorizontalDatum is based.

3.2.1.7 PrimeMeridian Interface

3.2.1.7.1 Purpose

The `PrimeMeridian` interface provides access to the definition of a system's prime meridian. This allows the selection of an arbitrary standard for the SRS. The prime meridian is defined with respect to the Greenwich Prime Meridian. Identity and Descriptive attributes are inherited from the `SpatialReferenceInfo` interface.

3.2.1.7.2 IDL Specification

```
interface PrimeMeridian : SpatialReferenceInfo {

    attribute double       longitude;
    attribute AngularUnit  angular_units;

};
```

3.2.1.7.3 Interface Description

longitude—is the longitude of the Prime Meridian relative to Greenwich in angular_units.

angular_units—specifies the angular units in which longitude is described.

3.2.1.8 SpatialReferenceSystem Interface

3.2.1.8.1 Purpose

The SpatialReferenceSystem interface is an abstraction of all Spatial Reference Systems. These may include non-geodetic systems and local SRSs. Identity and descriptive attributes are inherited from SpatialReferenceInfo; type-specific attributes are specified in the various interfaces derived from SpatialReferenceSystem.

3.2.1.8.2 IDL Specification

```
interface SpatialReferenceSystem : SpatialReferenceInfo {
};
```

3.2.1.9 GeodeticSpatialReferenceSystem Interface

3.2.1.9.1 Purpose

The GeodeticSpatialReferenceSystem interface is an abstraction of all geodetic Spatial Reference Systems including Geographic SRSs and Projected SRSs. Identity and descriptive attributes are inherited from SpatialReferenceInfo (through SpatialReferenceSystem); type-specific attributes are specified in the various interfaces derived from GeodeticSpatialReferenceSystem.

3.2.1.9.2 IDL Specification

```
interface GeodeticSpatialReferenceSystem : SpatialReferenceSystem {
};
```

3.2.1.10 GeographicCoordinateSystem Interface

3.2.1.10.1 Purpose

A geographic coordinate system uses spherical or ellipsoidal coordinates (latitudes & longitudes). The GeographicCoordinateSystem interface exposes the defining parameters of a geographic coordinate system. It inherits from GeodeticSpatialReferenceSystem.

3.2.1.10.2 IDL Specification

```
interface GeographicCoordinateSystem : GeodeticSpatialReference {

     attribute string            usage;            // description?
     attribute HorizontalDatum   horizontal_datum;
     attribute AngularUnit       angular_unit;
     attribute PrimeMeridian     prime_meridian;

};
```

3.2.1.10.3 Interface Description

usage—is a comment on the usage of the coordinate system.

horizontal_datum—is the horizontal datum on which the GeographicCoordinateSystem is based.

angular_unit—is the angular units used by the coordinates within the GeographicCoordinateSystem.

prime_meridian—is the reference meridian of the GeographicCoordinateSystem.

3.2.1.11 Parameter Interface

3.2.1.11.1 Purpose

Various components of spatial reference systems are defined using parameters. As the number and type of these parameters may vary with different components, a Parameter interface is used to expose this information to clients.

3.2.1.11.2 IDL Specification

```
interface Parameter : SpatialReferenceInfo {

     attribute Unit         units;
     attribute double       value;

};
```

3.2.1.11.3 Interface Description

units—exposes the units in which value is expressed. This attribute provides the semantics of value.

value—the parameter value in units.

3.2.1.12 ParameterList Interface

3.2.1.12.1 Purpose

The ParameterList interface exposes the set of parameters that are used to define a particular component of a spatial reference system.

3.2.1.12.2 IDL Specification

```
typedef sequence<Parameter> ParameterSeq;

interface ParameterList {

     readonly attribute long     number_parameters;

     ParameterSeq  get_default_parameters();

     void          set_parameters (in ParameterSeq parameters);
     ParameterSeq  get_parameters ();

};
```

3.2.1.12.3 Interface Description

number_parameters—is the number of parameters in the ParameterList.

`get_default_parameters`—returns a sequence of the default parameter values for the spatial reference system component .

`set_parameters`—sets the values of the `ParameterList`.

`get_parameters`—gets the current values of the `ParameterList`.

3.2.1.13 GeographicTransform Interface

3.2.1.13.1 Purpose

The `GeographicTransform` interface provides access to transformation facilities capable of transforming coordinate geometries in the form of Well-known Structures from one coordinate system into another.

3.2.1.13.2 IDL Specification

```
interface GeographicTransform : SpatialReferenceInfo {

     attribute     GeographicCoordinateSystem  source_gcs;
     attribute     GeographicCoordinateSystem  target_gcs;

     WKSGeometry   forward (in WKSGeometry source_geometry);
     WKSGeometry   inverse (in WKSGeometry source_geometry);

};
```

3.2.1.13.3 Interface Description

`source_gcs`—is the source coordinate system of the transformation.

`target_gcs`—is the destination coordinate system of the transformation.

`forward`—transforms the `source_geometry` from the `source_gcs` coordinate system to the `target_gcs` coordinate system and returns the result.

`inverse`—transforms the `source_geometry` from the `target_gcs` coordinate system to the `source_gcs` coordinate system and returns the result.

3.2.1.14 Projection Interface

3.2.1.14.1 Purpose

A projection maps between an ellipsoid and a coordinate plane. The `Projection` interface provides access to the parameters that define this mapping.

3.2.1.14.2 IDL Specification

```
interface Projection : SpatialReferenceInfo {

     readonly attribute string    usage;
     readonly attribute string    classification;

     WKSGeometry   forward (in WKSGeometry source_geometry);
     WKSGeometry   inverse (in WKSGeometry source_geometry);
```

```
    readonly attribute ParameterList    parameters;

    attribute AngularUnit       angular_units;
    attribute LinearUnit        linear_units;
    attribute Ellipsoid         base_ellipsoid;

};
```

3.2.1.14.3 Interface Description

`usage`—is a comment on the usage of the coordinate system.

`classification`—indicates the classification of the projection.

`forward`—transforms the Well-known Structure `source_geometry` from geographic coordinates into projected (planar) coordinates and returns the result.

`inverse`—transforms the Well-known Structure `source_geometry` from projected (planar) coordinates into geographic coordinates and returns the result.

`parameters`—is the set of parameters that defines the projection. These parameters vary between collection classes.

`angular_units`—are the angular units used for geographic coordinates.

`linear_units`—are the linear units used for planar coordinates.

`base_ellipsoid`—is the base ellipsoid of the `Projection`.

3.2.1.15 ProjectedCoordinateSystem Interface

3.2.1.15.1 Purpose

A projected coordinate system uses a geographic coordinate system and a projection to map from points on the earth's surface to those on a coordinate plane. The `ProjectedCoordinateSystem` interface exposes the defining parameters of a geographic coordinate system. It inherits from `GeodeticSpatialReferenceSystem`.

3.2.1.15.2 IDL Specification

```
interface ProjectedCoordinateSystem : GeodeticSpatialReferenceSystem {

    attribute string        usage;
    attribute GeographicCoordinateSystem    geographic_coordinate_system;
    attribute LinearUnit    linear_units;
    attribute Projection    base_projection;

    readonly attribute ParameterList    parameters;

    WKSGeometry     forward (in WKSGeometry source_geometry);
    WKSGeometry     inverse (in WKSGeometry source_geometry);

};
```

3.2.1.15.3 Interface Description

`usage`—is a comment on the usage of the coordinate system.

`geographic_coordinate_system`—is the underlying geographic coordinate system.

`linear_units`—are the linear units of the coordinate plane.

`base_projection`—is the projection used to project from the ellipsoid to the coordinate plane.

`parameters`—is the set of parameters that defines the projected coordinate system.

`forward`—transforms the Well-known Structure `source_geometry` from geographic coordinates into projected (planar) coordinates and returns the result.

`inverse`—transforms the Well-known Structure `source_geometry` from projected (planar) coordinates into geographic coordinates and returns the result.

3.2.1.16 SpatialReferenceSystemFactory Interface

3.2.1.16.1 Purpose:

The SpatialReferenceSystemFactory interface provides for the creation of a spatial reference system given a comma delimited EPSG textual definition (see section 3.2.6.3).

3.2.1.16.2 IDL Specification:

```
interface SpatialReferenceSystemFactory {

     SpatialReferenceSystem  create_from_WKT (in string srs_wkt);

};
```

3.2.1.16.3 Interface Description:

`create_from_WKT`—creates a new SpatialReferenceSystem object from the text description `srs_wkt`.

3.2.1.17 SpatialReferenceComponentFactory Interface

3.2.1.17.1 Purpose

The SpatialReferenceComponentFactory interface provides for the creation of spatial reference system components from a standard identification code.

3.2.1.17.2 IDL Specification

```
interface SpatialReferenceComponentFactory {

     readonly attribute  string  authority;

     ProjectedCoordinateSystem   create_projected_coordinate_system (in long code);
     GeographicCoordinateSystem  create_geographic_coordinate_system (in long code);
     Projection                  create_projection (in long code);
     GeographicTransform         create_geographic_transform (in long code);
     HorizontalDatum             create_horizontal_datum (in long code);
     Ellipsoid                   create_ellipsoid (in long code);
     PrimeMeridian               create_prime_meridian (in long code);
     LinearUnit                  create_linear_unit (in long code);
     AngularUnit                 create_angular_unit (in long code);

};
```

3.2.1.17.3 Interface Description

`authority`—is the standards authority which defines the components created by the factory (e.g. 'EPSG').

`create_projected_coordinate_system`—creates the ProjectedCoordinateSystem with the identifier `code`.

`create_geographic_coordinate_system`—creates the GeographicCoordinateSystem with the identifier `code`.

`create_projection`—creates the Projection with the identifier `code`.

`create_geographic_transform`—creates the GeographicTransform with the identifier `code`.

`create_horizontal_datum`—creates the HorizontalDatum with the identifier `code`.

`create_ellipsoid`—creates the Ellipsoid with the identifier `code`.

`create_prime_meridian`—creates the PrimeMeridian with the identifier `code`.

`create_linear_unit`—creates the LinearUnit with the identifier `code`.

`create_angular_unit`—creates the AngularUnit with the identifier `code`.

3.2.2 General Geometry Interfaces

3.2.2.1 Geometry Interface

3.2.2.1.1 Purpose

Any valid value of a geometric attribute of feature may be exposed through the `Geometry` interface. This interface provides access to the properties that are common to all geometric entities.

3.2.2.1.2 IDL Specification

```
interface Geometry {

     exception WKBNotImplemented {};

     enum EgenhoferElement {
               Empty, NotEmpty, NoTest
     };

     typedef EgenhoferElement EgenhoferOperator[3][3];

     readonly attribute     short    dimension;    // dimension of the geometry
                                                   // - not the coordinate system
     readonly attribute     Envelope  range_envelope;// minBoundBox in abstract spec

     readonly attribute     SpatialReferenceSystem      spatial_reference_system;

     // geometric characteristics
     boolean       is_empty();
     boolean       is_simple();
     boolean       is_closed();
```

```
    // constructive operators
    Geometry      copy();
    Geometry      boundary();
    Geometry      buffer (in double distance);
    Geometry      convex_hull();

    // WKS operators
    WKSGeometry   export();                        // export geometry to WKS
    OctetSeq      export_WKBGeometry()             // export geometry to WKB
                    raises (WKBNotImplemented);

        // relational operators
    boolean       equals (in Geometry other);
    boolean       touches (in Geometry other);
    boolean       contains (in Geometry other);
    boolean       within (in Geometry other);
    boolean       disjoint (in Geometry other);
    boolean       crosses (in Geometry other);
    boolean       overlaps (in Geometry other);
    boolean       intersects (in Geometry other);
    boolean       relate (in Geometry other, in EgenhoferOperator operator);

    // metric operators
    double        distance (in Geometry other);

    // set operators
    Geometry      intersection (in Geometry other);
    Geometry      union_op(in Geometry other);
    Geometry      difference (in Geometry other);
    Geometry      symmetric_difference (in Geometry other);

    void          destroy();

};
```

3.2.2.1.3 Interface Description

`dimension`—returns the dimension of the geometry. For a point this value will be 0, a line string it will be 1, for a polygon 2. Note: this value is not the dimension of the coordinate space in which the geometry is defined, nor the dimension of the extent of the geometry.

`range_envelope`—returns an envelope wholly containing the geometry.

`spatial_reference_system`—returns a reference to the Geometry's Spatial Reference System. The SRS provides the semantics of the coordinates exposed by the Geometry through well-known structures (i.e. it locates the Geometry with respect to the earth).

`is_empty`—returns true if the Geometry is the empty set.

`is_simple`—returns true if the Geometry has no anomalous geometric points, such as self intersection or self tangency.

`is_closed`—returns true if the Geometry's boundary is the empty set.

`copy`—creates a (deep) copy of the Geometry and returns a reference to it. The Geometry's associated Spatial Reference System is not copied (only a reference to it is copied).

`boundary`—returns the closure of the combinatorial boundary of the Geometry as described in the Abstract Specification [3.12.3.2].

`buffer`—returns a Geometry representing all points within `distance` of `this`.

convex_hull—returns a Geometry representing the convex hull of this.

export—returns the coordinate geometry in the form of a Well-known Structure (WKS) of the Geometry.

export_WKBGeometry—returns the coordinate geometry in the form of a Well-known Binary (WKB) representation of the Geometry (see section 3.2.6.2). This operation is optional: OpenGIS compliance does not demand its implementation. A WKBNotImplemented exception is raised if this operation is not implemented.

equals—returns true if this and other are equivalent geometries.

touches—returns true if this and other only share part of their boundaries.

contains—returns true if other is wholly contained within this.

within—returns true if this is wholly contained by other.

disjoint—returns true if this and other are disjoint geometries.

crosses—returns true if this crosses other.

overlaps—returns true if this overlaps other.

intersects—returns true if this intersects other.

relate—returns true if the Egenhofer relationship specified by operator exists between this and other. The Egenhofer operator is specified through the EgenhoferOperator array. This is a 3 x 3 array of trinary elements that correspond to the nine sets of intersection between two geometries. Each element may have the value Empty, NotEmpty and NoTest. NoTest elements always return true. Empty elements will return true only if the corresponding intersection set is empty. NotEmpty elements will only return true if the corresponding intersection set is not empty. The relate operation will only return true, if all nine elements return true. For more information on Egenhofer operators see [4].

distance—returns the minimum distance between this and other.

intersection—returns the point set intersection of this and other.

union_op—returns the point set union of this and other.

difference—returns the point set difference of this and other.

symmetric_difference—returns the point set symmetric difference of this and other.

destroy—destroys the Geometry.

