**i. Abstract**

This stand defines the open CityGML conceptual model for the storage and exchange of virtual 3D city models. The CityGML conceptual model is defined through a Unified Modeling Language (UML) object model. This UML model builds on the ISO Technical Committee 211 (ISO/TC211) conceptual model standards for spatial and temporal data. Building on the ISO foundation assures that the man-made features described in the city models share the same spatiotemporal universe as the surrounding countryside within which they reside.

A key goal for the development of the CityGML model is to provide a common definition of the basic entities, attributes, and relations of a 3D city model. This is especially important with respect to the cost-effective sustainable maintenance of 3D city models, allowing for the reuse of the same data in different application fields.

<<SNIP, SNIP>>

**1. Introduction**

An increasing number of cities and companies are building virtual 3D city models for different application areas like urban planning, mobile telecommunication, disaster management, 3D cadastre, tourism, vehicle and pedestrian navigation, facility management and environmental simulations. Furthermore, in the implementation of the European Environmental Noise Directive (END, 2002/49/EC) 3D geoinformation and 3D city models play an important role.

In recent years, most virtual 3D city models were defined as purely graphical or geometrical models, neglecting the semantic and topological aspects. Thus, these models could almost only be used for visualization purposes but not for thematic queries, analysis tasks, or spatial data mining. Since the limited reusability of models inhibits the broader use of 3D city models and may not justify the costs associated with maintaining city models, a more general modelling approach had to be taken in order to satisfy the information needs of the various application fields.

The CityGML standard defines a common semantic information model for the representation of 3D urban objects that can be shared over different applications. The latter capability is especially important with respect to the cost-effective sustainable maintenance of 3D city models, allowing governments and companies to reap the benefits of their investment in 3D city models by being able to put the same models into play in different application fields. The targeted application areas explicitly include city planning, architectural design, tourist and leisure activities, environmental simulation, mobile telecommunication, disaster management, homeland security, real estate management, vehicle and pedestrian navigation, and training simulators.

CityGML is an open conceptual model for the storage and exchange of virtual 3D city models. It is defined through a Unified Modeling Language (UML) object model. This UML model builds on the ISO Technical Committee 211 (ISO/TC211) conceptual model standards for spatial and temporal data. Building on the ISO foundation assures that the man-made features described in the city models share the same spatio-temporal universe as the surrounding countryside within which they reside.

The CityGML model defines the classes and relations for the most relevant topographic objects in cities and regional models with respect to their geometrical, topological, semantical, and appearance properties. “City” is broadly defined to comprise not just built structures, but also elevation, vegetation, water bodies, city furniture, and more. Included are generalization hierarchies between thematic classes, aggregations, relations between objects, and spatial properties. CityGML is applicable for large areas and small regions, and can represent the terrain and 3D objects in different levels of detail simultaneously. Since both simple, single scale models without topology and few semantics as well as very complex multi-scale models with full topology and fine-grained semantical differentiations can be represented, CityGML enables lossless information exchange between different geographic information systems and users.

**2. Scope**

This document is an OGC Conceptual Model (CM) Standard for specifying the representation of virtual 3D city and landscape models. The CityGML 3.0 Conceptual Model is expected to be the basis for a number of future Encoding Standards in which subsets of the Conceptual Model can be implemented. These Encoding Standards will enable both storage and exchange of data. At a minimum, support for a Geography Markup Language (GML) encoding is expected, though additional encodings in formats such as JSON and database schemas will be highly desirable.

In contrast to CityGML 2.0, the 3.0 CM is NOT implemented as an application schema of GML. However, a preliminary encoding in GML version 3.2 is available as an informative resource separately from this standard to allow implementation and testing of the CityGML CM.

The target of the conformance classes specified in this document are:

* CityGML encoding standards that provide encodings for the UML conceptual model specified in this document, and
* Additional UML models that can be created by users to extend this CM as Application Domain Extensions (ADEs).

CityGML models are comprised of georeferenced 3D vector data along with the semantics associated with the data. In contrast to other 3D vector formats, CityGML is based on a rich, general purpose information model in addition to geometry and appearance information that allows for the integration of a variety of source data to come together in a City Model. To enable the use of CityGML in specific domain areas, CityGML has historically provided an extension mechanism to enrich the data with identifiable features and properties, preserving semantic interoperability. Recognizing that an implementable expansion mechanism might have dependencies based on the encoding language, the CityGML 3.0 Conceptual Model specifies high level requirements rather than a full extension model.

Targeted application areas explicitly include:

* Urban and landscape planning;
* Architectural design;
* Tourist and leisure activities;
* Environmental, energy and mobility simulations;
* Mobile telecommunications; disaster management;
* Homeland security;
* Vehicle and pedestrian navigation;
* Training simulators and mobile robotics.

The future CityGML 3.0 encoding standards will be implementable source formats for 3D portraying or transformation into dedicated portrayal formats such as the OGC I3S or the OGC 3D Tiles Community Standards, OGC KML, COLLADA or glTF. The OGC 3D Portrayal Service (3DPS) may be used for content delivery.

Features of the CityGML 3.0 Conceptual Model:

* Geospatial Information Model (ontology) for urban landscapes based on the ISO 19100 family.
* Representation of 3D geometries, based on the ISO 19107 model, independent of data encodings, as well as of 3D point clouds.
* Grouping into space hierarchies, including concepts like stories/floors within buildings.
* Representation of object surface characteristics (e.g. textures, materials).
* Representation of dynamic, i.e. time-dependent, properties of city models.
* Taxonomies and aggregations including:
  + Digital Terrain Models as a combination of triangulated irregular networks (TINs), regular grids, break and skeleton lines, mass points.
  + Sites (currently buildings, other constructions, bridges, and tunnels).
  + Vegetation (areas, volumes, and solitary objects with vegetation classification).
  + Water bodies (volumes, surfaces).
  + Transportation facilities (graph structures, 3D space, and 3D surface data).
  + Land use (representation of areas of the earth’s surface dedicated to a specific land use).
  + City furniture.
  + Generic city objects and attributes.
  + User-definable (recursive) grouping.
* Multiscale model with 4 well-defined consecutive Levels of Detail (LOD), applicable to both interior and exterior:
  + LOD0 – Highly generalized model;
  + LOD1 – Block model / extrusion objects;
  + LOD2 – Realistic, but still generalized model;
  + LOD3 – Highly detailed model.
* Multiple representations in different LODs simultaneously and generalization relations between objects in different LODs
* Ability to combine different interior and exterior LoDs, including representation of floor plans.
* Optional topological connections between feature (sub)geometries.
* Enables a variety of different encoding specifications, including GML and JSON.
* Extension of the CM through code lists, generic objects and Application Domain Extensions (ADEs).
* With CityGML 3.0, ADEs become platform-independent models on a conceptual level that can be mapped to multiple and different target encodings. ADEs are implemented as UML models that extend the CM in this standard. This includes a mechanism that favors the insertion of additional feature properties into any defined feature class through 'hooks' over subtyping of features. This means that the existing feature classes can be used and additional properties from one or more ADEs can easily be supported in different encodings.
* Ability to specify an ADE that can be further extended.

<<SNIP, SNIP>>

**3.1. Conceptual Models**

A Conceptual Model standardization target is a version of the CityGML 3.0 Conceptual Model (CM) tailored for a specific user community. This tailoring can include:

1. Omission of one or more of the optional packages;
2. Reduction of the multiplicity for an attribute or association;
3. Restriction on the valid values for an attribute;
4. Additional concepts documented through ADEs.

Of these options, actions #1, #2, and #3 can be performed when creating an implementation standard. Only action #4 requires an extension of the CityGML CM. These extensions SHALL be accomplished using the ADE mechanism described in [Section 10](http://docs.ogc.org/DRAFTS/20-010.html#rc_ade_section) Application Domain Extensions (ADE).

Extensions of the CityGML CM SHALL conform with the ADE Conformance Class.

**3.2. Implementation Standards**

Implementation Standard define how a Conceptual Model shall be implemented using a specific technology. Conformant Implementation Standards provide evidence that they are an accurate representation of the Conceptual Model. This evidence shall include implementations of the abstract tests specified in [Annex A](http://docs.ogc.org/DRAFTS/20-010.html#abstract-test-suite) (normative) of this document.

Since this standard is agnostic to the implementing technologies, the specific techniques to be used for conformance testing cannot be specified. Implementation Standards need to provide evidence of conformance which is appropriate for the implementing technologies. This evidence should be provided as an annex to the Implementation Standard document.

**3.3. Conformance Classes**

This standard identifies seventeen (17) conformance classes. One conformance class is defined for each Package in the UML model. Each conformance class is defined by one requirements class. The tests in [Annex A](http://docs.ogc.org/DRAFTS/20-010.html#abstract-test-suite) are organized by Requirements Class. So an implementation of the *Core* conformance class must pass all tests specified in Annex A for the *Core* requirements class.

Of the seventeen conformance classes, only the *Core* conformance class is mandatory. All other conformance classes are optional. In the case where a conformance class has a dependency on another conformance class, that conformance class shall also be implemented.

The CityGML Conceptual Model is documented in the CityGML UML model. This standard is a representation of that UML model in document form. In the case of a discrepancy between the UML model and this document, the UML model is authoritative.

<<SNIP, SNIP>>

**6. Conventions**

**6.1. Identifiers**

The normative provisions in this document are denoted by the URI:

<http://www.opengis.net/spec/CityGML-1/3.0>

All requirements and conformance tests that appear in this document are denoted by partial URIs relative to this base.

**6.2. UML Notation**

The CityGML Conceptual Model (CM) standard is presented in this document in diagrams using the Unified Modeling Language (UML) static structure diagram (see Booch et al. 1997). The UML notations used in this standard are described in the diagram in [Figure 1](http://docs.ogc.org/DRAFTS/20-010.html#figure-1).

Figure 1. UML notation (see ISO TS 19103, Geographic information - Conceptual schema language).

All associations between model elements in the CityGML CM are unidirectional. Thus, associations in CM are navigable in only one direction. The direction of navigation is depicted by an arrowhead. In general, the context an element takes within the association is indicated by its role. The role is displayed near the target of the association. If the graphical representation is ambiguous though, the position of the role has to be drawn to the element the association points to.

The following stereotypes are used in this model:

* «ApplicationSchema» denotes a conceptual schema for data required by one or more applications. In the CM, every module is defined as a separate application schema to allow for modularization.
* «FeatureType» represents features that are similar and exhibit common characteristics. Features are abstractions of real-world phenomena and have an identity.
* «TopLevelFeatureType» denotes features that represent the main components of the CM. Top-level features may be further semantically and spatially decomposed and sub-structured into parts.
* «Type» denotes classes that are not directly instantiable, but are used as an abstract collection of operation, attribute and relation signatures. The stereotype is used in the CM only for classes that are imported from the ISO standards 19107, 19109, 19111, and 19123.
* «ObjectType» represents objects that have an identity, but are not features.
* «DataType» defines a set of properties that lack identity. A data type is a classifier with no operations, whose primary purpose is to hold information.
* «Enumeration» enumerates the valid attribute values in a fixed list of named literal values. Enumerations are specified in the CityGML CM schema.
* «BasicType» defines a basic data type.
* «CodeList» enumerates the valid attribute values. In contrast to Enumeration, the list of values is open and, thus, not given inline in the CityGML UML Model. The allowed values can be provided within an external code list.
* «Union» is a list of attributes. The semantics are that only one of the attributes can be present at any time.
* «Property» denotes attributes and association roles. This stereotype does not add further semantics to the CM, but is required to be able to add tagged values to the attributes and association roles that are relevant for the encoding.
* «Version» denotes that the value of an association role that ends at a feature type is a specific version of the feature, not the feature in general.

In order to enhance the readability of the CityGML UML diagrams, classes are depicted in different colors. The following coloring scheme is applied:

Classes painted in yellow belong to the Requirements Class which is subject of discussion in that clause of the standard in which the UML diagram is given. For example, in the context of [Section 8.2](http://docs.ogc.org/DRAFTS/20-010.html#rc_core_section), which introduces the *CityGML Core* module, the yellow color is used to denote classes that are defined in the *CityGML Core* Requirements Class. Likewise, the yellow classes shown in the UML diagram in [Section 8.17](http://docs.ogc.org/DRAFTS/20-010.html#rc_building-model_section) are associated with the *Building* Requirements Class that is subject of discussion in that chapter.

Classes painted in blue belong to a Requirements Class different to that associated with the yellow color. In order to explicitly denote to which Requirements Class these classes belong, their class names are preceded by the UML package name of that Requirements Class. For example, in the context of the *Building* Requirements Class, classes from the *CityGML Core* and the *Construction* Requirements Classes are painted in blue and their class names are preceded by *Core* and *Construction*, respectively.

Classes painted in green are defined in the ISO standards 19107, 19111, or 19123. Their class names are preceded by the UML package name, in which the classes are defined.

Classes painted in grey are defined in the ISO standard 19109. In the context of this standard, this only applies to the class *AnyFeature*. *AnyFeature* is an instance of the metaclass *FeatureType* and acts as super class of all classes in the CityGML UML model with the stereotype «FeatureType». A **metaclass** is a class whose instances are classes.

The color white is used for notes and OCL constraints that are provided in the UML diagrams.

The example UML diagram in [Figure 2](http://docs.ogc.org/DRAFTS/20-010.html#figure-2) demonstrates the UML notation and coloring scheme used throughout this standard. In this example, the yellow classes are associated with the *CityGML Building* module, the blue classes are from the *CityGML Core* and *Construction* modules, and the green class depicts a geometry element defined by ISO 19107.

Figure 2. Example UML diagram demonstrating the UML notation and coloring scheme used throughout the CityGML standard.

**6.3. Conceptual Modelling**

[ISO 19101](http://docs.ogc.org/DRAFTS/20-010.html" \l "iso19101) defines universe of discourse to be a view of the real or hypothetical world that includes everything of interest. ISO 19101 then defines conceptual model to be a model that defines concepts of a universe of discourse.

The CityGML Conceptual Model Standard establishes the limits of the universe of discourse for this Standard. The next task is to discover and standardize the concepts within this scope. CityGML will potentially support numerous diverse application software packages covering multiple disciplines and facility life cycle phases. Each conceivably can have its own universe of discourse and set of concepts.

The goal of the CityGML Conceptual Model is to establish and document a common set of concepts that spans the applications supported. This does not attempt to redefine application concepts, but merely present a common set of concepts from and to which their concepts can be understood and mapped.

GML and JSON encodings are planned and other encodings are anticipated. Each encoding addresses a specific information community and set of application software packages. However, with the increasing desire to share information between communities and applications having a common conceptual model across all of these encodings is highly advantageous.

An added benefit of the development of a conceptual model results from the rigor involved in achieving consensus. After numerous iterations, the end result is consistent, cohesive, and complete. Updating a conceptual model is far easier than rewriting software code. Further, the iterations help to flesh out details as well as to unearth differences in individual conceptualizations.

Perhaps the greatest benefit of the standards activity is the ability to communicate the resultant model. This is in part due to using a standardized conceptual modelling language like UML and the agreed OGC and ISO/TC211 conventions for using UML. The eventual outcome of being able to provide formal documentation for what is meant by each concept is invaluable in understanding the subsequent encodings and applications.

This will be the first OGC conceptual model standard without accompanying encodings. Yet the model is presented in a manner consistent with the formalisms adopted for writing OGC standards. This standard follows the [OGC Specification Model standard for modular specifications](http://docs.ogc.org/DRAFTS/20-010.html#ogc08-131) and is consistent with the OGC Naming Authority conventions and recommendations. The target of this Standard are the encoding standards which will follow and not the application software that will implement these encodings. Requirements for the encodings are explicit and grouped into Requirements Classes. Accompanying Conformance Classes are included to determine if an encoding conforms to the conceptual model.

UML has been used as the conceptual modelling language in this Standard. Class Diagrams have been created and inserted as Figures. The boxes in these diagrams (officially “Classifiers” in UML) typically represent classes, data types, enumerations, code lists, unions, etc. and this terminology is used throughout the Standard. However, since this is a Conceptual Model, these should all be interpreted to be “concepts”. For each Requirements Class, an introductory diagram is included which contains all of the concepts relevant to that Requirements Class.

Though redundant with the UML diagrams, all of the classes, class attributes, and associations are repeated in the Data Dictionary in [Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section). If these differ, the UML takes precedence.

**7. Overview of CityGML**

This standard defines a open CityGML conceptual model for the storage and exchange of virtual 3D city and landscape models. This document defines the conceptual schema for the most relevant entities of the urban space like buildings, roads, railways, tunnels, bridges, city furniture, water bodies, vegetation, and the terrain. The conceptual schema specifies how and into which parts and pieces physical objects of the real world should be decomposed and classified. All objects can be represented with respect to their semantics, 3D geometry, 3D topology, appearances, and their changes over time. Different spatial representations can be provided for each object (outdoor and indoor) in four predefined Levels of Detail (LOD 0-3). The CityGML 3.0 Conceptual Model (CM) ([Chapter 8](http://docs.ogc.org/DRAFTS/20-010.html#conceptual-model-section)) is formally specified using UML class diagrams, complemented by a data dictionary ([Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section)) providing the definitions and explanations of the object classes and attributes. This CM is the basis for multiple encoding standards, which map the concepts (or subsets thereof) onto exchange formats or database structures for data exchange and storage.

