CodeList Design

Version 2019.05.29

This is the first draft of a document outlining the codelist design for CityGML 3.0. It is untended to gather together the considerations and decisions made by the Modelling Subgroup for the CityGML 3.0 Conceptual Model. I have a number of comments in the first person ("I") and I will clean this up eventually but I think it is helpful to know which things are my inventions or opinions rather than derived from examples or the literature. I realize that codelists are only a tiny bit of the overall specification but I think we can do a better job both in the use and the specification and provision of codelists. …steve smyth

CityGML 3.0

Requirement for codelists

CityGML 3.0 makes extensive use of lists of possible attribute values. Some of these are fixed lists that cannot be extended. Some may be replaced or extended by external resources identified by a URI. Some are suggested values with the possibility of additional valid values, possibly restricted by a required pattern match. Each of these possibilities might be identified as a *codelist*. This document

follows use cases for partially or fully prescribed codelists containing *string* attribute values addressed by CityGML 3.0. Corresponding codelist designs are considered for each.

Codelist values: use and definition

A codelist value embedded in an instance is a simple thing. Publication, curation, and use of codelist data from which values may be selected is more complex. This database aspect is often ignored in standard specifications but is an important factor in practical systems expecting interoperability between applications and codelist data sources.

Codelists have been used for attributes appearing in many object-oriented architectures. A seemingly simple concept has a wide range of design possibilities and interpretations. Although formal modelling, for example via UML, can partially specify the requirements on codelists, there are some aspects that are outside the expressive capability of UML.

Three bases have been reviewed to provide guidance for CityGML 3.0:

- OGC GML, including the design of LandInfra [OGC 15-111r1]
- INSPIRE [ChangesToD2.5andD2.7.docx , D2.8.I.9_v3.2]
- ISO [various].

Scope

A review of examples drawn from each were used to develop a taxonomy of codelists and to illustrate the design choices. There are some other questions outside the strict scope of codelists that must be answered in an environment supporting multiple encodings:

1. How are the codelist structure and any explicit values encoded? [Suggestion: use a representation natural to the encoding.]
2. Must any new encoding also provide a procedure for converting a codelist specification in an/any existing encoding to the new encoding? [Suggestion: yes.]
3. Should we support an API for conversion of codelists between encodings? [Suggestion: ??]

4. Can an application use a codelist defined in one encoding in an instance with a different encoding? [Suggestion: yes. This may well be a common occurrence, especially with codelists published by state entities.]

5. Must an application be prepared to accept external codelists in any supported encoding even though an instance is in a single specific encoding? [Suggestion: simplifies some things but makes other things more difficult.]

To frame the discussion, I have identified some characteristics of codelists that can be used to structure a discussion. These characteristics are derived from the low-level use cases that determine how a codelist is to be used and where the content comes from.

CityGML attribute value use cases

Codelists may be characterized by *mode, source,* and *inheritance*. This is my observation after looking at several codelists in practical use. A codelist design need not and probably should not appear in all twelve possible combinations. One of the consequential design choices is to specify which should be used.

Codelists have two *modes*:

1. A normative non-empty list of prescribed values ("normative list")
2. An informative list of valid values ("informative list")

Codelists have three *sources* of values

1. A fixed list defined in the model definition ("enumeration")
2. A pattern defining which of all possible strings are valid ("restriction")
3. An external source, pointed to by a URI, and whose authority must be depended on for the validity of its values. External sources must always be considered normative. ("external")

Related codelists ("dependent codelists") may be derived by *inheritance* from a common parent ("parent codelist")

1. Codelists may be independent ("flat")
2. Codelists may be in a generalization hierarchy ("hierarchical")

The structure (or lack of structure) of codelist values is an important additional characteristic in some domains but CityGML 3.0 is focused on simple lists of unstructured text codelist values.

Codelist types considered for CityGML 3.0

The following is an informal taxonomy of codelists that we have discussed or which I otherwise think might be suited for use in CityGML 3.0. They are described by the triple [*mode, source, inheritance*].