3.2.2.2 GeometryFactory Interface

3.2.2.2.1 Purpose

The GeometryFactory interface allows client applications to create new Geometry objects which may then be assigned as values to geometric attributes of Features.

3.2.2.2.2 IDL Specification

```
interface GeometryFactory {

     exception     InvalidWKS {string why;};
     exception     InvalidWKB {string why;};
     exception     WKBNotImplemented {};

     Geometry      create(in Geometry existing);

     Geometry      create_from_WKS(in SpatialReferenceSystem srs,in WKSGeometry geo)
                        raises (InvalidWKS);

     Geometry      create_from_WKB(in SpatialReferenceSystem srs,in OctetSeq geo)
                        raises (InvalidWKB, WKBNotImplemented);
};
```

3.2.2.2.3 Interface Description

`create`—creates a new Geometry instance that is a deep-copy of `existing`.

`create_from_WKS`—creates a new Geometry instance given an spatial referencing system and a coordinate geometry encoded in the well-known structure (WKS). If the input parameter is not a valid WKS, an `InvalidWKS` exception will be thrown.

`create_from_WKB`—creates a new Geometry instance given an spatial referencing system and a coordinate geometry encoded in the well-known binary stream (WKB) representation format (see section 3.2.6.2). If the input parameter is not a valid WKB, an `InvalidWKB` exception will be thrown. This operation is optional: OpenGIS compliance does not demand its implementation. A `WKBNotImplemented` exception is raised if this operation is not implemented.

3.2.2.3 GeometryCollection Interface

3.2.2.3.1 Purpose:

In many cases a single simple geometry is insufficient to represent a geometric entity. In others, it is desirable to group arbitrary geometries, and treat them as a single geometry for various purposes. The `GeometryCollection` interface allows the grouping of possibly arbitrary Geometries for subsequent simultaneous manipulation. A `GeometryIterator` interface allows client access to the components of such groups or composite geometries, while hiding the structure of the underlying collection implementation. Particular implementations of GeometryCollections may restrict membership to Geometries with particular characteristics (e.g. all elements must be co-planar, of a particular dimension or may not overlap).

3.2.2.3.2 IDL Specification

```
interface GeometryIterator;
interface GeometryCollection;

interface GeometryCollection : Geometry {

     exception IteratorInvalid {};
     exception PositionInvalid {};
     exception GeometryInvalid {};

     readonly attribute     long     number_elements;

     // these operations allowing for arbitrary collections
     void      add_element (in Geometry element) raises (GeometryInvalid);
     void      merge (in GeometryCollection elements) raises (GeometryInvalid);
```

```
    void     insert_element_at (in Geometry element, in GeometryIterator where)
                 raises (GeometryInvalid, IteratorInvalid);
    void     replace_element_at (in Geometry element, in GeometryIterator where)
                 raises (GeometryInvalid, IteratorInvalid, PositionInvalid);

    void     remove_element_at (in GeometryIterator where)
                 raises (IteratorInvalid, PositionInvalid);
    void     remove_all_elements ();

    // retrieve a geometry from a collection
    Geometry retrieve_element_at (in GeometryIterator where)
                 raises (IteratorInvalid, PositionInvalid);

    // create an iterator over the collection
    GeometryIterator    create_iterator();
};

interface GeometryIterator {
     exception IteratorInvalid {};
     exception PositionInvalid {};

     Geometry next ()   raises (IteratorInvalid, PositionInvalid);
     void     reset()   raises (IteratorInvalid);
     boolean  more();
     void     destroy();
};
```

3.2.2.3.3 Interface Description

`number_elements`—returns the number of geometries in the GeometryCollection.

`add_element`—adds the Geometry `element` to the GeometryCollection. If `element` does not adhere to all membership restrictions a `GeometryInvalid` exception is raised.

`merge`—merges the GeometryCollection `elements` into the GeometryCollection. If a component of elements does not adhere to the membership restrictions of the GeometryCollection a `GeometryInvalid` exception is raised.

`insert_element_at`—inserts the Geometry `element` at the position in the Collection indicated by the GeometryIterator `where`. A `GeometryInvalid` exception is raised if `element` does not conform to the membership requirements of the GeometryCollection. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the GeometryCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Geometry.

`replace_element_at`—replaces the Geometry `element` at the position in the GeometryCollection indicated by the GeometryIterator `where`. A `GeometryInvalid` exception is raised if `element` does not conform to the membership requirements of the GeometryCollection. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the GeometryCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Geometry.

`remove_element_at`—removes the Geometry at the position in the GeometryCollection indicated by the GeometryIterator `where`. An `IteratorInvalid` exception is raised if `where` is not a valid iterator of the GeometryCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Geometry.

`remove_all_elements`—empties the GeometryCollection.

`retrieve_element_at`—returns the Geometry at the position in the GeometryCollection indicated by the GeometryIterator `where`. An `IteratorInvalid` exception is raised if `where` is not a valid

iterator of the GeometryCollection. A `PositionInvalid` exception is raised if the iterator is not pointing at a Geometry.

`create_iterator`—returns an iterator of the GeometryCollection.

3.2.2.3.4 GeometryIterator Interface Description

The `GeometryIterator` provides clients with access into the components of a GeometryCollection, and permits them to specify a location within a GeometryCollection without being exposed to the implementation of the GeometryCollection.

`next`—returns the next Geometry in the GeometryCollection. The order Geometries are returned is implementation specific.

`reset`—resets the GeometryIterator to the beginning of the GeometryCollection.

`more`—returns true if there are more geometries in the GeometryCollection.

`destroy`—destroys the GeometryIterator.

3.2.3 Zero Dimensional Geometries

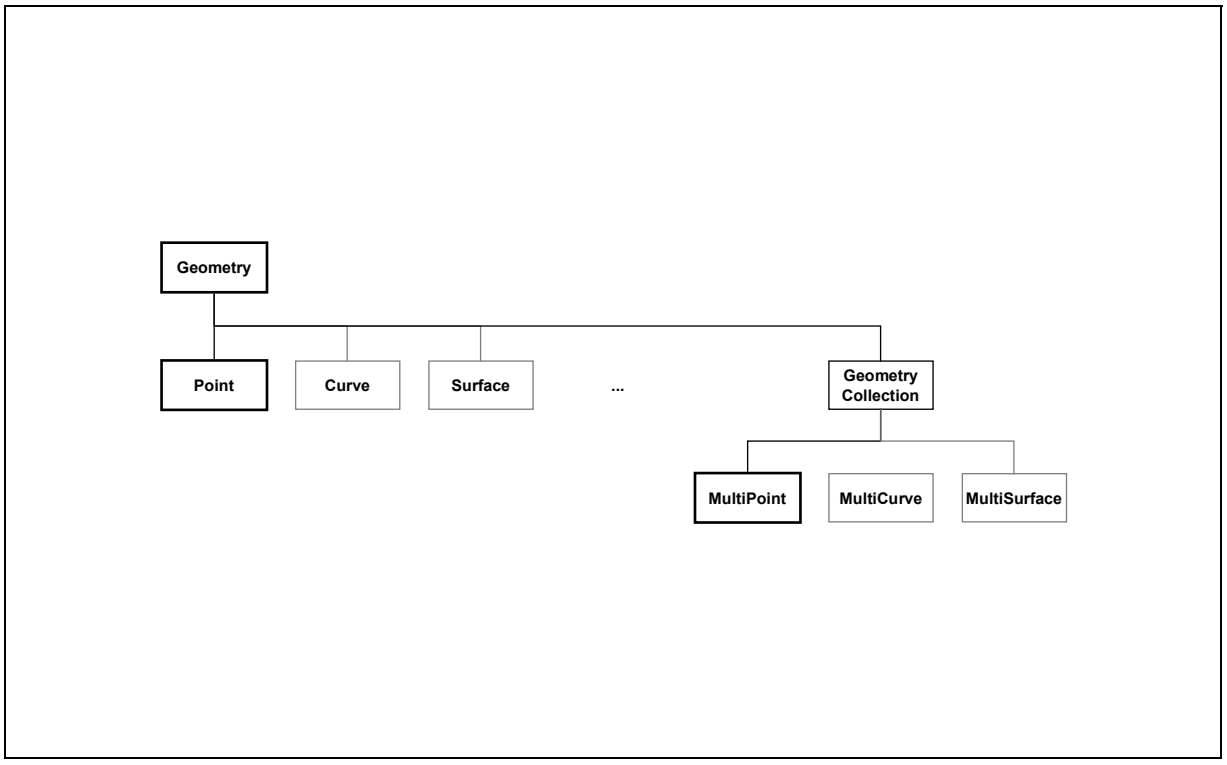

Figure 3.1—Zero dimensional Geometry Interfaces. Bold boxes indicate interfaces in this specification, all others are suggested future extensions.

3.2.3.1 Point Interface

3.2.3.1.1 Purpose

The Point interface exposes zero-dimensional geometries. It inherits from Geometry. The boundary of a Point is the empty set. All Points are simple and closed.

3.2.3.1.2 IDL Specification

```
interface Point : Geometry {
     attribute     WKSPoint  coordinates;

};
```

3.2.3.1.3 Interface Description

coordinates—this attribute may be used to retrieve and to set the coordinates of a Point geometry using the WKSPoint well-known structure.

3.2.3.2 PointFactory Interface

3.2.3.2.1 Purpose

This interface provides support for creating new Point objects. It enforces the geometric policies that a Point geometry must adhere to, given as input a Well-known Structure, a Well-known Binary representation, etc.

3.2.3.2.2 IDL Specification

```
interface PointFactory : GeometryFactory {
     exception InvalidWKSPoint {};
     exception InvalidWKBPoint {};

     Point         create_from_Point(in Point existing);

     Point         create_from_WKSPoint(in SpatialReferenceSystem srs, in WKSPoint geo)
                        raises (InvalidWKSPoint);

     Point         create_from_WKBPoint(in SpatialReferenceSystem srs, in OctetSeq geo)
                        raises (InvalidWKBPoint, WKBNotImplemented);
};
```

3.2.3.2.3 Interface Description

Inherited GeometryFactory interface— The PointFactory enforces that the arguments supplied to the inherited GeometryFactory interface methods are valid Point geometric constructs, be it a Point object, a WKSPoint or a WKBPoint. The object instance created is a Point object, but is widened to a Geometry object.

create_from_Point—creates an new Point instance that is a deep-copy of existing.

create_from_WKSPoint—creates a new Point instance given an spatial referencing system and a coordinate geometry encoded in the well-known structure (WKS). If the input parameter is not a valid WKSPoint, an InvalidWKSPoint exception will be thrown.

create_from_WKBPoint—creates a new Point instance given an spatial referencing system and a coordinate geometry encoded in the well-known binary (WKB) representation format (see section 3.2.6.2). If the input parameter is not a valid WKBPoint, an InvalidWKBPoint exception will be thrown. This operation is optional: OpenGIS compliance does not demand its implementation. A WKBNotImplemented exception is raised if this operation is not implemented.

3.2.3.3 MultiPoint Interface

3.2.3.3.1 Purpose

The MultiPoint interface exposes a collection of zero-dimensional geometries (points). It inherits from GeometryCollection. The `boundary` of a MultiPoint is the empty set. A MultiPoint is simple if no two components are coincident. All MultiPoints are closed.

3.2.3.3.2 IDL Specification

```
interface MultiPoint : GeometryCollection {

};
```

3.2.3.4 MultiPointFactory Interface

3.2.3.4.1 Purpose

This interface provides support for creating new MultiPoint objects. It enforces the geometric policies that a MultiPoint geometry must adhere to, given as input a Well-known Structure, a Well-known Binary representation, etc.

3.2.3.4.2 IDL Specification

```
interface MultiPointFactory : GeometryFactory {
     exception InvalidWKSMultiPoint {};
     exception InvalidWKBMultiPoint {};

     MultiPoint    create_from_MultiPoint(in MultiPoint existing);

     MultiPoint    create_from_WKSMultiPoint(in SpatialReferenceSystem srs,
                                             in WKSPointSeq geo)
                                        raises (InvalidWKSMultiPoint);

     MultiPoint    create_from_WKBMultiPoint(in SpatialReferenceSystem srs,
                                             in OctetSeq geo)
                                        raises (InvalidWKBMultiPoint,WKBNotImplemented);
};
```

3.2.3.4.3 Interface Description

Inherited `GeometryFactory` interface— The MultiPointFactory enforces that the arguments supplied to the inherited `GeometryFactory` interface methods are valid `MultiPoint` geometric constructs, be it a MultiPoint object, a `WKSPoint` or a `WKBPoint`. The object instance created is a MultiPoint object, but is widened to a Geometry object.

`create_from_MultiPoint`—creates an new MultiPoint instance that is a deep-copy of `existing`.

`create_from_WKSMultiPoint`—creates a new MultiPoint instance given an spatial referencing system and a coordinate geometry encoded in the well-known structure (WKS). If the input parameter is not a valid `WKSMultiPoint`, an `InvalidWKSMultiPoint` exception will be thrown.

`create_from_WKBMultiPoint`—creates a new MultiPoint instance given an spatial referencing system and a coordinate geometry encoded in the well-known binary (WKB) representation format (see section 3.2.6.2). If the input parameter is not a valid `WKBMultiPoint`, an `InvalidWKBMultiPoint` exception will be thrown. This operation is optional: OpenGIS

compliance does not demand its implementation. A `WKBNotImplemented` exception is raised if this operation is not implemented.