While the CityGML CM can be used for 3D visualization purposes, its special merits lie in applications that go beyond visualization such as decision support, urban and landscape planning, urban facility management, Smart Cities, navigation (both indoor and outdoor), Building Information Modelling (especially for as-built documentation), integration of city and BIM models, assisted and autonomous driving, and simulations in general (cf. [Kolbe 2009](http://docs.ogc.org/DRAFTS/20-010.html#Kolbe2009)). A comprehensive overview on the many different applications of virtual 3D city models is given in [[Biljecki et al. 2015](http://docs.ogc.org/DRAFTS/20-010.html#Biljecki2015)]. Many of the applications already use and some even require using CityGML.

In the CityGML CM, all 3D city objects can easily be enriched with thematic data. For example, street objects can be enriched with information about traffic density, speed limit, number of lanes etc., or buildings can be enriched by information on the heating and electrical energy demand, numbers of households and inhabitants, the appraised building value etc. Even building parts such as individual roof or wall surfaces can be enriched with information e.g. about solar irradiation and thermal insulation parameters. For many application domains specific extensions of the CityGML CM have already been created (cf. [Biljecki et al. 2018](http://docs.ogc.org/DRAFTS/20-010.html#Biljecki2018)).

**7.1. Modularisation**

The CityGML CM provides models for the most important types of objects within virtual 3D city and landscape models. These feature types have been identified to be either required or important in many different application areas. However, implementations are not required to support the complete CityGML CM model in order to be conformant to the standard. Implementations may employ a subset of constructs according to their specific information needs. For this purpose, modularization is applied to the CityGML CM.

Figure 3. CityGML 3.0 module overview. The vertical boxes show the different thematic modules. Horizontal modules specify concepts that are applicable to all thematic modules.

The CityGML CM is thematically decomposed into a *Core module* and different kinds of *extension modules* as shown in [Figure 3](http://docs.ogc.org/DRAFTS/20-010.html#figure-moduleoverview). The Core module (shown in green) comprises the basic concepts and components of the CityGML CM and, thus, must be implemented by any conformant system. Each red colored module covers a specific thematic field of virtual 3D city models.

The CityGML CM introduces the following eleven thematic extension modules: *Building*, *Bridge*, *Tunnel*, *Construction*, *CityFurniture*, *CityObjectGroup*, *LandUse*, *Relief*, *Transportation*, *Vegetation*, and *WaterBody*. All three modules *Building*, *Bridge*, and *Tunnel* model civil structures and share common concepts that are grouped within the *Construction* module. The five blue colored extension modules add specific modelling aspects that can be used in conjunction with all thematic modules:

* The *Appearance* module contains the concepts to represent appearances (like textures and colours) of city objects.
* The *PointCloud* module provides concepts to represent the geometry of city objects by 3D point clouds.
* The *Generics* module defines the concepts for generic objects, attributes, and relationships.
* *Versioning* adds concepts for the representation of concurrent versions, real world object histories and feature histories.
* The *Dynamizer* module contains the concepts to represent city object properties by time series data and to link them with sensors, sensor data services or external files.

Each CityGML encoding can specify supporting a subset of the CityGML modules. If a module is supported by an encoding, all concepts shall be mapped. However, the encoding specification can define so-called *null mappings* to restrict the use of specific elements of the conceptual model in an encoding. Null mappings can be expressed in an encoding specification for individual feature types, properties, and associations defined within a CityGML module. This means that the corresponding element will not be included in the respective encoding.

Note that also CityGML applications do not have to support all modules. Applications can also decide to only support a specific subset of CityGML modules. For example, when an application only has to work with building data, only the modules *Core*, *Construction*, and *Building* would have to be supported.

**7.2. General Modelling Principles**

**7.2.1. Semantic Modelling of Real-World Objects**

Real-world objects are represented by geographic features according to the definition in ISO 19109. Geographic features of the same type (e.g. buildings, roads) are modelled by corresponding feature types that are represented as classes in the Conceptual Model (CM). The objects within a 3D city model are instances of the different feature types.

In order to distinguish and reference individual objects, each object has unique identifiers. In the CityGML 3.0 CM each geographic feature has the mandatory *featureID* and an optional *identifier* property. The *featureID* is used to distinguish all objects and their possible multiple versions of the same real-world object. The *identifier* is identical for all versions of the same real-world object and can be used to reference specific objects independent from their actual object version. The *featureID* must at least be unique within the same CityGML dataset, but it is generally recommended to use globally unique identifiers like UUID values or identifiers maintained by an organization such as a mapping agency. Providing globally unique and stable identifiers for the *identifier* attribute is recommended. This means these identifiers should remain stable over the lifetime of the real-world object.

CityGML feature types typically have a number of spatial and non-spatial properties (also called attributes) as well as relationships with other feature or object types. Note that a single CityGML object can have different spatial representations at the same time: For example, different geometry objects representing the feature’s geometry in different levels of detail or as different spatial abstractions.

Many attributes have simple, scalar values like a number or a character string. However, some attributes are complex, i.e. they do not just have a single property value. In CityGML the following types of complex attributes occur:

* *qualified attribute values*: For example, a measure consists of the value and a reference to the unit of measure, or e.g. for relative and absolute height levels the reference level has to also be named.
* *code list values* for enumerative attributes: In addition to the value a link to the code list definition should be provided.
* attributes consisting of a *tuple of different fields and values* – e.g. addresses, space occupancy, and others.
* Attribute value consisting of a *list of numbers*. For example, representing coordinate lists or matrices.

In order to support history, CityGML 3.0 introduces bi-temporal timestamps for all objects. In CityGML 2.0 the attributes *creationDate* and *terminationDate* are supported. These refer to the time period in which a specific version of an object is an integral part of the 3D city model. In 3.0, all objects can now additionally have the attributes *validFrom* and *validTo*. These represent the lifespan a specific version of an object has in the real-world. Using these two time intervals a CityGML dataset could be queried both for how did the *city* look like at a specific point in time as well as how did the *city model* look at that time.

The combination of the two types of feature identifiers and bi-temporal timestamps enables encoding not only the current version of a 3D city model, but also the model’s entire history can be represented in CityGML and possibly exchanged within a single file.

**7.2.2. Class Hierarchy and Inheritance of Properties and Relations**

In CityGML, the specific feature types like *Building*, *Tunnel*, or *WaterBody* are defined as subclasses of more general higher-level classes. Hence, feature types build a hierarchy along specialization / generalization relationships where more specialized feature types inherit the properties and relationships of all their superclasses along the entire generalization path to the topmost feature type *AnyFeature*. NOTE: A **superclass** is the class from which many subclasses can be created.

**7.2.3. Relationships between CityGML objects**

In CityGML, objects can be related to each other and different kinds of relations are distinguished. First of all, complex objects like buildings or transportation objects typically consist of parts. These parts are individual features of their own, and can even be further decomposed. Therefore, CityGML objects can form aggregation hierarchies. Some feature types are marked in the conceptual model with the stereotype *«TopLevelFeatureType»*. These constitute the main objects of a city model and are typically the root of an aggregation hierarchy. Only top-level features are allowed as direct members of a *CityModel* object. The information about which feature types belong to the top level is required for software packages that want to filter imports, exports, and visualizations according to the general type of a city object (e.g. only show buildings, solitary vegetation objects, and roads). CityGML Application Domain Extensions should also make use of this concept, such that software tools can learn from inspecting their conceptual schema what are the main, i.e. the top-level, feature types of the extension.

Some relations in CityGML are qualified by additional parameters, typically to further specify the type of relationship. For example, a relationship can be qualified with a URI pointing to a definition of the respective relation type in an Ontology. Qualified relationships are used in CityGML, among others, for:

* General relationships between features – association *relatedTo* between city objects;
* User-defined aggregations using *CityObjectGroup*. This relation allows also for recursive aggregations;
* External references – linking of city objects with corresponding entities from external resources like objects in a cadastre or within a BIM dataset.

The CityGML CM contains many relationships that are specifically defined between certain feature types. For example, there is the *boundary* relationship from 3D volumetric objects to its thematically differentiated 3D boundary surfaces. Another example is the *generalizesTo* relation between feature instances that represent objects on different generalisation levels.

In the CityGML 3.0 CM there are new associations to express topologic, geometric, and semantic relations between all kinds of city objects. For example, information that two rooms are adjacent or that one interior building installation (like a curtain rail) is overlapping with the spaces of two connected rooms can be expressed. The CM also enables documenting that two wall surfaces are parallel and two others are orthogonal. Also distances between objects canbe represented explicitly using geometric relations. In addition to spatial relations, logical relations can be expressed.

**7.2.4. Definition of the Semantics for all Classes, Properties, and Relations**

The meanings of all elements defined in the CityGML conceptual model are normatively specified in the data dictionary in [Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section).

**7.3. Representation of Spatial Properties**

**7.3.1. Geometry and Topology**

Spatial properties of all CityGML feature types are represented using the geometry classes defined in ISO 19107. Spatial representations can have 0-, 1-, 2-, or 3-dimensional extents depending on the respective feature type and Levels of Detail (LOD; the LOD concept is discussed in [Section 7.4.4](http://docs.ogc.org/DRAFTS/20-010.html#overview-section-levelsofdetail) and [Section 8.2.5](http://docs.ogc.org/DRAFTS/20-010.html#geometry-lod-section)). With only a few exceptions, all geometries must use 3D coordinate values. Besides primitive geometries like single points, curves, surfaces, and solids, CityGML makes use of different kinds of aggregations of geometries like spatial aggregates (*MultiPoint*, *MultiCurve*, *MultiSurface*, *MultiSolid*) and composites (*CompositeCurve*, *CompositeSurface*, *CompositeSolid*). Volumetric shapes are represented in ISO 19107 according to the so-called *Boundary Representation* (B-Rep, for explanation see [Foley et al. 2002](http://docs.ogc.org/DRAFTS/20-010.html#Foley2002)) only.

The CityGML CM does not put any restriction on the usage of specific geometry types as defined in ISO 19107. For example, 3D surfaces could be represented in a dataset using 3D polygons or 3D meshes such as triangulated irregular networks (TINS) or by non-uniform rational B-spline surfaces (NURBS). However, an encoding may restrict the usage of geometry types. For example, curved lines like B-splines or clothoids, or curved surfaces like NURBS could be disallowed by explicitly defining *null encodings* for these concepts in the encoding specification (c.f. [Section 7.1](http://docs.ogc.org/DRAFTS/20-010.html#overview-section-modularisation) above).

Note that the conceptual schema of ISO 19107 allows composite geometries to be defined by a recursive aggregation for every primitive type of the corresponding dimension. This aggregation schema allows the definition of nested aggregations (hierarchy of components). For example, a building geometry (*CompositeSolid*) can be composed of the house geometry (*CompositeSolid*) and the garage geometry (*Solid*), while the house’s geometry is further decomposed into the roof geometry (*Solid*) and the geometry of the house body (*Solid*). This is illustrated in [Figure 4](http://docs.ogc.org/DRAFTS/20-010.html#figure-recursiveaggregation).

Figure 4. Recursive aggregation of objects and geometries in CityGML (graphic: IGG Uni Bonn).

While the CityGML CM does not employ the topology classes from ISO 19107, topological relations between geometries can be established by sharing geometries (typically parts of the boundary) between different geometric objects. One part of real-world space can be represented only once by a geometry object and is referenced by all features or more complex geometries which are defined or bounded by this geometry object. Thus redundancy can be avoided and explicit topological relations between parts are maintained.

Basically, there are three cases for sharing geometries:

* First, two different semantic objects may be spatially represented by the same geometry object. For example, if a foot path is both a transportation feature and a vegetation feature, the surface geometry defining the path is referenced by both the transportation object and by the vegetation object.
* Second, a geometry object may be shared between a feature and another geometry. For example, a geometry defining a wall of a building may be referenced twice: By the solid geometry defining the geometry of the building, and by the wall feature.
* Third, two geometries may reference the same geometry, which is in the boundary of both. For example, a building and an adjacent garage may be represented by two solids. The surface describing the area where both solids touch may be represented only once and it is referenced by both solids. As it can be seen from [Figure 4](http://docs.ogc.org/DRAFTS/20-010.html" \l "figure-recursiveaggregation), this requires partitioning of the respective surfaces.

In general, B-Rep only considers visible surfaces. However, to make topological adjacency explicit and to allow the possibility of deletion of one part of a composed object without leaving holes in the remaining aggregate, touching elements are included. Whereas touching is allowed, permeation of objects is not in order to avoid the multiple representation of the same space.

Another example of sharing geometry objects that are members of the boundaries in different higher-dimensional geometry objects is the sharing of point geometries or curve geometries, which make up the outer and inner boundaries of a polygon. This means that each point is only represented once, and different polygons could reference this point geometry. The same applies to the representation of curves for transportation objects like roads, whose end points could be shared such as between different road segments to topologically connect them.

Note that the use of topology in CityGML datasets by sharing geometries is optional. Furthermore, an encoding of the CityGML conceptual model might restrict the usage of shared geometries. For example, it might only be allowed to share identical (support) points from different 3D polygons or only entire polygons can be shared between touching solids (like shown in [Figure 4](http://docs.ogc.org/DRAFTS/20-010.html#figure-recursiveaggregation)).

**7.3.2. Prototypic Objects / Scene Graph Concepts**

In CityGML, objects of equal shape like trees and other vegetation objects, traffic lights and traffic signs can be represented as prototypes which are instantiated multiple times at different locations (see [Figure 5](http://docs.ogc.org/DRAFTS/20-010.html#figure-prototypicshapes)). The geometry of prototypes is defined in local coordinate systems. Every instance is represented by a reference to the prototype, a base point in the world coordinate reference system (CRS) and a transformation matrix that facilitates scaling, rotation, and translation of the prototype. The principle is adopted from the concept of scene graphs used in computer graphics standards like X3D and COLLADA. Since the ISO 19107 geometry model does not provide support for scene graph concepts, the CityGML class ImplicitGeometry has been introduced (for further description see [Section 8.2.5](http://docs.ogc.org/DRAFTS/20-010.html#geometry-lod-section)). The prototype geometry can be represented using ISO 19107 geometry objects or by referencing an external file containing the geometry in another data format like X3D or COLLADA.

Figure 5. Examples of prototypic shapes (source: Rheinmetall Defence Electronics).

**7.3.3. Point Cloud Representation**

In addition to the spatial representations defined in the *Core* module, the geometry of physical spaces and of thematic surfaces can now also be provided by 3D point clouds using MultiPoint geometry. This allows, for example, spatially representing the building hull, a room within a building or a single wall surface just by a point cloud. All thematic feature types including transportation objects, vegetation, city furniture, etc. can also be spatially represented by point clouds. In this way, the ClearanceSpace of a road or railway could, for instance, be modelled directly from the result of a mobile laser scanning campaign. Point clouds can either be included in a CityGML dataset or just reference an external file of some common types such as LAS or LAZ.

**7.3.4. Coordinate Reference Systems (CRS)**

CityGML is about 3D city and landscape models. This means that nearly all geometries use 3D coordinates, where each single point and also the points defining the boundaries of surfaces and solids have three coordinate values (x,y,z) each. Coordinates always have to be given with respect to a coordinate reference system (CRS) that relates them unambiguously with a specific position on the Earth. In contrast to CAD or BIM, each 3D point is absolutely georeferenced, which makes CityGML especially suitable to represent geographically large extended structures like airports, railways, bridges, dams, where the Earth curvature has a significant effect on the object’s geometry (for further explanations see [Kaden & Clemen 2017](http://docs.ogc.org/DRAFTS/20-010.html#Kaden2017)).

In most CRS, the (x,y) coordinates refer to the horizontal position of a point on the Earth’s surface. The z coordinate typically refers to the vertical height over (or under) the reference surface. Note that depending on the chosen CRS, x and y may be given as angular values like latitude and longitude or as distance values in meters or feet. According to ISO 19111, numerous 3D CRS can be used. This includes global as well as national reference systems using geocentric, geodetic, or projected coordinate systems.

**7.4. CityGML Core Model: Space Concept, Levels of Detail, Special Spatial Types**

**7.4.1. Spaces and Space Boundaries**

In the CityGML 3.0 CM, a clear semantic distinction of spatial features is introduced by mapping all city objects onto the semantic concepts of spaces and space boundaries. A *Space* is an entity of volumetric extent in the real world. Buildings, water bodies, trees, rooms, and traffic spaces are examples for such entities with volumetric extent. A *Space Boundary* is an entity with areal extent in the real world. Space Boundaries delimit and connect Spaces. Examples are the wall surfaces and roof surfaces that bound a building, the water surface as boundary between the water body and air, the road surface as boundary between the ground and the traffic space, or the digital terrain model representing the space boundary between the over- and underground space.

