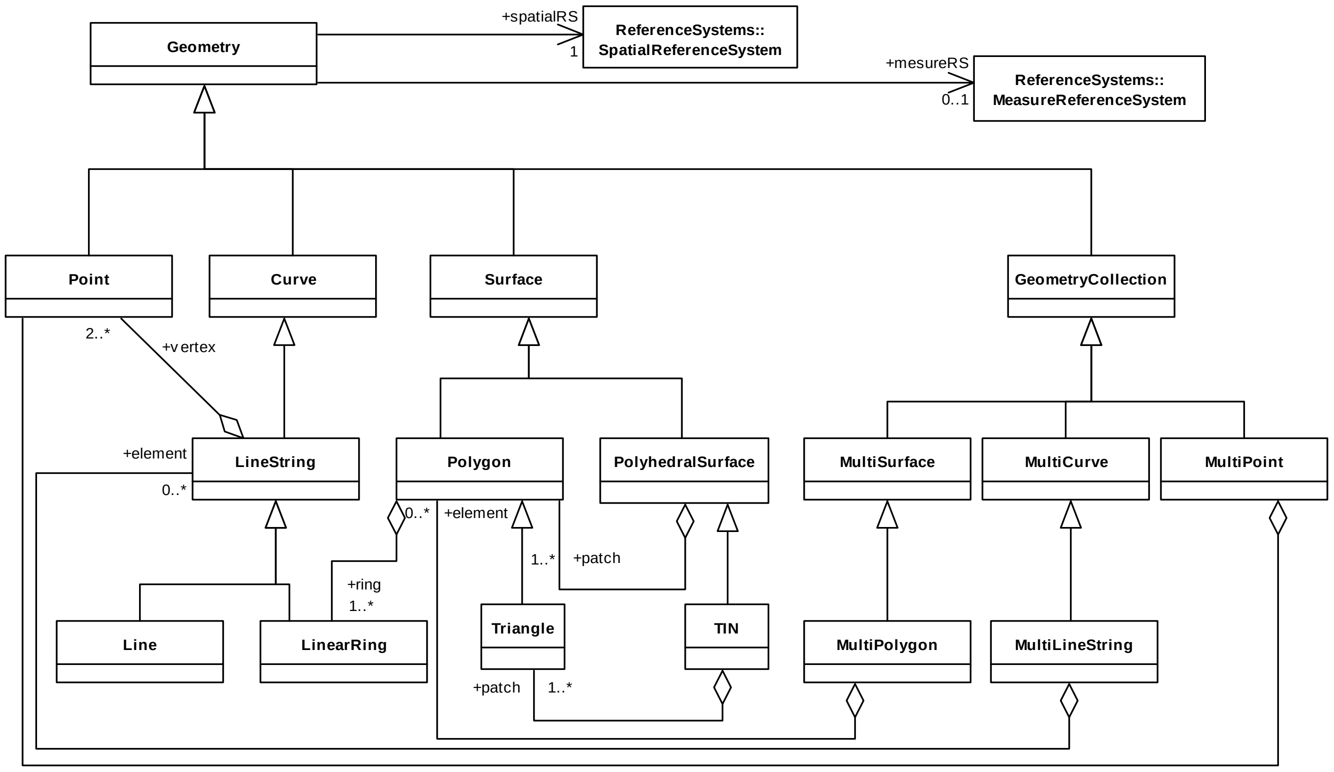


Figure 1: Geometry class hierarchy

Figure 1 is based on an extended Geometry model with specialized 0-, 1- and 2-dimensional collection classes named MultiPoint, MultiLineString and MultiPolygon for modeling geometries corresponding to collections of Points, LineStrings and Polygons, respectively. MultiCurve and MultiSurface are introduced as superclasses that generalize the collection interfaces to handle Curves and Surfaces. Figure 1 shows aggregation lines between the leaf-collection classes and their element classes; the aggregation lines for non-leaf-collection classes are described in the text. Non-homogeneous collections are instances of GeometryCollection.

The attributes, methods and assertions for each Geometry class are described below. In describing methods, this is used to refer to the receiver of the method (the object being messaged).

6.1.2 Geometry

6.1.2.1 Description

Geometry is the root class of the hierarchy. Geometry is an abstract (non-instantiable) class.

The instantiable subclasses of Geometry defined in this Standard are restricted to 0, 1 and 2-dimensional geometric objects that exist in 2, 3 or 4-dimensional coordinate space ($\Re^2$, $\Re^3$ or $\Re^4$). Geometry values in R^2 have points with coordinate values for x and y. Geometry values in R^3 have points with coordinate values for x, y and z or for x, y and m. Geometry values in R^4 have points with coordinate values for x, y, z and m. The interpretation of the coordinates is subject to the coordinate reference systems associated to the point. All coordinates within a geometry object should be in the same coordinate reference systems. Each coordinate shall be unambiguously associated to a coordinate reference system either directly or through its containing geometry.

The z coordinate of a point is typically, but not necessarily, represents altitude or elevation. The m coordinate represents a measurement.

All Geometry classes described in this standard are defined so that instances of Geometry are topologically closed, i.e. all represented geometries include their boundary as point sets. This does not affect their representation, and open version of the same classes may be used in other circumstances, such as topological representations.

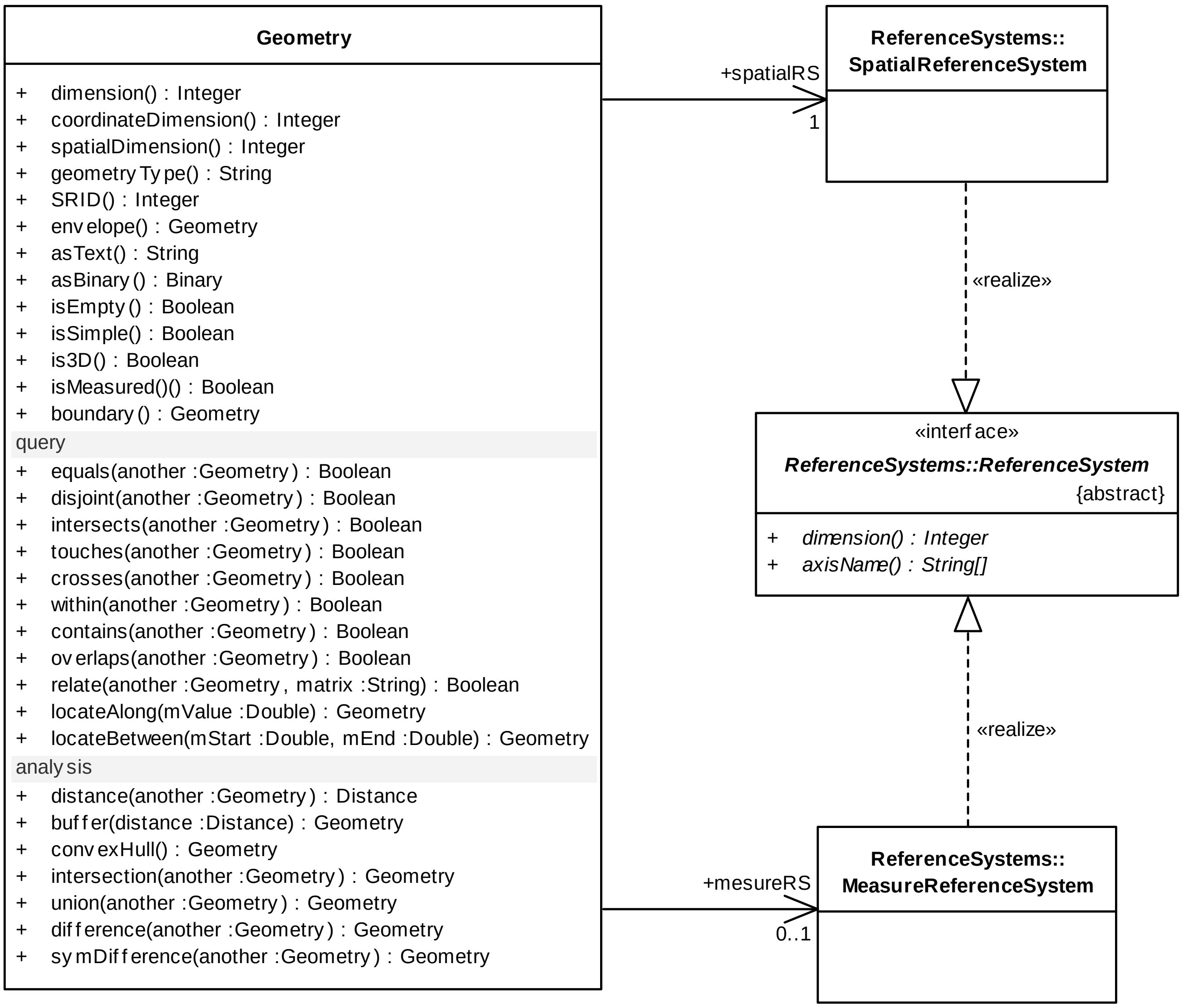

Geometry
+ dimension() : Integer
+ coordinateDimension() : Integer
+ spatialDimension() : Integer
+ geometryType() : String
+ SRID() : Integer
+ envelope() : Geometry
+ asText() : String
+ asBinary() : Binary
+ isEmpty() : Boolean
+ isSimple() : Boolean
+ is3D() : Boolean
+ isMeasured()() : Boolean
+ boundary() : Geometry
query
+ equals(another :Geometry) : Boolean
+ disjoint(another :Geometry) : Boolean
+ intersects(another :Geometry) : Boolean
+ touches(another :Geometry) : Boolean
+ crosses(another :Geometry) : Boolean
+ within(another :Geometry) : Boolean
+ contains(another :Geometry) : Boolean
+ overlaps(another :Geometry) : Boolean
+ relate(another :Geometry, matrix :String) : Boolean
+ locateAlong(mValue :Double) : Geometry
+ locateBetween(mStart :Double, mEnd :Double) : Geometry
analysis
+ distance(another :Geometry) : Distance
+ buffer(distance :Distance) : Geometry
+ convexHull() : Geometry
+ intersection(another :Geometry) : Geometry
+ union(another :Geometry) : Geometry
+ difference(another :Geometry) : Geometry
+ symDifference(another :Geometry) : Geometry
+spatialRS
1
ReferenceSystems::
SpatialReferenceSystem
«realize»
«interface»
ReferenceSystems::ReferenceSystem
{abstract}
+ dimension() : Integer
+ axisName() : String[]
«realize»
+mesureRS
0..1
ReferenceSystems::
MeasureReferenceSystem

Figure 2: Geometry class operations

6.1.2.2 Basic methods on geometric objects

— **Dimension** ( ): Integer — The inherent dimension of *this* geometric object, which must be less than or equal to the coordinate dimension. In non-homogeneous collections, this will return the largest topological dimension of the contained objects.

