

DIN EN ISO 19107



ICS 35.240.70

**Geoinformation –
Raumbezugsschema (ISO 19107:2003);
Englische Fassung EN ISO 19107:2005**

Geographic information –
Spatial schema (ISO 19107:2003);
English version EN ISO 19107:2005

Information géographique –
Schéma spatial (ISO 19107:2003);
Version anglaise EN ISO 19107:2005

Gesamtumfang 178 Seiten

Normenausschuss Bauwesen (NABau) im DIN

Nationales Vorwort

Der Text von ISO 19107:2003 wurde vom Technischen Komitee ISO/TC 211 „Geoinformation/Geomatik“ erarbeitet und als EN ISO 19107:2005 vom Technischen Komitee CEN/TC 287 „Geoinformation“ übernommen, dessen Sekretariat vom NEN gehalten wird.

Der Text von ISO 19107:2003 wurde von CEN als EN ISO 19107:2005 ohne Änderungen angenommen.

Der für die deutsche Mitarbeit zuständige Arbeitsausschuss im DIN Deutsches Institut für Normung e. V. ist der als Spiegelausschuss zum CEN/TC 287 und ISO/TC 211 eingesetzte Arbeitsausschuss 03.03.00 „Kartographie und Geoinformation“ des Normenausschusses Bauwesen (NABau).

Das Präsidium des DIN hat mit seinem Beschluss 1/2004 festgelegt, dass von dem in den Regeln der europäischen Normungsarbeit von CEN/CENELEC verankerten Grundsatz, wonach Europäische Normen in den drei offiziellen Sprachen Deutsch, Englisch und Französisch veröffentlicht werden, in begründeten Ausnahmefällen abgewichen und auf die deutsche Sprachfassung verzichtet werden kann.

Die Genehmigung dafür hat die DIN-Geschäftsleitung entsprechend ihren in Anlage 1 zu dem DIN-Rundschreiben A 5/2004 festgelegten Kriterien für die vorliegende Norm auf Antrag des NABau-FB 03 „Vermessungswesen; Geoinformation“ erteilt.

Allgemeines

Geographische Informationen werden in zunehmendem Maße von einer Vielzahl von Anwendungen benutzt, um vorliegende Datenbestände über einen Raumbezug mit Positionen auf der Erde zu verknüpfen. Daraus ergibt sich ein großer Bedarf an Standardisierung dieser geographischen Informationen, der Dienste zur Verarbeitung sowie der Zugriff auf diese Informationen. Zu diesem Zweck legen die Normen der Reihe ISO 19100 die wichtigsten Aspekte zur Beschreibung und zum Management von geographischen Informationen fest.

DIN EN ISO 19107 stellt die konzeptuellen Schemata zur Beschreibung der raumbezogenen Eigenschaften von geographischen Objekten bereit. Raumbezogene Eigenschaften können in zwei Formen beschrieben werden, in der Rasterform und in der Vektorform. In dieser Norm werden nur die Aspekte der Vektorform bis hin zur 3. Dimension behandelt.

Die raumbezogenen Eigenschaften werden in dieser Norm weiterhin unterschieden in geometrische und topologische Eigenschaften, wobei die geometrischen Eigenschaften die quantitativen Beschreibungen eines Objekts an Hand von Koordinaten, mathematischen Funktionen oder anderen messbaren Größen darstellen, während die topologischen Eigenschaften diejenigen Charakteristika der Objekte beschreiben, die bei (Koordinaten-) Transformationen invariant bleiben. Hierzu gehören insbesondere die Nachbarschaftsbeziehungen sowie die Netzwerkeigenschaften.

Als dritten Aspekt – neben Geometrie und Topologie – werden in dieser Norm raumbezogene Operatoren bzw. Funktionen beschrieben, die dazu dienen, die geometrischen und topologischen Objekte zu erzeugen, zu löschen, abzufragen, zu ändern, usw.

Die konzeptuellen Schemata dieser Norm dienen dazu, die gemeinsame Nutzung und den Austausch von geographischen Informationen zwischen Anwendungen zu erleichtern. Die Norm wendet sich insbesondere an Entwickler von geographischen Informationssystemen, aber auch an alle, die ein Anwendungsschema unter Einschluss von Objektarten mit Raumbezug entwickeln.

Nationaler Anhang NA

(informativ)

Englisch-deutsche Fachwörterliste

Application	Anwendung
Application schema	Konzeptuelles Schema der Anwendung
Curve	Eindimensionales Geometrieobjekt (Linie)
Curve segment	Teil einer Linie (curve) mit gleicher Interpolationsvorschrift
Direct position	Position, die durch einen Satz Koordinaten in einem Koordinatenreferenzsystem beschrieben wird
Edge	Eindimensionales Topologieobjekt (Kante)
Face	Zweidimensionales Topologieobjekt (Masse)
Feature	Abstraktion eines realen Objekts
Geographic information	Geographische Information (Geoinformation)
Geometric primitive	Geometrieobjekt, das ein einzelnes, homogenes, verbundenes Raumbezugselement darstellt.
Node	Nulldimensionales Topologieobjekt (Knoten)
Point	Nulldimensionales Geometrieobjekt (Punkt)
Ring	Geschlossene Linie (wird i. d. R. benutzt zur Beschreibung eines Flächenumrings)
Schema	Formelle Beschreibung eines Modells
Service	Dienst
Service interface	Exakt definierte Schnittstelle zur Nutzung eines Dienstes
Shell	Satz von Flächen, die die geschlossene Hülle eines Körpers im dreidimensionalen Raum darstellen
Solid	Dreidimensionales Geometrieobjekt (Körper)
Spatial object	Objekt mit Raumbezug
Surface	Zweidimensionales Geometrieobjekt (Fläche)
Surface patch	Teil einer Fläche (surface) mit gleicher Interpolationsvorschrift
Topological primitive	Topologieobjekt, das ein einzelnes, nicht teilbares Element darstellt
Topological solid	Dreidimensionales Topologieobjekt (topologischer Körper)

— Leerseite —

English version

Geographic information - Spatial schema (ISO 19107:2003)

Information géographique - Schéma spatial (ISO
19107:2003)

Geoinformation - Raumbezugsschema

This European Standard was approved by CEN on 24 December 2004.

CEN members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration. Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the Central Secretariat or to any CEN member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CEN member into its own language and notified to the Central Secretariat has the same status as the official versions.

CEN members are the national standards bodies of Austria, Belgium, Cyprus, Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Netherlands, Norway, Poland, Portugal, Slovakia, Slovenia, Spain, Sweden, Switzerland and United Kingdom.



EUROPEAN COMMITTEE FOR STANDARDIZATION
COMITÉ EUROPÉEN DE NORMALISATION
EUROPÄISCHES KOMITEE FÜR NORMUNG

Management Centre: rue de Stassart, 36 B-1050 Brussels

Contents

Page

Foreword	7
Introduction	8
1 Scope	9
2 Conformance	9
2.1 Overview	9
2.2 Conformance classes	11
3 Normative references	12
4 Terms and definitions	12
5 Symbols, notation and abbreviated terms	22
5.1 Presentation and notation	22
5.1.1 Unified Modeling Language (UML) concepts	22
5.1.2 Attributes, operations, and associations	22
5.1.3 Stereotypes	25
5.1.4 Data types and collection types	26
5.1.5 Strong substitutability	27
5.2 Organization	28
5.3 Abbreviated terms	30
6 Geometry packages	30
6.1 Semantics	30
6.2 Geometry root package	32
6.2.1 Semantics	32
6.2.2 GM_Object	33
6.3 Geometric primitive package	40
6.3.1 Semantics	40
6.3.2 GM_Boundary	41
6.3.3 GM_ComplexBoundary	42
6.3.4 GM_PrimitiveBoundary	42
6.3.5 GM_CurveBoundary	42
6.3.6 GM_Ring	42
6.3.7 GM_SurfaceBoundary	42
6.3.8 GM_Shell	43
6.3.9 GM_SolidBoundary	43
6.3.10 GM_Primitive	43
6.3.11 GM_Point	46
6.3.12 Bearing	47
6.3.13 GM_OrientablePrimitive	48
6.3.14 GM_OrientableCurve	50
6.3.15 GM_OrientableSurface	50
6.3.16 GM_Curve	51
6.3.17 GM_Surface	52
6.3.18 GM_Solid	54
6.4 Coordinate geometry package	55
6.4.1 DirectPosition	55
6.4.2 GM_PointRef	56
6.4.3 GM_Envelope	56
6.4.4 TransfiniteSet<DirectPosition>	57
6.4.5 GM_Position	57
6.4.6 GM_PointArray, GMPointGrid	57
6.4.7 GM_GenericCurve	57
6.4.8 GM_CurveInterpolation	61

6.4.9	GM_CurveSegment	62
6.4.10	GM_LineString	63
6.4.11	GM_LineSegment	64
6.4.12	GM_GeodesicString	65
6.4.13	GM_Geodesic	66
6.4.14	GM_ArcString	66
6.4.15	GM_Arc	68
6.4.16	GM_Circle	70
6.4.17	GM_ArcStringByBulge	70
6.4.18	GM_ArcByBulge	71
6.4.19	GM_Conic	72
6.4.20	GM_Placement	74
6.4.21	GM_AffinePlacement	75
6.4.22	GM_Clothoid	75
6.4.23	GM_OffsetCurve	76
6.4.24	GM_Knot	78
6.4.25	GM_KnotType	79
6.4.26	GM_SplineCurve	79
6.4.27	GM_PolynomialSpline	79
6.4.28	GM_CubicSpline	80
6.4.29	GM_SplineCurveForm	81
6.4.30	GM_BSplineCurve	81
6.4.31	GM_Bezier	82
6.4.32	GM_SurfaceInterpolation	83
6.4.33	GM_GenericSurface	83
6.4.34	GM_SurfacePatch	85
6.4.35	GM_PolyhedralSurface	86
6.4.36	GM_Polygon	86
6.4.37	GM_TriangulatedSurface	88
6.4.38	GM_Triangle	88
6.4.39	GM_Tin	89
6.4.40	GM_ParametricCurveSurface	90
6.4.41	GM_GriddedSurface	93
6.4.42	GM_Cone	94
6.4.43	GM_Cylinder	94
6.4.44	GM_Sphere	94
6.4.45	GM_BilinearGrid	95
6.4.46	GM_BicubicGrid	95
6.4.47	GM_BSplineSurfaceForm	95
6.4.48	GM_BSplineSurface	96
6.5	Geometric aggregate package	97
6.5.7	Semantics	97
6.5.8	GM_Aggregate	97
6.5.9	GM_MultiPrimitive	97
6.5.10	GM_MultiPoint	98
6.5.11	GM_MultiCurve	99
6.5.12	GM_MultiSurface	99
6.5.13	GM_MultiSolid	99
6.6	Geometric complex package	100
6.6.7	Semantics	100
6.6.8	GM_Complex	101
6.6.9	GM_Composite	102
6.6.10	GM_CompositePoint	103
6.6.11	GM_CompositeCurve	104
6.6.12	GM_CompositeSurface	105
6.6.13	GM_CompositeSolid	105
7	Topology packages	106
7.4	Semantics	106
7.5	Topology root package	108

7.5.1	Semantics	108
7.5.2	TP_Object.....	109
7.6	Topological primitive package	113
7.6.1	Semantics	113
7.6.2	TP_Boundary.....	113
7.6.3	TP_ComplexBoundary.....	113
7.6.4	TP_PrimitiveBoundary.....	113
7.6.5	TP_EdgeBoundary	114
7.6.6	TP_FaceBoundary.....	115
7.6.7	TP_SolidBoundary	115
7.6.8	TP_Ring.....	115
7.6.9	TP_Shell	115
7.6.10	TP_Primitive	116
7.6.11	TP_DirectedTopo	117
7.6.12	TP_Node.....	120
7.6.13	TP_DirectedNode	121
7.6.14	TP_Edge.....	122
7.6.15	TP_DirectedEdge	123
7.6.16	TP_Face.....	123
7.6.17	TP_DirectedFace	125
7.6.18	TP_Solid	125
7.6.19	TP_DirectedSolid	126
7.6.20	TP_Expression	126
7.7	Topological complex package.....	129
7.7.1	Semantics	129
7.7.2	TP_Complex	129
8	Derived topological relations.....	131
8.1	Introduction	131
8.2	Boolean or set operators.....	132
8.2.1	Form of the Boolean operators	132
8.2.2	Boolean Relate	132
8.2.3	Relation to set operations	133
8.3	Egenhofer operators	133
8.3.1	Form of the Egenhofer operators.....	133
8.3.2	Egenhofer relate.....	133
8.3.3	Relation to set operations	134
8.4	Full topological operators	134
8.4.1	Form of the full topological operators	134
8.4.2	Full topological relate	134
8.5	Combinations	134
Annex A	(normative) Abstract test suite	135
A.1	Geometric primitives	135
A.2	Geometric complexes.....	138
A.3	Topological complexes	140
A.4	Topological complexes with geometric realization	142
A.5	Boolean operators	144
Annex B	(informative) Conceptual organization of terms and definitions	146
B.1	Introduction	146
B.2	General terms	146
B.3	Collections and related terms.....	147
B.4	Modelling terms.....	147
B.5	Positioning terms.....	148
B.6	Geometric terms.....	148
B.7	Topological terms	151
B.8	Relationship of geometric and topological complexes	154
Annex C	(informative) Examples of spatial schema concepts	156
C.1	Geometry.....	156

Annex D (informative) Examples for application schemata	162
D.1 Introduction.....	162
D.2 Simple Topology.....	162
D.3 Feature Topology	166
D.4 MiniTopo.....	167
Bibliography.....	173

Figures

Figure 1 — UML example association	24
Figure 2 — UML example package dependency	28
Figure 3 — Normative clause as UML package dependencies	29
Figure 4 — Geometry package: Class content and internal dependencies	31
Figure 5 — Geometry basic classes with specialization relations	32
Figure 6 — GM_Object.....	34
Figure 7 — GM_Boundary	41
Figure 8 — GM_Primitive	44
Figure 9 — GM_Point.....	46
Figure 10 — GM_OrientablePrimitive	49
Figure 11 — GM_Curve	51
Figure 12 — GM_Surface.....	53
Figure 13 — GM_Solid.....	54
Figure 14 — DirectPosition	56
Figure 15 — Curve segment classes	58
Figure 16 — Linear, arc and geodesic interpolation	64
Figure 17 — Arcs.....	67
Figure 18 — Conics and placements	73
Figure 19 — Spline and specialty curves	77
Figure 20 — Surface patches.....	84
Figure 21 — Polygonal surface	87
Figure 22 — TIN construction	89
Figure 23 — GM_ParametricCurveSurface and its subtypes	91
Figure 24 — GM_Aggregate	98
Figure 25 — GM_Complex.....	102
Figure 26 — GM_Composite.....	103
Figure 27 — GM_CompositePoint	104
Figure 28 — GM_CompositeCurve	104
Figure 29 — GM_CompositeSurface	105
Figure 30 — GM_CompositeSolid.....	106
Figure 31 — Topology packages, class content and internal dependencies.....	107
Figure 32 — Topological class diagram	108
Figure 33 — Relation between geometry and topology	109
Figure 34 — TP_Object.....	110
Figure 35 — Boundary and coboundary operation represented as associations	111
Figure 36 — Important classes in topology.....	112
Figure 37 — Boundary relation data types.....	114
Figure 38 — TP_Primitive	116
Figure 39 — TP_DirectedTopo subclasses.....	118
Figure 40 — TP_DirectedTopo	118

Figure 41 — TP_Node	121
Figure 42 — TP_Edge	122
Figure 43 — TP_Face	124
Figure 44 — TP_Solid	125
Figure 45 — TP_Expression	127
Figure 46 — TP_Complex	130
Figure C.1 — A data set composed of the GM_Primitives	157
Figure C.2 — Simple cartographic representation of sample data	158
Figure C.3 — A 3D Geometric object with labeled coordinates	160
Figure C.4 — Surface example	161
Figure D.1 — Packages and classes for simple topology	163
Figure D.2 — Topology and geometry classes in simple topology	164
Figure D.3 — Feature components in simple topology	165
Figure D.4 — Theme based feature topology	167
Figure D.5 — Geometric example of MiniTopo topology structure	168
Figure D.6 — MiniTopo	169
Figure D.7 — Classic MiniTopo record illustration	171

Tables

Table 1 — Conformance classes for geometric primitives	11
Table 2 — Conformance classes for geometric complexes	11
Table 3 — Conformance classes for topological complexes	11
Table 4 — Conformance classes for topological complexes with geometric realizations	11
Table 5 — Conformance classes for Boolean operators	11
Table 6 — Package and classes	29
Table 7 — Various types of parametric curve surfaces	92
Table 8 — Meaning of Boolean intersection pattern matrix	132
Table 9 — Meaning of Egenhofer intersection pattern matrix	133
Table 10 — Meaning of full topological intersection pattern matrix	134
Table D.1 — Correspondence between original MiniTopo pointers and the current model	172

Foreword

The text of ISO 19107:2003 has been prepared by Technical Committee ISO/TC 211 „Geographic information/Geomatics“ of the International Organization for Standardization (ISO) and has been taken over as EN ISO 19107:2005 by Technical Committee CEN/TC 287 „Geographic Information“, the secretariat of which is held by NEN.