To obtain a more precise definition of spaces, they are further subdivided into physical spaces and logical spaces. Physical spaces are spaces that are fully or partially bounded by physical objects. Buildings and rooms, for instance, are physical spaces as they are bounded by walls and slabs. Traffic spaces of roads are physical spaces as they are bounded by road surfaces against the ground. Logical spaces, in contrast, are spaces that are not necessarily bounded by physical objects, but are defined according to thematic considerations. Depending on the application, logical spaces can also be bounded by non-physical, i.e. virtual boundaries, and they can represent aggregations of physical spaces. A building unit, for instance, is a logical space as it aggregates specific rooms to flats, the rooms being the physical spaces that are bounded by wall surfaces, whereas the aggregation as a whole is being delimited by a virtual boundary. Other examples are city districts which are bounded by virtual vertically extruded administrative boundaries, public spaces vs. Security zones in airports, or city zones with specific regulations stemming from urban planning. The definition of physical and logical spaces and of corresponding physical and virtual boundaries is in line with the discussion in [[Smith & Varzi 2000](http://docs.ogc.org/DRAFTS/20-010.html#Smith2000)] on the difference between bona fide and fiat boundaries to bound objects. Bona fide boundaries are physical boundaries; they correspond to the physical boundaries of physical spaces in the CityGML 3.0 CM. In contrast, fiat boundaries are man-made boundaries: They are equivalent to the virtual boundaries of logical spaces.

Physical spaces, in turn, are further classified into occupied spaces and unoccupied spaces. Occupied spaces represent physical volumetric objects that occupy space in the urban environment. Examples for occupied spaces are buildings, bridges, trees, city furniture, and water bodies. Occupying space means that some space is blocked by these volumetric objects. For instance, the space blocked by the building in [Figure 6](http://docs.ogc.org/DRAFTS/20-010.html#figure-occupiedandunoccupiedspaces) cannot be used any more for driving through this space or placing a tree on that space. In contrast, unoccupied spaces represent physical volumetric entities that do not occupy space in the urban environment, i.e. no space is blocked by these volumetric objects. Examples for unoccupied spaces are building rooms and traffic spaces. There is a risk of misunderstanding the term OccupiedSpace. However, we decided to use the term anyway, as it is established in the field of robotics for over three decades [[Elfes 1989](http://docs.ogc.org/DRAFTS/20-010.html#Elfes1989)]. The navigation of mobile robots makes use of a so-called occupancy map that marks areas that are occupied by matter and, thus, are not navigable for robots.

Figure 6. Occupied and unoccupied spaces

The new space concept offers several advantages:

* In the CityGML 3.0 CM, all geometric representations are only defined in the *Core* module. This makes (a) models of the thematic modules simpler as they no longer need to be associated directly with the geometry classes, and (b) implementation easier as all spatial concepts have only to be implemented once in the *Core* module and all thematic modules like *Building*, *Relief*, *WaterBody*, etc. are inheriting them.
* The space concept supports the expression of explicit topological, geometrical, and thematic relations between spaces and spaces, spaces and space boundaries, and space boundaries and space boundaries. Thus, implementing the checking of geometric-topological consistency will become easier This is because most checks can be expressed and performed on the CityGML *Core* module and then automatically applied to all thematic modules.
* For the analysis of navigable spaces (e.g. to generate IndoorGML data from CityGML) algorithms can be defined on the level of the *Core* module. These algorithms will then work with all CityGML feature classes and also ADEs as they are derived from the *Core*. The same is true for other applications of 3D city models listed in [[Biljecki et al. 2015](http://docs.ogc.org/DRAFTS/20-010.html#Biljecki2015)] such as visibility analyses, including shadow casting or solar irradiation analyses.
* Practitioners and developers do not see much of the space concept. This is because the space and space boundary classes are just abstract classes. Only elements representing objects from concrete subclasses such as Building, BuildingRoom, or TrafficSpace will appear in CityGML data sets.

**7.4.2. Modelling City Objects by the Composition of Spaces**

Semantic objects in CityGML are often composed of parts, i.e. they form multi-level aggregation hierarchies. This also holds for semantic objects representing occupied and unoccupied spaces. In general, two types of compositions can be distinguished:

1. **Spatial partitioning**: Semantic objects of either the space type OccupiedSpace or UnoccupiedSpace are subdivided into different parts that are of the same space type as the parent object. Examples are Buildings that can be subdivided into BuildingParts, or Buildings that are partitioned into ConstructiveElements. Buildings as well as BuildingParts and constructiveElements represent OccupiedSpaces. Similarly, Roads can be subdivided into TrafficSpaces and AuxiliaryTrafficSpaces, all objects being UnoccupiedSpaces.
2. **Nesting of alternating space types**: Semantic objects of one space type contain objects that are of the opposite space type as the parent object. Examples are Buildings (OccupiedSpace) that contain BuildingRooms (UnoccupiedSpace), BuildingRooms (UnoccupiedSpace) that contain Furniture (OccupiedSpace), and Roads (UnoccupiedSpace) that contain CityFurniture (OccupiedSpace). The categorization of a semantic object into occupied or unoccupied takes place at the level of the object in relation to the parent object. A building is part of a city model. Thus, in the first place the building occupies urban space within a city. As long as the interior of the building is not modelled in detail, the space covered by the building needs to be considered as occupied and only viewable from the outside. To make the building accessible inside, voids need to be added to the building in the form of building rooms. The rooms add free space to the building interior. In other words,. the OccupiedSpace now contains some UnoccupiedSpace. The free space inside the building can, in turn, contain objects that occupy space again, such as furniture or installations. In contrast, roads also occupy urban space in the city. However, this space is initially unoccupied as it is accessible by cars, pedestrian, or cyclists. Adding traffic signs or other city furniture objects to the free space results in specific sections of the road becoming occupied by these objects. Thus, one can also say that occupied spaces are mostly filled with matter; whereas, unoccupied spaces are mostly free of matter and, thus, realize free spaces.

**7.4.3. Rules for Surface Orientations of OccupiedSpaces and UnoccupiedSpaces**

The classification of feature types into OccupiedSpace and UnoccupiedSpace also defines the semantics of the geometries attached to the respective features. For OccupiedSpaces, the attached geometries describe volumes that are (mostly) physically occupied. For UnoccupiedSpaces, the attached geometries describe (or bound) volumes that are (mostly) physically unoccupied. This also has an impact on the required orientation of thesurface normal (at point *P* is a [vector](https://en.wikipedia.org/wiki/Vector_(geometry)" \o "Vector (geometry)) [perpendicular](https://en.wikipedia.org/wiki/Perpendicular" \o "Perpendicular) to the [tangent plane](https://en.wikipedia.org/wiki/Tangent_space" \o "Tangent space) of the surface at *P)* for attached thematic surfaces. For OccupiedSpaces, the normal vectors of thematic surfaces must point in the same direction as the surfaces of the outer shell of the volume. For UnoccupiedSpaces, the normal vectors of thematic surfaces must point in the opposite direction as the surfaces of the outer shell of the volume. This means that from the perspective of an observer of a city scene, the surface normal must always be directed towards the observer. In the case of OccupiedSpaces (e.g. Buildings, Furniture), the observer must be located outside the OccupiedSpace for the surface normal being directed towards the observer; whereas in the case of UnoccupiedSpaces (e.g. Rooms, Roads), the observer is typically inside the UnoccupiedSpace.

**7.4.4. Levels of Detail (LOD)**

The CityGML CM differentiates four consecutive Levels of Detail (LOD 0-3), where objects become more detailed with increasing LOD with respect to their geometry. CityGML datasets can - but do not have to - contain multiple geometries for each object in different LODs simultaneously. The LOD concept facilitates multi-scale modelling; i.e. having varying degrees of spatial abstractions that are appropriate for different applications or visualizations.

The classification of real-world objects into spaces and space boundaries is solely based on the semantics of these objects and not on their used geometry type, as the CityGML 3.0 CM allows various geometrical representations for objects. A building, for instance, can be spatially represented by a 3D solid (e.g. in LOD1), but at the same time, the real-world geometry can also be abstracted by a single point, footprint or roof print (LOD0), or by a 3D mesh (LOD3). The outer shell of the building may also be semantically decomposed into wall, roof, and ground surfaces. [Figure 7](http://docs.ogc.org/DRAFTS/20-010.html#figure-buildinglods) shows different representations of the same real-world building object in different geometric LODs (and appearances).

Figure 7. Representation of the same real-world building in the Levels of Detail 0-3.

The biggest changes between CityGML 3.0 and earlier versions are that:

1. LOD4 was dropped, because now all feature types can have outdoor and indoor elements in LODs 0-3 (for those city objects where it makes sense like buildings, tunnels, or bridges). This means that the outside shell, such as of a building, could be spatially represented in LOD2 and the indoor elements like rooms, doors, hallways, stairs etc. in LOD1. CityGML can now be used to represent building floor plans, which are LOD0 representations of building interiors (cf. [Konde et al. 2018](http://docs.ogc.org/DRAFTS/20-010.html#Konde2018)). It is even possible to model the outside shell of a building in LOD1, while representing the interior structure in LOD2 or 3. [Figure 8](http://docs.ogc.org/DRAFTS/20-010.html#figure-floorplan) shows different indoor/outdoor representations of a building. Details on the changes to the CityGML LOD concept are provided in [[[Lowner2016]](http://docs.ogc.org/DRAFTS/20-010.html#Lowner2016)].
2. Levels of Detail are no longer associated with the degree of semantic decomposition of city objects and refer to the spatial representations only. This means that, for example, buildings can have thematic surfaces (like WallSurface, GroundSurface) also in LODs 0 and 1 and windows and doors can be represented in all LODs 0-3. In CityGML 2.0 or earlier thematic surfaces were only allowed starting from LOD2, openings like doors and windows starting from LOD3, and interior rooms and furniture only in LOD4.
3. In the CityGML 3.0 CM, the geometry representations were moved from the thematic modules to the *Core* module and are now associated with the semantic concepts of *Spaces* and *Space Boundaries*. This led to a significant simplification of the models of the thematic modules. Since all feature types in the thematic modules are defined as subclasses of the space and space boundary classes, they automatically inherit the geometry classes and, thus, no longer require direct associations with them. This also led to a harmonized LOD representation over all CityGML feature types.
4. If new feature types are defined in Application Domain Extensions (ADEs) based on the abstract Space and Space Boundary classes from the Core module, they automatically inherit the spatial representations and the LOD concept.

Figure 8. Floor plan representation (LOD0) of a building (left), combined LOD2 indoor and outdoor representation (right). Image adopted from [[Lowner2016]](http://docs.ogc.org/DRAFTS/20-010.html#Lowner2016).

*Spaces* and all its subclasses like *Building*, *Room*, and *TrafficSpace* can now be spatially represented by single points in LOD0, multi-surfaces in LOD0/2/3, solids in LOD1/2/3, and multi-curves in LOD2/3. *Space Boundaries* and all its subclasses such as *WallSurface*, *LandUse*, or *Relief* can now be represented by multi-surfaces in LOD0/2/3 and as multi-curves in LOD2/3. See [Section 8.2.5](http://docs.ogc.org/DRAFTS/20-010.html#geometry-lod-section) for further details on the different Levels of Detail.

**7.4.5. Closure Surfaces**

Objects, which are spatially not represented by a volumetric geometry, must be virtually closed in order to compute their volume (e.g. pedestrian underpasses or airplane hangars). They can be sealed using a specific type of space boundary called ClosureSurface. These are virtual surfaces, which are taken into account, when needed to compute volumes and are neglected, when they are irrelevant or not appropriate, for example in visualizations.

The concept of ClosureSurface can also be employed to model the entrances of subsurface objects. Those objects like tunnels or pedestrian underpasses have to be modelled as closed solids in order to compute their volume. An example would be for use in flood simulations. The entrances to subsurface objects also have to be sealed to avoid holes in the digital terrain model (see [Figure 9](http://docs.ogc.org/DRAFTS/20-010.html#figure-closuresurfaces)). However, in close-range visualizations the entrance must be treated as open. Thus, closure surfaces are an adequate way to model those entrances.

Figure 9. Closure surfaces to seal open structures. Passages are subsurface objects (left). The entrance is sealed by a virtual ClosureSurface feature, which is both part of the DTM and the subsurface object (right) (graphic: IGG Uni Bonn).

**7.4.6. Terrain Intersection Curves**

An important issue in city modelling is the integration of 3D objects and the terrain. Problems arise if 3D objects float over or sink into the terrain. This is particularly the case when terrain and 3D objects in different LODs are combined, when the terrain and 3D models are updated independently from each other, or when they come from different data providers [[Kolbe & Gröger 2003](http://docs.ogc.org/DRAFTS/20-010.html#Kolbe2003)]. To overcome this problem, the TerrainIntersectionCurve (TIC) of a 3D object is introduced. These curves denote the exact position where the terrain touches the 3D object (see [Figure 10](http://docs.ogc.org/DRAFTS/20-010.html#figure-terrainintersectioncurves)). TICs can be applied to all CityGML feature types that are derived from AbstractPhysicalSpace, such as buildings, bridges, tunnels, but also city furniture, vegetation, and generic city objects.

If, for example, a building has a courtyard, the TIC consists of two closed rings: One ring representing the courtyard boundary, and one which describes the building’s outer boundary. This information can be used to integrate the building and a terrain by ‘pulling up’ or ‘pulling down’ the surrounding terrain to fit the TerrainIntersectionCurve. The digital terrain model (DTM) may be locally warped to fit the TIC. By this means, the TIC also ensures the correct positioning of textures or the matching of object textures with the DTM. Since the intersection with the terrain may differ depending on the LOD, a 3D object may have different TerrainIntersectionCurves for all LODs.

Figure 10. TerrainIntersectionCurve for a building (left, black) and a tunnel object (right, red). The tunnel’s hollow space is sealed by a triangulated ClosureSurface (graphic: IGG Uni Bonn).

**7.4.7. Coherent Semantical-Geometrical Modelling**

An important design principle for CityGML is the coherent modelling of semantic objects and their spatial representations. At the semantic level, real-world entities are represented by features, such as buildings, walls, windows, or rooms. The description also includes attributes, relations and aggregation hierarchies (part-whole-relations) between features. Thus the part-of-relationship between features can be derived at the semantic level only, without considering geometry. However, at the spatial level, geometry objects are assigned to features representing their spatial location, shape, and extent. So the model consists of two hierarchies: The semantic and the geometrical in which the corresponding objects are linked by relationships (cf. [Stadler & Kolbe 2007](http://docs.ogc.org/DRAFTS/20-010.html#Stadler2007)). The advantage of this approach is that it can be navigated in both hierarchies and between both hierarchies arbitrarily, for answering thematic and/or geometrical queries or performing analyses.

If both hierarchies exist for a specific object, they must be coherent (i.e. it must be ensured that they match and fit together). For example, if a building is semantically decomposed into wall surfaces, roof surfaces and so forth the polygons representing these thematic surfaces (in a specific LOD) must be part of the solid geometry representing the entire building (for the same LOD).

**7.5. Appearances**

In addition to semantics and geometry information about the appearance of surfaces, i.e. observable properties of the surface, is considered an integral part of virtual 3D city and landscape models. Appearance relates to any surface-based theme, such as infrared radiation or noise pollution, not just visual properties like RGB texture images. Consequently, data provided by appearances can be used as input for both, presentation of and analysis in virtual 3D city models.

The CityGML CM supports feature appearances for an arbitrary number of themes per city model. Each LOD of a feature can have an individual appearance. Appearances can represent – among others – textures and georeferenced textures. CityGML’s appearance model is packaged within the Appearance module (cf. [Section 8.3](http://docs.ogc.org/DRAFTS/20-010.html#rc_appearance_section)).

**7.6. Modelling Dynamic Data**

In general, city objects can have properties related to their geometry, topology, semantics, and appearance. All of these properties may change over time. For example, a construction event leads to the change in geometry of a building (i.e. addition of a new building floor or demolition of an existing door). The geometry of an object can be further classified according to its shape, location, and extent, which can also change over time. A moving car object involves changing only the location of the car object. However, a flood incident involves variations in the location and shape of water. There might be other properties, which change with respect to thematic data of city objects, such as hourly variations in energy or gas consumption of a building or changing the building usage from residential to commercial. Some properties involve changes in appearances over a time period, such as building textures changing over years or traffic cameras recording videos of moving traffic over definite intervals. 3D city models also represent interrelationships between objects and relations may change over time as well. Hence, it is important to consider that the representation of time-varying data is required to be associated with these different properties. A detailed discussion on the requirements of city model applications regarding the support of dynamic data is given in [[Chaturvedi & Kolbe 2019](http://docs.ogc.org/DRAFTS/20-010.html#Chaturvedi2019)].

The CityGML 3.0 CM introduces two concepts to manage dynamic, time-dependent, properties of city models. The *Versioning* module manages changes that are slower in nature: (1) The history or evolution of cities such as construction or demolition of buildings, and (2) managing multiple versions of the city models. The *Dynamizer* module manages higher-frequency or dynamic variations of object properties, including variations of (1) thematic attributes such as changes of physical quantities (energy demands, temperature, solar irradiation levels), (2) spatial properties such as change of a feature’s geometry, with respect to shape and location (moving objects), and (3) real-time sensor observations. The Dynamizer module allows establishing explicit links from city objects to sensors and sensor data services.