— **GeometryType** ( ): String — Returns the name of the instantiable subtype of Geometry of which *this* geometric object is an instantiable member. The name of the subtype of Geometry is returned as a string.

— **SRID** ( ): Integer — Returns the Spatial Reference System ID for *this* geometric object. This will normally be a foreign key to an index of reference systems stored in either the same or some other datastore.

— **Envelope** ( ): Geometry — The minimum bounding box for *this* Geometry, returned as a Geometry. The polygon is defined by the corner points of the bounding box [(MINX, MINY), (MAXX, MINY), (MAXX, MAXY), (MINX, MAXY), (MINX, MINY)]. Minimums for Z and M may be added. The simplest representation of an Envelope is as two direct positions, one containing all the minimums, and another all the maximums. In some cases, this coordinate will be outside the range of validity for the Spatial Reference System.

— **AsText** ( ): String — Exports *this* geometric object to a specific Well-known Text Representation of Geometry.

— **AsBinary** ( ): Binary — Exports *this* geometric object to a specific Well-known Binary Representation of Geometry.

— **IsEmpty** ( ): Integer — Returns 1 (TRUE) if *this* geometric object is the empty Geometry. If true, then this geometric object represents the empty point set ∅ for the coordinate space. The return type is integer, but is interpreted as Boolean, TRUE=1, FALSE=0.

— **IsSimple** ( ): Integer — Returns 1 (TRUE) if *this* geometric object has no anomalous geometric points, such as self intersection or self tangency. The description of each instantiable geometric class will include the specific conditions that cause an instance of that class to be classified as not simple. The return type is integer, but is interpreted as Boolean, TRUE=1, FALSE=0.

— **Is3D** ( ): Integer — Returns 1 (TRUE) if *this* geometric object has z coordinate values.

— **IsMeasured** ( ): Integer — Returns 1 (TRUE) if *this* geometric object has m coordinate values.

— **Boundary** ( ): Geometry — Returns the closure of the combinatorial boundary of *this* geometric object (Reference [1], section 3.12.2). Because the result of this function is a closure, and hence topologically closed, the resulting boundary can be represented using representational Geometry primitives (Reference [1], section 3.12.2). The return type is integer, but is interpreted as Boolean, TRUE=1, FALSE=0.

6.1.2.3 Methods for testing spatial relations between geometric objects

The methods in this subclause are defined and described in more detail following the description of the sub-types of Geometry. For each of the following, the return type is integer, but is interpreted as Boolean, TRUE=1, FALSE=0.

— **Equals** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object is “spatially equal” to anotherGeometry.

— **Disjoint** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object is “spatially disjoint” from anotherGeometry.

— **Intersects** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object “spatially intersects” anotherGeometry.

— **Touches** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object “spatially touches” anotherGeometry.

— **Crosses** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object “spatially crosses’ anotherGeometry.

— **Within** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object is “spatially within” anotherGeometry.

— **Contains** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object “spatially contains” anotherGeometry.

— **Overlaps** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object "spatially overlaps" anotherGeometry.

— **Relate** (anotherGeometry: Geometry, intersectionPatternMatrix: String): Integer — Returns 1 (TRUE) if *this* geometric object is spatially related to anotherGeometry by testing for intersections between the interior, boundary and exterior of the two geometric objects as specified by the values in the intersectionPatternMatrix. This returns FALSE if all the tested intersections are empty except exterior (this) intersect exterior (another).

— **LocateAlong** (mValue: Double): Geometry — Returns a derived geometry collection value that matches the specified m coordinate value. See Subclause 6.1.2.6 "Measures on Geometry" for more details.

— **LocateBetween** (mStart: Double, mEnd: Double): Geometry — Returns a derived geometry collection value that matches the specified range of m coordinate values inclusively. See Subclause 6.1.2.6 "Measures on Geometry" for more details.

6.1.2.4 Methods that support spatial analysis

All of the following are geometric analysis and depend on the accuracy of the coordinate representations and the limitations of linear interpolation in this standard. The accuracy of the result at a fine level will be limited by these and related issues.

— **Distance** (anotherGeometry: Geometry):Double — Returns the shortest distance between any two Points in the two geometric objects as calculated in the spatial reference system of *this* geometric object. Because the geometries are closed, it is possible to find a point on each geometric object involved, such that the distance between these 2 points is the returned distance between their geometric objects.

— **Buffer** (distance: Double): Geometry — Returns a geometric object that represents all Points whose distance from *this* geometric object is less than or equal to distance. Calculations are in the spatial reference system of *this* geometric object. Because of the limitations of linear interpolation, there will often be some relatively small error in this distance, but it should be near the resolution of the coordinates used.

— **ConvexHull** ( ): Geometry — Returns a geometric object that represents the convex hull of *this* geometric object. Convex hulls, being dependent on straight lines, can be accurately represented in linear interpolations for any geometry restricted to linear interpolations.

— **Intersection** (anotherGeometry: Geometry): Geometry — Returns a geometric object that represents the Point set intersection of *this* geometric object with anotherGeometry.

— **Union** (anotherGeometry: Geometry): Geometry — Returns a geometric object that represents the Point set union of *this* geometric object with anotherGeometry.

— **Difference** (anotherGeometry: Geometry): Geometry — Returns a geometric object that represents the Point set difference of *this* geometric object with anotherGeometry.

— **SymDifference** (anotherGeometry: Geometry): Geometry — Returns a geometric object that represents the Point set symmetric difference of *this* geometric object with anotherGeometry.

6.1.2.5 Use of Z and M coordinate values

A Point value may include a z coordinate value. The z coordinate value traditionally represents the third dimension (i.e. 3D). In a Geographic Information System (GIS) this may be height above or below sea level. For example: A map might have point identifying the position of a mountain peak by its location on the earth, with the x and y coordinate values, and the height of the mountain, with the z coordinate value.

A Point value may include an m coordinate value. The m coordinate value allows the application environment to associate some measure with the point values. For example: A stream network may be modeled as multilinestring value with the m coordinate values measuring the distance from the mouth of stream. The method LocateBetween may be used to find all the parts of the stream that are between, for example, 10 and 12 kilometers from the mouth. There are no constraints on the m coordinate values in a Geometry (e.g., the m coordinate values do not have to be continually increasing along a LineString value).

Observer methods returning Point values include z and m coordinate values when they are present.

Spatial operations work in the "map geometry" of the data and will therefore not reflect z or m values in calculations (e.g., Equals, Length) or in generation of new geometry values (e.g., Buffer, ConvexHull, Intersection). This is done by projecting the geometric objects onto the horizontal plane to obtain a "footprint" or "shadow" of the objects for the purposed of map calculations. In other words, it is possible to store and obtain z (and m) coordinate values but they are ignored in all other operations which are based on map geometries. Implementations are free to include true 3D geometric operations, but should be consistent with ISO 19107..

6.1.2.6 Measures on Geometry

The LocateAlong and LocateBetween methods derive MultiPoint or MultiCurve values from the given geometry that match a measure or a specific range of measures from the start measure to the end measure. The LocateAlong method is a variation of the LocateBetween method where the start measure and end measure are equal. (See SQL/MM [1])

6.1.2.6.1 Empty Sets

A null value is returned for empty sets.

6.1.2.6.2 Geometry values without m coordinate values.

An empty set of type Point is returned for geometry values without m coordinate values.

6.1.2.6.3 Zero-dimensional geometry values

Only points in the 0-dimensional geometry values with m coordinate values between *SM* and *EM* inclusively are returned as multipoint value. If no matching m coordinate values are found, then an empty set of type Point is returned.

For example:

a) If LocateAlong is invoked with an *M* value of 4 on a MultiPoint value with well-known text representation:
   multipoint m(1 0 4, 1 1 1, 1 2 2, 3 1 4, 5 3 4)
   then the result is the following MultiPoint value with well-known text representation:
   multipoint m(1 0 4, 3 1 4, 5 3 4)
b) If LocateBetween is invoked with an SM value of 2 and an EM value of 4 on a MultiPoint value with well-known text representation:
   multipoint m(1 0 4, 1 1 1, 1 2 2, 3 1 4, 5 3 5, 9 5 3, 7 6 7)
   then the result is the following MultiPoint value with well-known text representation:
   multipoint m(1 0 4, 1 2 2, 3 1 4, 9 5 3)
c) If LocateBetween is invoked with an *SM* value of 1 and an *EM* value of 4 on a Point value with well-known text representation:
   point m(7 6 7)

then the result is the following MultiPoint value with well-known text representation:

point m empty

d) If LocateBetween is invoked with an *SM* value of 7 and an *EM* value of 7 on a Point value with well-known text representation:

point m(7 6 7)

then the result is the following MultiPoint value with well-known text representation:

multipoint m(7 6 7)

6.1.2.6.4 One-dimensional geometry value

Interpolation is used to determine any points on the 1-dimensional geometry with an m coordinate value between *mStart* and *mEnd* inclusively. The implementation-defined interpolation algorithm is used to estimate values between measured values, usually using a mathematical function. The interpolation is within a Curve element and not across Curve elements in a MultiCurve. For example, given a measure of 6 and a 2-point LineString where the m coordinate value of start point is 4 and the m coordinate value of the end point is 8, since 6 is halfway between 4 and 8, the interpolation algorithm would be a point on the LineString halfway between the start and end points.

The results are produced in a geometry collection. If there are consecutive points in the 1-dimensional geometry with an m coordinate value between *mStart* and *mEnd* inclusively, then a curve value element is added to the geometry collection to represent the curve elements between these consecutive points. Any disconnected points in the 1-dimensional geometry values with m coordinate values between *mStart* and *mEnd* inclusively are also added to the geometry collection. If no matching m coordinate values are found, then an empty set of type ST_Point is returned.