3.2.4 One-dimensional Geometries

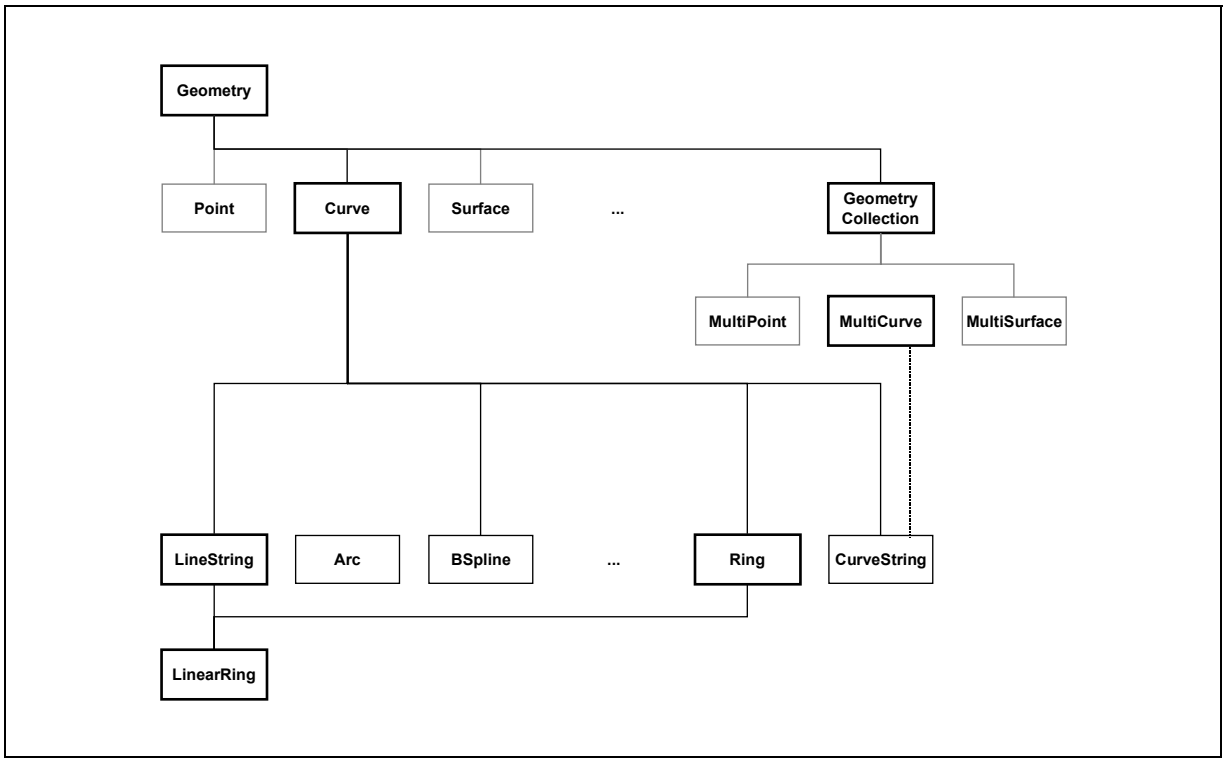

Figure 3.2—One-dimensional Geometry Interfaces. Bold boxes indicate interfaces in this specification, all others are suggested future extensions.

3.2.4.1 Curve Interface

3.2.4.1.1 Purpose

The Curve interface is supported by all one-dimensional continuous geometries. It inherits from Geometry. A Curve has a start point and end point. It is simple if it does not pass through the same point twice (except possibly at the start and end points). It is closed if the start point and the end point are the same. A Curve's `boundary` is a MultiPoint consisting of the start point and the end point for an unclosed Curve and the empty set for a closed Curve.

3.2.4.1.2 IDL Specification

```
exception OutOfDomain {};

interface Curve : Geometry {
     exception WKSNotImplemented {};

     readonly attribute     double    length;
     readonly attribute     Point     start_point;
     readonly attribute     Point     end_point;
     readonly attribute     WKSPoint  start_point_as_WKS;
     readonly attribute     WKSPoint  end_point_as_WKS;

     boolean   is_planar();
```

```
    Point    value (in double r) raises (OutOfDomain);
    WKSPoint value_as_WKS (in double r) raises (OutOfDomain);

};
```

3.2.4.1.3 Interface Description

`length`—returns the length of the curve (in coordinate units).

`start_point`—returns the starting Point of the curve.

`end_point`—returns the end Point of the curve.

`start_point_as_WKS`—returns the starting point of the curve as a `WKSPoint` Well-known Structure.

`end_point_as_WKS`—returns the end point of the curve as a `WKSPoint` Well-known Structure.

`is_planar`—returns true if the entire curve lies within a plane.

`value`—returns the point r coordinate units along the curve from the start point. If r is less than zero or greater than length a `OutOfDomain` exception is raised. Note that since value(0.0) = start_point and value(length) = end_point, the start_point and end_point attributes are technically redundant. They have been retained for ease of use.

`value_as_WKS`—returns the coordinates of the point r coordinate units along the curve from the start point in the form of a WKSPoint Well-known Structure. If r is less than zero or greater than length a `OutOfDomain` exception is raised.

3.2.4.1.4 Usage Scenarios

The curve interface will be supported by all one-dimensional geometries including, but not limited to, lines, line-strings, arcs (circular, elliptical, parabolic, etc.), quadratics and other mathematical curves and b-splines. Closed curves, including linear rings, circles and ellipses will also support the curve interface.

3.2.4.2 LineString Interface

3.2.4.2.1 Purpose:

The `LineString` interface exposes linear geometries: curves defined by a series of points with linear interpolation between points. It inherits from `Curve`.

3.2.4.2.2 IDL Specification

```
interface LineString : Curve  {

     exception     InvalidIndex{};
     exception     MinimumPoints{};

     readonly attribute     long     num_points;

     Point    get_point_by_index (in long index) raises (InvalidIndex);
     WKSPoint get_point_by_index_as_WKS (in long index) raises (InvalidIndex);

     void     set_point_by_index (in WKSPoint new_point, in long index)
                   raises (InvalidIndex);
     void     set_point_by_index_with_WKS (in WKSPoint new_point, in long index)
```

```
                    raises (InvalidIndex);

     void      insert_point_by_index (in Point new_point, in long index)
                    raises (InvalidIndex);
     void      insert_point_by_index_with_WKS (in WKSPoint new_point, in long index)
                    raises (InvalidIndex);

     void      append_point (in Point new_point);
     void      append_point_with_WKS (in WKSPoint new_point);

     void      delete_point_by_index (in long index)
                    raises (InvalidIndex, MinimumPoints);

};
```

3.2.4.2.3 Interface Description:

`num_points`—returns the number of points in the line string.

`get_point_by_index`—returns the `index`th point of the line string. An `InvalidIndex` exception is raised if `index` is less than zero or greater than `num_points`.

`get_point_by_index_as_WKS`—returns the `index`th point of the line string as a `WKSPoint` Well-known Structure. An `InvalidIndex` exception is raised if `index` is less than zero or greater than `num_points`.

`set_point_by_index`—sets the `index`th point of the line string to `new_point`. An `InvalidIndex` exception is raised if `index` is less than zero or greater than `num_points`.

`set_point_by_index_with_WKS`—sets the `index`th point of the line string to the point defined by the `WKSPoint` Well-known Structure `new_point`. An `InvalidIndex` exception is raised if `index` is less than zero or greater than `num_points`.

`insert_point_by_index`—inserts `new_point` into the linestring before the `index`th point. An `InvalidIndex` exception is raised if `index` is less than zero or greater than `num_points`.

`insert_point_by_index_with_WKS`—inserts the point defined by the WKSPoint Well-known Structure `new_point` into the linestring before the `index`th point. An `InvalidIndex` exception is raised if `index` is less than zero or greater than `num_points`.

`append_point`—appends `new_point` to the end of the line string.

`append_point_with_WKS`—appends the point defined by the WKSPoint Well-known Structure `new_point` to the end of linestring.

`delete_point_by_index`—deletes the `index`th point from the line string. A line string will typically not be permitted to have less than two points. An attempt to delete a point from a two-point line string will raise a `MinimumPoints` exception. An `InvalidIndex` exception is raised if `index` is less than zero or greater than `num_points`.

3.2.4.3 LineStringFactory Interface

3.2.4.3.1 Purpose

This interface provides support for creating new LineString objects. It enforces the geometric policies that a LineString geometry must adhere to, given as input a Well-known Structure, a Well-known Binary representation, etc.

3.2.4.3.2 IDL Specification

```
interface LineStringFactory : GeometryFactory {

     exception InvalidWKSLineString {};
     exception InvalidWKBLineString {};

     LineString         create_from_LineString(in LineString existing);

     LineString         create_from_WKSLineString (in SpatialReferenceSystem srs,
                                                   in WKSLineString geo)
                                              raises (InvalidWKSLineString);

     LineString         create_from_WKBLineString(in SpatialReferenceSystem srs,
                                                  in OctetSeq geo)
                                              raises
(InvalidWKBLineString,WKBNotImplemented);
};
```

3.2.4.3.3 Interface Description

Inherited `GeometryFactory` interface— The LineStringFactory enforces that the arguments supplied to the inherited `GeometryFactory` interface methods are valid `LineString` geometric constructs, be it a LineString object, a `WKSLineString` or a `WKBLineString`. The object instance created is a LineString object, but is widened to a Geometry object.

`create_from_LineString`—creates an new LineString instance that is a deep-copy of `existing`.

`create_from_WKSLineString`—creates a new LineString instance given an spatial referencing system and a coordinate geometry encoded in the well-known structure (WKS). If the input parameter is not a valid `WKSLineString`, an `InvalidWKSLineString` exception will be thrown.

`create_from_WKBLineString`—creates a new LineString instance given an spatial referencing system and a coordinate geometry encoded in the well-known binary (WKB) representation format (see section 3.2.6.2). If the input parameter is not a valid `WKBLineString`, an `InvalidWKBLineString` exception will be thrown. This operation is optional: OpenGIS compliance does not demand its implementation. A `WKBNotImplemented` exception is raised if this operation is not implemented.

3.2.4.4 Ring Interface

3.2.4.4.1 Purpose

Rings are planar, simple, closed curves i.e. they are non-self-intersecting, curves where `start_point` and `end_point` are coincident. The `Ring` interface simply provides the ‘building blocks’ for polygons, it does not add any functionality to curve.

3.2.4.4.2 IDL Specification

```
interface Ring : Curve  {
```

```
};
```

3.2.4.5 LinearRing Interface

3.2.4.5.1 Purpose

The LinearRing interface exposes closed, linear geometries: rings defined by a series of points with linear interpolation between points. It inherits from `Ring` and `LineString`.

3.2.4.5.2 IDL Specification

```
interface LinearRing : Ring, LineString  {
};
```

3.2.4.6 MultiCurve Interface

3.2.4.6.1 Purpose

The `MultiCurve` interface exposes a collection of one-dimensional geometries (curves). It inherits from `GeometryCollection`. The `boundary` of a MultiCurve is the modulo 2 union of all start and end points [1 para. 3.12.3.2]. A `MultiCurve` is simple if all its components are simple and the interiors of no two components intersect.

3.2.4.6.2 IDL Specification

```
interface MultiCurve : GeometryCollection  {

     readonly  attribute     double    length;
};
```

3.2.4.6.3 Interface Description

`length`—returns the sum of the lengths of all constituent curves (in coordinate units).

3.2.4.7 MultiLineString Interface

3.2.4.7.1 Purpose

The `MultiLineString` interface exposes a collection of `LineString`s. It inherits from `MultiCurve`.

3.2.4.7.2 IDL Specification

```
interface MultiLineString : MultiCurve {

};
```

3.2.4.8 MultiLineStringFactory Interface

3.2.4.8.1 Purpose

This interface provides support for creating new MultiLineString objects. It enforces the geometric policies that a MultiLineString geometry must adhere to, given as input a Well-known Structure, a Well-known Binary representation, etc.

3.2.4.8.2 IDL Specification

```
interface MultiLineStringFactory : GeometryFactory {

     exception          InvalidWKSMultiLineString {};
     exception          InvalidWKBMultiLineString {};

     MultiLineString    create_from_MultiLineString(in MultiLineString existing);

     MultiLineString    create_from_WKSMultiLineString(in SpatialReferenceSystem srs,
                                        in WKSLineStringSeq geo)
                                   raises (InvalidWKSMultiLineString);

     MultiLineString    create_from_WKBMultiLineString(in SpatialReferenceSystem srs,
                                        in OctetSeq geo)
                                   raises
(InvalidWKBMultiLineString,WKBNotImplemented);
};
```

3.2.4.8.3 Interface Description

Inherited `GeometryFactory` interface— The MultiLineString enforces that the arguments supplied to the inherited `GeometryFactory` interface methods are valid `MultiLineString` geometric constructs, be it a MultiLineString object, a `WKSMultiLineString` or a `WKBMultiLineString`. The object instance created is a MultiLineString object, but is widened to a Geometry object.

`create_from_MultiLineString`—creates an new MultiLineString instance that is a deep-copy of `existing`.

`create_from_WKSMultiLineString`—creates a new MultiLineString instance given an spatial referencing system and a coordinate geometry encoded in the well-known structure (WKS). If the input parameter is not a valid `WKSMultiLineString`, an `InvalidWKSMultiLineString` exception will be thrown.

`create_from_WKBMultiLineString`—creates a new MultiLineString instance given an spatial referencing system and a coordinate geometry encoded in the well-known binary (WKB) representation format (see section 3.2.6.2). If the input parameter is not a valid `WKBMultiLineString`, an `InvalidWKBMultiLineString` exception will be thrown. This operation is optional: OpenGIS compliance does not demand its implementation. A `WKBNotImplemented` exception is raised if this operation is not implemented.

3.2.5 Two-dimensional Geometries

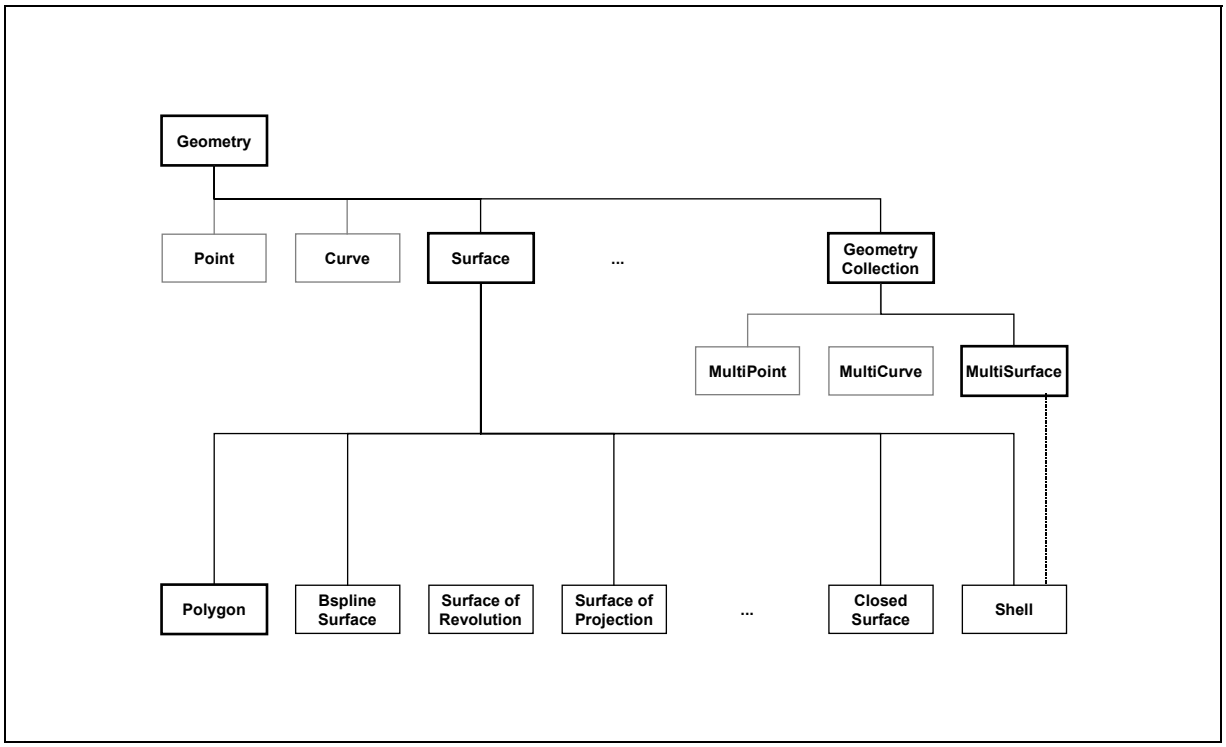

Figure 3.3—Two-dimensional Geometry Interfaces. Bold boxes indicate interfaces in this specification, all others are suggested future extensions.