This European Standard shall be given the status of a national standard, either by publication of an identical text or by endorsement, at the latest by July 2005, and conflicting national standards shall be withdrawn at the latest by July 2005.

According to the CEN/CENELEC Internal Regulations, the national standards organizations of the following countries are bound to implement this European Standard: Austria, Belgium, Cyprus, Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Netherlands, Norway, Poland, Portugal, Slovakia, Slovenia, Spain, Sweden, Switzerland and United Kingdom.

Endorsement notice

The text of ISO 19107:2003 has been approved by CEN as EN ISO 19107:2005 without any modifications.

Introduction

This International Standard provides conceptual schemas for describing and manipulating the spatial characteristics of geographic features. Standardization in this area will be the cornerstone for other geographic information standards.

A feature is an abstraction of a real world phenomenon; it is a geographic feature if it is associated with a location relative to the Earth. Vector data consists of geometric and topological primitives used, separately or in combination, to construct objects that express the spatial characteristics of geographic features. Raster data is based on the division of the extent covered into small units according to a tessellation of the space and the assignment to each unit of an attribute value. This International Standard deals only with vector data.

In the model defined in this International Standard, spatial characteristics are described by one or more spatial attributes whose value is given by a geometric object (GM_Object) or a topological object (TP_Object). Geometry provides the means for the quantitative description, by means of coordinates and mathematical functions, of the spatial characteristics of features, including dimension, position, size, shape, and orientation. The mathematical functions used for describing the geometry of an object depend on the type of coordinate reference system used to define the spatial position. Geometry is the only aspect of geographic information that changes when the information is transformed from one geodetic reference system or coordinate system to another.

Topology deals with the characteristics of geometric figures that remain invariant if the space is deformed elastically and continuously — for example, when geographic data is transformed from one coordinate system to another. Within the context of geographic information, topology is commonly used to describe the connectivity of an n -dimensional graph, a property that is invariant under continuous transformation of the graph. Computational topology provides information about the connectivity of geometric primitives that can be derived from the underlying geometry.

Spatial operators are functions and procedures that use, query, create, modify, or delete spatial objects. This International Standard defines the taxonomy of these operators in order to create a standard for their definition and implementation. The goals are to:

- a) Define spatial operators unambiguously, so that diverse implementations can be assured to yield comparable results within known limitations of accuracy and resolution.
- b) Use these definitions to define a set of standard operations that will form the basis of compliant systems, and, thus act as a test-bed for implementers and a benchmark set for validation of compliance.
- c) Define an operator algebra that will allow combinations of the base operators to be used predictably in the query and manipulation of geographic data.

Standardized conceptual schemas for spatial characteristics will increase the ability to share geographic information among applications. These schemas will be used by geographic information system and software developers and users of geographic information to provide consistently understandable spatial data structures.

1 Scope

This International Standard specifies conceptual schemas for describing the spatial characteristics of geographic features, and a set of spatial operations consistent with these schemas. It treats vector geometry and topology up to three dimensions. It defines standard spatial operations for use in access, query, management, processing, and data exchange of geographic information for spatial (geometric and topological) objects of up to three topological dimensions embedded in coordinate spaces of up to three axes.

2 Conformance

2.1 Overview

Clauses 6 and 7 of this International Standard use the Unified Modeling Language (UML) to present conceptual schemas for describing the spatial characteristics of geographic features. These schemas define conceptual classes that shall be used in application schemas, profiles and implementation specifications. The document concerns **ONLY** externally visible interfaces and places no restriction on the underlying implementations other than what is needed to satisfy the interface specifications in the actual situation such as:

- Interfaces to software services using techniques such as COM or CORBA
- Interfaces to databases using techniques such as SQL
- Data interchange using encoding as defined in ISO 19118.

Few applications will require the full range of capabilities described by this conceptual schema. This clause, therefore, defines a set of conformance classes that will support applications whose requirements range from the minimum necessary to define data structures to full object implementation. This flexibility is controlled by a set of UML types that can be implemented in a variety of manners. Implementations that define full object functionality must implement all operations defined by the types of the chosen conformance class, as is common for UML designed object implementations. Implementations that choose to depend on external “free functions” for some or all operations, or forgo them altogether, need not support all operation, but shall always support a data type sufficient to record the state of each of the chosen UML type as defined by its member variables. Common names for “metaphorically identical” but technically different entities are acceptable. The UML model in this International Standard defines abstract types, application schemas define conceptual classes, various software systems define implementation classes or data structures, and the XML from the encoding standard (ISO 19118) defines entity tags. All of these reference the same information content. There is no difficulty in allowing the use of the same name to represent the same information content even though at a deeper level there are significant technical differences in the digital entities being implemented. This “allows” types defined in the UML model to be used directly in application schemas.

There are 39 conformance options for application schemas that define types for the instantiation of geometric or topological objects. They are differentiated on the basis of three criteria.

The first two criteria (complexity and dimensionality) determine the types defined in this schema that shall be implemented according to an application schema that conforms to a given conformance option. In defining the dimensionality of object types to be implemented, the application schema will be required to specify which of

the interpolation types for curves or surfaces they wish to implement. Curve implementations, for those application schemas including 1-dimensional objects, shall always include a “linear” interpolation technique. Application schema including 1-dimensional objects should always include a mechanism to approximate any curve as a line string to allow for transfer of data into simpler schema where needed. Surface implementations, for those application schemas including 2-dimensional objects, shall always include a “planar” interpolation technique. Application schema should always include a mechanism to approximate any surface as collections of planar surface patches to allow for transfer of data into simpler schema where needed. Additional curve and surface interpolation mechanism are optional, but if implemented, they shall follow the definition included in this International Standard.

The third criterion (functional complexity) determines the member elements (attributes, association roles and operations) of those types that shall be implemented. The most limited of such schema would define only data types, and may be used in the transfer of data or the passing of operational parameters to service providers.

The first criterion is level of data complexity. Four levels are identified:

- Geometric primitives
- Geometric complexes
- Topological complexes
- Topological complexes with geometric realization

NOTE Schemas for what is commonly called “spaghetti” data use only unstructured collections of geometric primitives. If single definitions of each component of geometry are required, then geometric complexes are introduced into the schema. Primitives within the same geometric complex share only boundaries. If the schema requires explicit topological information then the geometric complex is expanded to include the structure of a topological complex. The types of object included in a complex are controlled by the dimension of that complex. What is commonly called “chain-node” topology is a 1-dimensional topological complex. What is commonly called “full topology” in a cartographic 2D environment is a 2-dimensional topological complex realized by geometric objects in a 2D coordinate system.

The second criterion is dimensionality. There are four levels for simple geometry:

- 0-dimensional objects
- 0- and 1-dimensional objects
- 0-, 1-, and 2-dimensional objects
- 0-, 1-, 2- and 3-dimensional objects

However, 0-dimensional complexes provide no useful information beyond that provided by 0-dimensional geometric primitives, so conformance classes are only defined for complexes of 1-, 2-, and 3-dimensions.

The third criterion is level of functional complexity. There are three levels.

- Data types only
- Simple operations
- Complete operations

Clause 8 of this International Standard defines three groups of Boolean operators that may be used to derive topological relations between geometric and topological objects. This International Standard defines four conformance classes for application schemas that implement these operators.

2.2 Conformance classes

To conform to this International Standard, an implementation shall satisfy the requirements of the Abstract test suite (ATS) in Annex A for a specified conformance class. Table 1 through Table 5 identify the clauses of the ATS that apply for each conformance class.

Table 1 — Conformance classes for geometric primitives

Dimension	Data Types	Simple Operations	Complete Operations
0	A.1.1.1	A.1.2.1	A.1.3.1
1	A.1.1.2	A.1.2.2	A.1.3.2
2	A.1.1.3	A.1.2.3	A.1.3.3
3	A.1.1.4	A.1.2.4	A.1.3.4

Table 2 — Conformance classes for geometric complexes

Dimension	Data Types	Simple Operations	Complete Operations
1	A.2.1.1	A.2.2.1	A.2.3.1
2	A.2.1.2	A.2.2.2	A.2.3.2
3	A.2.1.3	A.2.2.3	A.2.3.3

Table 3 — Conformance classes for topological complexes

Dimension	Data Types	Simple Operations	Complete Operations
1	A.3.1.1	A.3.2.1	A.3.3.1
2	A.3.1.2	A.3.2.2	A.3.3.2
3	A.3.1.3	A.3.2.3	A.3.3.3

Table 4 — Conformance classes for topological complexes with geometric realizations

Dimension	Data Types	Simple Operations	Complete Operations
1	A.4.1.1	A.4.2.1	A.4.3.1
2	A.4.1.2	A.4.2.2	A.4.3.2
3	A.4.1.3	A.4.2.3	A.4.3.3

Table 5 — Conformance classes for Boolean operators

Set operators	A.5.1
Egenhofer operators	A.5.2
Full topological operators	A.5.3
All operators	A.5.4

3 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 19109:—¹⁾, *Geographic information — Rules for application schema*

ISO 19111:—¹⁾, *Geographic information — Spatial referencing by coordinates*

ISO/IEC 11404:1996, *Information technology — Programming languages, their environments and system software interfaces — Language-independent datatypes*

4 Terms and definitions

For the purposes of this document, the following terms and definitions apply. The terms are listed alphabetically in this clause. In Annex B they are organized by their conceptual relationships.

4.1 application

manipulation and processing of data in support of user requirements [ISO 19101]

4.2 application schema

conceptual schema for data required by one or more **applications** [ISO 19101]

4.3 bag

finite, unordered collection of related items (**objects** or values) that may be repeated

NOTE Logically, a bag is a set of pairs <item, count>.

4.4 boundary

set that represents the limit of an entity

NOTE **Boundary** is most commonly used in the context of geometry, where the set is a collection of points or a collection of objects that represent those points. In other arenas, the term is used metaphorically to describe the transition between an entity and the rest of its domain of discourse.

4.5 buffer

geometric object that contains all **direct positions** whose distance from a specified **geometric object** is less than or equal to a given distance

4.6 circular sequence

sequence which has no logical beginning and is therefore equivalent to any circular shift of itself; hence the last item in the sequence is considered to precede the first item in the sequence

4.7 class

description of a **set** of **objects** that share the same attributes, operations, methods, relationships, and semantics [ISO/TS 19103]

1) To be published.

NOTE A class may use a **set** of **interfaces** to specify collections of **operations** it provides to its environment. The term was first used in this way in the general theory of object oriented programming, and later adopted for use in this same sense in UML.

4.8

closure

union of the **interior** and **boundary** of a **topological** or **geometric object**

4.9

coboundary

set of **topological primitives** of higher topological dimension associated with a particular **topological object**, such that this **topological object** is in each of their **boundaries**

NOTE If a node is on the boundary of an edge, that edge is on the coboundary of that node. Any orientation parameter associated to one of these relations would also be associated to the other. So that if the node is the end node of the edge (defined as the end of the positive directed edge), then the positive orientation of the node (defined as the positive directed node) would have the edge on its coboundary, see Figure 35.

4.10

composite curve

sequence of **curves** such that each curve (except the first) starts at the end point of the previous curve in the **sequence**

NOTE A composite curve, as a set of **direct positions**, has all the properties of a curve.

4.11

composite solid

connected **set** of **solids** adjoining one another along shared **boundary surfaces**

NOTE A composite solid, as a set of **direct positions**, has all the properties of a solid.

4.12

composite surface

connected **set** of **surfaces** adjoining one another along shared **boundary curves**

NOTE A composite surface, as a set of **direct positions**, has all the properties of a surface.

4.13

computational geometry

manipulation of and calculations with geometric representations for the implementation of geometric **operations**

EXAMPLE Computational geometry operations include testing for geometric inclusion or intersection, the calculation of **convex hulls** or **buffer zones**, or the finding of shortest distances between **geometric objects**.

4.14

computational topology

topological concepts, structures and algebra that aid, enhance or define **operations** on **topological objects** usually performed in **computational geometry**

4.15

connected

property of a **geometric object** implying that any two **direct positions** on the object can be placed on a **curve** that remains totally within the object

NOTE A topological object is connected if and only if all its **geometric realizations** are connected. This is not included as a definition because it follows from a theorem of topology.

4.16

connected node

node that starts or ends one or more **edges**

4.17

convex hull

smallest **convex set** containing a given **geometric object** [Dictionary of Computing [7]]

NOTE “Smallest” is the set theoretic smallest, not an indication of a measurement. The definition can be rewritten as “the intersection of all convex sets that contain the geometric object”.

4.18

convex set

geometric set in which any **direct position** on the straight-line segment joining any two **direct positions** in the geometric set is also contained in the **geometric set** [Dictionary of Computing [7]]

NOTE Convex sets are “simply connected”, meaning that they have no interior holes, and can normally be considered topologically isomorphic to a Euclidean ball of the appropriate dimension. So the surface of a sphere can be considered to be geodesically convex.

4.19

coordinate

one of a **sequence** of N-numbers designating the position of a **point** in N-dimensional space [ISO 19111]

NOTE In a **coordinate reference system**, the numbers must be qualified by units.

4.20

coordinate dimension

number of measurements or axes needed to describe a position in a **coordinate system**

4.21

coordinate reference system

coordinate system that is related to the real world by a datum [ISO 19111]

4.22

coordinate system

set of mathematical rules for specifying how **coordinates** are to be assigned to **points** [ISO 19111]

4.23

curve

1-dimensional **geometric primitive**, representing the continuous image of a line

NOTE The **boundary** of a **curve** is the **set of points** at either end of the **curve**. If the curve is a cycle, the two ends are identical, and the curve (if topologically closed) is considered to not have a boundary. The first **point** is called the **start point**, and the last is the **end point**. Connectivity of the curve is guaranteed by the “continuous image of a line” clause. A topological theorem states that a continuous image of a connected set is connected.