For example:

a) If LocateAlong is invoked with an M value of 4 on a LineString value with well-known text representation:

LineStringM(1 0 0, 3 1 4, 5 3 4, 5 5 1, 5 6 4, 7 8 4, 9 9 0)

then the result is the following MultiLineString value with well-known text representation:

MultiLineStringM((3 1 4, 5 3 4), (5 6 4, 7 8 4))

b) If LocateBetween is invoked with an *mStart* value of 2 and an *mend* value of 4 on a LineString value with well-known text representation:

LineStringM(1 0 0, 1 1 1, 1 2 2, 3 1 3, 5 3 4, 9 5 5, 7 6 6)

then the result is the following MultiLineString value with well-known text representation:

MultiLineStringM((1 2 2, 3 1 3, 5 3 4))

c) If LocateBetween is invoked with an SM value of 6 and an EM value of 9 on a LineString value with well-known text representation:

LineStringM(1 0 0, 1 1 1, 1 2 2, 3 1 3, 5 3 4, 9 5 5, 7 6 6)

then the result is the following MultiPoint value with well-known text representation:

MultiPointM(7 6 6)

d) If LocateBetween is invoked with an SM value of 2 and an EM value of 4 on a MultiLineString value with well-known text representation:

MultiLineStringM((1 0 0, 1 1 1, 1 2 2, 3 1 3), (4 5 3, 5 3 4, 9 5 5, 7 6 6))

then the result is the following MultiLineString value with well-known text representation:

MultiLineStringM((1 2 2, 3 1 3),(4 5 3, 5 3 4))

e) If LocateBetween is invoked with an SM value of 1 and an EM value of 3 on a LineString value with well-known text representation:

LineStringM(0 0 0, 2 2 2, 4 4 4)

then the result may be the following MultiLineString value with well-known text representation:

MultiLineStringM((1 1 1, 2 2 2, 3 3 3))

f) If LocateBetween is invoked with an SM value of 7 and an EM value of 9 on a MultiLineString value with well-known text representation:

MultiLineStringM((1 0 0, 1 1 1, 1 2 2, 3 1 3), (4 5 3, 5 3 4, 9 5 5, 7 6 6)) then the result is the following MultiLineString value with well-known text representation:

PointM empty

6.1.2.6.5 Two-dimensional geometry value

The computation for 2-dimensional geometries is implementation-defined.

6.1.3 GeometryCollection

6.1.3.1 Description

A GeometryCollection is a geometric object that is a collection of some number of geometric objects.

All the elements in a GeometryCollection shall be in the same Spatial Reference System. This is also the Spatial Reference System for the GeometryCollection.

GeometryCollection places no other constraints on its elements. Subclasses of GeometryCollection may restrict membership based on dimension and may also place other constraints on the degree of spatial overlap between elements.

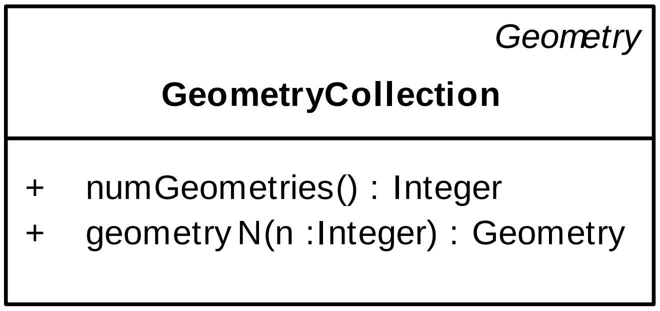

Geometry
GeometryCollection
+ numGeometries() : Integer
+ geometryN(n :Integer) : Geometry

Figure 3: Geometry collection operations

6.1.3.2 Methods

By the nature of digital representations, collections are inherently ordered by the underlying storage mechanism. Two collections whose difference is only this order are spatially equal and will return equivalent results in any geometric-defined operations.

— **NumGeometries** ( ): Integer — Returns the number of geometries in *this* GeometryCollection.

— **GeometryN** (N: integer): Geometry — Returns the Nth geometry in *this* GeometryCollection.

6.1.4 Point

6.1.4.1 Description

A Point is a 0-dimensional geometric object and represents a single location in coordinate space. A Point has an *x*-coordinate value, a *y*-coordinate value. If called for by the associated Spatial Reference System, it may also have coordinate values for *z* and *m*.

The boundary of a Point is the empty set.

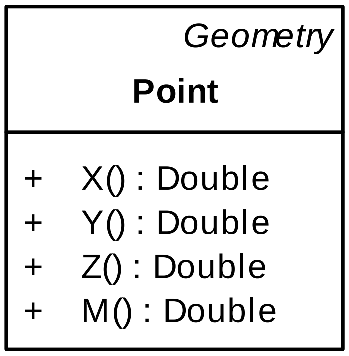


Figure 4: Point

6.1.4.2 Methods

— **X** ( ):Double — The *x*-coordinate value for *this* Point.

— **Y** ( ):Double — The *y*-coordinate value for *this* Point.

— **Z** ( ):Double — The *z*-coordinate value for *this* Point, if it has one. Returns NIL otherwise.

— **M** ( ):Double — The *m*-coordinate value for *this* Point, if it has one. Returns NIL otherwise.

6.1.5 MultiPoint

A MultiPoint is a 0-dimensional GeometryCollection. The elements of a MultiPoint are restricted to Points. The Points are not connected or ordered in any semantically important way (see the discussion at GeometryCollection).

A MultiPoint is simple if no two Points in the MultiPoint are equal (have identical coordinate values in X and Y). Every MultiPoint is spatially equal under the definition in Clause 6.1.15.3 to a simple Multipoint.

The boundary of a MultiPoint is the empty set.

6.1.6 Curve

6.1.6.1 Description

A Curve is a 1-dimensional geometric object usually stored as a sequence of Points, with the subtype of Curve specifying the form of the interpolation between Points. This standard defines only one subclass of Curve, LineString, which uses linear interpolation between Points.

A Curve is a 1-dimensional geometric object that is the homeomorphic image of a real, closed, interval:

$D = [a, b] = \{t \in \Re \mid a \leq t \leq b\}$

under a mapping

$f : [a, b] \rightarrow \Re^n$

where n is the coordinate dimension of the underlying Spatial Reference System.

A Curve is simple if it does not pass through the same Point twice with the possible exception of the two end points (Reference [1], section 3.12.7.3):

$$\forall\ c \in \text{Curve},\ [a, b] = c.\text{Domain},\ c =: f : [a, b] \rightarrow \Re^n$$
$$c.\text{IsSimple} \Leftrightarrow \forall\ x_1, x_2 \in [a, b]: [\ f(x_1) = f(x_2) \wedge x_1 < x_2] \Rightarrow [x_1 = a \wedge x_2 = b]$$

A Curve is closed if its start Point is equal to its end Point (Reference [1], section 3.12.7.3).

$$c.\text{IsClosed} \Leftrightarrow [f(a) = f(b)]$$

The boundary of a closed Curve is empty.

$$c.\text{IsClosed} \Leftrightarrow [c.\text{boundary} = \varnothing]$$

A Curve that is simple and closed is a Ring.

The boundary of a non-closed Curve consists of its two end Points (Reference [1], section 3.12.3.2).

A Curve is defined as topologically closed, that is, it contains its endpoints f(a) and f(b).

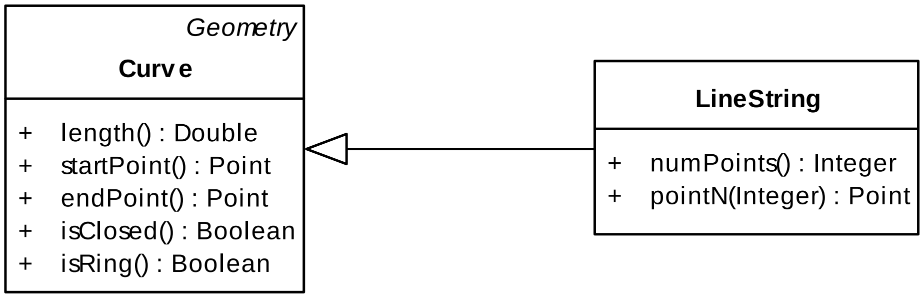


Figure 5: Curve

6.1.6.2 Methods

— **Length** ( ): Double — The length of *this* Curve in its associated spatial reference.

— **StartPoint** ( ): Point — The start Point of *this* Curve.

— **EndPoint** ( ): Point — The end Point of *this* Curve.

— **IsClosed** ( ): Integer — Returns 1 (TRUE) if *this* Curve is closed [StartPoint ( ) = EndPoint ( )].

— **IsRing** ( ): Integer — Returns 1 (TRUE) if *this* Curve is closed [StartPoint ( ) = EndPoint ( )] and *this* Curve is simple (does not pass through the same Point more than once).

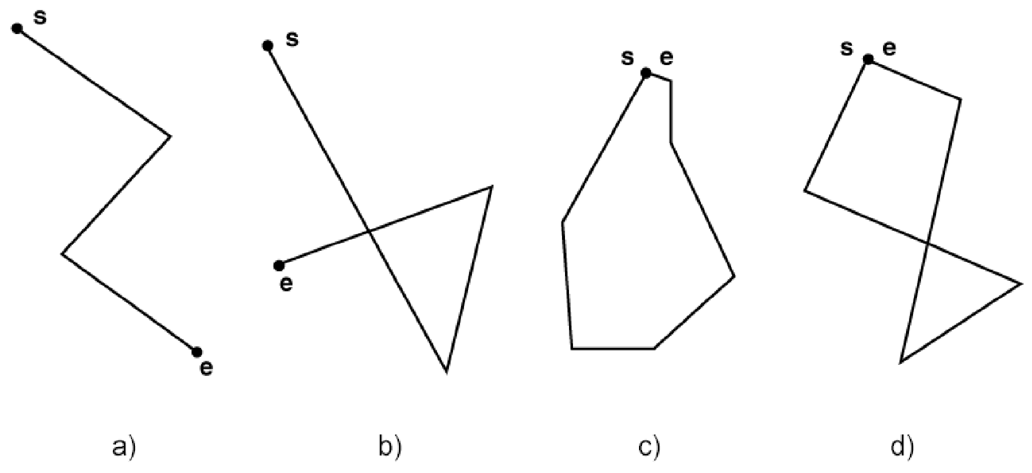
6.1.7 LineString, Line, LinearRing

6.1.7.1 Description

A LineString is a Curve with linear interpolation between Points. Each consecutive pair of Points defines a Line segment.

A Line is a LineString with exactly 2 Points.

A LinearRing is a LineString that is both closed and simple. The Curve in Figure 2, item (c), is a closed LineString that is a LinearRing. The Curve in Figure 2, item (d) is a closed LineString that is not a LinearRing.



