OGC® DOCUMENT: 23-999
External identifier of this OGC® document: http://www.opengis.net/doc/XXX/YYYYY

Open Geospatial Consortium

OGC MODEL-DRIVEN STANDARD MODELING BEST PRACTICES

BEST PRACTICE
General

DRAFT

Submission Date: 2023-02-01
Approval Date: 2023-02-01
Publication Date: 2023-02-01

Note

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

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

I ABSTRACT

This OGC best practice provides guidelines on how to create a model in Enterprise Architect suitable for the OGC model-driven standards process, described in OGC 21-035r1.

II KEYWORDS

The following are keywords to be used by search engines and document catalogues.

ogcdoc, OGC document, MDA, model-driven

III PREFACE

This OGC best practice provides guidelines on how to create a model in Enterprise Architect suitable for the OGC model-driven standards process.

IV SECURITY CONSIDERATIONS

No security considerations have been made for this document.

V SUBMITTERS

All questions regarding this document should be directed to the editor or the contributors:

NAME	ORGANIZATION	ROLE
Ronald Tse	Ribose	Editor
Foo	Bar	Editor
Foo	Bar	Editor

1

CONFORMANCE

1 CONFORMANCE

This OGC best practice ...

2

NORMATIVE REFERENCES

2 NORMATIVE REFERENCES

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Ronald Tse, Nick Nicholas: OGC 21-035r1, *OGC Testbed-17: Model-Driven Standards Engineering Report*. Open Geospatial Consortium (2022). https://docs.ogc.org/per/21-035r1.html.

3

TERMS AND DEFINITIONS

3 TERMS AND DEFINITIONS

This document uses the terms defined in OGC Policy Directive 49, which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this document and OGC documents do not use the equivalent phrases in the ISO/IEC Directives, Part 2.

This document also uses terms defined in the OGC Standard for Modular specifications (OGC 08-131r3), also known as the ‘ModSpec’. The definitions of terms such as standard, specification, requirement, and conformance test are provided in the ModSpec.

For the purposes of this document, the following additional terms and definitions apply.

3.1. conceptual model

CM ADMITTED

model that defines concepts of a universe of discourse

[**SOURCE:** ISO 19101-1, Clause 4.1.5]

3.2. conceptual schema

formal description of a *conceptual model* (Clause 3.1)

[**SOURCE:** ISO 19101-1, Clause 4.1.6]

3.3. model-driven standard

MDS ADMITTED

standard created using a model-driven architecture

[**SOURCE:** OGC 21-035r1, Clause 2.1.4]

3.4. model-driven architecture

MDA ADMITTED

software design approach for development of software systems centered around data models

[**SOURCE:** OMG UML 2.5]

3.5. model authoring tool

software used for authoring a *conceptual model* (Clause 3.1)

[**SOURCE:** OGC 21-035r1]

3.6. platform-independent model

PIM ADMITTED

conceptual model (Clause 3.1) that does not contain platform-specific concerns

[**SOURCE:** OGC 21-035r1]

3.7. platform-specific model

PSM ADMITTED

data model that contains platform-specific concerns

[**SOURCE:** OGC 21-035r1]

3.8. model transformation

model conversion from one form to another which may not preserve all semantics

[**SOURCE:** OGC 21-035r1]

3.9. model conversion

process that converts a data model in one format into another format that preserves all model semantics

[**SOURCE:** OGC 21-035r1]

3.10. stereotype

extension of an existing UML metaclass that enables the use of platform or domain specific terminology or notation in place of, or in addition to, those used for the extended metaclass

[**SOURCE:** OMG UML 2.5]

3.11. tagged value

attribute on a stereotype used to extend a UML model element

[**SOURCE:** OMG UML 2.5]

3.12. UML profile

predefined set of stereotypes, tagged values, constraints, and notation icons that collectively specialize and tailor UML for a specific domain or process

[**SOURCE:** ISO/IEC 19501]

3.13. **logical model**

implementation of one or more conceptual models

TODO.

4

INTRODUCTION

4 INTRODUCTION

The MDS process described in OGC 21-035r1 enables standardized documentation of conceptual models in UML, which could be platform-independent models (PIMs) or platform-specific models (PSMs).

In the past, UML modeling activity and the OGC authoring process used disparate tools, causing OGC authors and editors much pain in the synchronization of changes originating from either activity, as illustrated in Figure 1.

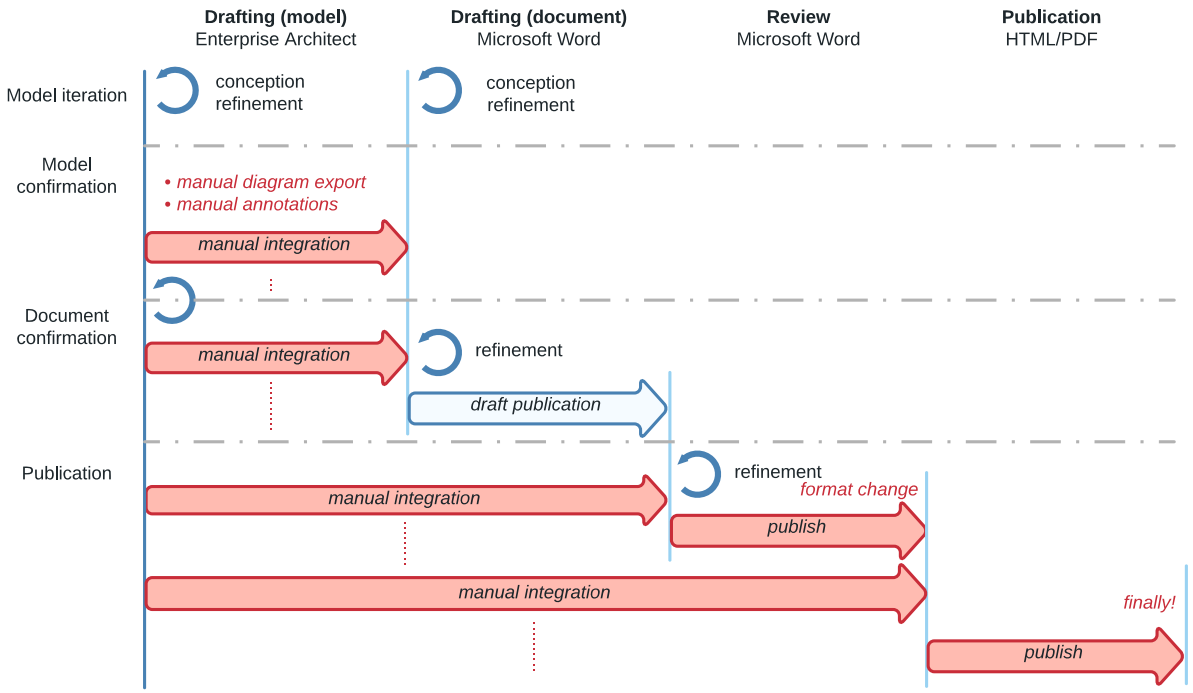


Figure 1 — Manual process for iterating a model-driven standard

As studied in OGC Testbed-17, OGC 21-035r1 has investigated several options in model-driven authoring, in which the OGC MUDDI SWG has decided to adopt and sponsor development of a particular approach that utilizes the following combination of tools:

- Enterprise Architect (from Sparx Systems) in the creation and maintenance of UML models, and
- Metanorma (from Ribose) in the authoring of OGC deliverables.

This combination of tools can provide a streamlined development environment for OGC working groups developing conceptual model standards:

- By maintaining standards content in the model, simplifying and decoupling the model maintenance process is possible.

- Storing annotations and guidance about the model together with the actual model enables a single source of truth that can streamline the standards authoring process.

This document is meant to describe best practices that enable achievement of these benefits.

By utilizing described practices of this document, the streamlined automated MDS process can be achieved as shown in Figure 2.

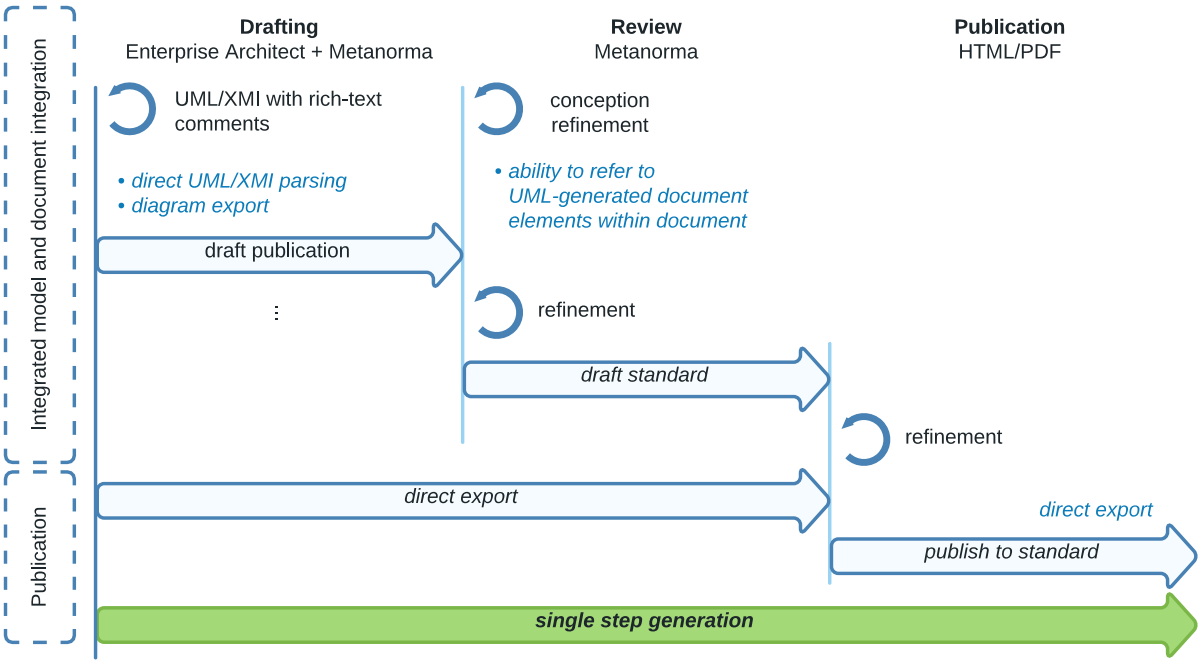


Figure 2 — One-step automated process for iterating a model-driven standard

5

DEVELOPING AN MDS

5 DEVELOPING AN MDS

5.1. General

The creation of an MDS must be planned. An MDS involves the synthesis of multiple data sources into a single one, therefore the MDS creator must be aware of the integration points and limitations of such synthesis process.

