INTERNATIONAL STANDARD

ISO 19107

First edition
2003-05-01

Geographic information — Spatial schema

Information géographique — Schéma spatial

Reference number
ISO 19107:2003(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Figures

Tables

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 19107 was prepared by Technical Committee ISO/TC 211, *Geographic information/Geomatics*.

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.

Geographic information — Spatial schema

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 $P(u, v) = A + u\,X + v\,Y$, 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 $a \leqslant j \leqslant b$; [$a..a$] is assumed to mean the same as [a].

g) $\text{parameter}_{\text{list}}$ — 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 than 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.

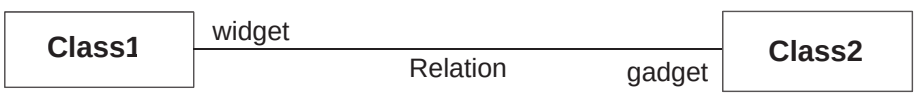


Figure 1 — UML example association

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.

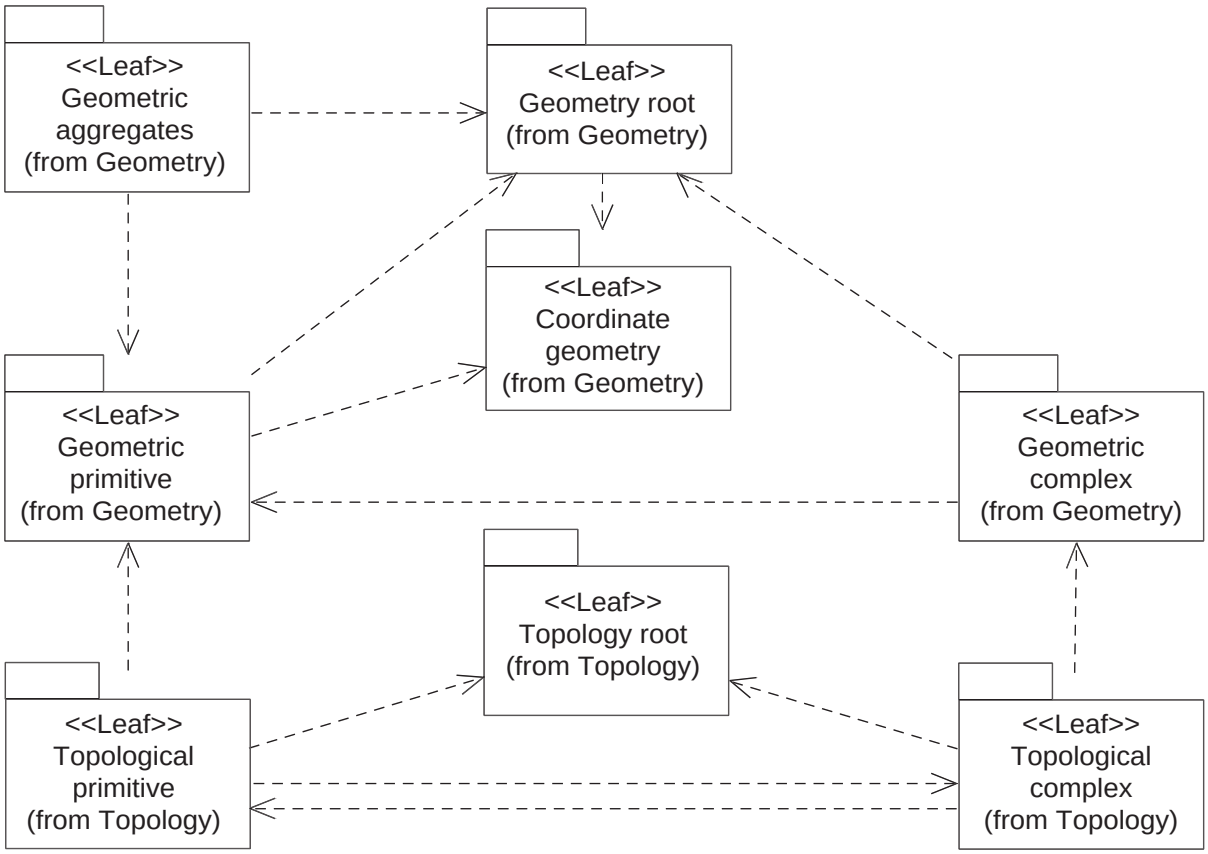


Figure 3 — Normative clause as UML package dependencies

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.

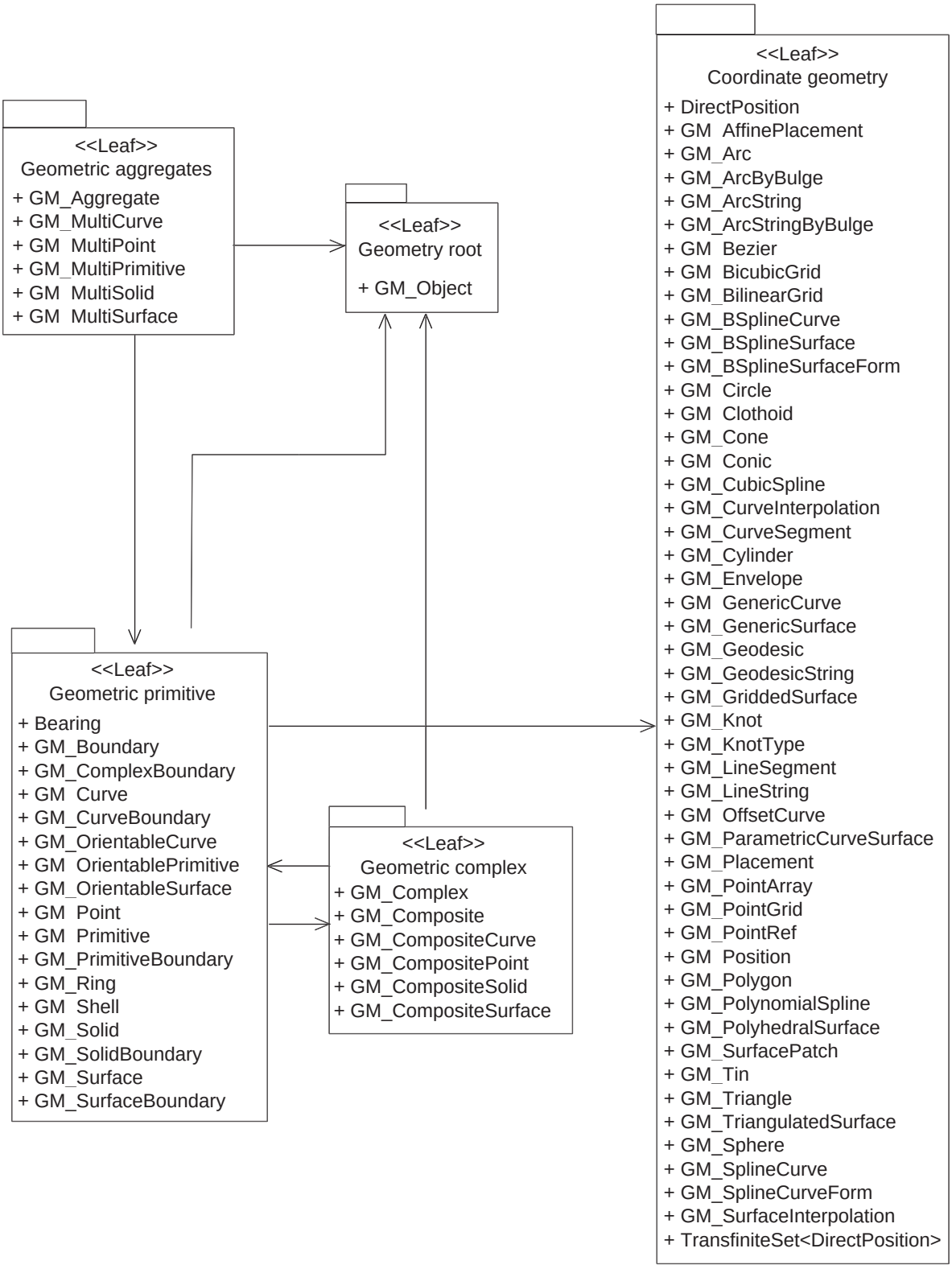


Figure 4 — Geometry package: Class content and internal dependencies

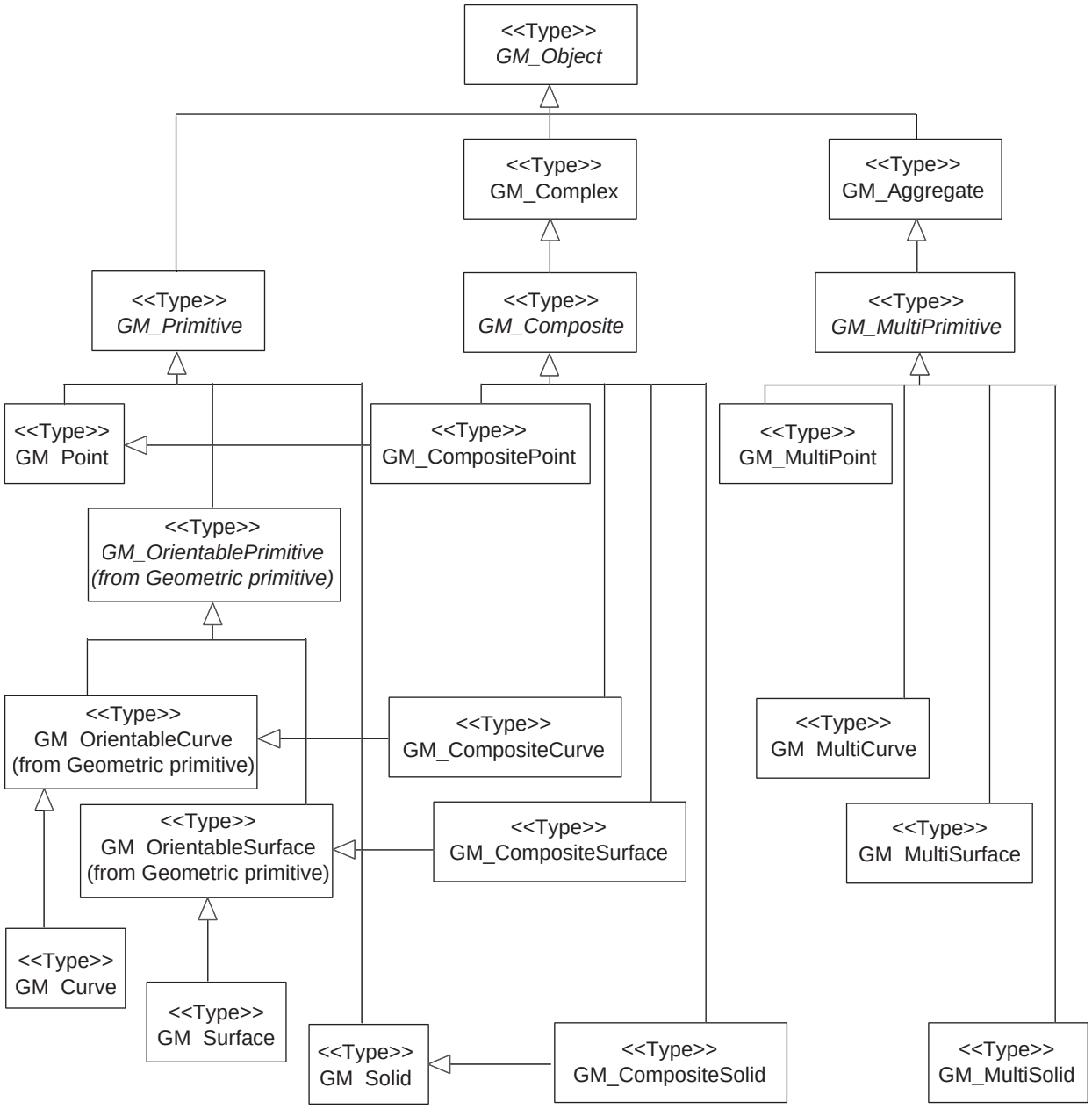


Figure 5 — Geometry basic classes with specialization relations

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 form 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.

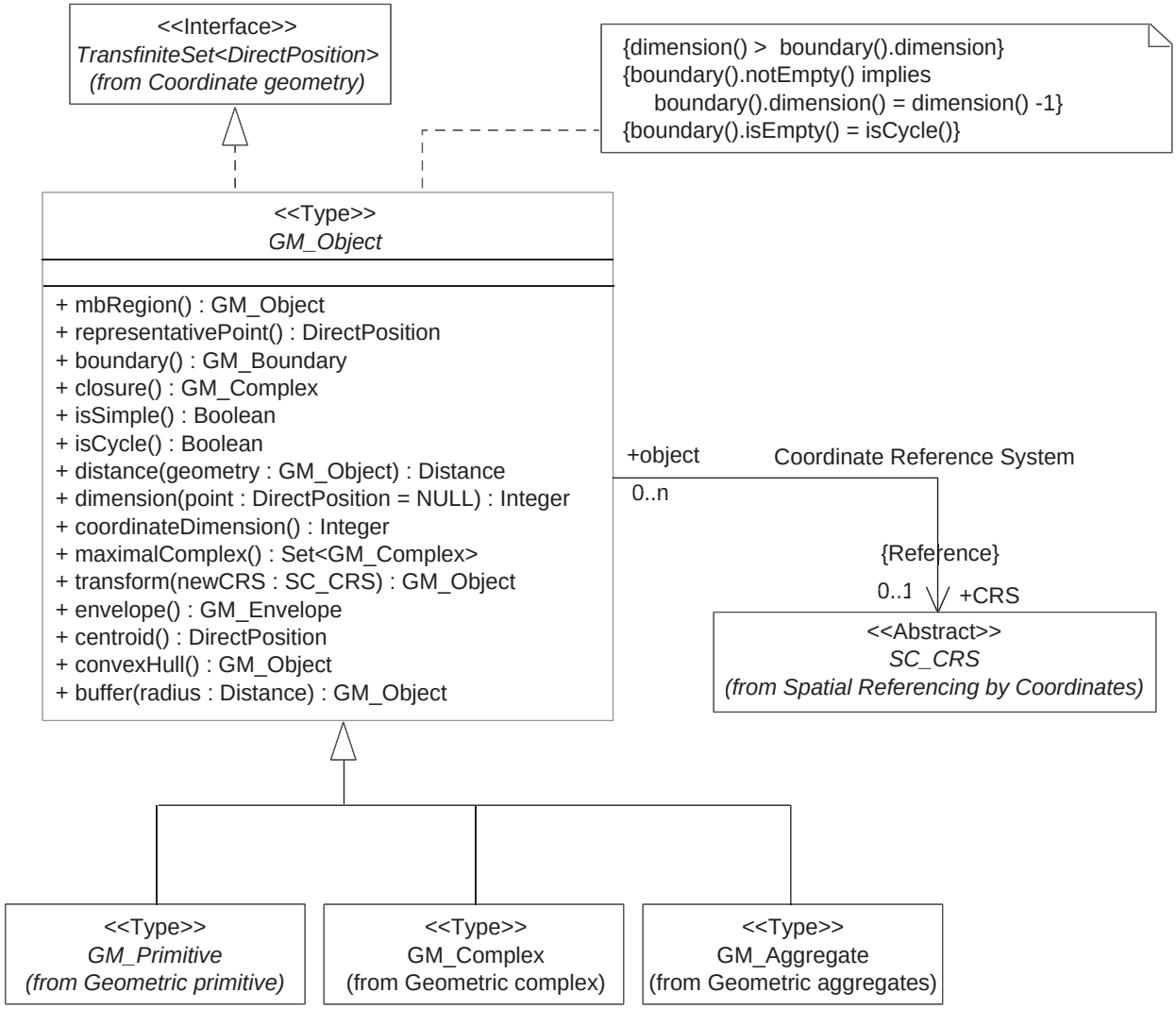


Figure 6 — GM_Object

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 overshot 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 overshot 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 emply. 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_Objects 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_Objects 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}
```

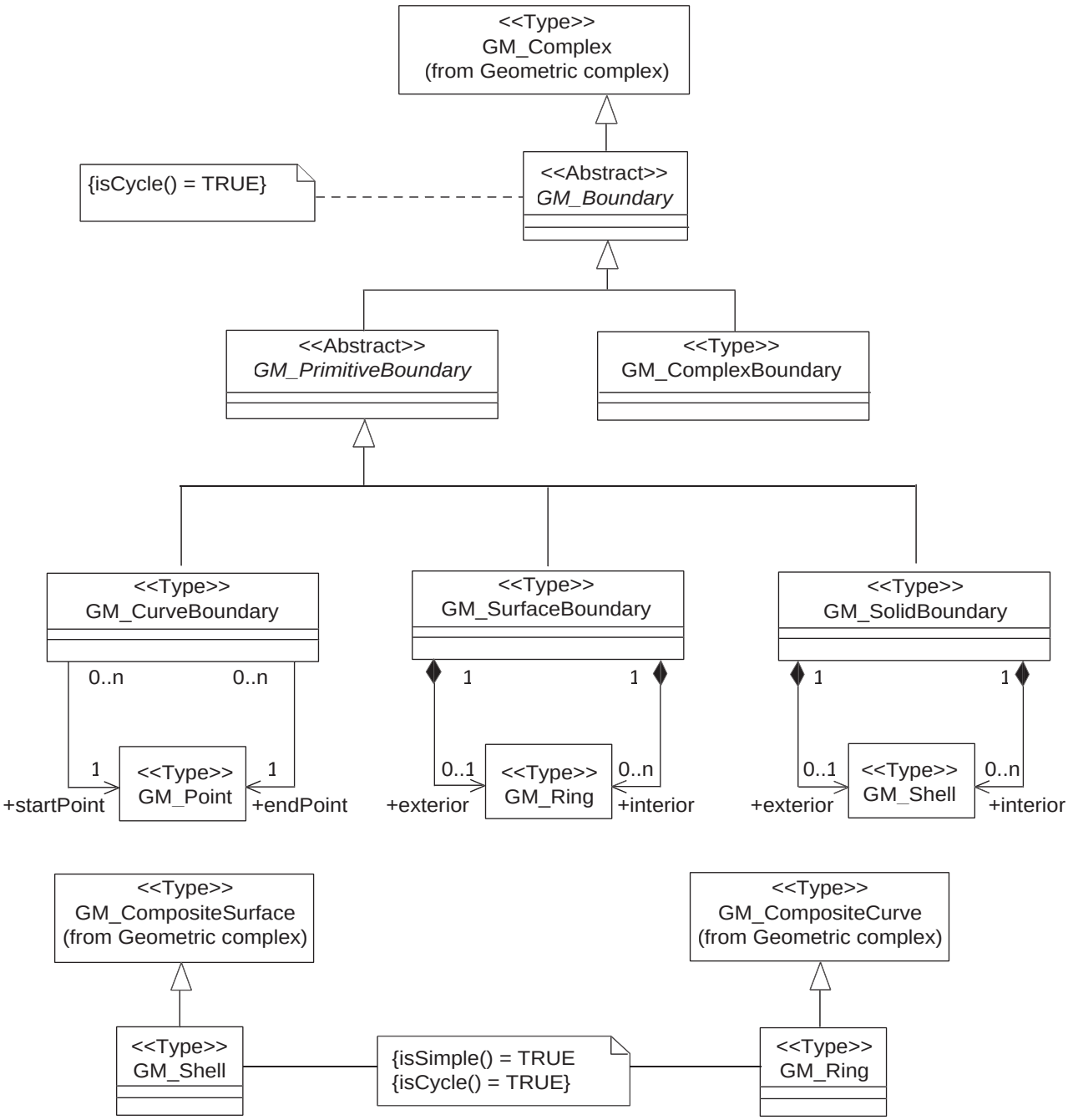


Figure 7 — GM_Boundary

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.

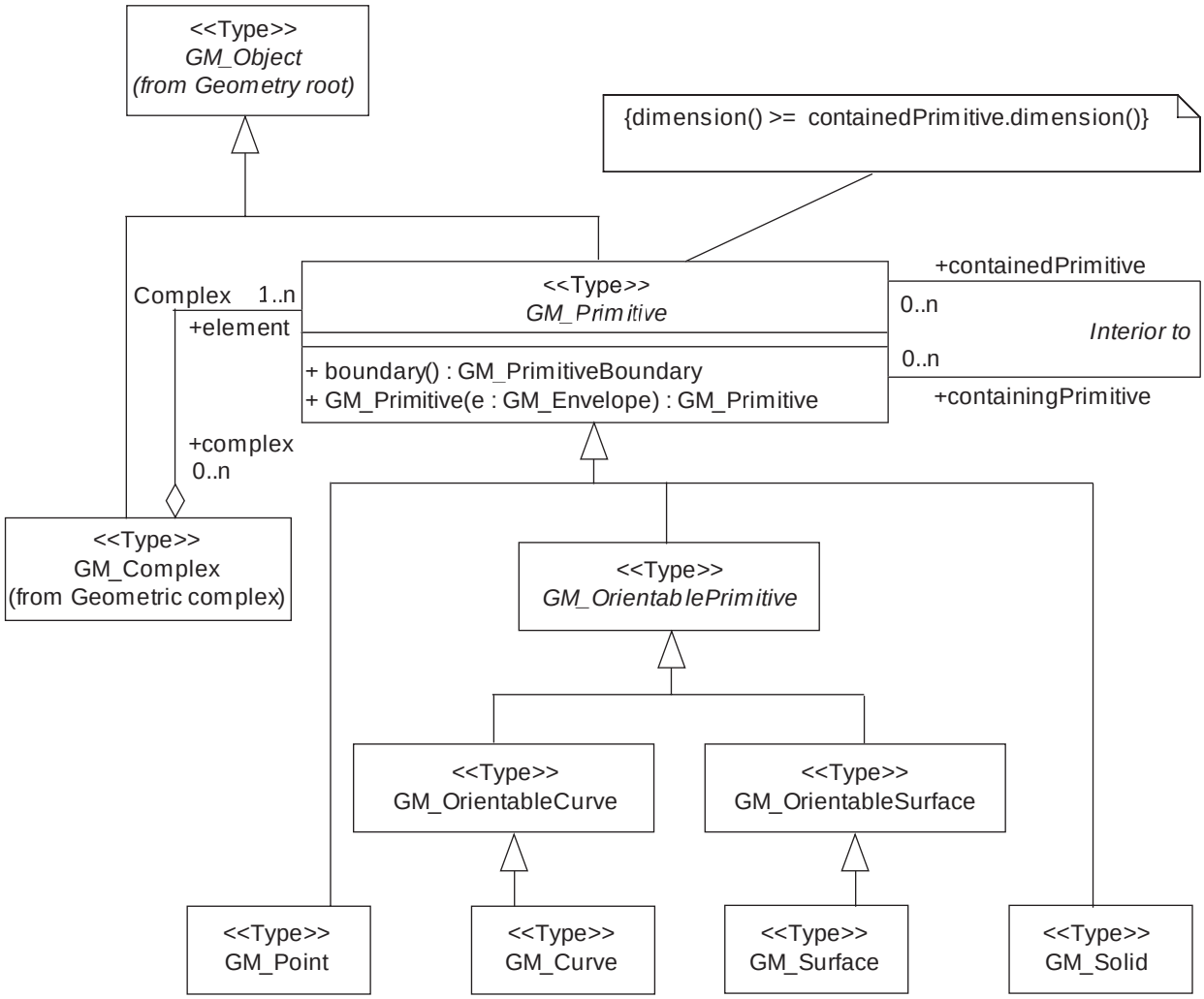


Figure 8 — GM_Primitive

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.

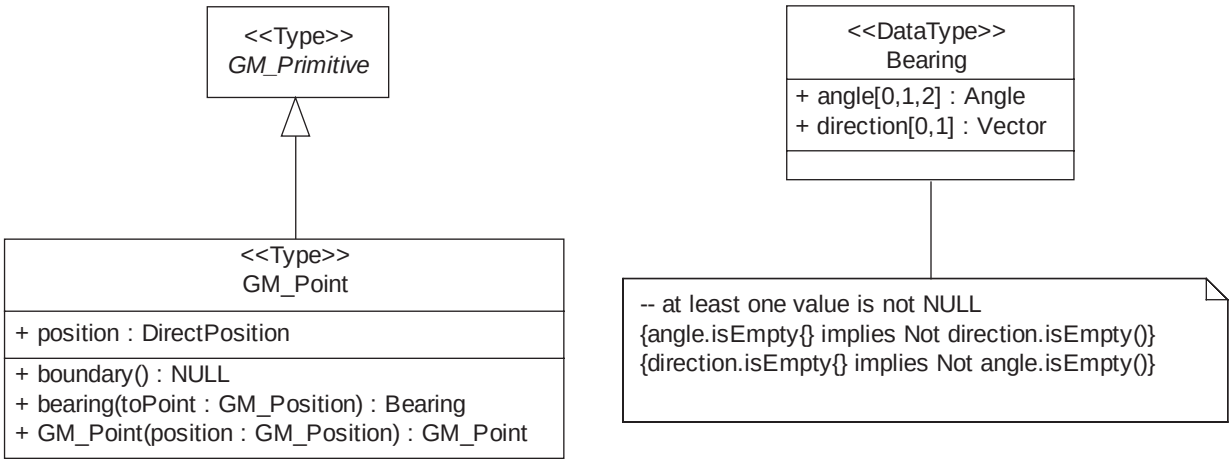


Figure 9 — GM_Point

6.3.11.2 position

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

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

GM_Point is the only subclass of GM_Primitive that cannot use GM_Positions to represent its defining geometry. A GM_Position is either a DirectPosition or a reference to a GM_Point (from which a DirectPosition may be obtained). By not allowing GM_Point to use this technique, infinitely recursive references are prevented. Applications may choose another mechanism to prevent this logical problem.

NOTE In most cases, the state of a GM_Point is fully determined by its position attribute. The only exception to this is if the GM_Point has been subclassed to provide additional non-geometric information such as symbology.

6.3.11.3 boundary

The operation "GM_Point::boundary" is a specialization of the boundary operation at GM_Object, and shall return an EMPTY value indication an empty set.

```
GM_Point::boundary() : EMPTY
```

6.3.11.4 bearing

The operation "bearing" shall return a Bearing of the tangent (at this GM_Point) to the curve between this GM_Point and a passed GM_Position.

```
GM_Point::bearing(toPoint : GM_Position) : Bearing
```

The choice of the curve type for defining the bearing is dependent on the SC_CRS in which this GM_Point is defined. For example, in the Mercator projection, the curve is the rhumb line. In 3D, geocentric coordinate system, the curve may be the geodesic joining the two points along the surface of the geoid or ellipsoid in use. Implementations that support this function shall specify the nature of the curve to be used.

NOTE The type "Vector" is a common data type defined in ISO/TS 19103.

6.3.11.5 GM_Point (constructor)

The constructor GM_Point creates a GM_Point at a given position.

```
GM_Point::GM_Point(position : GM_Position) : GM_Point
```

6.3.12 Bearing

6.3.12.1 Semantics

Bearing is a data type used to represent direction in the coordinate reference system. In a 2D coordinate reference system, this can be accomplished using a "angle measured from true north" or a 2D vector point in that direction. In a 3D coordinate reference system, two angles or any 3D vector is possible. If both a set of angles and a vector are given, then they shall be consistent with one another.

6.3.12.2 angle

In this variant of Bearing usually used for 2D coordinate systems, the first angle (azimuth) is measured from the first coordinate axis (usually north) in a counterclockwise fashion parallel to the reference surface tangent plane. If two angles are given, the second angle (altitude) usually represents the angle above (for positive angles) or below (for negative angles) a local plane parallel to the tangent plane of the reference surface.

```
Bearing::angle [0,1,2] : Angle
```

6.3.12.3 direction

In this variant of Bearing usually used for 3D coordinate systems, the direction is express as an arbitrary vector, in the coordinate system.

```
Bearing::direction [0,1] : Vector
```

6.3.13 GM_OrientablePrimitive

6.3.13.1 Semantics

Orientable primitives (Figure 10) are those that can be mirrored into new geometric objects in terms of their internal local coordinate systems (manifold charts). For curves, the orientation reflects the direction in which the curve is traversed, that is, the sense of its parameterization. When used as boundary curves, the surface being bounded is to the “left” of the oriented curve. For surfaces, the orientation reflects from which direction the local coordinate system can be viewed as right handed, the “top” or the surface being the direction of a completing z-axis that would form a right-handed system. When used as a boundary surface, the bounded solid is “below” the surface. The orientation of points and solids has no immediate geometric interpretation in 3-dimensional space.

GM_OrientablePrimitive objects are essentially references to geometric primitives that carry an “orientation” reversal flag (either “+” or “-”) that determines whether this primitive agrees or disagrees with the orientation of the referenced object.

NOTE There are several reasons for subclassing the “positive” primitives under the orientable primitives. First is a matter of the semantics of subclassing. Subclassing is assumed to be a “is type of” hierarchy. In the view used, the “positive” primitive is simply the orientable one with the positive orientation. If the opposite view were taken, and orientable primitives were subclassed under the “positive” primitive, then by subclassing logic, the “negative” primitive would have to hold the same sort of geometric description that the “positive” primitive does. The only viable solution would be to separate “negative” primitives under the geometric root as being some sort of reference to their opposite. This adds a great deal of complexity to the subclassing tree. To minimize the number of objects and to bypass this logical complexity, positively oriented primitives are self-referential (are instances of the corresponding primitive subtype) while negatively oriented primitives are not.

Orientable primitives are often denoted by a sign (for the orientation) and a base geometry (curve or surface). The sign datatype is defined in ISO TS 19103. If “c” is a curve, then “<+, c>” is its positive orientable curve and “<-, c>” is its negative orientable curve. In most cases, leaving out the syntax for record “< , >” does not lead to confusion, so “<+, c>” may be written as “+c” or simply “c”, and “<-, c>” as “-c”. Curve space arithmetic can be performed if the curves align properly, so that:

```
For c, d : GM_OrientableCurves such that c.endPoint = d.startPoint then
      ( c + d ) ==: GM_CompositeCurve = < c, d >
```

6.3.13.2 orientation

The “orientation” of an orientable primitive determines which of the two possible orientations this object represents.

```
GM_OrientablePrimitive::orientation : Sign
```

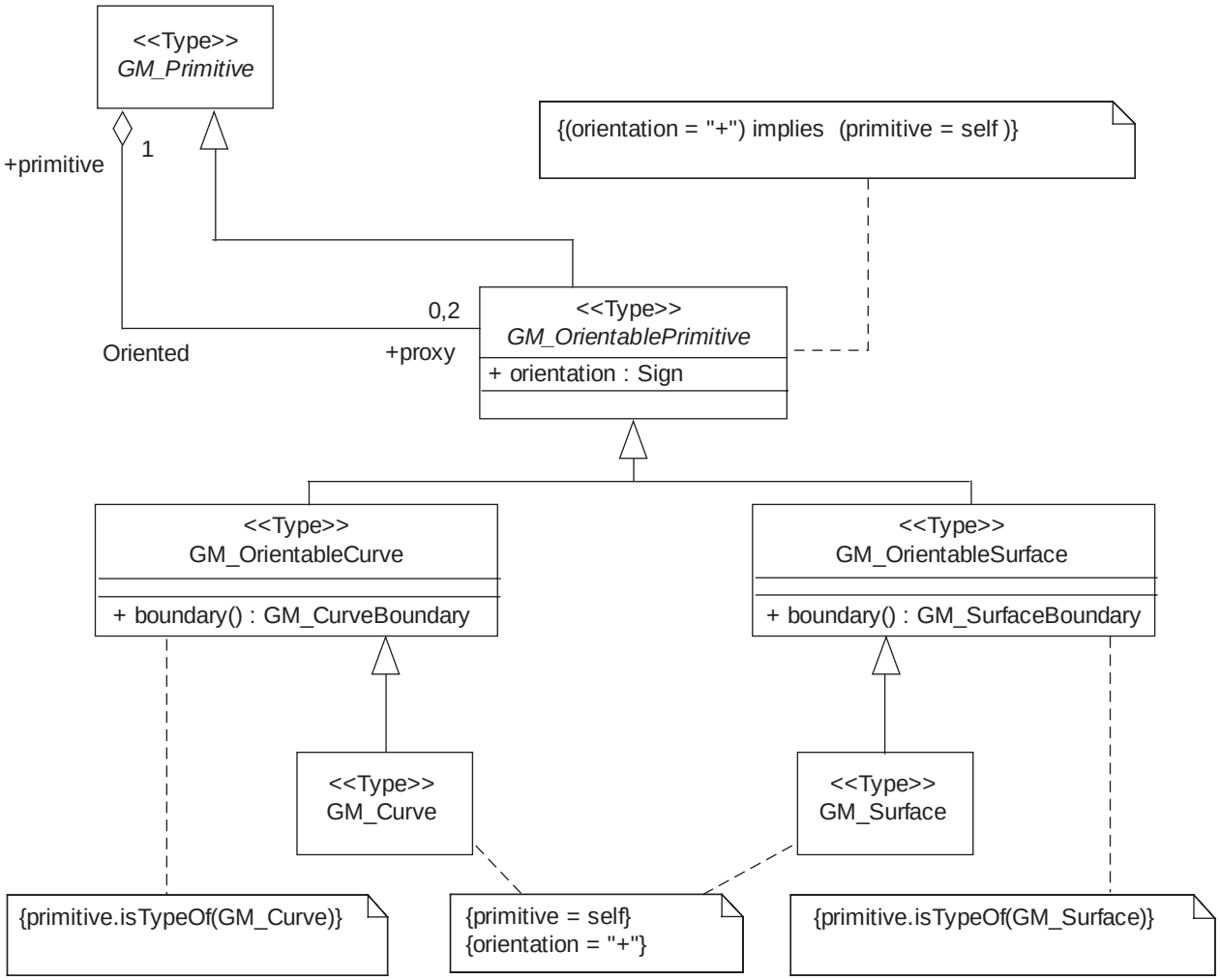


Figure 10 — GM_OrientablePrimitive

6.3.13.3 Oriented Association

Each GM_Primitive of dimension 1 or 2 is associated to two GM_OrientablePrimitives, one for each possible orientation.

```
GM_Primitive::proxy [0,2] : Reference<GM_OrientablePrimitive>;
GM_OrientablePrimitive::primitive [1] : Reference<GM_Primitive>;
```

For curves and surfaces, there are exactly two orientable primitives for each geometric object.

```
GM_Primitive:
      (proxy→notEmpty) = (dimension = 1 or dimension = 2);
GM_OrientablePrimitive:
      a, b :GM_OrientablePrimitive
      ((a.primitive=b.primitive)and(a.orientation=b.orientation)) implies a=b;
```

If the orientation is “+” (positive), then the GM_OrientablePrimitive shall be the corresponding GM_Curve or GM_Surface.

```
GM_OrientableCurve:
      orientation = "+" implies self.isTypeOf(GM_Curve);
```

```
GM_OrientableSurface:
      orientation = "+" implies self.isTypeOf(GM_Surface);
```

6.3.14 GM_OrientableCurve

6.3.14.1 Semantics

GM_OrientableCurve consists of a curve and an orientation inherited from GM_OrientablePrimitive. If the orientation is "+", then the GM_OrientableCurve is a GM_Curve. If the orientation is "-", then the GM_OrientableCurve is related to another GM_Curve with a parameterization that reverses the sense of the curve traversal.

```
GM_OrientableCurve:
{Orientation = "+" implies primitive = self};
{Orientation = "-" implies
      primitive.parameterization(length()-s) = parameterization(s)};
```

6.3.14.2 boundary

The operation "boundary" is a specialization of the boundary operation defined at GM_Object and at GM_Primitive. The boundary operation shall return an ordered pair of points, which are the start point and end point of the curve. If the curve is closed, then the boundary shall be empty. The data type GM_CurveBoundary is defined to simplify the structure of the boundary of the curve, see 6.3.5.

```
GM_OrientableCurve::boundary() : GM_CurveBoundary
```

6.3.15 GM_OrientableSurface

6.3.15.1 Semantics

GM_OrientableSurface consists of a surface and an orientation inherited from GM_OrientablePrimitive. If the orientation is "+", then the GM_OrientableSurface is a GM_Surface. If the orientation is "-", then the GM_OrientableSurface is a reference to a GM_Surface with an upNormal that reverses the direction for this GM_OrientableSurface, the sense of "the top of the surface" (see 6.4.33.2).

```
GM_OrientableSurface:
      {Orientation = "+" implies primitive = self};
      {(Orientation = "-" and TransfiniteSet::contains(p : DirectPosition))
            implies (primitive.upNormal(p) = - self.upNormal(p))};
```

6.3.15.2 boundary

The operation "boundary" specializes the boundary operation defined at GM_Object with the appropriate return type for GM_OrientableSurface. It shall return the set of circular sequences of GM_OrientableCurve that limit the extent of this GM_Surface. These curves shall be organized into one circular sequence of curves for each boundary component of the GM_Surface.

```
GM_OrientableSurface::boundary(): GM_SurfaceBoundary;
```

In cases where "exterior" boundary is not well defined, all the rings of the GM_SurfaceBoundary shall be listed as "interior".

NOTE The concept of exterior boundary for a surface is really only valid in a 2-dimensional plane. A bounded cylinder has two boundary components, neither of which can logically be classified as its exterior. Thus, in three dimensions, there is no valid definition of exterior that covers all cases.

6.3.16 GM_Curve

6.3.16.1 Semantics

GM_Curve (Figure 11) is a descendent subtype of GM_Primitive through GM_OrientablePrimitive. It is the basis for 1-dimensional geometry. A curve is a continuous image of an open interval and so could be written as a parameterized function such as $c(t):(a, b)\rightarrow E^n$ where "t" is a real parameter and E^n is Euclidean space of dimension n (usually two or three, as determined by the coordinate reference system). Any other parameterization that results in the same image curve, traced in the same direction, such as any linear shifts and positive scales such as $e(t) = c(a + t(b-a)):(0,1) \rightarrow E^n$, is an equivalent representation of the same curve. For the sake of simplicity, GM_Curves should be parameterized by arc length, so that the parameterization operation inherited from GM_GenericCurve (see 6.4.7) will be valid for parameters between 0 and the length of the curve.

Curves are continuous, connected, and have a measurable length in terms of the coordinate system. The orientation of the curve is determined by this parameterization, and is consistent with the tangent function, which approximates the derivative function of the parameterization and shall always point in the "forward" direction. The parameterization of the reversal of the curve defined by $c(t):(a, b)\rightarrow E^n$ would be defined by a function of the form $s(t) = c(a + b - t):(a, b)\rightarrow E^n$.

A curve is composed of one or more curve segments. Each curve segment within a curve may be defined using a different interpolation method. The curve segments are connected to one another, with the end point of each segment except the last being the start point of the next segment in the segment list.

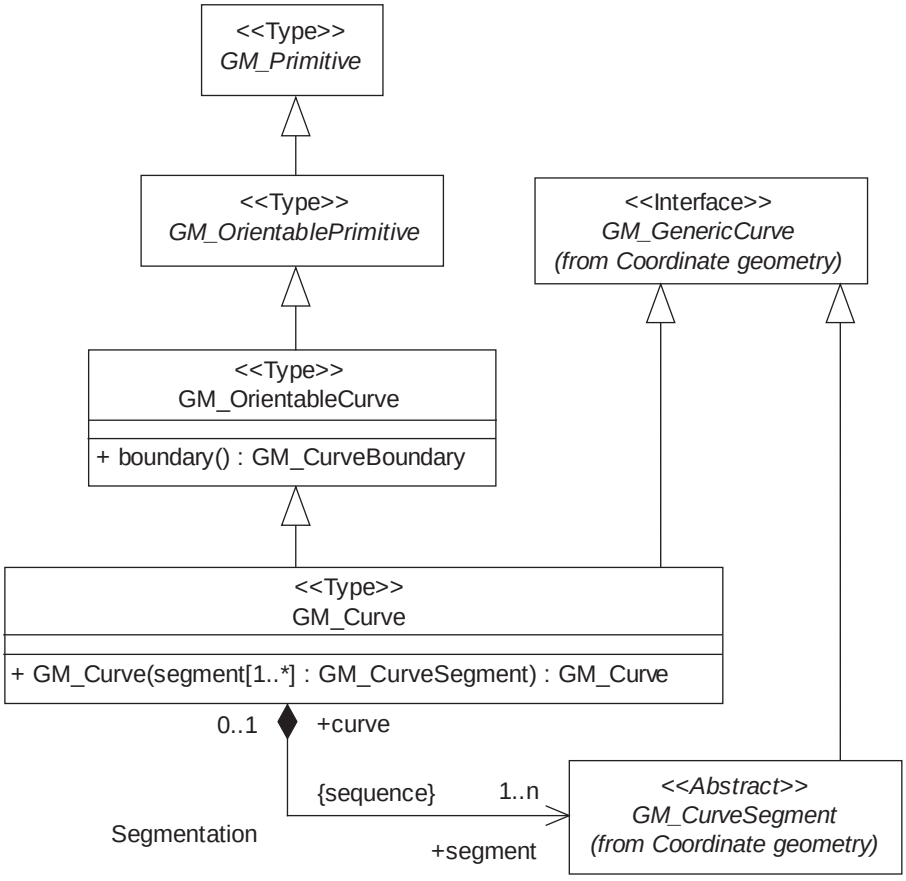


Figure 11 — GM_Curve

6.3.16.2 GM_Curve (constructor)

The constructor for GM_Curve takes a list of GM_CurveSegments with the appropriate end-to-start relationships and creates a GM_Curve.

```
GM_Curve::GM_Curve(segment[1..n] : GM_CurveSegment) : GM_Curve
```

6.3.16.3 Segmentation association

The association "segmentation" lists the components (GM_CurveSegments) of GM_Curve, each of which defines the direct position of points along a portion of the curve. The order of the GM_CurveSegments is the order in which they are used to trace the GM_Curve.

```
GM_Curve::segment [1..n] : Sequence<GM_CurveSegment>
      GM_CurveSegment::curve [0,1] : Reference<GM_Curve>
```

For a particular parameter interval, the GM_Curve and GM_CurveSegment agree.

```
GM_CurveSegment:
      {curve.startParam() <= self.startParam()};
      {curve.endParam() >= self.endParam()};
      {self.startParam() < self.endParam()};
      {s : Distance (startParam() <= s <= endParam())
         implies (curve.parameterization(s) = self.parameterization(s))};
```

NOTE In this International Standard, curve segments do not appear except in the context of a curve, and therefore the cardinality of the "curve" role in this association could be "1" which would preclude the use of curve segments except in this manner. While this would not affect this International Standard, leaving the cardinality as "0..1" allows other standards based on this one to use curve segments in a more open-ended manner.

6.3.17 GM_Surface

6.3.17.1 Semantics

GM_Surface (Figure 12) a subclass of GM_Primitive and is the basis for 2-dimensional geometry. Unorientable surfaces such as the Möbius band are not allowed. The orientation of a surface chooses an "up" direction through the choice of the upward normal, which, if the surface is not a cycle, is the side of the surface from which the exterior boundary appears counterclockwise. Reversal of the surface orientation reverses the curve orientation of each boundary component, and interchanges the conceptual "up" and "down" direction of the surface. If the surface is the boundary of a solid, the "up" direction is usually outward. For closed surfaces, which have no boundary, the up direction is that of the surface patches, which must be consistent with one another. Its included GM_SurfacePatches describe the interior structure of a GM_Surface.

NOTE Other than the restriction on orientability, no other "validity" condition is required for GM_Surface.

6.3.17.2 GM_Surface (constructor)

The first version of the constructor for GM_Surface takes a list of GM_SurfacePatches with the appropriate side-to-side relationships and creates a GM_Surface.

```
GM_Surface::GM_Surface(patch[1..n] : GM_SurfacePatch) : GM_Surface
```

The second version, which is guaranteed to work always in 2D coordinate spaces, constructs a GM_Surface by indicating its boundary as a collection of GM_Curves organized into a GM_SurfaceBoundary. In 3D

coordinate spaces, this second version of the constructor shall require all of the defining boundary GM_Curve instances to be coplanar (lie in a single plane) which will define the surface interior.

```
GM_Surface::GM_Surface(bdy : GM_SurfaceBoundary) : GM_Surface
```

6.3.17.3 Segmentation association

The “Segmentation” association relates this GM_Surface to a set of GM_SurfacePatches that shall be joined together to form this GM_Surface. Depending on the interpolation method, the set of patches may require significant additional structure. In general, the form of the patches shall be defined in the application schema.