3.2.5.1 Surface Interface

3.2.5.1.1 Purpose

The `Surface` interface is supported by all two-dimensional continuous geometries. It inherits from geometry. The boundary of a Surface is the collection of curves constituting its exterior boundary and any interior boundaries.

3.2.5.1.2 IDL Specification

```
interface Surface : Geometry {

     readonly attribute  double       area;
     readonly attribute  WKSPoint     centroid;
     readonly attribute  WKSPoint     centroid_as_WKS;
     readonly attribute  WKSPoint     point_on_surface;
     readonly attribute  WKSPoint     point_on_surface_as_WKS;

     boolean  is_planar();
};
```

3.2.5.1.3 Interface Description:

`area`—returns the length of the curve (in coordinate units).

`centroid`—returns the centroid of the Surface (assuming all points on the Surface are equally weighted).

centroid_as_WKS—returns the centroid of the Surface (assuming all points on the Surface are equally weighted) as a Well-known Structure.

point_on_surface—returns a Point on the Surface (i.e. a point within the external boundary and outside all interior boundaries).

point_on_surface_as_WKS—returns a point on the Surface (i.e. a point within the external boundary and outside all interior boundaries) as a Well-known Structure.

is_planar—returns true if the entire Surface lies within a plane.

3.2.5.2 Polygon Interface

3.2.5.2.1 Purpose

The Polygon interface exposes planar surfaces defined by one exterior ring and a series of internal rings. It inherits from surface.

3.2.5.2.2 IDL Specification

```
interface Polygon : Surface {

     readonly attribute     Ring          exterior_ring;
     readonly attribute     WKSGeometry   exterior_ring_as_WKS;

     readonly attribute     MultiCurve    interior_rings;
     readonly attribute     WKSGeometry   interior_rings_as_WKS;
};
```

3.2.5.2.3 Interface Description

exterior_ring—returns the ring defining the external boundary of the polygon.

exterior_ring_as_WKS—returns the ring defining the external boundary of the polygon as a Well-known Structure.

interior_rings— returns the collection of rings defining the interior rings (holes) within the polygon.

interior_rings_as_WKS— returns the collection of rings defining the interior rings (holes) within the polygon as a Well-known Structure.

3.2.5.3 LinearPolygon Interface

3.2.5.3.1 Purpose

The LinearPolygon interface inherits from Polygon and restricts the polygon geometry to be linear.

3.2.5.3.2 IDL Specification

```
interface LinearPolygon : Polygon {
};
```

3.2.5.4 LinearPolygonFactory Interface

3.2.5.4.1 Purpose

This interface provides support for creating new LinearPolygon objects. It enforces the geometric policies that a LinearPolygon geometry must adhere to, given as input a Well-known Structure, a Well-known Binary representation, etc.

3.2.5.4.2 IDL Specification

```
interface LinearPolygonFactory : GeometryFactory {

     exception InvalidWKSLinearPolygon {};
     exception InvalidWKBLinearPolygon {};

     LinearPolygon      create_from_LinearPolygon(in LinearPolygon existing);

     LinearPolygon      create_from_WKSLinearPolygon(in SpatialReferenceSystem srs,
                                                     in WKSLinearPolygon geo)
                                             raises (InvalidWKSLinearPolygon);

     LinearPolygon      create_from_WKBLinearPolygon(in SpatialReferenceSystem srs,
                                                     in OctetSeq geo)
               raises (InvalidWKBLinearPolygon,WKBNotImplemented);
};
```

3.2.5.4.3 Interface Description

Inherited `GeometryFactory` interface— The LinearPolygon enforces that the arguments supplied to the inherited `GeometryFactory` interface methods are valid `LinearPolygon` geometric constructs, be it a LinearPolygon object, a `WKSLinearPolygon` or a `WKBLinearPolygon`. The object instance created is a LinearPolygon object, but is widened to a Geometry object.

`create_from_LinearPolygon`—creates an new LinearPolygon instance that is a deep-copy of `existing`.

`create_from_WKSLinearPolygon`—creates a new LinearPolygon instance given an spatial referencing system and a coordinate geometry encoded in the well-known structure (WKS). If the input parameter is not a valid `WKSLinearPolygon`, an `InvalidWKSLinearPolygon` exception will be thrown.

`create_from_WKBLinearPolygon`—creates a new LinearPolygon instance given an spatial referencing system and a coordinate geometry encoded in the well-known binary (WKB) representation format (see section 3.2.6.2). If the input parameter is not a valid `WKBLinearPolygon`, an `InvalidWKBLinearPolygon` exception will be thrown. This operation is optional: OpenGIS compliance does not demand its implementation. A `WKBNotImplemented` exception is raised if this operation is not implemented.

3.2.5.5 MultiSurface Interface

3.2.5.5.1 Purpose

The `MultiSurface` interface exposes a collection of two-dimensional geometries (surfaces). It inherits from `GeometryCollection`.

3.2.5.5.2 IDL Specification

```
interface MultiSurface : GeometryCollection  {

     readonly  attribute     double    area;

};
```

3.2.5.5.3 Interface Description:

`area`—returns the sum of the areas of all constituent surfaces.

3.2.5.6 MultiPolygon Interface

3.2.5.6.1 Purpose

The `MultiPolygon` interface inherits from `MultiSurface.`

3.2.5.6.2 IDL Specification

```
interface MultiPolygon : MultiSurface {
};
```

3.2.5.7 MultiLinearPolygon Interface

3.2.5.7.1 Purpose

The `MultiLinearPolygon` interface inherits from `MultiPolygon` and restricts the polygon geometries to be linear.

3.2.5.7.2 IDL Specification

```
interface MultiLinearPolygon : MultiPolygon {
};
```

3.2.5.8 MultiLinearPolygonFactory Interface

3.2.5.8.1 Purpose

This interface provides support for creating new MultiLinearPolygon objects. It enforces the geometric policies that a MultiLinearPolygon geometry must adhere to, given as input a Well-known Structure, a Well-known Binary representation, etc.

3.2.5.8.2 IDL Specification

```
interface MultiLinearPolygonFactory : GeometryFactory {

     exception InvalidWKSMultiLinearPolygon {};
     exception InvalidWKBMultiLinearPolygon {};

     MultiLinearPolygon     create_from_MultiLinearPolygon(in MultiLinearPolygon
existing);

     MultiLinearPolygon     create_from_WKSMultiLinearPolygon(in SpatialReferenceSystem
srs,
                                                    in WKSLinearPolygonSeq geo)
                                               raises (InvalidWKSMultiLinearPolygon);
```

```
    MultiLinearPolygon     create_from_WKBMultiLinearPolygon(in SpatialReferenceSystem
srs,
                                                    in OctetSeq geo)
                                               raises (InvalidWKBMultiLinearPolygon);
};
```

3.2.5.8.3 Interface Description

Inherited `GeometryFactory` interface— The MultiLinearPolygon enforces that the arguments supplied to the inherited `GeometryFactory` interface methods are valid `MultiLinearPolygon` geometric constructs, be it a MultiLinearPolygon object, a `WKSMultiLinearPolygon` or a `WKBMultiLinearPolygon`. The object instance created is a MultiLinearPolygon object, but is widened to a Geometry object.

`create_from_MultiLinearPolygon`—creates an new MultiLinearPolygon instance that is a deep-copy of `existing`.

`create_from_WKSMultiLinearPolygon`–creates a new MultiLinearPolygon instance given an spatial referencing system and a coordinate geometry encoded in the well-known structure (WKS). If the input parameter is not a valid `WKSMultiLinearPolygon`, an `InvalidWKSMultiLinearPolygon` exception will be thrown.

`create_from_WKBMultiLinearPolygon`–creates a new MultiLinearPolygon instance given an spatial referencing system and a coordinate geometry encoded in the well-known binary (WKB) representation format (see section 3.2.6.2). If the input parameter is not a valid `WKBMultiLinearPolygon`, an `InvalidWKBMultiLinearPolygon` exception will be thrown. This operation is optional: OpenGIS compliance does not demand its implementation. A `WKBNotImplemented` exception is raised if this operation is not implemented.

3.2.6 Structures & Enumerations

3.2.6.1 Well-known Structures

3.2.6.1.1 Purpose:

The Well-known Structures (WKS) allow for the sharing of linear coordinate geometries and collections of such geometries between interoperating applications. The specification of these structures in IDL allows for the unambiguous representation of all such geometries independent of platform as required by FR #6 of RFP1. These structures are the geometric analogue of the basic arithmetic types (short, long, float, double).

3.2.6.1.2 IDL Specification

```
struct WKSPoint {
    double    x;
    double    y;
};

typedef  sequence<WKSPoint>     WKSPointSeq;
typedef  sequence<WKSPoint>     WKSLineString;
typedef  sequence<WKSLineString>  WKSLineStringSeq;
typedef  sequence<WKSPoint>     WKSLinearRing;
typedef  sequence<WKSLinearRing>     WKSLinearRingSeq;

struct WKSLinearPolygon {
    WKSLinearRing      externalBoundary;
    WKSLinearRingSeq   internalBoundaries;
};
```

```
typedef sequence <WKSLinearPolygon> WKSLinearPolyonSeq;

enum WKSType {
     WKSPointType,WKSMultiPointType, WKSLineStringType,WKSMultiLineStringType,
      WKSLinearRingType, WKSLinearPolygonType, WKSMultiLinearPolygonType,
     WKSCollectionType
};

union WKSGeometry       // near-equivalent to the 'CoordinateGeometry of the spec'
     switch (WKSType) {

          case WKSPointType:
               WKSPoint      point;

          case WKSMultiPointType:
               WKSPointSeq          multi_point;

          case WKSLineStringType:
               WKSLineString line_string;

          case WKSMultiLineStringType:
               WKSLineStringSeq     multi_line_string;

          case WKSLinearRingType:
               WKSLinearRing linear_ring;

          case WKSLinearPolygonType:
               WKSLinearPolygon   linear_polygon;

          case WKSMultiLinearPolygonType:
               WKSLinearPolygonSeq        multi_linear_polygon;

          case WKSCollectionType:
               sequence<WKSGeometry>   collection;
     };

struct Envelope {
     WKSPoint      minm;
     WKSPoint      maxm;
};
```

3.2.6.2 The Well-known Binary Representation for Geometry (WKBGeometry)

3.2.6.2.1 Component Overview

The Well-known Binary Representation for Geometry (`WKBGeometry`) provides a portable representation of a Geometry value as a contiguous stream of bytes. It permits Geometry values to be exchanged between an ODBC client and an SQL database in binary form.

3.2.6.2.2 Component Description

The Well-known Binary Representation for Geometry is obtained by serializing a geometry instance as a sequence of numeric types drawn from the set {`Unsigned Integer`, `Double`} and then serializing each numeric type as a sequence of bytes using one of two well defined, standard, binary representations for numeric types (NDR, XDR). The specific binary encoding (NDR or XDR) used for a geometry byte stream is described by a one byte tag that precedes the serialized bytes. The only difference between the two encodings of geometry is one of byte order, the XDR encoding is Big Endian, the NDR encoding is Little Endian.

3.2.6.2.2.1 Numeric Type Definitions

An `Unsigned Integer` is a 32-bit (4-byte) data type that encodes a nonnegative integer in the range [0, 4294967295].

A `Double` is a 64-bit (8-byte) double precision data type that encodes a double precision number using the IEEE 754 double precision format

The above definitions are common to both XDR and NDR.

3.2.6.2.2.2 XDR (Big Endian) Encoding of Numeric Types

The XDR representation of an `Unsigned Integer` is Big Endian (most significant byte first).

The XDR representation of a `Double` is Big Endian (sign bit is first byte).

3.2.6.2.2.3 NDR (Little Endian) Encoding of Numeric Types

The NDR representation of an `Unsigned Integer` is Little Endian (least significant byte first).

The NDR representation of a `Double` is Little Endian (sign bit is last byte).

3.2.6.2.2.4 Conversion between the NDR and XDR representations of WKBGeometry

Conversion between the NDR and XDR data types for `Unsigned Integer` and `Double` numbers is a simple operation involving reversing the order of bytes within each `Unsigned Integer` or `Double` number in the byte stream.

3.2.6.2.2.5 Relationship to other COM and CORBA data transfer protocols

The XDR representation for `Unsigned Integer` and `Double` numbers described above is also the standard representation for `Unsigned Integer` and for `Double` number in the CORBA Standard Stream Format for Externalized Object Data that is described as part of the CORBA Externalization Service Specification [15].

The NDR representation for `Unsigned Integer` and `Double` number described above is also the standard representation for `Unsigned Integer` and for `Double` number in the DCOM protocols that is based on DCE RPC and NDR [16].