While the MDS process is meant to be a streamlined, automated process, it is nonetheless dependent on the interaction of multiple state-of-the-art technologies and **requires** the MDS creator to have a thorough understanding of the MDS technologies and techniques involved.

The full process is shown in Figure 3.

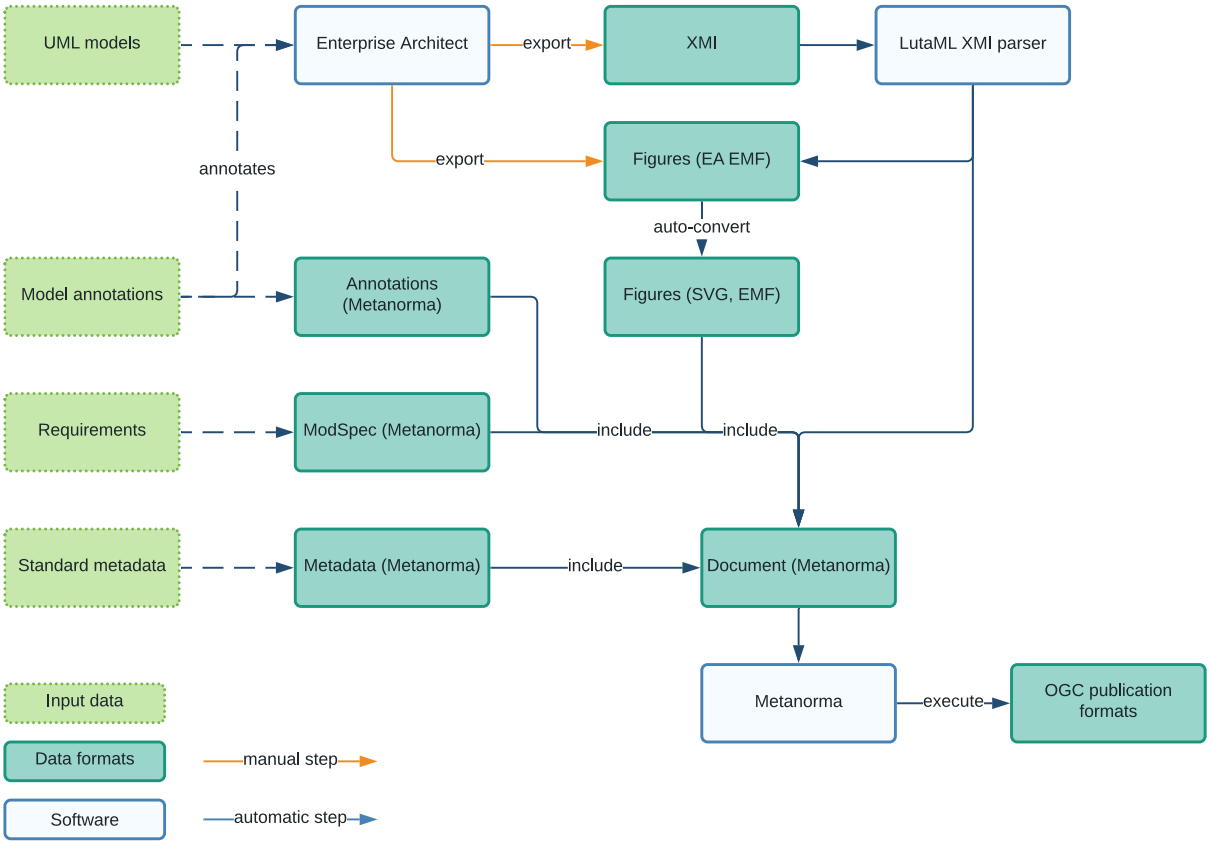


Figure 3 — Model-driven standard detailed publication flow

5.2. Data sources

Before embarking on an MBS, it is necessary for the MDS creator to know what kind of components there are.

In OGC, a model-driven standard is typically created with the following components:

- OGC document information in Metanorma AsciiDoc (scope, bibliography, etc.)
- UML model information in OMG XMI format (the EA UML models with annotations)
- OGC ModSpec information in Metanorma AsciiDoc format (requirements, conformance tests)

These components read into Metanorma using a defined processing configuration, and are then combined in Metanorma to form the MDS.

The resulting MDS represented in the Metanorma format will be expressed in the models provided in Figure 4.

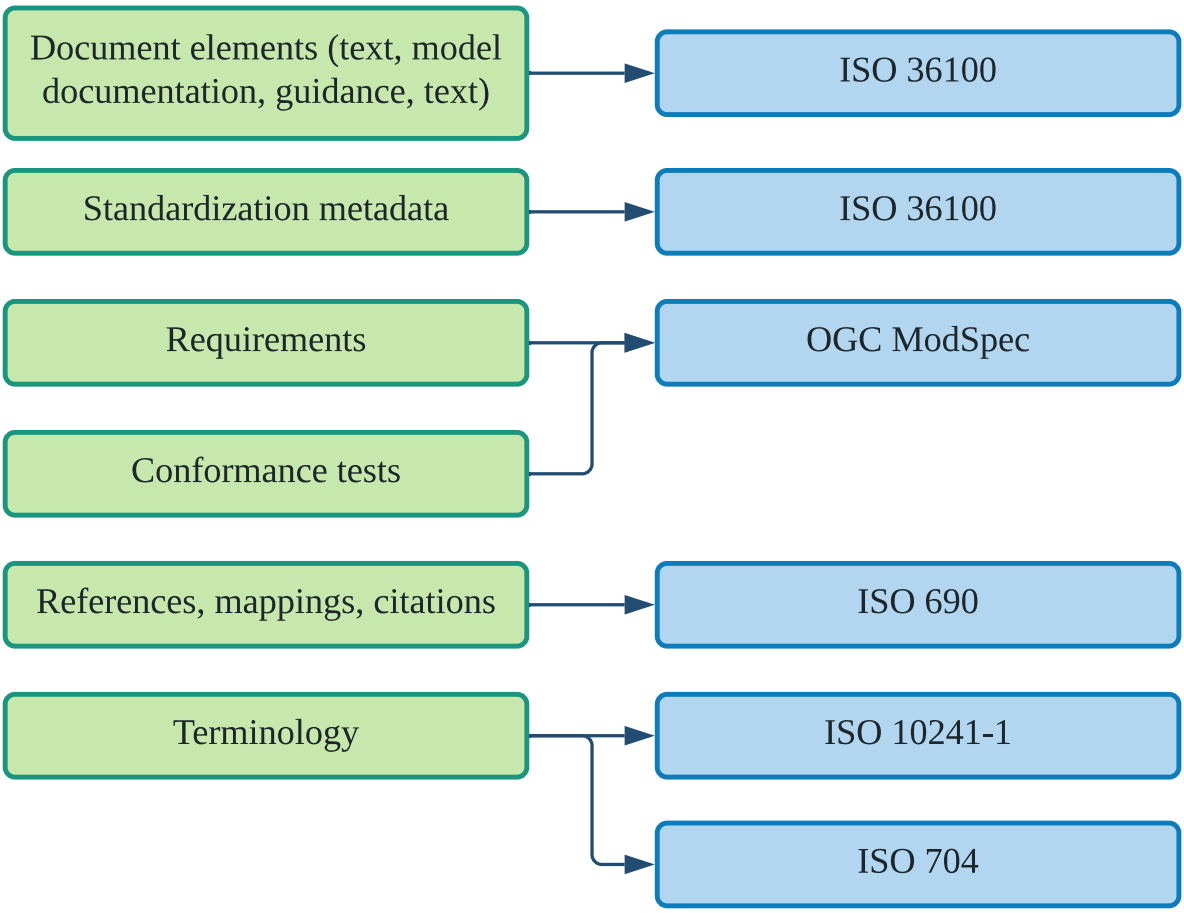


Figure 4 — Model-driven standard information components

5.3. Principles

OGC 21-035r1 describes generating an MDS involves the following steps:

- Export: Making the information model available for processing
- Authoring: Making the supplementary truth available for processing
- Data parsing: Parsing the truth of the model into derived truth in the document
- Integrating: Merging derived and supplementary truth into the target document
- Rendering: Generating human-consumable presentations of the target document

This document provides practices that allow the MDS author to plan out how the MDS automation process looks like across all these stages.

5.4. Export

Making source data available for the MDS involves exporting the information models in a standardized interoperable format from the model authoring tool.

For an OGC MDS document:

- the primary truth is typically a set of UML models. The initial step in processing is to export these UML models into interoperable XMI files.
- the secondary set of source data are the UML diagrams accompanying the UML models. They provide visual representations of UML classes described in the XMI files.

5.5. Authoring

This information is written in Metanorma, using the OGC flavor of the Metanorma AsciiDoc markup language.

Supplementary information in an MDS normally includes the following:

- Material such as bibliographies, terminological definitions, tutorial guidance, annexes, and prefatory material, which form part of a document presenting and explaining the model.
- Metadata about the document, such as keywords and identifiers.
- Requirements conforming to OGC ModSpec.

Where the supplementary information references specific model artifacts (annotating them), cross-references from Metanorma to the model become necessary; those cross-references are part of the integration of derived and supplementary truth.

5.6. Data parsing

Processing the XMI file is done under the Metanorma approach to MDA by LutaML, LutaML returns to Metanorma an array of objects, one for each of the objects in the source file parsed by LutaML, with a plugin structure to deal with the range of formats LutaML is called on to process (`lutaml-xmi`, in this instance).

Metanorma then uses Liquid directives to iterate through those objects, and insert information from them into Metanorma AsciiDoc templates. These templates are how information from the model is incorporated into the MDS as derived truth.

By using LutaML commands inside Metanorma, such as the `lutaml_uml_datamodel_description` command, UML class information is parsed from a nominated XMI file and transformed into Metanorma AsciiDoc.

Configuration files were used to specify which packages to render for each command call, in which sequence, and how to display them.

For complex documents such as CityGML 3.0 that require a non-default way of rendering, additional configuration can be used to achieve such results.

5.7. Integrating

A wide range of information is integrated into the target document.