1. Enumeration: [*normative list, enumeration, flat*]
2. Hierarchical codelist: [*normative list, enumeration, hierarchical*]
    a. A top-level empty list with one or more dependent codelists, any one of which may be identified as an instance of the top-level list and provide values for the top-level list
    b. A top-level list with an empty attribute, which must be overridden by any one of one or more sub-lists.
3. Patterned codelist: Union of [*informative list, enumeration, flat*] + [*informative list, pattern, flat*]
4. Externally extended codelist: Union of [*informative list, enumeration, flat*] + [*informative list, external, flat*], i.e. codelist values may come from either a list explicitly defined in the standard or from an external list.
5. Externally replaced codelist: Exclusive or of [*normative list, enumeration, flat*] $\oplus$ [*informative list, external, flat*], i.e. codelist values may come either from a list explicitly defined in the standard or from an external source but an external codelist replaces all values explicitly defined in the standard. This replacement has a slight subtlety in that its use implies a requirement that within a model instance all values for an attribute that is a codelist type must come from a single source, either from the explicit list in the standard or from the external source but not both.

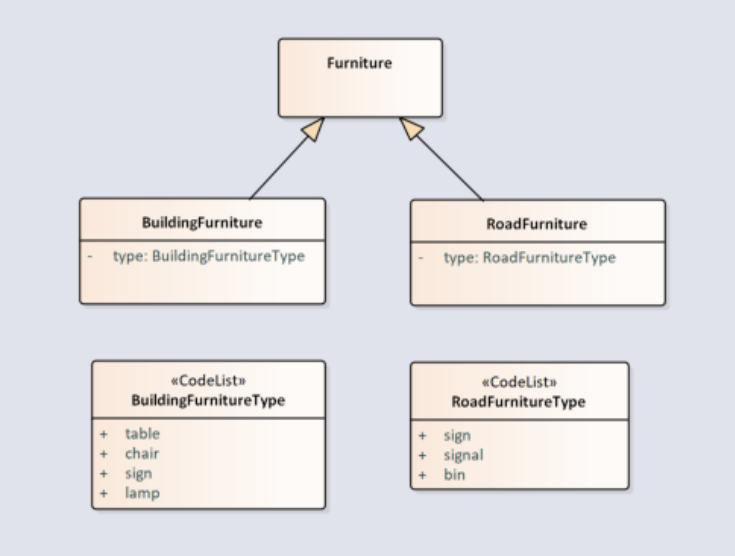
Source Declaration

An application or validator requires the source of a codelist value. One convention for string values that could be used in CityGML 3.0 is to specify a namespace using and alphanumeric identifier as a prefix,

except for explicitly specified values, which can be distinguished by the lack of a prefix. That is, a value without a prefix must be a value explicitly specified in the standard. For the source "pattern" , using the prefix "other:" would follow GML practice. All external sources could be distinguished by a prefix that is not the string "other:".

Codelist type structures

Each of the codelist structures is illustrated with a variation of the following simple model of a category "Furniture", which may contain elements of either "BuildingFurniture" or "RoadFurniture" subclasses.