3.2.6.2.2.6 Description of WKBGeometry Byte Streams

The Well-known Binary Representation for Geometry is described below. The basic building block is the byte stream for a `Point`, which consists of two `Double` numbers. The byte streams for other geometries are built using the byte streams for geometries that have already been defined.

```
// Basic Type definitions

// byte : 1 byte

// uint32 : 32 bit unsigned integer (4 bytes)

// double : double precision number (8 bytes)


// Building Blocks : Point, LinearRing


Point {
```

```
    double x;
    double y;
};

LinearRing  {
    uint32  numPoints;
    Point   points[numPoints];
}

enum wkbGeometryType {
    wkbPoint = 1,
    wkbLineString = 2,
    wkbPolygon = 3,
    wkbMultiPoint = 4,
    wkbMultiLineString = 5,
    wkbMultiPolygon = 6,
    wkbGeometryCollection = 7
};

enum wkbByteOrder {
    wkbXDR = 0,                          // Big Endian
    wkbNDR = 1                           // Little Endian
};

WKBPoint {
    byte                  byteOrder;
    uint32                wkbType;
    // 1
    Point                 point;
}

WKBLineString {
    byte                  byteOrder;
    uint32                wkbType;
    // 2
    uint32                numPoints;
    Point                 points[numPoints];
}

WKBPolygon {
    byte                  byteOrder;
    uint32                wkbType;
    // 3
    uint32                numRings;
```

```
    LinearRing              rings[numRings];
}


WKBMultiPoint {
    byte                    byteOrder;
    uint32                  wkbType;                                        // 4
    uint32                  num wkbPoints;
    WKBPoint                WKBPoints[num_wkbPoints];
}


WKBMultiLineString {
    byte                    byteOrder;
    uint32                  wkbType;
    // 5
    uint32                  num_wkbLineStrings;
    WKBLineString  WKBLineStrings[num wkbLineStrings];
}


wkbMultiPolygon {
    byte                    byteOrder;
    uint32                  wkbType;
    // 6
    uint32                  num wkbPolygons;
    WKBPolygon              wkbPolygons[num wkbPolygons];
}


WKBGeometry {
    union {
        WKBPoint                                point;
        WKBLineString                           linestring;
        WKBPolygon                              polygon;
        WKBGeometryCollection             collection;
        WKBMultiPoint                           mpoint;
        WKBMultiLineString                mlinestring;
        WKBMultiPolygon                   mpolygon;
    }
};


WKBGeometryCollection {
    byte                    byte order;
    uint32                  wkbType;
    // 7
    uint32                  num_wkbGeometries;
    WKBGeometry             wkbGeometries[num_wkbGeometries];
```

```
}
```

Figure 3.2 shows a pictorial representation of the Well-known Byte Stream for a `Polygon` with one outer ring and one inner ring.

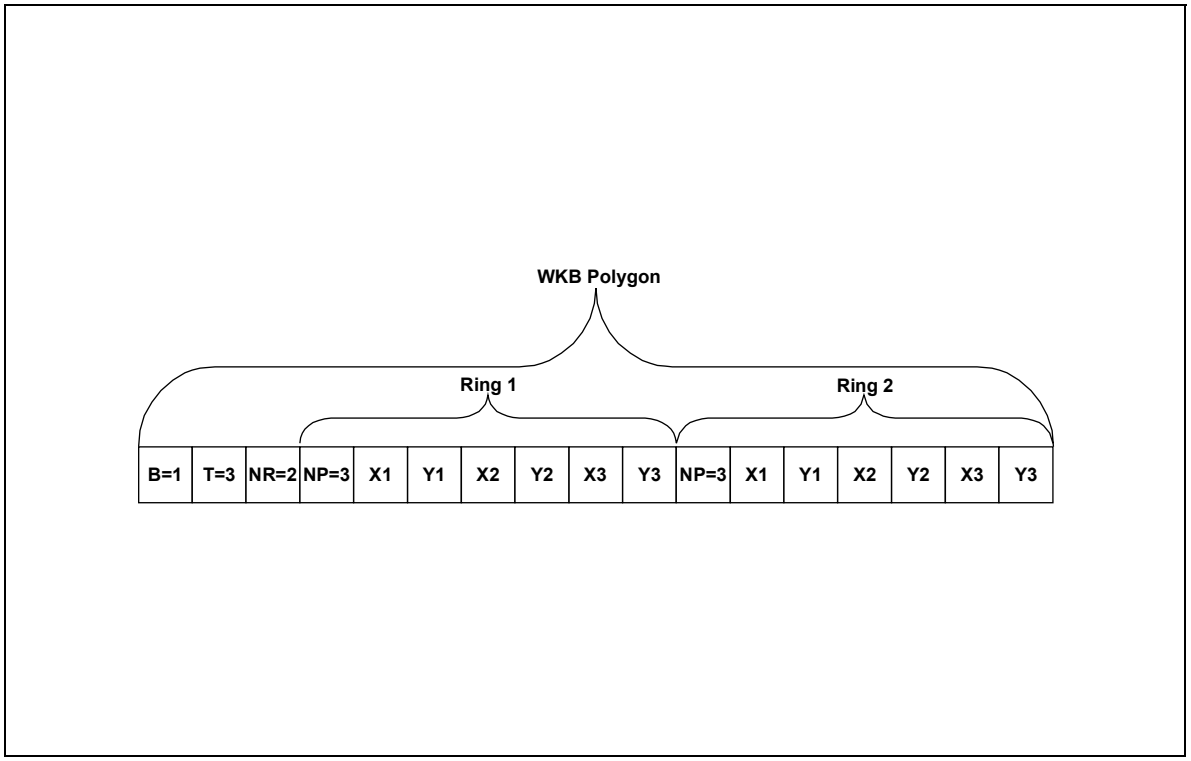

Figure 3.2—Well-known Binary Representation for a `Geometry` value in NDR format (B=1) of type Polygon (T=3) with 2 linear rings (NR = 2) each ring having 3 points (NP = 3).

3.2.6.2.2.7 Assertions for Well-known Binary Representation for Geometry

The Well-known Binary Representation for Geometry is designed to represent instances of the geometry types described in the Geometry Object Model and in the OpenGIS Abstract Specification. **Any `WKBGeometry` instance must satisfy the assertions for the type of `Geometry` that it describes**. These assertions may be found in the section 2.2 of the OpenGIS Simple Features for OLE/COM Specification.

These assertions imply the following for Rings, Polygons and MultiPolygons:

3.2.6.2.2.8 Linear Rings

Rings are simple and closed, which means that Linear Rings may **not** self-touch.

3.2.6.2.2.9 Polygons

No two Linear Rings in the boundary of a Polygon may cross each other, the Linear Rings in the boundary of a polygon may intersect at most at a single point but only as a tangent.

3.2.6.2.2.10 MultiPolygons

1. The interiors of 2 Polygons that are elements of a MultiPolygon may not intersect.

2. The Boundaries of any 2 Polygons that are elements of a MultiPolygon may touch at only a *finite* number of points.

For more details on the above assertions and for the assertions for each geometry type the reader is referred to the Geometry Object Model section of the OLE/COM specification.

3.2.6.3 Well-known Text Representation of Spatial Reference Systems

3.2.6.3.1 Component Overview

The Well-known Text Representation of Spatial Reference Systems provides a standard textual representation for spatial reference system information.

3.2.6.3.2 Component Description

The definitions of the well-known text representation are modeled after the POSC/EPSG coordinate system data model.

A spatial reference system, also referred to as a coordinate system, is a geographic (latitude-longitude), a projected (X,Y), or a geocentric (X,Y,Z) coordinate system.

The coordinate system is composed of several objects. Each object has a keyword in upper case (for example, `DATUM` or `UNIT`) followed by the defining, comma-delimited, parameters of the object in brackets. Some objects are composed of objects so the result is a nested structure. Implementations are free to substitute standard brackets ( ) for square brackets [ ] and should be prepared to read both forms of brackets.

The EBNF (Extended Backus Naur Form) definition for the string representation of a coordinate system is as follows, using square brackets, see note above:

```
<coordinate system> = <projected cs> | <geographic cs> | <geocentric cs>

<projected cs> = PROJCS["<name>", <geographic cs>, <projection>, {<parameter>,}* <linear
unit>]

<projection> = PROJECTION["<name>"]

<parameter> = PARAMETER["<name>", <value>]

<value> = <number>
```

A data set's coordinate system is identified by the `PROJCS` keyword if the data are in projected coordinates, by `GEOGCS` if in geographic coordinates, or by `GEOCCS` if in geocentric coordinates.

The `PROJCS` keyword is followed by all of the "pieces" which define the projected coordinate system. The first piece of any object is always the name. Several objects follow the projected coordinate system name: the geographic coordinate system, the map projection, 1 or more parameters, and the linear unit of measure. All projected coordinate systems are based upon a geographic coordinate system so we will describe the pieces specific to a projected coordinate system first. As an example, UTM zone 10N on the NAD83 datum is defined as:

```
    PROJCS["NAD 1983 UTM Zone 10N",
        <geographic cs>,
        PROJECTION["Transverse Mercator"],
        PARAMETER["False_Easting",500000.0],
        PARAMETER["False_Northing",0.0],
```

```
        PARAMETER["Central_Meridian",-123.0],
        PARAMETER["Scale Factor",0.9996],
        PARAMETER["Latitude of Origin",0.0],
        UNIT["Meter",1.0]]
```

The name and several objects define the geographic coordinate system object in turn: the datum, the prime meridian, and the angular unit of measure.

```
<geographic cs> = GEOGCS["<name>", <datum>, <prime meridian>, <angular unit>]

<datum> = DATUM["<name>", <spheroid>]

<spheroid> = SPHEROID["<name>", <semi-major axis>, <inverse flattening>]

<semi-major axis> = <number>        NOTE: semi-major axis is measured in meters and must be > 0.

<inverse flattening> = <number>

<prime meridian> = PRIMEM["<name>", <longitude>]

<longitude> = <number>
```

The geographic coordinate system string for UTM zone 10 on NAD83 is

```
    GEOGCS["GCS North American 1983",
        DATUM["D_North_American_1983",
        SPHEROID["GRS_1980",6378137,298.257222101]],
        PRIMEM["Greenwich",0],
        UNIT["Degree",0.0174532925199433]]
```

The `UNIT` object can represent angular or linear unit of measures.

```
<angular unit> = <unit>

<linear unit> = <unit>

<unit> = UNIT["<name>", <conversion factor>]

<conversion factor> = <number>
```

`<conversion factor>` specifies number of meters (for a linear unit) or number of radians (for an angular unit) per unit and must be greater than zero.

So the full string representation of UTM Zone 10N is

```
PROJCS["NAD 1983 UTM Zone 10N",
    GEOGCS["GCS North American 1983",
    DATUM[  "D North American 1983",SPHEROID["GRS 1980",6378137,298.257222101]],
    PRIMEM["Greenwich",0],UNIT["Degree",0.0174532925199433]],
    PROJECTION["Transverse_Mercator"],PARAMETER["False_Easting",500000.0],
    PARAMETER["False Northing",0.0],PARAMETER["Central Meridian",-123.0],
    PARAMETER["Scale Factor",0.9996],PARAMETER["Latitude of Origin",0.0],
    UNIT["Meter",1.0]]
```

A geocentric coordinate system is quite similar to a geographic coordinate system. It is represented by

```
<geocentric cs> = GEOCCS["<name>", <datum>, <prime meridian>, <linear unit>]
```

4 Feature Identity

4.1 Introduction

Unfortunately the issue of identity has become confused at the OpenGIS. This paper clarifies the interpretation of what the Abstract Specification intended with respect to feature identity that was used in this implementation specification.

4.2 Features vs. Real World Entities

'Features' are, according to the Abstract Specification, digital representations of real world entities. Feature Identity thus refers to mechanisms to identify such representations: not to identify the *real world entities* that are the subject of a representation. Thus two different representations of a real world entity (say the Mississippi River) will be two different features with distinct identities. Real world identification systems, such as title numbers, while possibly forming a sound basis for an implementation of a feature identity mechanism, are not of themselves such a mechanism.

4.3 Identity 'Ownership'

As feature identity thus pertains to the digital representation of real world entities and not to the entities themselves, designing an identity mechanism is clearly the province of the system implementer and not an Information Community.

4.4 Aspects of Identity

The term 'Identity' has been commonly used at OpenGIS in two utilitarian senses which, for convenience, we term 'discriminating identity' and 'referential identity'. Discriminating identity allows a client to reliably determine if two features it is holding are in fact the same, or not. A GUID is an example of a mechanism, which provides discriminating identity. Referential identity provides a client with a means of locating the implementation of the feature, subsequently allowing access to all state information (property values). CORBA IORs and COM Monikers are examples of referential identity mechanisms. Without some means of locating a feature, a GUID does not give referential identity.

When referring to identity, the Abstract Specification is referring to referential identity. This can be seen from its corresponding OIDs (Feature IDs) to pointers in object systems [1 para. 3.13.1.2]. Referential identity is important for a number of reasons. Clients often want to retrieve information from a particular feature within or across sessions and referential identity mechanisms provide a handle to do this. Referential identity is also important in the establishment of relationships between features. Any implementation providing only discriminating identity is not adequate for the requirements of OpenGIS.

4.5 Implementation Identity

Most existing and legacy feature implementations have some notion of identity (although some flat-file formats do not). In a typical RDBMS, a feature may be identified by its table (feature type) and a primary key consisting one or more attributes which together are unique. The table/key couplet provides discriminating identity within the context of the database. Provided an SQL evaluation engine is available, such a couplet can also provide referential identity.

In O-O systems, some form of persistent object reference usually provides identity. Some flat-file systems also provide feature identity through a unique integer identifier and associated location mechanism.

Each of these internal implementation identity mechanisms provides the identity services required for the functionality the implementation supports. For example, an RDBMS can easily relate two (or more) features together by establishing a relationship table, which uses keys to identify participating, features. Alternatively, a relationship can be established by defining a column in a feature table to be of foreign key type.

4.6 Identity and Database Federation

One of the goals of interoperability is the federation of different feature implementations (databases) into a unified network of cooperating databases: a ‘one-stop shop’ for feature data consumers. To achieve this federation and provide the basis for services such as the ability to establish relationships between features with differing implementation, some form of global identity scheme is required. The abstract specification recognized this requirement by mandating that feature identity “is unique for all features in all data sets everywhere through time” [1 para. 3.13.1.2]. This can best be done by extending the internal identity mechanisms into the global domain (thus avoiding the massive expense of building a new identity mechanism onto existing legacy data sets).

4.7 Exposing Identity

Using IORs for identity purposes obliges the server implementation to create one and only one CORBA object for each feature it exposes. For servers with some notion of persistent identity (usually the case) this is not a particularly onerous requirement. There will be problems with replication, client caching, smart proxies etc. but these problems are likely to arise for any reasonable complete identification scheme. These will need to be treated in due course. This implementation specification uses IORs for identity.