Key

s	start
e	end

Figure 6: Examples of LineStrings
Simple LineString (a),
Non-simple LineString (b),
Simple, closed LineString (a LinearRing) (c),
Non-simple closed LineString (d)

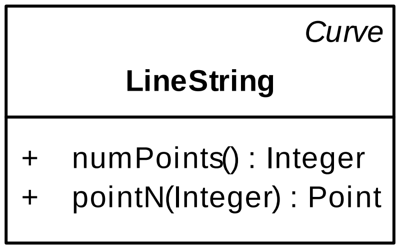


Figure 7: LineString

6.1.7.2 Methods

— **NumPoints** ( ): Integer — The number of Points in *this* LineString.

— **PointN** (N: Integer): Point — Returns the specified Point N in *this* LineString.

6.1.8 MultiCurve

6.1.8.1 Description

A MultiCurve is a 1-dimensional GeometryCollection whose elements are Curves as in Figure 3.

MultiCurve is a non-instantiable class in this standard; it defines a set of methods for its subclasses and is included for reasons of extensibility.

A MultiCurve is simple if and only if all of its elements are simple and the only intersections between any two elements occur at Points that are on the boundaries of both elements.

The boundary of a MultiCurve is obtained by applying the "mod 2" union rule: A Point is in the boundary of a MultiCurve if it is in the boundaries of an odd number of elements of the MultiCurve (Reference [1], section 3.12.3.2).

A MultiCurve is closed if all of its elements are closed. The boundary of a closed MultiCurve is always empty.

A MultiCurve is defined as topologically closed.

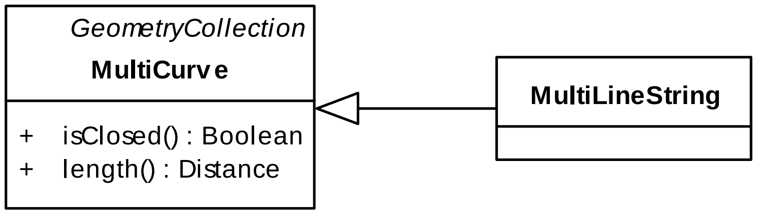


Figure 8: MultiCurve

6.1.8.2 Methods

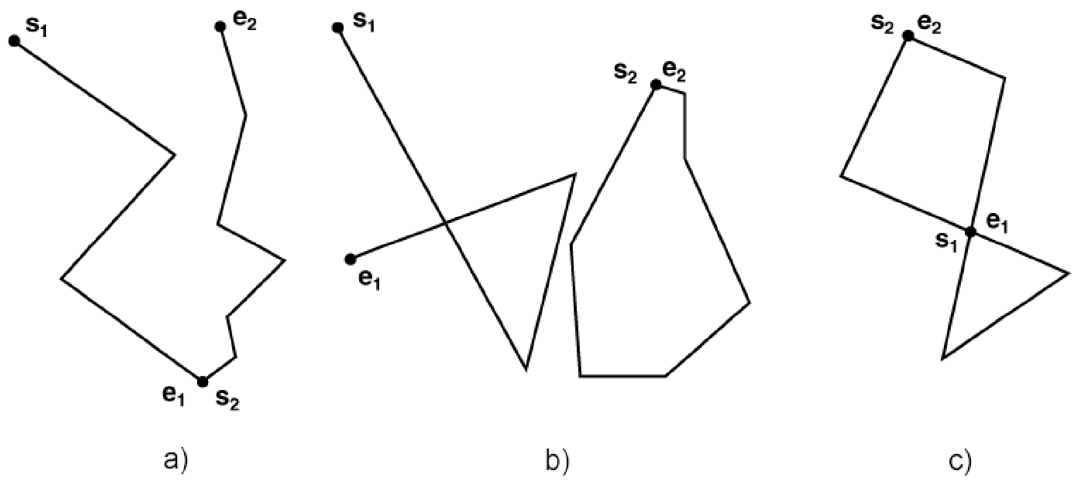
— **IsClosed** ( ): Integer — Returns 1 (TRUE) if *this* MultiCurve is closed [StartPoint ( ) = EndPoint ( ) for each Curve in *this* MultiCurve].

— **Length** ( ): Double — The Length of *this* MultiCurve which is equal to the sum of the lengths of the element Curves.

6.1.9 MultiLineString

A MultiLineString is a MultiCurve whose elements are LineStrings.

The boundaries for the MultiLineStrings in Figure 9 are (a)—{s1, e2}, (b)— {s1, e1}, (c)—∅.



Key

s	start
e	end

Figure 9: Examples of MultiLineStrings

Note: The diagram above contains: Simple MultiLineString (a), Non-simple MultiLineString with 2 elements (b), Non-simple, closed MultiLineString with 2 elements (c)

6.1.10 Surface

6.1.10.1 Description

A Surface is a 2-dimensional geometric object.

A simple Surface may consists of a single “patch” that is associated with one “exterior boundary” and 0 or more “interior” boundaries. A single such Surface patch in 3-dimensional space is isometric to planar Surfaces, by a simple affine rotation matrix that rotates the patch onto the plane z = 0. If the patch is not vertical, the projection onto the same plane is an isomorphism, and can be represented as a linear transformation, i.e. an affine.

Polyhedral Surfaces are formed by “stitching” together such simple Surfaces patches along their common boundaries. Such polyhedral Surfaces in a 3-dimensional space may not be planar as a whole, depending on the orientation of their planar normals (Reference [1], sections 3.12.9.1, and 3.12.9.3). If all the patches are in alignment (their normals are parallel), then the whole stitched polyhedral surface is co-planar and can be represented as a single patch if it is connected.

The boundary of a simple Surface is the set of closed Curves corresponding to its “exterior” and “interior” boundaries (Reference [1], section 3.12.9.4).

The only instantiable subclasses of Surface defined in this standard are Polygon and PolyhedralSurface. A Polygon is a simple Surface that is planar. A PolyhedralSurface is a simple surface, consisting of some number of Polygon patches or facets. If a PolyhedralSurface is closed, then it bounds a solid. A MultiSurface containing a set of closed PolyhedralSurfaces can be used to represent a Solid object with holes.

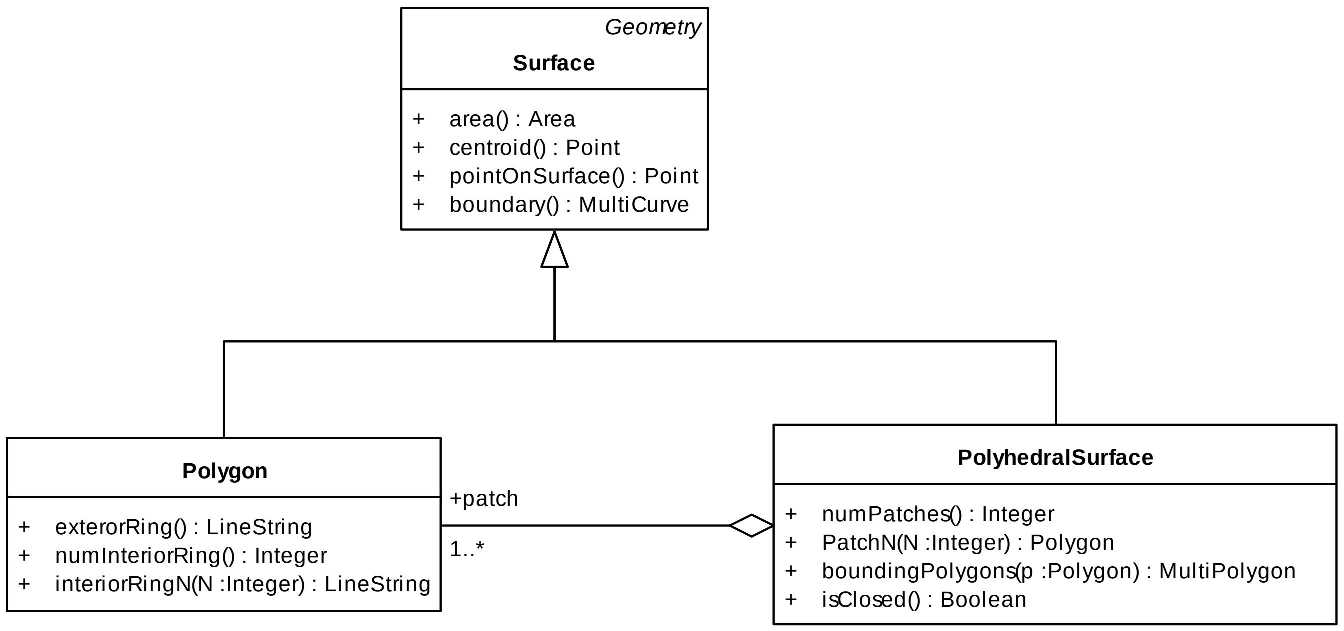

Geometry
Surface
+ area() : Area
+ centroid() : Point
+ pointOnSurface() : Point
+ boundary() : MultiCurve
Polygon
+ exterorRing() : LineString
+ numInteriorRing() : Integer
+ interiorRingN(N :Integer) : LineString
+patch
1..*
PolyhedralSurface
+ numPatches() : Integer
+ PatchN(N :Integer) : Polygon
+ boundingPolygons(p :Polygon) : MultiPolygon
+ isClosed() : Boolean

Figure 10: Surface

6.1.10.2 Methods

— **Area** ( ): Double — The area of *this* Surface, as measured in the spatial reference system of *this* Surface.

— **Centroid** ( ): Point — The mathematical centroid for *this* Surface as a Point. The result is not guaranteed to be on *this* Surface.

— **PointOnSurface** ( ): Point — A Point guaranteed to be on *this* Surface.

6.1.11 Polygon, Triangle

6.1.11.1 Description

A Polygon is a planar Surface defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the Polygon. A Triangle is a polygon with 3 distinct, non-collinear vertices and no interior boundary.