4.24

curve segment

1-dimensional **geometric object** used to represent a continuous component of a **curve** using homogeneous interpolation and definition methods

NOTE The **geometric set** represented by a single curve segment is equivalent to a **curve**.

4.25

cycle

⟨geometry⟩ **spatial object** without a **boundary**

NOTE Cycles are used to describe boundary components (see **shell**, **ring**). A cycle has no boundary because it closes on itself, but it is bounded (i.e., it does not have infinite extent). A circle or a sphere, for example, has no boundary, but is bounded.

4.26

direct position

position described by a single set of **coordinates** within a **coordinate reference system**

4.27**directed edge**

directed topological object that represents an association between an **edge** and one of its orientations

NOTE A directed edge that is in agreement with the orientation of the edge has a + orientation, otherwise, it has the opposite (–) orientation. Directed edge is used in **topology** to distinguish the right side (–) from the left side (+) of the same edge and the **start node** (–) and **end node** (+) of the same edge and in **computational topology** to represent these concepts.

4.28**directed face**

directed topological object that represents an association between a **face** and one of its orientations

NOTE The orientation of the **directed edges** that compose the exterior **boundary** of a directed face will appear positive from the direction of this vector; the orientation of a directed face that bounds a **topological solid** will point away from the **topological solid**. Adjacent solids would use different orientations for their shared boundary, consistent with the same sort of association between adjacent faces and their shared edges. Directed faces are used in the **coboundary** relation to maintain the spatial association between **face** and **edge**.

4.29**directed node**

directed topological object that represents an association between a **node** and one of its orientations

NOTE Directed nodes are used in the **coboundary** relation to maintain the spatial association between **edge** and **node**. The orientation of a node is with respect to an edge, “+” for end node, “–” for start node. This is consistent with the vector notion of “result = end - start”.

4.30**directed solid**

directed topological object that represents an association between a **topological solid** and one of its orientations

NOTE Directed solids are used in the **coboundary** relation to maintain the spatial association between **face** and **topological solid**. The orientation of a solid is with respect to a face, “+” if the upNormal is outward, “–” if inward. This is consistent with the concept of “up = outward” for a surface bounding a solid.

4.31**directed topological object**

topological object that represents a logical association between a **topological primitive** and one of its orientations

4.32**domain**

well-defined **set** [ISO/TS 19103]

NOTE Domains are used to define the domain and range of operators and **functions**.

4.33**edge**

1-dimensional **topological primitive**

NOTE The **geometric realization** of an edge is a **curve**. The **boundary** of an edge is the **set** of one or two **nodes** associated to the edge within a **topological complex**.

4.34**edge-node graph**

graph embedded within a **topological complex** composed of all of the **edges** and **connected nodes** within that **complex**

NOTE The **edge-node graph** is a subcomplex of the complex within which it is embedded.

4.35

end node

node in the **boundary** of an **edge** that corresponds to the **end point** of that **edge** as a **curve** in any valid **geometric realization** of a **topological complex** in which the **edge** is used

4.36

end point

last point of a **curve**

4.37

exterior

difference between the universe and the **closure**

NOTE The concept of exterior is applicable to both **topological** and **geometric complexes**.

4.38

face

2-dimensional **topological primitive**

NOTE The **geometric realization** of a face is a **surface**. The **boundary** of a face is the **set** of **directed edges** within the same **topological complex** that are associated to the face via the boundary relations. These can be organized as **rings**.

4.39

feature

abstraction of real world phenomena [ISO 19101]

NOTE A feature may occur as a type or an **instance**. Feature type or feature instance should be used when only one is meant.

4.40

feature attribute

characteristic of a **feature** [ISO 19101]

NOTE A feature attribute has a name, a data type, and a value **domain** associated to it. A feature attribute for a feature **instance** also has an attribute value taken from the value domain.

4.41

function

rule that associates each element from a **domain** (source, or domain of the function) to a unique element in another domain (target, co-domain, or range)

4.42

geographic information

information concerning phenomena implicitly or explicitly associated with a location relative to the Earth [ISO 19101]

4.43

geometric aggregate

collection of **geometric objects** that has no internal structure

NOTE No assumptions about the spatial relationships between the elements can be made.

4.44

geometric boundary

boundary represented by a **set** of **geometric primitives** of smaller **geometric dimension** that limits the extent of a **geometric object**

4.45**geometric complex**

set of disjoint **geometric primitives** where the **boundary** of each **geometric primitive** can be represented as the union of other **geometric primitives** of smaller dimension within the same **set**

NOTE The **geometric primitives** in the set are disjoint in the sense that no **direct position** is **interior** to more than one **geometric primitive**. The set is closed under **boundary operations**, meaning that for each element in the **geometric complex**, there is a collection (also a **geometric complex**) of **geometric primitives** that represents the boundary of that element. Recall that the **boundary** of a point (the only 0D primitive object type in geometry) is empty. Thus, if the largest dimension geometric primitive is a solid (3D), the composition of the boundary operator in this definition terminates after at most three steps. It is also the case that the boundary of any object is a **cycle**.

4.46**geometric dimension**

largest number n such that each **direct position** in a **geometric set** can be associated with a subset that has the **direct position** in its **interior** and is similar (isomorphic) to R^n , Euclidean n -space

NOTE Curves, because they are continuous images of a portion of the real line, have geometric dimension 1. Surfaces cannot be mapped to R^2 in their entirety, but around each point position, a small neighbourhood can be found that resembles (under continuous functions) the interior of the unit circle in R^2 , and are therefore 2-dimensional. In this International Standard, most surface patches (instances of GM_SurfacePatch) are mapped to portions of R^2 by their defining interpolation mechanisms.

4.47**geometric object**

spatial object representing a **geometric set**

NOTE A **geometric object** consists of a **geometric primitive**, a collection of **geometric primitives**, or a geometric complex treated as a single entity. A geometric object may be the spatial representation of an **object** such as a **feature** or a significant part of a **feature**.

4.48**geometric primitive**

geometric object representing a single, **connected**, homogeneous element of space

NOTE Geometric primitives are non-decomposed **objects** that present information about geometric configuration. They include **points**, **curves**, **surfaces**, and **solids**.

4.49**geometric realization**

geometric complex whose **geometric primitives** are in a 1-to-1 correspondence to the **topological primitives** of a **topological complex**, such that the **boundary** relations in the two complexes agree

NOTE In such a realization the topological primitives are considered to represent the **interiors** of the corresponding geometric primitives. Composites are closed.

4.50**geometric set**

set of **direct positions**

NOTE This set in most cases is infinite.

4.51**graph**

set of **nodes**, some of which are joined by **edges**

NOTE In geographic information systems, a graph can have more than one **edge** joining two **nodes**, and can have an **edge** that has the same **node** at both ends.

4.52

homomorphism

relationship between two **domains** (such as two complexes) such that there is a structure-preserving **function** from one to the other

NOTE **Homomorphisms** are distinct from **isomorphisms** in that no inverse function is required. In an isomorphism, there are essentially two **homomorphisms** that are functional inverses of one another. Continuous functions are topological homomorphisms because they preserve “topological characteristics”. The mapping of topological complexes to their geometric realizations preserves the concept of boundary and is therefore a homomorphism.

4.53

instance

object that realizes a **class**

4.54

interior

set of all **direct positions** that are on a **geometric object** but which are not on its **boundary**

NOTE The **interior** of a **topological object** is the homomorphic image of the interior of any of its **geometric realizations**. This is not included as a definition because it follows from a theorem of topology.

4.55

isolated node

node not related to any **edge**

4.56

isomorphism

relationship between two **domains** (such as two complexes) such that there are 1-to-1, structure-preserving **functions** from each **domain** onto the other, and the composition of the two **functions**, in either order, is the corresponding identity function

NOTE A **geometric complex** is isomorphic to a **topological complex** if their elements are in a 1-to-1, dimension- and **boundary**-preserving correspondence to one another.

4.57

neighbourhood

geometric set containing a specified **direct position** in its **interior**, and containing all direct positions within a specified distance of the specified direct position

4.58

node

0-dimensional **topological primitive**

NOTE The **boundary** of a node is the empty **set**.

4.59

object

entity with a well defined **boundary** and identity that encapsulates state and behaviour [UML Semantics [19]]

NOTE This term was first used in this way in the general theory of object oriented programming, and later adopted for use in this same sense in UML. An object is an instance of a **class**. Attributes and relationships represent state. Operations, methods, and state machines represent behaviour.

4.60

planar topological complex

topological complex that has a **geometric realization** that can be embedded in Euclidean 2 space

4.61

point

0-dimensional **geometric primitive**, representing a position

NOTE The **boundary** of a point is the empty set.

4.62

record

finite, named collection of related items (**objects** or values)

NOTE Logically, a record is a set of pairs <name, item>.

4.63

ring

simple curve which is a **cycle**

NOTE Rings are used to describe **boundary** components of surfaces in 2D and 3D **coordinate systems**.

4.64

sequence

finite, ordered collection of related items (**objects** or values) that may be repeated

NOTE Logically, a sequence is a set of pairs <item, offset>. LISP syntax, which delimits sequences with parentheses and separates elements in the sequence with commas, is used in this International Standard.

4.65

set

unordered collection of related items (**objects** or values) with no repetition

4.66

shell

simple surface which is a **cycle**

NOTE Shells are used to describe **boundary** components of solids in 3D **coordinate systems**.

4.67

simple

property of a **geometric object** that its **interior** is isotropic (all points have **isomorphic** neighbourhoods), and hence everywhere locally **isomorphic** to an open subset of a Euclidean coordinate space of the appropriate dimension

NOTE This implies that no interior **direct position** is involved in a self-intersection of any kind.

4.68

solid

3-dimensional **geometric primitive**, representing the continuous image of a region of Euclidean 3 space

NOTE A **solid** is realizable locally as a three parameter **set** of **direct positions**. The **boundary** of a **solid** is the set of oriented, closed **surfaces** that comprise the limits of the **solid**.

4.69

spatial object

object used for representing a spatial characteristic of a feature

4.70

spatial operator

function or procedure that has at least one spatial parameter in its **domain** or range

NOTE Any UML operation on a spatial object would be classified as a spatial operator as are the query operators in Clause 8 of this International Standard.

4.71

start node

node in the **boundary** of an **edge** that corresponds to the **start point** of that **edge** as a **curve** in a valid **geometric realization** of the **topological complex** in which the **edge** is used

4.72

start point

first **point** of a **curve**

4.73

strong substitutability

ability for any **instance** of a **class** that is a descendant under inheritance or realization of another **class**, type or **interface** to be used in lieu of an **instance** of its ancestor in any context

NOTE The weaker forms of substitutability make various restrictions on the context of the implied substitution.

4.74

subcomplex

complex all of whose elements are also in a larger complex

NOTE Since the definitions of **geometric complex** and **topological complex** require only that they be closed under **boundary operations**, the **set** of any primitives of a particular dimension and below is always a subcomplex of the original, larger complex. Thus, any full **planar topological complex** contains an **edge-node graph** as a subcomplex.

4.75

surface

2-dimensional **geometric primitive**, locally representing a continuous image of a region of a plane

NOTE The **boundary** of a **surface** is the set of oriented, closed **curves** that delineate the limits of the **surface**. **Surfaces** that are isomorphic to a sphere, or to an n-torus (a topological sphere with n “handles”) have no boundary. Such surfaces are called **cycles**.

4.76

surface patch

2-dimensional, **connected geometric object** used to represent a continuous portion of a **surface** using homogeneous interpolation and definition methods

4.77

topological boundary

boundary represented by a **set** of oriented **topological primitives** of smaller topological dimension that limits the extent of a **topological object**

NOTE The **boundary** of a **topological complex** corresponds to the boundary of the **geometric realization** of the topological complex.

4.78

topological complex

collection of **topological primitives** that is closed under the **boundary** operations

NOTE Closed under the boundary operations means that if a **topological primitive** is in the **topological complex**, then its **boundary** objects are also in the **topological complex**.

4.79

topological dimension

minimum number of free variables needed to distinguish nearby **direct positions** within a **geometric object** from one another

NOTE The free variables mentioned above can usually be thought of as a local coordinate system. In a 3D coordinate space, a plane can be written as $P(u, v) = A + uX + vY$, where u and v are real numbers and A is any point on the plane, and X and Y are two vectors tangent to the plane. Since the locations on the plane can be distinguished by u and v (here universally), the plane is 2D and (u, v) is a coordinate system for the points on the plane. On generic surfaces,

this cannot, in general, be done universally. If we take a plane tangent to the surface, and project points on the surface onto this plane, we will normally get a local isomorphism for small neighbourhoods of the point of tangency. This “local coordinate” system for the underlying surface is sufficient to establish the surface as a 2D topological object.

Since this International Standard deals only with spatial coordinates, any 3D object can rely on coordinates to establish its topological dimension. In a 4D model (spatio-temporal), tangent spaces also play an important role in establishing topological dimension for objects up to 3D.

4.80

topological expression

collection of oriented **topological primitives** which is operated upon like a multivariate polynomial

NOTE Topological expressions are used for many calculations in **computational topology**.

4.81

topological object

spatial object representing spatial characteristics that are invariant under continuous transformations

NOTE A **topological object** is a **topological primitive**, a collection of topological primitives, or a **topological complex**.

4.82

topological primitive

topological object that represents a single, non-decomposable element

NOTE A **topological primitive** corresponds to the interior of a **geometric primitive** of the same **dimension** in a **geometric realization**.

4.83

topological solid

3-dimensional **topological primitive**

NOTE The **boundary** of a topological solid consists of a set of **directed faces**.

4.84

universal face

unbounded **face** in a 2-dimensional complex

NOTE The **universal face** is normally not part of any feature, and is used to represent the unbounded portion of the data set. Its interior boundary (it has no exterior boundary) would normally be considered the exterior boundary of the map represented by the data set. This International Standard does not special case the **universal face**, but application schemas may find it convenient to do so.

4.85

universal solid

unbounded **topological solid** in a 3-dimensional complex

NOTE The **universal solid** is the 3-dimensional counterpart of the universal face, and is also normally not part of any feature.

4.86

vector geometry

representation of **geometry** through the use of constructive **geometric primitives**

5 Symbols, notation and abbreviated terms

5.1 Presentation and notation

5.1.1 Unified Modeling Language (UML) concepts

In this International Standard, conceptual schemas are presented in the Unified Modeling Language (UML).

A UML class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A parameterized class is a single description of a set of classes with common operations, methods and relationships that vary from one to another based upon a set of parameters which control the precise structure of the attributes and the behaviour of the operations. For example, integer parameters are used to fix the size of certain attribute arrays within realizations of the parameterized class (called instantiated classes).

The fact that an element of geometry or topology has been modelled in one way or another in this International Standard should not be considered a restriction on implementations. Attributes may be implemented either directly as data or as a pair of “accessor” and “mutator” operations for getting and setting values. Most diagrams in this document are “context diagrams” which center about a single class and display its attributes, operations, and important relationships. Other diagrams are overviews of class relationships. UML does not require all relationships to be displayed in all diagrams, and some of the more trivial ones have been left out of some diagrams to keep them simple. For example, GM_Object has an obvious relationship to Set<GM_Object>, but this is not explicitly shown in the context diagram for GM_Object.