It has been suggested that a server’s internal identity mechanism (e.g. a table name & primary key for RDBMSs) can be used externally to provide feature identity thus avoiding the requirement to create a CORBA object for each Feature. However, this internal identity would need to be encapsulated in a CORBA wrapper to isolate clients from internal implementation details of servers: (otherwise clients would need to be prepared to deal with any number of different identity schemes). This requires the ID to be embedded in a CORBA object. If identity is required and the server is willing to support it, it ought to be done through the provision of Feature objects.

Alternatively, OpenGIS could mandate the establishment of a global identity mechanism that all compliant systems would be required to support. In the CORBA context this would take the form of an OpenGIS object referencing protocol. This rather autocratic course has the disadvantages that it would (a) exclude all legacy systems without major (i.e. expensive) re-establishment of feature identity and (b) unnecessarily restrict the choices of system implementers

4.8 Feature & Object Identity in CORBA

The CORBA 2.0 Basic Object Adapter is in the process of being replaced by the Portable Object Adapter (POA) [2]. The Object Adapter is the component of the Object Request Broker (ORB) which is responsible for locating and activating the implementation of an object given an object reference.

The POA will be adopted as a CORBA standard later this year (September 1997). The POA clarifies how persistent object identity is maintained across sessions and server processes. In particular this specification mandates the ability of an object's implementation (i.e. the server process) to be maximally responsible for the object's behavior including the definition of the object's identity and the relationship between an object's identity and its state.

The POA Specification defines Object Identity thus: "An Object Id is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references" ([2] 3.2.1 p3.3).

In the OpenGIS context, this policy means that an OpenGIS server implementation may assign Object Ids to Features (and other OpenGIS entities) as it sees fit. A server implementation with an underlying RDBMS could use an identity mechanism based on the table name and primary key of the row in the database corresponding to the feature. This would wholly incorporate the identity of the feature into the object references passed out by the implementation negating the need for addition identity information to be kept. As all Object Ids are encapsulated, clients do not need to concern themselves with the formulation of Object Ids.

This approach allows minimal implementations of OpenGIS interoperability over legacy systems (as existing identity mechanisms may be leveraged) whilst not imposing the requirement that clients deal with multiple identity mechanisms (as native Object Ids may be encapsulated within the Object reference).

4.9 Conclusion

The identity requirements of OpenGIS features within the CORBA context are adequately fulfilled by mandating the encapsulation of an internal server feature identity into any feature object references passed by an OpenGIS server process to a client. This is the approach that has been adopted for this specification.

5 Exposing Feature Type

The Abstract Specification states that each OpenGIS Feature Instance belongs to a recognized Feature Type [1 para. 2.11.1.1 p.17]. The specification also demands that Feature Type information (Feature Schema) be exposed to clients by OpenGIS compliant applications. The CORBA implementation specification must use some form of CORBA construct to fulfill this requirement. Earlier submissions offered the following responses to this problem:

Bentley Submission: InterfaceDef - an Interface Repository (IR) construct.

UCLA Submission: Not specified: there is no entity explicitly representing either Features or Feature Type.

Bentley's solution was criticized because the IR was allegedly expensive and there were also significant costs in synchronizing the IDL/IR representation with the underlying datastore technology's type system.

UCLA's solution is deficient because it assumes and relies upon an SQL query capability from both feature clients and feature implementations: a situation that may or may not apply. (If such a situation did apply an ODBC/SQL based solution would probably be a better choice than CORBA).

To avoid the expense of the IR and its associated synchronization problems an alternative was needed. If feature types are not IR constructs it follows that features must be accessible through a generic 'Feature' interface rather than type-specific interfaces as required by the first Bentley submission. Thus, *a generic Feature interface is needed* and is specified in this proposal.

If the interface is generic how do clients know how to deal with type specific properties? The obvious solution, *prima facie,* is that clients can ask a feature through the generic Feature interface for a list (sequence) of its properties as name/value pairs. However this obvious solution has a major flaw: it does not allow the client to generalize features by type (essentially each feature is of its own unique type). The number of feature types in a typical GIS is small compared to the number of feature instances (usually by a factor of two or three orders of magnitude). Thus, requiring each feature to respond separately to a *get_properties* operation and then have the client parse through the response to ascertain type information would usually be grossly inefficient. To avoid this unnecessary overhead, a separate FeatureType object can supply type information to a client once, which it can then use to extract type-specific information from potentially large numbers of features. Such a FeatureType object will usually map closely to some entity in the underlying datastore technology (table for RDBMS, class for OO, some form of external lexicon for flat-files). Thus, *a FeatureType interface was needed.*

What are the characteristics of this FeatureType interface? This depended on the scope of the abstract definition of feature type. Unfortunately the Abstract Specification is extremely vague on this point.

According to the abstract specification's Open Geodata Specification Model: "The feature type is an abstract class. It has a set of properties, each of which is distinguished by a role name (corresponding to the attribute name in a relational model, or the member variable name in an object-oriented model) and a value from the types chosen for that property. The role name and property types are defined by the specific feature class specification. The class specification is the schema for the feature class. A subset, potentially empty, of the properties are of type geometry and represent the spatial-temporal extent of the feature. Property interfaces include information on the data, such as accuracy information, the object class of its value(s), and if it is NULL (has no value)." [1 para. 3.13.1.1].

There are a number of problems with this definition, including:

the use of the word 'class' in a context (interoperability) where it is ambiguous.

confusion between a feature type specification and a feature instance specification (e.g. property types must be defined in the former, property values in the latter).

confusion between property *type* and a property *interface*: some properties may be defined as object references (interfaces) but this is hardly essential for every property.

there is no notion of feature type identity. (Issues like: If two features have identical property sets then are they *ipso facto* of the same type? If one feature's properties are a superset of another's, is that feature a sub-class of the other's type (i.e. can an instance of the first be described as an instance of the second)? Can properties defined as feature OIDs be restricted to features of a particular type or types? etc. are not answered) Note: this issue is distinct from the notion of feature identity.

no provision is made for feature behavior: an omission that would compromise the potential of OpenGIS interoperability (it should be noted that the definition of 'property' may be broad enough to include operations with typed parameter lists and return types).

no provision has been made for polymorphism.

Unfortunately these omissions and ambiguities make the abstract definition too weak to be usable as the basis for interoperability across divergent platforms particularly those such as CORBA that build on the object-oriented paradigm. The last two are particularly grievous when defining standards for OO systems. Some clarification is needed. This implementation specification makes the following clarifying assumptions:

Every feature instance has feature type.

A feature type, like an instance, has a unique identity (OID) which can be used as a type for property sets.

A feature type is defined by a set of typed properties and operations with parameter lists. Either the set of properties or operations or both may be null sets. Note that rows in a relation table still fit this definition, while allowing for feature behavior in OO environments.

A subset of a feature type's property set may be of type 'Geometry'

A feature must yield its type on demand to an OpenGIS client in a 'well-known' format. This may be done directly or by passing a reference to a 'FeatureType' object.

Properties are either of 'well-known' type or expose their type in a 'well-known' format on demand. They may be simple types (longs, floats, strings, etc.) or compound types (structs, unions, sequences) or object references.

If the underlying platform supports it (i.e. this is not required for OpenGIS compliance), features may be polymorphic i.e. features may conform to a number of types. (Whether this is achieved through single or multiple inheritance, aggregation or some other means may be DCP and/or implementation dependent.)

'Well-known' means defined using some means commonly understandable by OpenGIS clients. This could be explicitly defined in the implementation specification but preferably some means available at the DCP level ought to be used if possible (e.g. TypeCodes or IDL in CORBA).

6 References

1. *The OpenGIS Abstract Specification* Rev 1, OpenGIS Consortium, Inc OpenGIS Project Document Number 96-015R1, 1996.

2. *ORB Portability Joint Submission*, Draft 14, Open Management Group Inc, Document Number orbos/97-04-04, April 1997.

3. Jon Siegel, *CORBA Fundamentals and Programming*, John Wiley & Sons, 1996

4. Max Egenhofer & John Herring, "A Mathematical Framework for the Definition of Topological Relationships", in K Brassel & H Kishimoto, *Proceedings of the 4th Internation Symposium on Spatial Data Handling*, Zurich, 1990, p803-813.

5. *Object Query Service Specification*, Open Management Group Inc, OMG TC Document 95-1-1, 1995.

6. *Epicentre Model,* Petrotechnical Open Software Consortium (POSC), available at www.petroconsultants.com.