```
GM_Surface::patch [1..n] : GM_SurfacePatch
GM_SurfacePatch::surface [0,1] : Reference<GM_Surface>
```

If the GM_Surface.coordinateDimension is 2, then the entire GM_Surface is one logical patch defined by linear interpolation from the boundary.

NOTE In this International Standard, surface patches do not appear except in the context of a surface, and therefore the cardinality of the “surface” role in this association could be “1” which would preclude the use of surface patches except in this manner. While this would not affect this International Standard, leaving the cardinality as “0..1” allows other standards based on this one to use surface patches in a more open-ended manner.

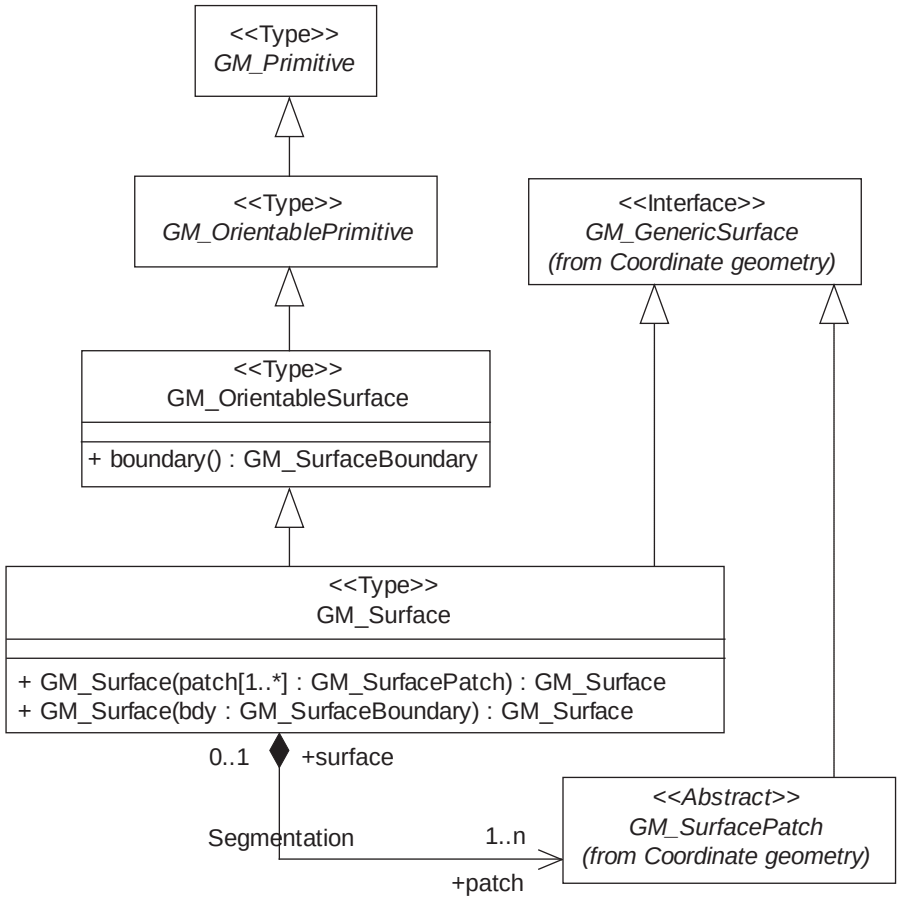


Figure 12 — GM_Surface

6.3.18 GM_Solid

6.3.18.1 Semantics

GM_Solid (Figure 13), a subclass of GM_Primitive, is the basis for 3-dimensional geometry. The extent of a solid is defined by the boundary surfaces.

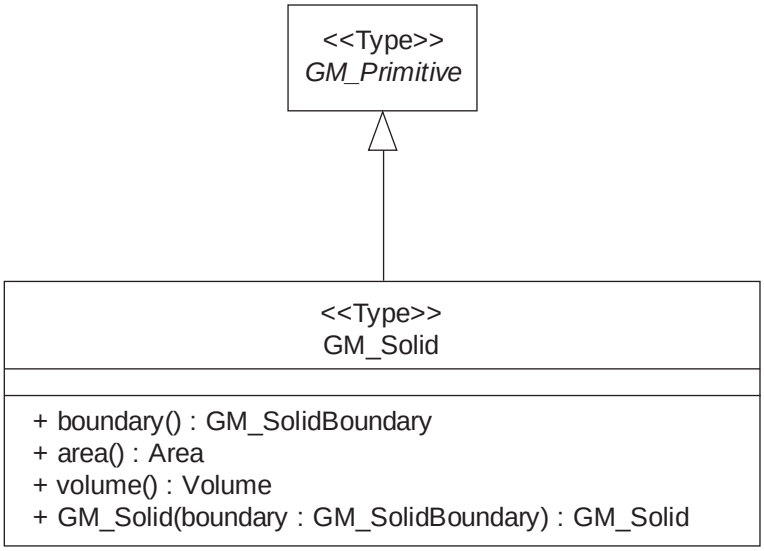


Figure 13 — GM_Solid

6.3.18.2 boundary

The operation “boundary” specializes the boundary operation defined at GM_Object and at GM_Primitive with the appropriate return type. It shall return a sequence of sets of GM_Surfaces that limit the extent of this GM_Solid. These surfaces shall be organized into one set of surfaces for each boundary component of the GM_Solid. Each of these shells shall be a cycle (closed composite surface without boundary).

```
GM_Solid::boundary() : GM_SolidBoundary
```

NOTE The exterior shell of a solid is defined only because the embedding coordinate space is always a 3D Euclidean one. In general, a solid in a bounded 3-dimensional manifold has no distinguished exterior boundary.

In cases where “exterior” boundary is not well defined, all the shells of the GM_SolidBoundary shall be listed as “interior”.

The GM_OrientableSurfaces that bound a solid shall be oriented outward – that is, the “top” of each GM_Surface as defined by its orientation shall face away from the interior of the solid.

Each GM_Shell, when viewed as a composite surface, shall be a cycle (see 6.2.2.6).

6.3.18.3 area

The operation “area” shall return the sum of the surface areas of all of the boundary components of a solid.

```
GM_Solid::area() : Area
```

The class Set<GM_Surface> has a “column operation” called “area” that accumulates the area of the components of the set. Using this, it can be said that for a GM_Solid:

```
GM_Solid:
      area() = boundary().area()
```

6.3.18.4 volume

The operation “volume” shall return the volume of this GM_Solid. This is the volume interior to the exterior boundary shell minus the sum of the volumes interior to any interior boundary shell.

```
GM_Solid::volume() : Volume
```

6.3.18.5 GM_Solid (constructor)

Since this International Standard is limited to 3-dimensional coordinate reference systems, any solid is definable by its boundary. The default constructor for a GM_Solid is from a properly structured set of GM_Shells organized as a GM_SolidBoundary.

```
GM_Solid::GM_Solid(boundary : GM_SolidBoundary) : GM_Solid
```

6.4 Coordinate geometry package

6.4.1 DirectPosition

6.4.1.1 Semantics

DirectPosition object data types (Figure 14) hold the coordinates for a position within some coordinate reference system. The coordinate reference system is described in ISO 19111. Since DirectPositions, as data types, will often be included in larger objects (such as GM_Objects) that have references to ISO19111::SC_CRS, the DirectPosition::cordinateReferenceSystem may be left NULL if this particular DirectPosition is included in a larger object with such a reference to a SC_CRS. In this case, the DirectPosition::cordinateReferenceSystem is implicitly assumed to take on the value of the containing object's SC_CRS.

6.4.1.2 coordinate

The attribute “coordinate” is a sequence of Numbers that hold the coordinate of this position in the specified reference system.

```
DirectPosition::coordinate : Sequence<Number>
```

6.4.1.3 dimension

The attribute “dimension” is the length of coordinate sequence (the number of entries). This is determined by the reference system.

```
/ DirectPosition::dimension : Integer = (coordinate.length)
```

6.4.1.4 coordinateReferenceSystem

The association role “coordinateReferenceSystem” is the coordinate system in which the coordinate is given. The type SC_CRS is described in ISO 19111.

```
DirectPosition::cordinateReferenceSystem [0,1] : ISO19111::SC_CRS
```

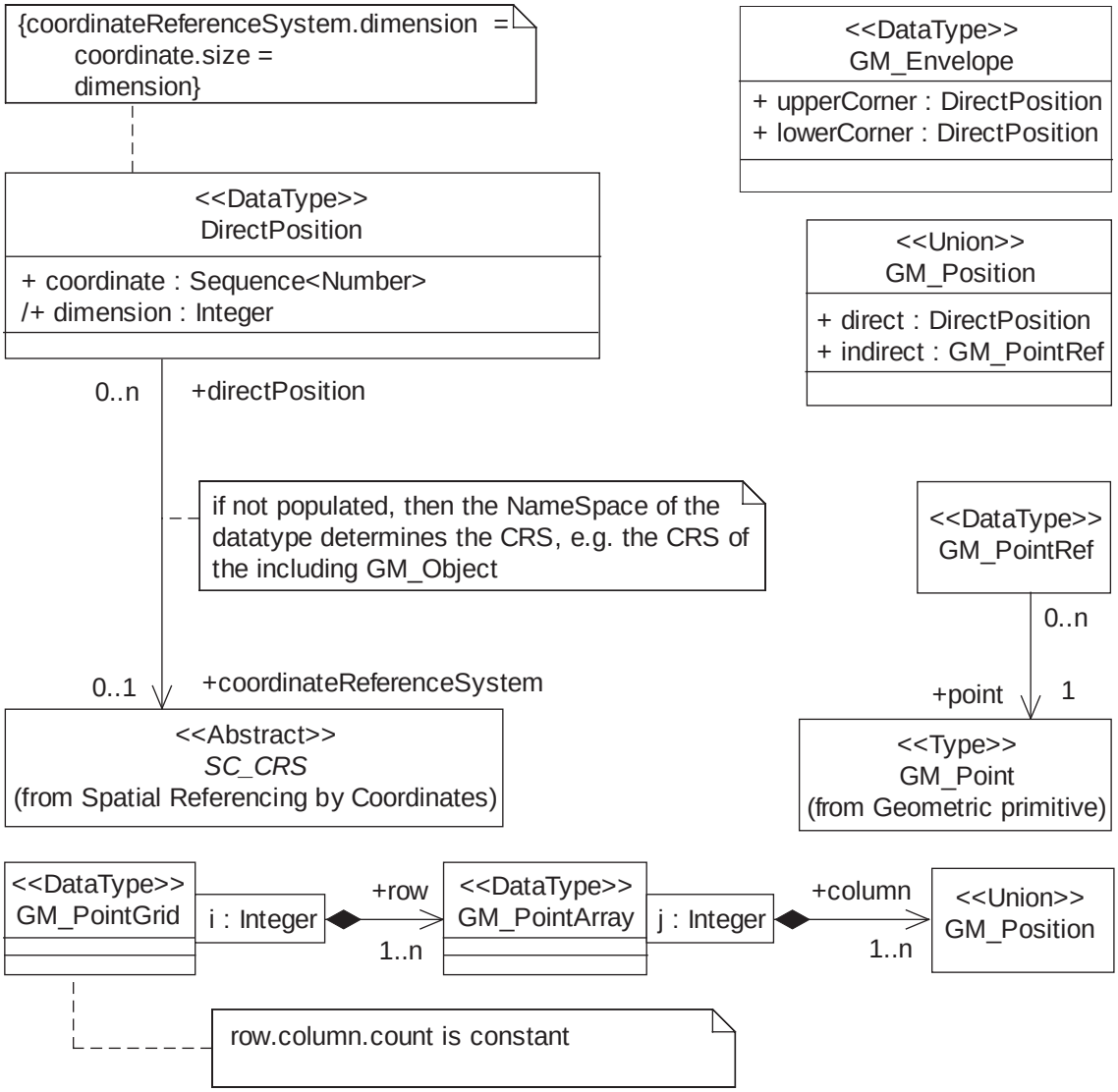


Figure 14 — DirectPosition

6.4.2 GM_PointRef

A GM_PointRef is used to reference an existing point. It is an instantiation of the template class Reference<GM_Point>.

```
GM_PointRef::point :: Reference<GM_Point>
```

6.4.3 GM_Envelope

6.4.3.1 Semantics

GM_Envelope is often referred to as a minimum bounding box or rectangle. Regardless of dimension, a GM_Envelope can be represented without ambiguity as two direct positions (coordinate points). To encode a GM_Envelope, it is sufficient to encode these two points. This is consistent with all of the data types in this International Standard, their state is represented by their publicly accessible attributes.

6.4.3.2 upperCorner

The “upperCorner” of a GM_Envelope is a coordinate position consisting of all the maximal coordinates for each dimension for all points within the GM_Envelope.

```
GM_Envelope::upperCorner : DirectPosition
```

6.4.3.3 lowerCorner

The “lowerCorner” of a GM_Envelope is a coordinate position consisting of all the minimal coordinates for each dimension for all points within the GM_Envelope.

```
GM_Envelope::lowerCorner : DirectPosition
```

6.4.4 TransfiniteSet<DirectPosition>

Much of the functionality of geometric objects is derived from viewing them as potentially infinite sets of DirectPositions (see Figure 6). The parameterized class TransfiniteSet<T> is defined in ISO TS 19103.

6.4.5 GM_Position

The data type GM_Position is a union type consisting of either a DirectPosition or of a reference to a GM_Point from which a DirectPosition shall be obtained. The use of this data type allows the identification of a position either directly as a coordinate (variant direct) or indirectly as a reference to a GM_Point (variant indirect).

```
GM_Position::direct [0,1] : DirectPosition
GM_Position::indirect [0,1] : GM_PointRef

GM_Position:
         {direct.isNull = indirect.isNotNull}
```

6.4.6 GM_PointArray, GMPointGrid

Many of the geometric constructs in this International Standard require the use of reference points which are organized into sequences or grids (sequences of equal length sequences).

```
GM_PointArray::column[1..n] : GM_Position
GM_PointGrid::row[1..n] : GM_PointArray
```

6.4.7 GM_GenericCurve

6.4.7.1 Semantics

GM_Curve and GM_CurveSegment both represent sections of curvilinear geometry, and therefore share a number of operation signatures. These are defined in the interface class GM_GenericCurve (Figure 15).

6.4.7.2 startPoint, endPoint

The operations “startPoint” and “endPoint” shall return the DirectPositions of the first point and last point, respectively on the GM_GenericCurve. This differs from the boundary operator in GM_Primitive, since it returns only the values of these two points, not representative objects.

```
GM_GenericCurve::startPoint() : DirectPosition
GM_GenericCurve::endPoint() : DirectPosition
```

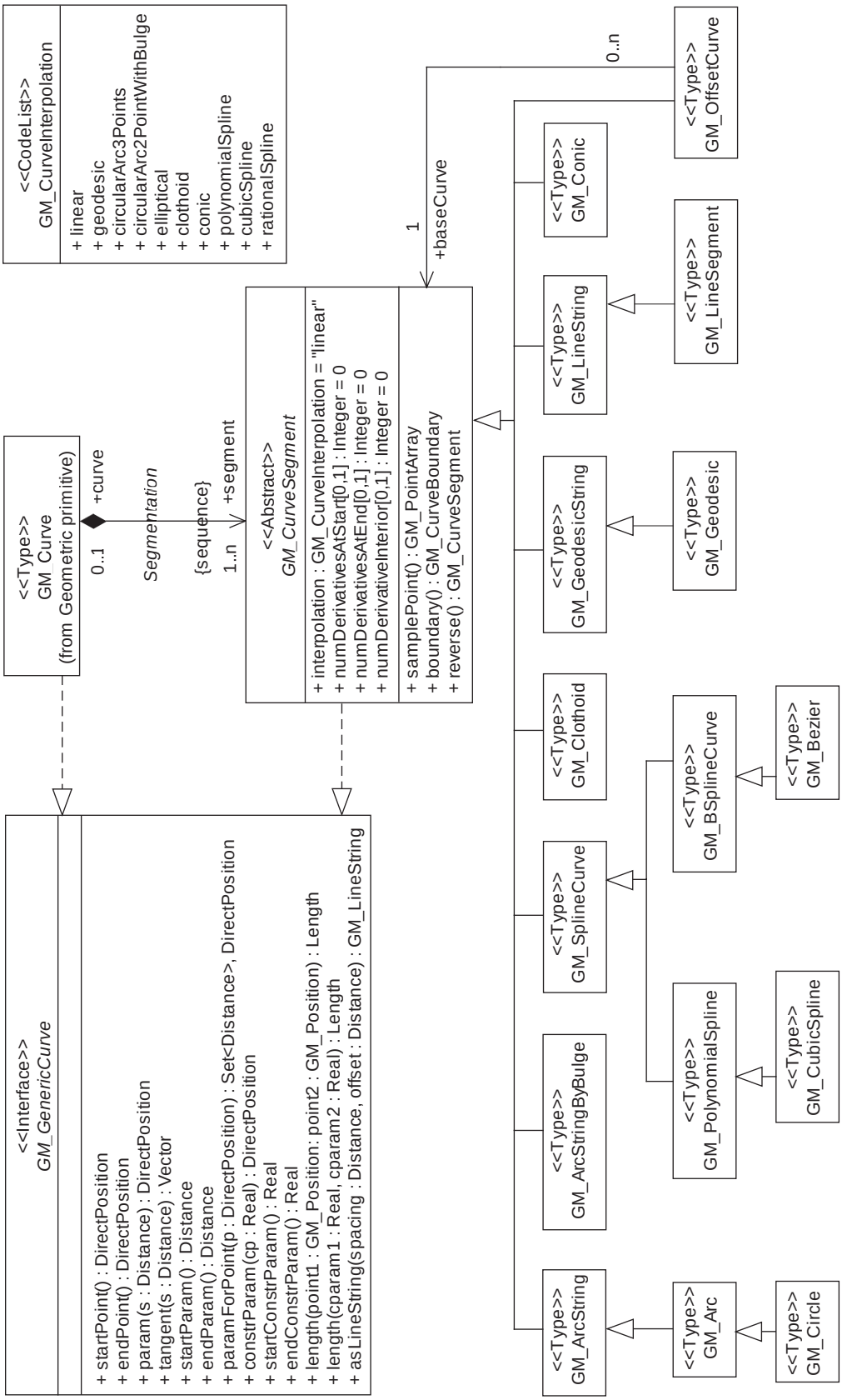


Figure 15 — Curve segment classes

6.4.7.3 tangent

The operation “tangent” shall return the tangent vector along this GM_GenericCurve at the passed parameter value. This vector approximates the derivative of the parameterization of the curve. The tangent shall be a unit vector (have length 1.0), which is consistent with the parameterization by arc length.

```
GM_GenericCurve::tangent(s : Distance) : Vector
```

6.4.7.4 startParam, endParam

The startParam and endParam indicate the parameters for the startPoint and endPoint respectively:

```
GM_GenericCurve::startParam() : Distance
GM_GenericCurve::endParam() : Distance
GM_GenericCurve:
      {parameterization(startParam()) = startPoint()};
      {parameterization(endParam()) = endPoint()};
      {length() = endParam() - startParam()}
```

The start and end parameter of a GM_Curve shall always be 0 and the arc length of the curve respectively. For GM_CurveSegments within a GM_Curve, the start and end parameters of the GM_CurveSegment shall be equal to those of the GM_Curve where this segment begins and ends respectively in the Segmentation association (see 6.3.16.3), so that the startParam of any segment (except the first) shall be equal to the endParam of the previous segment. If a GM_GenericCurve is used for other purposes, there shall be a restriction that the two parameters must differ by the arc length of the GM_GenericCurve.

6.4.7.5 paramForPoint

The operation “paramForPoint” shall return the parameter for this GM_GenericCurve at the passed DirectPosition. If the DirectPosition is not on the curve, the nearest point on the curve shall be used.

```
GM_GenericCurve::paramForPoint(p : DirectPosition) : Set<Distance>,
      DirectPosition
```

The DirectPosition closest is the actual value for the “p” used, that is, it shall be the point on the GM_GenericCurve closest to the coordinate passed in as “p”. The return set will contain only one distance, unless the curve is not simple. If there is more than one DirectPosition on the GM_GenericCurve at the same minimal distance from the passed “p”, the return value may be an arbitrary choice of one of the possible answers.

6.4.7.6 param

The operation “param” shall be the parameterized representation of the curve as the continuous image of a real number interval. The operation returns the DirectPosition on the GM_GenericCurve at the distance passed. The parameterization shall be by arc length, i.e. distance along the GM_GenericCurve measured from the start point and added to the start parameter.

```
GM_GenericCurve::param(s : Distance) : DirectPosition
```

6.4.7.7 startConstrParam, endConstrParam

The “startConstrParam” and “endConstrParam” indicate the parameters used in the constructive paramerization for the startPoint and endPoint respectively:

```
GM_GenericCurve::startConstrParam() : Real
GM_GenericCurve::endConstrParam() : Real
GM_GenericCurve:
      constrParam(startConstrParam()) = startPoint();
      constrParam(endConstrParam()) = endPoint();
```

There is no assumption that the startConstrParam is less than the endConstrParam, but the parameterization must be strictly monotonic (strictly increasing, or strictly decreasing).

NOTE Constructive parameters are often chosen for convenience of calculation, and seldom have any simple relation to arc distances, which are defined as the default parameterization. Normally, geometric constructions will use constructive parameters, as the programmer deems reasonable, and calculate arc length parameters when queried.

6.4.7.8 constrParam

The operation “constrParam” shall be an alternate representation of the curve as the continuous image of a real number interval without the restriction that the parameter represents the arc length of the curve, nor restrictions between a GM_Curve and its component GM_CurveSegments. The most common use of this operation is to expose the constructive equations of the underlying curve, especially useful when that curve is used to construct a parametric surface.

```
GM_GenericCurve::constrParam(cp : Real) : DirectPosition
```

6.4.7.9 length

The length of a piece of curvilinear geometry shall be a numeric measure of its length in a coordinate reference system. Since length is an accumulation of distance, its return value shall be in a unit of measure appropriate for measuring distances. The operation “length” shall return the distance between the two points along the curve. The default values of the two parameters shall be the start point and the end point, respectively. If either of the points is not on the curve, then it shall be projected to the nearest DirectPosition on the curve before the distance is calculated. If the curve is not simple and passes through either of the two points more than once, the distance shall be the minimal distance between the two points on this GM_Curve.

```
GM_GenericCurve::length(point1 : GM_Position = startPoint(),
                        point2 : GM_Position = endPoint()  ) : Length
```

The second form of the operation length shall work directly from the constructive parameters, allowing the direct conversion between the variables used in parameterization and constrParam.

```
GM_GenericCurve::length(cparam1 : Real = startConstrParam(),
                      cparam2 : Real = endConstrParam()) : Length
```

Distances between direct positions determined by the default parameterization are simply the difference of the parameter. The length function also allows for the conversion of the constructive parameter to the arc length parameter.

```
If p = length(startConstrParam, p2) + startParam
then parameterization(p) = constrParam(p2)
```

6.4.7.10 asLineString

The function “asLineString” constructs a line string (sequence of line segments) where the control points (ends of the segments) lie on this curve. If “maxSpacing” is given (not zero), then the distance between control points along the generated curve shall be not more than “maxSpacing”. If “maxOffset” is given (not zero), the

distance between generated curve at any point and the original curve shall not be more than the “maxOffset”. If both parameters are set, then both criteria shall be met. If the original control points of the curve lie on the curve, then they shall be included in the returned GM_LineString's controlPoints. If both parameters are set to zero, then the line string returned shall be constructed from the control points of the original curve.

```
GM_GenericCurve::asLineString(spacing : Distance = 0, offset : Distance = 0)
       : GM_LineString
```

NOTE This function is useful in creating linear approximations of the curve for simple actions such as display. It is often referred to as a “stroked curve”. For this purpose, the “maxOffset” version is useful in maintaining a minimal representation of the curve appropriate for the display device being targeted. This function is also useful in preparing to transform a curve from one coordinate reference system to another by transforming its control points. In this case, the “maxSpacing” version is more appropriate. Allowing both parameters to default to zero does not seem to have any useful geographic nor geometric interpretation unless further information is known about how the curves were constructed.

6.4.8 GM_CurveInterpolation

GM_CurveInterpolation is a list of codes that may be used to identify the interpolation mechanisms specified by an application schema. As a code list, there is no intention of limiting the potential values of GM_CurveInterpolation. Subtypes of GM_CurveSegment can be spawned directly through subclassing, or indirectly by specifying an interpolation method and an associated controlParameters record to support it. Valid meanings for “interpolation” include, but are not limited, to the following:

a) Linear (linear) – the interpolation mechanism shall return DirectPositions on a straight line between each consecutive pair of controlPoints.

b) Geodesic (geodesic) – the interpolation mechanism shall return DirectPositions on a geodesic curve between each consecutive pair of controlPoints. A geodesic curve is a curve of shortest length. The geodesic shall be determined in the coordinate reference system of the GM_Curve in which the GM_CurveSegment is used.

c) Circular arc by three points (circularArc3Points) – for each set of three consecutive controlPoints, the middle one being an even offset from the beginning of the sequence of control points, the interpolation mechanism shall return DirectPositions on a circular arc passing from the first point through the middle point to the third point. The sequence of control points shall have an odd number of elements. Note: if the three points are co-linear, the circular arc becomes a straight line.

d) Circular arc by two points and bulge factor (circularArc2PointWithBulge) – for each consecutive pair of controlPoints, the interpolation mechanism shall return DirectPositions on a circular arc passing from the first controlPoint to the second controlPoint, such that the associated control parameter determines the offset of the center of the arc from the center point of the chord, positive for leftward and negative for rightward. This form shall only be used in two dimensions because of the restricted nature of the definition technique.

e) Elliptical arc (elliptical) – for each set of four consecutive controlPoints, the interpolation mechanism shall return DirectPositions on an elliptical arc passing from the first controlPoint through the middle controlPoints in order to the fourth controlPoint. Note: if the four controlPoints are co-linear, the arc becomes a straight line. If the four controlPoints are on the same circle, the arc becomes a circular one.

f) Clothoid (clothoid) – uses a Cornu's spiral or clothoid interpolation.

g) Conic arc (conic) – same as elliptical arc but using five consecutive controlPoints to determine a conic section.

h) Polynomial Spline (polynomialSpline) – the controlPoints are ordered as in a line-string, but they are spanned by a polynomial spline function. Normally, the degree of continuity is determined by the degree of the polynomials chosen.

i) Cubic spline (cubicSpline) – the control points are interpolated using initial tangents and cubic polynomials, a form of degree 3 polynomial spline.

j) Rational Spline (rationalSpline) – the controlPoints are ordered as in a line string, but they are spanned by a rational (quotient of polynomials) spline function. Normally, the degree of continuity is determined by the degree of the polynomials chosen.

This list shall be implemented by a code list, and may vary in actual values from the above strings.

```
GM_CurveInterpolation::
      linear
      geodesic
      circularArc3Points
      circularArc2PointWithBulge
      elliptical
      clothoid
      conic
      polynomialSpline
      cubicSpline
      rationalSpline
```

6.4.9 GM_CurveSegment

6.4.9.1 Semantics

GM_CurveSegment defines a homogeneous segment of a GM_Curve. Each GM_CurveSegment shall be in, at most, one GM_Curve.

6.4.9.2 interpolation

The attribute “interpolation” specifies the curve interpolation mechanism used for this segment. This mechanism uses the control points and control parameters to determine the position of this GM_CurveSegment.

```
GM_CurveSegment::interpolation : GM_CurveInterpolation
```

6.4.9.3 numDerivatives

The attributes “numDerivativesAtStart” and “numDerivativesAtEnd” specify the type of continuity between this curve segment and its immediate neighbors, the first value for its predecessor, and the second for its successor. If this is the first or last curve segment in the curve, one of these values, as appropriate, is ignored. The attribute “numDerivativesInterior” specifies the type of continuity that is guaranteed interior to the curve. The default value of “0” means simple continuity, which is a mandatory minimum level of continuity. This level is referred to as “C^0” in mathematical texts. A value of 1 means that the function and its first derivative are continuous at the appropriate end point: “C^1” continuity. A value of “n” for any integer means the function and its first n derivatives are continuous: “C^n” continuity.

```
GM_CurveSegment::numDerivativesAtStart [0,1]: Integer = 0;
GM_CurveSegment::numDerivativesInterior [0,1]: Integer = 0;
GM_CurveSegment::numDerivativesAtEnd [0,1]: Integer = 0;
```

NOTE Use of these values is only appropriate when the basic curve definition is an underdetermined system. For example, line strings and segments cannot support continuity above C^0, since there is no spare control parameter to adjust the incoming angle at the end points of the segment. Spline functions on the other hand often have extra degrees of freedom on end segments that allow them to adjust the values of the derivatives to support C^1 or higher continuity.

6.4.9.4 samplePoint

The operation “samplePoint” returns an ordered array of point values (GM_PointArray) that lie on the GM_CurveSegment. In most cases, these will be related to control points used in the construction of the segment.

```
GM_CurveSegment::samplePoint() : GM_PointArray
```

NOTE The controlPoints of a curve segment are use to control its shape, and are not always on the curve segment itself. For example in a spline curve, the curve segment is given as a weighted vector sum of the controlPoints. Each weight function will have a maximum within the constructive parameter interval, which will roughly correspond to the point on the curve where it passes closest that the corresponding controlPoint. These points, the values of the curve at the maxima of the weight functions, will be the sample points for the curve segment.

6.4.9.5 boundary

The operation “boundary” on GM_CurveSegment operates with the same semantics as that on GM_Curve except that the end points of a GM_CurveSegment are not necessarily existing GM_Points and thus the boundary may contain transient GM_Points.

```
GM_CurveSegment::boundary() : GM_CurveBoundary
```

NOTE The above GM_CurveBoundary will almost always be two distinct positions, but, like GM_Curves, GM_CurveSegments can be cycles in themselves. The most likely scenario is that all of the points used will be transients (constructed to support the return value), except for the startPoint and endPoint of the aggregated GM_Curve. These two positions, in the case where the GM_Curve is involved in a GM_Complex, will be represented as GM_Points in the same GM_Complex.

6.4.9.6 reverse

The reverse of a GM_CurveSegment simply reverses the orientation of the parameterizations of the segment.

```
GM_CurveSegment::reverse() : GM_CurveSegment
```

6.4.10 GM_LineString

6.4.10.1 Semantics

A GM_LineString (Figure 16) consists of sequence of line segments, each having a parameterization like the one for GM_LineSegment (see 6.4.11). The class essentially combines a Sequence<GM_LineSegments> into a single object, with the obvious savings of storage space.

6.4.10.2 controlPoint

The controlPoints of a GM_LineString are a sequence of positions between which the curve is linearly interpolated. The first position in the sequence is the startPoint of the GM_LineString, and the last point in the sequence is the endPoint of the GM_LineString.

```
GM_LineString::controlPoint : GM_PointArray
```

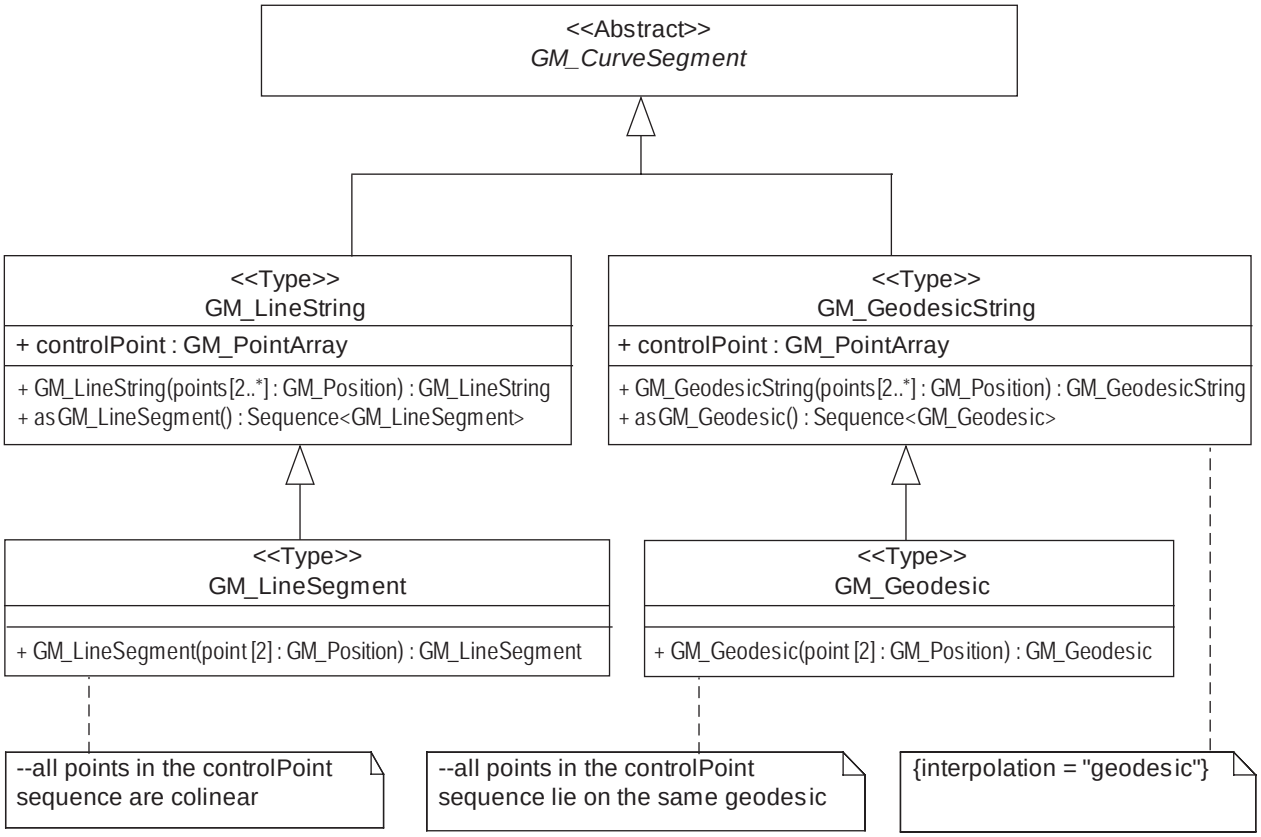


Figure 16 — Linear, arc and geodesic interpolation

6.4.10.3 GM_LineString (constructor)

The constructor for GM_LineString takes a sequence of points and constructs a GM_LineString with those points as controlPoints.

The constructor of a GM_LineString takes two or more positions and creates the appropriate line string joining them.

```
GM_LineString::GM_LineString(points[2..n]:GM_Position):GM_LineString
```

6.4.10.4 asGM_LineSegment

The operation asGM_LineSegment decomposes a line string into an equivalent sequence of line segments.

```
GM_LineString::asGM_LineSegment() : Sequence<GM_LineSegment>
```

6.4.11 GM_LineSegment

6.4.11.1 Semantics

A GM_LineSegment consists of two distinct DirectPositions (the startPoint and endPoint) joined by a straight line. Thus its interpolation attribute shall be “linear”. The default GM_GenericCurve::parameterization = c(s) is:

```
(L : Distance) = endParam – startParam
c(s) = ControlPoint[1]+((s-startParam)/L)*(ControlPoint[2]-ControlPoint[1])
```

Any other point in the controlPoint array must fall on this line. The control points of a GM_LineSegment shall all lie on the straight line between its start point and end point. Between these two points, other positions may be interpolated linearly.

NOTE The linear interpolation, given using a constructive parameter t, 0 ≤ t ≤ 1.0, where c(o) = c.startPoint() and c(1)=c.endPoint(), is:

$$c(t) = c(0)(1-t) + c(1)t$$

6.4.11.2 GM_LineSegment (constructor)

The constructor of a GM_LineSegment takes two positions and creates the appropriate line segment joining them. Constructors are class scoped.

```
GM_LineSegment::GM_LineSegment(point[2] : GM_Position) : GM_LineSegment
```

6.4.12 GM_GeodesicString

6.4.12.1 Semantics

A GM_GeodesicString consists of sequence of geodesic segments. The class essentially combines a Sequence<GM_Geodesic> into a single object, with the obvious savings of storage space.

6.4.12.2 controlPoint

The controlPoints of a GM_GeodesicString are a sequence of positions between which the GM_GeodesicString is interpolated using geodesics from the geoid or ellipsoid of the coordinate reference system being used. The organization of these points is identical to that in GM_LineString (6.4.10.2).

```
GM_GeodesicString::controlPoint : GM_PointArray
```

The interpolation for a GM_GeodesicString is “geodesic”.

```
GM_GeodesicString::interpolation : GM_CurveInterpolation = "geodesic"
```

6.4.12.3 GM_GeodesicString (constructor)

The constructor of a GM_GeodesicString takes two or more positions, interpolates using a geodesic defined from the geoid (or ellipsoid) of the coordinate reference system being used, and creates the appropriate geodesic string joining them.

```
GM_GeodesicString::GM_GeodesicString(points[2..n]:GM_Position):GeodesicString
```

6.4.12.4 asGM_Geodesic

The operation “asGM_Geodesic” decomposes a geodesic string into an equivalent sequence of geodesic segments.

```
GM_GeodesicString::asGM_Geodesic() : Sequence<GM_Geodesic>
```

6.4.13 GM_Geodesic

6.4.13.1 Semantics

A GM_Geodesic consists of two distinct positions joined by a geodesic curve. The control points of a GM_Geodesic shall all lie on the geodesic between its start point and end point. Between these two points, a geodesic curve defined from the ellipsoid or geoid model used by the coordinate reference system may be used to interpolate other positions. Any other point in the controlPoint array must fall on this geodesic.

6.4.13.2 interpolation

The interpolation for a GM_Geodesic is “geodesic”.

```
GM_Geodesic::interpolation : GM_CurveInterpolation = "geodesic"
```

6.4.13.3 GM_Geodesic (constructor)

The constructor of a GM_Geodesic takes two positions and creates the appropriate geodesic joining them. Constructors are class scoped.

```
GM_Geodesic:: GM_Geodesic(point[2] : GM_Position) : GM_Geodesic
```

6.4.14 GM_ArcString

6.4.14.1 Semantics

A GM_ArcString (Figure 17) is similar to a GM_LineString except that the interpolation is by circular arcs. Since it requires three points to determine a circular arc, the controlPoints are treated as a sequence of overlapping sets of three GM_Positions, the start of each arc, some point between the start and end, and the end of each arc. Since the end of each arc is the start of the next, this GM_Position is not repeated in the controlPoint sequence.

6.4.14.2 numArc

The attribute “numArc” shall be the number of circular arcs in the string. Since the interpolation method requires overlapping sets of three positions, the number of arcs determines the number of controlPoints.

```
GM_ArcString:numArc : Integer = ((controlPoint.length - 1)/2)
```

6.4.14.3 controlPoint

The attribute “controlPoint” is the sequence of points used to control the arcs in this string. The first three GM_Positions in the sequence determines the first arc. Any three consecutive GM_Positions beginning with an odd offset, determine another arc in the string.

```
GM_ArcString:controlPoint : GM_PointArray {size = 2*numArc +1}
```

6.4.14.4 interpolation

The interpolation for a GM_ArcString is “circularArc3Points”.

```
GM_ArcString::interpolation : GM_CurveInterpolation = "circularArc3Points"
```

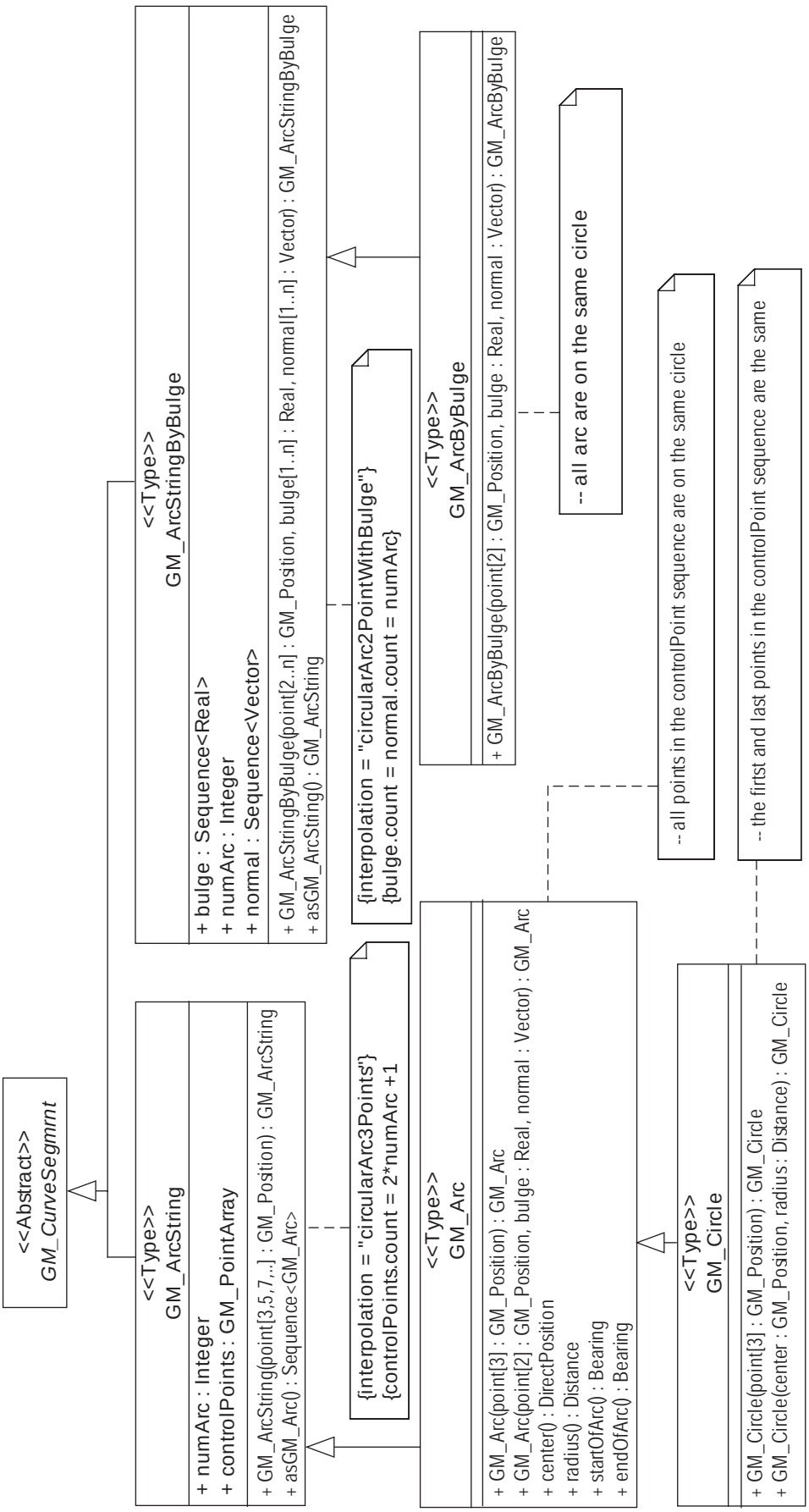


Figure 17 — Arcs

6.4.14.5 GM_ArcString (constructor)

The constructor GM_ArcString takes a sequence of points defined by GM_Positions and constructs a sequence of 3-point arcs jointing them. By the nature of an arc string, the sequence must have an odd number of positions.

```
GM_ArcString::GM_ArcSting( point[3, 5, 7...] : GM_Position): GM_ArcString
```

6.4.14.6 asGM_Arc

The operator asGM_Arc constructs a sequence of arcs that is the geometric equivalent of this arc string.

```
GM_ArcString::asGM_Arc() : Sequence<GM_Arc>
```

6.4.15 GM_Arc

6.4.15.1 Semantics

A GM_Arc is defined by three points, and consists of the arc of the circle determined by the three points, starting at the first, passing through the second and terminating at the third. If the three points are co-linear, then the arc shall be a 3-point line string, and will not be able to return values for center, radius, start angle and end angle.

NOTE In the model, a GM_Arc is a subclass of GM_ArcString, being a trivial arc string consisting of only one arc. This may be counter-intuitive in the sense that subclasses are often thought of as more complex than their superclass (with additional methods and attributes). A GM_Arc is simpler than a GM_ArcString in that it has less data, but it is more complex in that it can return geometric information such as “center”, “start angle”, and “end angle”. This additional computational complexity forces the subclassing to be the way it is. In addition the “is type of” semantics works this way and not the other.

In its simplest representation, the three points in the controlPoint sequence for an GM_Arc shall consist of, in order, the initial point on the arc, some point on the arc neither at the start or end, and the end point of the GM_Arc.

```
GM_Arc::controlPoint : GM_PointArray = <     startPoint : GM_Position,
                                             midPoint : GM_Position,
                                             endPoint : GM_Position>
```

If additional points are given, then all points must lie on the circle defined by any three non-colinear points in the control point array. All points shall lie on the same circle, and shall be given in the controlPoint array in the order in which they occur on the arc.

NOTE The use of the term “midPoint” for the center GM_Position of the controlPoint sequence is not meant to require that the GM_Position be the geometric midpoint of the arc. This is the best choice for this GM_Position from a computational stability perspective, but it is not absolutely necessary for the mathematics to work.

6.4.15.2 GM_Arc (constructor)

The constructor GM_Arc takes three positions and constructs the corresponding arc.

```
GM_Arc::GM_Arc(point[3] : GM_Position): GM_Arc
```

The second constructor GM_Arc takes two positions and the offset of the midpoint of the arc from the midpoint of the chord, given by a distance and direction, and constructs the corresponding arc.

```
GM_Arc::GM_Arc(point[2] : GM_Position, bulge : Real, normal : Vector) : GM_Arc
```

The midpoint of the resulting arc is given by:

```
midPoint = ((startPoint + endPoint)/2.0) + bulge*normal
```

In 2D coordinate reference systems, the bulge can be given a sign and the normal can be assumed to be the perpendicular to the line segment between the start and end point of the arc (the chord of the arc), pointing left.

EXAMPLE If the two points are $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$, and the bulge is b, then the vector in the direction of P_1 from P_0 is:

$$\vec{u} = (u_0, u_1) = (x_1 - x_0, y_1 - y_0) \Big/ \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

To complete a right-handed local coordinate system $\{\vec{u}, \vec{v}\}$, the two vectors must have a vector dot product of zero and a vector cross product of 1. By inspection, the leftward normal to complete the pair is:

$$\vec{v} = (v_0, v_1) = (-u_1, u_0)$$

The midpoint of the arc, which is the midpoint of the chord offset by the bulge, becomes:

$$m = \frac{P_0 + P_1}{2} + b\vec{v}$$

This is leftward if $b > 0$ and rightward if $b < 0$.

6.4.15.3 center

The operation center calculates the center of the circle of which this arc as a direct position. The coordinate reference system of the returned DirectPosition will be the same as that for the GM_Arc. In some extreme cases, the DirectPosition as calculated may lie outside the domain of validity of the coordinate reference system used by the GM_Arc (especially if the underlying arc has a very large radius). Application schemas may choose an appropriate course of action in such cases.

```
GM_Arc::center() : DirectPosition
```

6.4.15.4 radius

The operation radius calculates the radius of the circle of which this arc is a portion.

```
GM_Arc::radius() : Distance
```

6.4.15.5 startOfArc

The operation startOfArc calculates the bearing of the line from the center of the circle of which this arc is a portion to the start point of the arc. In the 2D case this will be a start angle. In the 3D case, the normal bearing angle implies that the arc is parallel to the reference circle. If this is not the case, then the bearing must include altitude information.

```
GM_Arc::startOfArc() : Bearing
```

6.4.15.6 endOfArc

The operation endOfArc calculates the bearing of the line from the center of the circle of which this arc is a portion to the end point of the arc. In the 2D case this will be a end angle. In the 3D case, the normal bearing angle implies that the arc is parallel to the reference circle. If this is not the case, then the bearing must include altitude information.