5.1.2 Attributes, operations, and associations

Attributes and operations are presented in the UML diagrams in compliance with the UML Notation Guide [18]. UML notation for an attribute has the form:

```
Attribute-declaration ::=
    "«" stereotype "»" visibility name multiplicity " : "
    type = initial-value {property, ...}
multiplicity ::= "[" cardinality-range, ... "]"
cardinality-range ::= begin-value {".." end-value}
```

UML notation for an operation has the form:

```
Operation ::= "«" stereotype "»" visibility
    name "(" parameterlist ")" " : " [return-type], ...
    {"{" property{=value}, ..."}"},...
parameterlist ::= [direction] parameter-name " : " type ["=" default-value]
```

where the various parts of the above syntax are as follows:

- a) stereotype — use tag for the attribute or operation being defined (see below)
- b) visibility — public (+), private (–) or protected (#) indicating the visibility of this attribute or operation from outside the object. If the visibility includes “/”, then the attribute is derived from some other part of the model.
- c) name — the name of the attribute or operation.
- d) multiplicity — the number of values that this attribute can have, assumed to be organized as a set unless otherwise specified; this is an extension of and consistent with the “size” mechanism of ISO/IEC 11404, except for the use of “[..]” which is UML notation. To maintain consistency of concept, this International

Standard uses a single multiplicity syntax (from UML) even when using it in conjunction with the “size” subtyping of ISO/IEC 11404.

- e) begin-value — any integer, a valid multiplicity; if no end-value follows, then only the begin-value is added as a possible multiplicity.
- f) end-value — any integer bigger than the preceding begin-value, or “*n*” meaning infinity or an unbounded cardinality-range, the meaning of “*a..b*” is any integer *j* such that $a \leq j \leq b$; [*a..a*] is assumed to mean the same as [*a*].
- g) parameter_{list} — a comma separated list of parameter declarations.
- h) parameter-name — name of a parameter to the operation, usually indicative of the role of the parameter in the operation being defined. Note that the syntax structures for an operation and for an attribute are identical except that an operation includes a parameter list and an attribute includes a multiplicity.
- i) direction — optional indicator of direction flow for this parameter being 'in' (the value is set before invocation of and affects the operation), 'out' (the value is set during the operation and its value is accessible by the invoker upon completion of the operation), or 'inout' (the value is set before the invocation, and affects the operations, and is reset by the operation by a value accessible by the invoker upon completion of the operation). The default direction of any parameter is “in”.
- j) type — the type, either object or value of the preceding parameter or attribute.
- k) default-value — the value of an in or inout parameter if not specified by the invoker. The value of an object's attribute if not set by any constructor.
- l) return-type — the type of the return value or object for the operation, essentially the type of the operation.
- m) property — additional information about the attribute or operation, such as NOT NULL or UNIQUE. Can be structured as a property name followed by a value, such as “{size = [0..n]}”. (See ISO/IEC 11404 for some interpretations of properties as subtypes.)
- n) ... — the preceding can be repeated any number of times.
- o) initial-value — default value of the attribute, used when a new object is constructed unless specifically overridden by the constructor's parameter list.

In the text, notation from the Object Constraint Language (OCL) is used with some slight modifications. The “ocl” prefix was dropped from many operators, since it was unnecessary and confusing, especially since these operators appear in the basic types section of ISO TS 19103 without the prefix. The “::” is a resolution operator indicating the name space of that which follows. In most cases in OCL, the name space is the class in which the operation is defined, but it can also include the package name in which a class is defined. In this document all name spaces are class identifier and can take only one of two forms:

```
class-identifier ::= class-name | package-name::class-identifier
type ::= class-identifier
```

Unless there is a potential of confusion or a need for emphasis, the package name is not included. In this International Standard, all class names include a two-letter package-identifier prefix followed by an underscore “_” and are unique within the model. This avoids the need for package names resolution in type and class names. Profiles of this International Standard are encouraged to retain this convention if possible. For attributes, role names and operations, the text description is as follows:

```
attribute-name {multiplicity} : attribute-type
{association-name "::"}role-name {multiplicity} " : " attribute-type
{type-1 "::"}operation-name(name-2 " : " type-2, name-3 " : " type-3, ... )
" : " return-type
```

For roles, if the multiplicity is not [1], then the assumption is that the role values are organized as a set. If the role values need to be organized in some other manner, then the attribute-type with the appropriate collection parameterized class should be used, with the multiplicity given as a size in accordance with ISO/IEC 11404 as in:

```
boundary : CircularSequence<GM_OrientableCurve> {size = [1..n]}
```

Object-oriented operator notation (such as would be used in C++) places the first parameter before the operation as in a method declaration as follows:

```
return-type type-1::operation(type-2, type-3 ... )
```

Such methods are restricted in name space, in the sense that they are available only if the “object-type-1” name space is available. In addition, during invocation, the identification of the implicit parameter of type “type-1” is known. In OCL, this object is identified as “self”. In C++, this object is identified as “this”. In non-object languages or for free functions in an object language, the functional notation for an operation does not distinguish the first parameter in any manner and is written:

```
operation(name-1 : type-1, name-2 : type-2, name-3 : type-3 ...) : return-type,  
...
```

These notations are equivalent (except for emphasis) and both may be used in profiles of this International Standard.

These operation definitions are called “operation signatures” or “protocols”. This distinguishes the operation from the invocation mechanism. In UML, the formal notation defines protocols, and the operations associated to them are defined only informally in the associated documentation, which can include OCL constraints.

In the view that an attribute can be considered a type of operation (mutator and accessor pairs), this term can be extended to include attribute “signatures”. The definition of a signature includes the operation name; the parameter names and types; and the return type. Methods or attributes can be overridden by providing a new method whose signature is the same as the original except that some of the types have been replaced, usually by subtypes of the originals. The reuse of signatures is called “polymorphism”. Polymorphism arising from class inheritance is called “structural”. Polymorphism arising from semantic similarity is called “natural” or “generic”. For example, in the geometry and topology classes, the common protocol for “boundary” is a natural polymorphism in that it arises from an operational constraint based on the definition of topology. It is not a structural polymorphism, since the two packages do not share a common superclass ancestor. Assuming that the class inheritance hierarchy is based on semantics of the objects, then structural polymorphism is natural. Polymorphism that does not depend on semantic similarity is “ad-hoc”. For example, the use of “+” in numbers to denote addition and in character string classes to denote concatenation is ad-hoc polymorphism. Ad-hoc polymorphism is semantically confusing, and is therefore not used in this International Standard, and should be avoided in profiles of this International Standard.

Most operations are defined in a functional style, that is all parameters are passed as read-only (direction = “in”), and the only modification or creation of objects is done by using the return type in an assignment operator. In describing interfaces, the adjective “this” indicates the entity whose object interfaces are being invoked. In OCL, this object is referred to as “self”. If an object is passed as a parameter to the method of another object, it is referred to as a “passed” object.



Figure 1 — UML example association

Each association in the model is given an association name, and each class that participates in the association is given a role name. In the case where “Class1” has an association “Relation” with “Class2” as diagrammed in Figure 1, then the two classes when implemented would normally include as “attributes” references to the other class named for the roles of the relation such as:

```
Class1::gadget : Set<Class2> // this is a many to many, strong linkage
Class2::widget : Set<Class1>
```

Or,

```
Class1::gadget [1..n] : Class2 // this is a many to many, strong linkage
Class2::widget [1..n] : Class1
```

Note that the role name for Class2 is used as the variable name in Class1 that points to Class2. The type of collection given will depend upon the type of sort criteria for the role.

```
Class1::gadget : Sequence<Class2> // this is the many to many, ordered,
                                // strong linkage
Class2::widget : Set<Class1>
```

Most roles will either correspond to Set (unordered relation) or Sequence (ordered relation). Some roles will be circular Sequences when the ordering does not have a natural start position. This notation is used where appropriate in the text below, but this is not meant to imply a particular implementation of associations. For weak linkages, where the target class does not depend on the existence of the association, the parameterized class Reference<.> is used where appropriate, as in:

```
Class1::gadget [1..n] : Reference<Class2> // weak linkage
Class2::widget [1..n] : Class1           // strong linkage
```

5.1.3 Stereotypes

Most entities in a UML model can be described by a “stereotype” which is included near the name of the object and enclosed in guillemets “«” and “»”. The stereotype allows the model to extend UML to include descriptions of elements of the model. In this International Standard the following stereotypes are used:

- a) «Interface» – the class is not directly instantiable, but is used as an abstract collection of operation signatures. The purpose of an «Interface» class is to define a reference class for a structural polymorphism for operations. In various programming languages, such a class might be called “virtual” or “abstract”. In the UML standard, «Interface» classes must not have attributes or associations visible from the interface. An interface may participate in an association provided the interface cannot see the association; that is, a class (other than an interface) may have an association to an interface that is navigable from the class but not from the Interface. As used in this International Standard, within application schemas, non-interface Classes with the same name as an interface class from this International Standard cannot be incorporated in an application schema without a logical contradiction.
- b) «Type» – the class is not directly instantiable, but is used as an abstract collection of operation, attribute and relation signatures. The purpose of a «Type» class is to define a reference class for a structural polymorphism that includes attributes and associations. The actual internal organization of the attributes and associations is implementation dependent. Because the organization of these elements is not known, UML does not allow «Type» classes to have any methods (implementations of operations). As used in this International Standard, within application schemas, Implementation Classes with the same name as a Type from this International Standard may be created as long as that Type is not also Abstract (as indicated through UML symbology, the name in an *italics* font).

- c) «Abstract» – (also represented in UML by the class name being in an *italics* font) the class is similar to a «Type» except that methods are allowed. Such classes define a root class for a structural polymorphism that includes these additional elements. As used in this International Standard, within application schemas, Concrete Classes with the same name as an Abstract Class from this International Standard cannot be incorporated in an application schema without a logical contradiction.
- d) «DataType» – the class is directly instantiable and its primary purpose is the encapsulation of data, as opposed to taxonomic or behavioural descriptions. «DataTypes» do not have an identity of their own and must be strongly aggregated into some sort of container such as being an attribute in another class, or being the target class of a strong aggregation. «DataType» types cannot be used as the target of weak aggregations, nor can they be used in the Reference< > parameterized class.
- e) «Union» – a type consisting of one and only one of several alternatives (listed as member attributes). This is similar to a discriminated union in many programming languages. In some languages using pointers, this requires a “void” pointer that can be cast to the appropriate type as determined by a discriminator attribute.
- f) «Leaf» – applicable to packages that contain no subpackages, only object classes and interface definitions. Although not a technical requirement, this International Standard places all object definitions in leaf packages, which are then organized, in larger, non-leaf packages.
- g) «CodeList» – similar to an enumeration, in that one of a number of values is possible, but differs in intent, in that a code list may be expanded over time. Most code list are stored as numeric values, but some implementations use character strings. In this International Standard, code list are declared as having character string codes, but this is an implementation detail, and pure numeric codes are acceptable.

5.1.4 Data types and collection types

Several collection types are required to make the standard consistent, but these types do not have to be specific in terms of their interfaces. While these types are not included in UML, they are often implied by usage of the Object Constraint Language (OCL), see ISO TS 19103.

The most common of these interfaces is the finite set. If we have a type “T”, we denote a new instantiated class type called “Set<T>” to consist of all finite, unordered sets of objects of type “T”. Implementation environments often supply several common collection types such as arrays, and we do not wish to try to impose a universal interface on these types. ISO TS 19103 includes an example interface definition for these types. This International Standard does not restrict the use of logically equivalent types native to particular implementation environments. Some basic class types and parameterized types, such as these collections types that are used in this International Standard include the following:

- a) TransfiniteSet<T> – a possibly infinite set; restricted only to values. For example, the integers and the real numbers are transfinite sets. This is actually the usual definition of set in mathematics, but programming languages restrict the term set to mean finite set.
- b) Set<T> – a finite set, usable for object types. Each object is considered to be in the set only once. Not usually a strong aggregation since each object can be an element of many sets. Unless otherwise noted, this International Standard will use Set to mean a weak aggregation (the equivalent of a strong aggregation of the form Set<Reference<T>>).
- c) Bag<T> – a finite, unordered set where each object may be considered to be in the set multiple times. Can be logically thought of as a set of pairs <object T, count Integer>.
- d) Sequence<T> – an ordered finite set of objects, possibly with repeated values. Can be logically thought of as a set of pairs <object : T, offset : Integer>, where the offset gives the position of the object in the sequence. Depending on the implementation, offsets can be counted from 1, 0 or any arbitrary point. Projection onto the object of each pair produces a Bag. Elimination of duplicates produces a Set.

- e) `CircularSequence<T>` – a sequence that wraps back on itself and is considered identical to all of its circular shifts. Can be considered as an equivalence class of Sequences, differing from one another by a circular shift of offset.
- f) `Reference<T>` – a reference to an identifiable object of class T, equivalent to a pointer in C++, a REF in SQL99 (previously called SQL 3), or a Java class variable. In the text declaration of classes in this International Standard, a weak association is represented as a reference to the appropriate class. Strong aggregations are represented in the same manner as attributes.
- g) Number, Float, Integer, Real – various simple value types usually instantiated as programming language primitives within the environment, see ISO TS 19103.
- h) Length, Area, Distance – various scalar values, associated to a particular unit of measure such as the meter or acre, see ISO TS 19103.

NOTE Representations of associations, in the “code listings”, have used the convention most common in code generators for UML. Association roles have been used as member names of type `Reference<T>` where T is the target class of the role name in the association. In cases where ambiguity could exist, the association name is used as the name space for the role `<association_name>::<role_name> : Reference<target_class>`. Logically, this could also have been done by using the source class of the association, `<source_class>::<role_name> : Reference<target_class>`. If the association is a strong aggregation then the reference can logically be removed, `<source_class>::<role_name> : target_class`. The semantics of a strong aggregation, one-way association is logically identical to a member attribute. One of the sources of alternative designs in UML is the use of associations versus the use of role-like attributes. Once a code generator has been used, the backward generation of UML association (lacking any other information) might round-trip engineer to pairs of role-like attributes.

Some data types are simply instances of the Record type defined in ISO TS 19103 and in slightly different terms in ISO/IEC 11404. Since the latter International Standard has a specification that might be confused with parameter lists, this International Standard uses a slightly modified syntax (“(.)” external parenthesis replaced by “<.>”):

```
Record Type ::= "<" field-name " : " type [= default-value],... ">"
Record Instance ::= "<" field-name " : " field-value,..." ">"
```

Note that the syntax for a multiple return type is consistent with this syntax for Record, except that the braces are omitted. This International Standard uses the Record syntax above when it is stand-alone, but uses the standard UML multiple return type syntax when specifying operations that return record-like structures as anonymous types.

Several of the operations defined in this International Standard use NULL and EMPTY as possible values. NULL means that the value asked for is undefined. This International Standard assumes that all NULL values are equivalent. If a NULL is returned when an object has been requested, then the assumption is that no object matching the criteria specified exists. EMPTY refers to objects that can be interpreted as sets of one form or another, and means that the set in question contains no elements. Unlike programming systems that have strongly typed aggregates, this International Standard uses the mathematical tautology that there is one and only one empty set and that any object representing that empty set is equivalent to any other set doing the same. Other than being empty, such sets lack any inherent information, and so a NULL value is assumed equivalent to an EMPTY set in appropriate context.