7 Full IDL Specification

```
module OGIS {

//------------------------------------------------------------------------
// Forward declarations of interfaces
//------------------------------------------------------------------------
interface Geometry;   // forward declarations
interface FeaturePropertySetIterator;
interface FeatureType;
interface PropertyDefIterator;
interface FeatureIterator;
interface QueryResultSetIterator;

//------------------------------------------------------------------------
// Common structures
//------------------------------------------------------------------------
typedef sequence<FeatureType> FeatureTypeSeq;
typedef string Istring;
typedef sequence <Istring> IStringSeq;
typedef sequence<octet> OctetSeq;

struct Decimal {
         long      precision;
         long      scale;
        OctetSeq   value;
     };


// Structure to describe name-value pairs
struct NVPair {
     Istring    name;   // name is a string
     any        value;  // value is an 'any' type
};

typedef sequence <NVPair> NVPairSeq;

struct PropertyDef {
     Istring       name;
     TypeCode      type;
     boolean       required;
};

typedef sequence<PropertyDef> PropertyDefSeq;

struct FeatureData {
     FeatureType   type;
     NVPairSeq props;
};

typedef sequence<FeatureData> FeatureDataSeq;
```

```
//------------------------------------------------------------------------
// Well-known Structures
//------------------------------------------------------------------------

struct WKSPoint {
     double    x;
     double    y;
};

typedef  sequence<WKSPoint>      WKSPointSeq;
typedef  sequence<WKSPoint>      WKSLineString;
typedef  sequence<WKSLineString>  WKSLineStringSeq;
typedef  sequence<WKSPoint>      WKSLinearRing;
typedef  sequence<WKSLinearRing> WKSLinearRingSeq;

struct WKSLinearPolygon {
     WKSLinearRing      externalBoundary;
     WKSLinearRingSeq   internalBoundaries;
};

typedef sequence <WKSLinearPolygon> WKSLinearPolygonSeq;

enum WKSType {
     WKSPointType,WKSMultiPointType, WKSLineStringType,WKSMultiLineStringType,
     WKSLinearRingType, WKSLinearPolygonType, WKSMultiLinearPolygonType,
     WKSCollectionType
};

union WKSGeometry       // near-equivalent to the 'CoordinateGeometry of the spec'

     switch (WKSType) {

          case WKSPointType:
               WKSPoint      point;

          case WKSMultiPointType:
               WKSPointSeq          multi_point;

          case WKSLineStringType:
               WKSLineString line_string;

          case WKSMultiLineStringType:
               WKSLineStringSeq     multi_line_string;

          case WKSLinearRingType:
               WKSLinearRing linear_ring;

          case WKSLinearPolygonType:
               WKSLinearPolygon   linear_polygon;

          case WKSMultiLinearPolygonType:
               WKSLinearPolygonSeq        multi_linear_polygon;

          case WKSCollectionType:
               sequence<WKSGeometry>   collection;
     };

struct Envelope {
     WKSPoint      minm;
     WKSPoint      maxm;
};

//------------------------------------------------------------------------
// Feature Interface

interface Feature {
     exception InvalidParams {string why;};
```

```
    exception PropertyNotSet {}; // Property does not exist.
    exception InvalidProperty {}; // Not a valid property for the given feature.
    exception InvalidValue {}; // value is not valid for property
    exception InvalidConversion {};
    exception RequiredProperty {}; // property is required for the given feature

    // feature type
    readonly attribute     FeatureType   feature_type;

    // geometry
    Geometry       get_geometry (in NVPairSeq geometry_context) raises (InvalidParams);

    // generic property methods to get/set property values
    boolean property_exists(in Istring name) raises(InvalidProperty);

    any get_property(in Istring name) raises(PropertyNotSet,InvalidProperty);

    void set_property(in Istring name, in any value)
                               raises(InvalidProperty, InvalidValue);

    void delete_property(in Istring name) raises(PropertyNotSet, InvalidProperty,
                                                 RequiredProperty);

   // accessing property values by property names
    string        get_string_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    float         get_float_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    double        get_double_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    long          get_long_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    short         get_short_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    boolean       get_boolean_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    Decimal       get_decimal_by_name(in Istring propertyName)
                  raises (InvalidConversion, InvalidProperty);

    OctetSeq      get_byte_stream_by_name(in Istring propertyName)
                  raises (InvalidConversion, InvalidProperty);

    Geometry      get_geometry_by_name(in Istring propertyName)
                  raises (PropertyNotSet, InvalidProperty);

    WKSGeometry   get_wksgeometry_by_name(in Istring propertyName)
                  raises (InvalidConversion, InvalidProperty);

    OctetSeq get_wkbgeometry_by_name(in Istring propertyName)
                  raises (InvalidConversion, InvalidProperty);


    NVPairSeq get_property_sequence(in unsigned long n);

    FeaturePropertySetIterator get_property_iterator();

    void destroy();

};

typedef sequence <Feature> FeatureSeq;

//------------------------------------------------------------------------
// FeaturePrepertySetIterator Interface
//------------------------------------------------------------------------
```

```
interface FeaturePropertySetIterator {

     exception IteratorInvalid {};

     // Get next NVPair structure
     boolean  next(out NVPair the_pair)
                  raises (IteratorInvalid);

     // Get next "n" NVPair structures.
     boolean  next_n(in unsigned long n, out NVPairSeq n_pairs)
                  raises (IteratorInvalid);

     // Discard the iterator
     void     destroy();

     // reset the iterator
     void     reset() raises (IteratorInvalid);

};

//------------------------------------------------------------------------
// FeatureFactory Interface
//------------------------------------------------------------------------
interface FeatureFactory {

     exception FeatureTypeInvalid {string why;};
     exception PropertiesInvalid {string why;};

     Feature  create_feature(in FeatureType type, in NVPairSeq properties)
                              raises (FeatureTypeInvalid, PropertiesInvalid);

     FeatureSeq  create_features(in FeatureDataSeq features)
                              raises (FeatureTypeInvalid, PropertiesInvalid);

};


//------------------------------------------------------------------------
// FeatureType Interface
//------------------------------------------------------------------------

interface FeatureType {

     exception     InheritanceUnsupported {};
     exception     PropertyDefInvalid {};

     // feature type name
     readonly attribute Istring   name;

     // feature type parents
     FeatureTypeSeq     get_parents() raises (InheritanceUnsupported);

     // feature type children
     FeatureTypeSeqget_children()raises (InheritanceUnsupported);

     // definition of properties for this feature type
     boolean            property_def_exists(in Istring name);
     PropertyDef        get_property_def(in Istring name) raises(PropertyDefInvalid);
     PropertyDefSeq     get_property_def_sequence(in long levels, in unsigned long n);
     PropertyDefIterator    get_property_def_iterator (in long levels);
     void               destroy();
};

typedef sequence<FeatureType> FeatureTypeSeq;
//------------------------------------------------------------------------
// FeatureTypeFactory Interface
//------------------------------------------------------------------------
interface FeatureTypeFactory {

     exception     InvalidParams {string why;};
```

```
     FeatureType   create(in string name, in PropertyDefSeq schema,
                          in FeatureTypeSeq parents)
                          raises(InvalidParams);
};

//------------------------------------------------------------------------
// PropertyDefIterator Interface
//------------------------------------------------------------------------
interface PropertyDefIterator {

     exception     IteratorInvalid {};

     // Get next PropertyDef structure
     boolean       next(out PropertyDef schema_property)
                          raises (IteratorInvalid);

     // Get next "n" PropertyDef structures
     boolean       next_n(in unsigned long n, out PropertyDefSeq schema_properties)
                          raises (IteratorInvalid);

     // Discard the iterator
     void          destroy();

     // reset the iterator
     void          reset() raises (IteratorInvalid);

};


//------------------------------------------------------------------------
// FeatureCollection Interface
//------------------------------------------------------------------------
interface FeatureCollection : Feature {

     exception     IteratorInvalid {};
     exception     PositionInvalid {};
     exception     FeatureInvalid {string why;};
     exception     PropertiesInvalid {string why;};

     readonly attribute long     number_features;
     FeatureTypeSeq   supported_feature_types();

     void     add_element (in Feature element) raises (FeatureInvalid);
     void     merge (in FeatureCollection elements) raises (FeatureInvalid);

     void     insert_element_at (in Feature element, in FeatureIterator where)
                  raises (FeatureInvalid, IteratorInvalid);
      void    replace_element_at (in Feature element, in FeatureIterator where)
                  raises (FeatureInvalid, IteratorInvalid, PositionInvalid);

     void     remove_element_at (in FeatureIterator where)
                  raises (IteratorInvalid, PositionInvalid);
     void     remove_all_elements ();

      Feature retrieve_element_at (in FeatureIterator where)
                  raises (IteratorInvalid, PositionInvalid);

      FeatureIterator    create_iterator();

 };

//------------------------------------------------------------------------
// FeatureCollectionFactory Interface
//------------------------------------------------------------------------
interface FeatureCollectionFactory {

     exception     FeatureTypeInvalid {string why;};
     exception     PropertyInvalid {string why;};
     exception     FeatureInvalid {string why;};

     FeatureCollection create(in FeatureType collection_type,
```

```
                    in NVPairSeq collection_properties,
                    in FeatureTypeSeq supported_feature_types)
                    raises (FeatureTypeInvalid, PropertyInvalid);

     FeatureCollection createFromCollection(in FeatureType collection_type,
                    in NVPairSeq collection_properties,
                    in FeatureTypeSeq supported_feature_types,
                    in FeatureCollection collection)
                    raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

     FeatureCollection createFromSequence(in FeatureType collection_type,
                    in NVPairSeq collection_properties,
                    in FeatureTypeSeq supported_feature_types,
                    in FeatureSeq list)
                    raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

};


//------------------------------------------------------------------------
// FeatureIterator Interface
//------------------------------------------------------------------------
interface FeatureIterator  {

     exception IteratorInvalid {};
     exception PositionInvalid {};
     exception FeatureNotAvailable {};

     exception InvalidConversion {};
     exception InvalidProperty {};
     exception PropertyNotSet {};
     exception InvalidParameters {};

     // iterating over features
     boolean     next (out Feature the_feature)
          raises (IteratorInvalid, PositionInvalid,
          FeatureNotAvailable);

     boolean   next_n (in short n, out FeatureSeq the_features)
          raises (IteratorInvalid, PositionInvalid,
          FeatureNotAvailable);

     void advance ()
          raises (IteratorInvalid, PositionInvalid);

     Feature  current ()
          raises (IteratorInvalid, PositionInvalid,
          FeatureNotAvailable);

     // accessing current feature via 'Feature'-like methods
     FeatureType   get_feature_type();

     Geometry      get_geometry(in NVPairSeq geometry_context) raises (InvalidParameters);

     boolean       property_exists(in Istring name) raises(InvalidProperty);
     any           get_property(in Istring name) raises (PropertyNotSet,
                                                         InvalidProperty,);

     string        get_string_by_name(in Istring propertyName)
                   raises (PropertyNotSet, InvalidProperty);

     float         get_float_by_name(in Istring propertyName)
                   raises (PropertyNotSet, InvalidProperty);

     double        get_double_by_name(in Istring propertyName)
                   raises (PropertyNotSet, InvalidProperty);

     long          get_long_by_name(in Istring propertyName)
                   raises (PropertyNotSet, InvalidProperty);

     short         get_short_by_name(in Istring propertyName)
```

```
                raises (PropertyNotSet, InvalidProperty);

    boolean     get_boolean_by_name(in Istring propertyName)
                raises (PropertyNotSet, InvalidProperty);

    Decimal     get_decimal_by_name(in Istring propertyName)
                raises (InvalidConversion, InvalidProperty);

    OctetSeq    get_byte_stream_by_name(in Istring propertyName)
                raises (InvalidConversion, InvalidProperty);

    Geometry    get_geometry_by_name(in Istring propertyName)
                raises (PropertyNotSet, InvalidProperty);

    WKSGeometry get_wksgeometry_by_name(in Istring propertyName)
                raises (InvalidConversion, InvalidProperty);

    OctetSeq get_wkbgeometry_by_name(in Istring propertyName)
                raises (InvalidConversion, InvalidProperty);


    NVPairSeq get_property_sequence(in unsigned long n);
    FeaturePropertySetIterator get_property_iterator();

    void    reset() raises (IteratorInvalid);

    boolean more();

    void    destroy();
};

//-------------------------------------------------------------------------
// ContainerFeatureCollection Interface
//-------------------------------------------------------------------------
interface ContainerFeatureCollection : FeatureCollection, FeatureFactory {
};


//-------------------------------------------------------------------------
// ContainerFeatureCollectionFactory Interface
//-------------------------------------------------------------------------
interface ContainerFeatureCollectionFactory {

    exception    FeatureTypeInvalid {string why;};
    exception    PropertyInvalid {string why;};
    exception    FeatureInvalid {string why;};

    ContainerFeatureCollection create(in FeatureType collection_type,
                in NVPairSeq collection_properties)
                raises (FeatureTypeInvalid, PropertyInvalid);

    ContainerFeatureCollection createFromCollection(in FeatureType collection_type,
                in NVPairSeq collection_properties,
                in FeatureCollection collection)
                raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

    ContainerFeatureCollection createFromSequence(in FeatureType collection_type,
                in NVPairSeq collection_properties,
                in FeatureSeq list)
                raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

    ContainerFeatureCollection createFromFeatureData(in FeatureType collection_type,
                in NVPairSeq collection_properties, in FeatureDataSeq list)
                raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

};

//-------------------------------------------------------------------------
// Queryable Collection Interfaces
//-------------------------------------------------------------------------
```

```
//------------------------------------------------------------------------
// QueryEvaluator Interface
//------------------------------------------------------------------------
interface QueryableFeatureCollection;     // forward declaration

interface QueryEvaluator {
     exception     QueryLanguageTypeNotSupported {};
     exception     InvalidQuery {string why;};
     exception     QueryProcessingError {string why;};
     exception     InvalidGeometry {string why;};
     exception     InvalidSpatialOperator {};

     enum QLType {
          SQLQuery, SQL_92Query, OQL, OQLBasic, OQL_93, OQL_93Basic
     };

     typedef sequence<QLType> QLTypeSeq;

     enum SpatialOperatorType {
          TouchOp, ContainsOp, WithinOp, DisjointOp, CrossesOp, OverlapsOp, IntersectsOp
     };

     readonly attribute QLTypeSeq ql_types;
     readonly attribute QLType default_ql_type;

     enum GeomSwitch { GeomType, WKSGeomType };
     union QueryGeom
          switch ( GeomSwitch ) {
               case GeomType:     Geometry      geom;

               case WKSGeomType:  WKSGeometry   wks_geom;
          };

     struct GeomConstraint {
          Istring             geom_name;
          SpatialOperatorType spatial_op;
          QueryGeom           geo;
     };

     typedef sequence<GeomConstraint> GeomConstraintSeq;

     QueryableFeatureCollection query(
                                 in string where_clause, in QLType ql_type,
                                 in GeomConstraintSeq geom_constraints )
                                 raises(QueryLanguageTypeNotSupported, InvalidQuery,
                                      InvalidGeometry, QueryProcessingError,
                                      InvalidSpatialOperator);
};

//------------------------------------------------------------------------
// QueryableFeatureCollection Interface
//------------------------------------------------------------------------
interface QueryableFeatureCollection : FeatureCollection, QueryEvaluator {
};


//------------------------------------------------------------------------
// QueryableFeatureCollectionFactory Interface
//------------------------------------------------------------------------
interface QueryableFeatureCollectionFactory {

     exception     FeatureTypeInvalid {string why;};
     exception     PropertyInvalid {string why;};
     exception     FeatureInvalid {string why;};

     QueryableFeatureCollection create(in FeatureType collection_type,
                       in NVPairSeq collection_properties)
                    raises (FeatureTypeInvalid, PropertyInvalid);

     QueryableFeatureCollection createFromCollection(in FeatureType collection_type,
                       in NVPairSeq collection_properties,
```

```
                    in FeatureCollection collection)
                raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

    QueryableFeatureCollection createFromSequence(in FeatureType collection_type,
                    in NVPairSeq collection_properties,
                    in FeatureSeq list)
                raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

    QueryableFeatureCollection createFromFeatureData(in FeatureType collection_type,
                    in NVPairSeq collection_properties, in FeatureDataSeq list)
                raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);
};

//------------------------------------------------------------------------
// QueryableContainerFeatureCollection Interface
//------------------------------------------------------------------------
interface QueryableContainerFeatureCollection: ContainerFeatureCollection, QueryEvaluator
{
};


//------------------------------------------------------------------------
// QueryableContainerFeatureCollectionFactory Interface
//------------------------------------------------------------------------
interface QueryableContainerFeatureCollectionFactory {

    exception     FeatureTypeInvalid {string why;};
    exception     PropertyInvalid {string why;};
    exception     FeatureInvalid {string why;};

    QueryableContainerFeatureCollection create(in FeatureType collection_type,
                in NVPairSeq collection_properties)
            raises (FeatureTypeInvalid, PropertyInvalid);

    QueryableContainerFeatureCollection createFromCollection(
                in FeatureType collection_type,
                in NVPairSeq collection_properties,in FeatureCollection collection)
            raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

    QueryableContainerFeatureCollection createFromSequence(
                in FeatureType collection_type, in NVPairSeq collection_properties,
                in FeatureSeq list)
            raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

    QueryableContainerFeatureCollection createFromFeatureData(
                in FeatureType collection_type,  in NVPairSeq collection_properties,
                in FeatureDataSeq list)
            raises (FeatureTypeInvalid,PropertyInvalid,FeatureInvalid);

};

//------------------------------------------------------------------------