The exterior boundary LinearRing defines the “top” of the surface which is the side of the surface from which the exterior boundary appears to traverse the boundary in a counter clockwise direction. The interior LinearRings will have the opposite orientation, and appear as clockwise when viewed from the “top”,

The assertions for Polygons (the rules that define valid Polygons) are as follows:

a) Polygons are topologically closed;

b) The boundary of a Polygon consists of a set of LinearRings that make up its exterior and interior boundaries;

c) No two Rings in the boundary cross and the Rings in the boundary of a Polygon may intersect at a Point but only as a tangent, e.g.

```
∀ P ∈ Polygon, ∀ c1,c2∈P.Boundary(), c1≠c2,
∀ p, q ∈Point, p, q ∈ c1, p ≠ q ,
[p ∈ c2] ⇒ [∃ δ > 0 ∍ [|p-q|<δ] ⇒ [q ∉ c2] ];
```

Note: This last condition says that at a point common to the two curves, nearby points cannot be common. This forces each common point to be a point of tangency.

d) A Polygon may not have cut lines, spikes or punctures e.g.:

```
∀ P ∈ Polygon, P = P.Interior.Closure;
```

e) The interior of every Polygon is a connected point set;

f) The exterior of a Polygon with 1 or more holes is not connected. Each hole defines a connected component of the exterior.

In the above assertions, interior, closure and exterior have the standard topological definitions. The combination of (a) and (c) makes a Polygon a regular closed Point set. Polygons are simple geometric objects. Figure 11 shows some examples of Polygons.

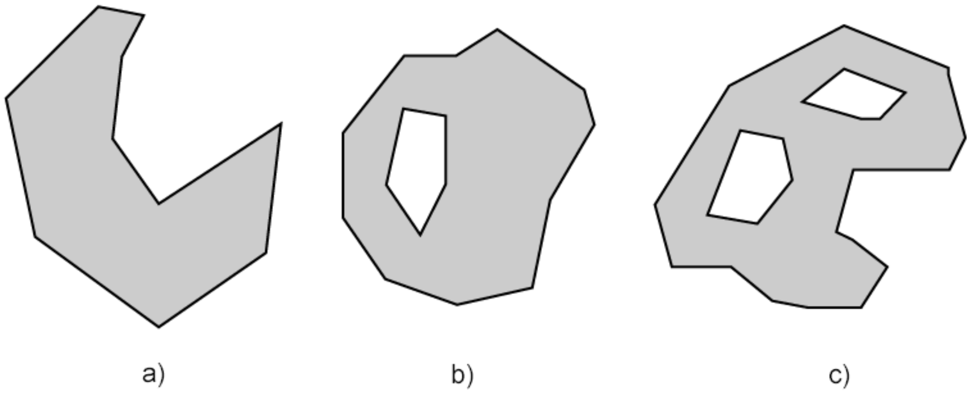


Figure 11: Examples of Polygons with 1 (a), 2 (b) and 3 (c) Rings, respectively

Figure 12 shows some examples of geometric objects that violate the above assertions and are not representable as single instances of Polygon.

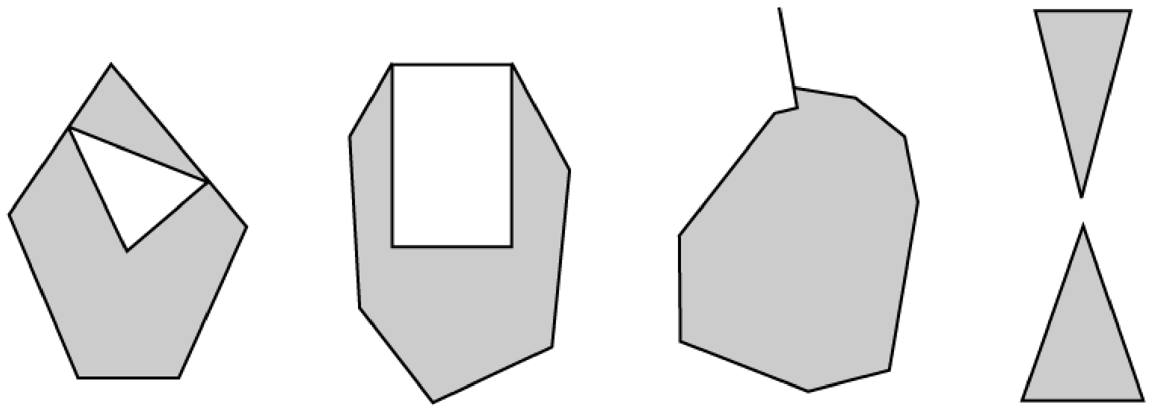


Figure 12: Examples of objects not representable as a single instance of Polygon

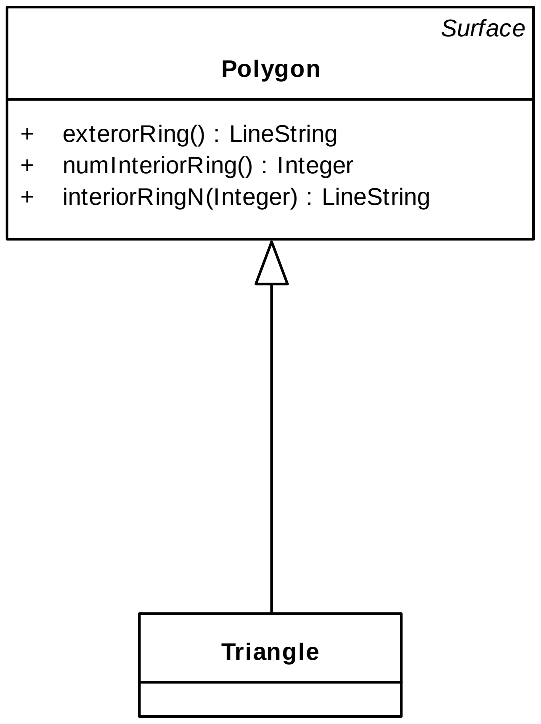

Surface
Polygon
+ exterorRing() : LineString
+ numInteriorRing() : Integer
+ interiorRingN(Integer) : LineString
Triangle

Figure 13: Polygon

6.1.11.2 Methods

— **ExteriorRing** ( ): LineString — Returns the exterior ring of *this* Polygon.

— **NumInteriorRing** ( ): Integer — Returns the number of interior rings in *this* Polygon.

— **InteriorRingN** (N: Integer): LineString — Returns the Nth interior ring for *this* Polygon as a LineString.

6.1.12 PolyhedralSurface

6.1.12.1 Description

A PolyhedralSurface is a contiguous collection of polygons, which share common boundary segments. For each pair of polygons that “touch”, the common boundary shall be expressible as a finite collection of LineStrings. Each such LineString shall be part of the boundary of at most 2 Polygon patches. A TIN (triangulated irregular network) is a PolyhedralSurface consisting only of Triangle patches.

For any two polygons that share a common boundary, the “top” of the polygon shall be consistent. This means that when two LinearRings from these two Polygons traverse the common boundary segment, they do so in opposite directions. Since the Polyhedral surface is contiguous, all polygons will be thus consistently oriented. This means that a non-oriented surface (such as Möbius band) shall not have single surface representations. They may be represented by a MultiSurface. Figure 14 shows an example of such a consistently oriented surface (from the top). The arrows indicate the ordering of the linear rings that from the boundary of the polygon in which they are located.

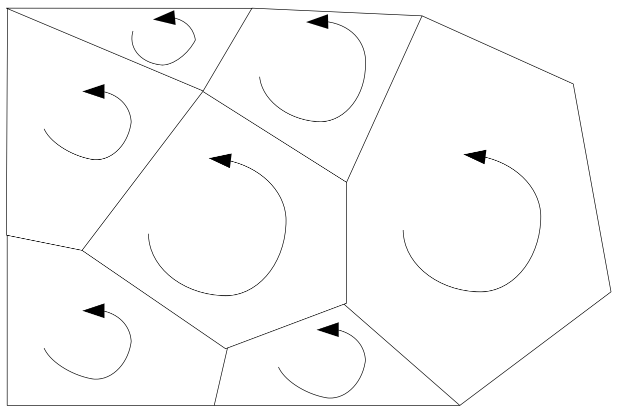


Figure 14: Polyhedral Surface with consistent orientation

If each such LineString is the boundary of exactly 2 Polygon patches, then the PolyhedralSurface is a simple, closed polyhedron and is topologically isomorphic to the surface of a sphere. By the Jordan Surface Theorem (Jordan’s Theorem for 2-spheres), such polyhedrons enclose a solid topologically isomorphic to the interior of a sphere; the ball. In this case, the “top” of the surface will either point inward or outward of the enclosed finite solid. If outward, the surface is the exterior boundary of the enclosed surface. If inward, the surface is the interior of the infinite complement of the enclosed solid. A Ball with some number of voids (holes) inside can thus be presented as one exterior boundary shell, and some number in interior boundary shells.

6.1.12.2 Methods

— **NumPatches ()** : **Integer** — Returns the number of including polygons

— **PatchN (N: Integer): Polygon** — Returns a polygon in this surface, the order is arbitrary.

— **BoundingPolygons (p: Polygon): MultiPolygon** — Returns the collection of polygons in this surface that bounds the given polygon “p” for any polygon “p” in the surface.

— **IsClosed (): Integer** — Returns 1 (True) if the polygon closes on itself, and thus has no boundary and encloses a solid

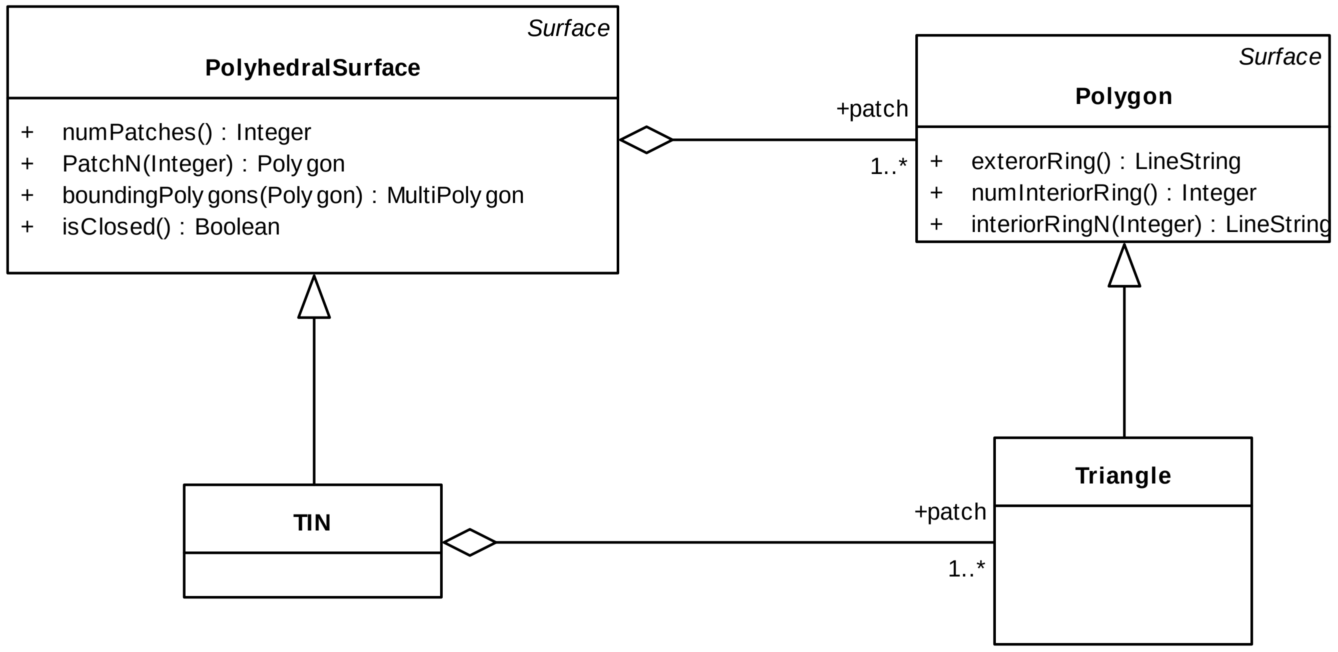

Surface
PolyhedralSurface
+ numPatches() : Integer
+ PatchN(Integer) : Polygon
+ boundingPolygons(Polygon) : MultiPolygon
+ isClosed() : Boolean
+patch
1..*
Surface
Polygon
+ exterorRing() : LineString
+ numInteriorRing() : Integer
+ interiorRingN(Integer) : LineString
TIN
+patch
1..*
Triangle