5.1.5 Strong substitutability

This International Standard assumes that implementation profiles and transfer schemas will be built using a strong version of substitutability. This means that at several places in designing an application schema, a profile builder may use a class in lieu of one defined in this schema as long as it supports the data, operation and associations required of the base class. The method of implementation of this substitutability is not normative, and may be done in a variety of manners depending on the characteristics of the implementation environment. This is especially true of transfer standards, which by their nature depend on data types. Entities in transfer sets may only be tenuously related to the base class in this International Standard, in that they may

be data-only representational forms. Places where substitutability is most useful are examined in subclauses associated to the class most likely to take advantage of this technique.

This assumption requires a strict adherence to the semantics of subclassing as an “is type of” hierarchy. Each instance of a class must be a member of all of the sets defined by the semantics of the supertypes of that class. Thus, we can define a Circle to be a subtype of Ellipse, but not the other way around, even though this is counterintuitive to the notion that subtypes are more complex than their supertypes.

5.2 Organization

The clauses in this document are organized in terms of UML packages. A package is a set of related types and interfaces that form a consistent component of a software system design. Packages do not usually form a complete system since they often invoke the services provided by other packages in the system. When one package, acting as a client, uses another, acting as a server, to supply needed services, the client package is said to be dependent on the server package. This dependency occurs when an object class in the package accesses another object defined in the server package. Since it is rare in geographic information for geometry to be purely client or server, these stereotypes are not used in this International Standard. Since dependent classes are associated to their server via an association that can carry the request, most object class dependencies derive from object class associations. Each dependency between objects in different packages must be reflected by a package dependency. This package dependency is indicated in package diagrams using the graphic notation as in Figure 2.



Figure 2 — UML example package dependency

Because of this client-server relation, inter-package dependencies define the criterion for viable application schemas. An application schema that contains an implementation of any package defined from this International Standard shall also contain implementations all of its dependencies.

Table 6 summarizes the packages specified in this International Standard. Packages in Clauses 6 and 7 are normative. They provide the geometry and topology components for an application schema that can form the basis for the external interface for compliant systems. Additional packages are referenced from other standards, such as the Spatial referencing by Coordinates package from ISO 19111 and the Basic Types package from ISO TS 19103. They are replicated here to the extent needed to provide a complete and readable picture of potential spatial schemas.

Table 6 — Package and classes

Clause number	Package Name	Major classes included
6	Geometry	geometry classes
6.2	Geometry root	root classes for geometry
6.3	Geometric primitive	geometric primitives
6.4	Coordinate geometry	coordinate geometry classes
6.5	Geometric aggregates	aggregates
6.6	Geometric complex	geometric complexes and composites
7	Topology	topology classes
7.2	Topology root	root classes for topology
7.3	Topological primitive	topological primitives
7.4	Topological complex	topological complexes

Figure 3 shows the leaf packages for the normative clauses of this International Standard.

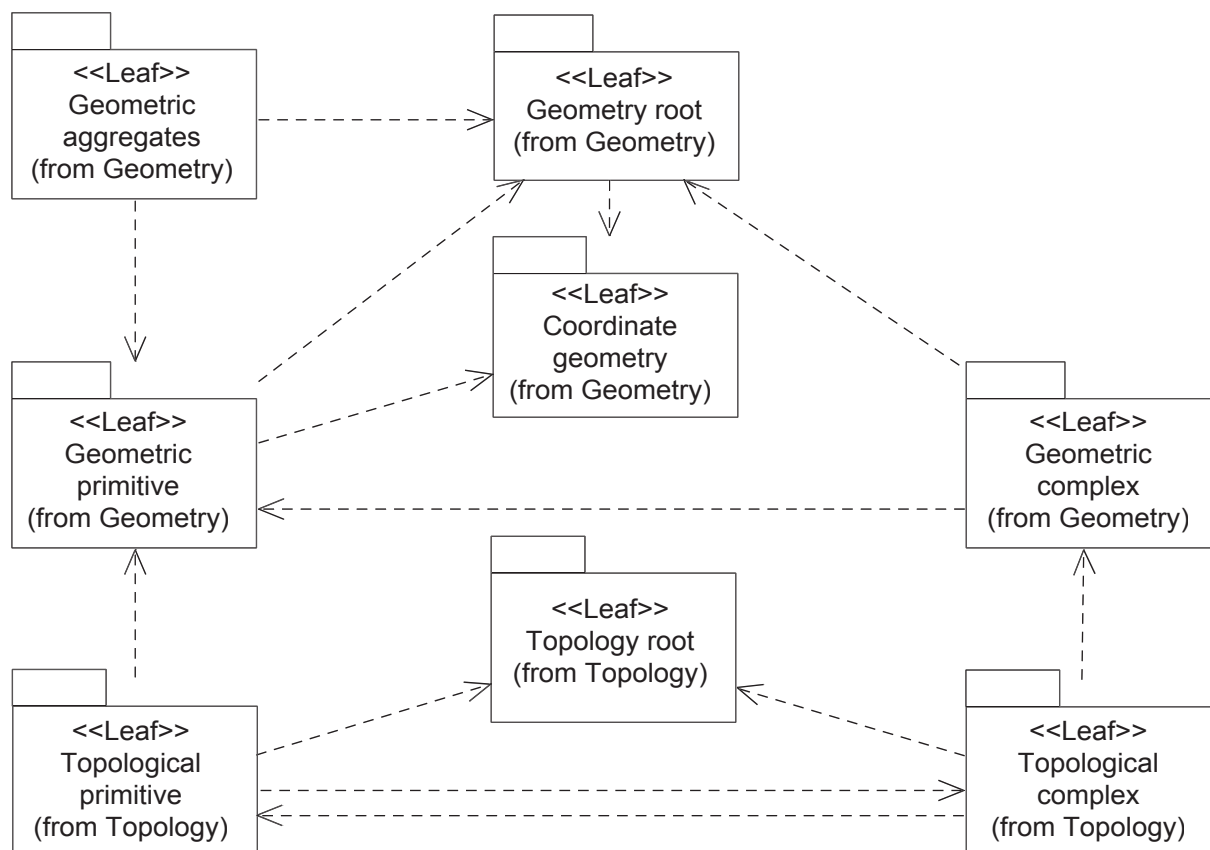


Figure 3 — Normative clause as UML package dependencies

NOTE Examples in the text are given where they are most appropriate to understanding the items in this International Standard, and, as such, often use implicit forward references to other items discussed later in the document. For example, the discussion of `GM_Envelope` includes a forward reference to `GM_LineString`. In most cases, this is not confusing, since the item (type, operation, or attribute) of the forward reference is often semantically rich and corresponds closely to a commonly used term. If the reader finds this confusing, it is suggested that the entire document be read skipping the examples to establish an overview, and then reread carefully to include the examples.

5.3 Abbreviated terms

ATS	Abstract test suite
API	Application program interface
C++	Programming language based on C with object-oriented extensions
LISP	Programming language based on LISt Processing. The standard is called Common LISP.
MBR	Minimum Bounding Region
OCL	Object Constraint Language
SQL 3	Common name for SQL 99 during its development
SQL 99	The SQL language specification adopted in 1999, which includes object-oriented data-type extension mechanisms
TIN	Triangulated Irregular Network
UML	Unified Modelling Language
2D	2-dimensional
3D	3-dimensional

6 Geometry packages

6.1 Semantics

The geometry packages (Figure 4) contain the various classes for coordinate geometry. All of these classes through the root class `GM_Object` inherit an optional association to a coordinate reference system. All direct positions exposed through the interfaces defined in this International Standard shall be in the coordinate reference system of the geometric object accessed. All elements of a geometric complex, composite, or aggregate shall be associated to the same coordinate reference system. When instances of `GM_Object` are aggregated in another `GM_Object` (such as a `GM_Aggregate`, or `GM_Complex`) which already has a coordinate reference system specified, then these elements are assumed to be in that same coordinate reference system unless otherwise specified.

The geometry package has several internal packages that separate primitive geometric objects, aggregates and complexes, which have a more elaborate internal structure than simple aggregates. Figure 4 shows the dependencies between the geometry packages as well as a list of classes for each package.

Figure 5 shows the basic classes defined in the geometry packages. Any object that inherits the semantics of the `GM_Object` acts as a set of direct positions. Its behaviour will be determined by which direct positions it contains. Objects under `GM_Primitive` will be open, that is, they will not contain their boundary points; curves will not contain their end points, surfaces will not contain their boundary curves, and solids will not contain their bounding surfaces. Objects under `GM_Complex` will be closed, that is, they will contain their boundary points. This leads to some apparent ambiguity. A representation of a line as a primitive must reference its end points, but will not contain these points as a set of direct positions. A representation of a line as a complex will also reference its end points, and will contain these points as a set of direct positions. This means that identical digital representations will have slightly different semantics depending on whether they are accessed as primitives or complexes.

This difference of semantics is most striking in the `GM_CompositeCurve`. Composite curves are used to represent features whose geometry could also be represented as curve primitives. From a cartographic point of view, these two representations are not different. From a topological point of view, they are different. This distinction appears in the inheritance diagram (Figure 5) as an inheritance relationship between `GM_CompositeCurve` and `GM_OrientableCurve`. The primary semantics of a `GM_CompositeCurve` (see 6.6.5) is as a closed `GM_Object`, but it may also act as an open `GM_Object` under `GM_Primitive` operations (see 6.3.10). Interface protocols depending upon the topological details of this object will have to be distinguished as to whether they have been inherited from `GM_Primitive` or `GM_Complex`, where the distinction first occurs. Even though these protocols have been inherited from the same operations defined at

GM_Object, they will act differently depending upon the branch of the inheritance tree from which they have inherited semantics. Creators of implementation profiles may take this into account and use a proxy mechanism for realization relationships that cause semantic dissonance. Such a procedure will be necessary in object-oriented programming and databases in systems that disallow multiple inheritance or make limiting assumptions about method binding.