**7.6.1. Versioning and Histories**

As described in [Section 7.2.1](http://docs.ogc.org/DRAFTS/20-010.html" \l "overview-section-semantic-modelling),the bi-temporal timestamps of all CityGML feature types allow representing the evolution of the real city and its model over time. The new *Versioning* module extends this concept by the possibility of representing multiple, concurrent versions of the city model. For that purpose, the module defines two new feature types: 1) *Version*, which can be used to explicitly define named states of the 3D city model and denote all the specific versions of objects belonging to such states. 2) *VersionTransition*, which allows to explicitly link different versions of the 3D city model by describing the reason of change and the modifications applied. Details on the versioning concept are given in [[Chaturvedi et al. 2015](http://docs.ogc.org/DRAFTS/20-010.html#Chaturvedi2015)].

This approach not only facilitates the explicit representation of different city model versions, but also allows distinguishing and referring to different versions of city objects in an interoperable exchange format. All object versions could be stored and exchanged within a single dataset. Software systems could use such a dataset to visualize and work with the different versions simultaneously. The conceptual model also takes into account the management of multiple histories or multiple interpretations of the past of a city, which is required when looking at historical city developments and for archaeological applications. In addition, the Versioning module supports collaborative work: All the functionality to represent a tree of workspaces as version control systems like *git* or *SVN* is provided. The Versioning module handles versions and version transitions as feature types, which allows the version management to be completely handled using the standard OGC Web Feature Service [[Vrenatos 2010](http://docs.ogc.org/DRAFTS/20-010.html#Vretanos2010)]. No extension of this standard is required to manage the versioning of city models.

**7.6.2. Dynamizers: Using Time-Series Data for Object Attributes**

The new Dynamizer module improves the usability of CityGML for different kinds of simulations as well as to facilitate the integration of devices from the Internet-of-Things (IoT) like sensors with 3D city models. Both simulations and sensors provide dynamic variations of some measured or simulated properties such as electricity consumption of a building or the traffic density within a road segment. The variations of the value are typically represented using time-series data. The data sources of the time-series data could be either sensor observations (e.g. from a smart meter), pre-recorded load profiles (e.g. from an energy company), or the results of some simulation run.

Figure 11. Dynamizers link timeseries data coming from different sources to specific properties of individual city objects.

As shown in [Figure 11](http://docs.ogc.org/DRAFTS/20-010.html#figure-dynamizers), Dynamizers serve three main purposes:

1. Dynamizer is a data structure to represent dynamic values in different and generic ways. Such dynamic values may be given by (1) tabulation of time/value pairs using its *AtomicTimeseries* class, (2) patterns of time/value pairs based on statistical rules using its *CompositeTimeseries* class, and (3) retrieving observations directly from external sensor/IoT services using its *SensorConnection* class. The values can be obtained from sensor services such as the OGC Sensor Observation Service or OGC SensorThings API, simulation specific databases, and also external files such as CSV or Excel sheets.
2. Dynamizer delivers a method to enhance static city models by specifying dynamic property values. It references a specific property (e.g. spatial, thematic or appearance properties) of a specific object within a 3D city model providing dynamic values overriding the static value of the referenced object attribute.
3. Dynamizer objects establish explicit links between sensor/observation data and the respective properties of city model objects that are measured by them. By making such explicit links with city object properties, the semantics of sensor data become implicitly defined by the city model.

Dynamizers are used to inject dynamic variations of city object properties into an otherwise static representation. The advantage in following such an approach is that it allows only selected properties of city models to be made dynamic. If an application does not support dynamic data, the application simply does not allow/include these special types of features.

Dynamizers have already been implemented as an Application Domain Extension (ADE) for CityGML 2.0 and were employed in the OGC Future City Pilot Phase 1. More details about Dynamizers are given in [[Chaturvedi & Kolbe 2017](http://docs.ogc.org/DRAFTS/20-010.html#Chaturvedi2017)].

**7.7. Extending CityGML**

CityGML is designed as a universal topographic information model that defines object types and attributes which are useful for a broad range of applications. In practical applications, the objects within specific 3D city models will most likely contain attributes which are not explicitly modelled in CityGML. Moreover, there might be 3D objects which are not covered by the CityGML CM thematic classes. The CityGML CM provides three different concepts to support the exchange of such data:

1. [Generic objects and attributes](http://docs.ogc.org/DRAFTS/20-010.html#rc_generics_section),
2. [Application Domain Extensions](http://docs.ogc.org/DRAFTS/20-010.html#rc_ade_section), and
3. [Code lists](http://docs.ogc.org/DRAFTS/20-010.html#codelist-definition).

The concept of generic objects and attributes enables the runtime extensions of CityGML applications such as any city object may be augmented by additional attributes and relations, whose names, data types, and values can be provided by a running application without requiring extensions to the CityGML conceptual schema and the respective encodings. Similarly, features not represented by the predefined thematic classes of the CityGML conceptual model may be modelled and exchanged using generic objects. The generic extensions of CityGML are provided by the *Generics* module (cf. [Section 8.7](http://docs.ogc.org/DRAFTS/20-010.html#rc_generics_section)).

Application Domain Extensions (ADE) specify additions to the CityGML conceptual model. Such additions comprise the introduction of new properties to existing CityGML feature types such as the energy demand of a building or the definition of additional feature types. The difference between ADEs and generic objects and attributes is that an ADE has to be defined in an extra conceptual schema (provided in UML) with its own namespace. Encodings have to be extended accordingly. The advantage of this approach is that the extension is formally specified. Extended CityGML datasets can be validated against the CityGML CM and the respective ADE schema. ADEs can be defined (and even standardized) by information communities which are interested in specific application fields. More than one ADE can be used simultaneously in the same dataset. Examples for popular ADEs are the Utility Network ADE [[Becker et al. 2011](http://docs.ogc.org/DRAFTS/20-010.html#Becker2011); [Kutzner et al. 2018](http://docs.ogc.org/DRAFTS/20-010.html#Kutzner2018)] and the Energy ADE [[Nouvel et al. 2015](http://docs.ogc.org/DRAFTS/20-010.html#Nouvel2015); [Agugiaro et al. 2018](http://docs.ogc.org/DRAFTS/20-010.html#Agugiaro2018)]. A comprehensive overview of CityGML ADEs is given in [[Biljecki et al. 2018](http://docs.ogc.org/DRAFTS/20-010.html#Biljecki2018)]. Further details on ADEs are given in [Chapter 10](http://docs.ogc.org/DRAFTS/20-010.html#rc_ade_section).

CityGML can also be extended with regard to the allowed values specified in code lists. Many attributes of CityGML types use a code list as a data type such as, for instance, the attributes *class*, *usage*, and *function* of city objects. A code list defines a value domain including a code for each permissible value. In contrast to fixed enumerations, modifications and extensions to the value domain become possible with code lists. The values for all code lists in CityGML have to be defined externally. This could for example, be by adopting classifications from global, national, or industrial standards.

Additional information about the extension features of CityGML can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

**8. CityGML UML Model**

The CityGML UML model is the normative definition of the CityGML Conceptual Model. The tables and figures in this section were software generated from the UML model. As such, this section provides a normative representation of the CityGML Conceptual Model.

An alternate representation can be found in the Data Dictionary in [Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section).

**8.1. Structural Overview of Requirements Classes**

The Requirements Classes for this standard are structured as UML Packages as illustrated in [Figure 12](http://docs.ogc.org/DRAFTS/20-010.html#package-diagram). Each Requirements Class is specified in detail in their respective subsections. These subsections include a UML diagram, data dictionary, and the applicable requirements.

Figure 12. CityGML UML Packages

**8.2. Core**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-core> | |
| Target type | Implementation Specification |
| Dependency | [ISO 19103:2015](http://docs.ogc.org/DRAFTS/20-010.html#iso19103) |
| Dependency | [ISO 19107:2003](http://docs.ogc.org/DRAFTS/20-010.html#iso19107) |
| Dependency | [ISO 19109:2015](http://docs.ogc.org/DRAFTS/20-010.html#iso19109) |
| Dependency | [ISO 19111:2019](http://docs.ogc.org/DRAFTS/20-010.html#iso19111) |
| Dependency | [ISO 19123:2005](http://docs.ogc.org/DRAFTS/20-010.html#iso19123) |
| Dependency | [OASIS xAL v3.0](http://docs.ogc.org/DRAFTS/20-010.html#xal2) |

The CityGML Core module defines the basic concepts and components of city models that can be modelled and encoded based on the CityGML CM. This rather large body of work is divided into seven sections. These sections build on each other from the fundamental principles specified by the relevant ISO standards up to the full CityGML model. These sections are summarized in [Table 3](http://docs.ogc.org/DRAFTS/20-010.html#citygml-core-table).

|  |  |
| --- | --- |
| Table 3. CityGML Core Sections | |
| [The Use of ISO Standards](http://docs.ogc.org/DRAFTS/20-010.html#ISO-dependencies-section) | Describes the use of the ISO 19100 series of International Standards to provide a foundation to the CityGML model. |
| [City Models and City Objects](http://docs.ogc.org/DRAFTS/20-010.html#city-objects-section) | Defines the basic building blocks of the CityGML model. |
| [Space Concept](http://docs.ogc.org/DRAFTS/20-010.html#space-concepts-section) | Defines the concepts of space as used in the CityGML model. |
| [Geometry and LOD](http://docs.ogc.org/DRAFTS/20-010.html#geometry-lod-section) | Defines the geometry and Levels Of Detail concepts. |
| [CityGML Core Model](http://docs.ogc.org/DRAFTS/20-010.html#CityGML-section) | Presents the complete Core model. |
| [Types, Enumerations, and Codelist](http://docs.ogc.org/DRAFTS/20-010.html#b-e-c-section) | Defines the little things which make this model work. |

**8.2.1. Requirements**

The CityGML Core defines technology-agnostic concepts. These concepts are then realized in technology-specific Implementation Specifications. The following requirements govern the creation of any CityGML compliant Implementation Specification (IS).

|  |  |
| --- | --- |
| **Requirement 1** | **/req/core/classes** |
| For each UML class defined or referenced in the Core Package: | |
| A | The IS SHALL contain an element which represents the same concept as that defined for the UML class. |
| B | The IS SHALL represent associations with the same source, target, direction, roles, and multiplicities as those of the UML class. |
| C | The IS SHALL represent the attributes of the UML class including the name, definition, type, and multiplicity. |
| D | The IS SHALL represent the attributes of all superclasses of the UML class including the name, definition, type, and mutiplicity. |
| E | The IS SHALL represent the associations of all superclasses of the UML class including the source, target, direction, roles, and multiplicity. |
| F | The IS SHALL specify how an implemenetion observes all constraints the Conceptual Model imposes on the UML class. |

While the CityGML Conceptual Model builds on ISO Standards, there are some restrictions on the use of those standards.

|  |  |
| --- | --- |
| **Requirement 2** | **/req/Core/isorestrictions** |
| ISO classes used in the CityGML Conceptual Model are subject to the following restrictions: | |
| A | Classes derived from the GM\_Solid class (ISO 19107) SHALL NOT include interior boundaries. (The interior association on the GM\_SolidBoundary shall not be defined) |

An implementing technology may not be able to support all of the concepts defined in the CityGML Conceptual Model. Alternately, some concepts from the Conceptual Model may be inappropriate for the application domain for which the IS was developed. In those cases, elements of the Conceptual Model may be mapped to null elements in the IS.

|  |  |
| --- | --- |
| **Permission 1** | **/per/Core/classes** |
| For each UML class defined or referenced in CityGML Conceptual Model: | |
| A | An IS MAY represent that class as a null class with no attributes, associations, or definition. |
| B | An IS MAY represent an association of the UML class with a null association. |
| C | An IS MAY represent an attribute of the UML class with a null attribute. |
| D | Whenever a null element is used to represent a concept from the Conceptual Model, the IS SHOULD document that mapping and provide an explanation for why that concept was not implemented. |

[Table 4](http://docs.ogc.org/DRAFTS/20-010.html#core-boundaries-table) lists the surfaces that are allowed as thematic surface boundaries of the space classes defined in the Core module:

|  |  |
| --- | --- |
| Table 4. Core space classes and their allowed thematic surface boundaries | |
| **Space class** | **Allowed space boundaries** |
| AbstractLogicalSpace | * Core::AbstractSpaceBoundary and the subclasses:     Core::AbstractThematicSurface,     Core::ClosureSurface * Generics::GenericThematicSurface * possible classes from ADEs |
| AbstractOccupiedSpace | * Core::AbstractSpaceBoundary and the subclasses:     Core::AbstractThematicSurface,     Core::ClosureSurface * Generics::GenericThematicSurface * possible classes from ADEs |
| AbstractPhysicalSpace | * Core::AbstractSpaceBoundary and the subclasses:     Core::AbstractThematicSurface,     Core::ClosureSurface * Generics::GenericThematicSurface * possible classes from ADEs |
| AbstractSpace | * Core::AbstractSpaceBoundary and the subclasses:     Core::AbstractThematicSurface,     Core::ClosureSurface * Generics::GenericThematicSurface * possible classes from ADEs |
| AbstractUnoccupiedSpace | * Core::AbstractSpaceBoundary and the subclasses:     Core::AbstractThematicSurface,     Core::ClosureSurface * Generics::GenericThematicSurface * possible classes from ADEs |

Surface boundaries are constrained by the following requirement:

|  |  |
| --- | --- |
| **Requirement 3** | **/req/core/boundaries** |
| [Table 4](http://docs.ogc.org/DRAFTS/20-010.html#core-boundaries-table) lists the surfaces that are allowed as thematic surface boundaries of the space classes defined in the Core module. An IS SHALL NOT specify boundaries except as specified in [Table 4](http://docs.ogc.org/DRAFTS/20-010.html#core-boundaries-table) | |

The use of extension capabilities by Core elements is constrained by the following requirement:

|  |  |
| --- | --- |
| **Requirement 4** | **/req/core/ade/use** |
| ADE element and property extensions SHALL NOT be used unless conformance with the ADE Requirements Class can be demonstrated. | |

**8.2.2. ISO Dependencies**

CityGML builds on the ISO 19100 family of standards. The applicable standards are identified in the diagram in [Figure 13](http://docs.ogc.org/DRAFTS/20-010.html#ISO-in-CityGML-diagram). Data dictionaries are included for all of the ISO-defined classes explicitly referenced in the CityGML UML model. These data dictionaries are provided for the convenience of the user. The ISO standards are the normative source.

Figure 13. Use of ISO Standards in CityGML

The ISO classes explicitly used in the CityGML UML model are introduced in [Table 5](http://docs.ogc.org/DRAFTS/20-010.html#iso-class-table). More details about these classes can be found in the Data Dictionary in [Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section).

|  |  |
| --- | --- |
| Table 5. ISO Classes used in CityGML | |
| **Class Name** | **Description** |
| [AnyFeature](http://docs.ogc.org/DRAFTS/20-010.html#AnyFeature-section) | A generalization of all feature types |
| [CV\_DiscreteGridPointCoverage](http://docs.ogc.org/DRAFTS/20-010.html#CV_DiscreteGridPointCoverage-section) | A coverage that returns the same feature attribute values for every direct position within any object in its domain. |
| [Direct Position](http://docs.ogc.org/DRAFTS/20-010.html#DirectPosition-section) | The coordinates for a position within some coordinate reference system. |
| [GM\_Object](http://docs.ogc.org/DRAFTS/20-010.html#GM_Object-section) | root class of the geometric object taxonomy. |
| [GM\_MultiCurve](http://docs.ogc.org/DRAFTS/20-010.html#GM_MultiCurve-section) | An aggregate class containing only instances of GM\_OrientableCurve. |
| [GM\_MultiPoint](http://docs.ogc.org/DRAFTS/20-010.html#GM_MultiPoint-section) | An aggregate class containing only points. |
| [GM\_MultiSurface](http://docs.ogc.org/DRAFTS/20-010.html#GM_MultiSurface-section) | An aggregate class containing only instances of GM\_OrientableSurface. |
| [GM\_Point](http://docs.ogc.org/DRAFTS/20-010.html#GM_Point-section) | The basic data type for a geometric object consisting of one and only one point. |
| [GM\_Solid](http://docs.ogc.org/DRAFTS/20-010.html#GM_Solid-section) | The basis for 3-dimensional geometry. The extent of a solid is defined by the boundary surfaces. |
| [GM\_Surface](http://docs.ogc.org/DRAFTS/20-010.html#GM_Surface-section) | The basis for 2-dimensional geometry. |
| [GM\_Tin](http://docs.ogc.org/DRAFTS/20-010.html#GM_Tin-section) | A GM\_TriangulatedSurface which uses the Delaunay or similar algorithm. |
| [GM\_TriangulatedSurface](http://docs.ogc.org/DRAFTS/20-010.html#GM_TriangulatedSurface-section) | A GM\_PolyhedralSurface that is composed only of triangles |
| [SC\_CRS](http://docs.ogc.org/DRAFTS/20-010.html#SC_CRS-section) | Coordinate reference system which is usually single but may be compound. |
| [TM\_Position](http://docs.ogc.org/DRAFTS/20-010.html#TM_Position-section) | A union class that consists of one of the data types listed as its attributes. |

**8.2.3. City Models and City Objects**

City models are virtual representations of real-world cities and landscapes. A city model aggregates different types of objects, which can be city objects, appearances, different versions of the city model, transitions between different versions of the city model, and feature objects. All objects defined in the CityGML CM are features with lifespan. This allows the optional specification of the real-world and database times for the existence of each feature, as is required by the Versioning module (cf. [Section 8.13](http://docs.ogc.org/DRAFTS/20-010.html#rc_versioning_section)). Features that define thematic concepts related to cities and landscapes, such as building, bridge, water body, or land use, are referred to as city objects. All city objects define properties that describe the objects in more detail. These static properties can be overridden with time-varying data through Dynamizers (cf. [Section 8.6](http://docs.ogc.org/DRAFTS/20-010.html#rc_dynamizer_section)).