This information is typically organized into separate directories in the source repository:

- The main Metanorma AsciiDoc document (`nn-mmm.adoc`), containing document metadata and directives to include sections contained in the document.
- The Metanorma AsciiDoc documents for each section in the standard (`sections/*.adoc`).
- Generic ModSpec requirements (not specific to the information models), each expressed as a separate file of Metanorma AsciiDoc, are included into the section documents at the appropriate point (`abstract_test`, `recommendations`, `requirements`).
- Non-model-generated images (`images`) and figures (`figures`) are included in the document as supplementary truths, as distinct from the UML diagrams exported into `/xmi-full/Images` as derived truths.

Supplementary truth is incorporated into the target document through standard AsciiDoc commands:

- `image::` for images and figures
- `include::` for content.

5.8. Rendering

Once the Metanorma AsciiDoc source is assembled out of its component truths, it can then be rendered using Metanorma into a number of output formats:

- Metanorma Semantic XML, capturing the structure and meaning of the standards document, and following the document model in ISO/AWI 36100.

- Metanorma Presentation XML, denormalizing the structure of the standards document in preparation for rendering, including resolving cross-references and generating auto-numbering.
- HTML
- PDF
- Microsoft Word

6

TECHNOLOGY AND TOOLS

6 TECHNOLOGY AND TOOLS

6.1. General

Best practices described in this document are meant for OGC working group participants fluent in the development of:

- UML conceptual models
- OGC authoring practices

This document does not delve into details in those areas — readers may wish to attempt further introductory for the full understanding of the practices described.

6.2. UML conceptual models

6.2.1. General

ISO/IEC 19501 specifies the UML modelling language, a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

UML specifies a set of methodologies for developing technical artifacts used in the design of a software system, ranging from business processes and system functions to programming language statements, database schemas, and reusable software components. UML is often used to develop domain-specific models (e.g., geospatial information) used in system development.

The usage of UML in MDS lies with two aspects:

- For model definition, the definition of information models and their relationships, that contain human- and machine-readable components; and
- For class diagrams, the visual arrangement of UML class relationships intended for human consumption only.

6.2.2. Modeling elements

NOTEA detailed description on UML modelling capabilities can be found in OGC 21-035r1, Clause 5.1.

UML provides 3 basic modeling elements:

Package	A package is a defined collection of interrelated classes.
Class	A class is an abstract representation of a real-world object, which contains properties.
Property	A property represents an aspect of a class.

UML allows additional modeling extensions in the following 3 ways:

Stereotype	A defined set of properties that a Class can adopt as a whole, commonly representing a platform-specific or domain-specific concern. More than one stereotype can be adopted by a single Class.
Tagged Value	A structured key-value pair defined for a UML element, allowing the attachment of additional (custom) information to the UML element.
Constraint	A string that limits possible value assignments to the property.

The UML “Profile” is another mechanism that allows for the easy application of stereotypes.

Profile	A profile contains multiple UML stereotypes that a UML model can adopt.

6.3. UML profiles for geospatial models

6.3.1. General

A number of common UML profiles are used for geospatial UML modeling.

6.3.2. UML Standard Profile

The UML Standard Profile is provided by the UML standard (OMG UML 2.5).

It provides the following stereotypes for Classes:

«Auxiliary»	A class that supports another class.
«Focus»	A class that specifies core logic or control with auxiliary classes that provide subordinate mechanisms.

«ImplementationClass»	An implementation class of a class.
«Metaclass»	A UML element that is meant to be extended.
«Realization»	A realization of an abstract UML element.
«Specification»	A specialization of a UML element.
«Type»	A data type.
«Utility»	A class that supports functionality of more than one class.

6.3.3. GML

In the geospatial domain, stereotypes from the GML profile (OGC 20-010) are often applied to geospatial UML elements.

The GML profile provides the following Stereotypes that apply to Classes:

«CodeList»	A list of enumerated codes. Practically an enumeration.
«DataType»	A basic type of information.
«FeatureType»	A type of feature.
«Type»	A type of information.
«Union»	A union of two classes.

The GML profile provides the following Stereotypes that apply to Properties:

«property»	A basic property.

6.3.4. ISO 19100-series profile: Conceptual schema language (ISO 19103:2015)

ISO 19103:2015 provides rules and guidelines for the use of a conceptual schema language to model geographic information, and specifies a profile of UML.

It includes 6 stereotypes:

«Interface»	(formerly «`Type`») is an abstract classifier with operations, attributes and associations, which can only inherit from or be inherited by other interfaces (or types).
«DataType»	is a set of properties that lack identity (independent existence and the possibility of side effects). A data type is a classifier with no operations, whose primary purpose is to hold information.

«Union»	is a type consisting of one and only one of several alternative datatypes (listed as member attributes); this is similar to a discriminated union in many programming languages.
«Enumeration»	is a fixed list of valid identifiers of named literal values. Attributes whose range type is an enumeration may only take values from the fixed list.
«CodeList»	is a flexible enumeration that uses string values for expressing a list of potential values. The allowed values are often held and managed using an online register.
«Leaf»	is a package that contains only classes (packages are disallowed).

The ISO 19103:2015 profile of UML also includes one tagged value:

- `codeList`, applies to stereotype «`CodeList`»: Code lists managed by a single external authority may carry a tagged value "codeList" whose value references the actual external code list. If the tagged value is set, only values from the referenced code list are valid.

The ISO 19103:2015 profile of UML is summarized in Figure 5.

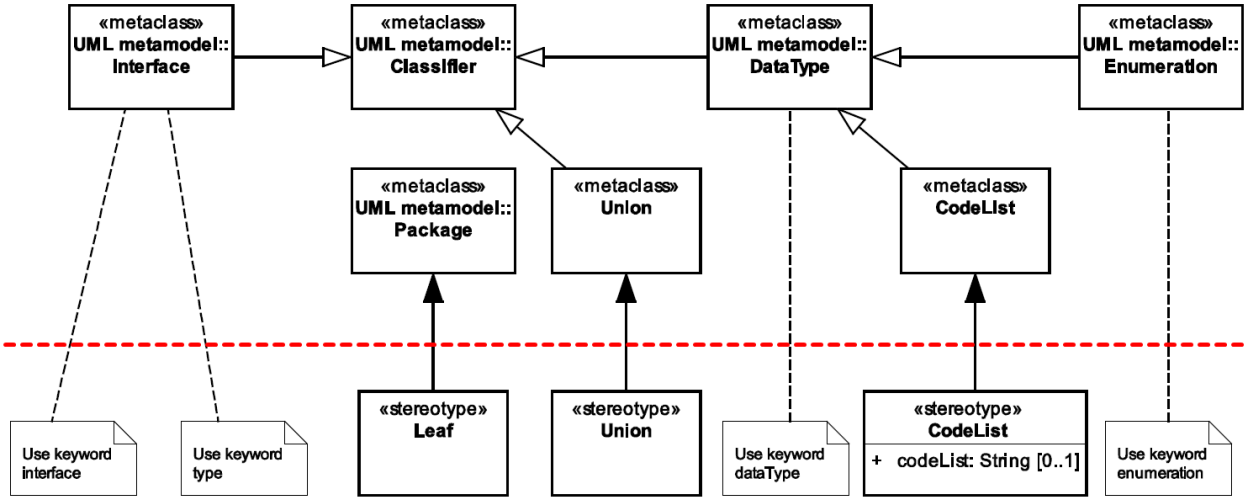


Figure 5 — ISO 19103:2015 stereotypes and keywords

6.3.5. ISO 19100-series profile: Rules for application schema (ISO 19109:2015)

ISO 19109:2015 defines rules for creating and documenting application schemas (conceptual schemas for data required by one or more applications), including principles for the definition of features, a fundamental unit of geographic information. As part of the general rules for application schemas it specifies the "General Feature Model" (GFM), the meta-model for application schemas.

The ISO 19109:2015 profile of UML that is used as the conceptual schema language for application schemas adds 2 stereotypes and 3 tagged values.

«ApplicationSchema»	(package) stereotype

«FeatureType» (class) stereotype

The following 3 tagged values apply to both of these stereotypes:

designation Natural language designator for the element to complement the name. Optional, with multiple designations allowed in order to support different languages.

definition Concise definition of the element. One definition is mandatory. Additional definitions can be provided in multiple languages if required.

description Description of the element, including information beyond that required for concise definition but which may assist in understanding its scope and application. Optional, with multiple descriptions allowed in order to support different languages.

The ISO 19109:2015 profile of UML is summarized in Figure 6:

Stereotype or keyword	UML metaclass	Constraints	Tagged values	Source
ApplicationSchema	Package		version	This International Standard, 8.2.3
			description catalogue-entry	This International Standard, 8.2.4
		Not nested in another applicationSchema package		This International Standard, 8.2.5
			language designation definition	This International Standard, 8.12
CodeList	Class			ISO 19103:—[5], 6.5.3, 6.10
dataType		Use in attributes or strong aggregation (composition)		ISO/IEC 19505-2:2012, 7.3.11 ISO 19103:—[6], 6.10
enumeration				ISO/IEC 19505-2:2012, 7.3.16 ISO 19103:—[7], 6.5.2, 6.10
Estimated	Property			This International Standard, 8.2.9
FeatureType	Class		description	This International Standard, 8.2.4, 8.2.6
			designation definition	This International Standard, 8.12
Leaf	Package	Cannot contain another package		ISO 19103:—[8], 6.10
Union	DataType			ISO 19103:—[9], 6.10
use				ISO/IEC 19505-2:2012, 7.3.40

Figure 6 — Summary of ISO 19109:2015 profile of UML

6.3.6. ISO 19118:2011 Geographic information — Encoding

ISO 19118:2011 specifies the requirements for defining encoding rules for use for the interchange of data that conform to the geographic information in the set of International Standards known as the "ISO 19100 series". It specifies requirements for creating encoding rules based on UML schemas, requirements for creating encoding services, and requirements for XML-based encoding rules for neutral interchange of data. It specifies a profile of UML that includes eight stereotypes, two of which are not previously defined similarly by either ISO 19103:2015 or ISO 19109:2015.

The profile provides the following stereotypes for Classes:

«BasicType» "Defines a basic data type that has defined a canonical encoding. " (ISO 19118:2011, Clause C.2.1.2)

Additionally stated is that: "This canonical encoding may define how to represent values of the type as bits in a memory location or as characters in a textual encoding. Examples of simple types are integer, float and string."