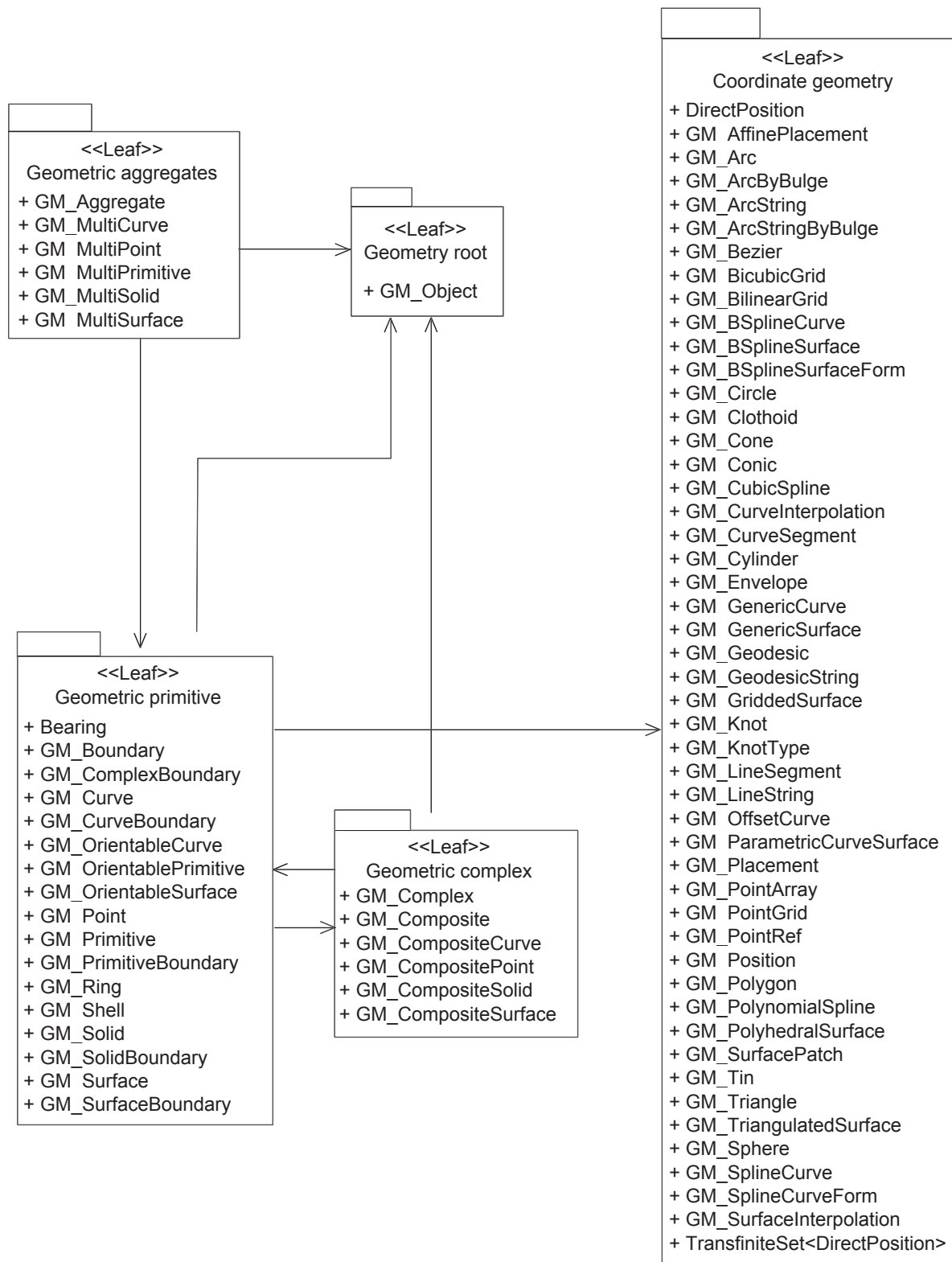


Figure 4 — Geometry package: Class content and internal dependencies

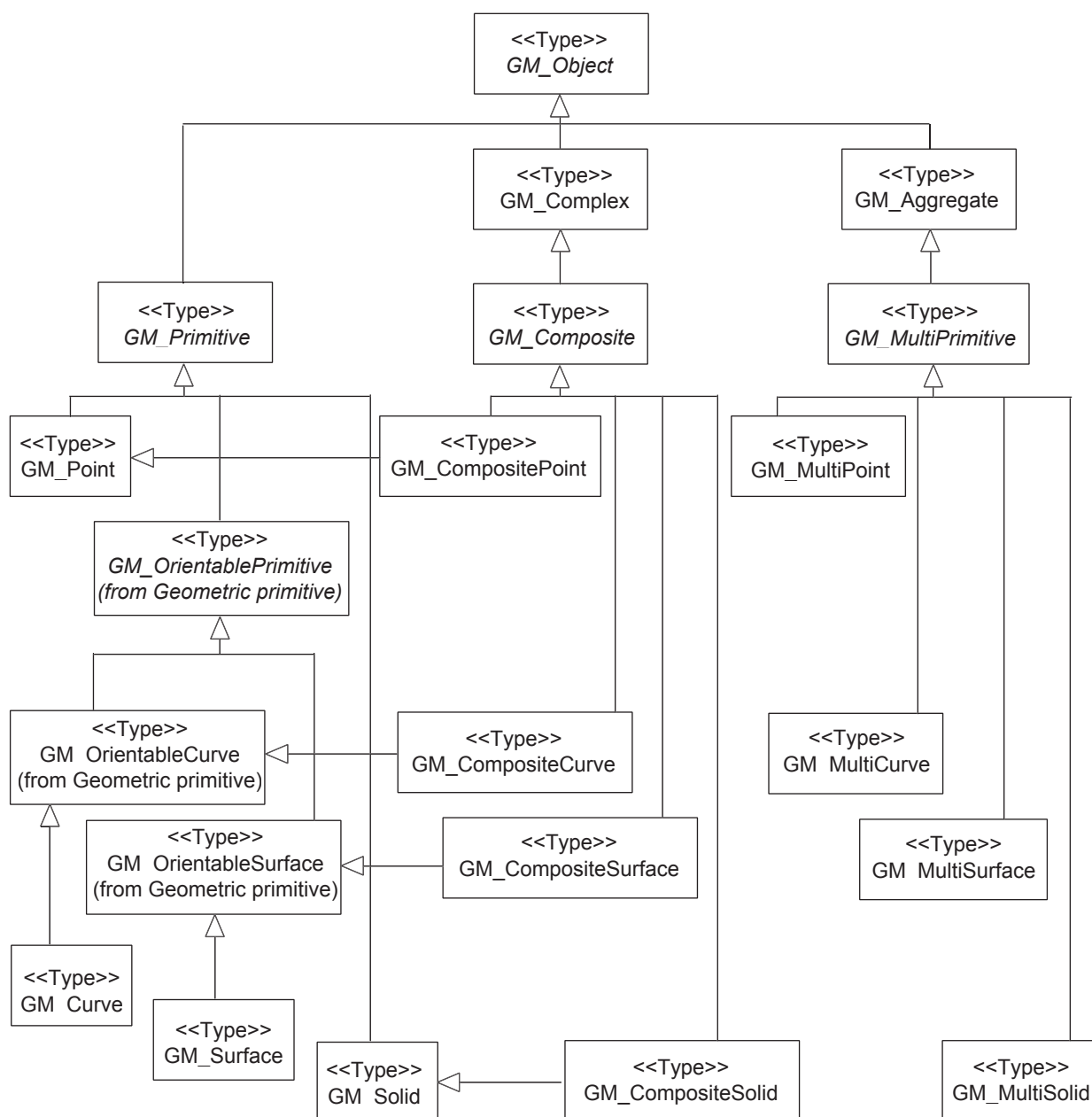


Figure 5 — Geometry basic classes with specialization relations

6.2 Geometry root package

6.2.1 Semantics

A geometric object shall be a combination of a coordinate geometry and a coordinate reference system. In all of the operations, all geometric calculations shall be done in the coordinate reference system of the first geometric object accessed, which is normally the object whose operation is being invoked. Returned objects shall be in the coordinate reference system in which the calculations are done unless explicitly stated otherwise. The interface protocols defined in this section are basically those of set theory. In general a geometric object is a set of geometric points, represented by `DirectPosition` (see 6.4.1). Object instantiations of geometric objects are `GM_Objects`. Object instantiations of geometric points, when used as values, are `DirectPositions`. General set theory operations defined at `GM_Object` differentiate further down the class hierarchy depending on whether or not the boundary `DirectPositions` are included as set elements. Subtypes of `GM_Primitive` do not contain boundary points, while subtypes of `GM_Complex` do.

GM_Object and GM_Primitive are purely abstract in the sense that no object or data structure from an application schema can instantiate them directly. Instances of these classes must be instances of one of their non-abstract subtypes, such as GM_Point, GM_Curve, or GM_Surface. This is not the case for GM_Complex, which can be directly instantiated by an application schema, and need not be an instance of one of the non-abstract subclasses of GM_Composite. Although GM_Complex is not explicitly implemented by this International Standard, it would be valid for an application schema to include a concrete class called “GM_Complex” in a class library conformant to this International Standard. Recall that the name space of the application schema is different from that of the standard and such seemingly logical abuses of name are valid. This is not the case for the abstract classes within this International Standard. These classes are logically incapable of supporting an implementation directly. Constructors on these classes result in instances of concrete subclasses of these types, not in direct logical instances of the abstract type.

This is a stricter interpretation of “abstract” than is commonly used in UML, but it is appropriate here as a guide to application schema developers.

6.2.2 GM_Object

6.2.2.1 Semantics

GM_Object (Figure 6) is the root class of the geometric object taxonomy and supports interfaces common to all geographically referenced geometric objects. GM_Object instances are sets of direct positions in a particular coordinate reference system. A GM_Object can be regarded as an infinite set of points that satisfies the set operation interfaces for a set of direct positions, TransfiniteSet<DirectPosition>. Since an infinite collection class cannot be implemented directly, a Boolean test for inclusion shall be provided by the GM_Object interface. This International Standard concentrates on vector geometry classes, but future work may use GM_Object as a root class without modification.

NOTE As a type, GM_Object does not have a well-defined default state or value representation as a data type. Instantiated subclasses of GM_Object will.

6.2.2.2 mbRegion

The operation “mbRegion” is included here only as an interface, as different applications may choose to implement it in different ways. It shall return a region in the coordinate reference system that contains this GM_Object. The default shall be to return an instance of an appropriate GM_Object subclass that represents the same spatial set returned from the operator “GM_Object::envelope”. The most common use of mbRegion will be to support indexing methods that use extents other than minimum bounding rectangles (MBR or envelopes).

```
GM_Object::mbRegion() : GM_Object
```

This does not restrict the returned GM_Object from being a non-vector geometric representation, although those types are not defined within this International Standard.

6.2.2.3 representativePoint

The operation “representativePoint” is included here only as an interface that may be implemented in different ways. It shall return a point value (DirectPosition) that is guaranteed to be on this GM_Object. The default logic may be to use the DirectPosition of the point returned by the operation “GM_Object::centroid” if that point is on the object.

```
GM_Object::representativePoint() : DirectPosition
```

Another use of representativePoint may be for the placement of labels in systems based on graphic presentation. Definitions for symbology and type placement are outside the scope of this International Standard.

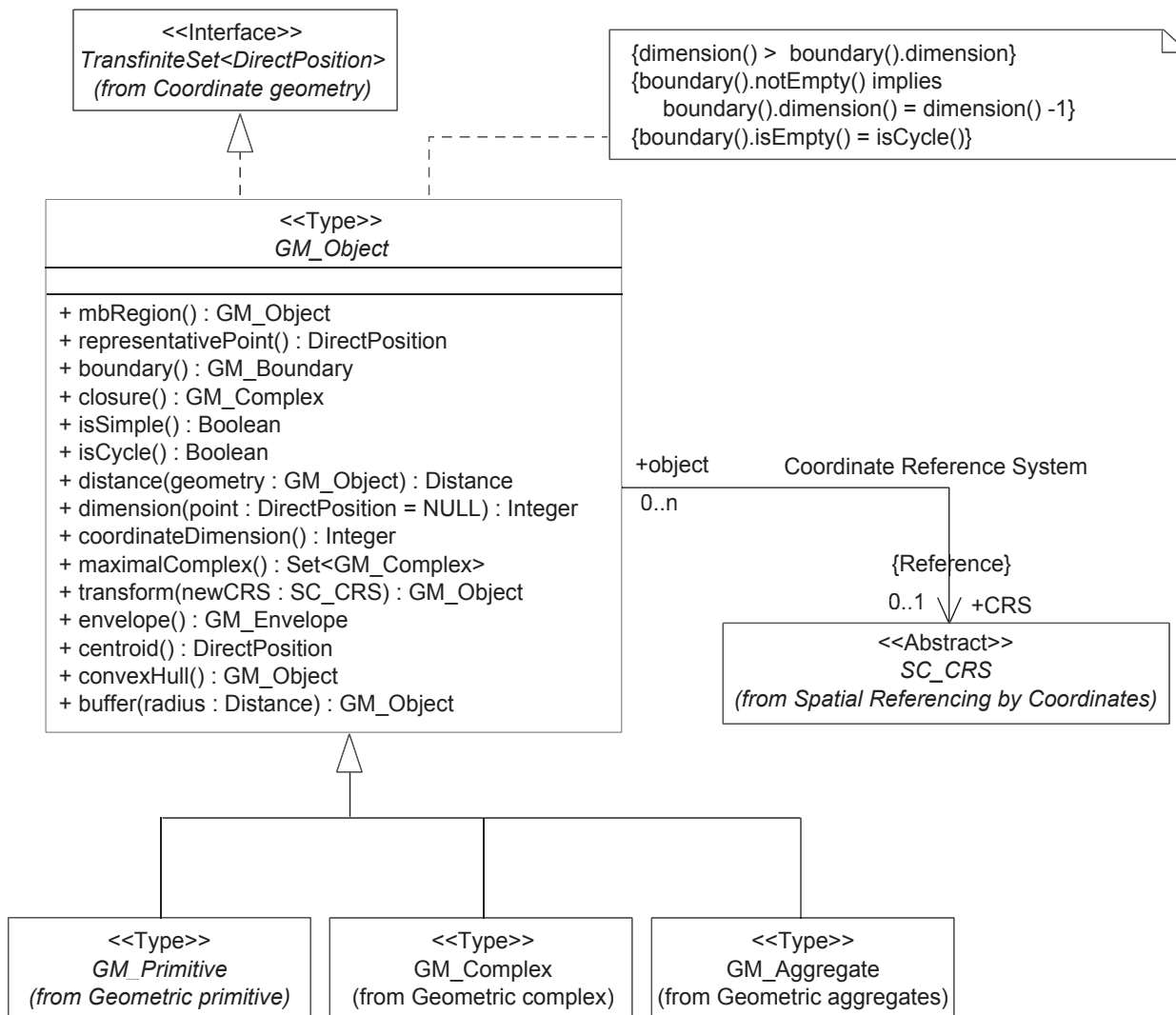


Figure 6 — GM_Object

6.2.2.4 boundary

The operation “boundary” shall return a finite set of GM_Objects containing all of the direct positions on the boundary of this GM_Object. These object collections shall have further internal structure where appropriate, and shall be represented as subclasses of the datatype GM_Boundary that is a subtype of GM_Complex. The finite set of GM_Objects returned shall be in the same coordinate reference system as this GM_Object. If the GM_Object is in a GM_Complex, then the boundary GM_Objects returned shall be in the same GM_Complex. If the GM_Object is not in any GM_Complex, then the boundary GM_Objects returned may have been constructed in response to the operation.

```
GM_Object::boundary() : GM_Boundary
```

The organization of the set returned is dependent on the type of GM_Object. Each of the subclasses of GM_Object described below specifies the organization of its boundary set more completely.

The elements of a boundary shall be smaller in dimension than the original element.

```
-- all objects in the boundary are of at least 1 dimension smaller
-- than the originalGM_Object:
boundary->select(dimension) <= self.dimension - 1
```

6.2.2.5 closure

The operation “closure” shall return a finite set of GM_Objects containing all of the points on the boundary of this GM_Object and this object (the union of the object and its boundary). These object collections shall have further internal structure where appropriate. The finite set of GM_Objects returned shall be in the same coordinate reference system as this GM_Object. If the GM_Object is in a GM_Complex, then the boundary GM_Objects returned shall be in the same GM_Complex. If the GM_Object is not in any GM_Complex, then the boundary GM_Objects returned may have been constructed in response to the operation.

```
GM_Object::closure() : GM_Complex
```

6.2.2.6 isSimple

The operation “isSimple” shall return TRUE if this GM_Object has no interior point of self-intersection or self-tangency. In mathematical formalisms, this means that every point in the interior of the object must have a metric neighborhood whose intersection with the object is isomorphic to an n-sphere, where n is the dimension of this GM_Object.

```
GM_Object::isSimple() : Boolean
```

Since most coordinate geometries are represented, either directly or indirectly by functions from regions in Euclidean space of their topological dimension, the easiest test for simplicity to require that a function exist that is 1-to-1 and bicontinuous (continuous in both directions). Such a function is a topological isomorphism. This test does not work for “closed” objects (that is, objects for which the isCycle operation returns TRUE).

While GM_Complexes shall contain only simple GM_Objects, non-simple GM_Objects are often used in “spaghetti” data sets.

NOTE “Spaghetti” is a pejorative (uncomplimentary) term, usually indicative of an unacceptable level of geometric anomalies and inconsistencies in the data that must be “cleaned” before use is made of it. Such inconsistencies can include (but are not limited to) any or all of the following anomaly types:

- 1) An undershot line is a line that should intersect another, but falls short leaving a small gap between it and the point of intersection. This is often hard to distinguish from real “near misses” between features (such as where a road is separated from another by a wall only one brick thick). This problem is especially difficult to handle when the undershoot fails to close a surface or polygon boundary. This is often indicative of the digitizer working at too small a scale and failing to “snap” to the end of lines.
- 2) An overshoot line is a line that should intersect and terminate at another, but goes too far, leaving a small excess line on the far side of the point of intersection. This is often indicative of the digitizer working at too small a scale and trying to visually “snap” the end of lines.
- 3) End loop (a line that should intersect and terminate at another, but goes too far and then returns, leaving a small excess loop on the far side of the point of intersection. This is often indicative of the digitizer working at too small a scale and “snapping” a line after he has already overshoot it.
- 4) Slivers and gaps are multiple lines that should represent the same geometry, but do not coincide, leaving areas of overlap between two surface boundaries (slivers), and gaps between them. This problem is particularly difficult to deal with in areas of braided streams, where the real geometry of the natural feature resembles the sliver and gaps of simple bad digitization practice. This is often indicative of multiple sources for the same data, which have been merged (but not properly conflated), into the same database.

The real problem with “spaghetti” comes in that the heuristics (either manual or automated) used to correct the problems often result in additional, but different factual errors. This can be a severe quality issue for geometry.

6.2.2.7 isCycle

The operation “isCycle” shall return TRUE if this GM_Object has an empty boundary after topological simplification (removal of overlaps between components in non-structured aggregates, such as subclasses of GM_Aggregate). This condition is alternatively referred to as being “closed” as in a “closed curve.” This creates some confusion since there are two distinct and incompatible definitions for the word “closed”. The use of the word cycle is rarer (generally restricted to the field of algebraic topology), but leads to less confusion. Essentially, an object is a cycle if it is isomorphic to a geometric object that is the boundary of a region in some Euclidean space. Thus a point is a cycle as a boundary of it is empty. A curve is a cycle if it is isomorphic to a circle (has the same start and end point). A surface is a cycle if it is isomorphic to the surface of a sphere, or some torus. A solid, with finite size, in a space of dimension 3 is never a cycle.

```
GM_Object::isCycle() : Boolean
```

EXAMPLE The following OCL uses the boundary operator to produce a GM_Object and then tests for an empty set using the operator TransfiniteSet<DirectPosition>::isEmpty().

```
GM_Object:
    isCycle() = boundary().isEmpty()
```

6.2.2.8 distance

The operation “distance” shall return the distance between this GM_Object and another GM_Object. This distance is defined to be the greatest lower bound of the set of distances between all pairs of points that include one each from each of the two GM_Objects. A “distance” value shall be a non-negative number associated to a distance unit such as meter or standard foot. If necessary, the second geometric object shall be transformed into the same coordinate reference system as the first before the distance is calculated.

```
GM_Object::distance(geometry : GM_Object) : Distance
```

If the geometric objects overlap, or touch, then their distance apart shall be zero. Some current implementations use a “negative” distance for such cases, but the approach is neither consistent between implementations, nor theoretically viable.