```
GM_Arc::endOfArc() : Bearing
```

6.4.16 GM_Circle

Same as GM_Arc, but closed to form a full circle. The “start” and “end” bearing are equal and shall be the bearing for the first controlPoint listed.

NOTE This still requires at least three distinct non-co-linear points to be unambiguously defined. The arc is simply extended until the first point is encountered.

6.4.17 GM_ArcStringByBulge

6.4.17.1 Semantics

This variant of the arc simply stores the parameters of the second constructor of the component GM_Arcs and recalculates the other attributes of the standard arc. The controlPoint sequence is similar to that in GM_ArcString, but the midPoint GM_Position is not needed since it is to be calculated. The control point sequence shall consist of the start and end points of each arc.

6.4.17.2 bulge

The bulge controls the offset of each arc's midpoint. The attribute “bulge” is the real number multiplier for the normal that determines the offset direction of the midpoint of each arc. The length of the bulge sequence is exactly one less than the length of the control point array, since a bulge is needed for each pair of adjacent points in the control point array.

```
GM_ArcByBulge::bulge : Sequence<Real>
```

The bulge is not given by a distance, since it is simply a multiplier for the normal, the unit of the offset distance is determined by the length function for vectors in the coordinate reference system. In the examples in this International Standard, the normal is often given as a Euclidean unit vector, which may or may not fix its length to one depending of the distance formulae used for the coordinate reference system.

The midpoint of the resulting arc is given by:

```
midPoint = ((startPoint + endPoint)/2.0) + bulge*normal
```

6.4.17.3 numArc

The attribute “numArc” shall be the number of circular arcs in the string. Since the interpolation method requires overlapping sets of two positions, the number of arcs determines the number of controlPoints.

```
GM_ArcStringByBulge:numArc : Integer = ((controlPoint.length – 1))
```

6.4.17.4 normal

The attribute “normal” is a vector normal (perpendicular) to the chord of the arc, the line joining the first and last point of the arc. In a 2D coordinate system, there are only two possible directions for the normal, and it is often given as a signed real, indicating its length, with a positive sign indicating a left turn angle from the chord line, and a negative sign indicating a right turn from the chord. In 3D, the normal determines the plane of the arc, along with the start and endPoint of the arc.

The normal is usually a unit vector, but this is not absolutely necessary. If the normal is a zero vector, the geometric object becomes equivalent to the straight line between the two end points. The length of the normal sequence is exactly the same as for the bulge sequence, one less than the control point sequence length.

```
GM_ArcByBulge::normal : Sequence<Vector>
```

NOTE A derived attribute “midPoint” may be defined as the midpoint of the arc as determined by the bulge and normal attributes.

```
/ GM_ArcByBulge::midPoint : Sequence<GM_Position>
midpoint(n) = (controlPoint(n) + contolPoint(n))/2.0) + bulge*normal
```

If each controlPoint pair were interspersed with its associated midpoint, then the result would be a valid set of control points for an GM_ArcString (which uses the 3-point interpolation method) that is geometrically equivalent to this GM_ArcStringByBulge.

6.4.17.5 interpolation

The interpolation attribute of a GM_ArcStringByBulge is always “circularArc2PointWithBulge”.

6.4.17.6 GM_ArcStringByBulge (constructor)

The constructor is equivalent to the second constructor of GM_Arc, except the bulge representation is maintained internal to the object.

```
GM_ArcByBulge::GM_ArcByBulge( point[2..n] : GM_Position,
      bulge[1..n] : Real, normal[1..n]  : Vector ) : GM_ArcStringByBulge
```

The midpoints of the resulting arc is given by:

```
midPoint(n) = ((point(n) + point(n+1))/2.0) + (bulge * normal)
```

6.4.17.7 asGM_ArcString

Each GM_ArcStringByBulge can be recast as a base GM_ArcString using the AsGM_ArcString operations.

```
GM_ArcStringByBulge::asGM_ArcString() : GM_ArcString;
```

6.4.18 GM_ArcByBulge

6.4.18.1 Semantics

GM_ArcByBulge is a restriction of GM_ArcStringByBulge (see 6.4.17). GM_ArcByBulge instances shall have all control points on the same circle.

6.4.18.2 GM_ArcByBulge (constructor)

The constructor is equivalent to the second constructor of GM_Arc, except the bulge representation is maintained.

```
GM_ArcByBulge::GM_ArcByBulge( point[2] : GM_Position,
      bulge : Real, normal : Vector ) : GM_ArcByBulge
```

The midpoint of the resulting arc is given by:

```
Midpoint = ((startPoint + endPoint)/2.0) + (bulge * normal)
```

6.4.19 GM_Conic

6.4.19.1 Semantics

The type GM_Conic (Figure 18) represents any general conic curve. Any of the conic section curves can be canonically represented in polar co-ordinates (ρ, φ) as:

$$\rho = \frac{P}{1 + e\cos(\varphi)} \quad \text{for} \quad \frac{-\pi}{2} \leqslant \varphi \leqslant \frac{\pi}{2}$$

where

P is semi-latus rectum;

e is the eccentricity.

This gives a conic with focus at the pole (origin), and the vertex on the conic nearest this focus in the direction of the polar axis, $\varphi = 0$ (at $(\rho,\varphi) = \left(\left(\frac{P}{1+e}\right),0\right)$ in polar coordinates). For $e = 0$, this is a circle. For $0 < e < 1$, this is an ellipse. For $e = 1$, this is a parabola. For $e > 1$, this is one branch of a hyperbola.

These generic conics can be viewed in a 2-dimensional Cartesian parameter space (u, v) given by the usual coordinate conversions $u = \rho\cos(\varphi)$ and $v = \rho\sin(\varphi)$. We can then convert this to a 3D coordinate reference system by using an affine transformation, $(u, v) \rightarrow (x, y, z)$ which is defined by:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u_x\, v_x \\ u_y\, v_y \\ u_z\, v_z \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}$$

This gives us φ as the constructive parameter. The DirectPosition given by (x_0, y_0, z_0) is the image of the origin in the local coordinate space (u, v).

Alternatively, the origin may be shifted to the vertex of the conic as

$$u' = \rho\cos(\varphi) - \left(P/_{1+e}\right) \quad \text{and} \quad v' = \rho\sin(\varphi)$$

and v can be used as the constructive parameter (see definition at GM_GenericCurve, 6.4.7.7).

In general, conics with small eccentricity and small P, use the first or “central” representation. Those with large eccentricity or large P tend to use the second or “linear” representation.

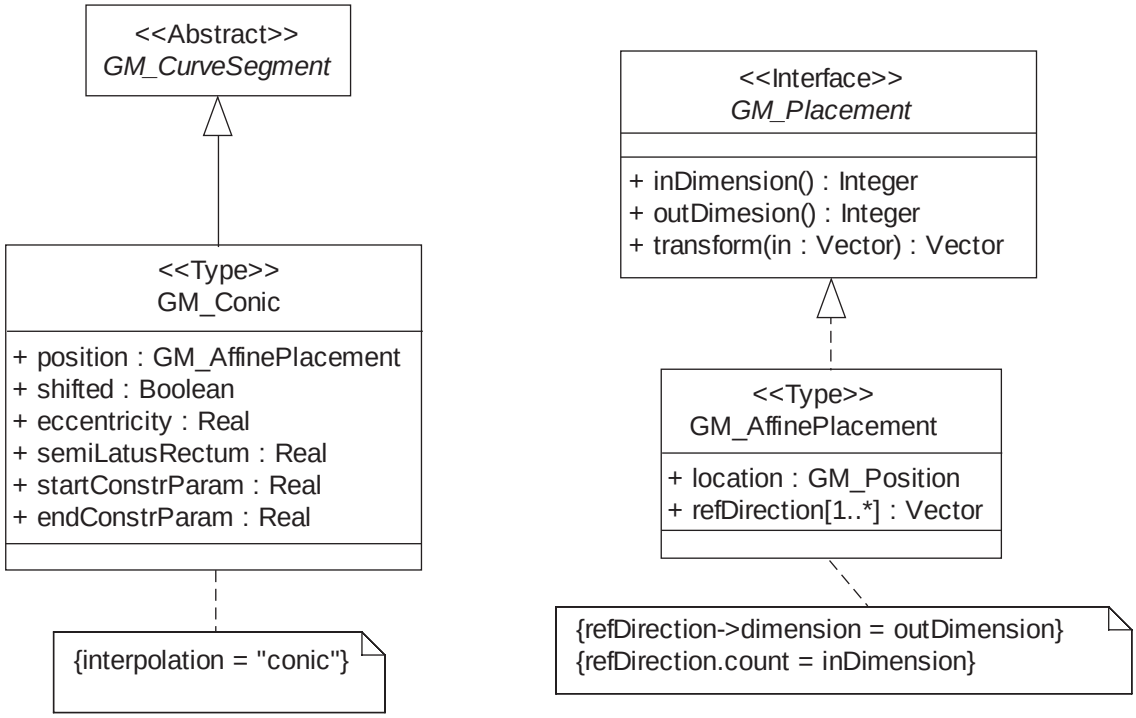


Figure 18 — Conics and placements

6.4.19.2 position

The attribute “position” will be an affine transformation object that maps the conic from parameter space into the coordinate space of the target coordinate reference system of the conic corresponding to the coordinate reference system of the GM_Object. This affine transformation is given by the formulae in the previous clause.

```
GM_Conic::position : GM_AffinePlacement
```

6.4.19.3 shifted

The attribute “shifted” is FALSE if the affine transformation is used on the unshifted (u, v) and TRUE if the affine transformation is applied to the shifted parameters (u', v'). This controls whether the focus or the vertex of the conic is at the origin in parameter space.

```
GM_Conic::shifted : Boolean
```

6.4.19.4 eccentricity

The attribute “eccentricity” is the value of the eccentricity parameter “e” used in the defining equation above. It controls the general shape of the curve, determining whether the curve is a circle, ellipse, parabola, or hyperbola.

```
GM_Conic::eccentricity : Real
```

6.4.19.5 semiLatusRectum

The attribute “semiLatusRectum” is the value of the parameter “P” used in the defining equation above. It controls how broad the conic is at each of its foci.

```
GM_Conic::semiLatusRectum : Real
```

6.4.19.6 startConstrParam, endConstrParam

The “startConstrParam” and “endConstrParam” indicate the parameters used in the constructive parameterization, given in 6.4.19.1, for the startPoint and endPoint respectively:

```
GM_Conic::startConstrParam : Real
GM_ Conic::endConstrParam : Real
GM_ Conic:
      constrParam(startConstrParam) = startPoint();
      constrParam(endConstrParam) = endPoint();
```

There is no assumption that the startConstrParam is less than the endConstrParam, but the parameterization must be strictly monotonic (strictly increasing, or strictly decreasing).

6.4.20 GM_Placement

6.4.20.1 Semantics

A placement takes a standard geometric construction and places it in geographic space. It defines a transformation from a constructive parameter space to the coordinate space of the coordinate reference system being used. Parameter spaces in formulae in this International Standard are given as (u, v) in 2D and (u, v, w) in 3D. Coordinate reference systems positions are given in formulae, in this International Standard, by either (x, y) in 2D, or (x, y, z) in 3D.

6.4.20.2 inDimension

The operation “inDimension()” shall return the dimension of the input parameter space.

```
GM_Placement::inDimension() : Integer
```

6.4.20.3 outDimension

The operation “outDimension()” shall return the dimension of the output coordinate reference system.

```
GM_Placement::outDimension() : Integer
```

NOTE Normally, outDimension (the dimension of the coordinate reference system) is larger than inDimension. If this is not the case, the transformation is probably singular, and may be replaceable by a simpler one from a smaller dimension parameter space.

6.4.20.4 transform

The operation “transform” maps the parameter coordinate points to the coordinate points in the output Cartesian space:

```
GM_Placement::transform(in :Vector {size = inDimension()}):
      Vector {size = outDimension()}
```

6.4.21 GM_AffinePlacement

6.4.21.1 Semantics

These placements are defined by linear transformation from the parameter space to the target coordinate space. 2-dimensional Cartesian parameter space, (u, v), transforms into a 3-dimensional coordinate reference system, (x, y, z), by using an affine transformation, $(u, v) \rightarrow (x, y, z)$, which is defined:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u_x & v_x \\ u_y & v_y \\ u_z & v_z \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}$$

Then, given this equation, the GM_AffinePlacement::location is the direct position (x_0, y_0, z_0), which is the target position of the origin in (u, v). The two reference directions (u_x, u_y, u_z) and (v_x, v_y, v_z) are the target directions of the unit basis vectors at the origin in (u, v).

6.4.21.2 location

The attribute “location” gives the target of the parameter space origin. This is the vector (x_0, y_0, z_0) in the formulae above.

```
GM_AffinePlacement::location : GM_Position
```

6.4.21.3 refDirection

The attribute “refDirection” gives the target directions for the coordinate basis vectors of the parameter space. These are the columns of the matrix in the formulae given above. The number of directions given shall be inDimension. The dimension of the directions shall be outDimension.

```
GM_AffinePlacement::refDirection [inDimension] : Vector {size = outDimension}
```

6.4.22 GM_Clothoid

6.4.22.1 Semantics

GM_Clothoid (Figure 19) implements the clothoid (or Cornu's spiral), which is a plane curve whose curvature is a fixed function of its length. In suitably chosen co-ordinates it is given by Fresnel's integrals:

$$x(t) = \int_0^t \cos(\frac{A\tau^2}{2})d\tau \quad \text{and} \quad y(t) = \int_0^t \sin(\frac{A\tau^2}{2})d\tau$$

See [16] in the bibliography for further properties of clothoid curves and piecewise clothoid curves.

This geometry is mainly used as a transition curve between curves of type straight line/circular arc or circular arc/circular arc. With this curve type it is possible to achieve a C2-continous transition between the above mentioned curve types. One formula for the clothoid is $A^2 = R^*t$ where A is a constant, R is the varying radius of curvature along the curve and t is the length along the curve and given in the Fresnel integrals.

6.4.22.2 refLocation

The attribute “refLocation” is an affine mapping that places the curve defined by the Fresnel Integrals into the coordinate reference system of this object.

```
GM_Clothoid::refLocation : GM_AffinePlacement
```

6.4.22.3 scaleFactor

The attribute “scaleFactor” gives the value for A in the equations above.

```
GM_Clothoid::scaleFactor: Number
```

6.4.22.4 startParameter

The attribute “startParameter” is the arc length distance from the inflection point that will be the start point for this curve segment. This shall be lower limit “t” used in the Fresnel integral and is the value of the constructive parameter of this curve segment at its start point. The “startParameter” can be either positive or negative. The parameter “t” acts as a constructive parameter, see 6.4.7.8.

```
GM_Clothoid::startParameter : Real
```

NOTE If 0.0 (zero), lies between the startConstrParam and endConstrParam of the clothoid, then the curve goes through the clothoid's inflection point, and the direction of its radius of curvature, given by the second derivative vector, changes sides with respect to the tangent vector. The term “length” for the parameter “t” is applicable only in the parameter space, and its relation to arc length after use of the placement, and with respect to the coordinate reference system of the curve is not deterministic.

6.4.22.5 endParameter

The attribute “ endParameter ” is the arc length distance from the inflection point that will be the end point for this curve segment. This shall be upper limit “t” used in the Fresnel integral and is the constructive parameter of this curve segment at its end point. The “endConstrParam” can be either positive or negative.

```
GM_Clothoid:: endParameter: Real
```

6.4.23 GM_OffsetCurve

6.4.23.1 Semantics

An offset curve is a curve at a constant distance from the basis curve. They can be useful as a cheap and simple alternative to constructing curves that are offsets by definition.

6.4.23.2 baseCurve

The attribute “baseCurve” is a reference to the curve from which this curve is defined as an offset.

```
GM_OffsetCurve::baseCurve : Reference<GM_Curve>
```

6.4.23.3 distance

The attribute “distance” is the distance at which the offset curve is generated from the basis curve. In a 2D system, positive distances are to be left of the basis curve, and negative distances are right of the basis curve.

```
GM_OffsetCurve::distance : Length
```

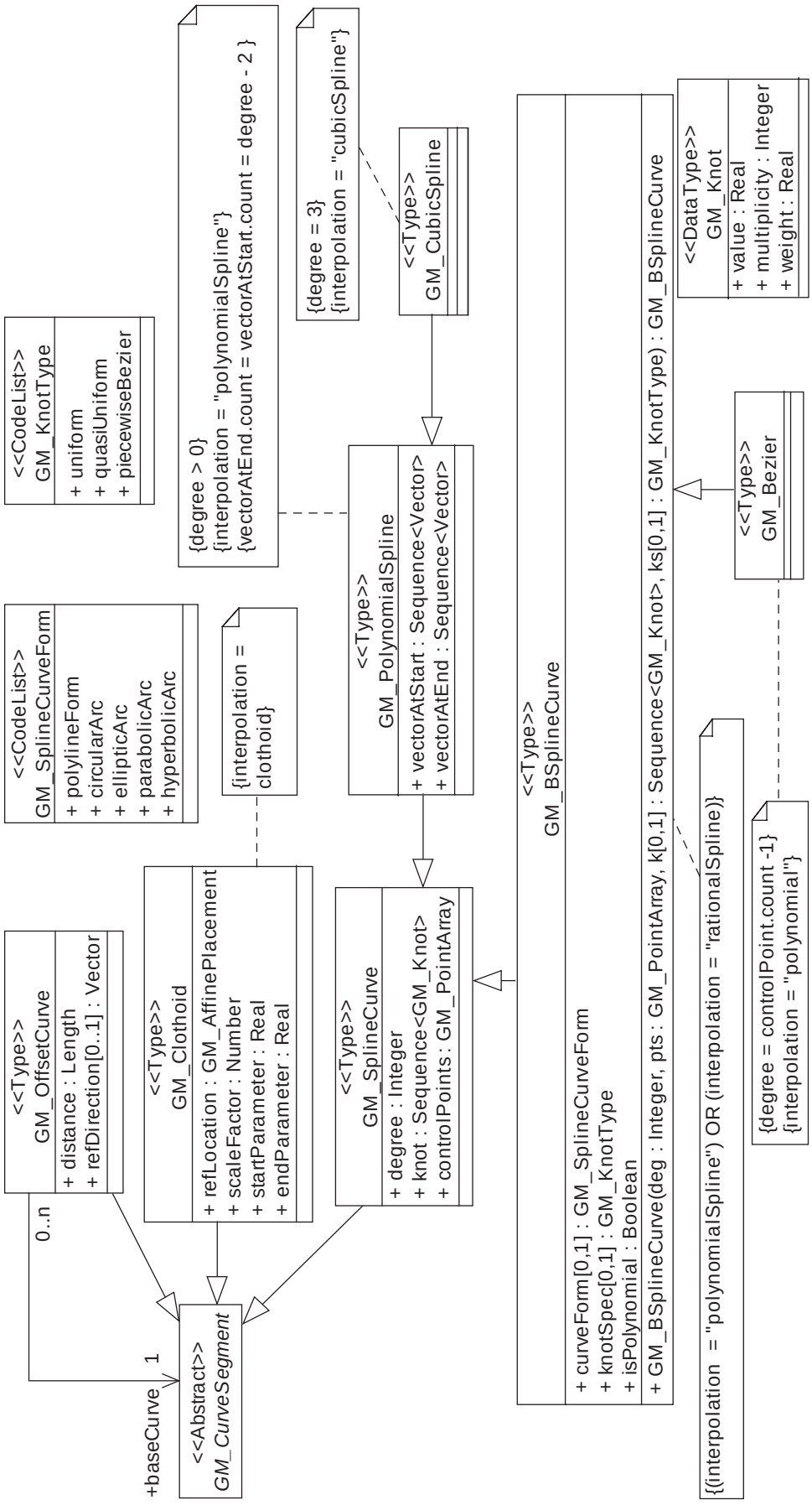


Figure 19 — Spline and specialty curves

6.4.23.4 refDirection

The attribute “refDirection” is used to define the vector direction of the offset curve from the basis curve. It can be omitted in the 2D case, where the distance can be positive or negative. In that case, distance defines left side (positive distance) or right side (negative distance) with respect to the tangent to the basis curve.

In 3D the basis curve shall have a well-defined tangent direction for every point. The offset curve at any point (parameter) on the basis curve “c” is in the direction

$\vec{s} = \vec{v} \times \vec{t}$ where $\vec{v} = c.refDirection()$ and $\vec{t} = c.tangent()$

For the offset direction to be well-defined, $\vec{v}$ shall not on any point of the curve be in the same, or opposite, direction as $\vec{t}$.

```
GM_OffsetCurve::refDirection : Vector
```

The default value of the refDirection shall be the local coordinate axis vector for elevation, which indicates up for the curve in a geographic sense.

NOTE If the refDirection is the positive tangent to the local elevation axis (“points upward”), then the offset vector points to the left of the curve when viewed from above.

6.4.24 GM_Knot

6.4.24.1 Semantics

GM_Knot is used to control the constructive parameter space for spline curves and surfaces. Each knot sequence is used for a dimension of the spline's parameter space. Thus, in a surface spline, there will be two knot sequences, one for each parameter (u, v). The i^{th}, j^{th} would be (u_i, v_j), where the original knot sequences were (u_i) and (v_j). Each knot of a spline curve or surface is described using a GM_Knot.

6.4.24.2 value

The attribute “value” is the value of the parameter at the knot of the spline. The sequence of knots shall be a non-decreasing sequence. That is, each knot's value in the sequence shall be equal to or greater than the previous knot's value. The use of equal consecutive knots is normally handled using the multiplicity.

```
GM_Knot::value : Real
```

6.4.24.3 multiplicity

The attribute “multiplicity” is the multiplicity of this knot used in the definition of the spline (with the same weight).

```
GM_Knot::multiplicity : Integer
```

6.4.24.4 weight

The attribute “weight” is the value of the averaging weight used for this knot of the spline.

```
GM_Knot::weight : Real
```

6.4.25 GM_KnotType

A B-spline is uniform if and only if all knots are of multiplicity one and they differ by a positive constant from the preceding knot. A B-spline is quasi-uniform if and only if the knots are of multiplicity (degree+1) at the ends, of multiplicity one elsewhere, and they differ by a positive constant from the preceding knot. This code list is used to describe the distribution of knots in the parameter space of various splines. The possible values are:

— uniform: the form of knots is appropriate for a uniform B-spline.

— quasiUniform: the form of knots is appropriate for a quasi-uniform B-spline.

— piecewiseBezier : the form of knots is appropriate for a piecewise Bezier curve.

```
GM_KnotType::
      uniform
      quasiUniform
      piecewiseBezier
```

6.4.26 GM_SplineCurve

6.4.26.1 Semantics

GM_SplineCurve (Figure 19) acts as a root for subtypes of GM_CurveSegment using some version of spline, either polynomial or rational functions.

6.4.26.2 knot

The attribute “knot” shall be the sequence of distinct knots used to define the spline basis functions. Recall that the knot data type holds information on knot multiplicity.

```
GM_SplineCurve::knot : Sequence<GM_Knot>
```

6.4.26.3 degree

The attribute “degree” shall be the degree of the polynomial used for interpolation in this GM_PolynomialSpline.

```
GM_SplineCurve::degree : Integer
```

6.4.26.4 controlPoints

The attribute “controlPoints” shall be an array of points that are used in the interpolation in this GM_SplineCurve.

```
GM_ SplineCurve::controlPoints : GM_PointArray
```

6.4.27 GM_PolynomialSpline

6.4.27.1 Semantics

An “n^{th} degree” polynomial spline shall be defined piecewise as an n-degree polynomial, with up to C^{n-1} continuity at the control points where the defining polynomial changes. This level of continuity is controlled by the attribute numDerivativesInterior. Parameters shall include directions for as many as degree – 2 derivatives

of the polynomial at the start and end point of the segment. GM_Linestring is equivalent to a 1st degree polynomial spline. It has simple continuity at the controlPoints (C^0), but does not require derivative information (degree – 2 = –1).

NOTE The major difference between the polynomial splines and the b-splines (basis splines) is that polynomial splines pass through their control points, making the control point and sample point array identical.

6.4.27.2 Interpolation

The interpolation mechanism for a GM_PolynomialSpline is “polynomialSpline”.

```
GM_PolynomialSpline::
      interpolation : GM_InterpolationMethod = "polynomialSpline"
```

6.4.27.3 vectorAtStart

The attribute “vectorAtStart” shall be the values used for the initial derivative (up to degree – 2) used for interpolation in this GM_PolynomialSpline at the start point of the spline.

```
GM_PolynomialSpline::vectorAtStart : Sequence<Vector> {size = degree - 2}
```

6.4.27.4 vectorAtEnd

The attribute “vectorAtEnd” shall be the values used for the final derivative (up to degree – 2) used for interpolation in this GM_PolynomialSpline at the start point of the spline.

```
GM_PolynomialSpline::vectorAtEnd :Sequence<Vector> {size = degree - 2}
```

6.4.28 GM_CubicSpline

Cubic splines are similar to line strings in that they are a sequence of segments each with its own defining function. A cubic spline uses the control points and a set of derivative parameters to define a piecewise third degree polynomial interpolation. Unlike line-strings, the parameterization by arc length is not necessarily still a polynomial. Splines have two parameterizations that are used in this International Standard, the defining one (constructive parameter) and the one that has been reparameterized by arc length to satisfy the requirements in GM_GenericCurve.

The function describing the curve must be C^2, that is, have a continuous first and second derivative at all points, and pass through the controlPoints in the order given. Between the control points, the curve segment is defined by a cubic polynomial. At each control point, the polynomial changes in such a manner that the first and second derivative vectors are the same from either side. The control parameters record must contain vectorAtStart, and vectorAtEnd which are the unit tangent vectors at controlPoint[1] and controlPoint[n] where n = controlPoint.count.

The restriction on “vectorAtStart” and “vectorAtEnd” reduce these sequences to a single tangent vector each.

```
GM_CubicSpline::vectorAtStart : Vector \\ "degree – 2" is 1
GM_CubicSpline::vectorAtEnd : Vector \\ "degree – 2" is 1
```

NOTE The actual implementation of the cubic polynomials varies, but the curve so generated is guaranteed to be unique. See [2], [10], [12], [18], and [19] in the bibliography for examples of implementations.

The interpolation mechanism for a GM_CubicSpline is “cubicSpline”.

```
GM_CubicSpline::interpolation : GM_InterpolationMethod = "cubicSpline"
```

The degree for a GM_CubicSpline is “3”.

```
GM_CubicSpline::degree : Integer = "3"
```

6.4.29 GM_SplineCurveForm

This code list is used to indicate which sort of curve may be approximated by a particular B-spline. The potential values are:

— polyine form: a connected sequence of line segments represented by a one degree B-spline (a line string).

— circular Arc: an arc of a circle or a complete circle.

— elliptic Arc: an arc of an ellipse or a complete ellipse.

— parabolic Arc: an arc of a finite length of a parabola.

— hyperbolic Arc: an arc of a finite length of one branch of a hyperbola.

```
GM_SplineCurveForm::
      polylineForm
      circularArc
      ellipticalArc
      parabolicArc
      hyperbolicArc
```

6.4.30 GM_BSplineCurve

6.4.30.1 Semantics

A B-spline (Figure 19) is a piecewise parametric polynomial or rational curve described in terms of control points and basis functions. If the weights in the knots are equal then it is a polynomial spline. If not, then it is a rational function spline. If the Boolean “isPolynomial” is set to TRUE then the weights shall all be set to 1. A B-spline curve is a piecewise Bézier curve if it is quasi-uniform except that the interior knots have multiplicity “degree” rather than having multiplicity one. In this subtype the knot spacing shall be 1.0, starting at 0.0. A piecewise Bézier curve that has only two knots, 0.0, and 1.0, each of multiplicity (degree+1), is equivalent to a simple Bézier curve.

6.4.30.2 degree

The attribute “degree” shall be the algebraic degree of the basis functions.

```
GM_BSplineCurve::degree : Integer
```

6.4.30.3 curveForm

The attribute “curveForm” is used to identify particular types of curve which this spline is being used to approximate. It is for information only, used to capture the original intention. If no such approximation is intended, then the value of this attribute is NULL.

```
GM_BSplineCurve::curveForm : GM_SplineCurveForm
```

6.4.30.4 knotSpec

The attribute “knotSpec” gives the type of knot distribution used in defining this spline. This is for information only and is set according to the different construction-functions.

```
GM_BSplineCurve::knotSpec[0,1] : GM_KnotType
```

6.4.30.5 isPolynomial

The attribute “isPolynomial” is set to “True” if this is a polynomial spline.

```
GM_BSplineCurve::isPolynomial : Boolean
```

6.4.30.6 GM_BSplineCurve (constructor)

The class constructor “GM_BSplineCurve” takes the pertinent information described in the attributes above and constructs a B-spline curve. If the knotSpec is not present, then the knotType is uniform and the knots are evenly spaced, and except for the first and last have multiplicity = 1. At the ends the knots are of multiplicity = degree+1. If the knotType is uniform they need not be specified.

```
GM_BSplineCurve::GM_BSplineCurve(deg : Integer, pts : GM_PointArray,
      k [0,1] : Sequence<GM_Knot>, ks [0,1] : GM_KnotType) : GM_BSplineCurve
```

NOTE If the B-spline curve is uniform and degree = 1, the B-spline is equivalent to a polyline (GM_LineString). If the knotType is “piecewiseBezier”, then the knots are defaulted so that they are evenly spaced, and except for the first and last have multiplicity equal to degree. At the ends the knots are of multiplicity = degree+1.

6.4.31 GM_Bezier

GM_Bezier are polynomial splines that use Bézier or Bernstein polynomials for interpolation purposes. An n-long control point array shall create a polynomial curve of degree n that defines the entire curve segment. These curves are defined in terms of a set of basis functions called the Bézier or Bernstein polynomials given by:

$$J_{n,i}(t) = \binom{n}{i} t^i (1-t)^{n-i} \quad \text{where} \quad \binom{n}{i} = \frac{n!}{i!(n-i)!} \quad \text{for i = 0, 1, 2 … n}$$

The set of “n+1” control points P_0, P_1… P_n, shall determine a curve segment given by:

$$\vec{c}(t) = \sum_{i=0}^{n} P_i J_{n,i}(t) \quad \text{for t} \in [0,1].$$

The sample points of this segment are the values of the curve defined at the maximum of each of the polynomials (i/n):

$$S_i = \vec{c}\left(\frac{i}{n}\right) \quad \text{for i = 0, 1, 2 … n}$$

NOTE For n = 1, the two weight functions are as follows:

$$J_{1,0}(t) = \binom{1}{0} t^0 (1-t)^1 = (1-t) \quad \text{and} \quad J_{1,1}(t) = \binom{1}{1} t^1 (1-t)^{1-1} = t$$

Given P_0 and P_1, the curve segment becomes:

$$\vec{c}(t) = (1-t)P_0 + tP_1 \quad \text{for } t \in [0,1].$$

In other words, for n = 1, the Bezier polynomial is geometrically equivalent to a simple line segment.

6.4.32 GM_SurfaceInterpolation

GM_SurfaceInterpolation (Figure 20) is a list of codes that may be used to identify the interpolation mechanisms specified by an application schema. Valid values for "interpolation" include, but are not limited, to the following:

a) None (none) – the interior of the surface is not specified. The assumption is that the surface follows the reference surface defined by the coordinate reference system.

b) Planar (planar) – the interpolation method shall return points on a single plane. The boundary in this case shall be contained within that plane.

c) Spherical (spherical), Elliptical (elliptical), Conic (conic) – the surface is a section of a spherical, elliptical or conic surface.

d) TIN (tin) – the control points are organized into adjoining triangles, which form small planar segments.

e) Parametric Curve (parametricCurve) – the control points are organized into a 2-dimensional grid and each cell within the grid is spanned by a surface which shall be defined by a family of curves.

f) Polynomial Spline (polynomialSpline) – the control points are organized into an irregular 2-dimensional grid and each cell within this grid is spanned by a polynomial spline function.

g) Rational Spline (rationalSpline) – the control points are organized into an irregular 2-dimensional grid and each cell within this grid is spanned by a rational (quotient of polynomials) spline function.

h) Triangulated Spline (triangulatedSpline) – the control points are organized into adjoining triangles, each of which is spanned by a polynomial spline function.

If more than one interpolation description fits the method used, then the most restrictive one will be used.

```
GM_SurfaceInterpolation::
      none
      planar
      spherical
      elliptical
      conic
      tin
      parametricCurve
      polynomialSpline
      rationalSpline
      triangualtedSpline
```

6.4.33 GM_GenericSurface

6.4.33.1 Semantics

GM_Surface and GM_SurfacePatch both represent sections of surface geometry, and therefore share a number of operation signatures. These are defined in the interface class GM_GenericSurface (Figure 20).

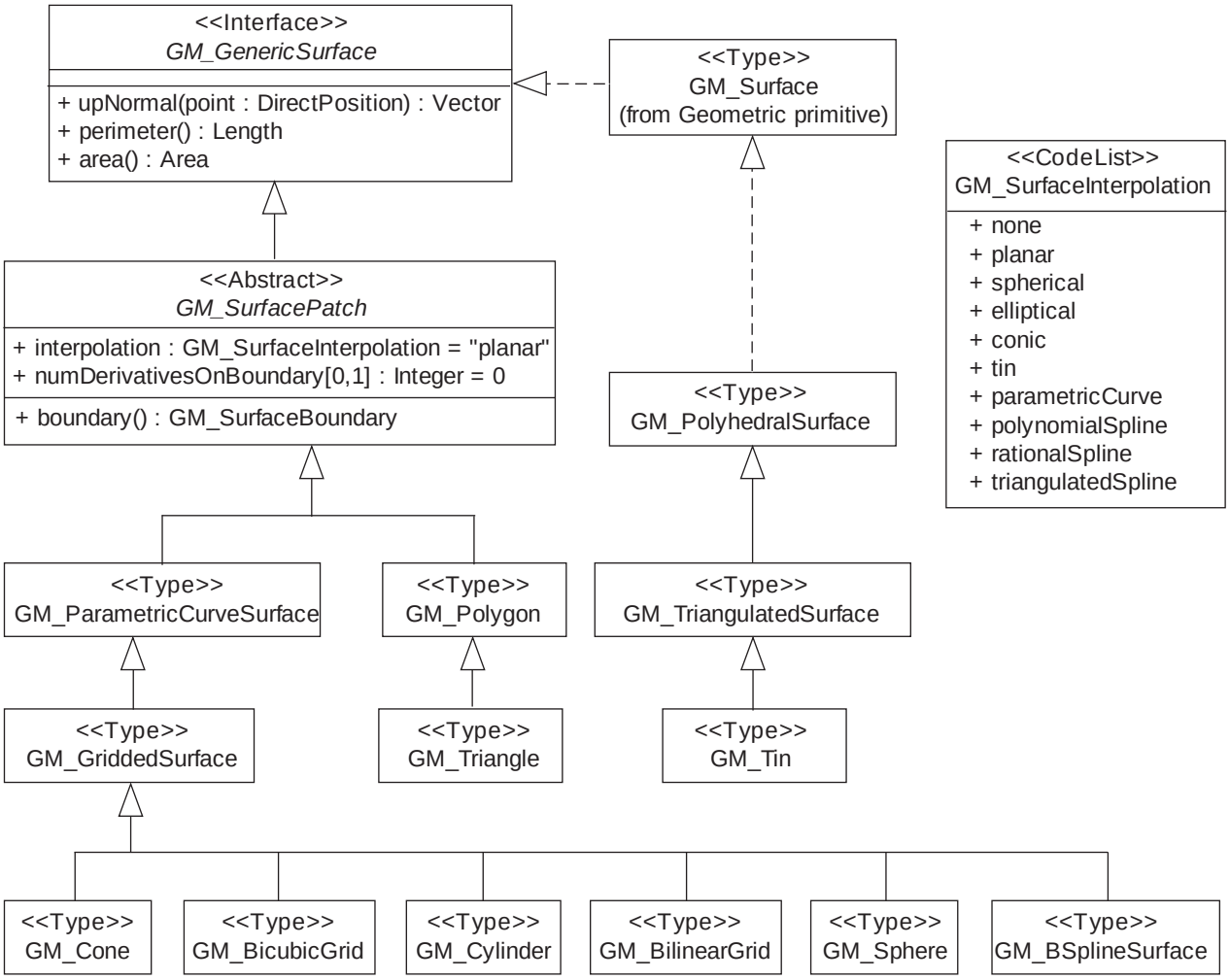


Figure 20 — Surface patches

6.4.33.2 upNormal

The operation “upNormal” returns a vector perpendicular to the GM_GenericSurface at the DirectPosition passed, which must be on the GM_GenericSurface.

```
GM_GenericSurface::upNormal(point : DirectPosition) : Vector
```

The upward normal always points upward in a manner consistent with the boundary. This means that the exterior boundary of the surface is counterclockwise when viewed from the side of the surface indicated by the upNormal. Interior boundaries are clockwise. The side of the surface indicated by the upNormal is referred to as the “top”. The function “upNormal” shall be continuous and the length of the normal shall always be equal to 1.0.

NOTE The upNormal along a boundary of a solid always points away from the solid. This is a slight semantics problem in dealing with voids within solids, where the upNormal (for sake of mathematical consistency) points into the centre of the voided region, which linguistically can be considered the interior of the void. What the confusion is here is that the basic linguistic metaphors used in most languages for “interior of solid” and for “interior of container” use “inward” in inconsistent manners from a topological point of view. The void “in” rock is not inside the rock in the same manner as the solid material that makes up the substance of the rock. Nor is the coffee “in” the cup the same “in” as the ceramic glass “in” the cup. The use of these culturally derived metaphors may not be consistent across all languages, some of which may use different prepositions for these two different concepts. This International Standard uses the linguistically neutral concept of “interior” derived from mathematics (topology).

6.4.33.3 perimeter

The operation “perimeter” shall return the sum of the lengths of all the boundary components of this GM_GenericSurface. Since perimeter, like length, is an accumulation (integral) of distance, its return value shall be in a reference system appropriate for measuring distances.

```
GM_GenericSurface::perimeter() : Length
```

NOTE The perimeter is defined as the sum of the lengths of all boundary components. The length of a curve or of a collection of curves is always positive and non-zero (unless the curve is pathological). This means that holes in surfaces will contribute positively to the total perimeter.

6.4.33.4 area

The area of a 2-dimensional geometric object shall be a numeric measure of its surface area (in a square unit of distance). Since area is an accumulation (integral) of the product of two distances, its return value shall be in a unit of measure appropriate for measuring distances squared, such as meters squared (m^2). The operation “area” shall return the area of this GM_GenericSurface.

```
GM_GenericSurface::area() : Area
```

The returned value shall take into account both the coordinate reference system and shape of the surface.

NOTE Consistent with the definition of surface as a set of DirectPositions, holes in the surfaces will not contribute to the total area. If the usual Green's Theorem (or more general Stokes' Theorem) integral is used, the integral around the holes in the surface are subtracted from the integral about the exterior of the surface patch.

6.4.34 GM_SurfacePatch

6.4.34.1 Semantics

GM_SurfacePatch (Figure 20) defines a homogeneous portion of a GM_Surface. The multiplicity of the association “Segmentation” (Figure 12) specifies that each GM_SurfacePatch shall be in at most one GM_Surface.

6.4.34.2 interpolation

The attribute “interpolation” determines the surface interpolation mechanism used for this GM_SurfacePatch. This mechanism uses the control points and control parameters defined in the various subclasses to determine the position of this GM_ SurfacePatch.

```
GM_SurfacePatch::Interpolation : GM_SurfaceInterpolation
```

6.4.34.3 numDerivativesOnBoundary

The attribute sequences “numDerivativesOnBoundary” specifies the type of continuity between this surface patch and its immediate neighbours with which it shares a boundary curve. The sequence of values corresponds to the GM_Rings in the GM_SurfaceBoundary returned by GM_GenericCurve::boundary for this patch. The default value of “0” means simple continuity, which is a mandatory minimum level of continuity. This level is referred to as “C^0” in mathematical texts. A value of one means that the functions are continuous and differentiable at the appropriate end point: “C^1” continuity. A value of “n” for any integer means n-times differentiable: “C^n” continuity.

```
GM_SurfacePatch::numDerivativesOnBoundary[0..1] : Integer
```

6.4.34.4 boundary

The operation “boundary” shall return the boundary of this GM_SurfacePatch represented as a collection of GM_OrientableCurves organized into GM_Rings by a GM_SurfaceBoundary.

```
GM_SurfacePatch::boundary() : GM_SurfaceBoundary
```

NOTE The semantics of this operation is the same as that of GM_Surface::boundary, except that the curves used here may be not be persistent GM_OrientableCurve instances. Transient data type values of GM_Curve are also valid. In the normal case, GM_SurfacePatches will share parts of their boundary with the aggregate GM_Surface, and other parts with GM_SurfacePatches (not necessarily distinct). In Annex C, the solid example (C.1.3) uses a single patch folded back on itself to form a topological cylinder, with two square end pieces to form a solid boundary. In this case, the first patch shares one boundary segment with each of the two endcaps, and another with itself.

6.4.35 GM_PolyhedralSurface

6.4.35.1 Semantics

A GM_PolyhedralSurface (Figure 21) is a GM_Surface composed of polygon surfaces (GM_Polygon) connected along their common boundary curves. This differs from GM_Surface only in the restriction on the types of surface patches acceptable.

6.4.35.2 GM_PolyhedralSurface (constructor)

The constructor for a GM_PolyhedralSurface takes the facet GM_Polygons and creates the necessary aggregate surface.

```
GM_PolyhedralSurface::GM_PolyhedralSurace(tiles[1..n]: GM_Polygon ) :
      GM_PolyhedralSurface
```

6.4.35.3 patch

The association role “patch” associates this surface with its individual facet polygons. It shall be non-empty.

```
GM_PolyhedralSurface::patch[1,n] : Reference<GM_Polygon>
```

6.4.36 GM_Polygon

6.4.36.1 Semantics

A GM_Polygon (Figure 21) is a surface patch that is defined by a set of boundary curves and an underlying surface to which these curves adhere. The default is that the curves are coplanar and the polygon uses planar interpolation in its interior.

6.4.36.2 boundary

The attribute “boundary” stores the GM_SurfaceBoundary that is the boundary of this GM_Polygon.

```
GM_Polygon::boundary : GM_SurfaceBoundary
```

NOTE The boundary of a surface patch need not be in the same GM_Complex as the containing GM_Surface. The curves that are contained in the interior of the GM_Surface (act as common boundary to two surface patches) are not part of any GM_Complex in which the GM_Surface is contained. They are purely constructive and would not play in any topological relation between GM_Surface and GM_Curve that defines the connectivity of the GM_Complex.

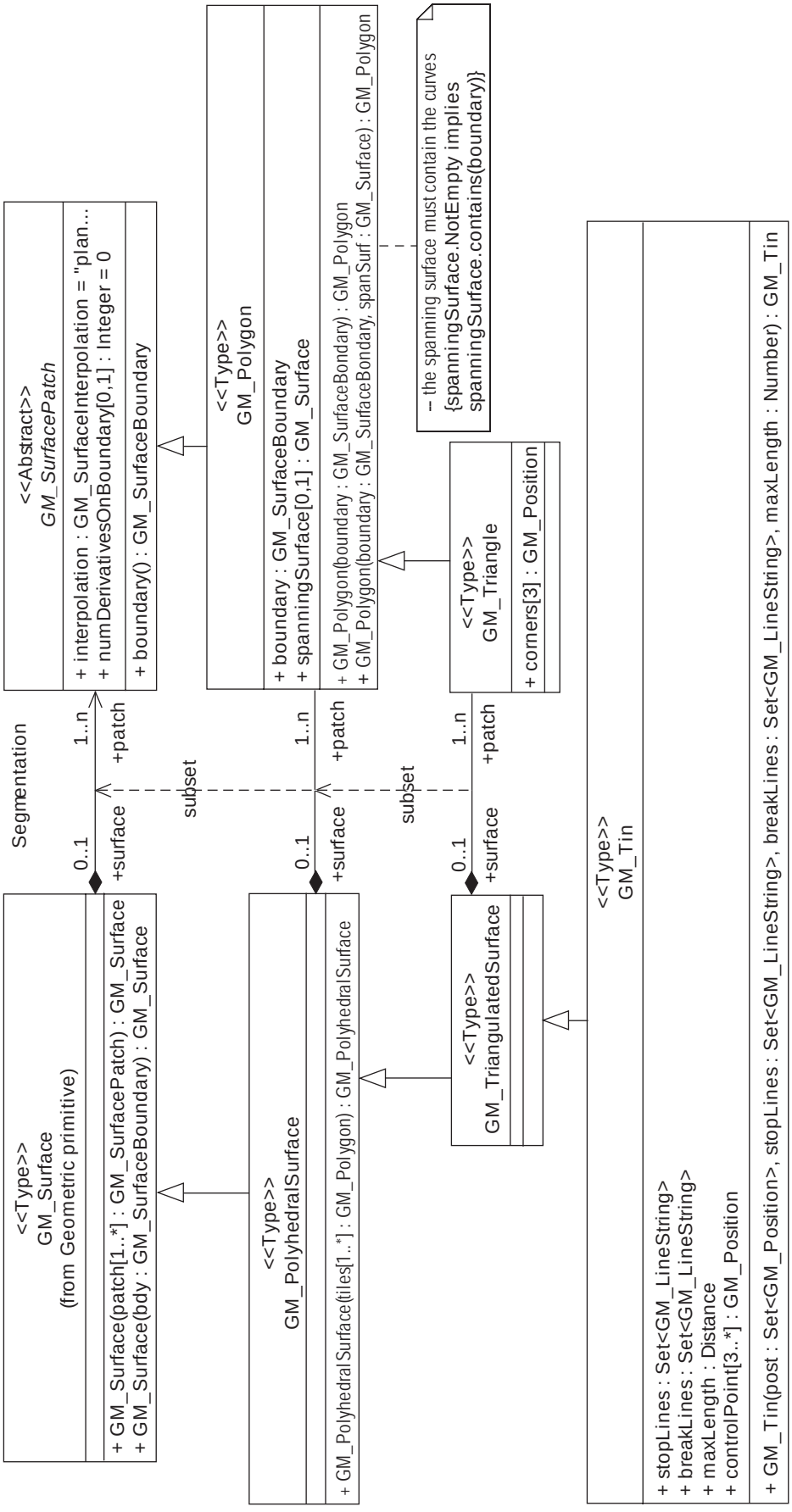


Figure 21 — Polygonal surface

6.4.36.3 spanningSurface

The optional spanning surface provides a mechanism for spanning the interior of the polygon.