Figure 15: Polyhedral Surface

6.1.13 MultiSurface

6.1.13.1 Description

A MultiSurface is a 2-dimensional GeometryCollection whose elements are Surfaces, all using coordinates from the same coordinate reference system. The geometric interiors of any two Surfaces in a MultiSurface may not intersect in the full coordinate system. The boundaries of any two coplanar elements in a MultiSurface may intersect, at most, at a finite number of Points. If they were to meet along a curve, they could be merged into a single surface.

MultiSurface is an instantiable class in this Standard, and may be used to represent heterogeneous surfaces collections of polygons and polyhedral surfaces. It defines a set of methods for its subclasses. The subclass of MultiSurface is MultiPolygon corresponding to a collection of Polygons only. Other collections shall use MultiSurface.

NOTE: The geometric relationships and sets are the common geometric ones in the full coordinate systems. The use of the 2D map operations defined Clause 6.1.15 may classify the elements of a valid 3D MultiSurface as having overlapping interiors in their 2D projections.

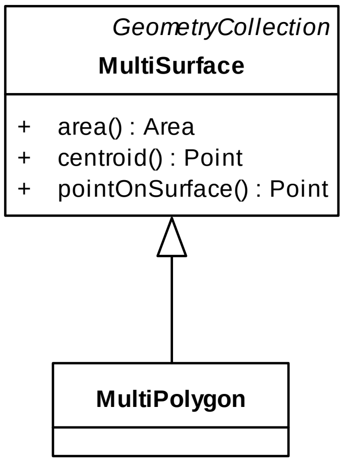


Figure 16: MultiSurface operations

6.1.13.2 Methods

MultiSurface inherits operations NumGeometries and GeometryN from GeometryCollection to access its individual component surfaces.

— **Area ( ): Double** — The area of *this* MultiSurface, as measured in the spatial reference system of *this* MultiSurface.

— **Centroid ( ): Point** — The mathematical centroid for *this* MultiSurface. The result is not guaranteed to be on *this* MultiSurface.

— **PointOnSurface ( ): Point** — A Point guaranteed to be on *this* MultiSurface.

6.1.14 MultiPolygon

A MultiPolygon is a MultiSurface whose elements are Polygons.

The assertions for MultiPolygons are as follows.

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

$$\forall M \in \text{MultiPolygon},\ \forall P_i, P_j \in M.\text{Geometries}(), i \neq j,\ \text{Interior}(P_i) \cap \text{Interior}(P_j) = \varnothing;$$

b) The boundaries of any 2 Polygons that are elements of a MultiPolygon may not “cross” and may touch at only a finite number of Points.

$$\forall M \in \text{MultiPolygon},\ \forall P_i, P_j \in M.\text{Geometries}(),\ \forall c_i, c_j \in \text{Curve}\ c_i \in P_i.\text{Boundaries}(),\ c_j \in P_j.\text{Boundaries}()\ \exists k \in \text{Integer} \ni c_i \cap c_j = \{p_1, \ldots, p_k \mid p_m \in \text{Point},\ 0<m<k\};$$

NOTE Crossing is prevented by assertion (a) above.

c) A MultiPolygon is defined as topologically closed.

d) A MultiPolygon may not have cut lines, spikes or punctures, a MultiPolygon is a regular closed Point set:

```
∀ M ∈ MultiPolygon, M = Closure(Interior(M))
```

e) The interior of a MultiPolygon with more than 1 Polygon is not connected; the number of connected components of the interior of a MultiPolygon is equal to the number of Polygons in the MultiPolygon.

The boundary of a MultiPolygon is a set of closed Curves (LineStrings) corresponding to the boundaries of its element Polygons. Each Curve in the boundary of the MultiPolygon is in the boundary of exactly 1 element Polygon, and every Curve in the boundary of an element Polygon is in the boundary of the MultiPolygon.

The reader is referred to works by Worboys et al.([13], [14]) and Clementini et al. ([5], [6]) for the definition and specification of MultiPolygons.

Figure 17 shows four examples of valid MultiPolygons with 1, 3, 2 and 2 Polygon elements, respectively.

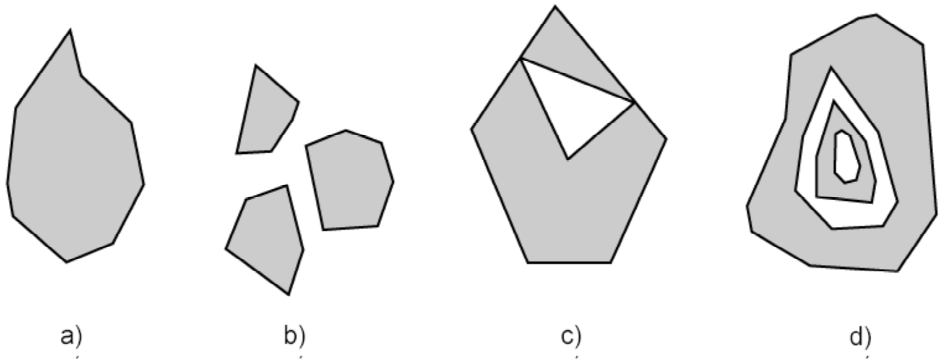


Figure 17: Examples of MultiPolygons with 1 (a), 3 (b) , 2 (c) and 2 (d) Polygon elements

Figure 18 shows examples of geometric objects not representable as single instances of MultiPolygons.

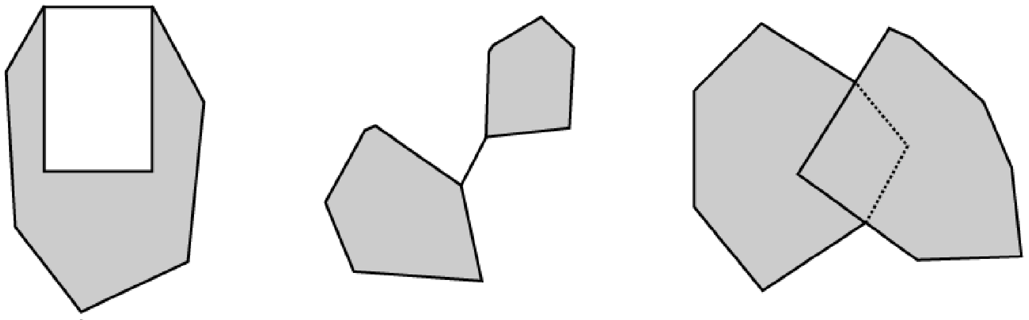


Figure 18: Geometric objects not representable as a single instance of a MultiPolygon

NOTE The subclass of Surface named Polyhedral Surface as described in Reference [1], is a faceted Surface whose facets are Polygons. A Polyhedral Surface is not a MultiPolygon because it violates the rule for MultiPolygons that the boundaries of the element Polygons intersect only at a finite number of Points.

6.1.15 Relational operators

6.1.15.1 Background

The relational operators are Boolean methods that are used to test for the existence of a specified topological spatial relationship between two geometric objects as they would be represented on a map. Topological spatial relationships between two geometric objects have been a topic of extensive study; see References in the Bibliography numbered [4], [5], [6], [7], [8], [9], and [10]. The basic approach to comparing two geometric objects is to project the objects onto the 2D horizontal coordinate reference system representing the Earth's surface, and then to make pair-wise tests of the intersections between the interiors, boundaries and exteriors of the two projections and to classify the map relationship between the two geometric objects based on the entries in the resulting 3 by 3 'intersection' matrix. The concepts of interior, boundary and exterior are well defined as sets of point geometry, and abstracted in general topology; see Reference [4].

It is important to note that the calculation of the following operations will give equivalent results whether the calculations are done using classical geometric representations or these same calculations are done with algebraic techniques in a well-structured and properly defined equivalent topological structure.

These concepts are applied in this standard for defining spatial relationships between 2-dimensional objects in 2-dimensional space ($\Re^2$) by the projection of the objects onto the horizontal surface usually represented in a map. This will give a different result than would be obtained if the full 3D geometry (or its corresponding 3D topology) because of the changes induced in the projection of the objects onto the horizontal map projection. It would be possible to define a full 3D set of operations, but the increase in computational complexity can be prohibitive to most implementations, and is generally not supported in many geographic information systems or other applications dealing with significant volumes of "mapping data." Specification of full 3D operators following this same pattern for higher dimensions is reserved for a future version of this standard.

NOTE It is important to remember that when reading this standard, that when spoken of in the abstract the relationship underlying these operations will refer to the full relationship in the coordinate reference system of the objects being spoken of, unless the operations defined in these clauses is specifically referenced.

In order to apply the concepts of interior, boundary and exterior to 1- and 0-dimensional objects in $\Re^2$, a combinatorial topology approach shall be applied (Reference [1], section 3.12.3.2). This approach is based on the accepted definitions of the boundaries, interiors and exteriors for simplicial complexes (see Reference [12]) and yields the following results.