The Furniture Model

Enumeration

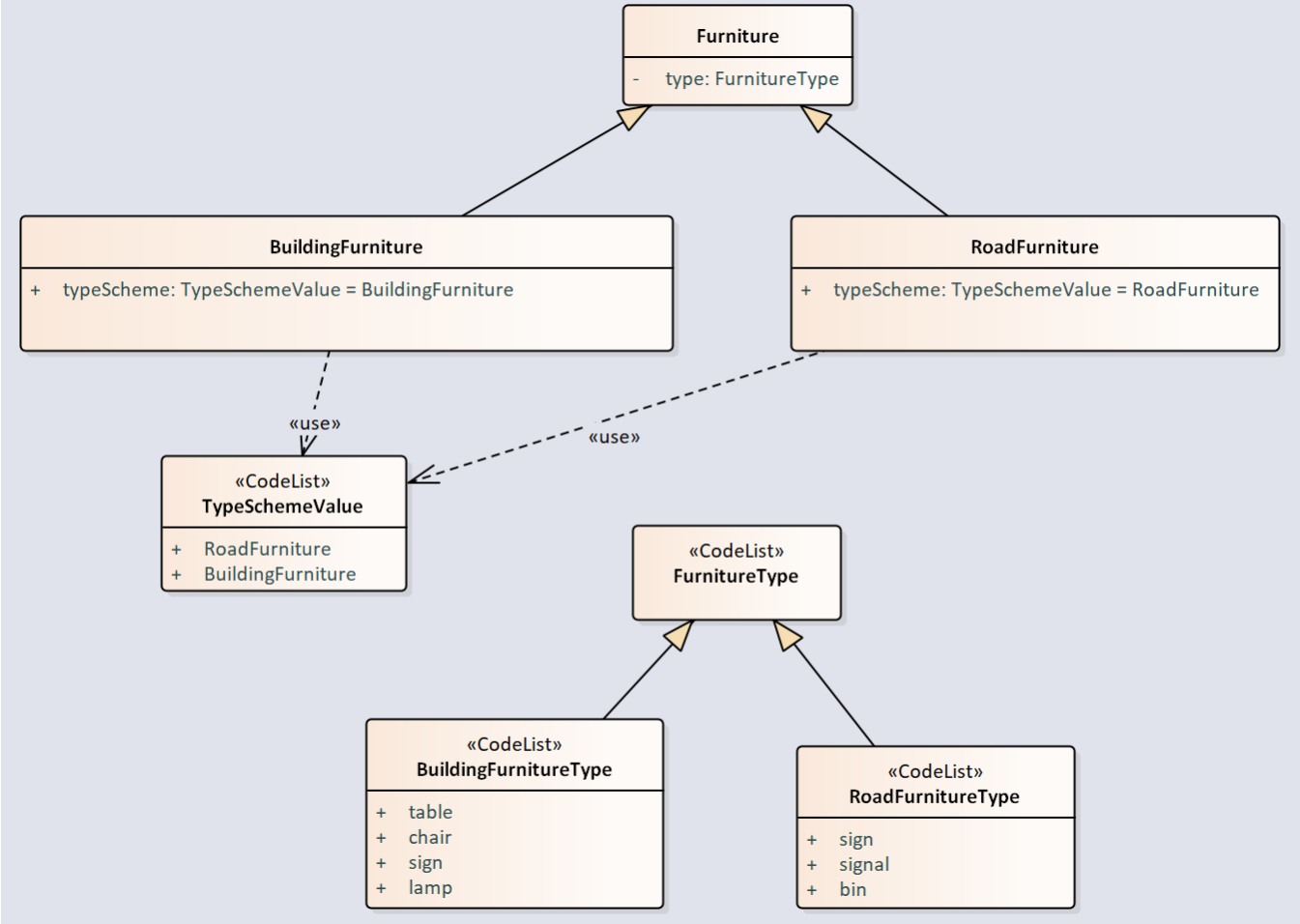
Enumerations are explicit and fixed lists of values. It is probably best to avoid the codelist stereotype and use built-in UML enumerations to model enumerations of valid string values in CityGML 3.0. The codelists in the Furniture model above are enumerations. One case where a codelist type might be chosen instead of an enumeration, even when the initial specification really is an enumeration, would be where it is anticipated that the range of attribute values would likely be extended by externally-specified values in a (near) future version of the specification.

Hierarchical codelist

Hierarchical codelists are more complex and that extra complexity may not make sense for CityGML 3.0. There are two main reasons to derive concrete codelist subclasses from a generic parent. The first is to allow a specification to list the different possible codelist instances that may be used as a codelist of the parent type. This does not follow normal object-oriented design principles but does provide a way to clearly show the relationship of sibling subclasses.