```
GM_Polygon::spanningSurface [0,1] : GM_Surface
```

NOTE The spanning surface should have no boundary components that intersect the boundary of the polygon, and there should be no ambiguity as to which portion of the surface is described by the bounding curves for the polygon. The most common spanning surface is an elevation model, which is not directly described in this International Standard, although Tins and gridded surfaces are often used in this role.

6.4.36.4 GM_Polygon (constructor)

This first variant of a constructor of GM_Polygon creates a GM_Polygon directly from a set of boundary curves (organized into a GM_SurfaceBoundary) which shall be defined using coplanar GM_Positions as controlPoints.

```
GM_Polygon::GM_Polygon(boundary : GM_SurfaceBondary) : GM_Polygon
```

NOTE The meaning of exterior in the GM_SurfaceBoundary is consistent with the plane of the constructed planar polygon.

This second variant of a constructor of GM_Polygon creates a GM_Polygon lying on a spanning surface. There is no restriction of the types of interpolation used by the composite curves used in the GM_SurfaceBoundary, but they must all be lie on the “spanningSurface” for the process to succeed.

```
GM_Polygon(boundary : GM_SurfaceBondary, spanSurf : GM_Surface) : GM_Polygon
```

NOTE It is important that the boundary components be oriented properly for this to work. It is often the case that in bounded manifolds, such as the sphere, there is an ambiguity unless the orientation is properly used.

6.4.37 GM_TriangulatedSurface

A GM_TriangulatedSurface (Figure 21) is a GM_PolyhedralSurface that is composed only of triangles (GM_Triangle). There is no restriction on how the triangulation is derived.

6.4.38 GM_Triangle

A GM_Triangle is a planar GM_Polygon defined by three corners; that is, a GM_Triangle would be the result of a constructor of the form:

```
GM_Polygon(GM_LineString(<P1, P2, P3, P1>))
```

where P1, P2, and P3 are three GM_Positions. GM_Triangles have no holes. GM_Triangle shall be used to construct GM_TriangulatedSurfaces.

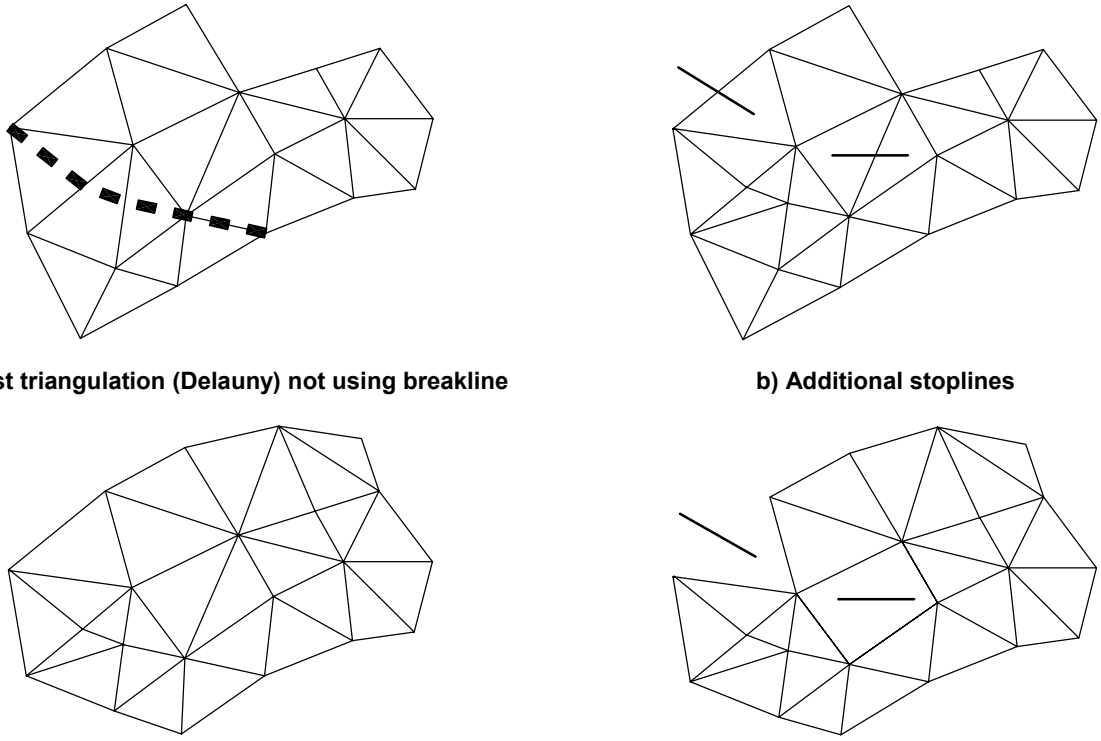
NOTE The points in a triangle can be located in terms of their corner points by defining a set of barycentric coordinates, three nonnegative numbers c_1, c_2, and c_3 such that $c_1 + c_2 + c_3 = 1.0$. Then, each point P in the triangle can be expressed for some set of barycentric coordinates as:

$$P = c_1 P_1 + c_2 P_2 + c_3 P_3$$

6.4.39 GM_Tin

6.4.39.1 Semantics

A GM_Tin (Figure 21) is a GM_TriangulatedSurface that uses the Delaunay algorithm or a similar algorithm complemented with consideration for breaklines, stoplines and maximum length of triangle sides (Figure 22). These networks satisfy the Delaunay criterion away from the modifications: For each triangle in the network, the circle passing through its vertexes does not contain, in its interior, the vertex of any other triangle.



a) First triangulation (Delauny) not using breakline

b) Additional stoplines

c) Retriangulation using breakline

d) Retriangulation using stoplines with resulting holes and boundary changes

Figure 22 — TIN construction

6.4.39.2 stopLines

Stoplines are lines where the local continuity or regularity of the surface is questionable. In the area of these pathologies, triangles intersecting a stopline shall be removed from the TIN surface, leaving holes in the surface. If coincidence occurs on surface boundary triangles, the result shall be a change of the surface boundary. The attribute “stopLines” contains all these pathological segments as a set of line strings.

```
GM_Tin::stopLines : Set<GM_LineString>
```

6.4.39.3 breakLines

Breaklines are lines of a critical nature to the shape of the surface, representing local ridges, or depressions (such as drainage lines) in the surface. As such their constituent segments must be included in the TIN even if doing so violates the Delaunay criterion. The attribute “breakLines” contains these critical segments as a set of line strings.

```
GM_Tin::breakLines : Set<GM_LineString>
```

6.4.39.4 maxLength

Areas of the surface where the data is not sufficiently dense to assure reasonable calculations shall be removed by adding a retention criterion for triangles based on the length of their sides. For any triangle sides exceeding maximum length, the adjacent triangles to that triangle side shall be removed from the surface.

```
GM_Tin::maxLength : Distance
```

6.4.39.5 controlPoint

The corners of the triangles in the TIN are often referred to as posts. The attribute "controlPoint" shall contain a set of the GM_Positions used as posts for this TIN. Since each TIN contains triangles, there must be at least three posts. The order in which these points are given does not affect the surface that is represented. Application schemas may add information based on the ordering of the control points to facilitate the reconstruction of the TIN from the controlPoints.

```
GM_Tin::controlPoint[3..n] : GM_Position
```

NOTE The control points of a TIN are often called "posts".

6.4.39.6 GM_Tin (constructor)

The constructor for a restricted Delaunay network requires the triangle corners (posts), breaklines, stoplines, and maximum length of a triangle side.

```
GM_Tin::GM_Tin(post : Set<GM_Position>, stopLines : Set<GM_LineString>,
      breakLines : Set<GM_LineString>, maxLength : Number): GM_Tin
```

6.4.40 GM_ParametricCurveSurface

6.4.40.1 Semantics

The surface patches that make up the parametric curve surfaces, GM_ParametricCurveSurface (Figure 23), are all continuous families of curves, given by a constructive function of the form:

```
surface(s,t): [a,b]×[c,d] →DirectPosition
```

By fixing the value of either parameter, we have a one-parameter family of curves.

```
c_t(s) = c_s(t) = surface(s,t)
```

The functions on GM_ParametricCurveSurface (Figure 23) shall expose these two families of curves. The first gives us the "horizontal" cross sections $c_t(s)$, the later the "vertical" cross sections $c_s(t)$. The terms "horizontal" and "vertical" refer to the parameter space and need not be either horizontal or vertical curves in the coordinate reference system. Table 7 lists some possible pairs of types for these surface curves (other representations of these same surfaces are possible). The two partial derivatives of the surface parameterization, i and j are given by:

$$i \equiv \frac{dsurface}{ds} = \frac{d}{ds} c_t(s) = \frac{\partial}{\partial s} surface(s,t)$$

and

$$j \equiv \frac{dsurface}{dt} = \frac{d}{dt} c_s(t) = \frac{\partial}{\partial t} surface(s,t)$$

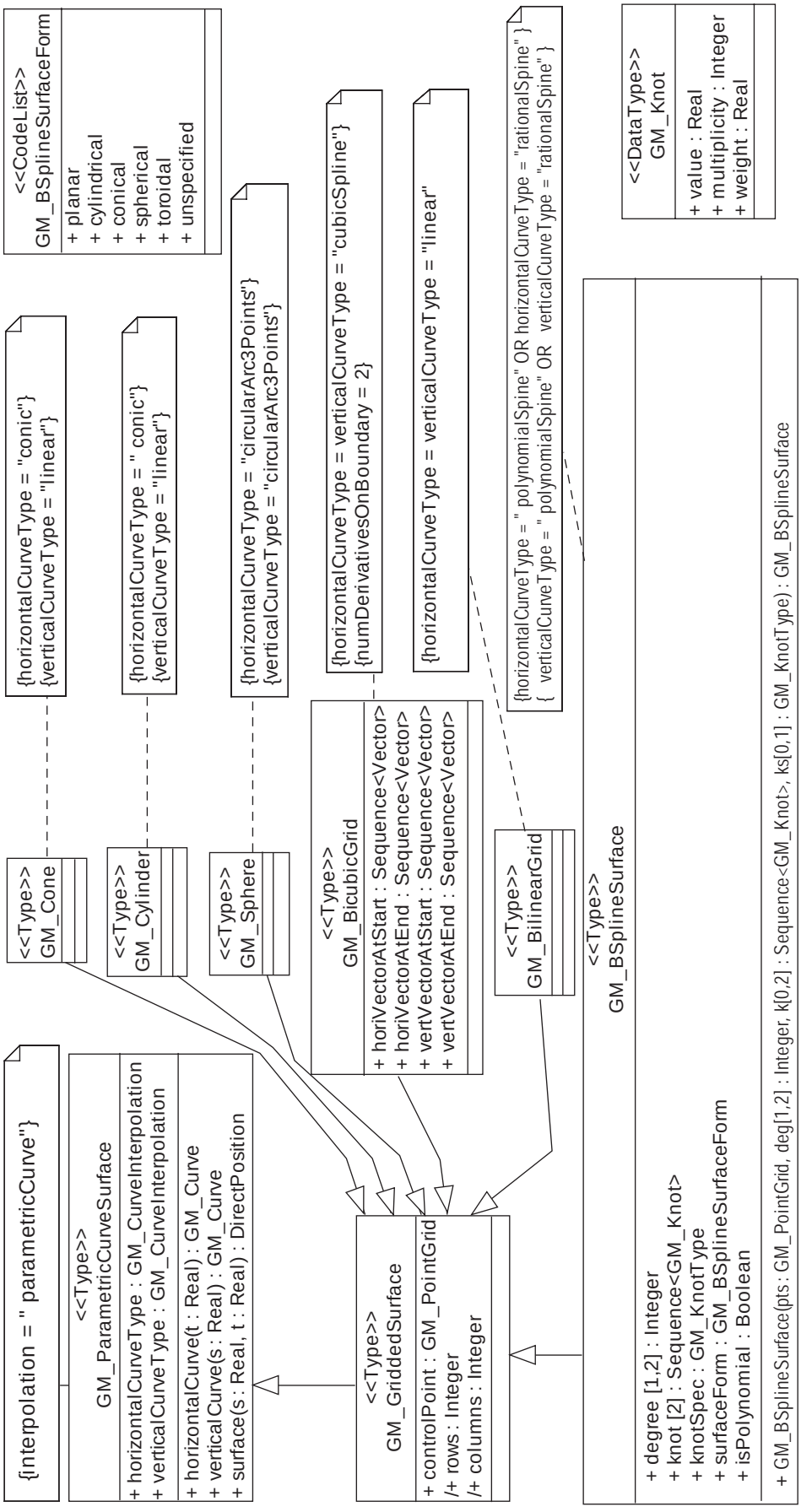


Figure 23 — GM_ParametricCurveSurface and its subtypes

The default upNormal for the surface shall be the vector cross product of these two curve derivatives when they are both non-zero:

$$k = i \times j$$

If the coordinate reference system is 2D, then the vector k extends the local coordinate system by supplying an "upward" elevation vector. In this case the vector basis (i, j) must be a right hand system, that is to say, the oriented angle from i to j must be less than 180°. This gives a right-handed "moving frame" of local coordinate axes given by $<i, j>$. A moving frame is defined to be a continuous function from the geometric object to a basis for the local tangent space of that object. For curves, this is the derivative of the curve, the local tangent. For surfaces, this is a local pair of tangents. Parameterized curve surfaces have a natural moving frame and it shall be used as defined in this paragraph to define the upNormal of the surface.

NOTE The existence of a viable moving frame is the definition of "orientable" manifold. This is why the existence of a continuous upNormal implies that the surface is orientable. Non-orientable surfaces, such as the Möbius band and Klein bottle are counter-intuitive. 6.3.17.1 forbids their use in application schemas conforming to this International Standard. Klein bottles cannot even be constructed in 3D space, but require 4D space for non-singular representations.

Table 7 — Various types of parametric curve surfaces

Surface type	Horizontal curve type	Vertical curve type
GM_Cylinder	Circle, constant radii	Line Segment
GM_Cone	Circle, decreasing radii	Line Segment
GM_Sphere	Circle of constant latitude	Circle of constant longitude
GM_BilinearGrid	Line string	Line string
GM_BicubicGrid	Cubic spline	Cubic spline

6.4.40.2 horizontalCurveType

The attribute "horizontalCurveType" indicates the type of surface curves used to traverse the surface horizontally with respect to the parameter "s".

```
GM_ParametricCurveSurface::horizontalCurveType : GM_CurveInterpolation
```

6.4.40.3 verticalCurveType

The attribute "verticalCurveType" indicates the type of surface curves used to traverse the surface vertically with respect to the parameter "t".

```
GM_ParametricCurveSurface::verticalCurveType : GM_CurveInterpolation
```

6.4.40.4 horizontalCurve

The operation "horizontalCurve" constructs a curve that traverses the surface horizontally with respect to the parameter "s". This curve holds the parameter "t" constant.

```
GM_ParametricCurveSurface::horizontalCurve(t : Real) : GM_Curve
```

NOTE The GM_Curve returned by this function or by the corresponding vertical curve function, are normally not part of any GM_Complex to which this surface is included. These are, in general, calculated transient values. The exceptions to this may occur at the extremes of the parameter space. The boundaries of the parameter space support for the surface map normally to the boundaries of the target surfaces.

6.4.40.5 verticalCurve

The operation “verticalCurve” constructs a curve that traverses the surface vertically with respect to the parameter “t”. This curve holds the parameter “s” constant.

```
GM_ParametricCurveSurface::verticalCurve(s : Real) : GM_Curve
```

6.4.40.6 surface

The operation “surface” traverses the surface both vertically and horizontally.

```
GM_ParametricCurveSurface::surface(s : Real, t : Real) : DirectPosition
```

6.4.41 GM_GriddedSurface

6.4.41.1 Semantics

The GM_GriddedSurface (Figure 23) is a GM_ParametricCurveSurface defined from a rectangular grid in the parameter space. The rows from this grid are control points for horizontal surface curves; the columns are control points for vertical surface curves. The working assumption is that for a pair of parametric coordinates (s, t), that the horizontal curves for each integer offset are calculated and evaluated at “s”. This defines a sequence of control points:

```
<c_n(s) : s = 1 … columns>
```

From this sequence, a vertical curve is calculated for “s”, and evaluated at “t”. In most cases, the order of calculation (horizontal-vertical versus vertical-horizontal) does not make a difference. Where it does, the horizontal-vertical order shall be the one used.

NOTE The most common case of a gridded surface is a 2D spline. In this case the weight functions for each parameter make order of calculation unimportant:

$$surface(s,t) = \sum_{i=0}^{row} \sum_{j=0}^{columns} w^{s}{}_{i}(s) w^{t}{}_{j}(t) \vec{P}_{i,j}$$

where $\vec{P}_{i,j}$ is the control point in the i^{th} row and j^{th} column.

Logically, any pair of curve interpolation types can lead to a subtype of GM_GriddedSurface. The following clauses define some of the most commonly encountered surfaces that can be represented in this manner.

6.4.41.2 controlPoint

This is the doubly indexed sequence of control points, given in row major form.

```
GM_GriddedSurface::controlPoint : GM_PointGrid
```

NOTE There is no assumption made about the shape of the grid. For example, the positions need not effect a “2½D” surface, consecutive points may be equal in any or all of their ordinates. Further, the curves in either or both directions may close.

6.4.41.3 rows

The derived attribute “rows” gives the number of rows in the parameter grid.

```
GM_GriddedSurface::rows : Integer = controlPoint→row.count : Integer
```

6.4.41.4 columns

The derived attribute “columns” gives the number of columns in the parameter grid.

```
GM_GriddedSurface::rows : Integer = controlPoint→row→column.count : Integer
```

6.4.42 GM_Cone

A GM_Cone is a GM_GriddedSurface given as a family of conic sections whose controlPoints vary linearly.

NOTE A 5-point ellipse with all defining positions identical is a point. Thus, a truncated elliptical cone can be given as a 2×5 set of control points <<P1, P1, P1, P1, P1>, <P2, P3, P4, P5, P6>>. P1 is the apex of the cone. P2, P3, P4, P5, and P6 are any five distinct points around the base ellipse of the cone. If the horizontal curves are circles as opposed to ellipses; then a circular cone can be constructed using <<P1, P1, P1>, <P2, P3, P4>>.

6.4.43 GM_Cylinder

A GM_Cylinder is a GM_GriddedSurface given as a family of circles whose positions vary along a set of parallel lines, keeping the cross sectional horizontal curves of a constant shape.

NOTE Given the same working assumptions as in the previous note, a GM_Cylinder can be given by two circles, giving us control points of the form <<P1, P2, P3>, <P4, P5, P6>>.

6.4.44 GM_Sphere

A GM_Sphere is a GM_GriddedSurface given as a family of circles whose positions vary linearly along the axis of the sphere, and whose radius varies in proportion to the cosine function of the central angle. The horizontal circles resemble lines of constant latitude, and the vertical arcs resemble lines of constant longitude.

NOTE If the control points are sorted in terms of increasing longitude, and increasing latitude, the upNormal of a sphere is the outward normal.

EXAMPLE If we take a gridded set of latitudes and longitudes in degrees, (u, v), such as

(–90, –180)	(–90, –90)	(–90,0)	(–90, 90)	(–90, 180)
(–45, –180)	(–45, –90)	(–45,0)	(–45, 90)	(–45, 180)
(0, –180)	(0, –90)	(0,0)	(0, 90)	(0, 180)
(45, –180)	(45, –90)	(45,0)	(45, –90)	(45, 180)
(90, –180)	(90, –90)	(90,0)	(90, –90)	(90, 180)

And map these points to 3D using the usual equations (where R is the radius of the required sphere).

```
z = R sin u
x = (R cos u) (sin v)
y = (R cos u) (cos v)
```

We have a sphere of radius R, centered at (0, 0), as a gridded surface. Notice that the entire first row and the entire last row of the control points map to a single point each in 3D Euclidean space, North and South poles respectively, and that each horizontal curve closes back on it self forming a geometric cycle. This gives us a metrically bounded (of finite size), topologically unbounded (not having a boundary, a cycle) surface.

6.4.45 GM_BilinearGrid

A GM_BilinearGrid is a GM_GriddedSurface that uses line strings as the horizontal and vertical curves.

NOTE This is not a polygonal surface, since each of the grid squares is a ruled surface, and not necessarily planar.

6.4.46 GM_BicubicGrid

6.4.46.1 Semantics

A GM_BicubicGrid is a GM_GriddedSurface that uses cubic polynomial splines as the horizontal and vertical curves.

NOTE The initial tangents for the splines are often replaced by an extra pair of rows (and columns) of control points.

6.4.46.2 horiVectorAtEnd, horiVectorAtStart, vertVectorAtEnd, vertVectorAtStart

The horizontal and vertical curves require initial and final tangent vectors for a complete definition. These values are supplied by four attributes:

```
GM_BicubicSpline::horiVectorAtEnd : Sequence<Vector>;
GM_BicubicSpline::horiVectorAtStart : Sequence<Vector>;
GM_BicubicSpline::vertVectorAtEnd : Sequence<Vector>;
GM_BicubicSpline::vertVectorAtStart : Sequence<Vector>;
```

6.4.47 GM_BSplineSurfaceForm

The code list “GM_BSplineSurfaceForm” shall be used to indicate a particular geometric form represented by a GM_BSplineSurface. The potential values are:

— planar — a bounded portion of a plane represented by a B-spline surface of degree 1 in each parameter.

— cylindrical — a bounded portion of a cylindrical surface represented by a B-spline surface.

— conical — a bounded portion of the surface of a right circular cone represented by a B-spline surface.

— spherical — a bounded portion of a sphere, or a complete sphere represented by a B-spline surface.

— toroidal — a torus or a portion of a torus represented by a B-spline surface.

— unspecified — no particular surface is specified.

```
GM_BSplineSurfaceForm::
      planar
      cylindrical
      conical
      spherical
      toroidal
      unspecified
```

6.4.48 GM_BSplineSurface

6.4.48.1 Semantics

A B-spline surface is a rational or polynomial parametric surface that is represented by control points, basis functions and possibly weights. If the weights are all equal then the spline is piecewise polynomial. If they are not equal, then the spline is piecewise rational. If the Boolean “isPolynomial” is set to TRUE then the weights shall all be set to 1.

6.4.48.2 degree

The attribute “degree” shall be the algebraic degree of the basis functions for the first and second parameter. If only one value is given, then the two degrees are equal.

```
GM_BSplineSurface::degree [1,2] : Integer
```

6.4.48.3 surfaceForm

The attribute “surfaceForm” is used to identify particular types of surface which this spline is being used to approximate. It is for information only, used to capture the original intention. If no such approximation is intended, then the value of this attribute is NULL.

```
GM_ BSplineSurface::surfaceForm: GM_BSplineSurfaceForm
```

6.4.48.4 knot

The attribute “knot” shall be two sequences of distinct knots used to define the B-spline basis functions for the two parameters. Recall that the knot data type holds information on knot multiplicity.

```
GM_BSplineSurface::knot [2] : Sequence<GM_Knot>
```

6.4.48.5 knotSpec

The attribute “knotSpec” gives the type of knot distribution used in defining this spline. This is for information only and is set according to the different construction-functions.

```
GM_BSplineSurface::knotSpec[0,1] : GM_KnotType
```

6.4.48.6 isPolynomial

The attribute “isPolynomial” is set to “True” if this is a polynomial spline.

```
GM_BSplineSurface::isPolynomial : Boolean
```

6.4.48.7 GM_BSplineSurface (constructor)

The class constructor “GM_BSplineSurface” takes the pertinent information described in the attributes above and constructs a B-spline surface. If the knotSpec is not present, then the knotType is uniform and the knots are evenly spaced, and, except for the first and last, have multiplicity = 1. At the ends the knots are of multiplicity = degree+1. If the knotType is uniform they need not be specified.

```
GM_BSplineSurface::GM_BSplineSurface(
      pts : Sequence<GM_PointArray>,
      deg[1,2] : Integer,
      k[0,2] : Sequence<GM_Knot>,
      ks[0,1] : GM_KnotType) : GM_BSplineSurface
```

6.5 Geometric aggregate package

6.5.1 Semantics

Arbitrary aggregations of geometric objects are possible. These are not assumed to have any additional internal structure and are used to “collect” pieces of geometry of a specified type. In this respect they differ from “composites” and “complexes”, which are defined in 6.6. Operations on these aggregations shall be the

accumulators that are derived from the class operations of their elements. Applications may use aggregates for features that use multiple geometric objects in their representations, such as a collection of points to represent a tank farm or orchard.

6.5.2 GM_Aggregate

6.5.2.1 Semantics

The aggregates, GM_Aggregates (Figure 24) gather geometric objects. Since they will often use orientation modification, the curve reference and surface references do not go directly to the GM_Curve and GM_Surface, but are directed to GM_OrientableCurve and GM_OrientableSurface.

Most geometric objects are contained in features, and cannot be held in collections that are strong aggregations. For this reason, the collections described in this clause are all weak aggregations, and shall use references to include geometric objects. The type relation between the various reference objects is given below.

NOTE The subclasses of GM_OrientablePrimitive are handled in such a manner that the reference object can link to a specific orientation of that object.

6.5.2.2 element

The association role “element” shall be the set of GM_Objects contained in this GM_Aggregate. In subclasses of GM_Aggregate, the elements shall be restricted to specific types of GM_Primitives.

```
GM_Aggregate::element : Set<GM_ObjectRef>
```

6.5.2.3 fromSet

The operation “fromSet” shall be a constructor that takes a set of the GM_Objects and creates a GM_Aggregate.

```
GM_Aggregate::fromSet(set : Set<GM_Object>) : GM_Aggregate
```

6.5.3 GM_MultiPrimitive

GM_MultiPrimitive is the root class for all primitive aggregates. The association role “element” shall be the set of GM_Primitives contained in this GM_MultiPrimitive. The attribute declaration here specializes the one at GM_Aggregate to include only GM_Primitives in this type of aggregate.

```
GM_MultiPrimitive::element : Set<GM_Primitive>
```

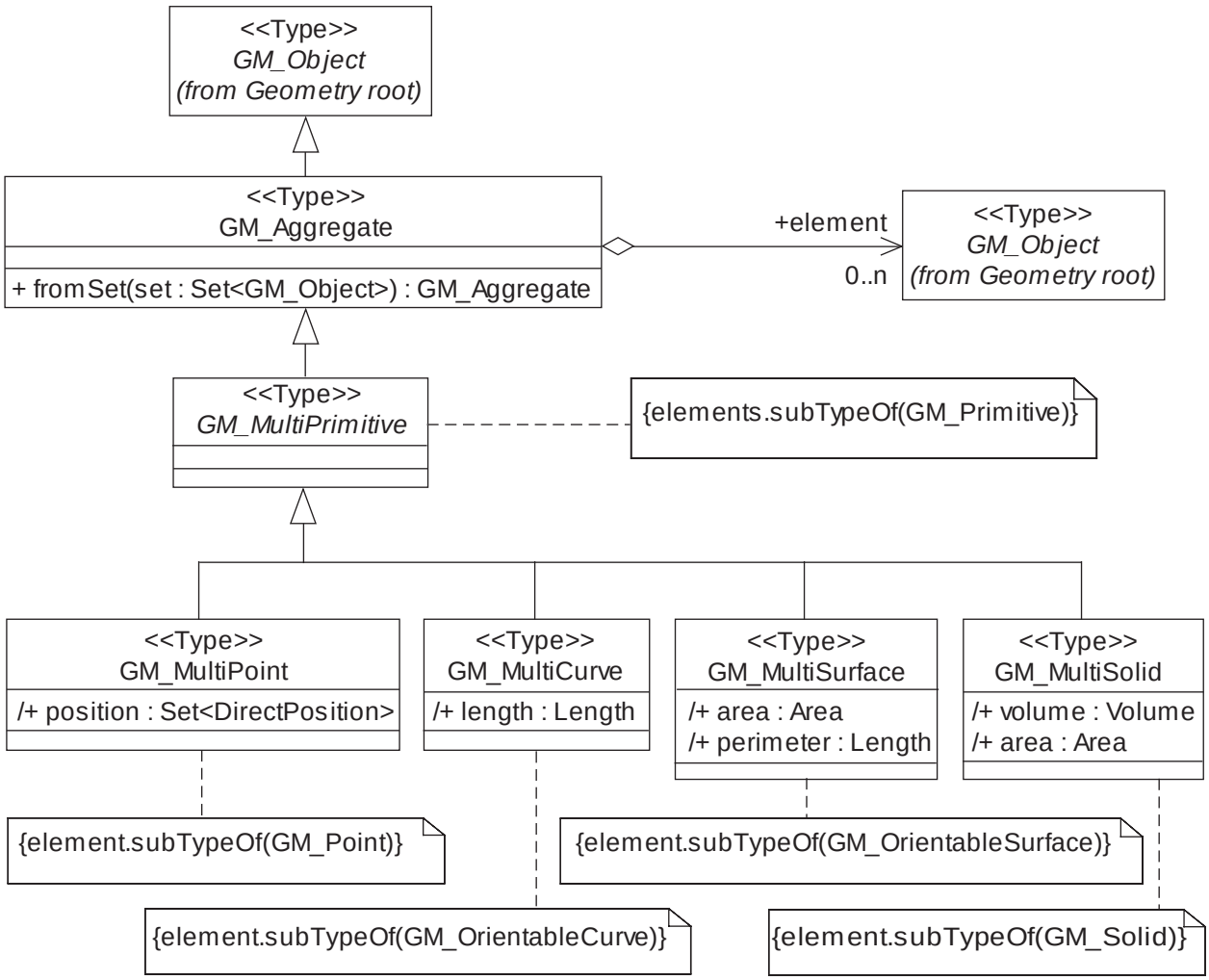


Figure 24 — GM_Aggregate

6.5.4 GM_MultiPoint

6.5.4.3 Semantics

GM_MultiPoint is an aggregate class containing only points. The association role “element” shall be the set of GM_Points contained in this GM_MultiPoint.

```
GM_MultiPoint::element : Set<GM_Point>
```

6.5.4.4 position

The derived attribute “position” shall be the set of DirectPositions of the GM_Points contained in this GM_MultiPoint.

```
GM_MultiPoint::position : Set<DirectPosition>
```

6.5.5 GM_MultiCurve

6.5.5.1 Semantics

GM_MultiCurve is an aggregate class containing only instances of GM_OrientableCurve. The association role “element” shall be the set of GM_OrientableCurves contained in this GM_MultiCurve.

```
GM_MultiCurve::element : Set<GM_OrientableCurve>
```

6.5.5.2 length

The derived attribute “length” shall be the accumulated length of all the GM_Curves contained in this GM_MultiCurve.

```
GM_MultiCurve::length : Length
```

6.5.6 GM_MultiSurface

6.5.6.1 Semantics

GM_MultiSurface is an aggregate class containing only instances of GM_OrientableSurface. The association role “element” shall be the set of GM_OrientableSurfaces contained in this GM_MultiSurface.

```
GM_MultiSurface::element : Set<GM_OrientableSurface>
```

6.5.6.2 area

The derived attribute “area” shall be the accumulated area of all the GM_Surfaces contained in this GM_MultiSurface.

```
GM_MultiSurface::area : Area
```

6.5.6.3 perimeter

The derived attribute “perimeter” shall be the accumulated perimeter of all the GM_Surfaces contained in this GM_MultiSurface.

```
GM_MultiSurface::perimeter : Length
```

6.5.7 GM_MultiSolid

6.5.7.1 Semantics

GM_MultiSolid is an aggregate class containing only solids. The association role “element” shall be the set of GM_Solids contained in this GM_MultiSolid.

```
GM_MultiSolid::element : Set<GM_Solid>
```

6.5.7.2 volume

The derived attribute “volume” shall be the accumulated volume of all the GM_Solids contained in this GM_MultiSolid.

```
GM_MultiSolid::volume : Volume
```

6.5.7.3 area

The derived attribute “area” shall be the accumulated surface area of all the GM_Solids contained in this GM_MultiSolid.

```
GM_MultiSolid::area : Area
```

6.6 Geometric complex package

6.6.1 Semantics

A geometric complex (GM_Complex) is a set of primitive geometric objects (in a common coordinate system) whose interiors are disjoint. Further, if a primitive is in a geometric complex, then there exists a set of primitives in that complex whose point-wise union is the boundary of this first primitive.

A subcomplex of a complex is a subset of the primitives of that complex that is, in its own right, a geometric complex. A supercomplex of a complex is a superset of primitives that is also a complex. These definitions are essentially subset and superset with the added restriction that they must be a complex. A complex is maximal if it is a subcomplex of no larger complex.

The boundary of a geometric object in a geometric complex is a subcomplex of that complex. The simplest complex is a single point. The simplest 1-dimensional complex is a curve with its two end points. The simplest 2-dimensional complex is a surface with its boundary curve, and the curve's start and end points.

The underlying geometry of a complex is usually referred to as a “manifold”. The structure of a complex organizes the geometry of the manifold into primitive elements, analogously to the way in which “charts” are organized by an “atlas” into a map of the world.

One way, but obviously not the only way, to generate a complex from a set of primitives is by beginning with those primitives and performing the following operations.

a) If two primitives overlap, then subdivide them, eliminating repetitions until there is no overlap.

b) Similarly, if a primitive is not simple, subdivide it where it intersects itself, eliminating repetitions until there is no overlap.

c) If a primitive is not a point, calculate its boundary as a collection of other primitives, using those already in the generating set if possible, and insert them into the complex.

d) Repeat step “a” through “c” until no new primitive is required.

Many systems have a concept of a universal face (for 2D) or universal solid (for 3D). This is valid only in the case where the underlying space of the complex is an unbounded Euclidean space. In this case, for 2D, the universal face is the surface in the GM_Complex that has only interior boundary rings (its exterior one being the “point at infinity”). Analogously, in 3D, the universal solid is the one that has only interior boundary shells. In bounded manifolds, such as the sphere, there is no point at infinity, and all primitives are bounded. Without the Jordan Separation Theorem, all boundaries are essentially interior boundaries. In other unbounded manifolds, such as a hyperbolic surface, there may be more than one unbounded primitive. Since this International Standard does not directly address these sorts of unbounded manifolds, the cardinality of some elements may require relaxing if this International Standard were to be applied to such non-geographic manifolds. This International Standard does not special case either the universal face or solid, and the

relationship between them and their boundaries are represented in the same manner as any other boundary relationship.

NOTE A maximal complex could reasonably be considered a strong aggregation of its primitives depending on the internal semantics of the application. For this reason, the mechanism for the containment of GM_Primitives in a GM_Complex is left unspecified. If a strong aggregation is used for maximal complexes, then the containment association for subcomplexes may have to use the maximal complex as a namespace for the references to primitives within it. In any case, once a GM_Primitive is within a complex, or a GM_Complex is a subcomplex of a maximal GM_Complex, its boundary operation will not need to construct representative GM_Objects, since by the definition of a complex, the objects needed to represent the boundary of the contained object will already exist, and only references to those objects are required by the GM_Object::boundary operation. Remember that the containment of GM_Complexes in one another is a subset-superset association, while the containment of GM_Primitives in a GM_Complex is an element-set association.

6.6.2 GM_Complex

6.6.2.1 Semantics

A GM_Complex (Figure 25) is a collection of geometrically disjoint, simple GM_Primitives. If a GM_Primitive (other than a GM_Point) is in a particular GM_Complex, then there exists a set of primitives of lower dimension in the same complex that form the boundary of this primitive.

NOTE A geometric complex can be thought of as a set in two distinct ways. First, it is a finite set of objects (via delegation to its elements member) and, second, it is an infinite set of point values as a subtype of geometric object. The dual use of delegation and subtyping is to disambiguate the two types of set interface. To determine if a GM_Primitive P is an element of a GM_Complex C, call: C.element().contains(P).

The “element” attribute allows GM_Complex to inherit the behavior of Set<GM_Primitive> without confusing the same sort of behaviour inherited from TransfiniteSet<DirectPosition> inherited through GM_Object.

Complexes shall be used in application schemas where the sharing of geometry is important, such as in the use of computational topology. In a complex, primitives may be aggregated many-to-many into composites for use as attributes of features. Examples of this are provided in the schemas in Annex D.

6.6.2.2 isMaximal

The Boolean valued operation “isMaximal” shall return TRUE if and only if this GM_Complex is maximal.

```
GM_Complex::isMaximal() : Boolean
```

6.6.2.3 Contains association

The association “Contains” instantiates the contains operation from Set<GM_Primitive> as an association.

```
GM_Complex::subComplex [0..n] : GM_Complex
GM_Complex::superComplex [0..n] : GM_Complex
```

6.6.2.4 Complex association

The association “Complex” is defined by the “contains” operation in GM_Object that is inherited from TransfiniteSet<DirectPosition>.

```
GM_Complex::element [1..n] : GM_Primitive
```

If a complex contains a GM_Primitive, then it must also contain the elements of its boundary.

```
GM_Complex:
      -- closed under the boundary operation
      self→forAll(self→includesAll(boundary()))
```

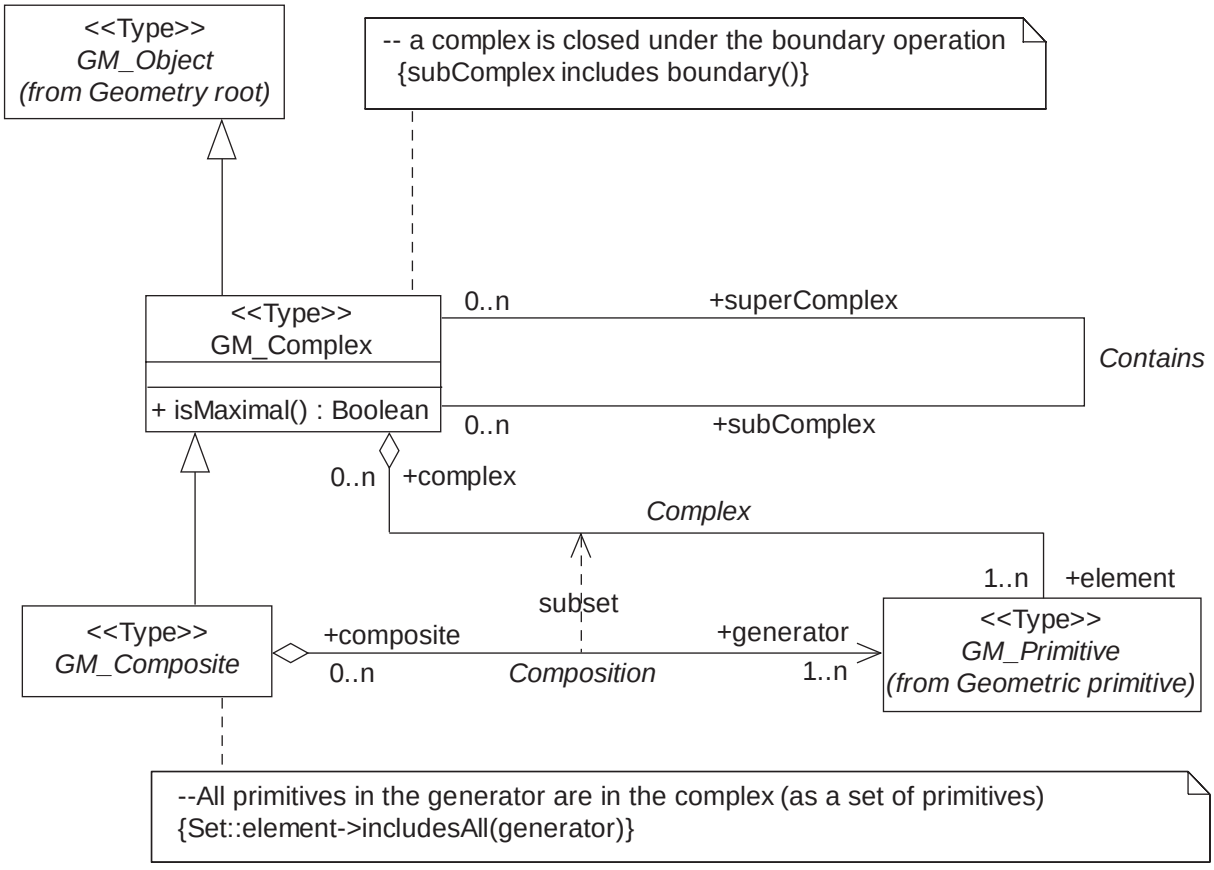


Figure 25 — GM_Complex

6.6.3 GM_Composite

6.6.3.1 Semantics

A geometric composite, GM_Composite (Figure 26), shall be a geometric complex with an underlying core geometry that is isomorphic to a primitive. Thus, a composite curve is a collection of curves whose geometry interface could be satisfied by a single curve (albeit a much more complex one). Composites are intended for use as attribute values in datasets in which the underlying geometry has been decomposed, usually to expose its topological nature.

6.6.3.2 generator

The association role Composition::generator shall be a homogeneous collection of GM_Primitives whose union would be the core geometry of the composite. The complex would include all primitives in the generator and all primitives on the boundary of these primitives, and so forth until GM_Points are included. Thus the association role Composition::generator on GM_Composite is a subset of the association role Complex::element on GM_Complex.

```
GM_Composite::generator[1..n] : GM_Primitive
```

The type of geometry in a generator shall be completely determined by the dimension of the composite object. The component curves and surfaces are oriented to allow assembly into the composite in a properly organized manner.

```
GM_CompositePoint:
      generator.type = GM_Point
GM_CompositeCurve:
      generator.type = GM_OrientableCurve
GM_CompositeSurface:
      generator.type = GM_OrientableSurface
GM_CompositeSolid:
      generator.type = GM_Solid
```

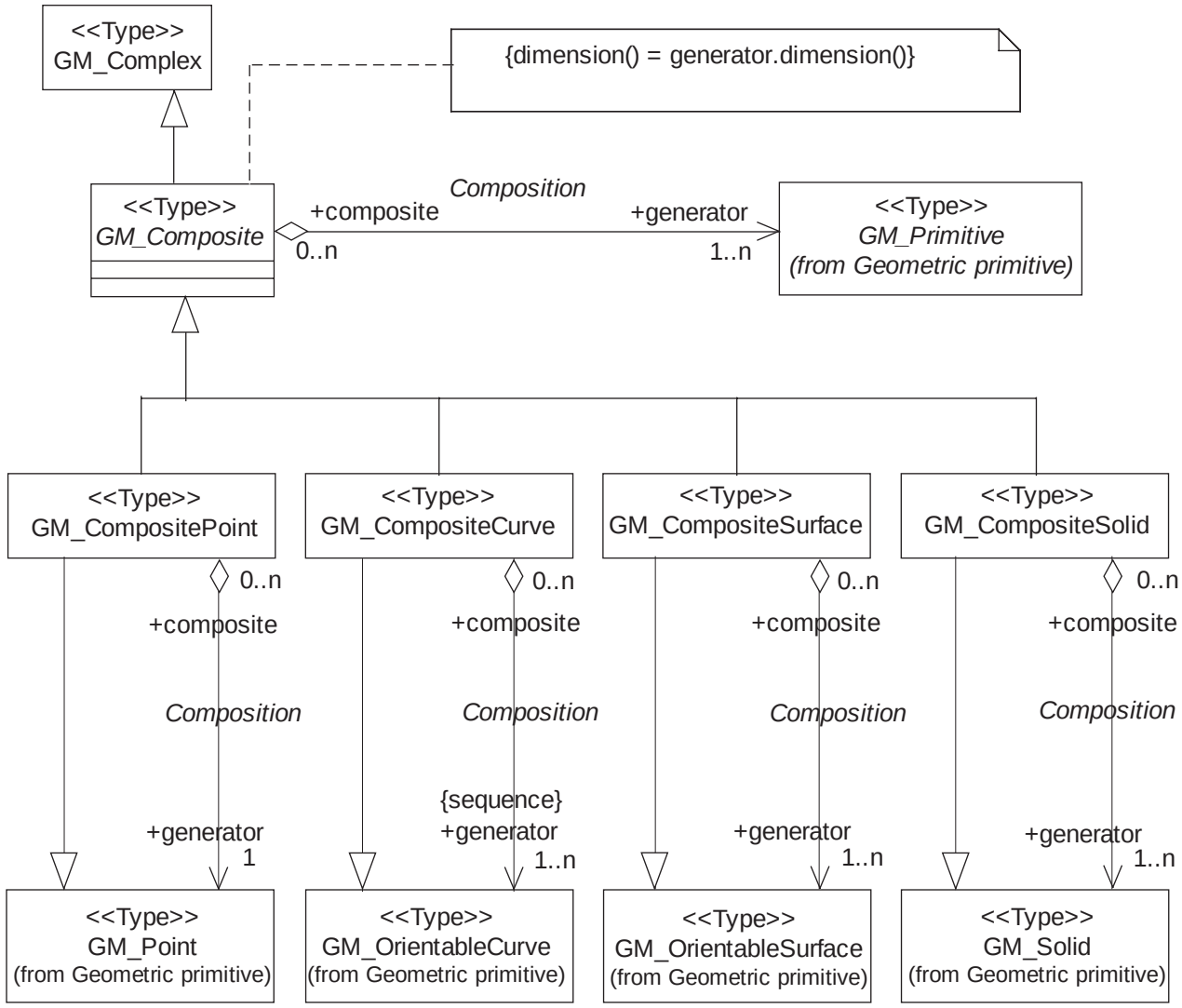


Figure 26 — GM_Composite

6.6.4 GM_CompositePoint

6.6.4.1 Semantics

A separate class for composite point, GM_CompositePoint (Figure 27) is included for completeness. It is a GM_Complex containing one and only one GM_Point.

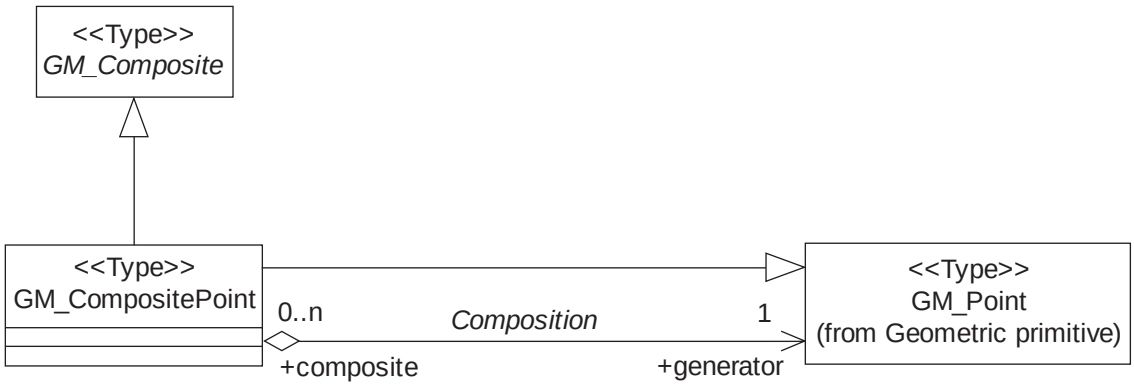


Figure 27 — GM_CompositePoint

6.6.4.2 generator

The association role Composition::generator associates this GM_Composite Point to the single primitive in this complex.