Figure 14. UML City Models and City Objects

The City Model and City Object classes defined in the CityGML UML model are introduced in [Table 6](http://docs.ogc.org/DRAFTS/20-010.html#Core-city-model-class-table). More details about these classes can be found in the Data Dictionary in [Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section).

|  |  |
| --- | --- |
| Table 6. City Model and City Object classes used in Core | |
| **Class** | **Description** |
| [AbstractAppearance](http://docs.ogc.org/DRAFTS/20-010.html#AbstractAppearance-section) «FeatureType» | AbstractAppearance is the abstract superclass to represent any kind of appearance objects. |
| [AbstractCityObject](http://docs.ogc.org/DRAFTS/20-010.html#AbstractCityObject-section) «FeatureType» | AbstractCityObject is the abstract superclass of all thematic classes within the CityGML CM. |
| [AbstractDynamizer](http://docs.ogc.org/DRAFTS/20-010.html#AbstractDynamizer-section) «FeatureType» | AbstractDynamizer is the abstract superclass to represent Dynamizer objects. |
| [AbstractFeature](http://docs.ogc.org/DRAFTS/20-010.html#AbstractFeature-section) «FeatureType» | AbstractFeature is the abstract superclass of all feature types within the CityGML CM. |
| [AbstractFeatureWithLifespan](http://docs.ogc.org/DRAFTS/20-010.html#AbstractFeatureWithLifespan-section) «FeatureType» | AbstractFeatureWithLifespan is the base class for all CityGML features. This class allows the optional specification of the real-world and database times for the existence of each feature. |
| [AbstractVersion](http://docs.ogc.org/DRAFTS/20-010.html#AbstractVersion-section) «FeatureType» | AbstractVersion is the abstract superclass to represent Version objects. |
| [AbstractVersionTransition](http://docs.ogc.org/DRAFTS/20-010.html#AbstractVersionTransition-section) «FeatureType» | AbstractVersionTransition is the abstract superclass to represent VersionTransition objects. |
| [CityModel](http://docs.ogc.org/DRAFTS/20-010.html#CityModel-section) «FeatureType» | CityModel is the container for all objects belonging to a city model. |

**8.2.4. Space Concept**

All city objects are differentiated into spaces and space boundaries. Spaces are entities of volumetric extent in the real world. Buildings, water bodies, trees, rooms, and traffic spaces, for instance, have a volumetric extent. Spaces can be classified into physical spaces and logical spaces. Physical spaces, in turn, can be further classified into occupied spaces and unoccupied spaces.

Space boundaries, in contrast, are entities with areal extent in the real world. Space boundaries can be differentiated into different types of thematic surfaces, such as wall surfaces and roof surfaces.

A detailed introduction to the Space concept can be found in [Section 7.4](http://docs.ogc.org/DRAFTS/20-010.html#overview-section-coremodel). In particular, the classification into OccupiedSpace and UnoccupiedSpace might not always be apparent at first sight. Carports, for instance, represent an OccupiedSpace, although they are not closed and most of the space is free of matter, see [Figure 15](http://docs.ogc.org/DRAFTS/20-010.html#figure-carport). Since a carport is a roofed, immovable structure with the purpose of providing shelter to objects (i.e. cars), carports are frequently represented as buildings in cadastres. Thus, also in CityGML, a carport should be modelled as an instance of the class Building. Since Building is transitively a subclass of OccupiedSpace, a carport is an OccupiedSpace as well. However, only in LOD1, the entire volumetric region covered by the carport would be considered as physically occupied. In LOD1, the occupied space is defined by the entire carport solid (unless a room would be defined in LOD1 that would model the unoccupied part below the roof); whereas in LOD2 and LOD3, the solids represent more realistically the really physically occupied space of the carport. In addition, for all OccupiedSpaces, the normal vectors of the thematic surfaces like the RoofSurface need to point away from the solids, i.e. consistent with the solid geometry.

Figure 15. Representation of a carport as OccupiedSpace in different LODs. The red boxes represent solids, the green area represents a surface. In addition, the normal vectors of the roof solid (in red) and the roof surface (in green) are shown.

In contrast, a room is a physically unoccupied space. In CityGML, a room is represented by the class BuildingRoom that is a subclass of UnoccupiedSpace. In LOD1, the entire room solid would be considered as unoccupied space, which can contain furniture and installations, though, as is shown in [Figure 16](http://docs.ogc.org/DRAFTS/20-010.html#figure-room). In LOD2 and 3, the solid represents more realistically the really physically unoccupied space of the room (possibly somewhat generalized as indicated in the figure). For all UnoccupiedSpaces, the normal vectors of the bounding thematic surfaces like the InteriorWallSurface need to point inside the object, i.e. opposite to the solid geometry.

Figure 16. Representation of a room as UnoccupiedSpace in different LODs. The red boxes represent solids, the green area represents a surface. In addition, the normal vectors of the room solid (in red) and the wall surface (in green) are shown.

The UML diagram of the Space concept classes is depicted in [Figure 17](http://docs.ogc.org/DRAFTS/20-010.html#core-spaceconcept).

Figure 17. UML Space Concepts

The Space Concept classes defined in the CityGML UML model are introduced in [Table 7](http://docs.ogc.org/DRAFTS/20-010.html#Core-spatial-class-table). More details about these classes can be found in the Data Dictionary in [Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section).

|  |  |
| --- | --- |
| Table 7. Space Classes used in Core | |
| **Class** | **Description** |
| [AbstractLogicalSpace](http://docs.ogc.org/DRAFTS/20-010.html#AbstractLogicalSpace-section) «FeatureType» | AbstractLogicalSpace is the abstract superclass for all types of logical spaces. Logical space refers to spaces that are not bounded by physical surfaces but are defined according to thematic considerations. |
| [AbstractOccupiedSpace](http://docs.ogc.org/DRAFTS/20-010.html#AbstractOccupiedSpace-section) «FeatureType» | AbstractOccupiedSpace is the abstract superclass for all types of physically occupied spaces. Occupied space refers to spaces that are partially or entirely filled with matter. |
| [AbstractPhysicalSpace](http://docs.ogc.org/DRAFTS/20-010.html#AbstractPhysicalSpace-section) «FeatureType» | AbstractPhysicalSpace is the abstract superclass for all types of physical spaces. Physical space refers to spaces that are fully or partially bounded by physical objects. |
| [AbstractSpace](http://docs.ogc.org/DRAFTS/20-010.html#AbstractSpace-section) «FeatureType» | AbstractSpace is the abstract superclass for all types of spaces. A space is an entity of volumetric extent in the real world. |
| [AbstractSpaceBoundary](http://docs.ogc.org/DRAFTS/20-010.html#AbstractSpaceBoundary-section) «FeatureType» | AbstractSpaceBoundary is the abstract superclass for all types of space boundaries. A space boundary is an entity with areal extent in the real world. Space boundaries are objects that bound a Space. They also realize the contact between adjacent spaces. |
| [AbstractThematicSurface](http://docs.ogc.org/DRAFTS/20-010.html#AbstractThematicSurface-section) «FeatureType» | AbstractThematicSurface is the abstract superclass for all types of thematic surfaces. |
| [AbstractUnoccupiedSpace](http://docs.ogc.org/DRAFTS/20-010.html#AbstractUnoccupiedSpace-section) «FeatureType» | AbstractUnoccupiedSpace is the abstract superclass for all types of physically unoccupied spaces. Unoccupied space refers to spaces that are entirely or mostly free of matter. |
| [ClosureSurface](http://docs.ogc.org/DRAFTS/20-010.html#ClosureSurface-section) «FeatureType» | ClosureSurface is a special type of thematic surface used to close holes in volumetric objects. Closure surfaces are virtual (non-physical) surfaces. |

**8.2.5. Geometry and LOD**

Spaces and space boundaries can have various geometry representations depending on the Levels of Detail (LOD). Spaces can be spatially represented as single points in LOD0, multi-surfaces in LOD0/2/3, solids in LOD1/2/3, and multi-curves in LOD2/3. Space boundaries can be represented as multi-surfaces in LOD0/2/3 and as multi-curves in LOD2/3. All Levels of Detail allow for the representation of the interior of city objects.

The different Levels of Detail are defined in the following way:

* LOD 0: Volumetric real-world objects (Spaces) can be spatially represented by a single point, by a set of curves, or by a set of surfaces. Areal real-world objects (Space Boundaries) can be spatially represented in LOD0 by a set of curves or a set of surfaces. LOD0 surface representations are typically the result of a projection of the shape of a volumetric object onto a plane parallel to the ground, hence, representing a footprint (e.g. a building footprint or a floor plan of the rooms inside a building). LOD0 curve representations are either the result of a projection of the shape of a vertical surface (e.g. a wall surface) onto a grounding plane or the skeleton of a volumetric shape of longitudinal extent such as a road or river segment.
* LOD 1: volumetric real-world objects (Spaces) are spatially represented by a vertical extrusion solid, i.e. a solid created from a horizontal footprint by vertical extrusion. Areal real-world objects (Space Boundaries) can be spatially represented in LOD1 by a set of horizontal or vertical surfaces.
* LOD 2: volumetric real-world objects (Spaces) can be spatially represented by a set of curves, a set of surfaces, or a single solid geometry. Areal real-world objects (Space Boundaries) can be spatially represented in LOD2 by a set of surfaces. The shape of the real-world object is generalized in LOD2 and smaller details (e.g. bulges, dents, sills, but also structures like e.g. balconies or dormers of buildings) are typically neglected. LOD2 curve representations are skeletons of volumetric shapes of longitudinal extent like an antenna or a chimney.
* LOD 3: volumetric real-world objects (Spaces) can be spatially represented by a set of curves, a set of surfaces, or a single solid geometry. Areal real-world objects (Space Boundaries) can be spatially represented in LOD3 by a set of surfaces. LOD3 is the highest level of detail and respective geometries include all available shape details.

In addition, the geometry can also be represented implicitly. The shape is stored only once as a prototypical geometry, which then is re-used or referenced, wherever the corresponding feature occurs in the 3D city model.

The thematic classes, such as building, tunnel, road, land use, water body, or city furniture are defined as subclasses of the space and space boundary classes within the thematic modules. Since all city objects in the thematic modules represent subclasses of the space and space boundary classes, they automatically inherit the geometries defined in the Core module.

The UML diagram of the Geometry and LoD concept classes is depicted in [Figure 18](http://docs.ogc.org/DRAFTS/20-010.html#core-geometry).

Figure 18. UML Geometry and LoD Concepts

The Geometry and LOD Concept classes defined in the CityGML UML model are introduced in [Table 8](http://docs.ogc.org/DRAFTS/20-010.html#Core-geometry-class-table). More detail about these classes can be found in the Data Dictionary in [Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section).

Of particular note is the Implicit Geometry concept. Many of the objects encountered in a city landscape have the same geometry. How many types of street lamps can there be? An Implicit Geometry captures that geometry once, and re-uses that one geometry for all similar street lamp objects.

|  |  |
| --- | --- |
| Table 8. Geometry Classes used in Core | |
| **Class** | **Description** |
| [AbstractOccupiedSpace](http://docs.ogc.org/DRAFTS/20-010.html#AbstractOccupiedSpace-section) «FeatureType» | AbstractOccupiedSpace is the abstract superclass for all types of physically occupied spaces. Occupied space refers to spaces that are partially or entirely filled with matter. |
| [AbstractPhysicalSpace](http://docs.ogc.org/DRAFTS/20-010.html#AbstractPhysicalSpace-section) «FeatureType» | AbstractPhysicalSpace is the abstract superclass for all types of physical spaces. Physical space refers to spaces that are fully or partially bounded by physical objects. |
| [AbstractPointCloud](http://docs.ogc.org/DRAFTS/20-010.html#AbstractPointCloud-section) «FeatureType» | AbstractPointCloud is the abstract superclass to represent PointCloud objects. |
| [AbstractSpace](http://docs.ogc.org/DRAFTS/20-010.html#AbstractSpace-section) «FeatureType» | AbstractSpace is the abstract superclass for all types of spaces. A space is an entity of volumetric extent in the real world. |
| [AbstractThematicSurface](http://docs.ogc.org/DRAFTS/20-010.html#AbstractThematicSurface-section) «FeatureType» | AbstractThematicSurface is the abstract superclass for all types of thematic surfaces. |
| [ImplicitGeometry](http://docs.ogc.org/DRAFTS/20-010.html#ImplicitGeometry-section) «ObjectType» | ImplicitGeometry is a geometry representation where the shape is stored only once as a prototypical geometry For example a tree or other vegetation object, a traffic light or a traffic sign. This prototypic geometry object can be re-used or referenced many times, wherever the corresponding feature occurs in the 3D city model. |

**8.2.6. CityGML Core Model**

The [City Model and City Object](http://docs.ogc.org/DRAFTS/20-010.html#city-objects-section) classes, the [Space Concept](http://docs.ogc.org/DRAFTS/20-010.html#space-concepts-section) classes, and the [Geometry and LOD](http://docs.ogc.org/DRAFTS/20-010.html#geometry-lod-section) classes define the majority of the CityGML Core module. In addition to these concepts, the Core module also specifies that city objects can have relations to other city objects and that they can have address information. All other modules defined in the CityGML model refer to the Core module.

The UML diagram of the complete Core module is depicted in [Figure 19](http://docs.ogc.org/DRAFTS/20-010.html#core-uml).

Figure 19. UML diagram of CityGML’s core module.

[Table 6](http://docs.ogc.org/DRAFTS/20-010.html#Core-city-model-class-table), [Table 7](http://docs.ogc.org/DRAFTS/20-010.html#Core-spatial-class-table), and [Table 8](http://docs.ogc.org/DRAFTS/20-010.html#Core-geometry-class-table) introduce already most of the classes of the CityGML Core module. The additional classes required complete this section of the standard are introduced in [Table 9](http://docs.ogc.org/DRAFTS/20-010.html#Core-class-table). More details about these classes can be found in the Data Dictionary in [Chapter 9](http://docs.ogc.org/DRAFTS/20-010.html#data-dictionary-section).

|  |  |
| --- | --- |
| Table 9. Additional Classes used in Core | |
| **Class** | **Description** |
| [Address](http://docs.ogc.org/DRAFTS/20-010.html#Address-section) «FeatureType» | Address represents an address of a city object. |
| [CityObjectRelation](http://docs.ogc.org/DRAFTS/20-010.html#CityObjectRelation-section) «ObjectType» | CityObjectRelation represents a specific relation from the city object in which the object is included to another city object. |

**8.2.7. Data types, Enumerations, and Code lists**

While FeatureTypes capture the real-world concepts in the CityGML Conceptual Model, they would be incomplete without the additional concepts from which they are made. These supporting constructs are illustrated in the following figures.

The ADE data types provided for in the Core module are illustrated in the figure [Figure 20](http://docs.ogc.org/DRAFTS/20-010.html#core-uml-ade-types).

Figure 20. ADE classes of the CityGML Core module.

The Basic Types, Enumerations, and Code Lists provided for the Core module are illustrated in the figure [Figure 21](http://docs.ogc.org/DRAFTS/20-010.html#core-uml-codelists).

Figure 21. Basic Types, Enumerations, and Codelists from the CityGML Core module.