“Distance” is one of the units of measure data types defined in ISO TS 19103.

NOTE The role of the reference system in distance calculations is important. Generally, there are at least three types of distances that may be defined between points (and therefore between geometric objects): map distance, geodesic distance, and terrain distance.

Map distance is the distance between the points as defined by their positions in a coordinate projection (such as on a map when scale is taken into account). Map distance is usually accurate for small areas where scale functions have well-behaved derivatives.

Geodesic distance is the length of the shortest curve between those two points along the surface of the Earth model being used by the coordinate reference system. Geodesic distance behaves well for wide areas of coverage, and takes the earth's curvature into account. It is especially handy for air and sea navigation, although care should be taken to distinguish between rhumb line (curves of constant bearing) and geodesic curve distance.

Terrain distance takes into account the local vertical displacements (hypsography). Terrain distance can be based either on a geodesic distance or a map distance.

6.2.2.9 dimension

The operation “dimension” shall return the inherent dimension of this GM_Object, which shall be less than or equal to the coordinate dimension. The dimension of a collection of geometric objects shall be the largest dimension of any of its pieces. Points are 0-dimensional, curves are 1-dimensional, surfaces are 2-dimensional, and solids are 3-dimensional. Locally, the dimension of a geometric object at a point is the

dimension of a local neighborhood of the point – that is the dimension of any coordinate neighborhood of the point. Dimension is unambiguously defined only for DirectPositions interior to this GM_Object. If the passed DirectPosition is NULL, then the operation shall return the largest possible dimension for any DirectPosition in this GM_Object.

```
GM_Object::dimension(point : DirectPosition = NULL) : Integer
```

6.2.2.10 coordinateDimension

The operation “coordinateDimension” shall return the dimension of the coordinates that define this GM_Object, which must be the same as the coordinate dimension of the coordinate reference system for this GM_Object.

```
GM_Object::coordinateDimension() : Integer
```

6.2.2.11 maximalComplex

As a set of primitives, a GM_Complex may be contained as a set in another larger GM_Complex, referred to as a “super complex” of the original. A GM_Complex is maximal if there is no such larger super complex. The operation “maximalComplex” shall return the set of maximal GM_Complexes within which this GM_Object is contained.

```
GM_Object::maximalComplex() : Set<GM_Complex>
```

If the application schema used does not include GM_Complex, then this operation shall return a NULL value.

NOTE The usual semantics of maximal complexes does not allow any GM_Primitive to be in more than one maximal complex, making it a strong aggregation. This is not an absolute, and depending on the semantics of the implementation, the association between GM_Primitives and maximal GM_Complexes could be many to many. From a programming point of view, this would be a difficult (but not impossible) dynamic structure to maintain, but as a static query-only structure, it could be quite useful in minimizing redundant data inherent in two representations of the same primitive geometric object.

6.2.2.12 transform

The operation “transform” shall return a new GM_Object that is the coordinate transformation of this GM_Object into the passed coordinate reference system within the accuracy of the transformation.

```
GM_Object::transform(newCRS : SC_CRS) : GM_Object
```

6.2.2.13 envelope

The operation “envelope” shall return the minimum bounding box for this GM_Object. This shall be the coordinate region spanning the minimum and maximum value for each ordinate taken on by DirectPositions in this GM_Object. The simplest representation for an envelope consists of two DirectPositions, the first one containing all the minimums for each ordinate, and second one containing all the maximums. However, there are cases for which these two positions would be outside the domain of validity of the object's coordinate reference system. This operation is included here only as an interface, as applications may choose to implement in different manners.

```
GM_Object::envelope() : GM_Envelope
```

6.2.2.14 centroid

The operation “centroid” shall return the mathematical centroid for this GM_Object. The result is not guaranteed to be on the object. For heterogeneous collections of primitives, the centroid only takes into account those of the largest dimension. For example, when calculating the centroid of surfaces, an average is taken weighted by area. Since curves have no area they do not contribute to the average.

```
GM_Object::centroid() : DirectPosition
```

NOTE There may be cases for which this position would be outside the domain of validity of the object's coordinate reference system, but this is unlikely, since the domain of validity of most coordinate reference systems is convex. If this unlikely case should arise the implementation shall decide on appropriate action.

6.2.2.15 convexHull

The operation “convexHull” shall return a GM_Object that represents the convex hull of this GM_Object.

```
GM_Object::convexHull() : GM_Object
```

NOTE There may be cases for which this GM_Object would be partially outside the domain of validity of the object's coordinate reference system, but this is unlikely, since the domain of validity of most coordinate reference systems is convex. If this unlikely case should arise the implementation shall decide on appropriate action.

Convexity requires the use of “lines” or “curves of shortest length” and the use of different coordinate systems may result in different versions of the convex hull of an object. Each implementation shall decide on an appropriate solution to this ambiguity. For two reasonable coordinate systems, a convex hull of an object in one will be very closely approximated by the transformed image of the convex hull of the same object in the other.

6.2.2.16 buffer

The operation “buffer” shall return a GM_Object containing all points whose distance from this GM_Object is less than or equal to the “distance” passed as a parameter. The GM_Object returned is in the same reference system as this original GM_Object. The dimension of the returned GM_Object is normally the same as the coordinate dimension - a collection of GM_Surfaces in 2D space and a collection of GM_Solids in 3D space, but this may be application defined.

```
GM_Object::buffer(radius : Distance) : GM_Object
```

NOTE There are cases for which this GM_Object would be partially outside the domain of validity of the object's coordinate reference system. If this case should arise the implementation shall decide on appropriate action.

6.2.2.17 Coordinate Reference System association

The association role “Coordinate Reference System::CRS” links this GM_Object to the coordinate reference system used in its DirectPosition coordinates. If this association is empty, then the GM_Object uses the SC_CRS from another GM_Object in which it is contained.

```
GM_Object::CRS[0,1] : SC_CRS
```

NOTE The most common example where this association can be empty is the elements and subcomplexes of a maximal GM_Complex. The GM_Complex can carry the SC_CRS for all GM_Primitive elements and for all GM_Complex subcomplexes. This association is only navigable from GM_Object to SC_CRS. This means that the coordinate reference system objects in a data set do not keep a list of GM_Objects that use them.

6.2.2.18 Operations from TransfiniteSet realization

6.2.2.18.1 Semantics

The `TM_Object` realizes the following operations from the Interface `TransfiniteSet<DirectPosition>` (ISO/TS 19103).

6.2.2.18.2 contains

The Boolean valued operation “contains” shall return TRUE if this `GM_Object` contains another `GM_Object`, or a single point given by a coordinate (`DirectPosition`). The purpose of this operator is to instantiate `TransfiniteSet<DirectPosition>::contains`.

```
GM_Object::contains(pointSet : GM_Object) : Boolean // subset operator
GM_Object::contains(point : DirectPosition) : Boolean // element containment
operator
```

If the passed `GM_Object` is a `GM_Point`, then this operation is the equivalent of a set-element test for the `DirectPosition` of that point within this `GM_Object`. Since point and other geometric objects share a common ancestor (`GM_Object`), it is not normally necessary to differentiate between point containment and set containment for `GM_Object`. The following OCL reiterates basic set theory axioms.

```
GM_Object:
contains(that : GM_Object) and that.contains(other : GM_Object) implies
    contains(other)
contains(that : GM_Object) and that.contains(p : DirectPosition) implies
    contains(p)
contains(point : GM_Point) implies contains(point.position)
```

NOTE “Contains” is strictly a set theoretic containment, and has no dimensionality constraint. In a `GM_Complex`, no `GM_Primitive` will contain another unless a dimension is skipped. See 6.3.11.3.

6.2.2.18.3 intersects

The Boolean valued operation “intersects” shall return TRUE if this `GM_Object` intersects another `GM_Object`. The purpose of this operator is to instantiate `TransfiniteSet<DirectPosition>::intersects`.

```
GM_Object::intersects(pointSet : GM_Object) : Boolean
```

Within a `GM_Complex`, the `GM_Primitives` do not intersect one another. In general, topologically structured data uses shared geometric objects to capture intersection information.

NOTE This intersect is strictly a set theoretic common containment of `DirectPositions`. Two `GM_Curves` (under `GM_Primitive`) do not intersect if they share a common end point because `GM_Primitives` are considered to be open (do not contain their boundary). If two `GM_CompositeCurves` (under `GM_Complex`) share a common end point, then they intersect because `GM_Complexes` are considered to be closed (contain their boundary).

6.2.2.18.4 equals

The Boolean valued operation “equals” shall return TRUE if this `GM_Object` is equal to another `GM_Object`. The purpose of this operator is to instantiate `TransfiniteSet<DirectPosition>::equals`.

```
GM_Object::equals(pointSet : GM_Object) : Boolean
```

Two different `GM_Object`s are equal if they return the same Boolean value for the operation `GM_Object::contains` for every tested `DirectPosition` within the valid range of the coordinate reference system associated to the object.

NOTE Since an infinite set of direct positions cannot be tested, the internal implementation of equal must test for equivalence between two, possibly quite different, representations. This test may be limited to the resolution of the coordinate system or the accuracy of the data. Application schemas may define a tolerance that returns true if the two `GM_Object`s have the same dimension and each direct position in this `GM_Object` is within a tolerance distance of a direct position in the passed `GM_Object` and vice versa.

6.2.2.18.5 union

The “union” operation shall return the set theoretic union of this `GM_Object` and the passed `GM_Object`.

The purpose of union is to instantiate `TransfiniteSet<DirectPosition>::union`.

```
GM_Object::union(pointSet : GM_Object) : GM_Object
```

6.2.2.18.6 intersection

The “intersection” operation shall return the set theoretic intersection of this `GM_Object` and the passed `GM_Object`.

The purpose of intersection is to instantiate `TransfiniteSet<DirectPosition>::intersection`.

```
GM_Object::intersection(pointSet : GM_Object) : GM_Object
```

6.2.2.18.7 difference

The “difference” operation shall return the set theoretic difference of this `GM_Object` and the passed `GM_Object`.

The purpose of difference is to instantiate `TransfiniteSet<DirectPosition>::difference`.

```
GM_Object::difference(pointSet : GM_Object) : GM_Object
```

NOTE The difference operation is not symmetric and `A.difference(B)` is usually not the same as `B.difference(A)`.

6.2.2.18.8 symmetricDifference

The “symmetricDifference” operation shall return the set theoretic symmetricDifference of this `GM_Object` and the passed `GM_Object`. The purpose of symmetricDifference is to instantiate `TransfiniteSet<DirectPosition>::symmetricDifference`.

```
GM_Object::symmetricDifference(pointSet : GM_Object) : GM_Object
```

6.3 Geometric primitive package

6.3.1 Semantics

The Geometric primitive package contains all the geometric primitives and supporting data types used in describing their boundaries.

6.3.2 GM_Boundary

The abstract root data type for all the data types used to represent the boundary of geometric objects is GM_Boundary (Figure 7). Any subclass of GM_Object will use a subclass of GM_Boundary to represent its boundary through the operation GM_Object::boundary. By the nature of geometry, boundary objects are cycles.

```
GM_Boundary:
    {isCycle() = TRUE}
```

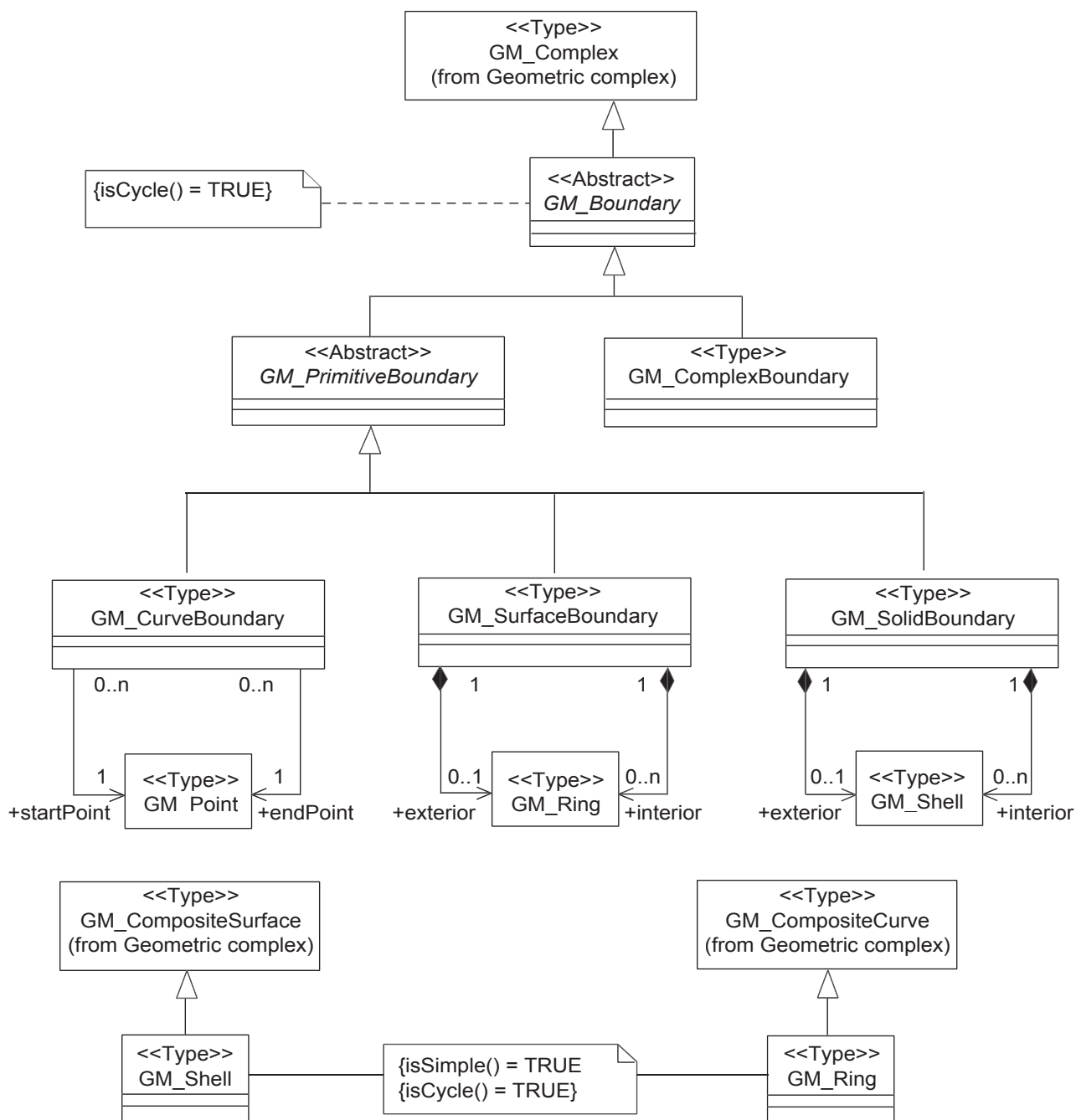


Figure 7 — GM_Boundary

6.3.3 GM_ComplexBoundary

The boundary operation for GM_Complex objects shall return a GM_ComplexBoundary, which is a collection of primitives and a GM_Complex of dimension one less than the original object.

6.3.4 GM_PrimitiveBoundary

The abstract class GM_PrimitiveBoundary is the root for the various return types of the boundary operator for subtypes of GM_Primitive. Since points have no boundary, no special subclass is needed for their boundary.