```
GM_CompositePoint::generator [1] : GM_Point
```

6.6.5 GM_CompositeCurve

6.6.5.1 Semantics

A composite curve, GM_CompositeCurve (Figure 28) shall be a GM_Composite with all the geometric properties of a curve. These properties are instantiated in the operation “curve”. Essentially, a composite curve is a list of orientable curves (GM_OrientableCurve) agreeing in orientation in a manner such that each curve (except the first) begins where the previous one ends.

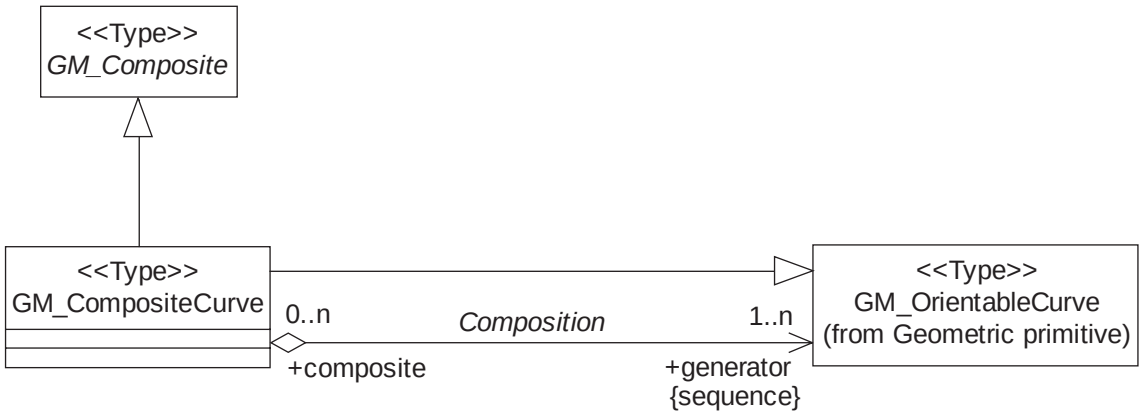


Figure 28 — GM_CompositeCurve

6.6.5.2 generator

The association role Composition::generator associates this GM_CompositeCurve to the primitive GM_Curves and GM_OrientableCurves in its generating set, the curves that form the core of this complex.

```
GM_CompositeCurve::generator : Sequence<GM_OrientableCurve>
-- the start point of each orientable curve in the generator is the
      -- end point of the previous one
      GM_CompositeCurve:
      forAll (1 < j < generator.count - 1)→
            generator[j].endPoint = generator[j+1].startPoint;
```

NOTE To get a full representation of the elements in the GM_Complex, the GM_Points on the boundary of the generator set of GM_Curve would be added to the curves in the generator list.

6.6.6 GM_CompositeSurface

6.6.6.1 Semantics

A composite surface, GM_CompositeSurface (Figure 29) shall be a GM_Complex with all the geometric properties of a surface, and thus can be considered as a type of orientable surface (GM_OrientableSurface). Essentially, a composite surface is a collection of oriented surfaces that join in pairs on common boundary curves and which, when considered as a whole, form a single surface.

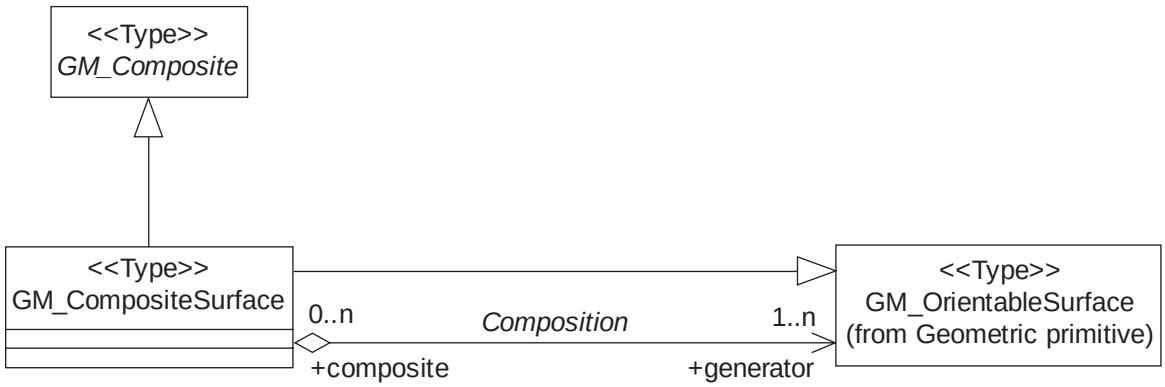


Figure 29 — GM_CompositeSurface

6.6.6.2 generator

The association role Composition::generator associates this GM_CompositeSurface to the primitive GM_Surfaces and GM_OrientableSurfaces in its generating set, a list of the GM_Surfaces that form the core of this complex.

```
GM_CompositeSurface::generator : Set<GM_OrientableSurface>
```

NOTE To get a full representation of the elements in the GM_Complex, the GM_Curves and GM_Points on the boundary of the generator set of GM_Surfaces would be added to the curves in the generator list.

6.6.7 GM_CompositeSolid

6.6.7.1 Semantics

A GM_CompositeSolid (Figure 30) shall be a GM_Complex with all the geometric properties of a solid. Essentially, a composite solid is a set of solids that join in pairs on common boundary surfaces to form a single solid.

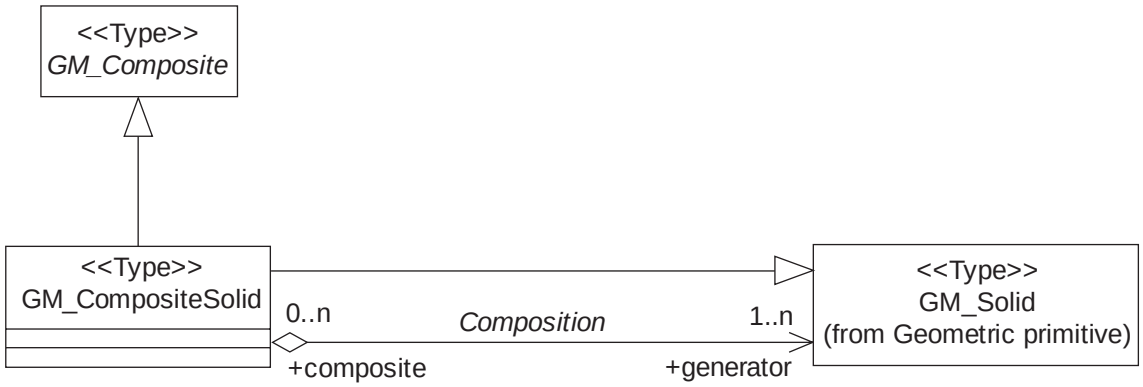


Figure 30 — GM_CompositeSolid

6.6.7.2 generator

The association role Composition::generator associates this GM_CompositeSolid to the primitive GM_Solids in its generating set, that is, the solids that form the core of this complex.

```
GM_CompositeSolid::generator : Set<GM_Solid>
```

NOTE To get a full representation of the elements in the GM_Complex, the GM_Surfaces, GM_Curves and GM_Points on the boundary of the generator set if GM_Solids would have to be added to the generator list.

7 Topology packages

7.1 Semantics

The most productive use of topology is to accelerate computational geometry. The method by which this is accomplished is to associate explicitly feature instances and geometric object instances in a manner consistent with and derived from their implicit geometric relations (see D.3). In some cases, these associations are derived from a conceptual geometry that does not agree with the representation of the feature instances. For this purpose, it is necessary to define topology packages that parallel the geometry packages in Clause 6. Figure 31 shows these packages and their dependencies.

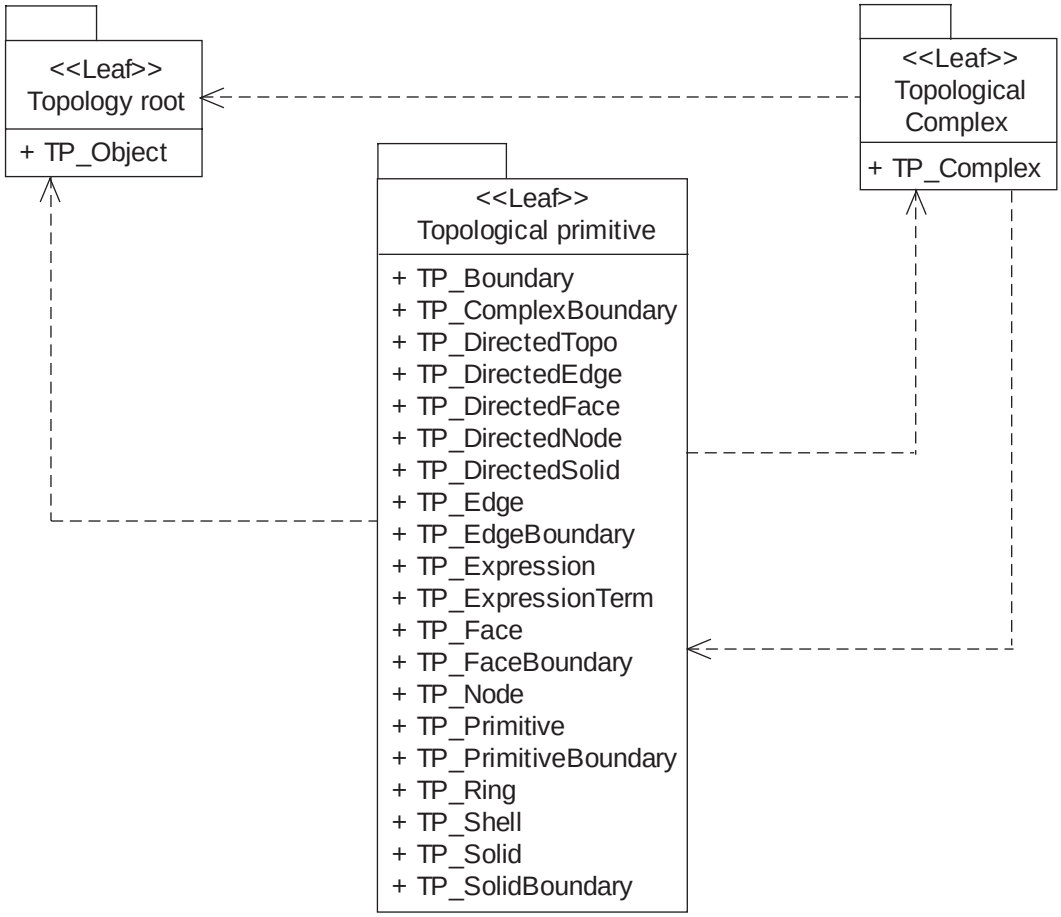


Figure 31 — Topology packages, class content and internal dependencies

Figure 32 gives an overview of the class structure of the basic topological packages. The root class of the diagram is TP_Object. Under this, there are TP_Primitive, and TP_Complex, which are related in way similar to the GM_Primitive and GM_Complex, so that a TP_Complex is an organized structure of TP_Primitives. The major difference being that a GM_Primitive is more loosely coupled to a GM_Complex, allowing it to stand alone, whereas a TP_Primitive must be in at least one TP_Complex. An instance of TP_DirectedTopo shall contain a reference to a TP_Primitive and an orientation parameter, similar to the GM_OrientablePrimitive in 6.3.13. Since only two orientations are possible, regardless of dimension, each primitive is associated to two directed topological entities similar to the relation between GM_OrientableCurve and GM_Curve, and between, GM_OrientableSurface and GM_Surface. To conserve on the number of objects and to make the natural identification of a primitive with its positive orientation, each primitive in each dimension is subclassed under its corresponding directed topological object. This is further explained in 7.3.11.1.

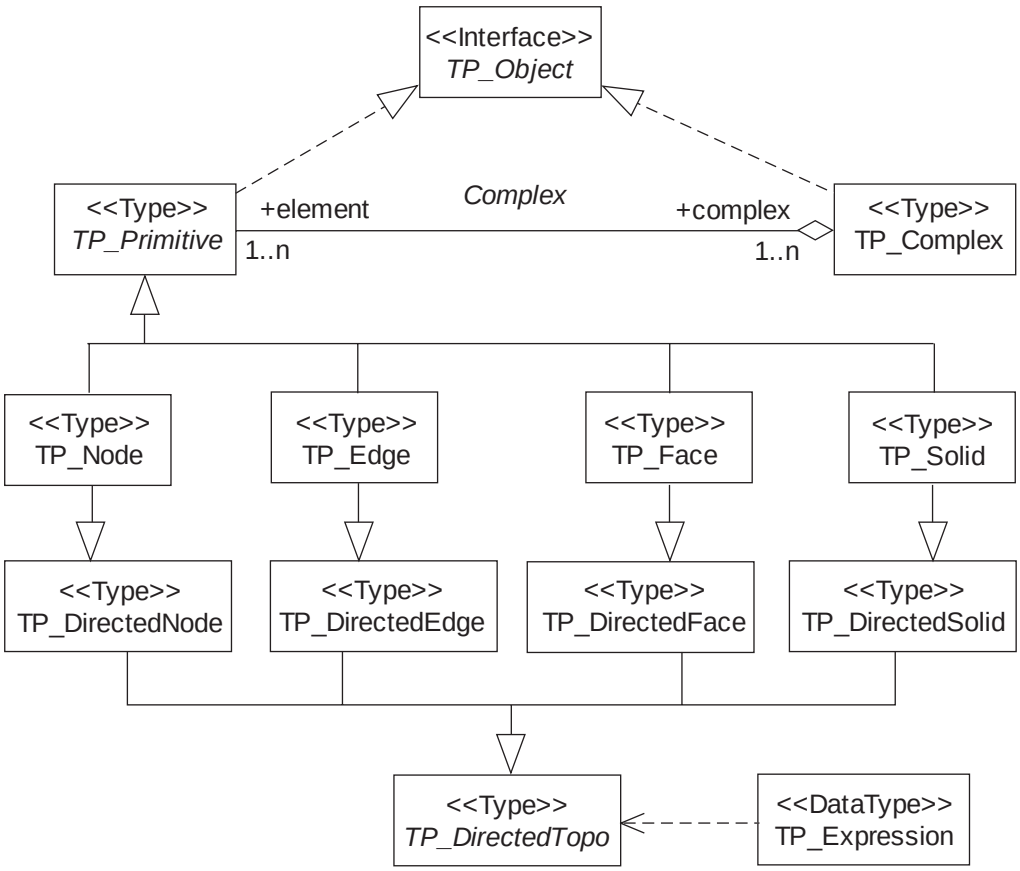


Figure 32 — Topological class diagram

7.2 Topology root package

7.2.1 Semantics

Geometric calculations such as containment (point-in-polygon), adjacency, boundary, and network tracking are computationally intensive. For this reason, combinatorial structures known as topological complexes are constructed to convert computational geometry algorithms into combinatorial algorithms. Another purpose is, within the geographic information domain, to relate feature instances independently of their geometry. For the first purpose, topology definitions in this clause parallel the structure of the geometric definitions in Clause 6. For the second purpose, the classes in these packages are specified so that they can be used independently of the geometry.

A topological complex consists of collections of topological primitives of all kinds up to the dimension of the complex. Thus, a 2-dimensional complex must contain faces, edges, and nodes, while a 1-dimensional complex or graph contains only edges and nodes.

NOTE Topological primitives are equivalent to but are not subclasses of geometric primitives. This is consistent with the view that topological complexes are constructed to optimize computational geometry procedures by the use of combinatorial algorithms. This also permits the creation of structures that ignore geometric constraints by using a topological complex that is not realized by a geometric complex.

The key to understanding the use of computational topology is to see the related procedures in both systems. As Figure 33 shows, there is a great deal of parallelism between the ways in which primitives and complexes are related in the two class systems.

The topological system is based on algebraic manipulations of multivariate polynomials. The definitions of the procedures, functions, and operations in the topology packages are done so that geometric problems in the

geometric domain can be translated into algebraic problems in the topology domain, solved there, and the solutions translated back to the geometric domain. A topological expression in this algebra is a multivariate, degree one polynomial, where the variables correspond to topological primitives.

The diagram in Figure 33 summarizes the relation between topology and geometry. The OCL constraint means that the diagram commutes such that navigation of the roles TP_Primitive::complex followed by TP_Complex.geometry is the same as navigation of the roles TP_Primitive::geometry followed by GM_Primitive::complex.

NOTE A single GM_Primitive may be involved in many independent GM_Complexes, each of which may be a realization of a different TP_Complex. Thus, a GM_Primitive may be the realization of many different TP_Primitives, since a TP_Primitive must occur in one and only one maximal TP_Complex (see 7.3.10.2). Since it is possible for an instantiable class to implement TP_Primitive and TP_Complex, or both GM_Primitive and GM_Composite, it is possible that a particular instance of TP_Primitive may be realized by a GM_Composite, for example, see D.3.

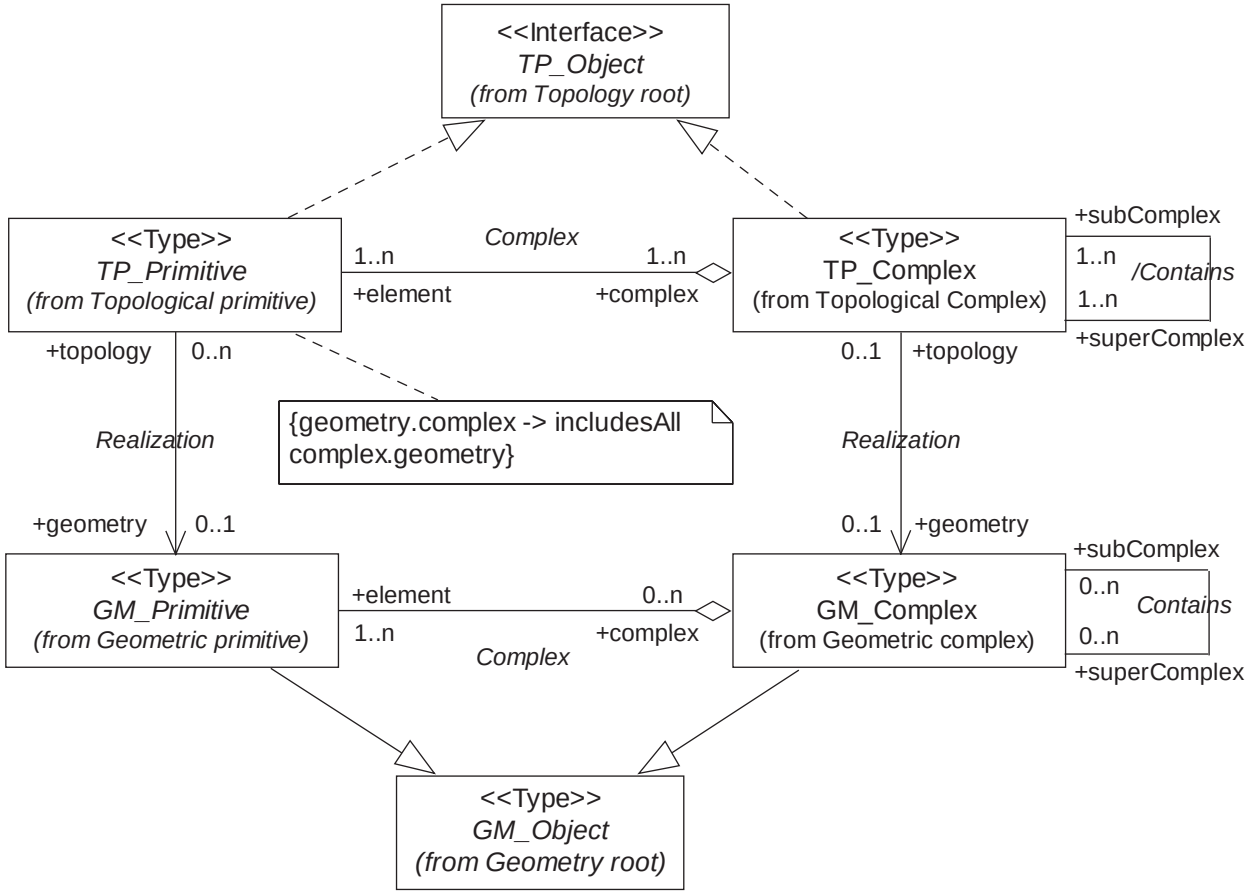


Figure 33 — Relation between geometry and topology

7.2.2 TP_Object

7.2.2.1 Semantics

Topological object, TP_Object (Figure 34) is an abstract class that supplies a root type for topological complexes and topological primitives.

Logically and structurally, topological objects and geometric objects could share the same subclass structure, but since there is a categorical homomorphism from topology to geometry that preserves boundary operations, this approach could cause confusion between the boundary of a topological object and the boundary of the corresponding geometric object. While the two mechanisms share many computational

characteristics, as demonstrated by the homomorphism, they are different operations and need to be clearly separated.

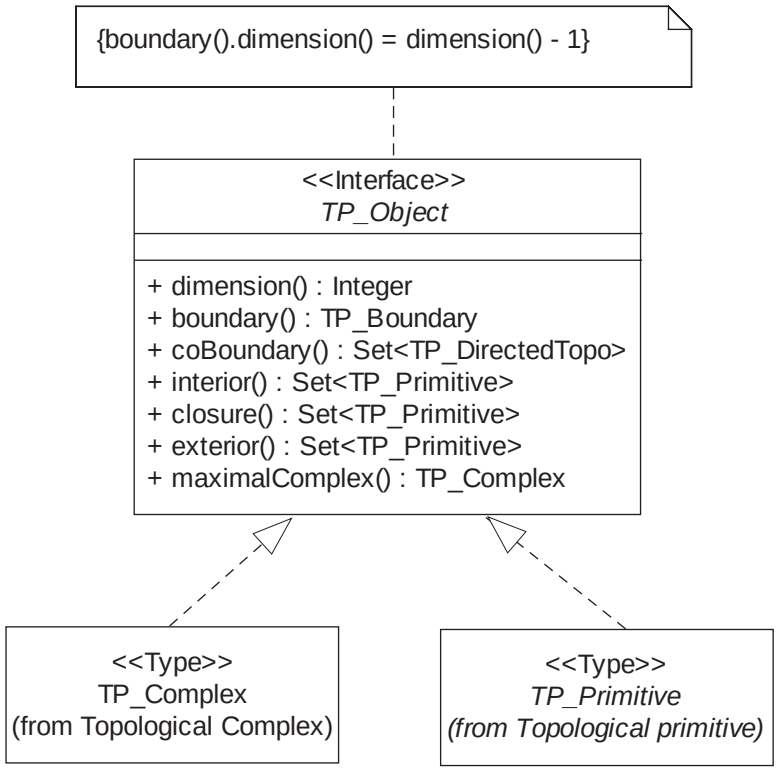


Figure 34 — TP_Object

7.2.2.2 dimension

The integer returned by the operation “dimension” shall be the topological dimension of this TP_Object. It shall be solely dependent on the instantiated class of the object and shall not be changed for a particular object without changing that object's class. For example, the value for dimension is 0 for nodes, one for edges, two for faces, and three for solids. Any GM_Object associated to this TP_Object shall have this same dimension.

```
TP_Object::dimension() : Integer
```

7.2.2.3 boundary

The operation “boundary” shall return a set of TP_DirectedTopo structured as a TP_Boundary that represents the boundary of the TP_Object.

```
TP_Object::boundary() : TP_Boundary
```

If this TP_Object is associated to a GM_Object, its boundary shall be consistent in orientation with that GM_Object as described in the geometry packages.

As a constraint, the dimension of a boundary shall always be one less than the dimension of the original object. For this reason, the dimension of the empty set shall be considered to be “–1”.

```
TP_Object:
      boundary.dimension() = dimension() - 1
```

Figure 35 shows how the boundary function can be visualized as an association from objects of each dimension to objects of one less dimension.

In most cases the return value will be a valid value of a TP_Expression (7.3.20). The boundary returned can fail to be a valid TP_Expression because of the requirement for simplest terms. A dangling or isolated edge in a face (one that has the same face on both sides) would cancel out in the conversion to a topological expression.

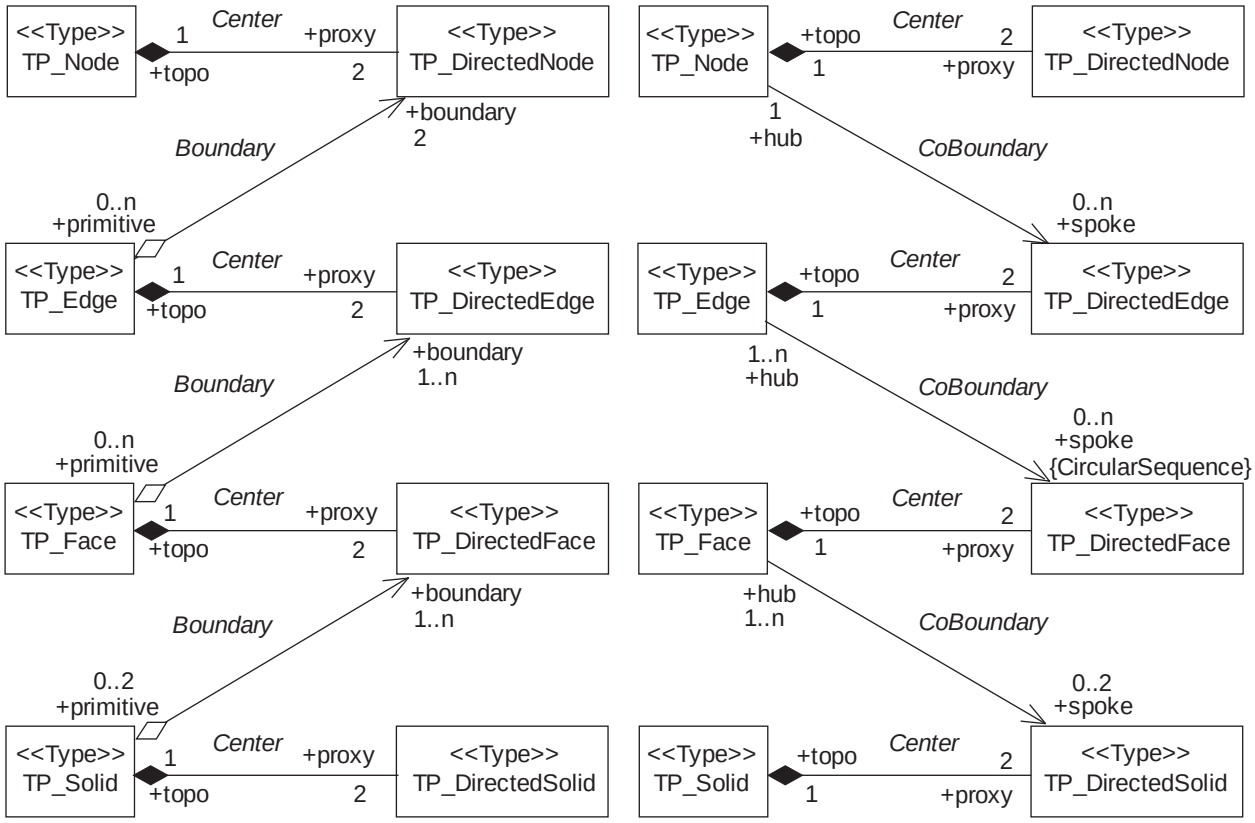


Figure 35 — Boundary and coboundary operation represented as associations

7.2.2.4 coBoundary

The operation “coBoundary” shall return a Set of TP_DirectedTopo that represents all the TP_Objects that have this TP_Object on their boundary.

In most cases the return value will be a valid value of a TP_Expression (7.3.20). An exception to this is when the corresponding GM_Object is on the boundary of a closed object (such as a curve that begins and ends at the same point). The TP_Object corresponding to that GM_Object would appear in the Set of TP_DirectedTopo twice with opposite orientations and therefore cancel out when the coBoundary is cast from Set of TP_DirectedTopo to TP_Expression.

```
TP_Object::coBoundary() : Set<TP_DirectedTopo>
```

Figure 36 illustrates how this operation can be visualized as a relation between dimension levels of the TP_Primitives, similar to the boundary operation, but directed in the opposite direction, increasing dimension instead of reducing it.

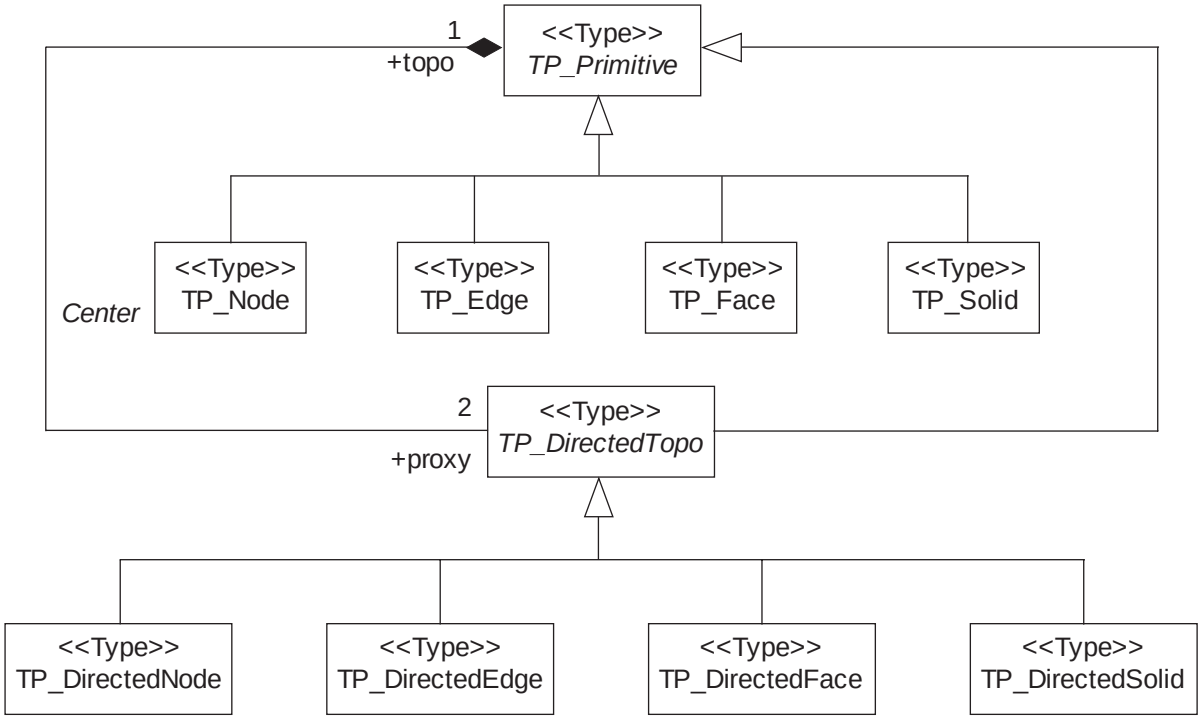


Figure 36 — Important classes in topology

7.2.2.5 interior

The operation “interior” shall return the finite set of TP_Primitives that comprises the interior of this object within the maximal complex of this object. For a TP_Primitive this will be a self-reference. For a TP_Complex this will be all TP_Primitive elements in the TP_Complex not on the boundary of the TP_Complex. This is the homomorphic equivalent of the interior of a geometric realization of this TP_Object.

```
TP_Object::interior() : Set<TP_Primitive>
```

7.2.2.6 exterior

The operation “exterior” shall return the finite set of TP_Primitives that comprises the exterior of this object within the maximal complex of this object. This consists of all TP_Primitives in the maximal TP_Complex that are not in the interior or the boundary of this TP_Object.

```
TP_Object::exterior() : Set<TP_Primitive>
```

7.2.2.7 closure

The operation “closure” is often useful; it is defined as a union of the interior and boundary of an object, and is thus not required in a basic implementation.

```
TP_Object::closure() = interior().union(boundary())
```

7.2.2.8 maximalComplex

The operation “maximalComplex()” shall return the maximal TP_Complex that contains this TP_Object.

```
TP_Object::maximalComplex() : TP_Complex
```

A TP_Object shall be included in one and only one maximal TP_Complex.

NOTE A complex is maximal if it is contained in no larger complex. The cardinality restriction implied by this operation means that any TP_Object is in one and only one maximal complex.

7.3 Topological primitive package

7.3.1 Semantics

The Topological primitive package contains all the primitives for each dimension and supports classes for representations of their structural relationships.

7.3.2 TP_Boundary

This class is a root class for all boundary data types used in the topological package. It requires no further detail except that it is a TP_Expression and a cycle.

```
TP_Boundary:
          IsCycle();
```

7.3.3 TP_ComplexBoundary

This class is a root class for all boundary data types used in the topological package for topological complexes. It requires no further detail except that it is a TP_Expression.

7.3.4 TP_PrimitiveBoundary

Each topological primitive is capable of returning its boundary. Data types under TP_PrimitiveBoundary (Figure 37) are used to structure those boundaries in a convenient manner. Since TP_Node has an empty boundary, no special data type is defined for its boundary.

It is a simple fact that the boundary of any geometric object is a cycle (has no boundary). A surface's boundary components are a set of circular composite curves, each closing on itself. For consistency between topology and geometry, this is also a requirement for all subclasses of TP_Boundary. If a TP_Complex represents the topology of a GM_Complex, then the geometric realities will enforce this constraint.

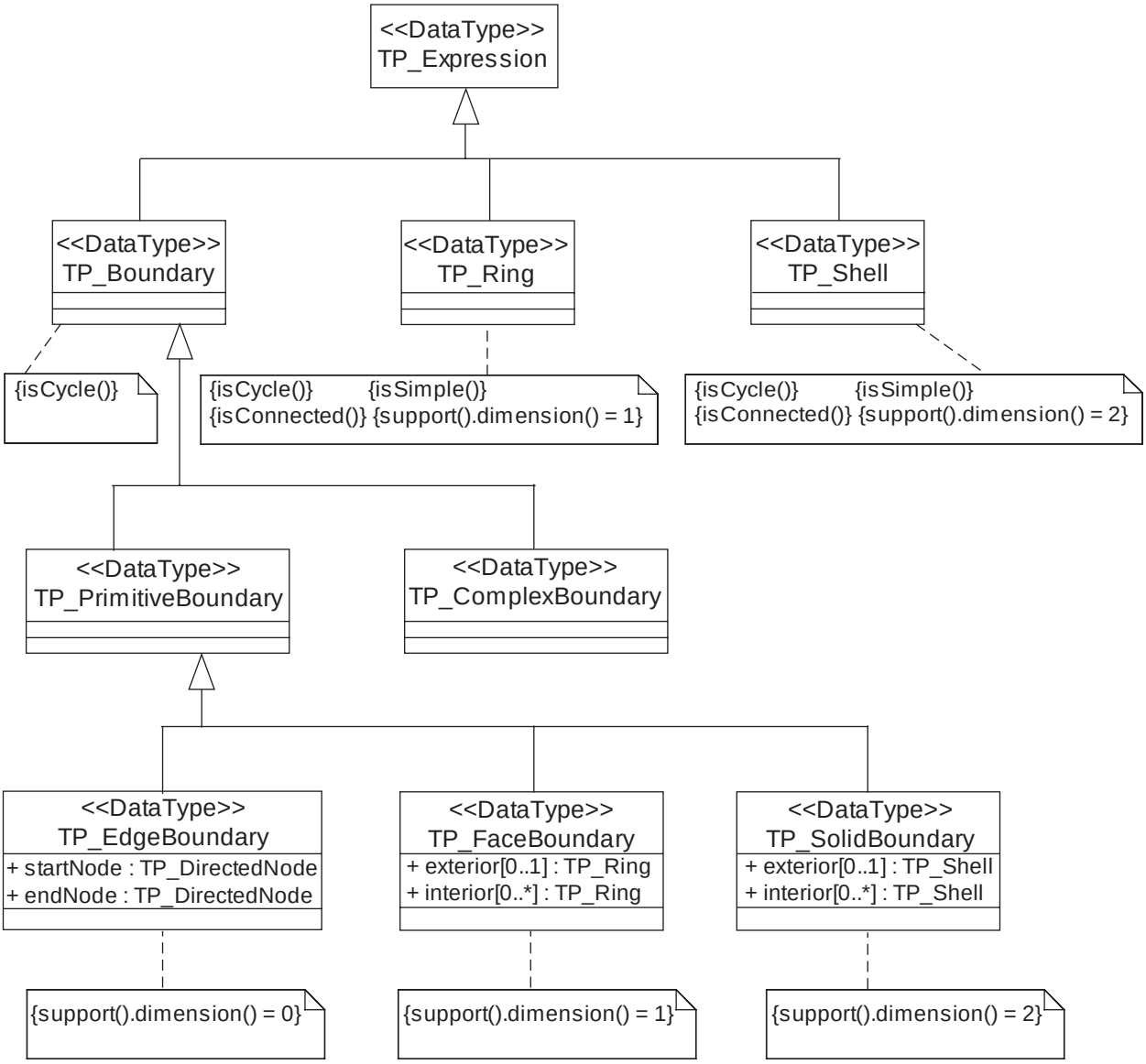


Figure 37 — Boundary relation data types

7.3.5 TP_EdgeBoundary

A TP_EdgeBoundary (Figure 37) contains two TP_Node references as TP_DirectedNode instances. The startNode shall have a positive orientation, and the endNode, a negative Orientation. As a TP_Expression, a TP_EdgeBoundary shall look like:

```
Edge.boundary() = +endNode-startNode
```

The attributes of a TP_EdgeBoundary are:

```
TP_EdgeBoundary::startNode : TP_DirectedNode;
TP_EdgeBoundary::endNode : TP_DirectedNode;
```

7.3.6 TP_FaceBoundary

A TP_FaceBoundary consists of some number of TP_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.

```
TP_FaceBoundary::exterior[0,1] : TP_Ring;
TP_FaceBoundary::interior[0..n] : TP_Ring;
```

Recalling that each ring is oriented so that the face is on its left, we get the boundary of a face as an expression as:

```
Boundary(face)= b : TP_FaceBoundary = b.exterior + b.interior
```

7.3.7 TP_SolidBoundary

TP_SolidBoundaries are similar to TP_FaceBoundaries. In normal Euclidean space, one shell is distinguished as the exterior. In the more general case, this is not always possible.

```
TP_SolidBoundary::exterior[0,1] : TP_Shell;
TP_SolidBoundary::interior[0..n] : TP_Shell;
```

Recalling that each shell is oriented so that the solid is on bottom, we get the boundary of a solid as an expression as:

```
Boundary(solid)= b : TP_SolidBoundary = b.exterior + b.interior
```

7.3.8 TP_Ring

A TP_Ring is used to represent a single connected component of a TP_FaceBoundary. It consists of a number of TP_DirectedEdges connected in a cycle (an object whose boundary is empty). A TP_Ring is structurally similar to a GM_CompositeCurve in that the endNode of each TP_DirectedEdge in the sequence is the startNode of the next TP_DirectedEdge in the Sequence. Since the sequence is circular, there is no exception to this rule.

As a TP_Expression, the interpretation of a TP_Ring is a sequence of oriented edges. Each edge "e" which is used in its positive orientation shows up in the expressions as a "+e", and each edge "d" which shows up in its negative orientation shows up in the expression as a "–d". Since TP_Rings are used in TP_FaceBoundary objects, the ring will be oriented so that the face is on its "left" in any geometric realization.

7.3.9 TP_Shell

A TP_Shell is used to represent a single connected component of a TP_SolidBoundary. It consists of a number of TP_Faces connected in a topological cycle (an object whose boundary is empty). Unlike a TP_Ring, a TP_Shell has no natural sort order.

As a TP_Expression, the interpretation of a TP_Shell is a set of oriented faces. Each face "f" which is used in its positive orientation shows up in the expressions as a "+f", and each edge "g" which shows up in its negative orientation shows up in the expression as a "–g". Since TP_Shells are used in TP_SolidBoundary objects, the shell will be oriented so that the upNormal points away from the solid.

7.3.10 TP_Primitive

7.3.10.1 Semantics

Topological primitives, TP_Primitive (Figure 38), are the non-decomposed elements of a topological complex. As such, they normally correspond to the geometric primitives of a like dimension that are the components of a geometric complex. When a geometric complex is the realization of a topological complex, then the primitives in each shall be in a dimension-preserving, 1-to-1 correspondence.

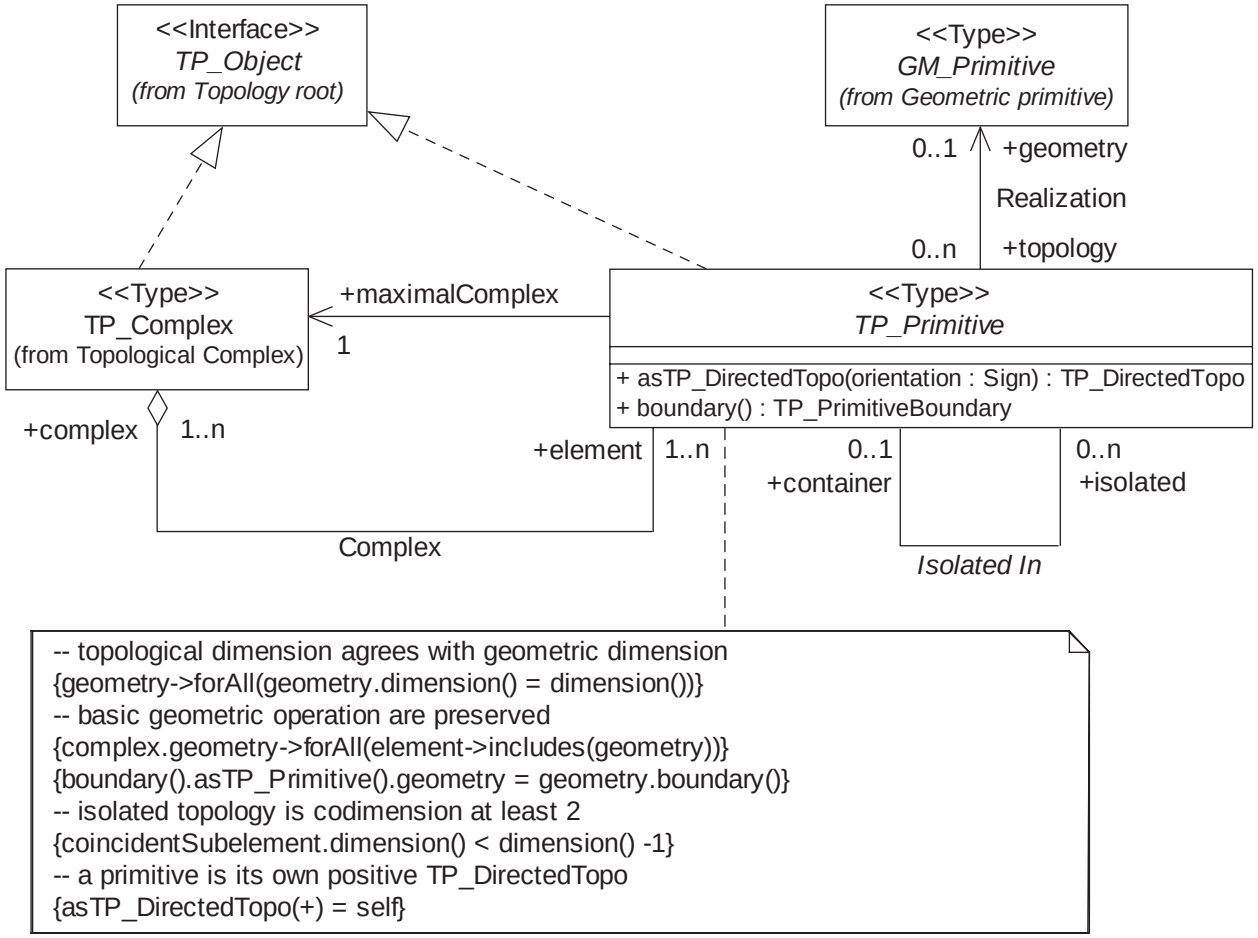


Figure 38 — TP_Primitive

7.3.10.2 Realization

The association “Realization” links this TP_Primitive to the GM_Primitive that it represents in its maximal complex. If this TP_Primitive is used to describe a logical topological structure that is not realized by a GM_Complex, then this relationship shall be empty for all TP_Primitives contained in this TP_Primitive's maximal TP_Complex. Each GM_Primitive may be associated to at most one TP_Primitive in any TP_Complex. If this TP_Primitive is in any realized TP_Complex, then it shall be associated to exactly one GM_Primitive. A GM_Primitive may be associated to different TP_Primitives in different TP_Complexes.