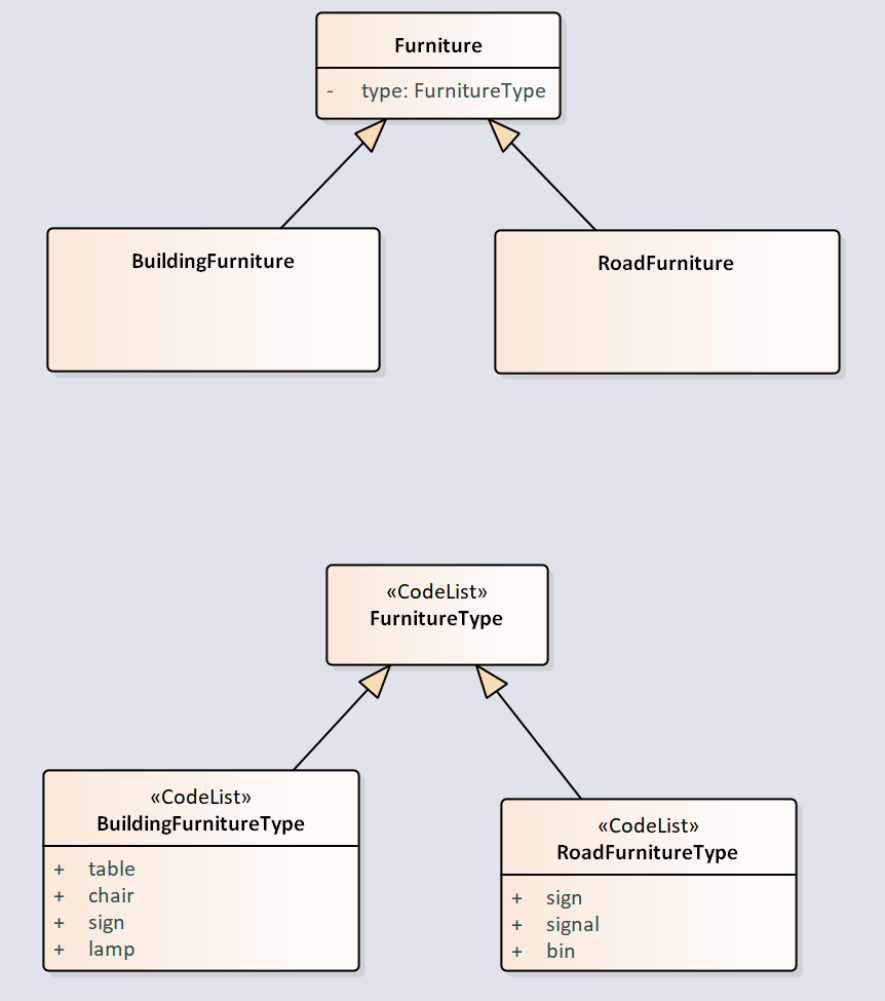
One way to regularize this usage would be to define a parallel class hierarchy in the objects containing an attribute selected from a codelist. If a superclass has an attribute value with a codelist type and it is allowed for subclasses to have an attribute with the same name and type of the corresponding codelist subclass, then the attribute value could be overridden by a subclass. This is entirely within the object-oriented design paradigm but involves a mechanism that operates behind the scenes.

The usage in INSPIRE follows a different pattern. Subclasses that need to use a set of codelist values from a specific codelist subclass have an attribute that specifies the codelist subclass (called a "scheme"). This allows a subclass to indicate the correct codelist subclass.