NOTE 1 For translation into XML, ISO 19118:2011, Clause C.5.2.1.1 states: "A class stereotyped «`BasicType`» shall be converted to a simpleType declaration in XML Schema. Any of the data types defined in XML Schema can be used as building blocks to define user-defined basic types. The encoding of the basic types shall follow the canonical representation defined in XML Schema Part 2: Datatypes (W3C xmlschema-2)."

NOTE 2 The different types are not clearly defined in ISO/TS 19103:2005 and neither is the «`BasicType`» stereotype used. The following declarations, therefore, follow a subset of the data type definitions in W3C xmlschema-2. Declared are the types: Number, Integer, Decimal, Real, Vector, Character, CharacterString, Date, Time, DateTime, Boolean, Logical, Probability, Binary, and UnlimitedInteger (where the symbol "*" is used to represent the infinite value).

«Interface» "Defines a service interface and shall not be encoded." (ISO 19118:2011, Clause C.2.1.2)

This definition is inconsistent with that of the subsequently published ISO 19103:2015. While this inconsistency may be useful in contexts where it is clear which definition applies, in general it is undesirable to overload the meanings of stereotypes within the OGC community, and in particular thereby coming into conflict with a stereotype specified in ISO 19103:2015.

While the stereotype «`Interface`» as defined in ISO 19118:2011 can be (and is here) subsequently ignored, the stereotype «`BasicType`» is used in the CityGML 3.0 Conceptual Model where it results in difficulties given its tie to a specific encoding technology — XML Schema — and thus lack of true platform independence. The CityGML 3.0 Conceptual Model redefines the stereotype «`BasicType`» to mean "defines a basic data type", which is both circular and differs from that of ISO 19118:2011.

6.4. Sparx Systems Enterprise Architect (EA)

Sparx Systems Enterprise Architect is widely used in OGC and ISO/TC 211 for the authoring and management of UML models.

EA Version 16 is a Windows application, it can be run in 32-bit or 64-bit mode on Windows, and can be run on other platforms using CrossOver (which is based on WINE technology) with 32-bit emulation.

6.5. Metanorma for OGC

Metanorma is an open-source framework for creating and publishing standardization artifacts with the focus on semantic authoring and flexible output support.

“Metanorma for OGC” is an OGC-specific implementation that has been approved as an official way to publish new OGC Standard documents since 2021-09-17. Metanorma-based document templates have been approved by the OGC Document SubCommittee on 2022-02-25.

Metanorma for OGC documents are created in the Metanorma AsciiDoc format. Metanorma AsciiDoc is a textual syntax for preparing a ISO/AWI 36100 compliant document model tree which can be rendered in a variety of presentation formats.

At its core, Metanorma provides a model-based documentation system and prioritizes automation, through the following features:

- a set of standard document metamodels (according to ISO/AWI 36100) that allows different standardization bodies to create their own standardized deliverable model, which in turn relies on the following standardized models:
    - ISO/PWI 36200 standards metadata specification metamodels;
    - ISO 690 bibliographic and citation item models;
    - ISO 10241-1 and ISO 704 concept organization and terminology models;
- a standard XML serialization (ISO/PWI 36300) for machine-readable standardization documents; and
- an open-source publishing toolchain that enables editors of standard documents to handle their documents from authoring to publishing in an end-to-end, “author-to-publish” fashion.

For OGC usage, it provides the following additional features:

- Rendering outputs in PDF, HTML, Microsoft Word, and ISO/AWI 36100 XML formats;

- Support for specification of OGC Standards metadata, including document types, stages, identifiers and authorship;
- Support for specification of OGC ModSpec (OGC 08-131r3) model instances through a specialized syntax.
- For OGC MDS usage, Metanorma supports navigation for information models in the OMG UML/XMI format (OMG UML within OMG XMI in XML format, OMG UML 2.5, OMG XMI 2.5) generated from Enterprise Architect, through the LutaML information model parser.

Figure 7 shows the range of models used in Metanorma, including the OGC-specific use of OGC ModSpec.

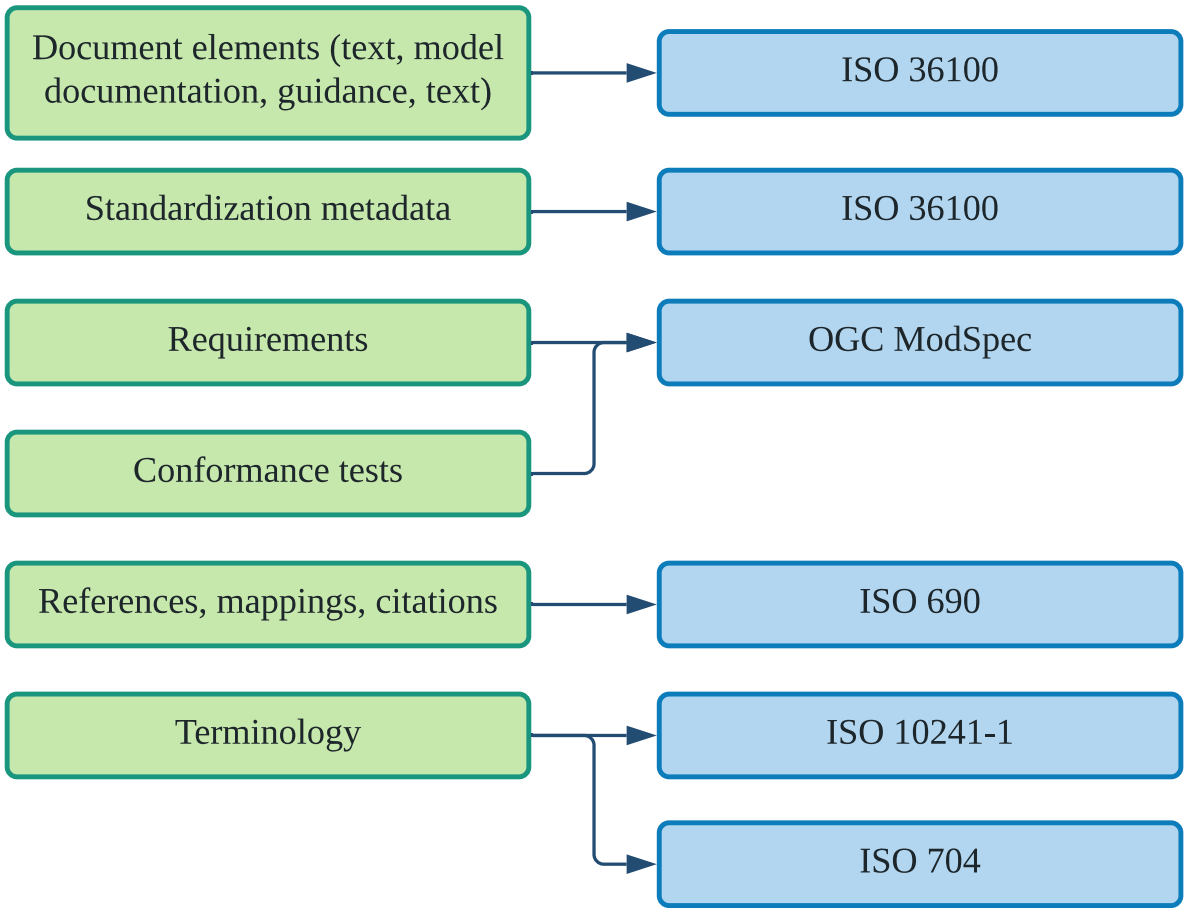


Figure 7 — Models used in Metanorma

6.6. LutaML information model interface

LutaML is an initiative grown out of Metanorma that allows parsing various machine-interpretable information models. LutaML adopts an extensible processing architecture to allow parsing different information model languages, through LutaML extensions.

Supported LutaML extensions include:

- EXPRESS, as specified in ISO 10303-11, is used heavily in smart manufacturing, Industry 4.0 use cases and in BIM, where EXPRESS itself served as the foundation of the IFC classes. The LutaML EXPRESS extension is available at: https://github.com/lutaml/lutaml-express.
- OMG UML in OMG XMI, which is the canonical format of representing UML models within XMI, an XML language defined by OMG OMG XMI 2.5. The LutaML XMI extension is available at: https://github.com/lutaml/lutaml-xmi.
- Sparx Systems Enterprise Architect XMI, the proprietary extension of Sparx Systems Enterprise Architect for the representation of UML. The LutaML Enterprise Architect-specific XMI extension is implemented within the LutaML XMI extension.
- LutaML UML, which is an ASCII syntax used to author OMG UML-compliant UML models with the possibility to be exported into OMG XMI format. The LutaML UML extension is available at: https://github.com/lutaml/lutaml-uml.

  NOTEThe LutaML UML language is documented at https://github.com/lutaml/lutaml-uml/blob/master/LUTAML.adoc

LutaML supports the dynamic referencing of elements from within a UML model. For example, individual UML classes, attributes, stereotypes, Enterprise Architect diagrams, can all be referenced through the unified interface provided by LutaML.

Collection filtering, such as to find UML classes that match certain UML stereotype, is also supported.

LutaML-XMI is the LutaML extension that parses OMG XMI 2.5 into a LutaML-UML model.

Of course, each format that it reads in requires a separate plug-in to be written to process it, and the processing of different formats can be highly specialized work. That makes it important for MDA to coalesce around standard ways of expressing models as much as possible, to minimize the up-front effort of developing a new plug-in to read a new model format.

The LutaML-XMI plug-in supports parsing the proprietary XMI files generated by Sparx Systems Enterprise Architect, incorporating details only available in the vendor proprietary XML portion of the XMI file.

This plug-in has been successful in recognizing the classes it expresses, their attributes, and the relations between classes, as documented in OGC 21-035r1.

6.7. Metanorma LutaML plugin

Metanorma interfaces with information models through the Metanorma LutaML plugin (https://github.com/metanorma/metanorma-plugin-lutaml). This plugin is used to render information models in human-readable formatting for MDS.

It provides a set of commands to be used within a Metanorma authoring context that invokes LutaML processing of a specified file, which generates a representation of that data usable within Metanorma.

Model navigation, dynamic referencing and collection filtering capabilities to UML models are accessible within a Metanorma document through the corresponding LutaML commands.

By default, LutaML is invoked to parse an external information model through a Metanorma AsciiDoc block command, which requires the input of the following information:

- as an argument, name of the source information model file;
- as an argument, the named context, which is the object variable name into which the data file contents are parsed, as object attributes, recursively;
- as the contents of the block, a template, in Metanorma AsciiDoc format with the Liquid template language (https://shopify.github.io/liquid/).