```
TP_Primitive::geometry [0,1] : GM_Primitive
GM_Primitive::topology [0..n] : TP_Primitive
```

NOTE Since GM_Composites are subtyped under the corresponding primitives, it is possible to define a schema where the realization of a TP_Primitive is a GM_Composite of the same dimension. Thus a TP_Edge can be realized as a GM_CompositeCurve, see D.3.

To preserve the homomorphism between topological objects and their boundary operators, and the corresponding geometric objects and their boundary operators, the mapping defined by this association shall be dimension-preserving and the associations between element and complex in the two domains shall be preserved.

```
TP_Primitive:
      dimension() = geometry.dimension();
```

7.3.10.3 Complex association

The association "Complex" shall link this TP_Primitive to the finite set of TP_Complexes that contain it. Every TP_Primitive shall be in some number of TP_Complexes that are all subcomplexes of a unique maximal TP_Complex containing this TP_Primitive.

```
TP_Primitive::complex [1..n] : TP_Complex
TP_Complex::element [1..n] : TP_Primitive
```

7.3.10.4 Isolated In association

All of the adjacency relations in topology between primitives whose dimensions differ by one or 0 are handled by the boundary and coboundary operations. These operations only deal with instances of one primitive lying on the boundary of another primitive of one higher dimension, or with instances of the same dimension that share a common boundary element. This includes instances where a "dangling" edge has the same face on both sides, or a "dangling" face has the same solid on both sides. The exception to this is when one primitive is completely surrounded by a primitive of at least two higher dimensions, with no intermediate primitive. These are truly isolated. In faces, this includes nodes that are not attached to an intermediate edge on the boundary of that face. In a 3D space, the isolated node could be connected to another edge that is not on the boundary of the surface in question, such as in the case where the edge is realized by a curve perpendicular to the surface that the face realizes. In solids, this can include nodes or edges that are not attached to surfaces in the boundary of the solid.

```
TP_Primitive::isolated [0..n] : TP_Primitive
TP_Primitive::container [0,1] : TP_Primitive

TP_Primitive:
      isolated.dimension() < self.dimension() - 1;
      container.count = 0 implies
      TP_Primitive→exists(boundary().topo→includes(self))
```

7.3.10.5 boundary

The boundary operation for TP_Primitive shall overrides that defined at TP_Object by adding more structure to the set of TP_DirectedTopo.

```
TP_Primitive::boundary() : TP_PrimitiveBoundary
```

Since TP_Primitive is abstract, the additional structure will be defined for each of its subtypes.

7.3.11 TP_DirectedTopo

7.3.11.1 Semantics

From a computational point of view, elements of TP_DirectedTopo (Figure 39, Figure 40) are equivalent to the various orientable geometric objects (GM_OrientableObject) in the geometry packages (GM_OrientableCurve and GM_OrientableSurface). TP_DirectedNode and TP_DirectedSolid do not have separate geometric object equivalents.

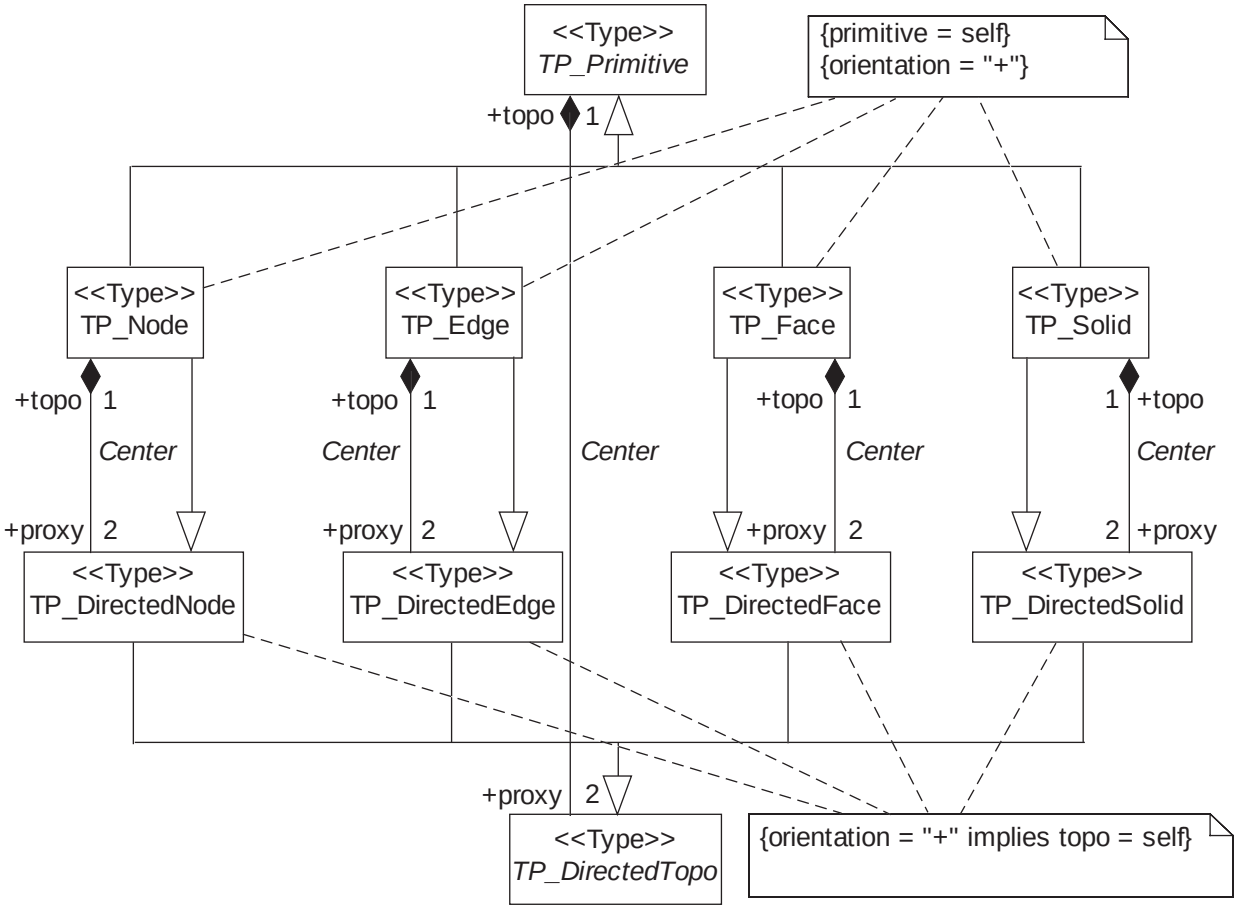


Figure 39 — TP_DirectedTopo subclasses

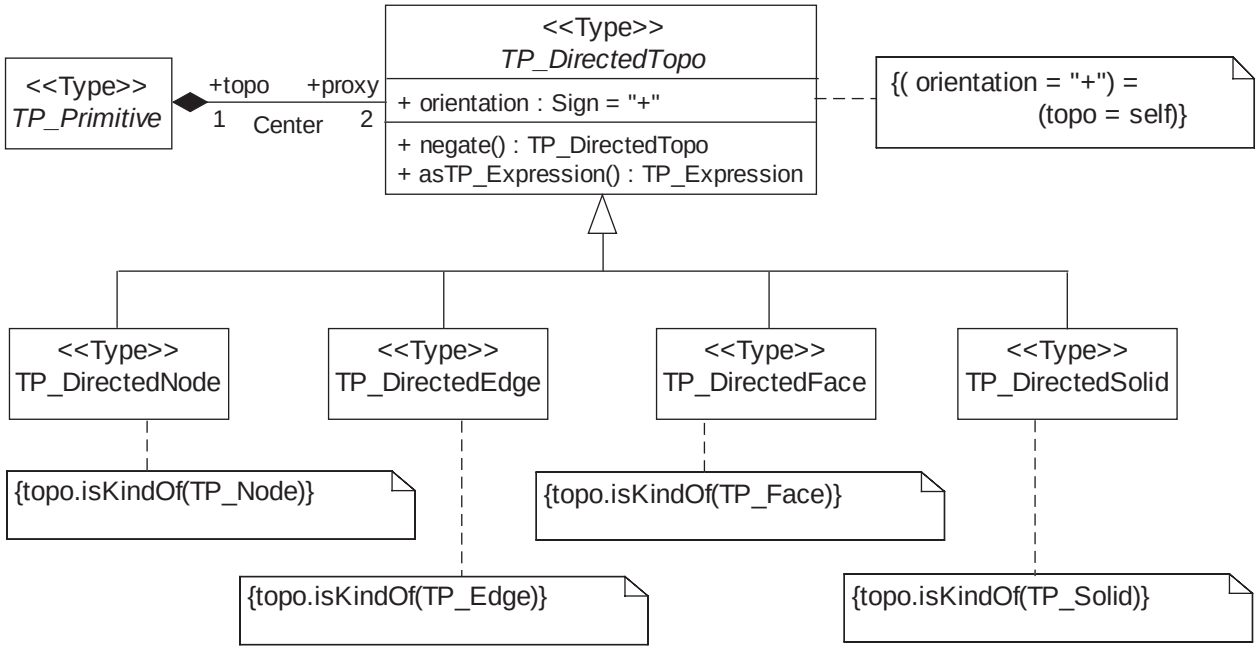


Figure 40 — TP_DirectedTopo

As in the geometry, each topological primitive inherits from its corresponding directed topological primitive, but it satisfies more constraints. This means that TP_Node is equivalent to a positive TP_DirectedNode, a TP_Edge to a positive TP_DirectedEdge, etc.

NOTE An alternative type hierarchy would have separated TP_Primitive and TP_DirectedTopo, which would have entailed three objects for each primitive: the primitive itself, its equivalent positive directed topological primitive, and its reversal (a negative directed) topological primitive. This alternative is a valid implementation of the abstract types in this model, but it does not emphasize the logical equivalence of a topological primitive and its positive directed topological primitive. From an algebraic point of view, the subclassing and OCL constraints that identify a primitive with its positive directed primitive make it equivalent to the standard interpretation of the unary "+" (plus) in algebra as in "x = + x". Since the most powerful use of topological objects is in their symbolic manipulation, maintaining an algebraic metaphor is appropriate.

There is an implicit relation between the directed topological objects of adjacent dimensions. The boundary and coboundary operations and relations use them to carry the same orientation sense. Thus if a positive directed edge is on the boundary of a face, then the positive directed face is on the coboundary of the associated edge. If a positive directed node is on the boundary of an edge, then the corresponding positive directed edge is on the coboundary of the associated node.

7.3.11.2 Orientation

The attribute "orientation" shall be the sense in which this directed topological object is related to its underlying TP_Primitive.

```
TP_DirectedTopo::orientation : Sign = "+"
```

7.3.11.3 Negate

The operation "negate" shall return the opposite orientation of this primitive.

```
TP_DirectedTopo::negate() : TP_DirectedTopo
```

7.3.11.4 asTP_Expression

The operation "asTP_Expression" shall create a TP_Expression from this TP_DirectedTopo, and shall retain the sign and the sense of the orientation. This operator shall be the constructor from the class TP_Expression.

```
TP_DirectedTopo::asTP_Expression() : TP_Expression
```

7.3.11.5 Center Association

The role "topo" in the association "Center" shall identify the associated TP_Primitive. The inverse role "proxy" shall identify the two TP_DirectedTopo instances associated to the particular TP_Primitive.

```
TP_DirectedTopo::topo [1] : TP_Primitive
TP_Primitive::proxy [2] : TP_DirectedTopo
```

7.3.11.6 Constraints

Following the logic of the semantics of directed topological objects, the associated topology for each directed topological object shall be of the appropriate type.

```
TP_DirectedNode:
         topo.isKindOf(TP_Node);
TP_DirectedEdge:
         topo.isKindOf(TP_Edge);
```

```
TP_DirectedFace:
         topo.isKindOf(TP_Face);
TP_DirectedSolid:
         topo.isKindOf(TP_Solid);
```

NOTE These constraints use the OCL operator “isKindOf” to indicate that the class of a directed topological primitive corresponding to a topological primitive must be a realization of the corresponding topological primitive type.

The Center association forms an important part of the algebra of the boundary and coBoundary operations.

```
TP_DirectedTopo:
      [boundary() = (orientation)*topo.boundary()]
TP_Primitive:
      [boundary() = (proxy.orientation)*proxy.boundary()]
TP_DirectedTopo:
         negate.topo = topo;
         negate.orientation <> orientation;
```

7.3.12 TP_Node

7.3.12.1 Semantics

TP_Node (Figure 41) inherits all of its interfaces from TP_Primitive, with some elaboration on the structure of boundary and coboundary.

For TP_Node, the operation “coBoundary” defined at TP_Object shall always return a set of references to TP_DirectedEdges indicating which edges enter (positive TP_DirectedEdges) and which leave (negative TP_DirectedEdges) the node. This operation is overridden from TP_Object. The same information may be represented as an association.

NOTE In 2-dimensional maximal TP_Complex containing this TP_Node, the coBoundary may be sorted as a clockwise circular sequence in any geometric realization of this maximal TP_Complex. In a 3D complex, the ordering is arbitrary.

```
TP_Node::coBoundary : Set<TP_DirectedEdge> {size = [0..n]}
TP_Node::coBoundary.spoke : Set<TP_DirectedEdge> {size = [0..n]}
```

7.3.12.2 Center association

Each TP_Primitive, including TP_Node, is associated to two TP_DirectedTopo instances.

```
TP_Node::proxy [2] : TP_DirectedNode
TP_DirectedNode [1] : Reference<TP_Node>
```

7.3.12.3 boundary

The boundary operation for TP_Node shall overrides that defined at TP_Object by specifying the Empty set.

```
TP_Primitive::boundary() : NULL
```

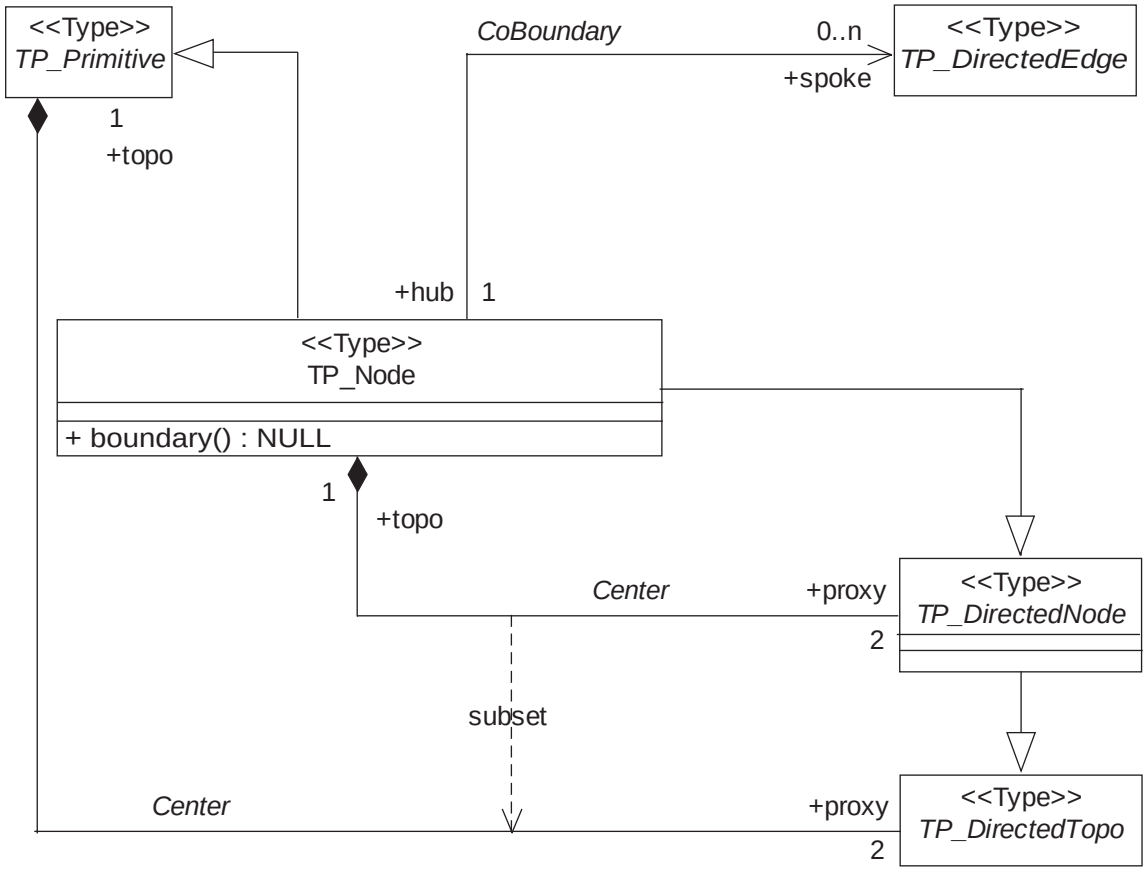


Figure 41 — TP_Node

7.3.12.4 Constraints

The TP_Node's dimension shall be 0, and its boundary is empty (NULL).

```
TP_Node:
            TP_Object::dimension = 0;
            TP_Object::boundary() = NULL;
```

NOTE A node may still be isolated in a face and be the end of an edge, as long as that edge is not on the boundary of the containing face. The geometric realization of this would be a curve that dangles in space, but terminates at its intersection with a surface.

7.3.13 TP_DirectedNode

The class “TP_DirectedNode” supports TP_Node in the computational topology class TP_Expression. For TP_Node, the operation “boundary” defined at TP_Object shall always return a zero-valued expression, corresponding to empty geometry. This operation is overridden from TP_Object.

```
TP_Node::boundary() : NULL
```

7.3.14 TP_Edge

7.3.14.1 Semantics

The primitive TP_Edge (Figure 42) is the 1-dimensional primitive for topology. For TP_Edge, the operation "boundary" defined at TP_Object shall return a pair of nodes, one at the start of the edge (negative TP_DirectedNode) and one at the end (positive TP_DirectedNode). This operation is overridden from TP_Object. The same information may be represented as an association.

```
TP_Edge::boundary() : Set<TP_DirectedNode> {size = 2}
TP_Edge::boundary.boundary : Set<TP_DirectedNode> {size = 2}
```

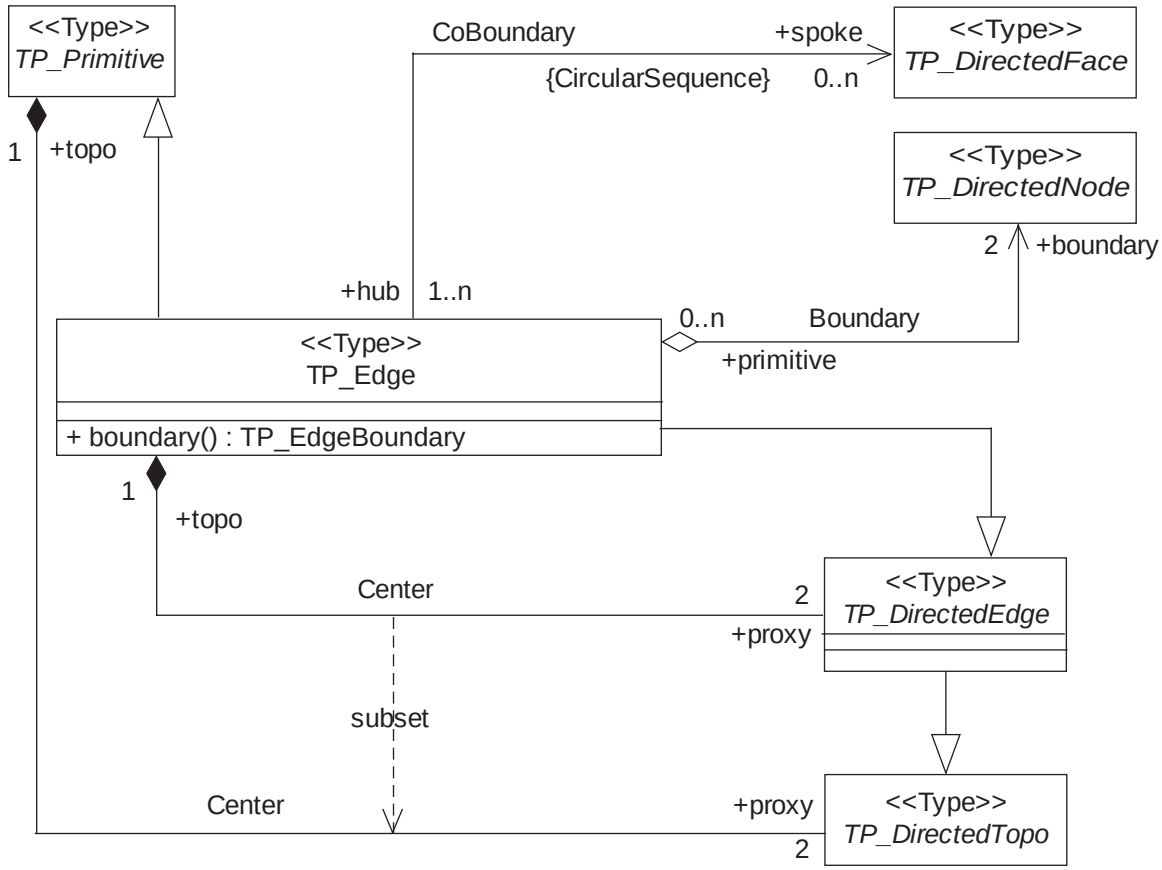


Figure 42 — TP_Edge

7.3.14.2 coBoundary

For TP_Edge, the operation "coBoundary" defined at TP_Object shall return a circular sequence of directed faces indicating which faces use this edge (positive TP_DirectedFace) or its negative proxy (negative TP_DirectedFace) on their boundary. The circular sequence shall represent a clockwise enumeration of these faces as viewed from the end point of the associated curve in any geometric realization of the maximal TP_Complex in which this TP_Edge is contained. This operation is overridden from TP_Object. The same information may be implemented as an association.

```
TP_Edge::coBoundary() : CircularSequence<TP_DirectedFace> {size = [0..n]}
TP_Edge::coBoundary.spoke : CircularSequence<TP_DirectedFace> {size = [0..n]}
```

NOTE In the 2-dimensional planar case, the coboundary has at most two faces. In the full topology case, there are precisely 2, one directed face having a positive "+" orientation and the associated face lying to the left of the edge, and the other directed face having a negative "-" orientation, and the associated face lying to the right of the edge.

7.3.14.3 boundary

The boundary operation for TP_Edge shall overrides that defined at TP_Object by specifying a TP_EdgeBoundary, consisting of a start node and end node.

```
TP_Edge::boundary() : TP_EdgeBoundary
```

The TP_Edge shall also has an association Boundary with association role boundary which specifies this same information as two directed edges, oriented positively for the end node and negatively for the start node.

```
TP_Edge::boundary [2] : TP_DirectedNode
```

7.3.14.4 Center association

Each TP_Primitive, including TP_Edge is associated to two TP_DirectedTopo instances.

```
TP_Edge::proxy [2] : TP_DirectedEdge
      TP_DirectedEdge::topo [1] : Reference<TP_Edge>
```

NOTE In the 2-dimensional planar case, each directed edge bounds at most one face, precisely one face in a full planar topology. In the 3-dimensional case, or in a non-planar 2D complex, a directed edge can bound several faces.

7.3.14.5 Constraints

The TP_Edge shall have dimension 1.

```
TP_Edge:
         TP_Object::dimension() = 1
```

7.3.15 TP_DirectedEdge

The class “TP_DirectedEdge” supports TP_Edge in the computational topology class TP_Expression. It is analogous to the concept of a GM_OrientableCurve, in the sense that it acts as a proxy for the base curve/edge when needed.

7.3.16 TP_Face

7.3.16.1 Semantics

The class “TP_Face” (Figure 43) provides topological primitives for GM_Surface.

7.3.16.2 boundary

For TP_Face, the operation “boundary” defined at TP_Object shall return a set of directed edges with appropriate orientation. This operation is overridden from TP_Object. The same information may be represented as an association.

```
TP_Face::boundary() : TP_FaceBoundary
```

NOTE The same restriction on the meaning of exterior applies to the topology as did to the geometry.

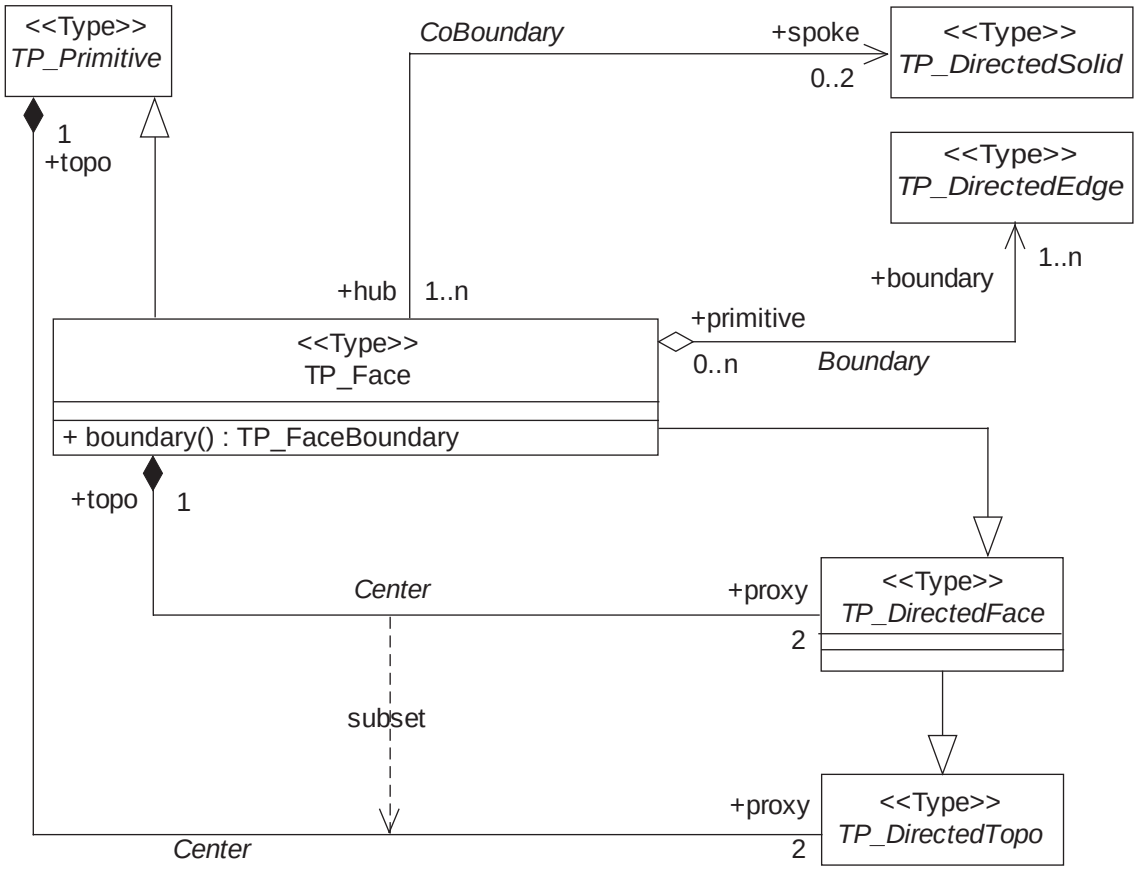


Figure 43 — TP_Face

The TP_Face shall also has an association Boundary with association role boundary that specifies this same information as directed edges, oriented positively for the left side of the edge and negatively for the right.

```
TP_Face::boundary [1..*] : TP_DirectedEdge
```

The additional information that is returned by the boundary operator is the organization of the TP_FaceBoundary into rings and an indication as to which ring is the exterior.

7.3.16.3 coBoundary

For TP_Face, the operation “coBoundary” defined at TP_Object shall return a set of references to directed solids indicating which solids use this face (positive TP_DirectedSolid) or its negative proxy (negative TP_DirectedSolid) on their boundary. This operation is overridden from TP_Object. The same information may be implemented as an association.

```
TP_Face::coBoundary() [0..2] : Reference<TP_DirectedSolid>
TP_Face::coBoundary.spoke [0..2] : Reference<TP_DirectedSolid>
```

7.3.16.4 Center association

Each TP_Primitive, including TP_Face is associated to two TP_DirectedTopo instances.

```
TP_Face::proxy [2] : TP_DirectedFace
TP_DirectedFace::topo [1] : Reference<TP_Face>
```

7.3.16.5 Constraints

TP_Face's dimension shall be 2.

```
TP_Face:
        TP_Face: TP_Object::dimension = 2
```

7.3.17 TP_DirectedFace

TP_DirectedFaces shall be used in defining the boundary of a TP_Solid. It is analogous to the concept of a GM_OrientableSurface, in the sense that it acts as a proxy for the base surface/face when needed.

7.3.18 TP_Solid

7.3.18.1 Semantics

The class “TP_Solid” (Figure 44) provides topological primitives for GM_Solid.

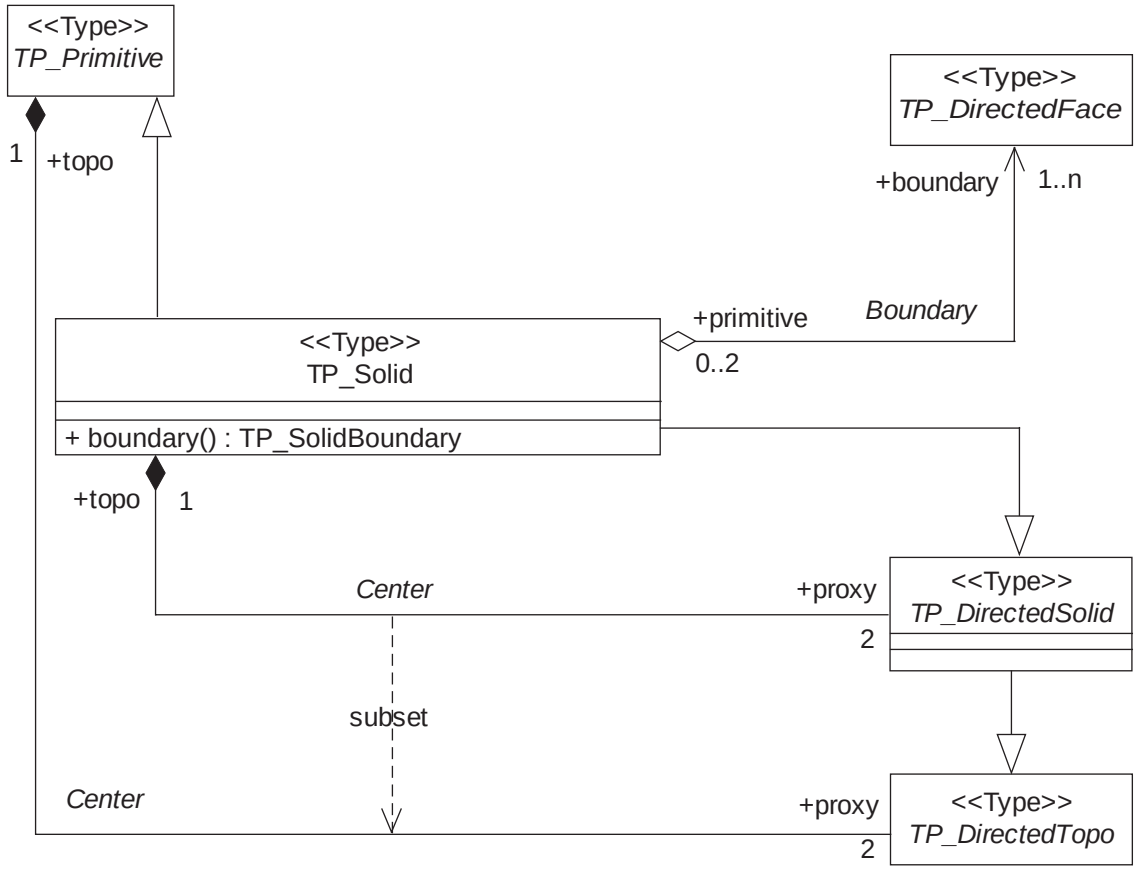


Figure 44 — TP_Solid

7.3.18.2 boundary

For TP_Solid, the operation “boundary” defined at TP_Object shall return a collection of faces or their negative proxies. This operation is overridden from TP_Object. The same information may be represented as an association.

```
TP_Solid::boundary() : TP_SolidBoundary
```

The TP_Solid shall also has an association Boundary with association role boundary that specifies this same information as directed edges, oriented positively for below the face and negatively for above the face.

```
TP_Solid::boundary [1..*] : TP_DirectedFace
```

The additional information that is returned by the boundary operator is the organization of the TP_SolidBoundary into shells and an indication as to which shell is the exterior.

7.3.18.3 coBoundary

For TP_Solid, the operation “coBoundary” shall return NULL.

```
TP_Solid::coBoundary() : NULL
```

7.3.18.4 Center association

Each TP_Primitive, including TP_Solid is associated to two TP_DirectedTopo instances.

```
TP_Solid::proxy [2] : TP_DirectedSolid
TP_DirectedSolid::topo [1] : Reference<TP_Solid>
```

7.3.18.5 Constraints

A TP_Solid's dimension shall be 3.

```
TP_Solid:
         TP_Object::dimension = 3
```

7.3.19 TP_DirectedSolid

The class “TP_DirectedSolid” supports TP_Solid in the computational topology class TP_Expression.

7.3.20 TP_Expression

7.3.20.1 Semantics

Algebraic or computational topology is most easily conceptualized as the manipulation of multivariate, degree-one polynomials where the variables correspond to TP_Primitives. The TP_DirectedTopo class represents the terms in this algebra. The TP_Expression class (Figure 45) represents the polynomial expressions.

The order of the terms in a polynomial does not affect its value, so the TP_Expression class has been subclassed from Set<TP_DirectedTopo>. The operations of the TP_Expression class are those needed to construct, manipulate, and test these “polynomials”.

The key to computational topology is the ability to treat pieces of topology in an algebraic or combinatorial manner. The primitives in this algebra are the TP_Primitives. The monomials (single variable, single term polynomials) are the instances of TP_Primitives, each with an integer coefficient, instantiated as TP_ExpressionTerm.

Any constraint that would be consistent for multivariate, first-order polynomial algebra shall be valid for TP_Expression, such as:

```
TP_DirectedTopo:
         negate().asTP_Expression() = as TP_Expression().negate()
         asTP_Expression.negate().Plus(asTP_Expression).isZero()
```

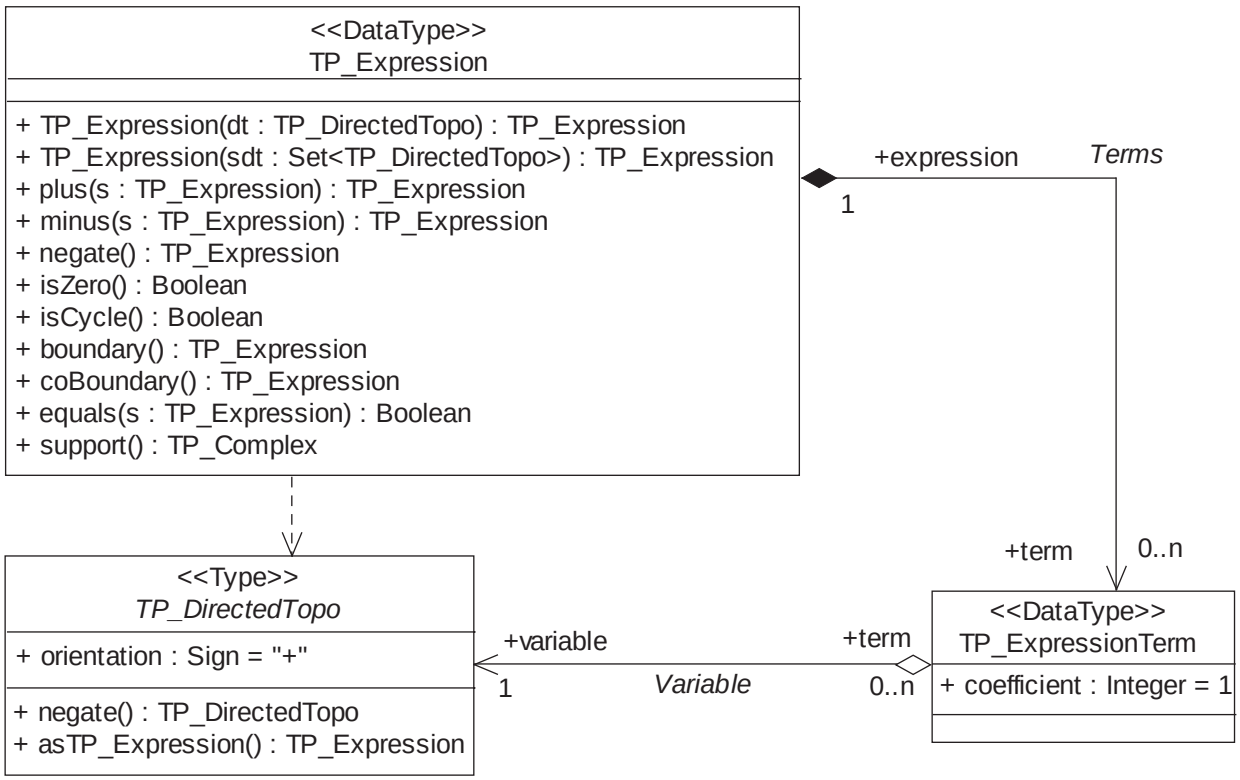


Figure 45 — TP_Expression

7.3.20.2 TP_ExpressionTerm

TP_Expressions, like polynomials, consist of a set of terms, which consist of a variable and a coefficient.

```
TP_ExpressionTerm = <coefficient : Integer = 1, variable :
      Reference<TP_DirectedTopo>>
```

Arithmetic shall be consistent with normal polynomial manipulation.

7.3.20.3 TP_Expression : constructor

The constructor “TP_Expression” shall create a TP_Expression from a TP_DirectedTopo. This operation shall be used by other classes (such as TP_Object) for the creation of expressions.

```
TP_Expression( dt : TP_DirectedTopo) : TP_Expression = { <1, dt> }
      TP_Expression(sdt : Set<TP_DirectedTopo>) : TP_Expression = { <1, dt> |
      sdt.contains(dt)}
```

7.3.20.4 plus

The operation “plus” acts as polynomial addition for TP_Expressions. It shall combine TP_DirectedTopo elements that have the same underlying instances of TP_Primitive by adding their “orientation” coefficients. It shall remove any terms with zero coefficient.

```
TP_Expression::plus(s : TP_Expression) : TP_Expression
```

7.3.20.5 minus

The operation “minus” acts as polynomial subtraction for TP_Expressions. It shall combine TP_DirectedTopo elements that have the same underlying instances of TP_Primitive by subtracting their “orientation” coefficients. It shall remove any terms with zero coefficients.

```
TP_Expression::minus(s : TP_Expression) : TP_Expression
```

7.3.20.6 negate

The operation “negate” shall negate each of the terms in the TP_Expression. It is the unary minus operator for the polynomials.

```
TP_Expression::negate() : TP_Expression
```

7.3.20.7 isZero

The operation “isZero” shall return TRUE for the zero polynomial. It is equivalent to the “Set.IsEmpty” operation.

```
TP_Expression::isZero() : Boolean
```

7.3.20.8 isCycle

The operation “isCycle” shall return TRUE for a polynomial whose boundary (defined by TP_Expression::boundary()) is zero. A TP_Expression is a cycle if it represents a closed geometric object, such as the boundary of a polygon. In most GIS cases, a TRUE value returned by “isCycle” implies that the underlying geometric object is the boundary of some other geometric object. It is equivalent to “isZero(boundary())”.

```
TP_Expression::isCycle() : Boolean
```

NOTE Any image of a boundary operation is a cycle. That means boundary().boundary().isZero() = TRUE.

7.3.20.9 boundary

The operation “boundary” shall replace each TP_Primitive in each TP_DirectedTopo in this TP_Expression with its boundary and shall simplify the resultant expression. Boundaries always consist of TP_Primitives of one lower dimension. If the dimension of all the TP_Primitives in this TP_Expression is zero (the TP_Primitives are all nodes), then the boundary operation shall return a zero TP_Expression.

```
TP_Expression::boundary() : TP_Expression
```

7.3.20.10 coBoundary

The operation “coBoundary” shall replace each TP_Primitive in each TP_DirectedTopo in this TP_Expression with its coBoundary and shall simplify the resultant expression. Coboundaries always consist of TP_Primitives of one higher dimension. If the dimension of all the TP_Primitives in this TP_Expression is the same as the dimension of the corresponding maximal TP_Complex, then the coBoundary operation shall return a zero TP_Expression.

```
TP_Expression::coBoundary() : TP_Expression
```

7.3.20.11 equals

The operation “equals” shall return TRUE for a polynomial equality. The order of the elements (terms) is not significant.

```
TP_Expression::equals(s : TP_Expression) : Boolean
```

7.3.20.12 support

The operation “support” shall cast this TP_Expression as a set of TP_Primitives for use in calculating geometric operators. The operation is essentially the “asSet” operation followed by a traversal of the Center association between TP_DirectedTopo and TP_Primitive.

```
TP_Expression::support() : Set<TP_Primitive>
```

7.3.20.13 asSet

The operation “asSet” shall cast this TP_Expression as a set of TP_DirectedTopo for use in calculating geometric operators. This cast shall include adding all boundary elements to the set until TP_DirectedNodes are reached. In other words, the support of a TP_Expression shall be a valid TP_Complex.

```
TP_Expression::asSet() : Set<TP_DirectedTopo>
```

7.4 Topological complex package

7.4.1 Semantics

The package “Topological complex” provides additional classes for the creation of TP_Complexes.

7.4.2 TP_Complex

7.4.2.1 Semantics

This clause contains the definition of topological complexes that parallel the geometric complexes introduced earlier in 6.6. A TP_Complex (Figure 46) may use set operations on its elements to perform the equivalent set operations on the underlying sets of direct positions that are represented by the geometric elements of a geometric realization (a GM_Complex).

7.4.2.2 TP_Complex: constructor of a topological complex

The default construction of a topological complex shall be to generate it from a geometric complex. After the construction, the geometric complex shall be the geometric realization of the topological complex. Only geometric complexes that consist of mutually disjoint geometric primitives will generate a topological complex without error.

```
TP_Complex::TP_Complex(GC : GM_Complex) : TP_Complex
```

The use of the default constructor to define a default topological complex for each geometric complex assures that the topology represented by the TP_Complex is the topology of a geometric configuration as represented by the GM_Complex. The association “Realization” shall trace each part of the TP_Complex back to the appropriate part of the GM_Complex. This allows us to speak of topological operations within a topological complex (TP_Complex) as if they occurred directly on a geometric complex (GM_Complex).

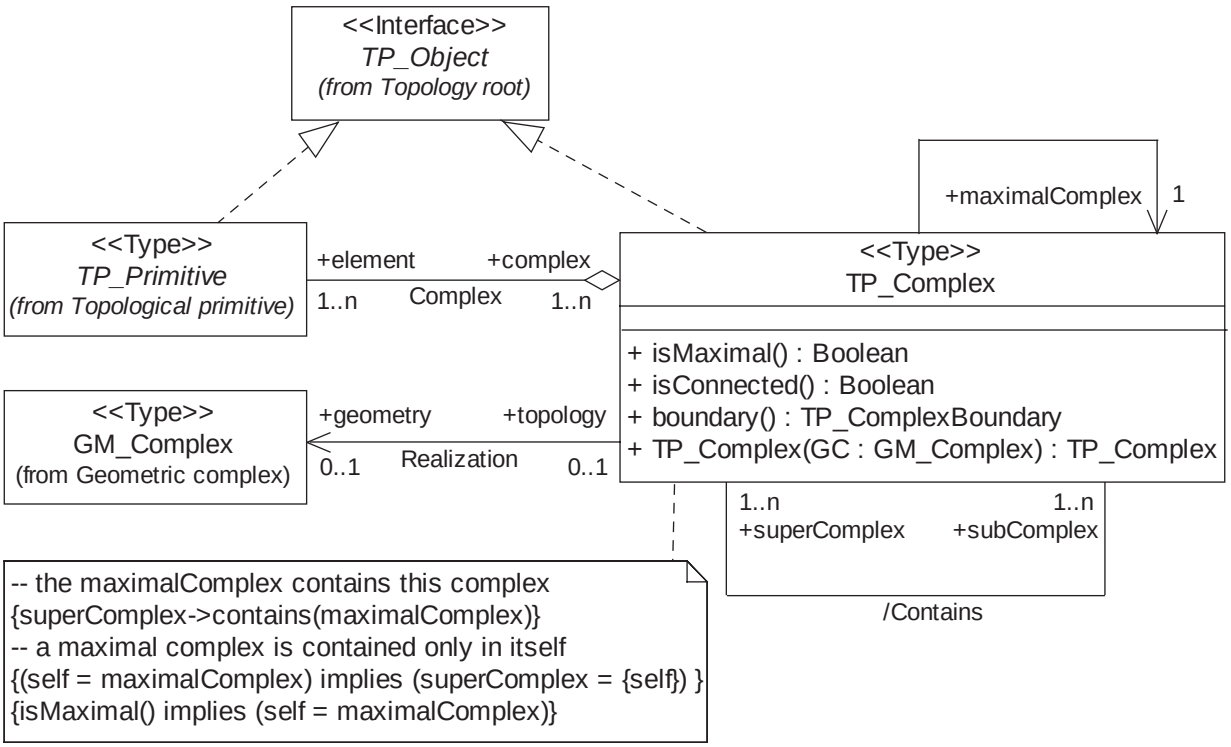


Figure 46 — TP_Complex

7.4.2.3 maximalComplex

The private attribute “maximalComplex” contains a reference to the unique maximal topological complex of which this TP_Complex is a member. This is needed for encoding to determine the limits of an export data set.

```
- TP_Complex::maximalComplex : Reference<TP_Complex>
```

7.4.2.4 isMaximal

The Boolean operation “isMaximal” shall return TRUE if this TP_Complex is contained in no larger TP_Complex.