6.3.5 GM_CurveBoundary

6.3.5.1 Semantics

The boundary of GM_Curves shall be represented as GM_CurveBoundary.

6.3.5.2 startPoint, endPoint

A GM_CurveBoundary contains two GM_Point references.

```
GM_CurveBoundary::startPoint : Reference<GM_Point>;  
GM_CurveBoundary::endPoint : Reference<GM_Point>;
```

6.3.6 GM_Ring

A GM_Ring is used to represent a single connected component of a GM_SurfaceBoundary. It consists of a number of references to GM_OrientableCurves connected in a cycle (an object whose boundary is empty).

A GM_Ring is structurally similar to a GM_CompositeCurve in that the endPoint of each GM_OrientableCurve in the sequence is the startPoint of the next GM_OrientableCurve in the Sequence. Since the sequence is circular, there is no exception to this rule. Each ring, like all boundaries is a cycle and each ring is simple.

```
GM_Ring:  
    {isSimple() = TRUE}
```

Even though each GM_Ring is simple, the boundary need not be simple. The easiest case of this is where one of the interior rings of a surface is tangent to its exterior ring. Implementations may enforce stronger restrictions on the interaction of boundary elements.

6.3.7 GM_SurfaceBoundary

6.3.7.1 Semantics

The boundary of GM_Surfaces shall be represented as GM_SurfaceBoundary.

6.3.7.2 exterior, interior

A GM_SurfaceBoundary consists of some number of GM_Rings, corresponding to the various components of its boundary. In the normal 2D case, one of these rings is distinguished as being the exterior boundary. In a general manifold this is not always possible, in which case all boundaries shall be listed as interior boundaries, and the exterior will be empty.

```
GM_SurfaceBoundary::exterior[0,1] : GM_Ring;  
GM_SurfaceBoundary::interior[0..n] : GM_Ring;
```

NOTE The use of exterior and interior here is not intended to invoke the definitions of “interior” and “exterior” of geometric objects. The terms are in common usage, and reflect a linguistic metaphor that uses the same linguistic constructs for the concept of being inside an object to being inside a container. In normal mathematical terms, the exterior boundary is the one that appears in the Jordan Separation Theorem (Jordan Curve Theorem extended beyond 2D). The exterior boundary is the one that separates the surface (or solid in 3D) from infinite space. The interior boundaries separate the object at hand from other bounded objects. The uniqueness of the exterior comes from the uniqueness of unbounded space. Essentially, the Jordan Separation Theorem shows that normal 2D or 3D space separates into bounded and unbounded pieces by the insertion of a ring or shell, respectively. It goes beyond that, but this International Standard is restricted to at most three dimensions.

EXAMPLE 1 If the underlying manifold is an infinite cylinder, then two transverse cuts of the cylinder define a compact surface between the cuts, and two separate unbounded portions of the cylinders. In this case, either cut could reasonably be called exterior. In cases of such ambiguity, the International Standard chooses to list all boundaries in the “interior” set. The only guarantee of an exterior boundary being unique is in the 2-dimensional plane, E^2 .

EXAMPLE 2 Taking the equator of a sphere, and generating a 1 m buffer, we have a surface with two isomorphic boundary components. There is no unbiased manner to distinguish one of these as an exterior.

6.3.8 GM_Shell

A `GM_Shell` is used to represent a single connected component of a `GM_SolidBoundary`. It consists of a number of references to `GM_OrientableSurfaces` connected in a topological cycle (an object whose boundary is empty). Unlike a `GM_Ring`, a `GM_Shell`'s elements have no natural sort order. Like `GM_Rings`, `GM_Shells` are simple.

```
GM_Shell:
    {isSimple() = TRUE}
```

6.3.9 GM_SolidBoundary

6.3.9.1 Semantics

The boundary of `GM_Solids` shall be represented as `GM_SolidBoundary`.

6.3.9.2 exterior, interior

`GM_SolidBoundaries` are similar to `GM_SurfaceBoundaries`. In normal 3-dimensional Euclidean space, one shell is distinguished as the exterior. In the more general case, this is not always possible.

```
GM_SolidBoundary::exterior[0,1] : GM_Shell;
GM_SolidBoundary::interior[0..n] : GM_Shell;
```

NOTE An alternative use of solids with no external shell would be to define “complements” of finite solids. These infinite solids would have only interior boundaries. If this International Standard is extended to 4D Euclidean space, or if 3D compact manifolds are used (probably not in geographic information), then other examples of bounded solids without exterior boundaries are possible.

6.3.10 GM_Primitive

6.3.10.1 Semantics

`GM_Primitive` (Figure 8) is the abstract root class of the geometric primitives. Its main purpose is to define the basic “boundary” operation that ties the primitives in each dimension together. A geometric primitive (`GM_Primitive`) is a geometric object that is not decomposed further into other primitives in the system. This includes curves and surfaces, even though they are composed of curve segments and surface patches, respectively. This composition is a strong aggregation: curve segments and surface patches cannot exist outside the context of a primitive.

NOTE Most geometric primitives are decomposable infinitely many times. Adding a centre point to a line may split that line into two separate lines. A new curve drawn across a surface may divide that surface into two parts, each of which is a surface. This is the reason that the normal definition of primitive as “non-decomposable” is not plausible in a geometry model – the only non-decomposable object in geometry is a point.

Any geometric object that is used to describe a feature is a collection of geometric primitives. A collection of geometric primitives may or may not be a geometric complex. Geometric complexes have additional properties such as closure by boundary operations and mutually exclusive component parts.

GM_Primitive and GM_Complex share most semantics, in the meaning of operations, attributes and associations. There is an exception in that a GM_Primitive shall not contain its boundary (except in the trivial case of GM_Point where the boundary is empty), while a GM_Complex shall contain its boundary in all cases. This means that if an instantiated object implements GM_Object operations both as GM_Primitive and as a GM_Complex, the semantics of each set theoretic operation is determined by the its name resolution. Specifically, for a particular object such as GM_CompositeCurve, GM_Primitive::contains (returns FALSE for end points) is different from GM_Complex::contains (returns TRUE for end points). Further, if that object is cast as a GM_Primitive value and as a GM_Complex value, then the two values need not be equal as GM_Objects.

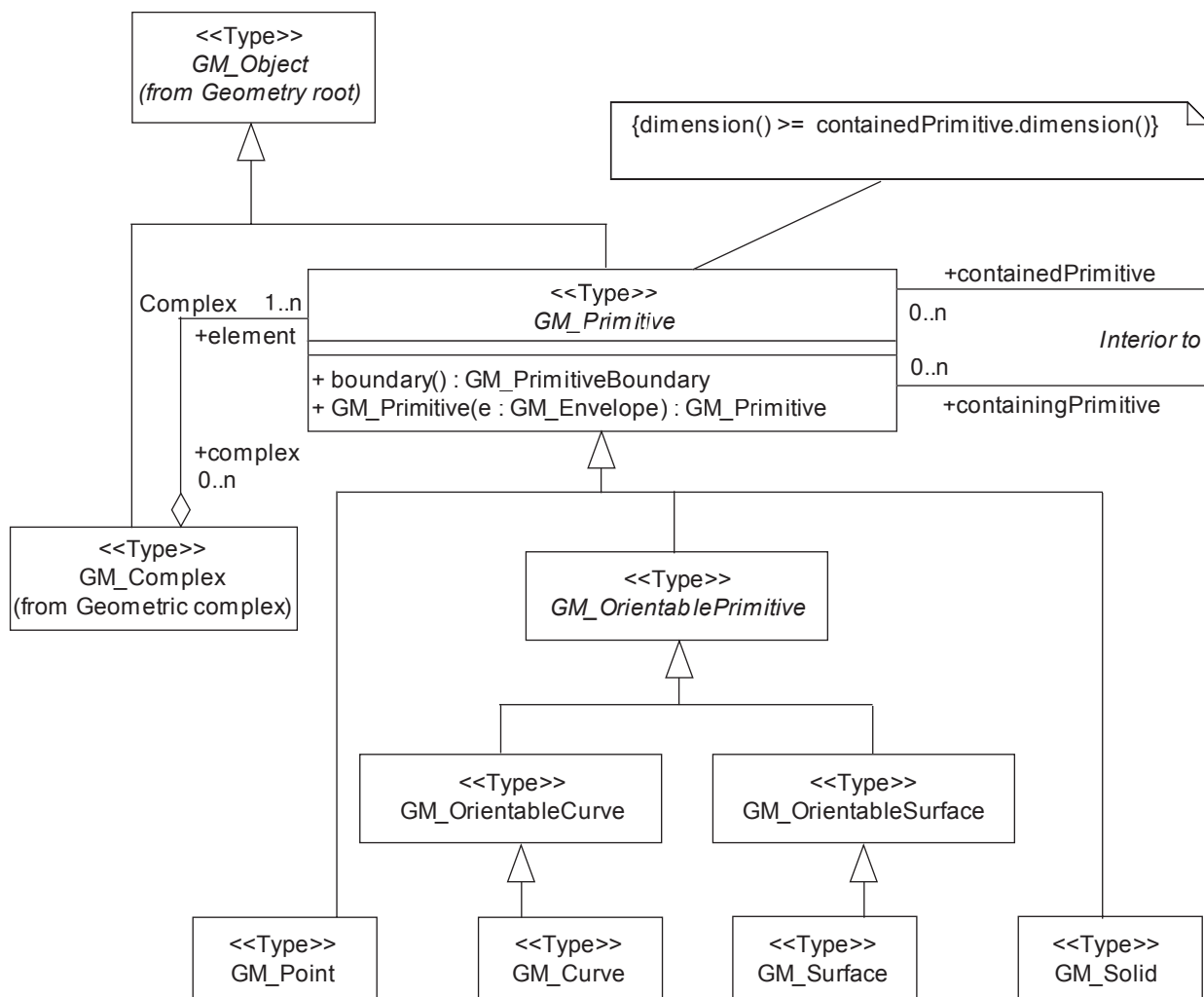


Figure 8 — GM_Primitive

6.3.10.2 boundary

The operation “boundary” shall return the boundary of a GM_Primitive as a set of GM_Primitives. This is a specialization of the operation at GM_Object, which does not restrict the class of the returned collection. The organization of the boundary set of a GM_Primitive depends on the type of the primitive.

```
GM_Primitive::boundary() : GM_PrimitiveBoundary
```

6.3.10.3 GM_Primitive (constructor)

GM_Envelope will often be used in query operations, and therefore must have a cast operation that returns a GM_Object. The constructor at GM_Primitive provides this.

```
GM_Primitive::GM_Primitive(env : GM_Envelope) : GM_Primitive.
```

NOTE The actual return of the operation depends upon the dimension of the coordinate reference system and the extent of the envelope. In a 2D system, the primitive returned will be a GM_Surface (if the envelope does not collapse to a point or line). In 3D systems, the usual return is a GM_Solid.

EXAMPLE In the case where the GM_Envelope is totally contained in the domain of validity of its SC_CRS (coordinate reference system) object, its associated GM_Primitive is the convex hull of the various permutations of the coordinates in the corners. For example, suppose that a particular envelope in 2D is defined as (we ignore the SC_CRS below, assuming that it is a global variable):

```
env : GM_Envelope = <lowerCorner = (x1, y1), upperCorner = (x2, y2)>
```

Then we can take the various permutations of the coordinate values to create a list of polygon corners:

```
multi_point : GM_MultiPoint = { (x1, y1), (x1, y2), (x2, y1), (x2, y2) }
```

If we then apply the convex hull function defined at GM_Object to the multi_point, we get a polygon,

```
multi_point.convexHull () → polygon : GM_Surface
```

The extent of a polygon in 2D is totally defined by its boundary (internal surface patches are planar and do not need interior control points) which gives us a data type to represent GM_Surface in 2D:

```
polygon.boundary → ring : GM_Ring = { string : GM_Linestring =  
  <(x1, y1), (x1, y2), (x2, y2), (x2, y1), (x1, y1)> }
```

So that the GM_SurfaceBoundary record is (convex sets have no “interior” holes):

```
boundary : GM_SurfaceBoundary = < exterior = ring, interior = { } >
```

See the relevant clauses for the formal definition of each of these types.

6.3.10.4 “Interior to” association

The “Interior to” association associates GM_Primitives which are by definition coincident with one another. This allows applications to override the Set<DirectPosition> interpretation and its associated computational geometry, and declare one GM_Primitive to be “interior to” another.

This association should normally be empty when the GM_Primitives are within a GM_Complex, since in that case the boundary information is sufficient for most cases.

```
GM_Primitive::coincidentSubelement [0..n] : Reference<GM_Primitive>
GM_Primitive::superElement [0..n] : Reference<GM_Primitive>
```

This association is constrained by the set theory operators and dimension operators defined at GM_Object.

```
GM_Primitive:
    superElement->includes(p: GM_Primitive) = GM_Object::contains(p)
    dimension() >= coincidentSubelement.dimension()
```

NOTE This association should not be used when the two GM_Primitives are not close to one another. The intent is to allow applications to compensate for inherent and unavoidable round off, truncation, and other mathematical problems indigenous to computer calculations.

6.3.10.5 Complex association

A GM_Primitive may be in several GM_Complexes, see 6.6.2. This association may not be navigable in this direction (from primitive to complex), depending on the application schema.

```
GM_Primitive::complex [0..n] : Reference<GM_Complex>
```

6.3.11 GM_Point

6.3.11.1 Semantics

GM_Point (Figure 9) is the basic data type for a geometric object consisting of one and only one point.

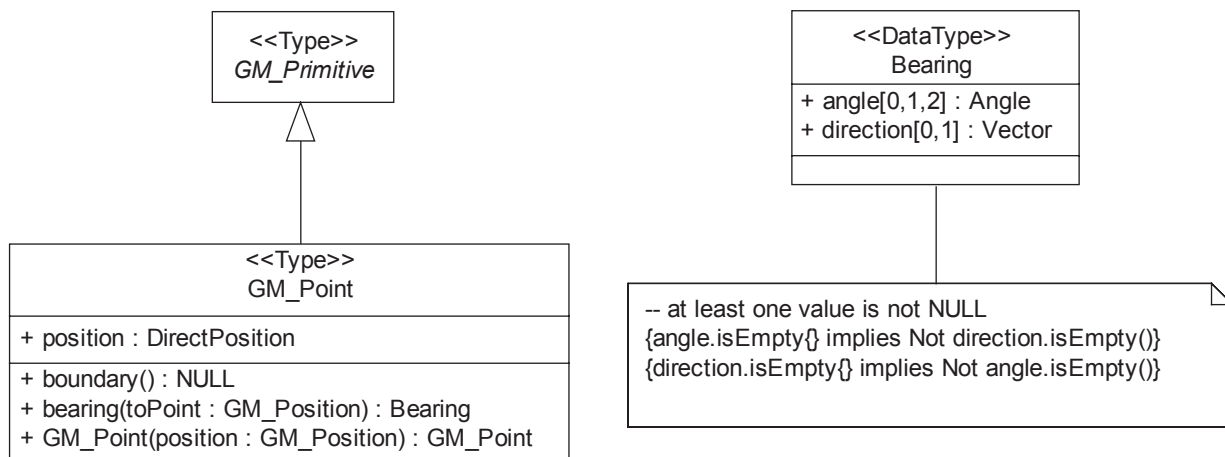


Figure 9 — GM_Point

6.3.11.2 position

The attribute “position” shall be the DirectPosition of this GM_Point.

```
GM_Point::position : DirectPosition
```