In effect, this provides a “meta-authoring” environment from within Metanorma. In particular, the template language allows the attributes parsed by LutaML to be incorporated in the block under the command.

7

BASICS OF ENTERPRISE ARCHITECT

7 BASICS OF ENTERPRISE ARCHITECT

7.1. Launch screen

Once the EA application is launched with a model file, the screen is shown as in Figure 8.

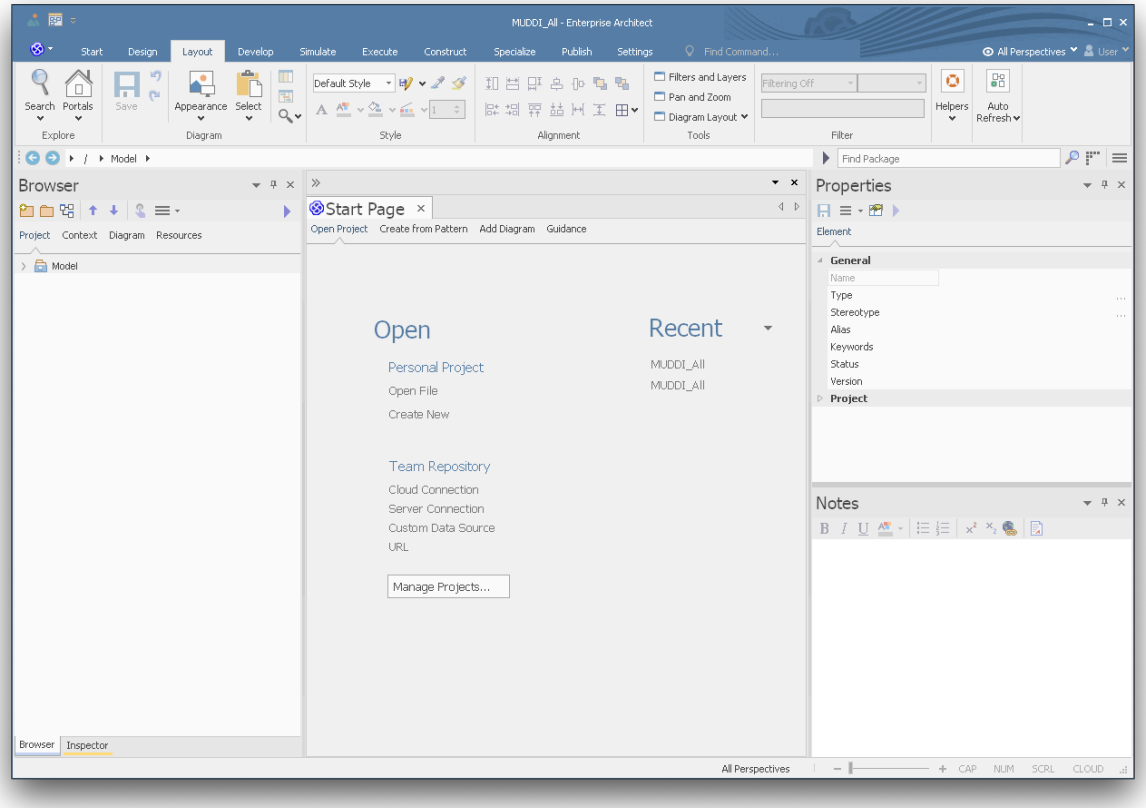


Figure 8 — Launch screen of Enterprise Architect

There are 4 basic panes in this screen:

- Browser: where the UML packages, models and properties are shown and can be navigated.
- Main pane: the area in the middle (labelled with the tab “Start Page”). It is typically used to show and work with diagrams.
- Properties: shows all properties and attributes of the selected UML element, whether it is a figure, package, class or property.

- Notes: shows textual annotations made to the selected UML element.

Relevant best practices:

- In the Notes pane, enter plain text in the Metanorma AsciiDoc format. While the pane supports rich-text entry, the text is encoded in HTML based on the antiquated Microsoft RTF format, and makes it difficult to perform any post processing upon extraction.

7.2. Using the Browser pane

The top-level package in the Enterprise Architect file can be expanded and drilled-down into.

Figure 9 shows how the hierarchy looks like.

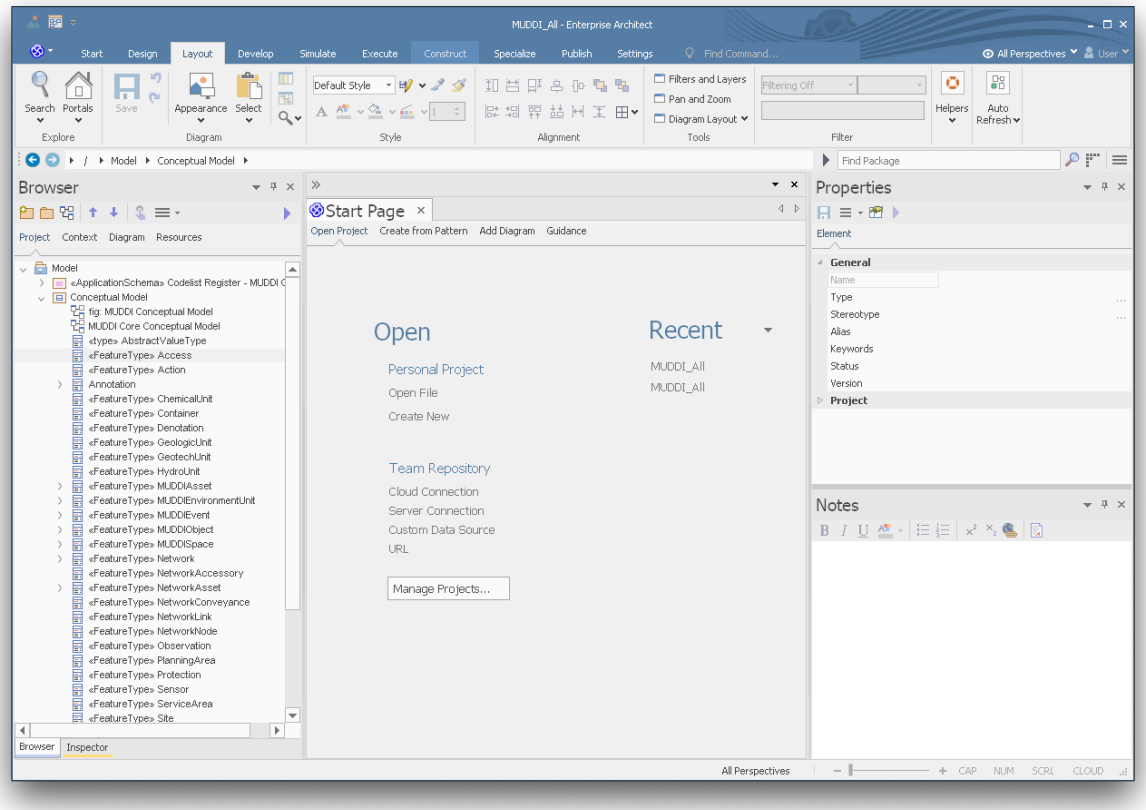


Figure 9 — Example of expanding the UML model hierarchy (source: MUDDI)