```
TP_Complex::isMaximal() : Boolean
```

7.4.2.5 isConnected

The Boolean valued operation “isConnected” shall return TRUE if this TP_Complex is topologically connected.

```
TP_Complex::isConnected() : Boolean
```

NOTE If a TP_Complex is connected, then its geometric realization is also connected. This does not imply that it is a composite (geometric or topological), since composites must comply with the stronger constraint of being isomorphic to a primitive. To test whether or not a topological complex is connected without referring to a geometric realization requires that the transitive closure of the boundary, coBoundary, and IsolatedIn associations be calculated. If every primitive in the complex is linked to every other primitive in the complex by a sequence of these association roles where each intermediate primitive is in the complex, then the complex is connected.

7.4.2.6 Contains association

The derived association “Contains” shall describe which other TP_Complexes are contained in this TP_Complex as sets of TP_Primitives. The “superComplex” role is the larger of the two complexes and the “subComplex” role is the smaller. This relation shall be consistent with the “contains” operation inherited from Set<TP_Primitive>.

```
TP_Complex::subComplex [1..n] : Reference<TP_Complex>
TP_Complex::superComplex [1..n] : Reference<TP_Complex>
```

7.4.2.7 Complex association

The “Complex” association shall relate the TP_Primitive elements to this TP_Complex. It is this association that makes the TP_Complex a Set<TP_Primitives>. The set operations implied by “Contains” should be consistent with this definition of the TP_Complex as a set of primitives.

```
TP_Complex::element [1..n] : Reference<TP_Primitive>
TP_Primitive::complex [[1..n] : Reference<TP_Complex>
```

7.4.2.8 Realization association

The realization association links this TP_Complex to its corresponding GM_Complex (if any).

```
TP_Complex::geometry [0,1] : GM_Complex
GM_Complex::topology [0,1] : TP_Complex
```

8 Derived topological relations

8.1 Introduction

This clause specifies a mechanism for characterizing topological relations as operators to be used in query. These query operators can be calculated using the set theoretic operations defined on GM_Object and its subtypes and on algebraic operations defined on TP_Expression. These two mechanisms are equivalent for geometric complexes that are realizations of the corresponding topological complexes. The operators defined in this clause are meant mainly for query evaluation and are defined in such a manner as to allow a variety of implementations to be assured of equivalent results against datasets with equivalent information content.

This International Standard does not assign specific names to particular spatial operators. It is assumed that application schemas will use any or all of the following three classification techniques to specify application specific operators. In the cases below, the classification scheme is based on TP_Objects. This also defines the same operators on GM_Objects given that the restrictions defined above for the creation of TP_Complexes from collections of GM_Objects are followed. What is to follow is only valid for point, curve, surface, and solid objects. The theory for aggregate objects that are not homogeneous in dimension is not yet satisfactory enough to base a standard on.

The conformance of a query system to this part of this International Standard shall mean that the supported topological query operations can be defined according to the characterizations laid out on one of the subsequent clauses and that all operators defined in the clause can be made available directly or through a well understood combination of supported operators. Minimal compliance to this clause implies that:

1) Boolean query operations are defined in terms consistent with the included subclauses.

2) All valid Boolean operators definable within the context of one or more of the 8.2, 8.3, or 8.4 are available for use.

Complete compliance requires support of all of the valid Boolean operators definable within the context of this entire clause.

8.2 Boolean or set operators

8.2.1 Form of the Boolean operators

Set theoretic operators are sometimes referred to as Boolean operators. Since such operators do not distinguish between the interior and boundary of a set, the closure operation is used to combine them:

```
GM_Object::closure() :== interior().union(boundary())
```

For two objects, A and B the following four intersection operations may be done:

intersection [closure(A), closure (B)]	intersection [closure(A), exterior (B)]
intersection [exterior (A), closure (B)]	intersection [exterior (A), exterior (B)]

This matrix of sets may be tested to see if each set is empty or not. This classifies the relationship between A and B into one of 2^4, or 16, classes.

An operator may be defined as a template that is applied to the intersection matrix to test for a particular spatial relationship between the two objects. The template is a matrix of four extended Boolean Values whose interpretation is given in Table 8. There are 3^4 or 81 possible operator templates.

Table 8 — Meaning of Boolean intersection pattern matrix

Symbol	Non Empty?	Meaning
T	TRUE	The intersection at this position of the matrix is non-empty.
F	FALSE	The intersection at this position of the matrix is empty.
N	NULL	This operator does not test the intersection at this position of the matrix.
NOTE The value TRUE means the set is non-empty (see column header).		

To test if two objects are related in agreement with a particular operator template, the intersections not associated to NULL are calculated and tested for non-empty according to the pattern in the matrix. If there is agreement, the value of the operator for these two objects is TRUE, and if not, the value is FALSE.

8.2.2 Boolean Relate

The operator “bRelate” shall return TRUE if these objects are spatially related by testing for intersections between the closure and exterior of the two geometric objects as controlled by the values in the intersectionPatternMatrix.

```
Boolean bRelate(GM_Object, GM_Object, intersectionPatternMatrix)
      Boolean bRelate(TP_Object, TP_Object, intersectionPatternMatrix)
```

The “intersectionPatternMatrix” is listed as a string of 4 characters from T, F, or N, given in row major form, i.e., the two values for the first row, followed by the two for the second row of the matrix.

8.2.3 Relation to set operations

The Boolean relate can be used to implement the “contains”, “intersects” and “equals” operations of GM_Object defined in 6.2.2.18.

EXAMPLE

```
C : GM_Composite, G : GM_Object;
      C.contains(G) = bRelate(C, G, "TNFT" );
```

8.3 Egenhofer operators

8.3.1 Form of the Egenhofer operators

For two objects, A and B the following 9 intersection operations may be done (see references [8] and [9]).

intersection [boundary (A), boundary (B)]	intersection [boundary (A), interior (B)]	intersection [boundary (A), exterior (B)]
intersection [interior (A), boundary (B)]	intersection [interior (A), interior (B)]	intersection [interior (A), exterior (B)]
intersection [exterior(A), boundary(B)]	intersection [exterior (A), interior (B)]	intersection [exterior (A), exterior (B)]

This matrix of sets (called the 9 matrix) may be tested to see if each is empty or not. This classifies the relationship between A and B into one of 2^9, or 512, classes. Actually, not all 512 are geometrically possible, but that is not of consequence to what is to follow.

An operator may be defined as a template that is applied to the intersection matrix to test for a particular spatial relationship between the two objects. The template is a matrix of nine extended Boolean Values whose interpretation is given in Table 9, the content of which is identical to the previous table. There are 3^9 or 19 683 possible operator templates.

Table 9 — Meaning of Egenhofer intersection pattern matrix

Symbol	Non Empty?	Meaning
T	TRUE	The intersection at this position of the matrix is non-empty.
F	FALSE	The intersection at this position of the matrix is empty.
N	NULL	This operator does not test the intersection at this position of the matrix.

To test if two objects are related in agreement a particular operator, the intersections not associated to NULL are calculated and tested for non-empty according to the pattern in the matrix. If there is agreement, the value of the operator for these two objects is TRUE, and if not, the value is FALSE.

8.3.2 Egenhofer relate

The operator “eRelate” shall return TRUE if these objects are spatially related by testing for intersections between the interior, boundary and exterior of the two geometric objects as controlled by the values in the intersectionPatternMatrix.

```
Boolean eRelate(GM_Object, GM_Object, intersectionPatternMatrix)
      Boolean eRelate(TP_Object, TP_Object, intersectionPatternMatrix)
```

The “intersectionPatternMatrix” is listed as a string of nine characters (each being a T, F, or N) in row major form.

8.3.3 Relation to set operations

The Egenhofer relate can be used to implement the “contains”, “intersects” and “equals” operations of GM_Object defined in 6.2.2.18.

EXAMPLE

```
C : GM_Primitive, G : GM_Primitive;
      C.contains(G) = eRelate(C, G, "NFNNTNNFT" );
C : GM_Primitive, G : GM_Composite;
      C.contains(G) = eRelate(C, G, "FFNTTNFFT" );
```

8.4 Full topological operators

8.4.1 Form of the full topological operators

The full topological operators take dimension differences into account (see references [4] and [5] for further analysis of this extension) and are done in a manner similar to the Egenhofer operators, but a finer distinction is made on the possible values.

Table 10 — Meaning of full topological intersection pattern matrix

Symbol	Non Empty?	Meaning
0	TRUE	The intersection at this position of the matrix contains only points.
1	TRUE	The intersection at this position of the matrix contains only points, and curves.
2	TRUE	The intersection at this position of the matrix contains only points, curves, and surfaces.
3	TRUE	The intersection at this position of the matrix contains only points, curves, surfaces and solids.
F	FALSE	The intersection at this position of the matrix is empty.
N	NULL	This operator does not test the intersection at this position of the matrix.

To test if two objects are related in agreement with one of the possible 6^9 = 10 077 696 operator templates, the intersections not associated to NULL are calculated and tested for non-empty and dimension, according to the pattern in the matrix. If there is agreement, the value of the operator for these two objects is TRUE, and if not, the value is FALSE.

8.4.2 Full topological relate

The operator “cRelate” shall return TRUE if these objects are spatially related by testing for intersections between the interior, boundary and exterior of the two geometric objects as controlled by the values in the intersectionPatternMatrix.

```
Boolean cRelate(GM_Object, GM_Object, intersectionPatternMatrix)
      Boolean cRelate(TP_Object, TP_Object, intersectionPatternMatrix)
```

The “intersectionPatternMatrix” is listed as nine characters (each being a 0, 1, 2, 3, F, or N) in row major form.

8.5 Combinations

Operators may be defined as any Boolean combination of one or more of the primitive operations in the preceding sections.

Annex A
(normative)

Abstract test suite

A.1 Geometric primitives

A.1.1 Data types for geometric primitives

A.1.1.1 Data types for 0-dimensional geometry

a) Test Purpose: Verify that an application schema or profile instantiates GM_Point with the attribute *position* and the association *Coordinate Reference System* inherited from GM_Object. If the application schema or profile also instantiates GM_MultiPoint, verify that it includes the attribute *position* and the association to *element*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, 6.1, 6.2.1, 6.2.2.17, 6.3.10.1, 6.3.11.1, 6.3.11.2, 6.4.1, 6.5.1, 6.5.2.1, 6.5.2.2, 6.5.3 and 6.5.4.

d) Test Type: Capability.

A.1.1.2 Data types for 1-dimensional geometry

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.1 and instantiates GM_Curve with the attribute *orientation* and the association *segmentation*, and at least one instantiable subtype of GM_CurveSegment with all of its attributes. If an application schema or profile also instantiates GM_MultiCurve, verify that it includes the attributes *element* and *length*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.1, 6.3.5, 6.3.13, 6.3.14.1, 6.3.16, 6.4.1, 6.4.6, 6.4.8 – 6.4.31, and 6.5.5.

d) Test Type: Capability.

A.1.1.3 Data types for 2-dimensional geometry

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.2 and instantiates GM_Surface with the attribute *orientation* and the associations *interiorTo* and *segmentation*, and at least one subtype of GM_SurfacePatch with all of its attributes. If the application schema or profile also instantiates GM_MultiSurface, verify that it includes the attributes *element*, *area*, and *perimeter*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.2, 6.3.6, 6.3.7, 6.3.10.4, 6.3.15, 6.3.17.1, 6.3.17.3, 6.4.6, 6.4.32 – 6.4.48, and 6.5.6.

d) Test Type: Capability.

A.1.1.4 Data types for geometric 3-dimensional geometry

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.3 and instantiates GM_Solid with the association *interiorTo*. If the application schema or profile also instantiates GM_MultiSolid, verify that it includes the attributes *element*, *area* and *volume*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.3, 6.3.8, 6.3.9, 6.3.10.4, 6.3.18.1 and 6.5.7.

d) Test Type: Capability.

A.1.2 Simple operations for geometric primitives

A.1.2.1 Simple operations for 0-dimensional geometric primitives

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.1 and that the instantiation of GM_Point includes the operations *boundary*, *mbRegion* and *representativePoint*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.1, 6.2.2.2, 6.2.2.3, 6.2.2.4, 6.3.10.2 and 6.3.11.3.

d) Test Type: Capability.

A.1.2.2 Simple operations for 1-dimensional geometric primitives

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.2 and A.1.2.1 and that the instantiation of GM_Curve includes the operations *boundary*, *mbRegion* and *representativePoint*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.2, A.1.2.1, 6.2.2.2, 6.2.2.3, 6.2.2.4, 6.3.10.2 and 6.3.14.2.

d) Test Type: Capability.

A.1.2.3 Simple operations for 2-dimensional geometric primitives

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.3 and A.1.2.2 and that the instantiation of GM_Surface supports the operations *boundary*, *mbRegion* and *representativePoint*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.3, A.1.2.2, 6.2.2.2, 6.2.2.3, 6.2.2.4, 6.3.10.2 and 6.3.15.2.

d) Test Type: Capability.

A.1.2.4 Simple operations for 3-dimensional geometric primitives

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.4 and A.1.2.3 and that the instantiation of GM_Solid supports the operations *boundary*, *mbRegion* and *representativePoint*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.4, A.1.2.3, 6.2.2.2 6.2.2.3, 6.2.2.4, 6.3.10.2 and 6.3.18.2.

d) Test Type: Capability.

A.1.3 Complete operations for geometric primitives

A.1.3.1 Complete operations for 0-dimensional geometric primitives

a) Test Purpose: Verify that an application schema or profile instantiates GM_Point and GM_MultiPoint with all attributes, operations, and associations defined specifically for those classes as well as those inherited from GM_Object and GM_Primitive, except for the association *Complex* and the operation *maximalComplex*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, 6.1, 6.2, 6.3.10, 6.3.11, 6.3.12, 6.4.1 – 6.4.5 and 6.5.1 – 6.5.4.

d) Test Type: Capability.

A.1.3.2 Complete operations for 1-dimensional geometric primitives

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.3.1, and that it instantiates GM_Curve, GM_CurveSegment, and GM_MultiCurve, either directly or through a non-abstract subtype. Verify that these instantiations support all attributes, operations, and associations defined specifically for those classes as well as those inherited from GM_Object, GM_GenericCurve, and GM_Primitive, except for the association *Complex* and the operation *maximalComplex*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.3.1, 6.3.1, 6.3.2, 6.3.4, 6.3.5, 6.3.13, 6.3.14, 6.3.16, 6.4.6 – 6.4.31, and 6.5.5.

d) Test Type: Capability.

A.1.3.3 Complete operations for 2-dimensional geometric primitives

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.3.2, and that it instantiates GM_Surface, GM_SurfacePatch, and GM_MultiSurface either directly or through a non-abstract subtype. Verify that these instantiations support all attributes, operations, and associations defined specifically for those classes as well as those inherited from GM_Object, GM_GenericSurface, and GM_Primitive, except for the association *Complex* and the operation *maximalComplex*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.3.2, 6.3.6, 6.3.7, 6.3.15, 6.3.17, 6.4.32 – 6.4.48, and 6.5.6.

d) Test Type: Capability.

A.1.3.4 Complete operations for 3-dimensional geometric primitives

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.3.3, and that it instantiates GM_Solid and GM_MultiSolid. Verify that these instantiations support all attributes, operations, and associations defined specifically for those classes as well as those inherited from GM_Object, GM_GenericSolid, and GM_Primitive, except for the association *Complex* and the operation *maximalComplex*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.3.3, 6.3.8, 6.3.18, and 6.5.7.

d) Test Type: Capability.

A.2 Geometric complexes

A.2.1 Data types for geometric complexes

A.2.1.1 Data types for 1-dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.2 and that it instantiates GM_Complex, GM_CompositePoint, and GM_CompositeCurve. Verify that it supports the associations *Contains* between Set<GM_Primitive> and GM_Complex, *Complex* between the GM_Primitives (GM_Point and GM_Curve) and GM_Complex, and *Composition* between GM_Point and GM_CompositePoint and between GM_Curve and GM_CompositeCurve.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.2, 6.6.3, 6.6.1, 6.6.2.1, 6.6.2.3, 6.6.2.4, and 6.6.3 – 6.6.5.

d) Test Type: Capability.

A.2.1.2 Data types for 2-dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.3 and A.2.1.1, and that it instantiates GM_CompositeSurface with the associations *Complex* between GM_Surface and GM_Complex and *Composition* between GM_Surface and GM_CompositeSurface.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.3, A.2.1.1, and 6.6.6.

d) Test Type: Capability.

A.2.1.3 Data types for 3- dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.4 and A.2.1.2, and that it instantiates GM_CompositeSolid with the associations *Complex* between GM_Solid and GM_Complex and *Composition* between GM_Solid and GM_CompositeSolid.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.4, A.2.1.2, and 6.6.7.

d) Test Type: Capability.

A.2.2 Simple operations for geometric complexes

A.2.2.1 Simple operations for 1-dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.2.2 and A.2.1.1. Verify that the instantiations of GM_Complex, GM_CompositePoint, and GM_CompositeCurve support the operations *boundary*, *envelope*, *representative point*, and *isMaximal*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.2.2, A.2.1.1, and 6.6.2.2.

d) Test Type: Capability.

A.2.2.2 Simple operations for 2-dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.2.3, A.2.1.2 and A.2.2.1. Verify that the instantiations of GM_CompositeSurface support the operations *boundary*, *envelope*, *representative point*, and *isMaximal*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.2.3, A.2.1.2, and A.2.2.1.

d) Test Type: Capability.

A.2.2.3 Simple operations for 3-dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.2.4, A.2.1.3 and A.2.2.1. Verify that the instantiations of GM_CompositeSolid support the operations *boundary*, *envelope*, *representative point*, *maximal*, and *isMaximal*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.2.4, A.2.1.3, and A.2.2.1.

d) Test Type: Capability.

A.2.3 Complete operations for geometric complexes

A.2.3.1 Complete operations for 1-dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.3.2 and A.2.2.1. Verify that instantiations of GM_CompositePoint, GM_CompositeCurve, and GM_Complex support all attributes, operations, and associations defined specifically for those classes as well as those inherited from GM_Object, GM_Complex and GM_Composite.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.3.2 and A.2.2.1.

d) Test Type: Capability.

A.2.3.2 Complete operations for 2-dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.3.3 and A.2.2.2. Verify that instantiations of GM_CompositeSurface support all attributes, operations, and associations defined specifically for that class as well as those inherited from GM_Object, GM_Complex, and GM_Composite.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.3.3 and A.2.2.2.

d) Test Type: Capability.

A.2.3.3 Complete operations for 3-dimensional geometric complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.3.4 and A.2.2.3. Verify that instantiations of GM_CompositeSolid support all attributes, operations, and associations defined specifically for that class as well as those inherited from GM_Object, GM_Complex, and GM_Composite.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.3.4, and A.2.2.3.

d) Test Type: Capability.

A.3 Topological complexes

A.3.1 Topological complexes for data types

A.3.1.1 1-dimensional topological complexes for data types

a) Test Purpose: Verify that an application schema or profile instantiates TP_Complex, TP_Node, TP_DirectedNode, TP_Edge and TP_DirectedEdge. Verify that the instantiations of both TP_DirectedNode and TP_DirectedEdge support the attribute *orientation*. Verify that the application schema or profile supports the association *Complex* between TP_Complex and each of the TP_Primitives (TP_Node and TP_Edge). Verify that it supports the association *Center* between TP_Node and TP_DirectedNode, and between TP_Edge and TP_DirectedEdge. Verify that it supports the derived associations *Boundary* between TP_DirectedNode and TP_Edge, and *Coboundary* between TP_Node and TP_DirectedEdge.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, 7.1, 7.2.1, 7.3.1, 7.3.2, 7.3.3, 7.3.8.1, 7.3.8.3, 7.3.9.1, 7.3.9.2, 7.3.9.5, 7.3.9.6, 7.3.10, 7.3.11, 7.3.12.1, 7.3.12.3, 7.3.12.4, 7.3.13, 7.4.1, 7.4.2.1, 7.4.2.3, 7.4.2.6, and 7.4.2.7.

d) Test Type: Capability.

A.3.1.2 2-dimensional topological complexes for data types

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.3.1.1 and that it instantiates TP_Face and TP_DirectedFace. Verify that the instantiation of TP_DirectedFace supports the attribute *orientation*. Verify that the application schema or profile supports the association *IsolatedIn* between TP_Primitives, the association *Complex* between TP_Complex and TP_Face, and the association *Center* between TP_Face and TP_DirectedFace. Verify that it supports the associations *Boundary* between TP_DirectedEdge and TP_Face, and *Coboundary* between TP_Edge and TP_DirectedFace.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.3.1.1, 7.3.4, 7.3.6, 7.3.8.4, 7.3.14.1, 7.3.14.4, and 7.3.14.5.

d) Test Type: Capability.

A.3.1.3 3-dimensional topological complexes for data types

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.3.1.2 and that it instantiates TP_Solid and TP_DirectedSolid. Verify that the instantiation of TP_DirectedSolid supports the attribute *orientation*. Verify that the application schema or profile supports the association *Complex* between TP_Complex and TP_Solid and the association *Center* between TP_Solid and TP_DirectedSolid. Verify that it supports the associations *Boundary* between TP_DirectedFace and TP_Solid, and *Coboundary* between TP_Face and TP_DirectedSolid.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.3.1.2, 7.3.5, 7.3.7, 7.3.16.1, 7.3.16.4, and 7.3.16.5.

d) Test Type: Capability.

A.3.2 Simple operations for topological complexes

A.3.2.1 Simple operations for 1-dimensional topological complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.3.1.1 and that the instantiations of TP_Complex, TP_Node, and TP_Edge each support the operations *boundary*, *coboundary*, and *maximalComplex*. Verify that the instantiations of TP_Complex support the operation *isMaximal*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.3.1.1, 7.2.2.3, 7.2.2.4, 7.2.2.8, 7.3.12.2, and 7.4.2.4.

d) Test Type: Capability.

A.3.2.2 Simple operations for 2-dimensional topological complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.3.1.2 and A.3.2.1 and that the instantiations of TP_Face each support the operations *boundary*, *coboundary*, and *maximal*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.3.1.2, A.3.2.1, 7.3.14.2 and 7.4.14.3.

d) Test Type: Capability.

A.3.2.3 Simple operations for 3-dimensional topological complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.3.1.3 and A.3.2.2 and that the instantiations of TP_Solid each support the operations *boundary*, *coboundary*, and *maximal*.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.3.1.3, A.3.2.2, 7.3.16.2 and 7.3.16.3.

d) Test Type: Capability.

A.3.3 Complete operations for topological complexes

A.3.3.1 Complete operations for 1-dimensional topological complexes

a) Test Purpose: Verify that an application schema or profile instantiates TP_Complex, TP_Expression, TP_ExpressionTerm, TP_Node, TP_DirectedNode, TP_Edge and TP_DirectedEdge. Verify that these instantiations support all attributes, associations, and operations defined for those classes and inherited from TP_Object, TP_Primitive, and TP_DirectedTopo, except for the *Realization* associations.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, 7.1, 7.2, 7.3.1, 7.3.2, 7.3.3, 7.3.6, 7.3.8.1, 7.3.8.3, 7.3.8.4, 7.3.9 – 7.3.13, 7.3.18, 7.4.1 and 7.4.2.1 – 7.4.2.7.

d) Test Type: Capability.

A.3.3.2 Complete operations for 2-dimensional topological complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.3.3.1 and that it instantiates TP_Face and TP_DirectedFace. Verify that the instantiations of TP_Face and TP_DirectedFace support all attributes, associations, and operations defined for those classes and inherited from TP_Object, TP_Primitive, and TP_DirectedTopo, except for the *Realization* associations.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.3.3.1, 7.3.4, 7.3.14, and 7.3.15.

d) Test Type: Capability.

A.3.3.3 Complete operations for 3-dimensional topological complexes

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.3.3.2 and that it instantiates TP_Solid and TP_DirectedSolid. Verify that the instantiations of TP_Solid and TP_DirectedSolid support all attributes, associations, and operations defined for those classes and inherited from TP_Object, TP_Primitive, and TP_DirectedTopo, except for the *Realization* associations.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.3.3.2, 7.3.5, 7.3.16 and 7.3.17.

d) Test Type: Capability.

A.4 Topological complexes with geometric realization

A.4.1 Topological complexes with geometric realization for data types

A.4.1.1 1-dimensional topological complexes with geometric realization for data types

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.2 and A.3.1.1. Verify that it also supports the *Realization* associations between the instantiations of the TP_Primitives (TP_Node and TP_Edge) and the GM_Primitives (GM_Point and GM_Curve), and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.2, A.3.1.1, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.4.1.2 2-dimensional topological complexes with geometric realization for data types

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.3 and A.3.1.2 and that it supports the *Realization* associations between the instantiations of TP_Face and GM_Surface, and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.3, A.3.1.2, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.4.1.3 3-dimensional topological complexes with geometric realization for data types

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.1.4 and A.3.1.3 and that it supports the *Realization* associations between the instantiations of TP_Solid and GM_Solid, and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.1.4, A.3.1.3, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.4.2 Simple operations for topological complexes with geometric realization

A.4.2.1 Simple operations for 1-dimensional topological complexes with geometric realization

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.2.2 and A.3.2.1 and that it supports the *Realization* associations between the instantiations of TP_Primitives (TP_Node and TP_Edge) and GM_Primitives (GM_Point and GM_Curve), and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.2.2, A.3.2.1, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.4.2.2 Simple operations for 2-dimensional topological complexes with geometric realization

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.2.3 and A.3.2.2 and that it supports the *Realization* associations between the instantiations of TP_Face and GM_Surface, and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.2.3, A.3.2.2, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.4.2.3 Simple operations for 3-dimensional topological complexes with geometric realization

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.1.2.4 and A.3.2.3 and that it supports the *Realization* associations between the instantiations of TP_Solid and GM_Solid, and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.1.2.4, A.3.2.3, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.4.3 Complete operations for topological complexes with geometric realization

A.4.3.1 Complete operations for 1-dimensional topological complexes with geometric realization

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.2.3.1 and A.3.3.1. Verify that it also supports the *Realization* associations between the instantiations of the

TP_Primitives (TP_Node and TP_Edge) and the GM_Primitives (GM_Point and GM_Curve), and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.2.3.1, A.3.3.1, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.4.3.2 Complete operations for 2-dimensional topological complexes with geometric realization

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.2.3.2 and A.3.3.2 and that it supports the *Realization* associations between the instantiations of TP_Face and GM_Surface, and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.2.3.1, A.3.3.1, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.4.3.3 Complete operations for 3-dimensional topological complexes with geometric realization

a) Test Purpose: Verify that an application schema or profile satisfies all the requirements of A.2.3.3 and A.3.3.3 and that it supports the *Realization* associations between the instantiations of TP_Solid and GM_Solid, and between the instantiations of TP_Complexes and GM_Complexes.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.2.3.1, A.3.3.1, 7.3.8.2 and 7.4.2.8.

d) Test Type: Capability.

A.5 Boolean operators

A.5.1 Set operators

a) Test Purpose: Verify that an application schema or profile defines all the set operators specified in 8.2 consistently with that subclause.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, 8.2.

d) Test Type: Capability.

A.5.2 Egenhofer operators

a) Test Purpose: Verify that an application schema or profile defines all the Egenhofer operators specified in 8.3 consistently with that subclause.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, 8.3.

d) Test Type: Capability.

A.5.3 Full topological operators

a) Test Purpose: Verify that an application schema or profile defines all the set operators specified in 8.4 consistently with that subclause.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, 8.4.

d) Test Type: Capability.

A.5.4 All topological operators

a) Test Purpose: Verify that an application schema or profile defines all the set operators specified in Clause 8 consistently with that clause.

b) Test Method: Inspect the documentation of the application schema or profile.

c) Reference: ISO 19107:2003, A.5.1, A.5.2, and A.5.3.

d) Test Type: Capability.

Annex B
(informative)

Conceptual organization of terms and definitions

B.1 Introduction

This Annex presents the terms and definitions from Clause 4 in an arrangement based on their conceptual relationships.

B.2 General terms

application (4.1)
manipulation and processing of data in support of user requirements [ISO 19101]

application schema (4.2)
conceptual schema for data required by one or more **applications** [ISO 19101]

boundary (4.4)
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.

feature (4.39)
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.

feature attribute (4.40)
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. [ISO 19109]

geographic information (4.42)
information concerning phenomena implicitly or explicitly associated with a location relative to the Earth [ISO 19101]

spatial object (4.69)
object used for representing a spatial characteristic of a feature

spatial operator (4.70)
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.

B.3 Collections and related terms

set (4.65)
unordered collection of related items (**objects** or values) with no repetition

sequence (4.64)
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.

bag (4.3)
finite, unordered collection of related items (**objects** or values) that may be repeated

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

circular sequence (4.6)
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

record (4.62)
finite, named collection of related items (**objects** or values)

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

domain (4.32)
well-defined **set** [ISO/TS 19103]

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

function (4.41)
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)

B.4 Modelling terms

class (4.7)
description of a **set** of **objects** that share the same attributes, operations, methods, relationships, and semantics [ISO/TS 19103]

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.

object (4.59)
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.

instance (4.53)
object that realizes a **class**

strong substitutability (4.73)
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.

B.5 Positioning terms

direct position (4.26)
position described by a single set of **coordinates** within a **coordinate reference system**

coordinate (4.19)
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.

coordinate reference system (4.21)
coordinate system that is related to the real world by a datum [ISO 19111]

coordinate system (4.22)
set of mathematical rules for specifying how **coordinates** are to be assigned to **points** [ISO 19111]

coordinate dimension (4.20)
number of measurements or axes needed to describe a position in a **coordinate system**

B.6 Geometric terms

B.6.1 General geometric concepts

vector geometry (4.86)
representation of **geometry** through the use of constructive **geometric primitives**

computational geometry (4.13)
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**.

geometric set (4.50)
set of **direct positions**

NOTE This set in most cases is infinite.

convex set (4.18)
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.

convex hull (4.17)
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”.

neighbourhood (4.57)
geometric set containing a specified **direct position** in its **interior**, and containing all direct positions within a specified distance of the specified direct position

geometric dimension (4.46)
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.

B.6.2 Geometric objects

B.6.2.1 General consepts

geometric object (4.47)
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**.

geometric boundary (4.44)
boundary represented by a **set** of **geometric primitives** of smaller **geometric dimension** that limits the extent of a **geometric object**

cycle (4.25)
<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.

interior (4.54)
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.

exterior (4.37)
difference between the universe and the **closure**

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

closure (4.8)
union of the **interior** and **boundary** of a **topological** or **geometric object**

simple (4.67)
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.

connected (4.15)
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.

buffer (4.5)
geometric object that contains all **direct positions** whose distance from a specified **geometric object** is less than or equal to a given distance

geometric aggregate (4.43)
collection of **geometric objects** that has no internal structure

geometric boundary (4.44)
boundary represented by a set of **geometric primitives** of smaller **geometric dimension** that limits the extent of a **geometric object**

B.6.2.2 Geometric primitives and related terms

geometric primitive (4.48)
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**.

point (4.61)
0-dimensional **geometric primitive**, representing a position

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

curve (4.23)
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.

start point (4.72)
first **point** of a **curve**

end point (4.36)
last **point** of a **curve**

curve segment (4.24)
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.

ring (4.63)
simple curve which is a **cycle**

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

surface (4.75)
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**.

surface patch (4.76)
2-dimensional, **connected geometric object** used to represent a continuous portion of a **surface** using homogeneous interpolation and definition methods

shell (4.66)
simple surface which is a **cycle**

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

solid (4.68)
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**.

B.6.2.3 Geometric complexes

geometric complex (4.45)
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**.

subcomplex (4.74)
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.

composite curve (4.10)
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.

composite solid (4.11)
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.

composite surface (4.12)
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.

B.7 Topological terms

B.7.1 Topological concepts

computational topology (4.14)
topological concepts, structures and algebra that aid, enhance or define **operations** on **topological objects** usually performed in **computational geometry**

B.7.2 Topological objects

B.7.2.1 General concepts

topological boundary (4.77)
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.

topological complex (4.78)
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**.

topological object (4.81)
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**.

coboundary (4.9)
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.

topological dimension (4.79)
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 $P(u, v) = A + u\,X + v\,Y$, 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.

B.7.2.2 Topological primitives and related terms

topological primitive (4.82)
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**.

node (4.58)
0-dimensional **topological primitive**

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

connected node (4.16)
node that starts or ends one or more **edges**

isolated node (4.55)
node not related to any **edge**

start node (4.71)
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

end node (4.35)
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

edge (4.33)
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**.

face (4.38)
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**.

topological solid (4.83)
3-dimensional **topological primitive**

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

B.7.2.3 Topological complexes and related terms

topological complex (4.78)
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.**

universal face (4.84)
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.

universal solid (4.85)
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.

topological expression (4.80)
collection of oriented **topological primitives** which is operated upon like a multivariate polynomial

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

directed topological object (4.31)
topological object that represents a logical association between a **topological primitive** and one of its orientations

directed node (4.29)
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".

directed edge (4.27)
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.

directed face (4.28)
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**.

directed solid (4.30)
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.

B.7.2.4 Types of topological complexes

graph (4.51)
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.

edge-node graph (4.34)
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.

planar topological complex (4.60)
topological complex that has a **geometric realization** that can be embedded in Euclidean 2 space

B.8 Relationship of geometric and topological complexes

homomorphism (4.52)
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. Automated translators from one language to another are usually homomorphic in that they can preserve the sense of the statements. They are seldom isomorphic, since they cannot be made to always map target sentences back to their original source, due to idiomatic distinctions and irregularities, and the culturally specific use of metaphor to convey meaning. Even in simple cases where the vocabulary and grammar are essentially the same, such as British English and American English, mainly due to idiomatic expressions that are culturally derived, such as the American phrase “that dog won't hunt” which means a particular line of reasoning is invalid.

isomorphism (4.56)
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.

geometric realization (4.49)
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.

Annex C
(informative)

Examples of spatial schema concepts

C.1 Geometry

C.1.1 Semantics

The examples here use the names of the types in the normative part of this document as if they were instantiable classes. While not the normal UML semantics for type, this mnemonic is justifiable under several interpretations. First, the conformance clause does not require that the types in this International Standard be included in an application schema, but that classes in the application schema realize these types. This logical requirement does not require the instantiated classes to be named differently from the standard's types and interfaces. Second, assuming a design system that uses a strong name space convention, items in different name spaces can have the same local name. In other words, local names are not globally unique. Third, the examples are valid for any implementation classes that realize the types so identified. Any implementation (application schema) would have to have a schema map that associates these types with implementation classes that realize them. The proper use of that map would result in valid syntax.

In general, it is valid to use common names for “metaphorically identical” but technically different entities. 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 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.

C.1.2 Geometric objects in a 2-dimensional coordinate reference system

This example is based on a simple decoding scenario. This is used as opposed to an editing use case because it eliminates the need to discuss the fine points of creating a viable topology editor. The following assumptions are made about the application schema (defined in accordance with Rules for application schema):

a) The geometry and topology schema are compliant with the spatial schema defined in this document, and therefore include instantiable subclasses of the major geometry and topology types defined in the normative part of this document. For the sake of readability, the type names used in the normative part of this document are used in lieu of their instantiable subtypes.

b) The schema includes the requirement to use a full planar topology.

c) The schema includes a 2D coordinate reference system.

d) The feature schema includes the equivalent of theme, feature and feature components as described in the discussion of MiniTopo in Annex D.

e) Persistent objects, after creation, are inserted into a datastore called “Datastore”.

Figure C.1 represents the geometry of a GM_Complex, based on a planar manifold. To construct this complex, the following example uses a functional cascade, where objects are created with constructors based on the coordinates given in the diagram. Once an object has been created it can be used in any subsequent formulation. For objects not given formal constructors in the normative section, a default one is assumed that simply takes a record representation of the state of the object and uses it as a parameter to a data type-like

constructor. This is very consistent with how this would be done in SQL 99. SQL automatically creates default constructors for any UDT (user defined type) based on the requirements of an insert semantics. Recall that "< >" denotes a record, or an ordered set (list), and that "{ }" denotes an unordered set or bag.

Construction can begin with the creation of the points. There is a minor issue here since GM_Point, being a type, cannot be instantiated. To be a compliant application schema, a instantiable class that is a subtype of GM_Point must be included, and this class would have to be substituted in the creation cascade below for each use of GM_Point. First, the 7 GM_Points, indicated by dots and identified as {P1, . . . P7} are created:

```
P1 = GM_Point < position = < 1.00, 5.00 > >
P2 = GM_Point < position = < 3.00, 5.00 > >
P3 = GM_Point < position = < 3.00, 2.00 > >
P4 = GM_Point < position = < 1.75, 2.75 > >
P5 = GM_Point < position = < 1.50, 4.50 > >
P6 = GM_Point < position = < 2.00, 3.25 > >
P7 = GM_Point < position = < 5.00, 4.00 > >
Insert P1, P2, P3, P4, P5, P6, P7 into Datastore
```

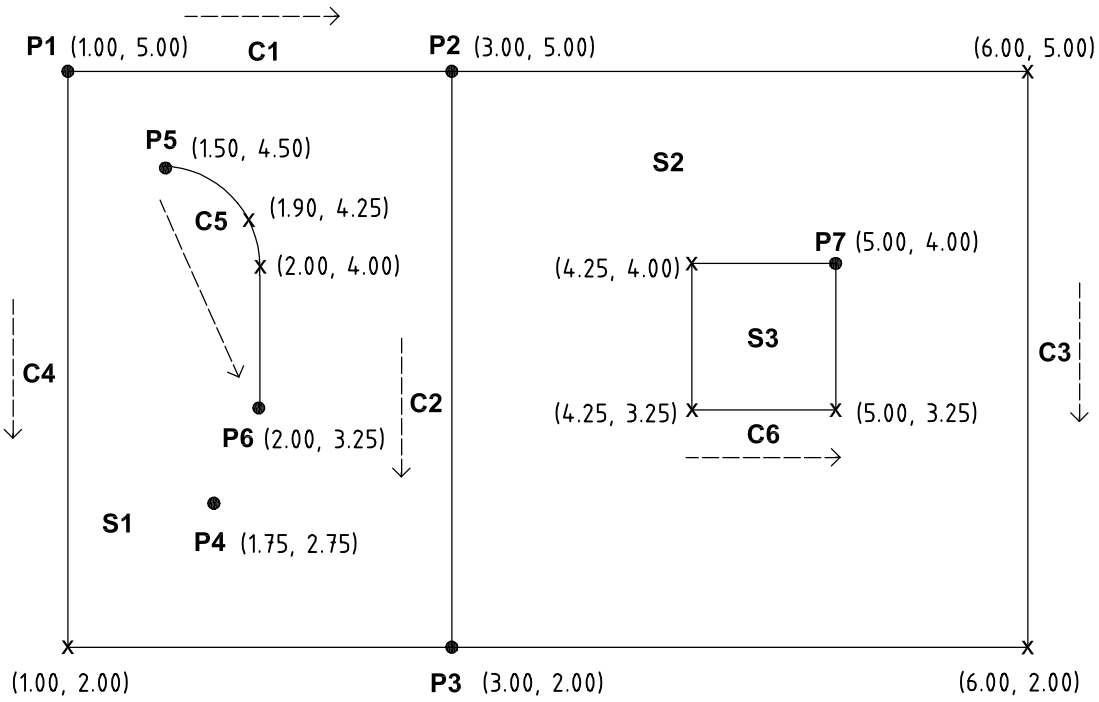


Figure C.1 — A data set composed of the GM_Primitives

With the existence of the points, the cascade can continue with the construction of the 7 GM_CurveSegments, identified {CS1, CS2, CS3, CS4, CS5, CS6, CS7} which can be used to construct the curves to follow. Recall that subtypes of GM_CurveSegment are data types and cannot hold persistent identification. Thus, the variables used to define the curve segments below are "heap" or local variables, defined within the context of the construction, but not persistently stored until they are included as members of an object type (in this case, the curves defined later). All of the curve segments defined here are either line strings or arcs.

```
CS1 = GM_CurveSegment <controlPoint = <P1,P2>, interpolation = "linear" >
CS2 = GM_CurveSegment <controlPoint = <P2,P3 >, interpolation = "linear" >
CS3 = GM_CurveSegment <controlPoint = <P2,(6,5),(6,2),P3>,
      interpolation = "linear" >
CS4 = GM_CurveSegment <controlPoint = <P1,(1,2), P3> ,
      interpolation = "linear" >
CS5 = GM_CurveSegment <controlPoint = <P5,(1.9,4.25), (2,4)>
      interpolation = "arc">
```

```
CS6 = GM_CurveSegment <controlPoint = <(2,4),P6>, interpolation = "linear" >
CS7 = GM_CurveSegment <controlPoint = <P7,(4.25,4),(4.25,3.25),(5,3.25),P7 >,
      interpolation ="linear">
```

There is a hidden assumption here that the persistent variables, such as P1, which have previously been entered into the datastore can be accessed so that the local copy and persistent copy are maintained in synchrony. This allows the insertion of the curve segments (as members of the curves below) to proceed while still using the GM_Point variant of the GM_Position data type. In an object relational database scenario using only an SQL language application program interface (API), the application would track references to variables and use them in subsequent insert statements. In a similar scenario using an object interface to the same datastore, the database API would make this tracking issue transparent to the programmer.

The curve segments can now be used to construct persistent objects: 6 GM_Curves, identified as {C1, . . . C6}. The same comment about instantiable types applies, in that the local application schemas' required subtype of GM_Curve would have to be used instead of GM_Curve.

```
C1 = GM_Curve segments = <CS1>
C2 = GM_Curve segments = <CS2>
C3 = GM_Curve segments = <CS3>
C4 = GM_Curve segments = <CS4>
C5 = GM_Curve segments = <CS5, CS6>
C6 = GM_Curve segments = <CS7>
Insert C1, C2, C3, C4, C5, C6 into Datastore
```

The curves can then be used in the construction of surfaces. In this case, the planar polygon constructor can be used, since our coordinate space is 2D. The upNormal of the surfaces is the standard upNormal of the surface (often denoted as k), and need not be specified. Since the intent is to define a full topology complex, we need a complete coverage by surfaces of the area of the coordinate surfaces. Since the universal face is often referred to as "Face 0", we define here a S0 to be the geometric realization of that face. Thus, the 4 GM_Surfaces are identified as {S0, S1, S2, S3}.

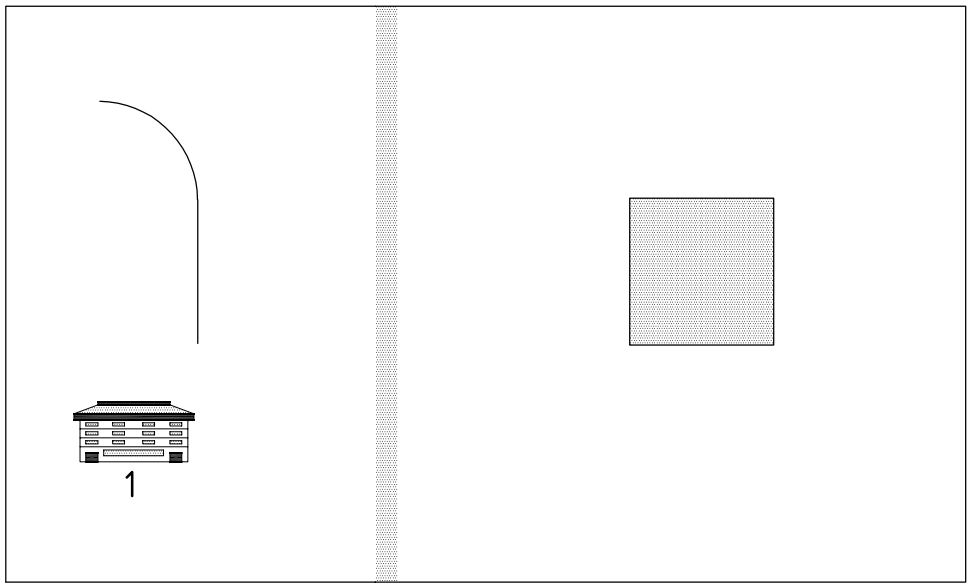
```
S0 = GM_Surface patch = <GM_Polygon interior = << C1, C3, -C4 >> >
-- this universal face is only needed to construct a topological complex
-- with a full planar graph
S1 = GM_Surface patch = <GM_Polygon exterior = < C4, -C2, -C1 >,
                                    interior = << C5, -C5 >> >
S2 = GM_Surface patch = <GM_Polygon exterior = < -C3, C2 >,
                                    interior = << -C6 >> >
S3 = GM_Surface patch = <GM_Polygon exterior = < C6 > >
Insert S0, S1, S2, S3 into Datastore
```

All the necessary pieces of geometry exist for the creation of a GM_Complex, which is a type of GM_Object collection, it is necessary only to give an exhaustive list of the required objects. This can cascade directly into creation of a TP_Complex.

```
GComplex = GM_Complex < surfaces = {S0, S1, S2, S3},
                               curves   = {C1, C2, C3, C4, C5, C6}
                               points   = {P1, P2, P3, P4, P5, P6, P7} >
TComplex = TP_Complex < realization = GComplex >
Insert GComplex, TComplex into Datastore
```

This concludes the geometric constructions describing the geometry and topology in the diagram at the beginning of this clause. Although out of the control of this document, the construction of features (Figure C.2) might conclude this scenario as follows:

```
Lake  =  AreaFeature featureType="Hydrography::WaterBody", extent = S3
RoadCenterline =  LineFeature featureType = "Transportation::Road",
                        centerline = C2
RoadArea =  RoadCenterLine.centerline.buffer < distance = 10m >
RoadExtent  =  AreaFeature featureType = "LandCover::Road"
                        extent = RoadArea
RoadInstance = ComplexFeature featureType = "LandUse::Road",
                        featureComponents = {RoadCenterline, RoadArea }
Trail       =  LineFeature featureType = "CulturalFacilities::HikingTrail",
                        centerline = C5
School      =  PointFeature featureType = "CulturalFacilities::School",
                        Location = P4
Insert Lake, RoadCenterline, RoadExtent, RoadInstance, Trail, School
            into Datastore
```



Key

1 school

Figure C.2 — Simple cartographic representation of sample data

C.1.3 Geometric objects in a 3-dimensional coordinate reference system

In Figure C.3, we have a 3D solid with planar facets. It is a rectangular block into which has been cut a rectangular slot, which is counter sunk by one unit.