The boundary of a geometric object is a set of geometric objects of the next lower dimension. The boundary of a Point or a MultiPoint is the empty set. The boundary of a non-closed Curve consists of its two end Points; the boundary of a closed Curve is empty. The boundary of a MultiCurve consists of those Points that are in the boundaries of an odd number of its element Curves. The boundary of a Polygon consists of its set of Rings. The boundary of a MultiPolygon consists of the set of Rings of its Polygons. The boundary of an arbitrary collection of geometric objects whose interiors are disjoint consists of geometric objects drawn from the boundaries of the element geometric objects by application of the "mod 2" union rule (Bibliographic Reference [1], section 3.12.3.2).

The domain of geometric objects considered is those that are topologically closed. The interior of a geometric object consists of those Points that are left when the boundary Points are removed. The exterior of a geometric object consists of Points not in the interior or boundary.

Studies on the relationships between two geometric objects both of maximal dimension in $\Re^1$ and $\Re^2$ considered pair-wise intersections between the interior and boundary sets and led to the definition of a four-intersection model; see Reference [8]. The model was extended to consider the exterior of the input geometric objects, resulting in a nine-intersection model (see Reference [11]) and further extended to include information on the dimension of the results of the pair-wise intersections resulting in a dimensionally extended nine-intersection model; see Reference [5]. These extensions allow the model to express spatial relationships between points, lines and areas, including areas with holes and multi-component lines and areas; see Reference [6].

6.1.15.2 The Dimensionally Extended Nine-Intersection Model (DE-9IM)

Given a geometric object *a*, let *I(a)*, *B(a)* and *E(a)* represent the interior, boundary and exterior of "a", respectively.

Let *dim(x)* return the maximum dimension (-1, 0, 1, or 2) of the geometric objects in *x*, with a numeric value of -1 corresponding to $dim(\emptyset)$.

The intersection of any two of *I(a)*, *B(a)* and *E(a)* can result in a set of geometric objects, *x*, of mixed dimension. For example, the intersection of the boundaries of two Polygons may consist of a point and a line.

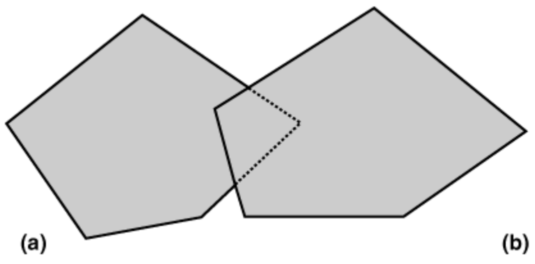
Table 1 shows the general form of the dimensionally extended nine-intersection matrix (DE-9IM).

Table 1: The DE-9IM

	Interior	**Boundary**	**Exterior**
Interior	$dim(I(a) \cap I(b))$	$dim(I(a) \cap B(b))$	$dim(I(a) \cap E(b))$
Boundary	$dim(B(a) \cap I(b))$	$dim(B(a) \cap B(b))$	$dim(B(a) \cap E(b))$
Exterior	$dim(E(a) \cap I(b))$	$dim(E(a) \cap B(b))$	$dim(E(a) \cap E(b))$

For regular, topologically closed input geometric objects, computing the dimension of the intersection of the interior, boundary and exterior sets does not have, as a prerequisite, the explicit computation and representation of these sets. To compute if the interiors of two regular closed Polygons intersect, and to ascertain the dimension of this intersection, it is not necessary to explicitly represent the interior of the two Polygons, which are topologically open sets, as separate geometric objects. In most cases, the dimension of the intersection value at a cell is highly constrained, given the type of the two geometric objects. In the Line-Area case, the only possible values for the interior-interior cell are drawn from {-1, 1} and in the Area-Area case, the only possible values for the interior-interior cell are drawn from {-1, 2}. In such cases, no work beyond detecting the intersection is required.

Figure 8 shows an example DE-9IM for the case where "a" and "b" are two Polygons that overlap.



	Interior	Boundary	Exterior
Interior	2	1	2
Boundary	1	0	1
Exterior	2	1	2

Figure 19: An example instance and its DE-9IM

On two geometric objects, a spatial relationship predicate can be expressed as a formula that takes as input a pattern matrix representing the set of acceptable values for the DE-9IM for the two geometric objects. If the spatial relationship between the two geometric objects corresponds to one of the acceptable values as represented by the pattern matrix, then the predicate returns TRUE.

The pattern matrix consists of a set of nine pattern-values, one for each cell in the matrix. The possible pattern-values of *p* are {T, F, *, 0, 1, 2} and their meanings for any cell where x is the intersection set for the cell are as follows:

```
p = T ⇒ dim(x) ∈ {0, 1, 2}, i.e. x ≠ ∅
p = F ⇒ dim(x) = -1, i.e. x = ∅
p = * ⇒ dim(x) ∈ {-1, 0, 1, 2}, i.e. Don't Care
p = 0 ⇒ dim(x) = 0
p = 1 ⇒ dim(x) = 1
p = 2 ⇒ dim(x) = 2
```

The pattern matrix can be represented as an array or list of nine characters in row major order. As an example, the following code fragment could be used to test for “Overlap” between two areas:

```
char * overlapMatrix = "T*T***T**";
Geometry* a, b;
Boolean b = a->Relate(b, overlapMatrix);
```

6.1.15.3 Named spatial relationship predicates based on the DE-9IM

The Relate predicate based on the pattern matrix has the advantage that clients can test for a large number of spatial relationships and fine tune the particular relationship being tested. It has the disadvantage that it is a lower-level building block and does not have a corresponding natural language equivalent. Users of the proposed system include IT developers using the COM API from a language such as Visual Basic, and interactive SQL users who may wish, for example, to select all features ‘spatially within’ a query Polygon, in addition to more spatially “sophisticated” GIS developers.

To address the needs of such users, a set of named spatial relationship predicates has been defined for the DE-9IM; see References [5, 6]. The five predicates are named Disjoint, Touches, Crosses, Within and Overlaps. The definition of these predicates (see References [5, 6]) is given below. In these definitions, the term P is used to refer to 0-dimensional geometries (Points and MultiPoints), L is used to refer to 1-dimensional geometries (LineStrings and MultiLineStrings) and A is used to refer to 2-dimensional geometries (Polygons and MultiPolygons).

Equals

Given two (topologically closed) geometric objects “a” and “b”:

```
a.Equals(b) ⇔ a ⊆ b ∧ b ⊆ a
```

Expressed in terms of the DE-9IM:

```
a.Equals(b)  ⇔ [  (I(a) ∩ I(b) ≠ ∅) ∧
```

```
                  (I(a) ∩ B(b) = ∅) ∧
                  (I(a) ∩ E(b) = ∅) ∧
                  (B(a) ∩ I(b) = ∅) ∧
                  (B(a) ∩ B(b) ≠ ∅) ∧
                  (B(a) ∩ E(b) = ∅) ∧
                  (E(a) ∩ I(b) = ∅) ^
                  (E(a) ∩ B(b) = ∅) ^
                  (E(a) ∩ E(b) ≠ ∅) ]
             ⇔  a.Relate(b, "TFFFTFFFT")
```

Disjoint

Given two (topologically closed) geometric objects "a" and "b":

```
a.Disjoint(b) ⇔ a ∩ b = ∅
```

Expressed in terms of the DE-9IM:

```
a.Disjoint(b) ⇔ [  (I(a) ∩ I(b) = ∅) ∧
                   (I(a) ∩ B(b) = ∅) ∧
                   (B(a) ∩ I(b) = ∅) ∧
                   (B(a) ∩ B(b) = ∅) ]
             ⇔  a.Relate(b, "FF*FF****")
```

Touches

The Touches relationship between two geometric objects "a" and "b" applies to the A/A, L/L, L/A, P/A and P/L groups of relationships but not to the P/P group. It is defined as

```
a.Touch(b) ⇔ (I(a)∩I(b)=∅)∧(a∩b)≠∅
```

Expressed in terms of the DE-9IM:

```
[a.Touch(b) ⇔ [    (I(a) ∩ I(b) = ∅) ∧
                   [  (B(a) ∩ I(b) ≠ ∅) ∨
                      (I(a) ∩ B(b) ≠ ∅) ∨
                      (B(a) ∩ B(b) ≠ ∅) ] ]
             ⇔ [  a.Relate(b, "FT*******") ∨
                  a.Relate(b, "F**T*****") ∨
                  a.Relate(b, "F***T****") ]
```

Figure 9 shows some examples of the Touches relationship.

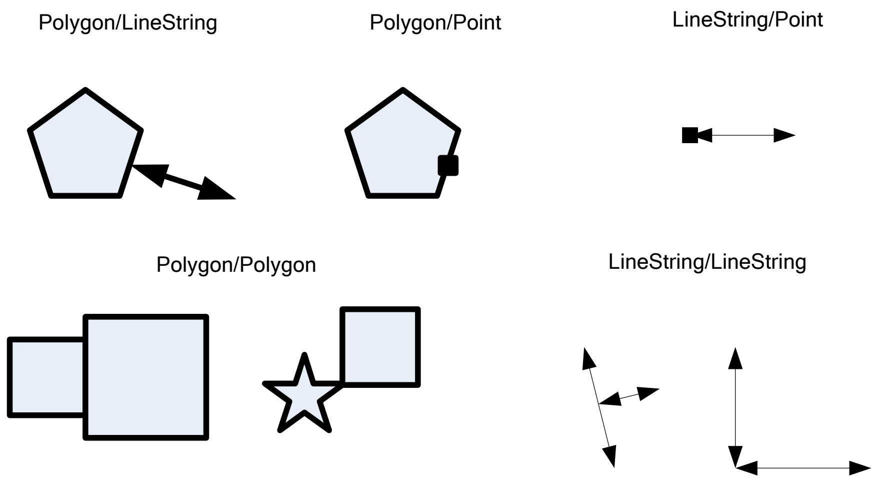


Figure 20: Examples of the Touches relationship

Crosses

The Crosses relationship applies to P/L, P/A, L/L and L/A situations. It is defined as

```
a.Cross(b) ⇔ [I(a)∩I(b)≠∅ ∧ (a ∩ b ≠a) ∧ (a ∩ b ≠b)]
```

Note: Previous definition had an unnecessary statement on dimension which was always true.