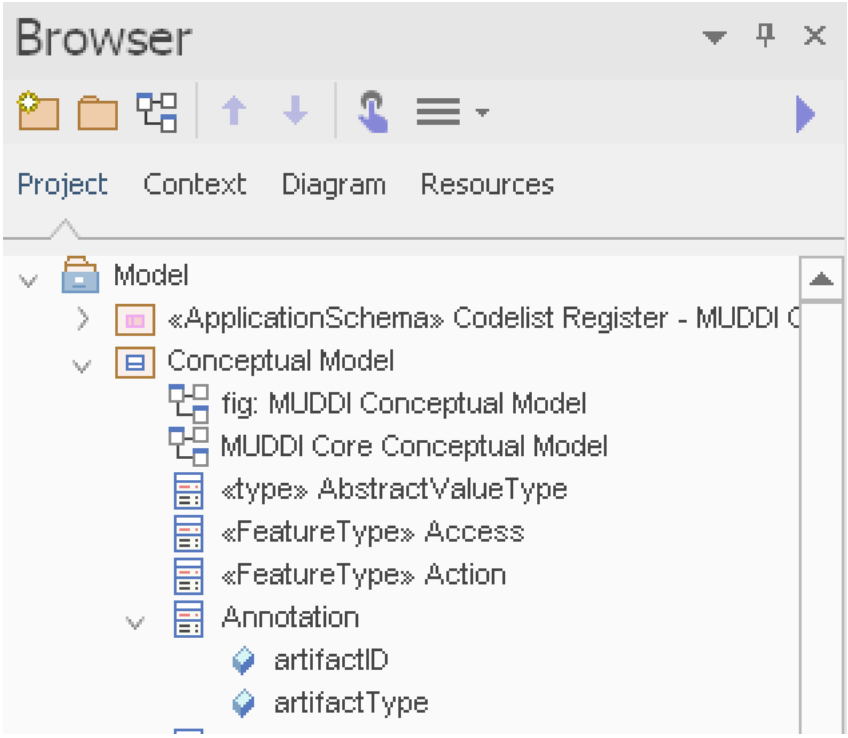


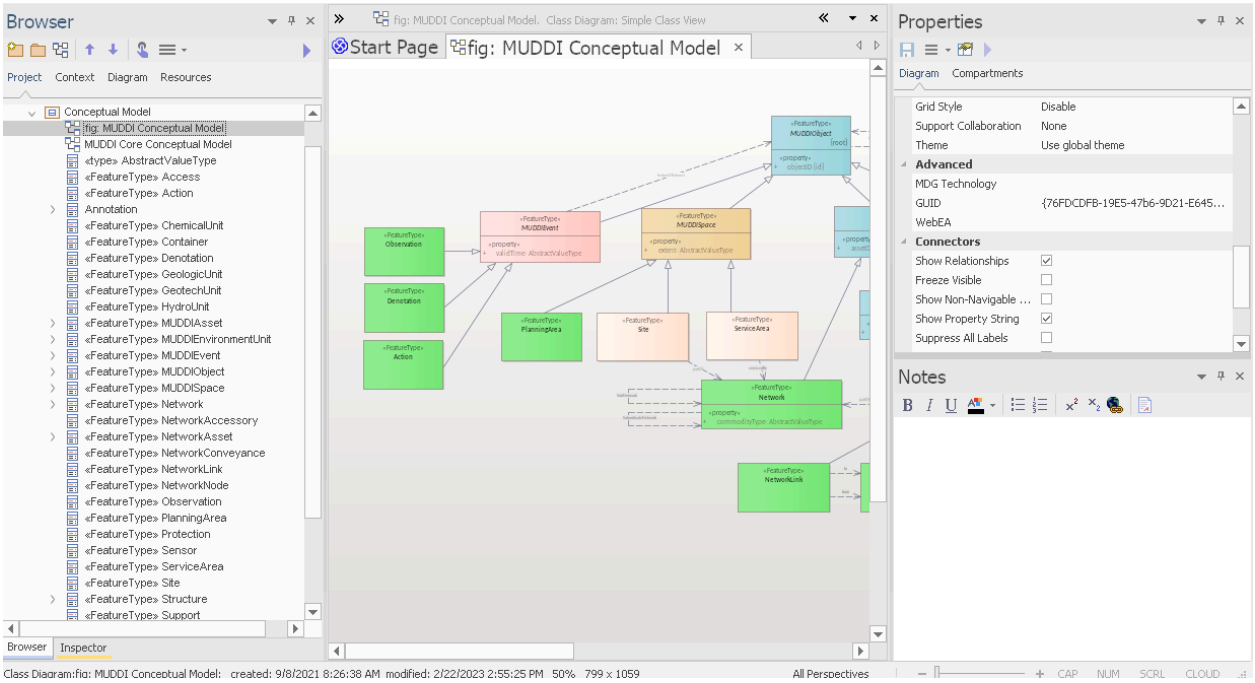
Figure 10 — Browser item types

In the Browser, there are 4 (basic) types of elements seen in its hierarchy (see Figure 10):

- Packages: UML packages.
  - The top-level item shown in Figure 9 is a UML package called “Model”.
  - The second item is a UML package called “Conceptual Model”.
- Diagrams: UML diagrams.
  - The 3rd and 4th items named: “fig: MUDDI Conceptual Model” and “MUDDI Core Conceptual Model” are figures.
- Classes: UML classes.
  - The 5th to 8th items are all UML classes.
- Property: UML element property.
  - The 9th to 10th items are UML properties that belong to the class “Annotation”.

7.3. Diagrams

When opening a diagram from the Browser pane, a tab will be opened in the middle pane showing the UML diagram (see Figure 11).



NOTEThe UML diagram can be zoomed in via the "View" action in the ribbon tab.

Figure 11 — UML diagram in EA

When a diagram is selected in the Browser, the Properties and Notes panes will be changed to reflect information about the selected diagram.

The MDS process uses the following information from an EA UML Diagram:

- Graphics of the diagram: is exported in the vector format and included in the OGC deliverable.
- Title of the diagram: as the caption of the Figure in the OGC deliverable.
- Notes of the diagram: contents of the Notes (seen in the Notes pane) is used as a “NOTE to Figure” in the OGC deliverable.

The title of the diagram is edited within the Properties pane when the diagram is selected. See Figure 12.

Model authors commonly create multiple diagrams but only wish to selectively include diagrams in the MDS process.

To indicate to the MDS process that a diagram is to be included, the title must be prefixed with `fig: ` (“fig:” with a space).

Diagrams that are not prefixed with `fig: ` will not be included in the MDS process.

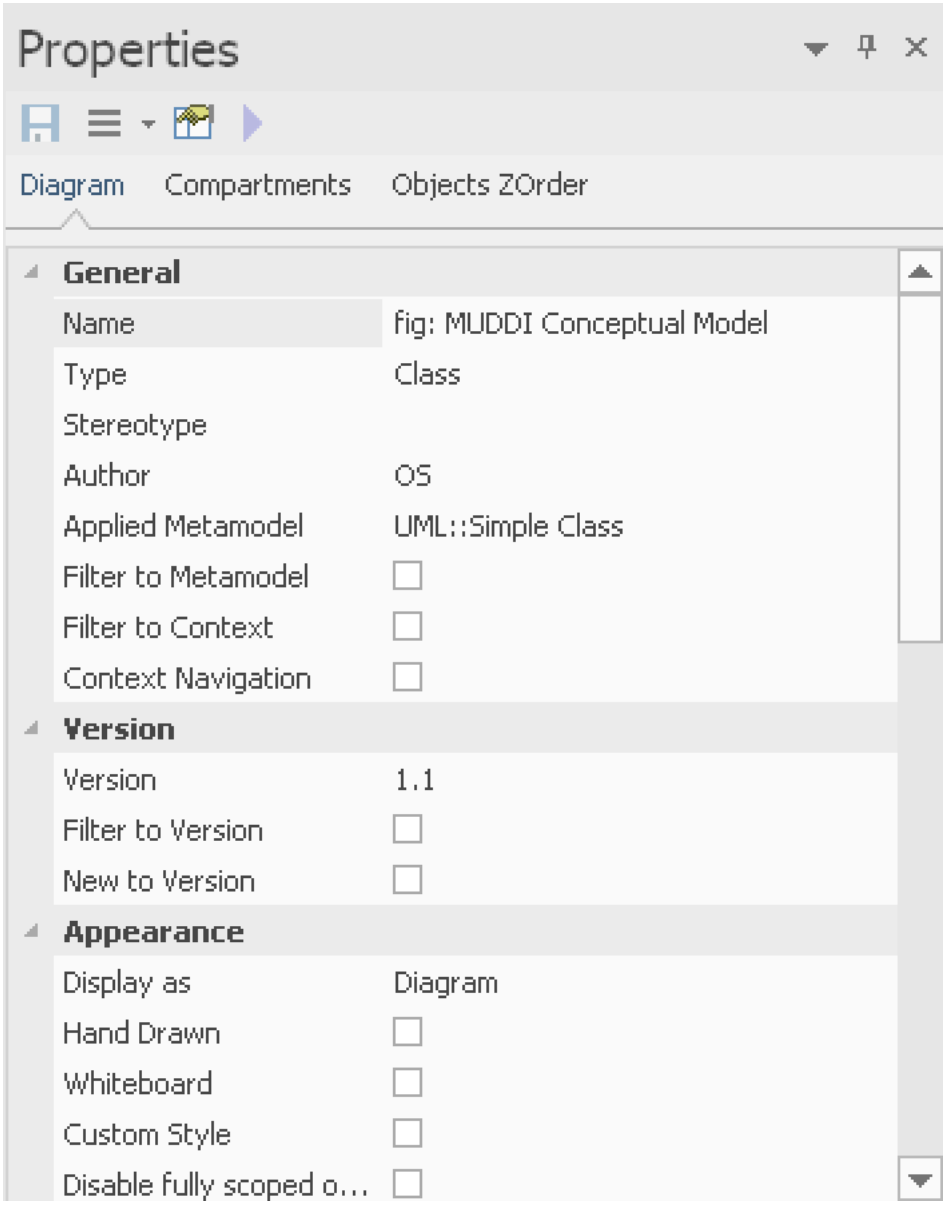


Figure 12 — EA Diagram Properties pane

7.4. Packages

On selection of a UML Package, the Properties and Notes panes will reflect the selected item.

The MDS process incorporates information of the UML Package, including:

- Notes of the UML Package: as the definition (description) of the UML Package (as in the Notes pane) (see Figure 13).
- Name of the UML Package: name of the UML Package is used as the clause heading in the OGC deliverable (see Figure 14).
- Package details:
  - URI: Identifier in URI format.
  - "Visibility": `Public`, `Private`, `Protected` or `Package` visibility.

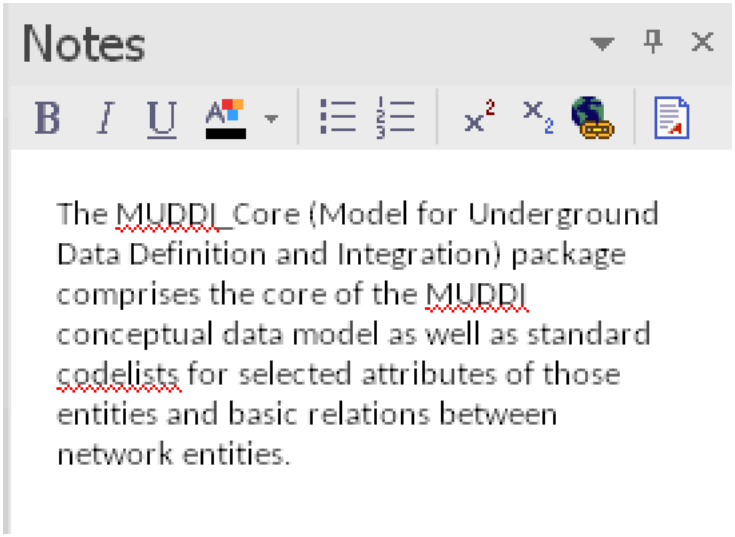


Figure 13 — EA UML package Notes pane

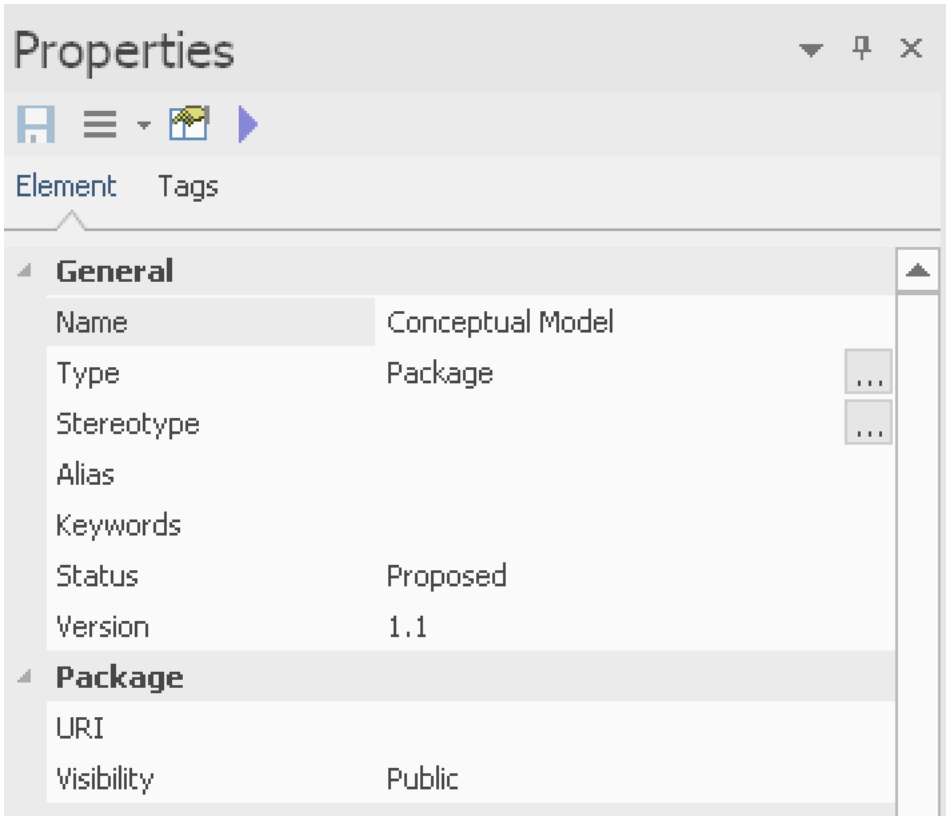


Figure 14 — EA UML package Properties pane

7.5. Classes

On selection of a UML Class in the Browser pane, the Properties and Notes panes will reflect the selected item.