```
P1 = GM_Point position = <2.00, 5.00, 4.00>
P2 = GM_Point position = <5.00, 5.00, 4.00>
P3 = GM_Point position = <5.00, 3.00, 4.00>
P4 = GM_Point position = <2.00, 3.00, 4.00>
P5 = GM_Point position = <2.00, 5.00, 2.00>
P6 = GM_Point position = <5.00, 5.00, 2.00>
P7 = GM_Point position = <5.00, 3.00, 2.00>
P8 = GM_Point position = <2.00, 3.00, 2.00>
P9 = GM_Point position = <1.00, 5.00, 1.00>
P10 = GM_Point position = <9.00, 5.00, 1.00>
P11 = GM_Point position = <9.00, 1.00, 1.00>
P12 = GM_Point position = <1.00, 1.00, 1.00>
P13 = GM_Point position = <1.00, 5.00, 7.00>
P14 = GM_Point position = <9.00, 5.00, 7.00>
P15 = GM_Point position = <9.00, 1.00, 7.00>
P16 = GM_Point position = <1.00, 1.00, 7.00>
```

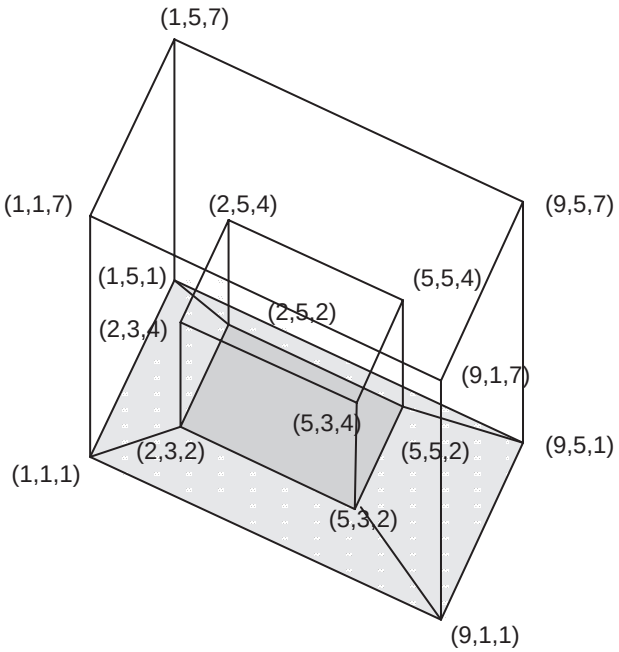


Figure C.3 — A 3D Geometric object with labeled coordinates

The surface can be expressed as a GM_GriddedSurface (wrapped around on itself to make a topological cylinder) and 2 GM_Polygons (to act as end caps for the topological cylinder), all with planar interpolations.

```
S1 = GM_Surface patch =
      < <GM_BilinearGrid rows = 4, columns = 5,
         controlPoint = <  <P1, P2, P3, P4, P1>,
                           <P5, P6, P7, P8, P5>
                           <P9, P10,P11,P12,P9>,
                           <P13,P14,P15,P16,P13>   >  ,
         GM_Polygon exteriorVertices = <P1, P2, P3, P4, P1 >,
         GM_Polygon exteriorVertices = <P16,P15,P14,P13,P16> >
```

The example in Figure C.4 consists of a GM_Point [P1], a GM_Curve [C1], and a GM_Surface [S1]. The segmentation association of the GM_Surface points to 9 GM_SurfacePatches. The first GM_SurfacePatch represents the area to the left of the dashed line. The other 8 GM_SurfacePatches, all GM_Triangles, represent the area to the right of the dashed line.

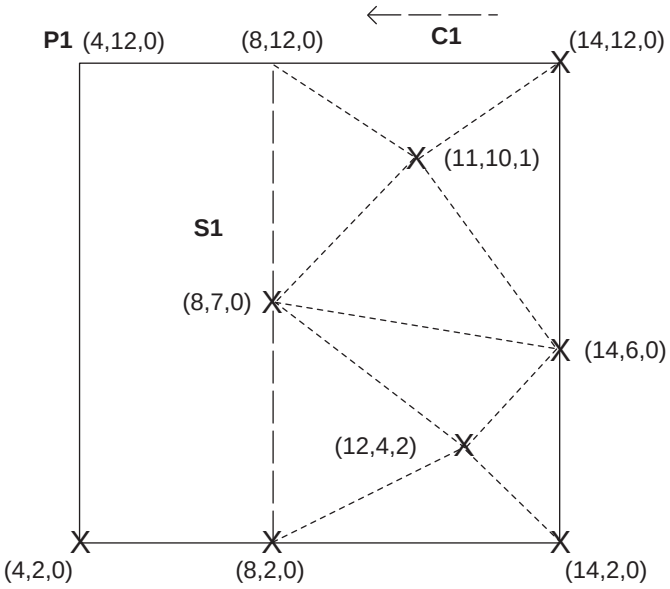


Figure C.4 — Surface example

```
P1 = GM_Point (4,12,0)
C1 = GM_Curve segment = <Segment 1>
Segment 1 = GM_CurveSegment controlPoint = <(4,12,0), (4,2,0), (14,2,0),
      (14,12,0), (4,12,0)>
Patch1 = GM_Polygon exterior = <P1, (4,2,0), (8,2,0), (8,12,0), P1>
Post1 = GM_Position (8,12,0)
Post2 = GM_Position (14,12,0)
Post3 = GM_Position (11,10,1)
Post4 = GM_Position (8,7,0)
Post5 = GM_Position (14,6,0)
Post6 = GM_Position (12,4,2)
Post7 = GM_Position (8,2,0)
Post8 = GM_Position (14,2,0)
T1 = GM_Triangle exterior = <Post1, Post2, Post3, Post1>
T2 = GM_Triangle exterior = <Post1, Post3, Post4, Post1>
T3 = GM_Triangle exterior = <Post3, Post5, Post4, Post3>
T4 = GM_Triangle exterior = <Post2, Post5, Post3, Post2>
T5 = GM_Triangle exterior = <Post4, Post5, Post6, Post4>
T6 = GM_Triangle exterior = <Post4, Post6, Post7, Post4>
T7 = GM_Triangle exterior = <Post5, Post8, Post6, Post5>
T8 = GM_Triangle exterior = <Post7, Post6, Post8, Post7>
S1 = GM_Surface patch = <Patch1, T1, T2, T3, T4, T5, T6, T7, T8>
```

Note that the same example could be described as a set of two GM_Surfaces, one composed of a single GM_SurfacePatch, P1, and the other, a GM_TriangulatedSurface composed of the eight GM_Triangles. Those two GM_Surfaces could then be combined into a GM_CompositeSurface equivalent to the single GM_Surface described above.

Annex D
(informative)

Examples for application schemata

D.1 Introduction

Application schemas built using ISO 19109, may use the packages defined in this International Standard by defining subclasses of the classes and interfaces in these packages with extensions to the member protocols (attributes, operations or both) defined here.

This mechanism defines instantiable classes that support the needed interfaces from the packages within this International Standard through structural polymorphism.

This Annex contains skeletal application schemas for geometry that have been created using this mechanism.

D.2 Simple Topology

D.2.1 Packages for Simple topology

The construction of concrete topology classes is similar to that for the geometry classes, except that the option of using multiple inheritance for the dual topological and geometric objects is used. This does not create the type of problems usually associated to a multiple inheritance schema which are associated to multiple inheritance of implementations, but care must be taken when distinguishing between the geometric boundary (GM_Object::boundary) and the topological boundary (TP_Object::boundary) of an object.

D.2.2 Classes for Simple Topology

D.2.2.1 Semantics

The types defined in this package (Figure D.1, Figure D.2) all doubly inherit the boundary operation, once from various types of TP_Primitive and once from various types of GM_Primitive. Although we have used multiple inheritance that allows different semantics for the two inheritance paths, this is not the case here. Essentially, even though the two boundary operators began in different inheritance trees, in Simple topology they are identical.

D.2.2.2 TS_Root

TS_Root acts as the root class, allowing the schema to make restrictions on all of the geometry and topology classes used in the package. The boundary of a TS_Root object only contains other TS_Root objects.

```
TS_Root:
         TP_Primitive::boudary→ isTypeOf(TS_Root);
         TP_Primitive::boundary = GM_Primitive::boundary;
```

NOTE TS_Root is subtyped from TP_Primitive and from GM_Primitive. This means that the boundary operator is doubly defined. The second constraint says that even so they are identical. This allow for a well-formed constraint based on the boundary operator without using resolutions. Since the boundary operator for GM_Primitive and TP_Primitive are isomorphic and are identical in this case, the constraint could just have easily been done on a boundary operation inherited from either primitive.

D.2.2.3 TS_Node

TS_Node multiply inherits from TP_Node and GM_Point, allowing it to support both topological and geometric data and functionality.

D.2.2.4 TS_Edge

TS_Edge multiply inherits from TP_Edge and GM_Curve, allowing it to support both topological and geometric data and functionality.

D.2.2.5 TS_DirectedEdge

TS_DirectedEdge multiply inherits from TP_DirectedEdge and GM_OrientableCurve, allowing it to support both topological and geometric data and functionality.

D.2.2.6 TS_Face

TS_Face multiply inherits from TP_Face and GM_OrientableSurface, allowing it to support both topological and geometric data and functionality.

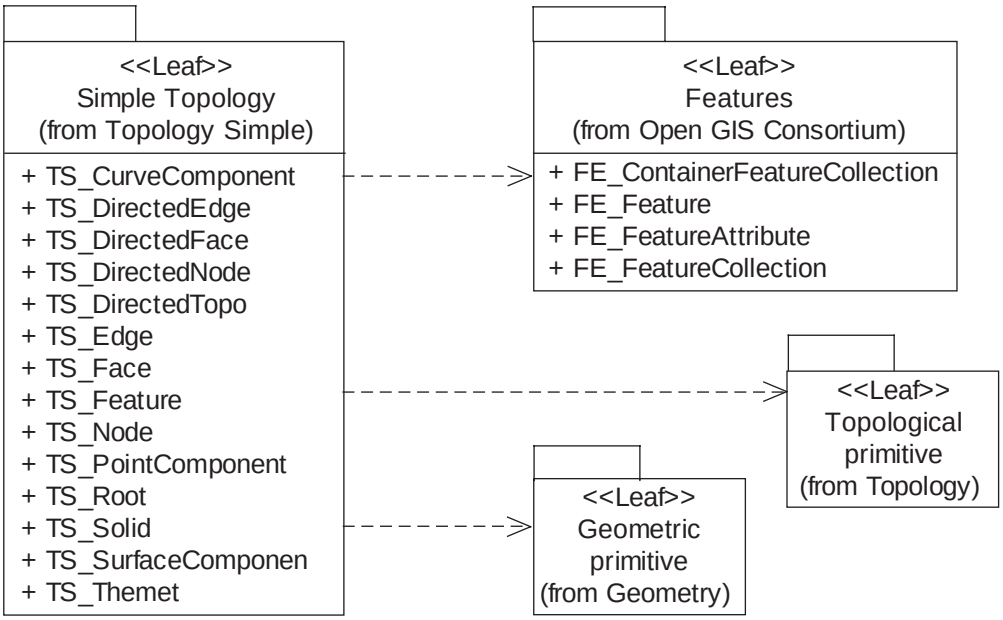


Figure D.1 — Packages and classes for simple topology

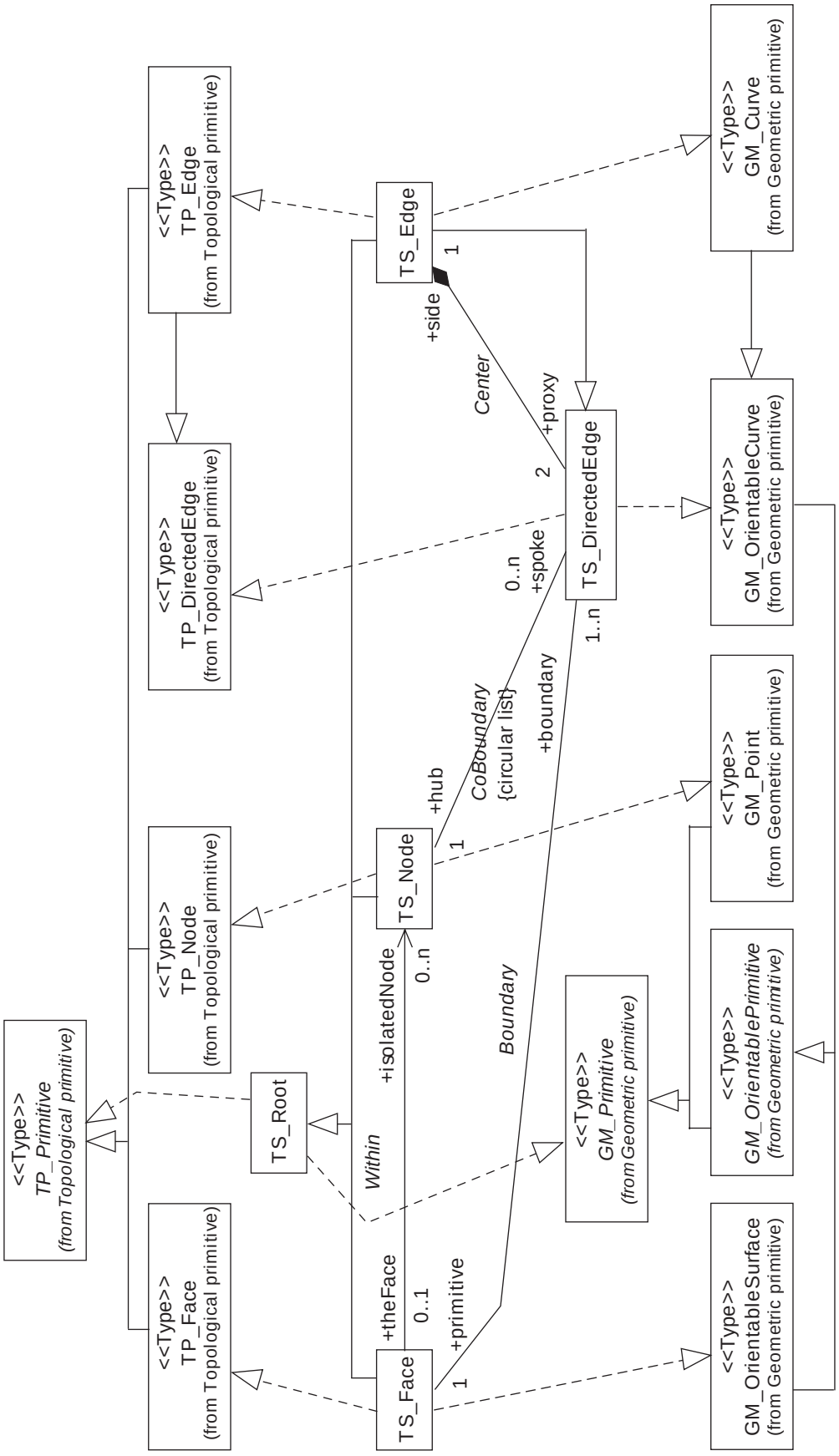


Figure D.2 — Topology and geometry classes in simple topology

D.2.2.7 TS_Theme

TS_Theme (Figure D.3) act analogously to GM_Complex, by gathering together similar geometric objects, in this case, the various features and feature components of related types, such as transportation, or political boundaries. TS_Theme inherits from GM_Complex. It could also be subclassed from Feature to allow it to hold Feature Attributes.

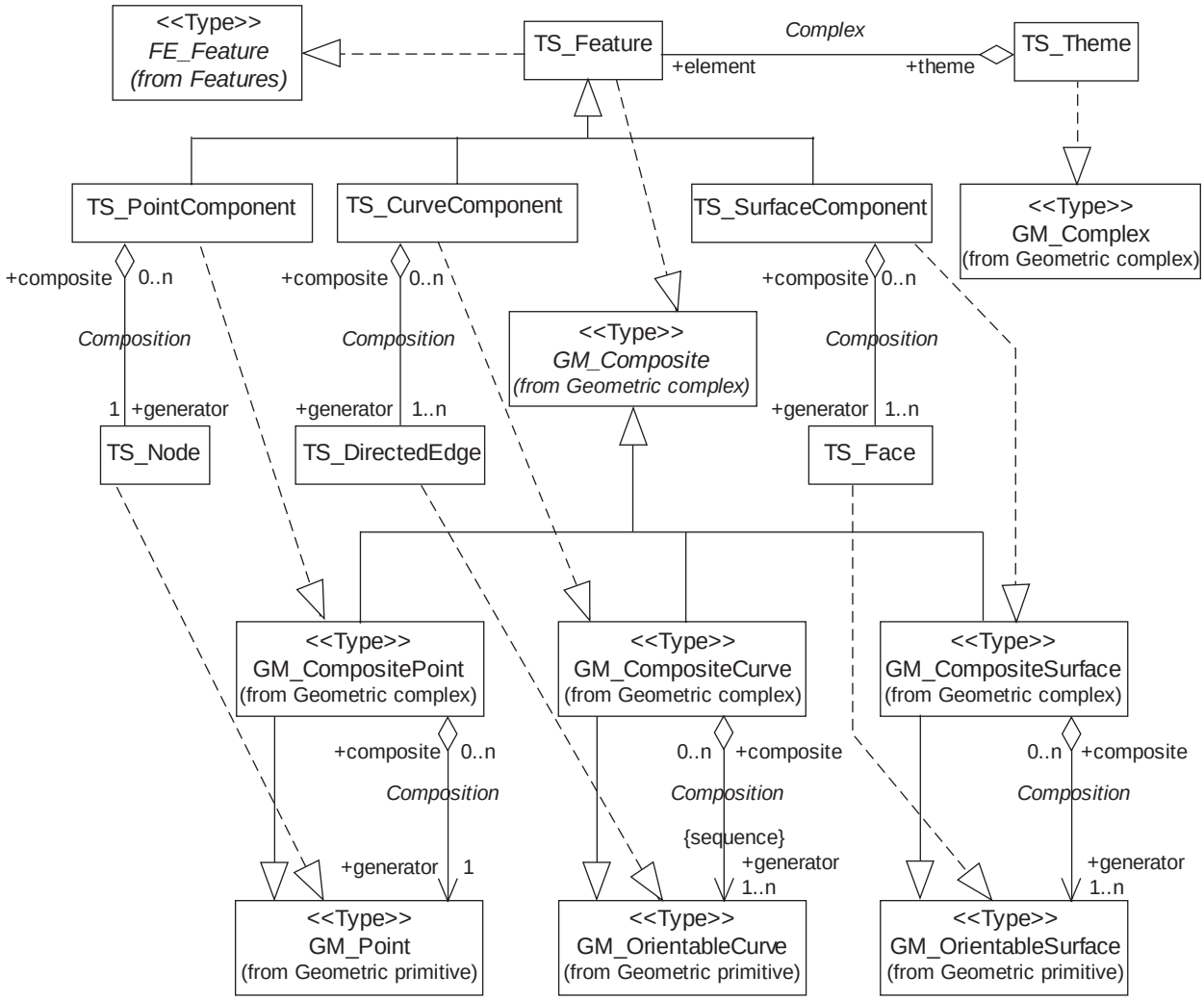


Figure D.3 — Feature components in simple topology

D.2.2.8 TS_Feature

TS_Feature acts as the root class for the feature class for this package. It inherits from the class Feature from the general feature model described in ISO 19109, allowing all of the feature objects in this package to take on attributes of any appropriate kind.

D.2.2.9 TS_PointComponent

TS_PointComponent multiply inherits from TS_Feature and GM_CompositePoint, allowing it to act as an independent geometry and a feature.

D.2.2.10 TS_CurveComponent

TS_CurveComponent multiply inherits from TS_Feature and GM_CompositeCurve, allowing it to act as an independent geometry and a feature.

D.2.2.11 TS_SurfaceComponent

TS_SurfaceComponent multiply inherits from TS_Feature and GM_CompositeSurface, allowing it to act as an independent geometry and a feature.

D.3 Feature Topology

D.3.1 Semantics

The basic concept behind this package is to allow composite geometric objects, here defined as feature components, to be organized into a topological structure independent of (but consistent with) the topological structure of their spatial attributes. Thus, within a theme (subclassed under TP_Complex and GM_Complex) the feature components can be related to one another based on topological structures identical to those used for the basic panthematic (all themes) geometric objects. This makes the assumption that feature components are broken at intersections with other feature objects within their theme.

D.3.2 Classes for feature topology at the theme level

D.3.2.1 FT_Complex

FT_Complex (Figure D.4) multiply inherits from TP_Complex and GM_Complex (through TS_Theme), allowing it to aggregate both topological and geometric information. The way this is structured allows each theme within a dataset to carry theme specific topological information. The simplifying assumption that each feature component is in one and only one theme can be lifted with a slightly more complex structure that maintains the dichotomy of topological and geometric objects.

D.3.2.2 FT_Primitive

FT_Primitive supports the same functions as TP_Primitive, and becomes the basic building block of the TP_Complex instantiated in FT_Complex.

D.3.2.3 FT_Node

An FT_Node is both a TP_Node and a TS_PointComponent. Thus, if needed, the point feature components within a theme play the role of the nodes within a feature topological complex.

D.3.2.4 FT_Edge

An FT_Edge is both a TP_Edge and a TS_CurveComponent. Thus, if needed, the curve feature components within a theme play the role of the edges within a feature topological complex.

D.3.2.5 FT_Face

An FT_Face is both a TP_Face and a TS_AreaComponent. Thus, if needed, the area feature components within a theme play the role of the faces within a feature topological complex.

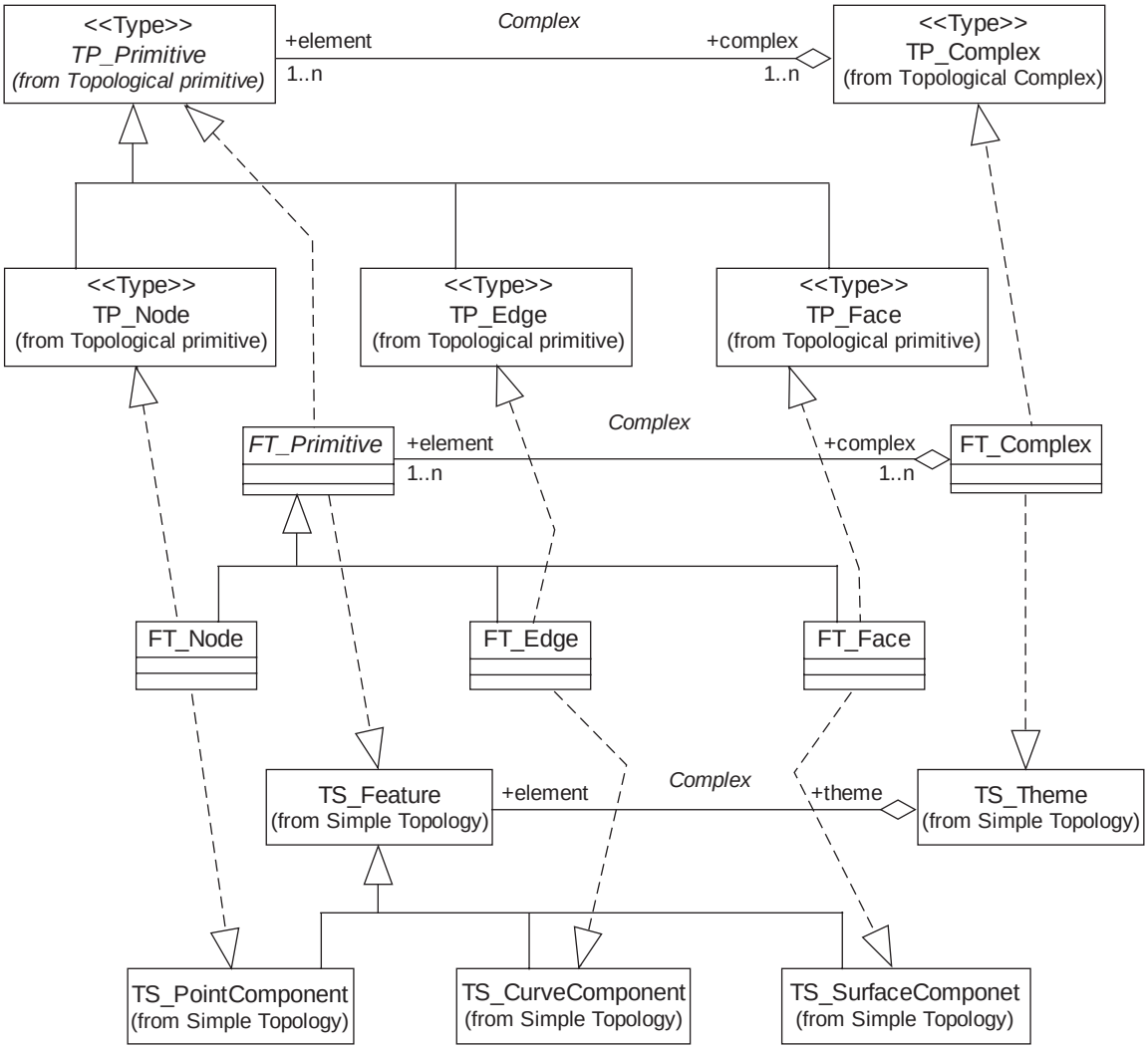


Figure D.4 — Theme based feature topology

D.4 MiniTopo

The MiniTopo Profile does not define any new classes, but simply makes restrictions on the use of existing classes. It specializes “Isolated In” association to “Within” that only shows nodes isolated in faces.

In DIGEST's underlying topology model, derived from an earlier model called MC&G or MiniTopo, most of the information concerning topological adjacency is carried by the pairs of directed edges (DE) associated to a face, Figure D.5. The corresponding information in this schema (Figure D.6) is carried by the structure of the boundary and coboundary operators/relations. Table D.1, below, relates the MiniTopo pointers to the appropriate information in the current model.

The MiniTopo record structure had nine basic types of records, four for features and four for geometry and topology, and an ancillary concept for references to these types, which is usually implemented by a record number. The MiniTopo topology-geometry record types were node, edge, directed edge, and face.

The record types for the geometry and topology had their structure defined as is given below (SQL 99 like syntax):

```
Create Node record as {
         nodeID : RecordIdentifier NOT NULL PrimaryKey,
         containingFace : RecordIdentifer ForeignKey to Face,
                  -- NULL for nodes connected to edges.
         position : CoordinatePoint NOT NULL }
Create Edge record as {
      edgeID : RecordIdentifier NOT NULL Primary Key,
         positiveDE : RecordIdentifer NOT NULL Foreign Key to DirectedEdge,
      negativeDE : RecordIdentifer NOT NULL Foreign Key to DirectedEdge,
      coordinatList : Variable Array Of CoordinatePoint NOT NULL }
Create DirectedEdge record as {
      directedEdgeID : RecordIdentifier NOT NULL Primary Key,
      nodeID : RecordIdentifer NOT NULL Foreign Key to Node,
         nextDE : RecordIdentifer NOT NULL Foreign Key to DirectedEdge,
      face : RecordIdentifer NOT NULL Foreign Key to Face }
Create Face record as {
         faceID : RecordIdentifier NOT NULL Primary Key  }
```

One of the primary advantages of this structure was the fixed size of each record, and its high level of normalization. The structure was considered to contain the minimal amount of redundancy – hence the name, minimally redundant topology.

Assuming that the edge coordinates were held as a reference to a graphics record, each of the MiniTopo objects was a fixed size. There were variants of this structure based on whether or not the reverse keys were given for the various relations. The original flat file exchange structure did not carry such reverse keys, since they would have entailed variable length records. Another variant was to combine the edge and directed edge records, which essentially gave a record with three semantically primary keys, usually written as edgeID, +edgeID and -edgeID.

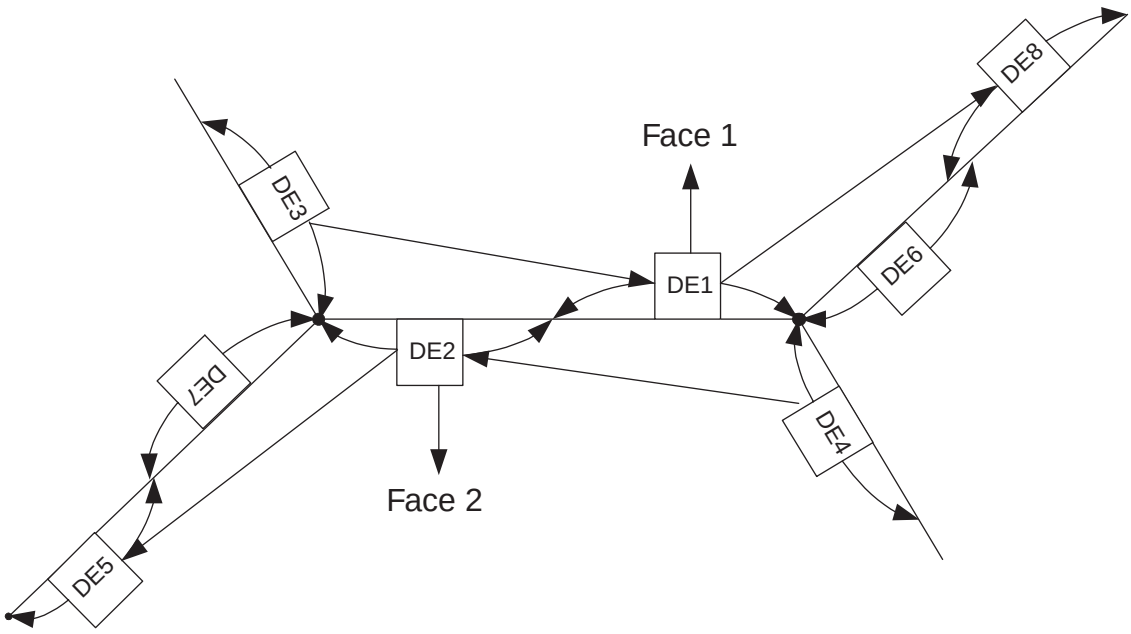


Figure D.5 — Geometric example of MiniTopo topology structure

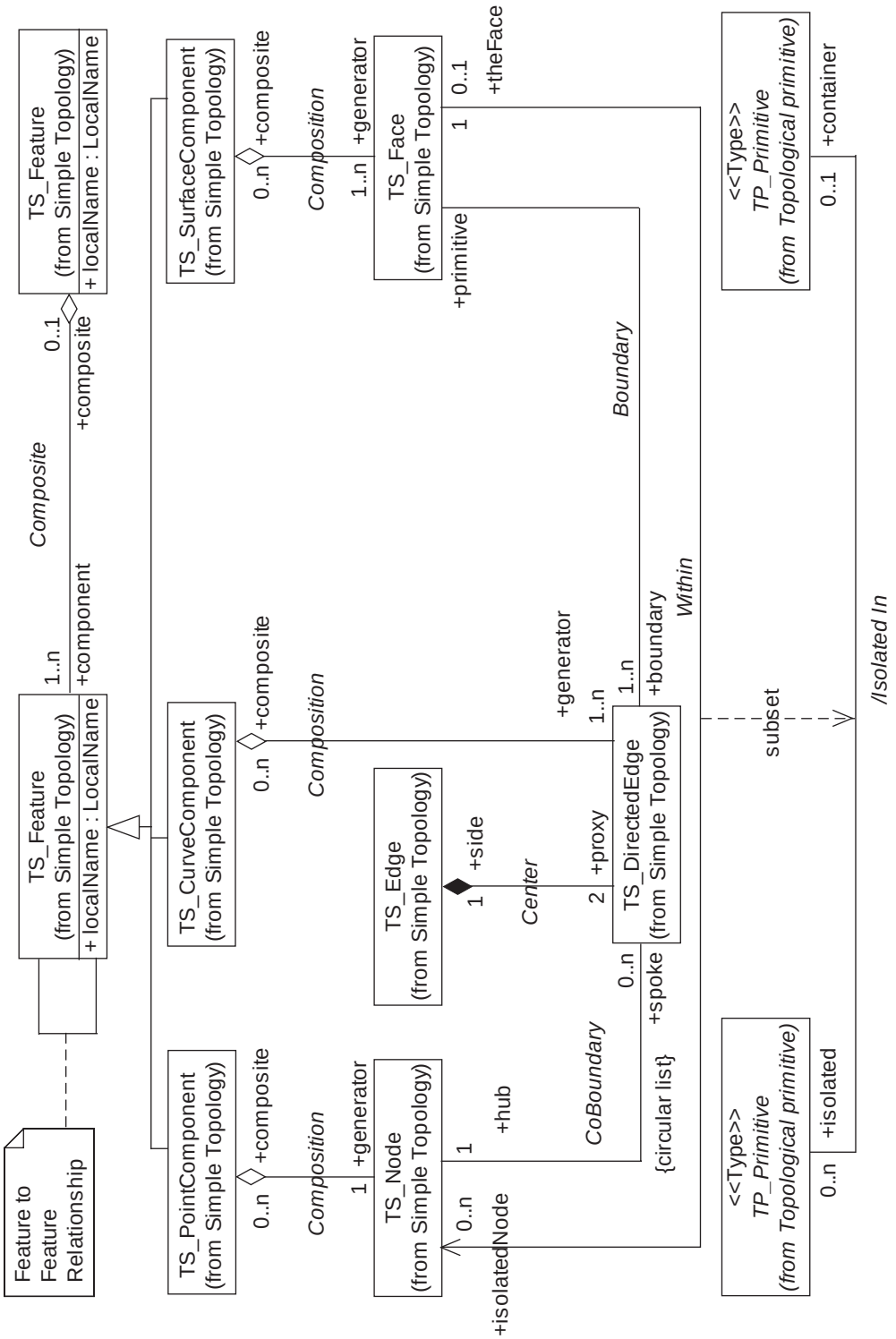


Figure D.6 — MiniTopo

In addition to the geometry/topology records, there were four types of feature records:

```
Create PointFeature as { -- essentially a multi point
         featureID : RecordIdentifier NOT NULL Primary Key,
         nodeID : Variable Array Of RecordIdentifier NOT NULL Foreign Key to
      Node,
      attribute : Variable Array of
                           <Name : CharacterString, Value : CharacterString> }
Create LineFeature as { -- essentially a composite curve
         featureID : RecordIdentifier NOT NULL Primary Key,
         directedEdgeID : Variable Array Of RecordIdentifier NOT NULL
                           Foreign Key to DirectedEdge,
      attribute : Variable Array of
                           <Name : CharacterString, Value : CharacterString> }
Create AreaFeature as { -- essentially a composite surface
         featureID : RecordIdentifier NOT NULL Primary Key,
         faceID : Variable Array Of RecordIdentifier NOT NULL   Foreign Key to
      Face,
      attribute : Variable Array of
                           <Name : CharacterString, Value : CharacterString> }
Create ComplexFeature as { -- essentially an aggregate feature
      featureID : RecordIdentifier NOT NULL Primary Key,
         pointComponentID : Variable Array Of RecordIdentifier
                           Foreign Key to PointFeature,
      lineComponentID : Variable Array Of RecordIdentifier
                           Foreign Key to LineFeature,
      areaComponentID : Variable Array Of RecordIdentifier
                           Foreign Key to AreaFeature,
         subfeatureID : Variable Array Of RecordIdentifier
                           Foreign Key to ComplexFeature,
         attribute : Variable Array of
                           <Name : CharacterString, Value : CharacterString> }
```

Variants of these records included the concept of a theme record, which was a type of complex feature that was not contained in any other feature, and a theme mask attribute which carried for each feature and each topology record information of which themes it was transitively a member.

```
Create Theme as { -- essentially a variant of complex feature
         featureID : RecordIdentifier NOT NULL Primary Key,
         pointComponentID : Variable Array Of RecordIdentifier
                        Foreign Key to PointFeature,
      lineComponentID : Variable Array Of RecordIdentifier
                        Foreign Key to LineFeature,
      areaComponentID : Variable Array Of RecordIdentifier
                        Foreign Key to AreaFeature,
      subfeatureID : Variable Array Of RecordIdentifier
                        Foreign Key to ComplexFeature,
      -- since theme is not a Complex feature, it could not be owned by any
      -- other ComplexFeature or Theme (all pointers are strongly typed)
      attribute : Variable Array of
                        <Name : CharacterString, Value : CharacterString>
      ThemeMask : Integer -- used as a bit mask, added to each
               -- type of ComplexFeature and Feature Component
               -- Size of theme mask usually limited number of themes to 32}
```

Figure D.7 was the standard record layout illustration.

The major difference between the current model and the MiniTopo model derives from their origins. MiniTopo was originally designed as an exchange structure, and did not have a rich object model due to the constraints

of sequential flat file structures. The conceptual level model behind the MiniTopo structure did agree precisely with the current model in terms of boundary structures between edges and faces and between edges and nodes. To save space and to expedite conversion from exchange structure to computational structure, MiniTopo (minimum topology) was introduced. MiniTopo used a dispersed linked list structure, illustrated in Figure D.5, using the directed edges to represent the internal structure of the boundary and coboundary structures represented explicitly in the current model. With the richer object modeling capabilities inherent in UML used for this current model, the original conceptual MiniTopo model corresponds more closely with the explicit object structure.

This illustrates one of the big advantages of a rich object modeling environment – narrowing of the “semantic gap”. The “semantic gap” is the informal term used to describe the differences between a conceptual model and an implementation model. Most of the “gap” is caused by the need to recast conceptual constructs into programming language constructs. Since a rich object model gives a much more robust vocabulary of language constructs, the “gap” can be narrowed. This comes with a cost tradeoff between algorithmic complexity and data structure complexity and size. In 1984, when MiniTopo was being designed, the cost tradeoffs favored smaller, more compact, data structures, from which the more robust conceptual model could be reconstructed, at a cost, in memory-based, object structures. Due to the restrictions surrounding the development of the MiniTopo structures, they were never fully documented by their original authors, and only the exchange structures, with their tradeoffs in place but not explained, were ever published in the open literature. Today, the tradeoffs have significantly changed and preservation of conceptual constructs in both computational and persistent storage models is favored. The costs have included more verbose and extensive persistent storage models, and some increase in the complexity of computational models. The benefits have been models that are more consistent with one another, narrowing the “semantic gap”, and a computational environment that is much closer to the logically consistent, and semantically rich, conceptual model. In effect, this model does not contradict the original MiniTopo conceptual model, but documents it and updates its implementation to a more modern object environment.

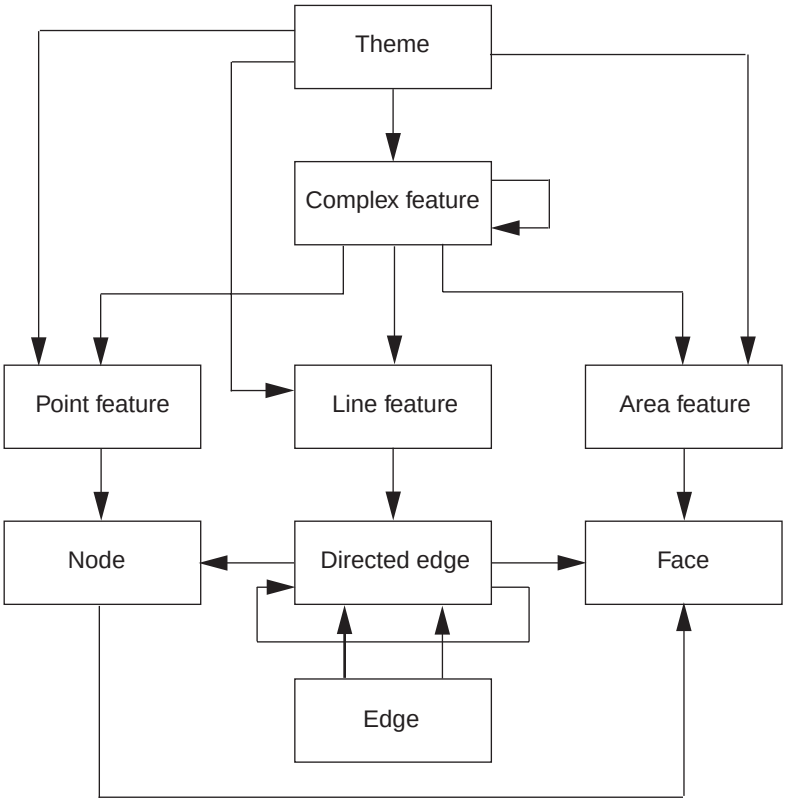


Figure D.7 — Classic MiniTopo record illustration

Table D.1 — Correspondence between original MiniTopo pointers and the current model

MiniTopo Pointer	Current Model	Description
Edge to Positive DE	TS_Edge to self in association inherited from TP_Edge, Center::side : TP_DirectedEdge	Each edge is its own positive directed edge.
Edge to Negative DE	TS_Edge to TS_DirectedEdge with negative orientation in association inherited from TP_Edge, Center::side : TP_DirectedEdge	Each edge is associated to one negatively oriented directed edge.
Positive DE to End Node	TS_Edge to TS_DirectedNode with positive orientation in association inherited from TP_Edge derived from boundary operator: /boundary::boundary : TP_DirectedNode	An edge has a boundary consisting of two directed nodes, one positive and one negative in orientation. The positive one corresponds to the end node and the negative one to the start node.
Negative DE to Start Node	TS_Edge to TS_DirectedNode with negative orientation in association inherited from TP_Edge derived from boundary operator: /boundary::boundary : TP_DirectedNode	
Positive DE to Left Face	TS_Edge to TS_DirectedFace with positive orientation in association inherited from TP_Edge derived from coBoundary operator, /coBoundary::spoke : TP_DirectedFace	An edge has a coboundary consisting of two directed faces, one positive and one negative in orientation. The positive one corresponds to the left face and the negative one to the right face.
Negative DE to Right Face	TS_Edge to TS_DirectedFace with positive orientation in association inherited from TP_Edge derived from coBoundary operator, /coBoundary::spoke : TP_DirectedFace	
DE to next DE (around Face)	Structure of role “boundary” from TS_Face to TS_DirectedEdge in association inherited from TP_Face derived from boundary operator, /boundary::boundary : Set<TP_Ring>	The boundary of a face is a set of rings. Each ring is a circular sequence of directed edges. The adjacent edges in this sequence are DE → (next DE) pairs from the original MiniTopo model.

Bibliography

NOTE Some definitions and additional information were taken from the following sources as indicated, with some edits for clarity or brevity in the particular context of this International Standard:

[1] ABADI, M. and CARDELLI, L. *A Theory of Objects*, Springer-Verlag, New York, 1996

[2] BARTELS, R.H., BEATTY J.C. and BARSKY, B.A. *An Introduction to Splines for Use in Computer Graphics & Geometric Modeling*, Morgan Kaufmann Publishers, Inc., 1987

[3] CLAPHAM, C., *Concise Dictionary of Mathematics*, Second Edition, Oxford University Press, 1996

[4] CLEMENTINI, E. and DI FELICE, P. *A Comparison of Methods for Representing Topological Relationships.* Information Sciences 80, 1-34, 1994

[5] CLEMENTINI, E. and DI FELICE, P. *A Model for Representing Topological Relationships Between Complex Geometric Features in Spatial Databases*. Information Sciences 90 (1-4):121-136 , 1996

[6] DAINTITH, J. and NELSON, R.D. *Dictionary of Mathematics*, Penguin Books, London, 1989

[7] *Dictionary of Computing*, Fourth Edition, Oxford University Press, 1996

[8] EGENHOFER, M.F. and FRANZOSA. *Point Set Topological Spatial Relations.* International Journal of Geographical Information Systems, vol 5, no 2, 161-174, 1991

[9] EGENHOFER, M.J., CLEMENTINI, E. and DI FELICE, P. *Topological relations between regions with holes.* International Journal of Geographical Information Systems, vol 8, no 2, pp 129—142, 1994

[10] FARIN, G. *Curves and Surfaces for Computer Aided Geometric Design, A Practical Guide*, Second Edition, Academic Press, Inc., Boston, 1990

[11] FARIN, G. *NURB Curves and Surfaces, From Projective Geometry to Practical Use*, A. K. Peters, Wellesley, Massachusetts, 1995

[12] FAUX, I.D. and PRATT, M.J. *Computational Geometry for Design and Manufacture*, Ellis Norwood Publishers, reprinted with corrections, 1981

[13] ISO/IEC 13249-3:1999, *Information technology — Database languages — SQL Multimedia and Application Packages — Part 3: Spatial*, ISO/IEC JTC1 SC 32, 1999

[14] ISO 19101:2002, *Geographic information — Reference model*

[15] ISO 19118:—[1)], *Geographic information — Encoding*

[16] KOSTOV, V. and DEGTIARIOVA-KOSTOVA, E. *Some properties of clothoids*. INRIA, Nice – Sophia Antipolis, Research Report N0 2752, December 1995

[17] OMG/UML, *Object Constraint Language Specification, version 1.3*, 1997. Available at <http://www.rational.com/uml/resources/documentation/formats.jsp>

[18] OMG/UML, *UML Notation Guide, version 1.3*, 1997. Available at <http://www.rational.com/uml/resources/documentation/formats.jsp>

1) To be published.

[19] OMG/UML, *UML Semantics, version 1.3*, 1997. Available at <http://www.rational.com/uml/resources/documentation/formats.jsp>

[20] PRENTER, P. M. *Splines and Variational Methods*, John Wiley & Sons, New York, 1975, republished in Wiley Classics Edition, 1989

[21] ROGERS, D.F. and ADAMS, J.A. *Mathematical Elements for Computer Graphics*, Second Edition, McGraw Hill Publishing Company, 1990

[22] *The OpenGIS Abstract Specification*, Volumes 1-14, OpenGIS™ Consortium, 1997

[23] ISO/TS 19103:—[1)], *Geographic information — Conceptual schema language*

ICS 35.240.70

Price based on 166 pages