Expressed in terms of the DE-9IM:

```
Case a∈P, b∈L or
Case a∈P, b∈A or
Case a∈L, b∈A:
  a.Cross(b) ⇔ [ I(a)∩I(b)≠∅ ∧ I(a)∩E(b)≠∅ ]
                ⇔ a.Relate(b, "T*T******")

Case a∈L, b∈L:
  a.Cross(b) ⇔ dim(I(a)∩I(b)) = 0
                   ⇔ a.Relate(b, "0********");
```

Figure 10 shows some examples of the Crosses relationship.

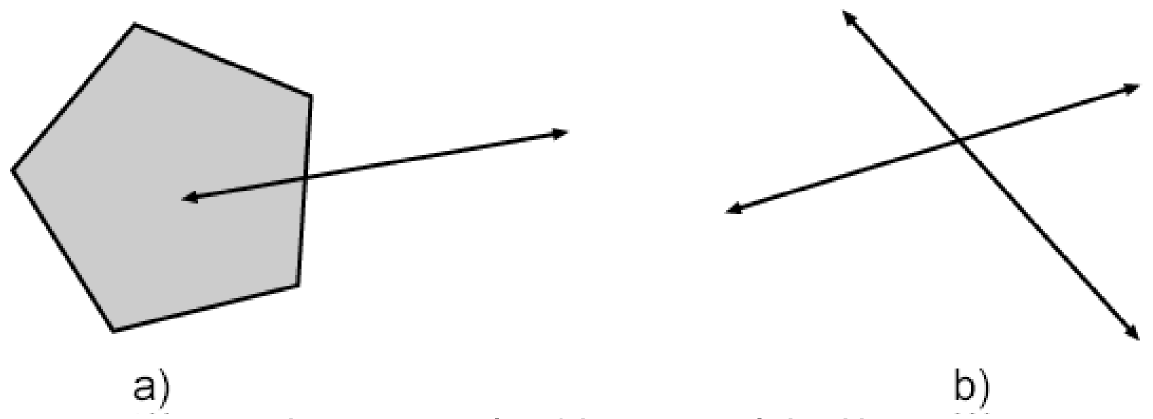


Figure 21: Examples of the Crosses relationship Polygon/LineString (a) and LineString/LineString (b)

Within

The Within relationship is defined as

```
a.Within(b) ⇔ (a∩b=a) ∧ (I(a)∩E(b)=∅)
```

Expressed in terms of the DE-9IM:

```
a.Within(b)  ⇔ [ I(a)∩I(b)≠∅ ∧ I(a)∩E(b)=∅ ∧ B(a)∩E(b)=∅ ]
 ⇔ a.Relate(b, "T*F**F***")
```

Figure 11 shows some examples of the "Within" relationship.

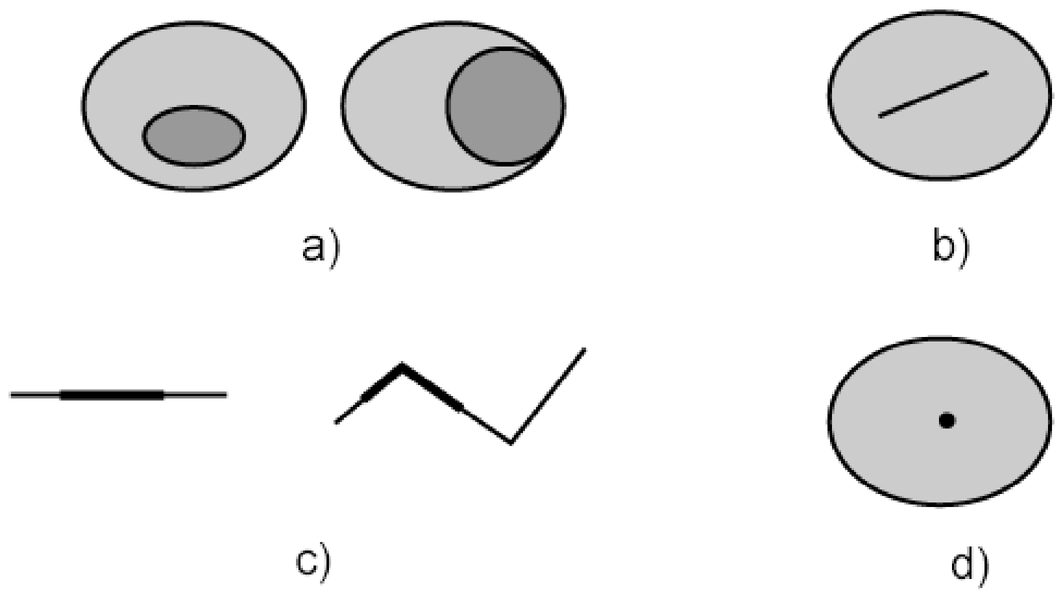


Figure 22: Examples of the "Within" relationship Polygon/Polygon (a), Polygon/LineString (b), LineString/LineString (c), and Polygon/Point (d)

Overlaps

The Overlaps relationship is defined for A/A, L/L and P/P situations.

It is defined as

```
            a.Overlaps(b) ⇔ ( dim(I(a)) = dim(I(b)) = dim(I(a) ∩ I(b)))
                                ∧ (a ∩ b ≠ a) ∧ (a ∩ b ≠ b)
```

Expressed in terms of the DE-9IM:

```
Case a ∈ P, b ∈ P or Case a ∈ A, b ∈ A:
   a.Overlaps(b)  ⇔     (I(a) ∩ I(b) ≠ ∅) ∧
                        (I(a) ∩ E(b) ≠ ∅) ∧
                        (E(a) ∩ I(b) ≠ ∅)
                 ⇔ a.Relate(b, "T*T***T**")
Case a ∈ L, b ∈ L:
   a.Overlaps(b) ⇔ (dim(I(a) ∩ I(b) = 1) ∧ (I(a) ∩ E(b) ≠ ∅) ∧ (E(a) ∩ I(b) ≠
      ∅) ⇔ a.Relate(b, "1*T***T**")
```

Figure 12 shows some examples of the Overlaps relationship.

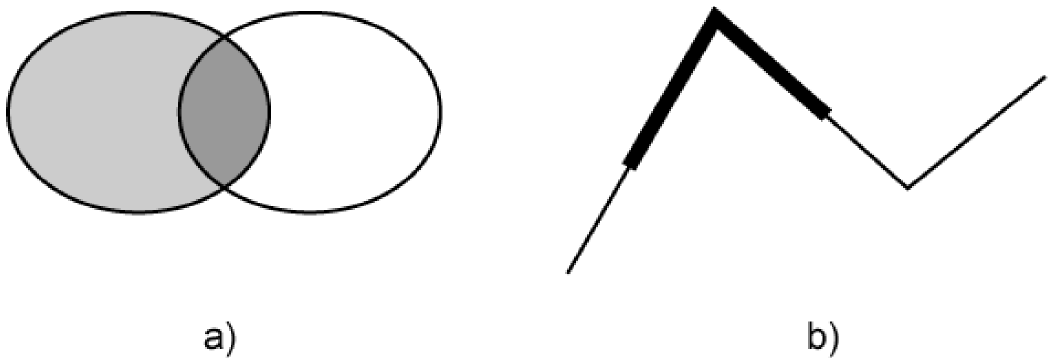


Figure 23: Examples of the Overlaps relationship Polygon/LineString (a) and LineString/LineString (b)

The following additional named predicates are also defined for user convenience:

Contains

```
     a.Contains(b) ⇔ b.Within(a)
```

Intersects

```
     a.Intersects(b) ⇔ ! a.Disjoint(b)
```

Based on the above operators the following methods are defined on Geometry:

— **Equals** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object is spatially equal to anotherGeometry.

— **Disjoint** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object is spatially disjoint'from anotherGeometry.

— **Intersects** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object spatially intersects anotherGeometry.

— **Touches** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object spatially touches anotherGeometry.

— **Crosses** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object spatially crosses'anotherGeometry.

— **Within** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object is spatially within anotherGeometry.

— **Contains** (anotherGeometry: Geometry): Integer — Returns 1 (TRUE) if *this* geometric object spatially contains anotherGeometry.

— **Overlaps** (AnotherGeometry: Geometry) Integer — Returns 1 (TRUE) if *this* geometric object spatially overlaps anotherGeometry.

— **Relate** (anotherGeometry: Geometry, intersectionPatternMatrix: String): Integer — Returns 1 (TRUE) if *this* geometric object is spatially related to anotherGeometry, by testing for intersections between the interior, boundary and exterior of the two geometric objects.

6.2 Annotation Text

Spatially placed text is a common requirement of applications. Many application have stored their text placement information in proprietary manners due lack of a consistent and usable standard. Although the mechanisms for text storage have tended to be compatible, the actual format for exchange has been sufficiently different, and, therefore, non-standardized to interfere with complete data exchange and common usage. To overcome this interoperability gap, this standard, using best engineering practices, defines an implementation of annotation text.

Annotation text is simply placed text that can carry either geographically-related or ad-hoc data and process-related information is displayable text. This text may be used for display in editors or in simpler maps. It is usually lacking in full cartographic quality, but may act as an approximation to such text as needed by any application.

The primary purpose of standardizing this concept is to enable any application using any version of Simple Features data storage or XML to read and write text objects that will describe where and how the text should be displayed. This design ensures that applications that do text placement should have no problem storing their results and that applications that comply with the standard should have no problem exchanging information on text and its placement.

Unlike spatial geometries, text display is very dependent on client text rendering engines and the style and layout attributes applied. The spatial area covered by text is only partially determined by the locating geometry. Style and layout attributes along with the actual text and locating geometry are all needed to display text correctly. Thus, it is critical to have a place to store these attributes in the feature database. While it is impossible to guarantee absolute fidelity of display on all rendering systems, applications can interoperate at a useful level.

The most common perception of text display is for cartographic purposes, for printed maps of high technical and artistic quality. While this is a potential use of placed text, its more every-day use is for identification of features in any display, regardless of the purpose of that display. So both cartographic preprint and data collection edit displays have a requirement for placed-text, albeit at different levels of artistic quality. The purpose is still the same, to aid in the understanding of the "mapped" features, either for map use or feature edit and analysis.

Text can also be used for less precise annotation purposes and more for quick display of text labels that make a display more understandable. The text so placed may not even have any associations to real-world features, but may be used to store information pertinent to the process that the data is undergoing at the moment. Thus, in a data collecting and edit display, a particular placed text may be used to indicate an error in the data that needs to be resolved, such as "sliver," "gap" and "loop" error in digitization. Here the annotation is placed near the geometric error, but is not necessarily associated to a particular feature, as much as to a portion or portions of feature geometry objects.