These supporting constructs are defined in the following tables.

|  |  |
| --- | --- |
| Table 10. Data Types used in Core | |
| **Name** | **Description** |
| [AbstractGenericAttribute](http://docs.ogc.org/DRAFTS/20-010.html#AbstractGenericAttribute-section) «DataType» | AbstractGenericAttribute is the abstract superclass for all types of generic attributes. |
| [ADEOfAbstractAppearance](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractAppearance-section) «DataType» | ADEOfAbstractAppearance acts as a hook to define properties within an ADE that are to be added to AbstractAppearance. |
| [ADEOfAbstractCityObject](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractCityObject-section) «DataType» | ADEOfAbstractCityObject acts as a hook to define properties within an ADE that are to be added to AbstractCityObject. |
| [ADEOfAbstractDynamizer](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractDynamizer-section) «DataType» | ADEOfAbstractDynamizer acts as a hook to define properties within an ADE that are to be added to AbstractDynamizer. |
| [ADEOfAbstractFeature](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractFeature-section) «DataType» | ADEOfAbstractFeature acts as a hook to define properties within an ADE that are to be added to AbstractFeature. |
| [ADEOfAbstractFeatureWithLifespan](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractFeatureWithLifespan-section) «DataType» | ADEOfAbstractFeatureWithLifespan acts as a hook to define properties within an ADE that are to be added to AbstractFeatureWithLifespan. |
| [ADEOfAbstractLogicalSpace](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractLogicalSpace-section) «DataType» | ADEOfAbstractLogicalSpace acts as a hook to define properties within an ADE that are to be added to AbstractLogicalSpace. |
| [ADEOfAbstractOccupiedSpace](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractOccupiedSpace-section) «DataType» | ADEOfAbstractOccupiedSpace acts as a hook to define properties within an ADE that are to be added to AbstractOccupiedSpace. |
| [ADEOfAbstractPhysicalSpace](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractPhysicalSpace-section) «DataType» | ADEOfAbstractPhysicalSpace acts as a hook to define properties within an ADE that are to be added to AbstractPhysicalSpace. |
| [ADEOfAbstractPointCloud](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractPointCloud-section) «DataType» | ADEOfAbstractPointCloud acts as a hook to define properties within an ADE that are to be added to AbstractPointCloud. |
| [ADEOfAbstractSpace](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractSpace-section) «DataType» | ADEOfAbstractSpace acts as a hook to define properties within an ADE that are to be added to AbstractSpace. |
| [ADEOfAbstractSpaceBoundary](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractSpaceBoundary-section) «DataType» | ADEOfAbstractSpaceBoundary acts as a hook to define properties within an ADE that are to be added to AbstractSpaceBoundary. |
| [ADEOfAbstractThematicSurface](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractThematicSurface-section) «DataType» | ADEOfAbstractThematicSurface acts as a hook to define properties within an ADE that are to be added to AbstractThematicSurface. |
| [ADEOfAbstractUnoccupiedSpace](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractUnoccupiedSpace-section) «DataType» | ADEOfAbstractUnoccupiedSpace acts as a hook to define properties within an ADE that are to be added to AbstractUnoccupiedSpace. |
| [ADEOfAbstractVersion](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractVersion-section) «DataType» | ADEOfAbstractVersion acts as a hook to define properties within an ADE that are to be added to AbstractVersion. |
| [ADEOfAbstractVersionTransition](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractVersionTransition-section) «DataType» | ADEOfAbstractVersionTransition acts as a hook to define properties within an ADE that are to be added to AbstractVersionTransition. |
| [ADEOfAddress](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAddress-section) «DataType» | ADEOfAddress acts as a hook to define properties within an ADE that are to be added to an Address. |
| [ADEOfCityModel](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfCityModel-section) «DataType» | ADEOfCityModel acts as a hook to define properties within an ADE that are to be added to a CityModel. |
| [ADEOfClosureSurface](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfClosureSurface-section) «DataType» | ADEOfClosureSurface acts as a hook to define properties within an ADE that are to be added to a ClosureSurface. |
| [ExternalReference](http://docs.ogc.org/DRAFTS/20-010.html#ExternalReference-section) «DataType» | ExternalReference is a reference to a corresponding object in another information system, for example in the German cadastre (ALKIS), the German topographic information system (ATKIS), or the OS UK MasterMap®. |
| [Occupancy](http://docs.ogc.org/DRAFTS/20-010.html#Occupancy-section) «DataType» | Occupancy is an application-dependent indication of what is contained by a feature. |
| [QualifiedArea](http://docs.ogc.org/DRAFTS/20-010.html#QualifiedArea-section) «DataType» | QualifiedArea is an application-dependent measure of the area of a space or of a thematic surface. |
| [QualifiedVolume](http://docs.ogc.org/DRAFTS/20-010.html#QualifiedVolume-section) «DataType» | QualifiedVolume is an application-dependent measure of the volume of a space. |
| [XALAddress](http://docs.ogc.org/DRAFTS/20-010.html#XALAddress-section) «DataType» | XALAddress represents address details according to the OASIS xAL standard. |

|  |  |
| --- | --- |
| Table 11. Primitive Data Types used in Core | |
| **Name** | **Description** |
| [Code](http://docs.ogc.org/DRAFTS/20-010.html#Code-section) «BasicType» | Code is a basic type for a String-based term, keyword, or name that can additionally have a code space. |
| [DoubleBetween0and1](http://docs.ogc.org/DRAFTS/20-010.html#DoubleBetween0and1-section) «BasicType» | DoubleBetween0and1 is a basic type for values, which are greater or equal than 0 and less or equal than 1. The type is used for color encoding, for example. |
| [DoubleBetween0and1List](http://docs.ogc.org/DRAFTS/20-010.html#DoubleBetween0and1List-section) «BasicType» | DoubleBetween0and1List is a basic type that represents a list of double values greater or equal than 0 and less or equal than 1. The type is used for color encoding, for example. |
| [DoubleList](http://docs.ogc.org/DRAFTS/20-010.html#DoubleList-section) «BasicType» | DoubleList is an ordered sequence of double values. |
| [DoubleOrNilReasonList](http://docs.ogc.org/DRAFTS/20-010.html#DoubleOrNilReasonList-section) «BasicType» | DoubleOrNilReasonList is a basic type that represents a list of double values and/or nil reasons. |
| [ID](http://docs.ogc.org/DRAFTS/20-010.html#ID-section) «BasicType» | ID is a basic type that represents a unique identifier. |
| [IntegerBetween0and3](http://docs.ogc.org/DRAFTS/20-010.html#IntegerBetween0and3-section) «BasicType» | IntegerBetween0and3 is a basic type for integer values, which are greater or equal than 0 and less or equal than 3. The type is used for encoding the LOD number. |
| [MeasureOrNilReasonList](http://docs.ogc.org/DRAFTS/20-010.html#MeasureOrNilReasonList-section) «BasicType» | MeasureOrNilReasonList is a basic type that represents a list of double values and/or nil reasons together with a unit of measurement. |
| [TransformationMatrix2x2](http://docs.ogc.org/DRAFTS/20-010.html#TransformationMatrix2x2-section) «BasicType» | TransformationMatrix2x2 is a 2 by 2 matrix represented as a list of four double values in row major order. |
| [TransformationMatrix3x4](http://docs.ogc.org/DRAFTS/20-010.html#TransformationMatrix3x4-section) «BasicType» | TransformationMatrix3x4 is a 3 by 4 matrix represented as a list of twelve double values in row major order. |
| [TransformationMatrix4x4](http://docs.ogc.org/DRAFTS/20-010.html#TransformationMatrix4x4-section) «BasicType» | TransformationMatrix4x4 is a 4 by 4 matrix represented as a list of sixteen double values in row major order. |

|  |  |
| --- | --- |
| Table 12. Union types used in Core | |
| **Name** | **Description** |
| [CityModelMember](http://docs.ogc.org/DRAFTS/20-010.html#CityModelMember-section) «Union» | CityModelMember is a union type that enumerates the different types of objects that can occur as members of a city model. |
| [DoubleOrNilReason](http://docs.ogc.org/DRAFTS/20-010.html#DoubleOrNilReason-section) «Union» | DoubleOrNilReason is a union type that allows for choosing between a double value and a nil reason. |
| [NilReason](http://docs.ogc.org/DRAFTS/20-010.html#NilReason-section) «Union» | NilReason is a union type that allows for choosing between two different types of nil reason. |

|  |  |
| --- | --- |
| Table 13. Enumerated Classes used in Core | |
| **Name** | **Description** |
| [RelativeToTerrain](http://docs.ogc.org/DRAFTS/20-010.html#RelativeToTerrain-section) «Enumeration» | RelativeToTerrain enumerates the spatial relations of a city object relative to terrain in a qualitative way. |
| [RelativeToWater](http://docs.ogc.org/DRAFTS/20-010.html#RelativeToWater-section) «Enumeration» | RelativeToWater enumerates the spatial relations of a city object relative to the water surface in a qualitative way. |
| [SpaceType](http://docs.ogc.org/DRAFTS/20-010.html#SpaceType-section) «Enumeration» | SpaceType is an enumeration that characterises a space according to its closure properties. |

|  |  |
| --- | --- |
| Table 14. CodeList Classes used in Core | |
| **Name** | **Description** |
| [IntervalValue](http://docs.ogc.org/DRAFTS/20-010.html#IntervalValue-section) «CodeList» | IntervalValue is a code list used to specify a time period. |
| [MimeTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#MimeTypeValue-section) «CodeList» | MimeTypeValue is a code list used to specify the MIME type of a referenced resource. |
| [NilReasonEnumeration](http://docs.ogc.org/DRAFTS/20-010.html#NilReasonEnumeration-section) «CodeList» | NilReasonEnumeration is a code list that enumerates the different nil reasons. |
| [OccupantTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#OccupantTypeValue-section) «CodeList» | OccupantTypeValue is a code list used to classify occupants. |
| [OtherRelationTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#OtherRelationTypeValue-section) «CodeList» | OtherRelationTypeValue is a code list used to classify other types of city object relations. |
| [QualifiedAreaTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#QualifiedAreaTypeValue-section) «CodeList» | QualifiedAreaTypeValue is a code list used to specify area types. |
| [QualifiedVolumeTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#QualifiedVolumeTypeValue-section) «CodeList» | QualifiedVolumeTypeValue is a code list used to specify volume types. |
| [RelationTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#RelationTypeValue-section) «CodeList» | RelationTypeValue is a code list used to classify city object relations. |
| [TemporalRelationTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#TemporalRelationTypeValue-section) «CodeList» | TemporalRelationTypeValue is a code list used to classify temporal city object relations. |
| [TopologicalRelationTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#TopologicalRelationTypeValue-section) «CodeList» | TopologicalRelationTypeValue is a code list used to classify topological city object relations. |

**8.2.8. Additional Information**

A detailed discussion of the CityGML Core can be found in the [OGC CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

**8.3. Appearance**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-appearance> | |
| Target type | Implementation Standard (IS) |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The Appearance module provides the representation of surface data such as observable properties for surface geometry objects in the form of textures and material.

Appearances are not limited to visual data but represent arbitrary categories called themes such as infrared radiation, noise pollution, or earthquake-induced structural stress. A single surface geometry object may have surface data for multiple themes. Similarly, surface data can be shared by multiple surface geometry objects (e.g. road paving).

Surface data that is constant across a surface is modelled as material based on the material definitions from the X3D and COLLADA standards. Surface data that depends on the exact location within the surface is modelled as a texture. This can either be a parameterized texture(a texture that uses texture coordinates) or a transformation matrix for parameterization, or a georeferenced texture (a texture that uses a planimetric projection).

Each surface geometry object can have both, a material and a texture per theme and side. This allows for providing a constant approximation and a complex measurement of a surface’s property simultaneously.

The UML diagram of the Appearance module is illustrated in [Figure 22](http://docs.ogc.org/DRAFTS/20-010.html#appearance-uml). A detailed discussion of this Requirements Class can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

Figure 22. UML diagram of CityGML’s Appearance model.

The ADE data types provided for the Appearance module are illustrated in the figure [Figure 23](http://docs.ogc.org/DRAFTS/20-010.html#appearance-uml-ade-types).

Figure 23. ADE classes of the CityGML Appearance Module.

**8.3.1. Requirements**

The following requirement defines the rules governing implementation of the CityGML Appearance Module as an Implementation Specification (IS).

|  |  |
| --- | --- |
| **Requirement 5** | **/req/appearance/classes** |
| For each UML class defined or referenced in the Appearance Package: | |
| A | The IS SHALL contain an element which represents the same concept as that defined for the UML class. |
| B | The IS SHALL represent associations with the same source, target, direction, roles, and multiplicities as those of the UML class. |
| C | The IS SHALL represent the attributes of the UML class including the name, definition, type, and multiplicity. |
| D | The IS SHALL represent the attributes of all superclasses of the UML class including the name, definition, type, and mutiplicity. |
| E | The IS SHALL represent the associations of all superclasses of the UML class including the source, target, direction, roles, and multiplicity. |
| F | The IS SHALL specify how it observes all constraints the Conceptual Model imposes on the UML class. |

The use of extension capabilities by Appearance elements is constrained by the following requirement:

|  |  |
| --- | --- |
| **Requirement 6** | **/req/appearance/ade/use** |
| ADE element and property extensions SHALL NOT be used unless conformance with the ADE Requirements Class can be demonstrated. | |

**8.3.2. Class Definitions**

|  |  |
| --- | --- |
| Table 15. Classes used in Appearance | |
| **Class** | **Description** |
| [AbstractSurfaceData](http://docs.ogc.org/DRAFTS/20-010.html#AbstractSurfaceData-section) «FeatureType» | AbstractSurfaceData is the abstract superclass for different kinds of textures and material. |
| [AbstractTexture](http://docs.ogc.org/DRAFTS/20-010.html#AbstractTexture-section) «FeatureType» | AbstractTexture is the abstract superclass to represent the common attributes of the classes ParameterizedTexture and GeoreferencedTexture. |
| [Appearance](http://docs.ogc.org/DRAFTS/20-010.html#Appearance-section) «FeatureType» | An Appearance is a collection of surface data, i.e. observable properties for surface geometry objects in the form of textures and material. |
| [GeoreferencedTexture](http://docs.ogc.org/DRAFTS/20-010.html#GeoreferencedTexture-section) «FeatureType» | A GeoreferencedTexture is a texture that uses a planimetric projection. It contains an implicit parameterization that is either stored within the image file, an accompanying world file or specified using the orientation and referencePoint elements. |
| [ParameterizedTexture](http://docs.ogc.org/DRAFTS/20-010.html#ParameterizedTexture-section) «FeatureType» | A ParameterizedTexture is a texture that uses texture coordinates or a transformation matrix for parameterization. |
| [X3DMaterial](http://docs.ogc.org/DRAFTS/20-010.html#X3DMaterial-section) «FeatureType» | X3DMaterial defines properties for surface geometry objects based on the material definitions from the X3D and COLLADA standards. |
| [TextureAssociation](http://docs.ogc.org/DRAFTS/20-010.html#TextureAssociation-section) «ObjectType» | TextureAssociation denotes the relation of a texture to a surface geometry object. |

|  |  |
| --- | --- |
| Table 16. Data Types used in Appearance | |
| **Name** | **Description** |
| [AbstractTextureParameterization](http://docs.ogc.org/DRAFTS/20-010.html#AbstractTextureParameterization-section) «DataType» | AbstractTextureParameterization is the abstract superclass for different kinds of texture parameterizations. |
| [ADEOfAbstractSurfaceData](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractSurfaceData-section) «DataType» | ADEOfAbstractSurfaceData acts as a hook to define properties within an ADE that are to be added to AbstractSurfaceData. |
| [ADEOfAbstractTexture](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractTexture-section) «DataType» | ADEOfAbstractTexture acts as a hook to define properties within an ADE that are to be added to AbstractTexture. |
| [ADEOfAppearance](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAppearance-section) «DataType» | ADEOfAppearance acts as a hook to define properties within an ADE that are to be added to an Appearance. |
| [ADEOfGeoreferencedTexture](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfGeoreferencedTexture-section) «DataType» | ADEOfGeoreferencedTexture acts as a hook to define properties within an ADE that are to be added to a GeoreferencedTexture. |
| [ADEOfParameterizedTexture](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfParameterizedTexture-section) «DataType» | ADEOfParameterizedTexture acts as a hook to define properties within an ADE that are to be added to a ParameterizedTexture. |
| [ADEOfX3DMaterial](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfX3DMaterial-section) «DataType» | ADEOfX3DMaterial acts as a hook to define properties within an ADE that are to be added to an X3DMaterial. |
| [TexCoordGen](http://docs.ogc.org/DRAFTS/20-010.html#TexCoordGen-section) «DataType» | TexCoordGen defines texture parameterization using a transformation matrix. |
| [TexCoordList](http://docs.ogc.org/DRAFTS/20-010.html#TexCoordList-section) «DataType» | TexCoordList defines texture parameterization using texture coordinates. |

|  |  |
| --- | --- |
| Table 17. Primitive Data Types used in Appearance | |
| **Name** | **Description** |
| [Color](http://docs.ogc.org/DRAFTS/20-010.html#Color-section) «BasicType» | Color is a list of three double values between 0 and 1 defining an RGB color value. |
| [ColorPlusOpacity](http://docs.ogc.org/DRAFTS/20-010.html#ColorPlusOpacity-section) «BasicType» | Color is a list of four double values between 0 and 1 defining an RGBA color value. Opacity value of 0 means transparent. |

|  |  |
| --- | --- |
| Table 18. Enumerated Classes used in Appearance | |
| **Name** | **Description** |
| [TextureType](http://docs.ogc.org/DRAFTS/20-010.html#TextureType-section) «Enumeration» | TextureType enumerates the different texture types. |
| [WrapMode](http://docs.ogc.org/DRAFTS/20-010.html#WrapMode-section) «Enumeration» | WrapMode enumerates the different fill modes for textures. |

**8.3.3. Additional Information**

Additional information about the Appearance Module can be found in the [OGC CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html)

<<SNIP, SNIP>>

**8.4. City Furniture**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-cityfurniture> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The CityFurniture module provides the representation of objects or pieces of equipment that are installed in the outdoor environment for various purposes, such as decoration, explanation or control. City furniture objects are relatively small, immovable objects and usually are of stereotypical form. Examples include road signs, traffic signals, bicycle racks, street lamps, fountains, flower buckets, advertising columns, and benches.