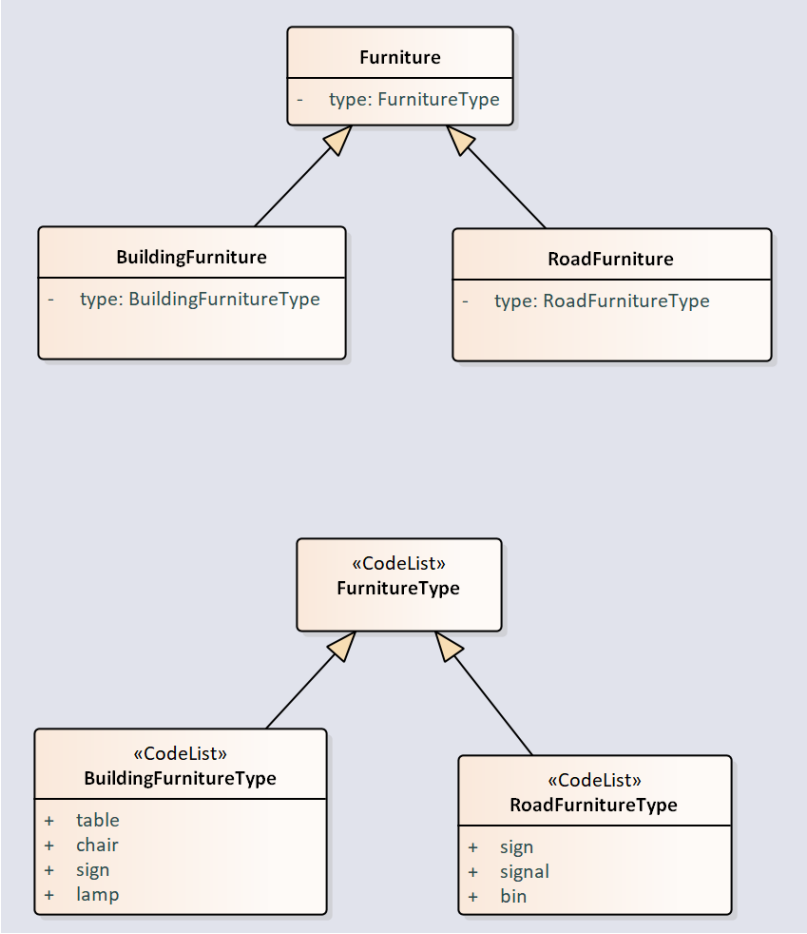
Codelist Subclasses Following the INSPIRE Pattern

Another pattern that is similar to the INSPIRE pattern is to again define the attribute in a superclass and to parallel that with a codelist class hierarchy but dispense with the extra framework for "schemes" and make an additional requirement, not expressed in UML, that the subclass of the concrete object class ("Furniture") must has a value from the parallel subclass of the "FurnitureType" codelist hierarchy. This removes a lot of extra structure that would otherwise end up in the model



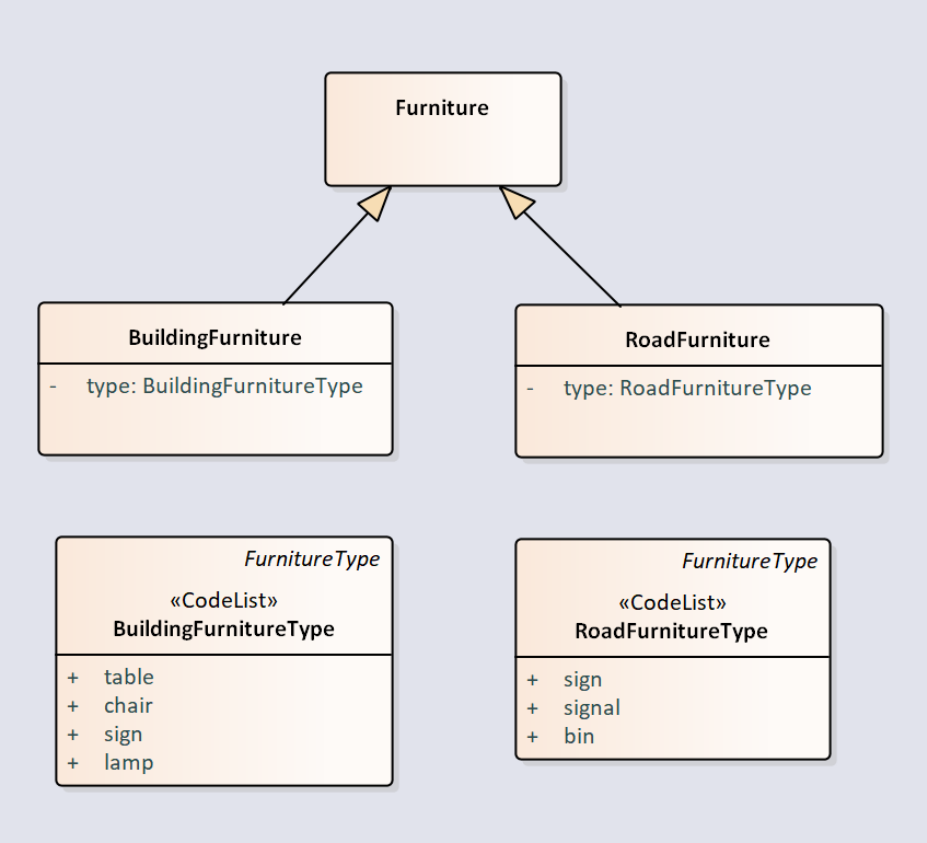
INSPIRE Pattern without Scheme Information in Data