The MDS process heavily incorporates information of the UML Class, including:

- Notes of the UML class: as the definition (description) of the UML Class (as in the Notes pane) (see Figure 15).
- Name of the UML class: name of the UML class, used as a clause heading in the OGC deliverable (see Figure 16).
- Stereotype of the UML class: stereotype of the UML class, wrapped with « and » characters in the OGC deliverable.
- Class properties:
  - "Abstract" status: whether it is an Abstract class
  - "Visibility": `Public`, `Private`, `Protected` or `Package` visibility

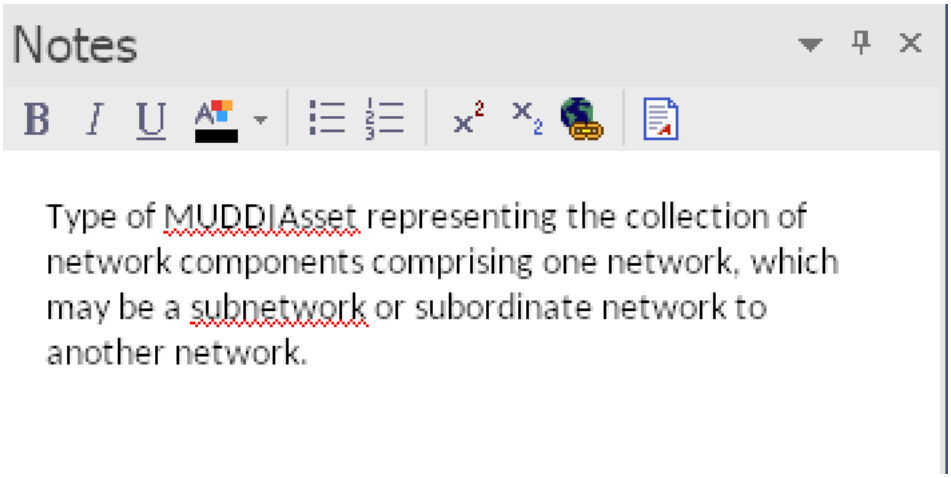


Figure 15 — EA UML class Notes pane

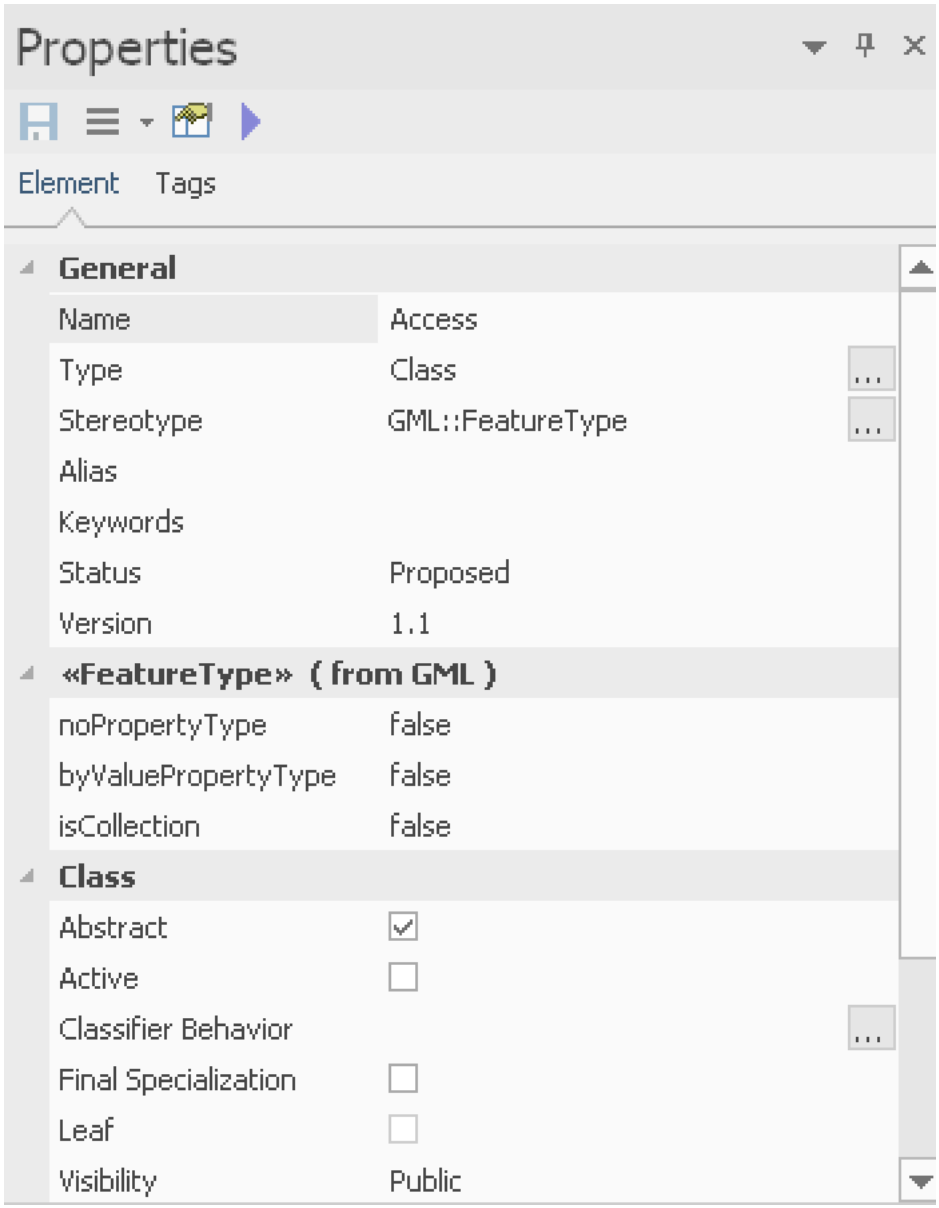


Figure 16 — EA UML class Properties pane

To set Stereotypes, click on the "..." to the right of the Stereotypes row in the Properties pane. A dialog box will be opened to allow selection of Stereotypes.

For geospatial modeling, EA supports setting Stereotypes from the following profiles:

- UML Standard Profile (see Figure 17)
- GML Profile (see Figure 18)