City furniture is represented in the UML model by the top-level feature type *CityFurniture*, which is also the only class of the CityFurniture module.

The UML diagram of the CityFurniture module is depicted in [Figure 24](http://docs.ogc.org/DRAFTS/20-010.html#cityfurniture-uml). A detailed discussion of this Requirements Class can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

<<SNIP, SNIP>>

**8.6. Dynamizer**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-dynamizer> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The Dynamizer module provides the concepts that enable representation oftime-varying data for city object properties as well as for integrating sensors with 3D city models. Dynamizers are objects that inject timeseries data for an individual attribute of the city object in which the timeseries data is included. In order to represent dynamic (time-dependent) variations of its value the timeseries data overrides the static value of the referenced city object attribute.

The dynamic values may be given by retrieving observation results directly from external sensor/IoT services using a sensor connection (e.g. OGC SensorThings API, Sensor Observation Service, or other sensor data platforms including MQTT). Alternatively, the dynamic values may be provided as atomic timeseries that represent time-varying data of a specific data type for a single contiguous time interval. The data can be provided in external tabulated files, such as CSV or Excel sheets, in external files that format timeseries data according to the OGC TimeseriesML Standard or the OGC Observations & Measurements standards, or inline as embedded time-value-pairs. Furthermore, timeseries data can also be aggregated to form composite timeseries with non-overlapping time intervals.

By using the Dynamizer module, fast changes over a short or longer time period with respect to cities and city models can be represented. This includes variations of spatial properties, such as change of a feature’s geometry, both in respect to shape and to location (e.g. moving objects), variations of thematic attributes, changes of physical quantities like energy demands, temperatures, solar irradiation, traffic density, pollution concentration, or overpressure on building walls, and variations with respect to sensor or real-time data resulting from simulations or measurements.

The UML diagram of the Dynamizer module is depicted in [Figure 29](http://docs.ogc.org/DRAFTS/20-010.html#dynamizer-uml). A detailed discussion of this Requirements Class can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

The use of extension capabilities by Dynamizer elements is constrained by the following requirement:

|  |  |
| --- | --- |
| **Requirement 14** | **/req/dynamizer/ade/use** |
| ADE element and property extensions SHALL NOT be used unless conformance with the ADE Requirements Class can be demonstrated. | |

**8.6.2. Class Definitions**

|  |  |
| --- | --- |
| Table 27. Classes used in Dynamizer | |
| **Class** | **Description** |
| [AbstractAtomicTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#AbstractAtomicTimeseries-section) «FeatureType» | AbstractAtomicTimeseries represents the attributes and relationships that are common to all kinds of atomic timeseries (GenericTimeseries, TabulatedFileTimeseries, StandardFileTimeseries). An atomic timeseries represents time-varying data of a specific data type for a single contiguous time interval. |
| [AbstractTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#AbstractTimeseries-section) «FeatureType» | AbstractTimeseries is the abstract superclass representing any type of timeseries data. |
| [CompositeTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#CompositeTimeseries-section) «FeatureType» | A CompositeTimeseries is a (possibly recursive) aggregation of atomic and composite timeseries. The components of a composite timeseries must have non-overlapping time intervals. |
| [Dynamizer](http://docs.ogc.org/DRAFTS/20-010.html#Dynamizer-section) «FeatureType» | A Dynamizer is an object that injects timeseries data for an individual attribute of the city object in which it is included. The timeseries data overrides the static value of the referenced city object attribute in order to represent dynamic (time-dependent) variations of its value. |
| [GenericTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#GenericTimeseries-section) «FeatureType» | A GenericTimeseries represents time-varying data in the form of embedded time-value-pairs of a specific data type for a single contiguous time interval. |
| [StandardFileTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#StandardFileTimeseries-section) «FeatureType» | A StandardFileTimeseries represents time-varying data for a single contiguous time interval. The data is provided in an external file referenced in the StandardFileTimeseries. The data within the external file shall be encoded according to a dedicated format for the representation of timeseries data such as using the OGC TimeseriesML or OGC Observations & Measurements Standards. The data type of the data has to be specified within the external file. |
| [TabulatedFileTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#TabulatedFileTimeseries-section) «FeatureType» | A TabulatedFileTimeseries represents time-varying data of a specific data type for a single contiguous time interval. The data is provided in an external file referenced in the TabulatedFileTimeseries. The file shall contain table structured data using an appropriate file format such as comma-separated values (CSV), Microsoft Excel (XLSX) or Google Spreadsheet. The timestamps and the values are given in specific columns of the table. Each row represents a single time-value-pair. A subset of rows can be selected using the idColumn and idValue attributes. |

|  |  |
| --- | --- |
| Table 28. Data Types used in Dynamizer | |
| **Name** | **Description** |
| [ADEOfAbstractAtomicTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractAtomicTimeseries-section) «DataType» | ADEOfAbstractAtomicTimeseries acts as a hook to define properties within an ADE that are to be added to AbstractAtomicTimeseries. |
| [ADEOfAbstractTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfAbstractTimeseries-section) «DataType» | ADEOfAbstractTimeseries acts as a hook to define properties within an ADE that are to be added to AbstractTimeseries. |
| [ADEOfCompositeTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfCompositeTimeseries-section) «DataType» | ADEOfCompositeTimeseries acts as a hook to define properties within an ADE that are to be added to a CompositeTimeseries. |
| [ADEOfDynamizer](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfDynamizer-section) «DataType» | ADEOfDynamizer acts as a hook to define properties within an ADE that are to be added to a Dynamizer. |
| [ADEOfGenericTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfGenericTimeseries-section) «DataType» | ADEOfGenericTimeseries acts as a hook to define properties within an ADE that are to be added to a GenericTimeseries. |
| [ADEOfStandardFileTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfStandardFileTimeseries-section) «DataType» | ADEOfStandardFileTimeseries acts as a hook to define properties within an ADE that are to be added to a StandardFileTimeseries. |
| [ADEOfTabulatedFileTimeseries](http://docs.ogc.org/DRAFTS/20-010.html#ADEOfTabulatedFileTimeseries-section) «DataType» | ADEOfTabulatedFileTimeseries acts as a hook to define properties within an ADE that are to be added to a TabulatedFileTimeseries. |
| [SensorConnection](http://docs.ogc.org/DRAFTS/20-010.html#SensorConnection-section) «DataType» | A SensorConnection provides all details that are required to retrieve a specific datastream from an external sensor web service. It comprises the service type (e.g. OGC SensorThings API, OGC Sensor Observation Services, MQTT, proprietary platforms), the URL of the sensor service, the identifier for the sensor or thing, and its observed property as well as information about the required authentication method. |
| [TimeseriesComponent](http://docs.ogc.org/DRAFTS/20-010.html#TimeseriesComponent-section) «DataType» | TimeseriesComponent represents an element of a CompositeTimeseries. |
| [TimeValuePair](http://docs.ogc.org/DRAFTS/20-010.html#TimeValuePair-section) «DataType» | A TimeValuePair represents a value that is valid for a given timepoint. For each TimeValuePair, only one of the value properties can be used mutually exclusive. Which value property has to be provided depends on the selected value type in the GenericTimeSeries feature, in which the TimeValuePair is included. |

|  |  |
| --- | --- |
| Table 29. Enumerated Classes used in Dynamizer | |
| **Name** | **Description** |
| [TimeseriesTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#TimeseriesTypeValue-section) «Enumeration» | TimeseriesTypeValue enumerates the possible value types for GenericTimeseries and TimeValuePair. |

|  |  |
| --- | --- |
| Table 30. CodeList Classes used in Dynamizer | |
| **Name** | **Description** |
| [AuthenticationTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#AuthenticationTypeValue-section) «CodeList» | AuthenticationTypeValue is a code list used to specify the authentication method to be used to access the referenced sensor service. Each value shall provide enough information such that a software application could determine the required access credentials. |
| [SensorConnectionTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#SensorConnectionTypeValue-section) «CodeList» | SensorConnectionTypeValue is a code list used to specify the type of the referenced sensor service. Each value shall provide enough information such that a software application would be able to identify the API type and version. |
| [StandardFileTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#StandardFileTypeValue-section) «CodeList» | StandardFileTypeValue is a code list used to specify the type of the referenced external timeseries data file. Each value shall provide information about the standard and version. |
| [TabulatedFileTypeValue](http://docs.ogc.org/DRAFTS/20-010.html#TabulatedFileTypeValue-section) «CodeList» | TabulatedFileTypeValue is a code list used to specify the data format of the referenced external tabulated data file. |

**8.6.3. Additional Information**

Additional information about the Dynamizer Module can be found in the [OGC CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html)

**8.7. Generics**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-generics> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The Generics module provides the representation of generic city objects. These are city objects that are not covered by any explicitly modelled thematic class within the CityGML CM. Generics module also provides the representation of generic attributes, which are attributes that are not explicitly represented in the CityGML CM. In order to avoid problems concerning semantic interoperability, generic city objects and generic attributes shall only be used if appropriate thematic classes and attributes are not provided by any other CityGML module.

In accordance with the CityGML Space concept defined in the Core module [(cf. Section Core)](http://docs.ogc.org/DRAFTS/20-010.html#rc_core_section) generic city objects can be represented as generic logical spaces, generic occupied spaces, generic unoccupied spaces, and generic thematic surfaces. In this way, spaces and surfaces can be defined that are not represented by any explicitly modelled class within CityGML that is a subclass of the classes AbstractLogicalSpace, AbstractOccupiedSpace, AbstractUnoccupiedSpace or AbstractThematicSurface, respectively. Generic city objects are represented in the UML model by the top-level feature types *GenericLogicalSpace*, *GenericOccupiedSpace*, *GenericUnoccupiedSpace* and *GenericThematicSurface*.

Generic attributes are defined as name-value pairs and are always associated with a city object. Generic attributes can be of type String, Integer, Double, Date, URI, Measure, and Code. In addition, generic attributes can be grouped under a common name as generic attribute sets.

The UML diagram of the Generics module is depicted in [Figure 32](http://docs.ogc.org/DRAFTS/20-010.html#generics-uml). A detailed discussion of this Requirements Class can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

<<SNIP, SNIP>>

**8.8. Land Use**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-landuse> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The LandUse module defines objects that can be used to describe areas of the earth’s surface dedicated to a specific land use or having a specific land cover with or without vegetation, such as sand, rock, mud flats, forest, grasslands, or wetlands (i.e. the physical appearance). Land use and land cover are different concepts: The first describes human activities on the earth’s surface, the second describes its physical and biological cover. However, the two concepts are interlinked and often mixed in practice. Land use objects in CityGML support both concepts: They can be employed to represent parcels, spatial planning objects, recreational objects, and objects describing the physical characteristics of an area in 3D. Land use objects are represented in the UML model by the top-level feature type *LandUse*, which is also the only class of the LandUse module.

The UML diagram of the LandUse module is depicted in [Figure 35](http://docs.ogc.org/DRAFTS/20-010.html#landuse-uml). A detailed discussion of this Requirements Class can be found in the [CityGML User Guide](https://github.com/opengeospatial/CityGML3-Workspace/blob/master/19-072UG.html#bp_landuse_section).

<<SNIP, SNIP>>

**8.9. Point Cloud**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-pointcloud> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The PointCloud specifies how to encode the geometry of physical spaces and of thematic surfaces as 3D point clouds. In this way, the building hull, a room within a building or a single wall surface can be spatially represented by a point cloud only. The same applies to all other thematic feature types including transportation objects, vegetation, city furniture, etc. Point clouds can either be provided inline within a CityGML file or as reference to external point cloud files of common file types such as LAS or LAZ. Point clouds are represented in the UML model by the feature type *PointCloud*, which is also the only class of the PointCloud module.

The UML diagram of the PointCloud module is depicted in [Figure 38](http://docs.ogc.org/DRAFTS/20-010.html#pointcloud-uml). A detailed discussion of this Requirements Class can be found in the [CityGML User Guide](https://github.com/opengeospatial/CityGML3-Workspace/blob/master/19-072UG.html#bp_pointcloud_section).

<<SNIP, SNIP>>

**8.10. Relief**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-relief> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The Relief module provides the representation of terrain which is an essential part of city models. In CityGML, the terrain is modelled by relief features. They are represented in the UML model by the top-level feature type *ReliefFeature*, which is the main class of the Relief module. The relief features, in turn, are collections of relief components that describe the Earth’s surface, also known as the Digital Terrain Model. The relief components can have different terrain representations which can coexist. Each relief component may be specified as a regular raster or grid, as a TIN (Triangulated Irregular Network), by break lines, or by mass points. In addition, the validity of the relief components may be restricted to certain areas.

The UML diagram of the Relief module is depicted in [Figure 40](http://docs.ogc.org/DRAFTS/20-010.html#relief-uml). A detailed discussion of this Requirements Class can be found in the [CityGML User Guide](https://github.com/opengeospatial/CityGML3-Workspace/blob/master/19-072UG.html#bp_relief_section).

<<SNIP, SNIP>>

**8.11. Transportation**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-transportation> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The Transportation module defines central elements of the traffic infrastructure. This includes the transportation objects road, track, and square for the movement of vehicles, bicycles, and pedestrians, the transportation object railway for the movement of wheeled vehicles on rails, as well as the transportation object waterway for the movement of vessels upon or within water bodies. The transportation objects are represented in the UML model by the top-level feature types *Road*, *Track*, *Square*, *Railway*, and *Waterway*, which are the main classes of the Transportation module. Transportation objects can be subdivided into sections, which can be regular road, track or railway legs, into intersection areas, and into roundabouts.

For each transportation object, traffic spaces and auxiliary traffic spaces can be provided, which are bounded at the bottom by traffic areas and auxiliary traffic areas, respectively. Traffic areas are elements that are important in terms of traffic usage, such as driving lanes, sidewalks, and cycle lanes, whereas auxiliary traffic areas describe further elements, such as kerbstones, middle lanes, and green areas. The corresponding spaces define the free space above the areas. In addition, each traffic space can have an optional clearance space. The transportation objects can be represented in different levels of granularity, either as a single area, split up into individual lanes or even decomposed into individual (carriage)ways. Furthermore, holes in the surfaces of roads, tracks or squares, such as road damages, manholes or drains, can be represented including their corresponding boundary surfaces. In addition, markings for the structuring or restriction of traffic can be added to the transportation areas. Examples are road markings and markings related to railway or waterway traffic.

The UML diagram of the Transportation module is depicted in [Figure 42](http://docs.ogc.org/DRAFTS/20-010.html#transportation-uml). A detailed discussion of this Requirements Class can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

<<SNIP, SNIP>>

**8.13. Versioning**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-versioning> | |
| Target type | Implementation Specification |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

The Versioning module defines the concepts that enable encoding representing multiple versions of a city model. A specific version represents a defined state of a city model consisting of the dedicated versions of all city object instances that belong to the respective city model version. Each version can be complemented by version transitions that describe the change of the state of a city model from one version to another and that give the reason for the change and the modifications applied. In addition, the Versioning module introduces bi-temporal timestamps for all objects. This allows for providing all objects with information on 1) the time period a specific version of an object is an integral part of the 3D city model and 2) the lifespan a specific version of an object exists in the real world.

By using the Versioning module, slow changes over a long time period with respect to cities and city models can be represented. This includes the creation and termination of objects (e.g. construction or demolition of sites, planting of trees, construction of new roads), structural changes of objects (e.g. raising of buildings), and changes in the status of an object (e.g. change of building owner, change of the traffic direction of a road to a one-way street). In this way, the history or evolution of cities and city models can be modelled, parallel or alternative versions of cities and city models can be managed, and changes of geometries and thematic properties of individual city objects over time can be tracked.

The UML diagram of the Versioning module is depicted in [Figure 48](http://docs.ogc.org/DRAFTS/20-010.html#versioning-uml). A detailed discussion of this Requirements Class can be found in the [CityGML User Guide](https://github.com/opengeospatial/CityGML3-Workspace/blob/master/19-072UG.html#bp_versioning_section).

<<SNIP, SNIP>>

**8.15. Construction**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-construction> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |
| Dependency | [/req/req-class-generics](http://docs.ogc.org/DRAFTS/20-010.html#rc_generics) |

The Construction module defines concepts that are common to all forms of constructions. Constructions are objects that are manufactured by humans from construction materials, are connected to earth, and are intended to be permanent. The Construction module focuses on as-built representations of constructions and integrates all concepts that are similar over different types of constructions, in particular buildings, bridges, and tunnels. In addition, for representing man-made structures that are neither buildings, nor bridges, nor tunnels so-called other constructions (e.g. large chimneys or city walls) can be defined.