Another possibility discussed (and possibly rejected) in the Modelling Subgroup was to simply allow subclasses of concrete object classes to override a an attribute defined in a superclass, giving it the correct type for values from the correct codelist. This seems like a simple way to accomplish the goal of the INSPIRE scheme mechanism without the extra structure.



INSPIRE Pattern without Scheme Information Using Attribute Override in Subclasses

Finally, and most directly, subclasses could directly reference the corresponding codelist and skip the codelist hierarchy altogether. This seems like a very direct approach, simplifies the data model and corresponding complexity of model instances. The cost is that the family relationship of the codelists is not explicit but the benefit is that no extra effort is required to specify exactly the codelist to be used.



Direct Specification of Values from Related Codelists without a Hierarchy

The second reason one might derive codelist subclasses is to allow a range of subclass codelists to share common valid codelist values, extended by lists of values specific to each subclass. The same effect can be achieved without the complexity of a hierarchy by copying the common codelist values into each of the (former) codelist subclasses. This flattening is probably the best choice.

Patterned codelist

A patterned codelist is simply one where valid values must match a regular expression in the codelist specification and values must be prefixed in a way that declares the source, e.g. with the string "other:" as already suggested.

Externally extended codelist

An externally extended codelist is one with a reference to an external source with additional valid values.

Externally replaced codelist

An externally replaced codelist is one with a reference to an external source with valid codelist values that replace any specified explicitly in the standard.

Encoding considerations

The primary extra consideration that must be given to encoding is that the string values must indicate their authoritative source of the code value. One simple approach is to use a namespace indicator. Enumeration values may be indicated by the lack of a namespace indicator. This means that any value lacking a namespace indicator must appear explicitly in the conceptual model definition. Following GML, patterned values may use the "other:" namespace indicator. Finally, URIs referencing external sources must have a namespace prefix indicator in the listed URI. The requires that all URI have a unique prefix within the specification and that prefix shall appear before the URI itself. For example, an external Irish road furniture type list might appear as "ierdf=https://schemas.nsai.ie/ogc/roadfurnituretypes" where "ierdf" is the prefix that must appear as a prefix on any attribute vale obtained from https://schemas.nsai.ie/roadfurnituretypes.

In summary, codelist attribute values appearing in a model instance either have no codelist prefix (enumerated values appearing explicitly in the standard), the prefix "other:" (values that match a pattern specified in the standard), or a prefix string defined in the corresponding URI entry in a codelist in the standard. It may be allowable to use a URI not specified in the standard but only if its prefix is not used in the standard. Such usage runs the risk of being invalidated by new versions of the

standard. To avoid this, the standard may specify the character patterns that may appear in the standard (e.g. specify a maximum length such that longer prefixes are guaranteed not to clash).

Remainder TBD
Possible Encodings for CodeList Variants

Enumeration

Specification

JSON

{

}

XML

<tag>

<tag/>

GML

SQL

Instance

JSON

{

}

XML

<tag>

<tag/>

GML

SQL

Hierarchy

JSON
XML
GML
SQL

Pattern

JSON
XML
GML
SQL

External Extension

JSON
XML
GML
SQL

External Replacement

JSON
XML
GML
SQL

Recommendations for CityGML 3.0

TBD

References