// Spatial Reference Systems
//------------------------------------------------------------------------


//------------------------------------------------------------------------
// SpatialReferenceInfo
//------------------------------------------------------------------------
interface SpatialReferenceInfo {

    attribute string       name;
    attribute string       authority;
    attribute long         code;
    attribute string       alias;
    attribute string       abbreviation;
    attribute string       remarks;

    readonly attribute string       well_known_text;
};
```

```
//------------------------------------------------------------------------
// Unit interface
//------------------------------------------------------------------------
interface Unit : SpatialReferenceInfo {
};

//------------------------------------------------------------------------
// AngularUnit interface
//------------------------------------------------------------------------
interface AngularUnit : Unit {

     attribute double   radians_per_unit;

};

//------------------------------------------------------------------------
// LinearUnit interface
//------------------------------------------------------------------------
interface LinearUnit : Unit {

     attribute double   metres_per_unit;

};

//------------------------------------------------------------------------
// Ellipsoid interface
//------------------------------------------------------------------------
interface Ellipsoid : SpatialReferenceInfo {

     attribute double   semi_major_axis;
     attribute double   semi_minor_axis;
     attribute double   inverse_flattening;
     attribute LinearUnit    axis_unit;

};

//------------------------------------------------------------------------
// HorizontalDatum interface
//------------------------------------------------------------------------
interface HorizontalDatum : SpatialReferenceInfo {

     attribute Ellipsoid    base_ellipsoid;

};


//------------------------------------------------------------------------
// PrimeMeridian interface
//------------------------------------------------------------------------
interface PrimeMeridian : SpatialReferenceInfo {

     attribute double   longitude;
     attribute AngularUnit   angular_units;

};

//------------------------------------------------------------------------
// SpatialReferenceSystem interface
//------------------------------------------------------------------------
interface SpatialReferenceSystem : SpatialReferenceInfo {
};

//------------------------------------------------------------------------
// GeodeticSpatialReferenceSystem interface
//------------------------------------------------------------------------
interface GeodeticSpatialReferenceSystem : SpatialReferenceSystem {
};

//------------------------------------------------------------------------
// GeographicCoordinateSystem interface
```

```
//------------------------------------------------------------------------
interface GeographicCoordinateSystem : GeodeticSpatialReferenceSystem {

     attribute string       usage;            // description?
     attribute HorizontalDatum   horizontal_datum;
     attribute AngularUnit       angular_unit;
     attribute PrimeMeridian     prime_meridian;

};

//------------------------------------------------------------------------
// Parameter interface
//------------------------------------------------------------------------
interface Parameter : SpatialReferenceInfo {

     attribute Unit     units;
     attribute double        value;

};

typedef sequence<Parameter> ParameterSeq;

//------------------------------------------------------------------------
// ParameterList interface
//------------------------------------------------------------------------
interface ParameterList {

     readonly attribute long number_parameters;

     ParameterSeq  get_default_parameters();

     void          set_parameters (in ParameterSeq parameters);
     ParameterSeq  get_parameters ();

};

//------------------------------------------------------------------------
// GeographicTransform interface
//------------------------------------------------------------------------
interface GeographicTransform : SpatialReferenceInfo {

     attribute GeographicCoordinateSystem  source_gcs;
     attribute GeographicCoordinateSystem  target_gcs;

     WKSGeometry   forward (in WKSGeometry source_geometry);
     WKSGeometry   inverse (in WKSGeometry source_geometry);

};


//------------------------------------------------------------------------
// Projection interface
//------------------------------------------------------------------------
interface Projection : SpatialReferenceInfo {

     readonly attribute string    usage;
     readonly attribute string    classification;

     WKSGeometry   forward (in WKSGeometry source_geometry);
     WKSGeometry   inverse (in WKSGeometry source_geometry);

     readonly attribute ParameterList    parameters;

     attribute AngularUnit  angular_units;
     attribute LinearUnit   linear_units;
     attribute Ellipsoid    base_ellipsoid;

};

//------------------------------------------------------------------------
// ProjectedCoordinateSystem interface
```

```
//------------------------------------------------------------------------
interface ProjectedCoordinateSystem : GeodeticSpatialReferenceSystem {

     attribute string       usage;
     attribute GeographicCoordinateSystem    geographic_coordinate_system;
     attribute LinearUnit   linear_units;
     attribute Projection   base_projection;

     readonly attribute ParameterList    parameters;

     WKSGeometry     forward (in WKSGeometry source_geometry);
     WKSGeometry     inverse (in WKSGeometry source_geometry);

};


//------------------------------------------------------------------------
// SpatialReferenceSystemFactory interface
//------------------------------------------------------------------------
interface SpatialReferenceSystemFactory {

     SpatialReferenceSystem  create_from_WKT (in string srs_wkt);

};

//------------------------------------------------------------------------
// SpatialReferenceComponentFactory interface
//------------------------------------------------------------------------
interface SpatialReferenceComponentFactory {

     readonly attribute  string  authority;

     ProjectedCoordinateSystem   create_projected_coordinate_system (in long code);
     GeographicCoordinateSystem  create_geographic_coordinate_system (in long code);
     Projection                  create_projection (in long code);
     GeographicTransform         create_geographic_transform (in long code);
     HorizontalDatum             create_horizontal_datum (in long code);
     Ellipsoid                   create_ellipsoid (in long code);
     PrimeMeridian               create_prime_meridian (in long code);
     LinearUnit                  create_linear_unit (in long code);
     AngularUnit                 create_angular_unit (in long code);

};

//------------------------------------------------------------------------
// Geometry interface
//------------------------------------------------------------------------
interface Geometry {

     exception WKBNotImplemented {};

     enum EgenhoferElement {
          Empty, NotEmpty, NoTest
     };

     struct EgenhoferOperator {
          EgenhoferElement elements[3][3];
     };

     readonly  attribute short    dimension;    // dimension of the geometry
                                       // - not the coordinate system
     readonly  attribute Envelope range_envelope;// minBoundBox in abstract spec

     readonly  attribute SpatialReferenceSystem  spatial_reference_system;

     // geometric characteristics
     boolean       is_empty();
     boolean       is_simple();
     boolean       is_closed();

     // constructive operators
```

```
     Geometry      copy();
     Geometry      boundary();
     Geometry      buffer (in double distance);
     Geometry      convex_hull();

     // WKS operators
     WKSGeometry   export();        // export geometry to WKS
     OctetSeq      export_WKBGeometry()    // export geometry to WKB
                        raises (WKBNotImplemented);

     // relational operators
     boolean       equals (in Geometry other);
     boolean       touches (in Geometry other);
     boolean       contains (in Geometry other);
     boolean       within (in Geometry other);
     boolean       disjoint (in Geometry other);
     boolean       crosses (in Geometry other);
     boolean       overlaps (in Geometry other);
     boolean       intersects (in Geometry other);
     boolean       relate (in Geometry other, in EgenhoferOperator operator);

     // metric operators
     double        distance (in Geometry other);

     // set operators
     Geometry      intersection (in Geometry other);
     Geometry      union_op(in Geometry other);
     Geometry      difference (in Geometry other);
     Geometry      symmetric_difference (in Geometry other);

     void          destroy();
};

//------------------------------------------------------------------------
// GeometryFactory interface
//------------------------------------------------------------------------
interface GeometryFactory {
     exception InvalidWKS {string why;};
     exception InvalidWKB {string why;};
     exception WKBNotImplemented {};

     Geometry  create(in Geometry existing);

     Geometry  create_from_WKS(in SpatialReferenceSystem srs,in WKSGeometry geo)
                        raises (InvalidWKS);

     Geometry  create_from_WKB(in SpatialReferenceSystem srs,in OctetSeq geo)
                        raises (InvalidWKB, WKBNotImplemented);
};


interface GeometryCollection;
interface GeometryIterator;

//------------------------------------------------------------------------
// GeometryCollection interface
//------------------------------------------------------------------------
interface GeometryCollection : Geometry {

     exception IteratorInvalid {};
     exception PositionInvalid {};
     exception GeometryInvalid {};

     readonly  attribute    long number_elements;

     // these operations allowing for arbitrary collections
     void     add_element (in Geometry element) raises (GeometryInvalid);
     void     merge (in GeometryCollection elements) raises (GeometryInvalid);

     void     insert_element_at (in Geometry element, in GeometryIterator where)
                  raises (GeometryInvalid, IteratorInvalid);
```

```
    void     replace_element_at (in Geometry element, in GeometryIterator where)
                 raises (GeometryInvalid, IteratorInvalid, PositionInvalid);

    void     remove_element_at (in GeometryIterator where)
                 raises (IteratorInvalid, PositionInvalid);
    void     remove_all_elements ();

    // retrieve a geometry from a collection
    Geometry retrieve_element_at (in GeometryIterator where)
                 raises (IteratorInvalid, PositionInvalid);

    // create an iterator over the collection
    GeometryIterator    create_iterator();
};


//------------------------------------------------------------------------
// GeometryIterator interface
//------------------------------------------------------------------------

interface GeometryIterator {

    exception     IteratorInvalid {};
    exception     PositionInvalid {};

    Geometry      next ()  raises (IteratorInvalid, PositionInvalid);
    void          reset()  raises (IteratorInvalid);
    boolean       more();
    void          destroy();
};


//------------------------------------------------------------------------
// Point interface
//------------------------------------------------------------------------
interface Point : Geometry {

    attribute     WKSPoint coordinates;

};

//------------------------------------------------------------------------
// PointFactory interface
//------------------------------------------------------------------------
interface PointFactory : GeometryFactory {

    exception     InvalidWKSPoint {};
    exception     InvalidWKBPoint {};

    Point         create_from_Point(in Point existing);

    Point         create_from_WKSPoint(in SpatialReferenceSystem srs, in WKSPoint geo)
                      raises (InvalidWKSPoint);

    Point         create_from_WKBPoint(in SpatialReferenceSystem srs, in OctetSeq geo)
                      raises (InvalidWKBPoint, WKBNotImplemented);
};

//------------------------------------------------------------------------
// MultiPoint interface
//------------------------------------------------------------------------
interface MultiPoint : GeometryCollection {
};


//------------------------------------------------------------------------
// MultiPointFactory interface
//------------------------------------------------------------------------
interface MultiPointFactory : GeometryFactory {

    exception     InvalidWKSMultiPoint {};
```

```
     exception     InvalidWKBMultiPoint {};

     MultiPoint    create_from_MultiPoint(in MultiPoint existing);

     MultiPoint    create_from_WKSMultiPoint(in SpatialReferenceSystem srs,
                              in WKSPointSeq geo)
                         raises (InvalidWKSMultiPoint);

     MultiPoint    create_from_WKBMultiPoint(in SpatialReferenceSystem srs,
                              in OctetSeq geo)
                         raises (InvalidWKBMultiPoint,WKBNotImplemented);

};


//------------------------------------------------------------------------
// Curve interface
//------------------------------------------------------------------------
interface Curve : Geometry {

     exception     OutOfDomain {};

     readonly  attribute  double length;
     readonly  attribute  Point  start_point;
     readonly  attribute  Point  end_point;
     readonly  attribute  WKSPoint    start_point_as_WKS;
     readonly  attribute  WKSPoint    end_point_as_WKS;

     boolean  is_planar();

     Point  value (in double r) raises (OutOfDomain);
     WKSPoint  value_as_WKS (in double r) raises (OutOfDomain);

};

//------------------------------------------------------------------------
// LineString interface
//------------------------------------------------------------------------
interface LineString : Curve  {

     exception     InvalidIndex{};
     exception     MinimumPoints{};

     readonly attribute     long     num_points;

     Point     get_point_by_index (in long index) raises (InvalidIndex);
     WKSPoint  get_point_by_index_as_WKS (in long index) raises (InvalidIndex);

     void      set_point_by_index (in WKSPoint new_point, in long index)
                    raises (InvalidIndex);
     void      set_point_by_index_with_WKS (in WKSPoint new_point, in long index)
                    raises (InvalidIndex);

     void      insert_point_by_index (in Point new_point, in long index)
                    raises (InvalidIndex);
     void      insert_point_by_index_with_WKS (in WKSPoint new_point, in long index)
                    raises (InvalidIndex);

     void      append_point (in Point new_point);
     void      append_point_with_WKS (in WKSPoint new_point);

     void      delete_point_by_index (in long index)
                    raises (InvalidIndex, MinimumPoints);
};

//------------------------------------------------------------------------
// LineStringFactory interface
//------------------------------------------------------------------------
interface LineStringFactory : GeometryFactory {

     exception     InvalidWKSLineString {};
```

```
     exception     InvalidWKBLineString {};

     LineString    create_from_LineString(in LineString existing);

     LineString    create_from_WKSLineString (in SpatialReferenceSystem srs,
                                              in WKSLineString geo)
                               raises (InvalidWKSLineString);

     LineString    create_from_WKBLineString(in SpatialReferenceSystem srs,
                                             in OctetSeq geo)
                               raises (InvalidWKBLineString,WKBNotImplemented);
};

//------------------------------------------------------------------------
// Ring interface
//------------------------------------------------------------------------
interface Ring : Curve  {
};

//------------------------------------------------------------------------
// LinearRing interface
//------------------------------------------------------------------------
interface LinearRing : Ring, LineString  {
};

//------------------------------------------------------------------------
// MultiCurve interface
//------------------------------------------------------------------------
interface MultiCurve : GeometryCollection  {

     readonly attribute double  length;

};

//------------------------------------------------------------------------
// MultiLineString interface
//------------------------------------------------------------------------
interface MultiLineString : MultiCurve {
};


//------------------------------------------------------------------------
// MultiLineStringFactory interface
//------------------------------------------------------------------------
interface MultiLineStringFactory : GeometryFactory {
     exception     InvalidWKSMultiLineString {};
     exception     InvalidWKBMultiLineString {};

     MultiLineString    create_from_MultiLineString(in MultiLineString existing);

     MultiLineString    create_from_WKSMultiLineString(in SpatialReferenceSystem srs,
                              in WKSLineStringSeq geo)
                         raises (InvalidWKSMultiLineString);

     MultiLineString    create_from_WKBMultiLineString(in SpatialReferenceSystem srs,
                              in OctetSeq geo)
                         raises (InvalidWKBMultiLineString,WKBNotImplemented);
};


//------------------------------------------------------------------------
// Surface interface
//------------------------------------------------------------------------
interface Surface : Geometry {

     readonly attribute  double       area;
     readonly attribute  Point        centroid;
     readonly attribute  WKSPoint     centroid_as_WKS;
     readonly attribute  Point        point_on_surface;
     readonly attribute  WKSPoint     point_on_surface_as_WKS;
```

```
    boolean  is_planar();
};

//-------------------------------------------------------------------------
// Polygon interface
//-------------------------------------------------------------------------
interface Polygon : Surface {

    readonly attribute     Ring          exterior_ring;
    readonly attribute     WKSGeometry   exterior_ring_as_WKS;

    readonly attribute     MultiCurve    interior_rings;
    readonly attribute     WKSGeometry   interior_rings_as_WKS;
 };


//-------------------------------------------------------------------------
// LinearPolygon interface
//-------------------------------------------------------------------------
interface LinearPolygon : Polygon {

};


//-------------------------------------------------------------------------
// LinearPolygonFactory interface
//-------------------------------------------------------------------------
interface LinearPolygonFactory : GeometryFactory {

    exception     InvalidWKSLinearPolygon {};
    exception     InvalidWKBLinearPolygon {};

    LinearPolygon create_from_LinearPolygon(in LinearPolygon existing);

    LinearPolygon create_from_WKSLinearPolygon(in SpatialReferenceSystem srs,
                          in WKSLinearPolygon geo)
                      raises (InvalidWKSLinearPolygon);

    LinearPolygon create_from_WKBLinearPolygon(in SpatialReferenceSystem srs,
                          in OctetSeq geo)
                      raises (InvalidWKBLinearPolygon,WKBNotImplemented);
};


//-------------------------------------------------------------------------
// MultiSurface interface
//-------------------------------------------------------------------------
interface MultiSurface : GeometryCollection  {

    readonly attribute     double    area;

};

//-------------------------------------------------------------------------
// MultiPolygon interface
//-------------------------------------------------------------------------
interface MultiPolygon : MultiSurface {
};

//-------------------------------------------------------------------------
// MultiLinearPolygon interface
//-------------------------------------------------------------------------
interface MultiLinearPolygon : MultiPolygon {
};

//-------------------------------------------------------------------------
// MultiLinearPolygon interface
//-------------------------------------------------------------------------
interface MultiLinearPolygonFactory : GeometryFactory {

    exception     InvalidWKSMultiLinearPolygon {};
```

```
    exception     InvalidWKBMultiLinearPolygon {};

    MultiLinearPolygon create_from_MultiLinearPolygon(in MultiLinearPolygon existing);

    MultiLinearPolygon create_from_WKSMultiLinearPolygon(in SpatialReferenceSystem srs,
                              in WKSLinearPolygonSeq geo)
                         raises (InvalidWKSMultiLinearPolygon);

    MultiLinearPolygon create_from_WKBMultiLinearPolygon(in SpatialReferenceSystem srs,
                              in OctetSeq geo)
                         raises (InvalidWKBMultiLinearPolygon);
};


}; // End OGIS Module
```