Furniture, installations, and constructive elements are further concepts that are defined in the Construction module. Installations are permanent parts of a construction that strongly affect the outer or inner appearance of the construction and that cannot be moved (e.g. balconies, chimneys, or stairs), whereas furniture represent moveable objects of a construction (e.g. tables and chairs). Constructive elements allow for decomposing a construction into volumetric components, such as walls, beams, and slabs. Constructions and constructive elements can be bounded by different types of surfaces. In this way, the outer structure of constructions and constructive elements can be differentiated semantically into wall surfaces, roof surfaces, ground surfaces, outer floor surfaces, and outer ceiling surfaces, whereas the visible surface of interior spaces can be structured into interior wall surfaces, floor surfaces, and ceiling surfaces. Furthermore, the openings of constructions, i.e. windows and doors, can be represented as so-called filling elements including their corresponding filling surfaces.

The UML diagram of the Construction module is depicted in [Figure 53](http://docs.ogc.org/DRAFTS/20-010.html#construction-uml). The Construction module defines concepts that are inherited and, where necessary, are specialized by the modules Building, Bridge, and Tunnel (cf. [Section 8.17](http://docs.ogc.org/DRAFTS/20-010.html#rc_building-model_section), [Section 8.16](http://docs.ogc.org/DRAFTS/20-010.html#rc_bridge-model_section), and [Section 8.18](http://docs.ogc.org/DRAFTS/20-010.html#rc_tunnel_section)). A detailed discussion of the Requirements Class Construction can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

**8.17. Building**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-building> | |
| Target type | Implementation Standard |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |
| Dependency | [/req/req-class-construction](http://docs.ogc.org/DRAFTS/20-010.html#rc_construction) |

The Building module provides the representation of thematic and spatial aspects of buildings. Buildings are free-standing, self-supporting constructions that are roofed and usually walled, and that can be entered by humans and are normally designed to stand permanently in one place. Buildings are intended for human occupancy (e.g. a place of work or recreation), habitation and/or shelter of humans, animals or things. Buildings are represented in the UML model by the top-level feature type *Building*, which is the main class of the Building module. Buildings can physically or functionally be subdivided into building parts, and logically into storeys and building units (e.g. apartments). In addition, buildings can be decomposed into structural elements, such as walls, slabs, staircases, and beams.

The interior of buildings is represented by rooms. This allows a virtual accessibility of buildings, such as for visitor information in a museum (“Location Based Services“), the examination of accommodation standards, or the presentation of daylight illumination of a building. Buildings can contain installations and furniture. Installations are permanent parts of a building that strongly affect the outer or inner appearance of the building and that cannot be moved. Examples are balconies, chimneys, dormers or stairs. In contrast, furniture represents moveable objects inside a building, like tables and chairs. Buildings can be bounded by different types of surfaces. In this way, the outer façade of buildings can be differentiated semantically into wall surfaces, roof surfaces, ground surfaces, outer floor surfaces, and outer ceiling surfaces, whereas the visible surface of rooms can be structured into interior wall surfaces, floor surfaces, and ceiling surfaces. Furthermore, the openings of buildings, i.e. windows and doors, can be represented including their corresponding surfaces.

The UML diagram of the building module is depicted in [Figure 59](http://docs.ogc.org/DRAFTS/20-010.html#building-uml). The Building module inherits concepts from the Construction module (cf. [Section 8.15](http://docs.ogc.org/DRAFTS/20-010.html#rc_construction_section)). The Construction module defines objects that are common to all types of construction, such as the different surface types and the openings. A detailed discussion of the Requirements Class Building can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

<<SNIP, SNIP>>

**8.18. Tunnel**

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-tunnel> | |
| Target type | Implementation Specification |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |
| Dependency | [/req/req-class-construction](http://docs.ogc.org/DRAFTS/20-010.html#rc_construction) |

The Tunnel module provides the representation of thematic and spatial aspects of tunnels. Tunnels are horizontal or sloping enclosed passage ways of a certain length, mainly underground or underwater. Tunnels are intended for passing obstacles such as mountains, waterways or other traffic routes by humans, animals or things. Tunnels are represented in the UML model by the top-level feature type *Tunnel*, which is the main class of the Tunnel module. Tunnels can physically or functionally be subdivided into tunnel parts. In addition, tunnels can be decomposed into structural elements, such as walls, slabs, staircases, and beams.

The interior of tunnels is represented by hollow spaces. This allows a virtual accessibility of tunnels, such as for driving through a tunnel, for simulating disaster management, or for presenting the light illumination within a tunnel. Tunnels can contain installations and furniture. Installations are permanent parts of a tunnel that strongly affect the outer or inner appearance of the tunnel and that cannot be moved. Examples are stairs, railings, radiators or pipes. In contrast, furniture represents moveable objects inside a tunnel, like movable equipment in control areas. Tunnels can be bounded by different types of surfaces. In this way, the outer structure of tunnels can be differentiated semantically into wall surfaces, roof surfaces, ground surfaces, outer floor surfaces, and outer ceiling surfaces, whereas the visible surface of hollow spaces can be structured into interior wall surfaces, floor surfaces, and ceiling surfaces. Furthermore, the openings of tunnels, i.e. windows and doors, can be represented including their corresponding surfaces.

The UML diagram of the Tunnel module is depicted in [Figure 62](http://docs.ogc.org/DRAFTS/20-010.html#tunnel-uml). The Tunnel module inherits concepts from the Construction module (cf. [Section 8.15](http://docs.ogc.org/DRAFTS/20-010.html#rc_construction_section)). The Construction module defines objects that are common to all types of construction, such as the different surface types and the openings. A detailed discussion of the Requirements Class Tunnel can be found in the [CityGML 3.0 Users Guide](http://docs.opengeospatial.org/DRAFTS/20-066.html).

<<SNIP, SNIP>>

**10. Application Domain Extension (ADE)**

An *Application Domain Extension* (ADE) is a formal and systematic extension of the CityGML Conceptual Model (CM) for a specific application or domain. The ADE is expressed in the form of a UML model. The domain data is mapped to a set of additional classes, attributes, and relations. ADEs may use elements from the CityGML CM to derive application-specific subclasses, to inject additional properties, to associate application data with predefined CityGML content, or to define value domains for attributes.

The ADE mechanism allows application-specific information to be aligned with the CityGML CM) in a well-structured and systematic way. By this means, CityGML can be extended to meet the information needs of an application while at the same time preserving its concepts and semantic structures. Moreover, and in contrast to generic city objects and attributes, application data can be validated against the formal definition of an ADE to ensure semantic interoperability.

Previous versions of the CityGML Standard defined the ADE mechanism solely at the level of the XML Schema encoding. With CityGML 3.0, ADEs become platform-independent models at a conceptual level that can be mapped to multiple and different target encodings.

ADEs have successfully been implemented in practice and enable a wide range of applications and use cases based on the CityGML Standard. An overview and discussion of existing ADEs is provided in [[Biljecki et al. 2018](http://docs.ogc.org/DRAFTS/20-010.html#Biljecki2018)].

**10.1. General Rules for ADEs**

An ADE shall be defined as a conceptual model documented in UML in accordance with the ISO 19100 conceptual modelling framework and by adhering to the General Feature Model and the rules and constraints for application schemas as specified in ISO 19109 and ISO/TS 19103. The [UML notations and stereotypes](http://docs.ogc.org/DRAFTS/20-010.html#uml_notation_section) used in the CityGML conceptual model should also be applied to corresponding model elements in an ADE.

Every ADE shall be organized into one or more UML packages having globally unique namespaces and containing all UML model elements defined by the ADE. An ADE may additionally import and use predefined classes from external conceptual UML models such as the CityGML modules or the standardized schemas of the ISO 19100 series of International Standards.

**10.2. Defining New ADE Model Elements**

Following ISO 19109, the primary view of geospatial information and the core elements of application schemas is the *feature*. ADEs therefore typically extend CityGML by defining new feature types appropriate to the application area together with additional content such as object types, data types, code lists, and enumerations.

Every feature type in an ADE is derived either directly or indirectly from either the CityGML root feature type *Core::AbstractFeature* or, depending on its type and characteristics, from a more appropriate subclass thereof. According to the general CityGML space concept, features representing spaces or space boundaries are derived either directly or indirectly from *Core::AbstractSpace* or *Core::AbstractSpaceBoundary* respectively. UML classes representing top-level feature types shall use the *«TopLevelFeatureType»* stereotype.

In contrast to feature types, object types and data types are not required to be derived from a predefined CityGML class unless explicitly stated otherwise.

ADE classes may have an unlimited number of attributes and associations in addition to those inherited from their parents. Attributes can be modelled with either simple or complex data types. To ensure semantic interoperability, the predefined types from CityGML or the standardized schemas of the ISO 19100 series of International Standards should be used wherever appropriate. This includes, among others, basic types from ISO/TS 19103, geometry and topology objects from ISO 10107, and temporal geometry and topology objects from ISO 19108.

If a predefined type is not available, ADEs can either define their own data types or import data types from external conceptual models. This explicitly includes the possibility of defining new geometry types not offered by ISO 19107. Designers of an ADE should however note that software might not be able to properly identify and consume such geometry types.

A feature type capturing a real-world feature with geometry should be derived either directly or indirectly from *Core::AbstractSpace* or *Core::AbstractSpaceBoundary*. By this means, the CityGML predefined spatial properties and the associated LOD concept are inherited and available for the feature type. If, however, these superclasses are either inappropriate or lack a spatial property required to represent the feature, an ADE may define new and additional spatial properties. If such a spatial property should belong to one of the predefined LODs, then the property name will start with the prefix “lod*X*”, where *X* is to be replaced by an integer value between 0 and 3 indicating the target LOD. This enables software to derive the LOD of the geometry.

Constraints on model elements should be expressed using a formal language such as the Object Constraint Language (OCL). The ADE specifies the manner of application of constraints. However, following the CityGML CM, constraints should at least be expressed on ADE subclasses of *Core::AbstractSpace* to limit the types of space boundaries (i.e., instances of *Core::AbstractSpaceBoundary*) that may be used to model the boundary of a space object.

Illustrative examples for ADEs can be found in the [CityGML 3.0 User Guide](http://docs.ogc.org/DRAFTS/20-010.html#_user_guide).

**10.3. Augmenting CityGML Feature Types with Additional ADE Properties**

If a predefined CityGML feature type lacks one or more properties required for a specific application, a feasible solution in CityGML 2.0 was to derive a new ADE feature type as subclass of the CityGML class and to add the properties to this subclass. While conceptually clean, this approach also faces drawbacks. If multiple ADEs require additional properties for the same CityGML feature type, this will lead to many subclasses of this feature type in different ADE namespaces. Information about the same real-world feature might therefore be spread over various instances of the different feature classes in an encoding making it difficult for software to consume the feature data.

For this reason, CityGML 3.0 provides a way to augment the predefined CityGML feature types with additional properties from the ADE domain without the need for subclassing. Each CityGML feature type has an extension attribute of name “adeOf*FeatureTypeName*” and type “*ADEOfFeatureTypeName”*, where *FeatureTypeName* is replaced by the class name in which the attribute is defined. For example, the *Building::Building* class offers the attribute *adeOfBuilding* of type *Building::ADEOfBuilding*. Each of these extension attributes can occur zero to unlimited times, and the attribute types are defined as abstract and empty data types.

If an ADE augments a specific CityGML feature type with additional ADE properties, the ADE will create a subclass of the corresponding abstract data type associated with the feature class. This subclass shall also be defined as data type using the stereotype *«DataType»*. The additional application-specific attributes and associations are then modelled as properties of the ADE subclass. This may include, among others, attributes with simple or complex data type, spatial properties or associations to other object and feature types from the ADE or external models such as CityGML.

The predefined “*ADEOfFeatureTypeName”* data types are called “hooks” because they are used as the head of a hierarchy of ADE subclasses attaching application-specific properties. When subclassing the “hook” of a specific CityGML feature type in an ADE, the properties defined in the subclass can be used for that feature type as well as for all directly or indirectly derived feature types, including feature types defined in the same or another ADE.

Multiple distinct ADEs can use the “hook” mechanism to define additional ADE properties for the same CityGML feature type. Since the “adeOf*FeatureTypeName*” attribute may occur multiple times, the various ADE properties can be exchanged as part of the same CityGML feature instance in an encoding. Software can therefore easily consume the default CityGML feature data plus the additional properties from the different ADEs.

Content from unknown or unsupported ADEs may be ignored by an application or service consuming an encoded CityGML model.

Designers of an ADE should favor using this “hook” mechanism over subclassing a CityGML feature type when possible. If an ADE must enable other ADEs to augment its own feature types (so-called ADE of an ADE), then it will implement “hooks” for its feature types following the same schema and naming concept as in the CityGML conceptual model.

The UML fragment in [Figure 66](http://docs.ogc.org/DRAFTS/20-010.html#figure-adeusage) shows an example for using the "hook" mechanism. For more details on this and other example ADEs, please see the [CityGML 3.0 User Guide](http://docs.ogc.org/DRAFTS/20-010.html#_user_guide) for an example ADE.

Figure 66. The CityGML feature type Building is augmented with additional ADE properties by defining the data type EnergyProperties as a subclass of the ADE data type ADEOfBuilding.

**10.4. Encoding of ADEs**

This document only addresses the conceptual modelling of ADEs. Rules and constraints for mapping a conceptual ADE model to a target encoding are expected to be defined in a corresponding CityGML Encoding Standard. If supported, an ADE may provide additional mapping rules and constraints in conformance with a corresponding CityGML Encoding Standard.

**10.5. Requirements and Recommendations**

The following requirements and recommendations specify how ADEs shall be used as an extension capability to the CityGML Conceptual Model.

|  |  |
| --- | --- |
| **Requirements Class** | |
| <http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-ade> | |
| Target type | Conceptual Model |
| Dependency | [/req/req-class-core](http://docs.ogc.org/DRAFTS/20-010.html#rc_core) |

**10.5.1. UML**

Any extension to the CityGML Conceptual Model should be a faithful continuation of the styles and techniques used in that model. The following Requirements and Recommendations define a "faithful continuation".

|  |  |
| --- | --- |
| **Requirement 48** | **/req/ade/uml** |
| An ADE SHALL be defined as conceptual model in UML in accordance with the conceptual modelling framework of the ISO 19100 series of International Standards | |
| A | The UML model SHALL adhere to the General Feature Model as specified in ISO 19109. |
| B | The UML model SHALL adhere to rules and constraints for application schemas as specified in ISO/TS 19103. |
| C | Every ADE SHALL be organized into one or more UML packages having globally unique namespaces and containing all UML model elements defined by the ADE. |

|  |  |
| --- | --- |
| **Recommendation 1** | **/rec/ade/uml** |
| In addition to meeting the requirements for a CityGML ADE, an ADE should: | |
| A | The [UML notations and stereotypes](http://docs.ogc.org/DRAFTS/20-010.html#uml_notation_section) used in the CityGML conceptual model SHOULD be applied to corresponding model elements in an ADE. |
| B | An ADE SHOULD import and use predefined classes from external conceptual UML models such as the CityGML modules or the standardized schemas of the ISO 19100 series of International Standards. |

**10.5.2. Classes**

The following Requirements and Recommendations define how CityGML classes should be extended by an ADE.

|  |  |
| --- | --- |
| **Requirement 49** | **/req/ade/elements** |
| ADEs typically extend CityGML by defining new Feature Types together with additional content such as Object Types, Data Types, Code Lists, and Enumerations. | |
| A | Every Feature Type in an ADE SHALL be derived either directly or indirectly from the CityGML root Feature Type *core:AbstractFeature* or a subclass thereof. |
| B | UML classes representing Top-Level Feature Types SHALL use the *«TopLevelFeatureType»* stereotype. |
| C | Features representing spaces or space boundaries SHALL be derived either directly or indirectly from *core:AbstractSpace* or *core:AbstractSpaceBoundary* respectively. |
| D | An ADE may define new and additional spatial properties. If such a spatial property should belong to a predefined LOD, then the property name SHALL start with the prefix “lod*X*”, where *X* is an integer value indicating the target LOD. |

|  |  |
| --- | --- |
| **Recommendation 50** | **/rec/ade/elements** |
| ADEs typically extend CityGML by defining new feature types together with additional content such as object types, data types, code lists, and enumerations. | |
| A | ADEs SHOULD use the predefined types from CityGML or the standardized schemas of the ISO 19100 series of International Standards. |
| B | Constraints on model elements SHOULD be expressed using a formal language such as the Object Constraint Language (OCL). |
| C | ADE subclasses of *core:AbstractSpace* SHOULD include constraints to limit the boundaries of the space object. |

**10.5.3. Properties**

The following Requirements define how to use the CityGML extension properties to add attributes to an existing CityGML Feature Type.

|  |  |
| --- | --- |
| **Requirement 51** | **/req/ade/properties** |
| Every Feature Type includes an extension property (hook) of type “ADEOf<FeatureTypeName>” where <FeatureTypeName> is the name of that Feature Type. To add an extension property to a Feature Type: | |
| A | The ADE SHALL create a subclass of the abstract data type associated with the hook. |
| B | This subclass SHALL be defined as a data type using the stereotype *«DataType»*. |
| C | Application-specific attributes and associations SHALL be modeled as properties of the ADE subclass. |