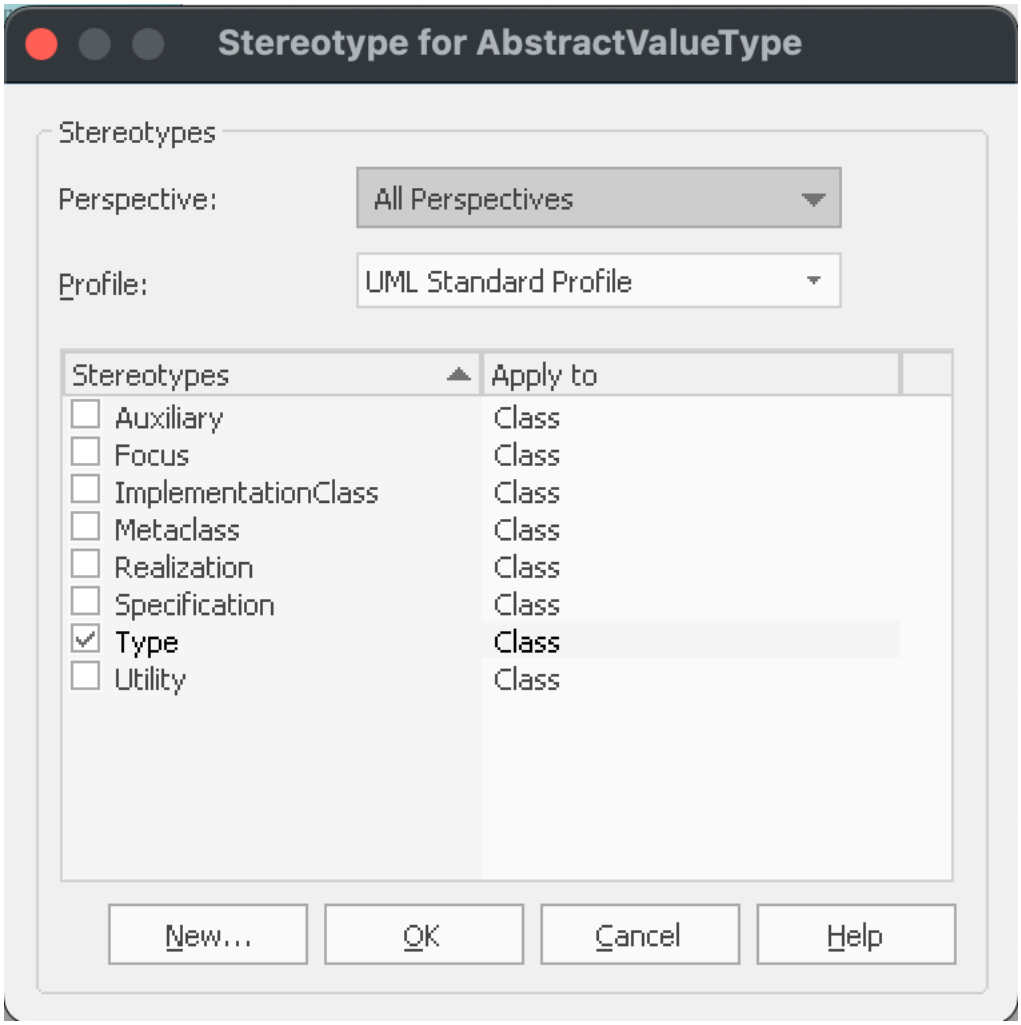


Figure 17 — EA UML Class Stereotypes: UML Standard Profile

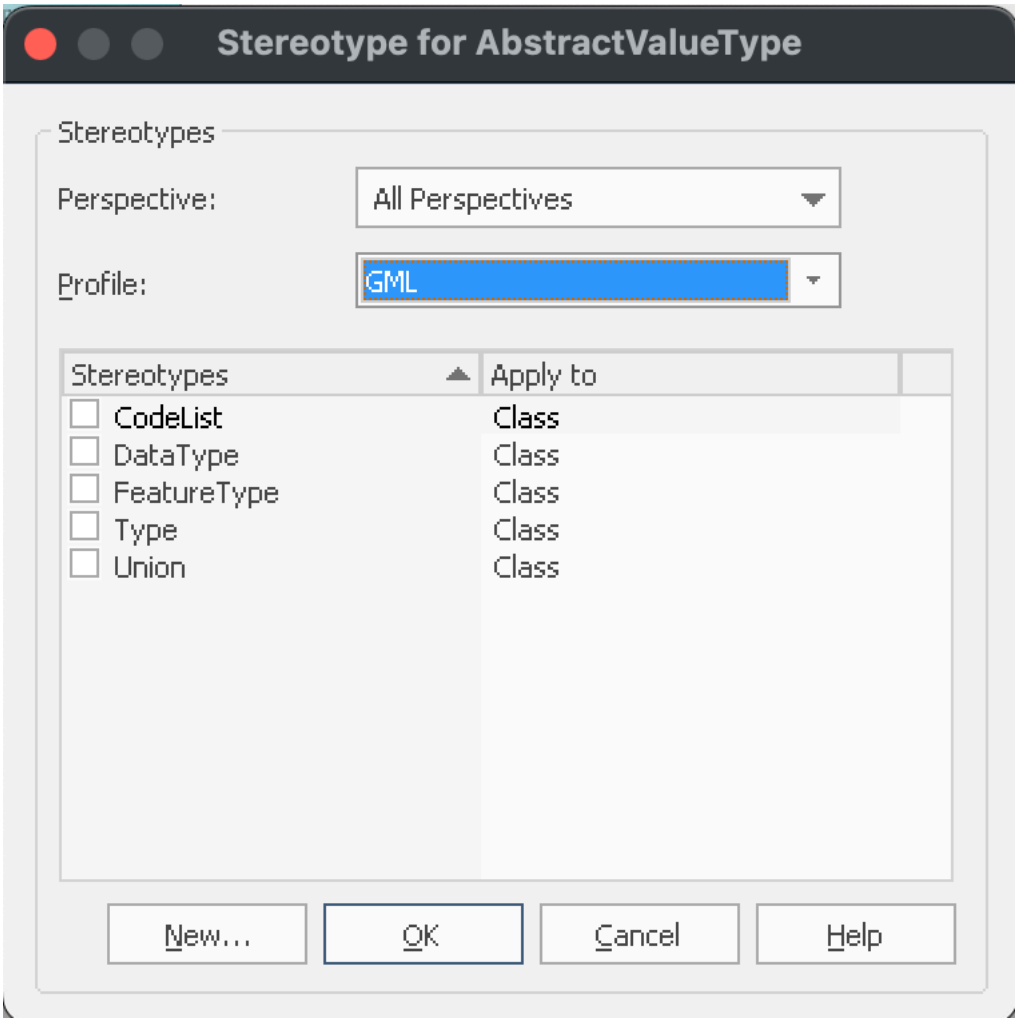


Figure 18 — EA UML Class Stereotypes: GML

7.6. Properties

On selection of a UML Property (under a UML Class), the Properties and Notes panes will reflect the selected item.

The MDS process heavily incorporates information of the UML Property, including:

- Notes of the UML Property: as the definition (description) of the UML Property (as in the Notes pane) (see Figure 19).
- Name of the UML Property: name of the UML Property, used as a clause heading in the OGC deliverable (see Figure 20).
- Stereotype of the UML Property: stereotype of the UML Property, wrapped with « and » characters in the OGC deliverable.

- Property details:
  - Initial value: default value if not specified.
  - Multiplicity: `0`, `1`, `0..1`, `0..`, `1..`, `*`

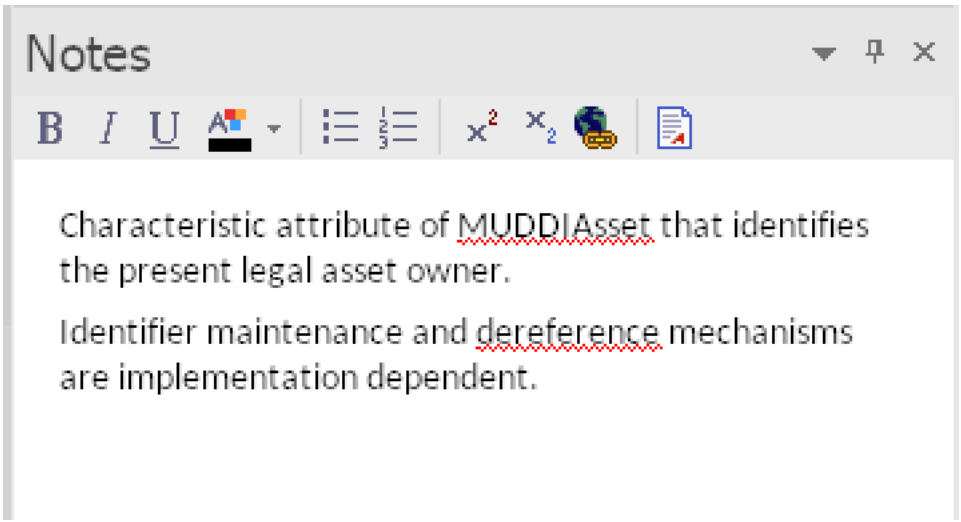


Figure 19 — EA UML property Notes pane

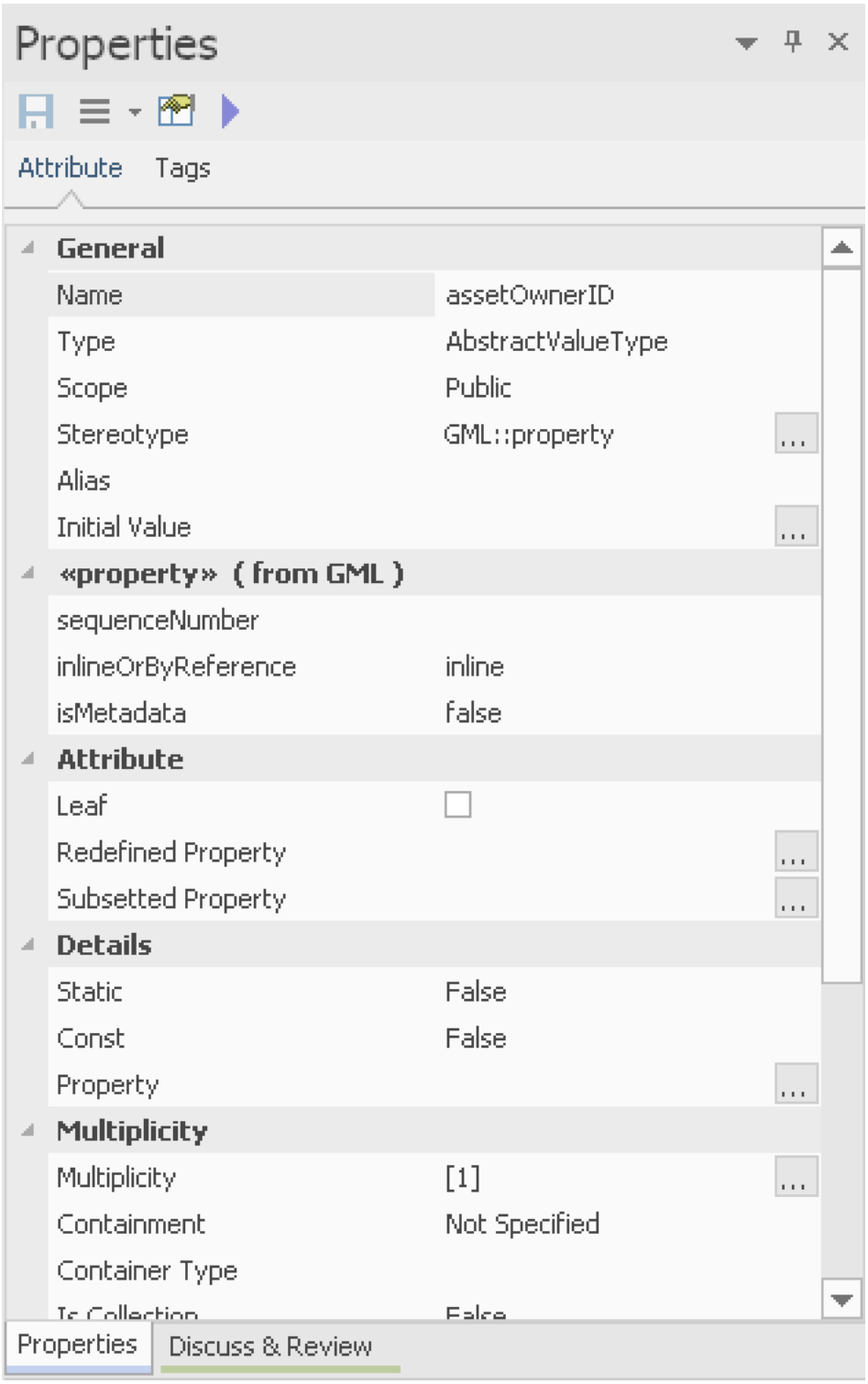


Figure 20 — EA UML property Properties pane

8

BASICS OF METANORMA

8 BASICS OF METANORMA

8.1. General

Metanorma uses a syntax called Metanorma AsciiDoc, which is based on the AsciiDoc format with a number of extensions.

An OGC Metanorma document is composed of two parts:

- Metadata
- Content body

8.2. Encoding

8.2.1. Metadata

8.2.1.1. General

The metadata portion is composed of the document header and attributes. An example of the metadata portion is shown below.

Example — Example of Metanorma AsciiDoc metadata

```
= OGC MUDDI Conceptual Model
:doctype: standard
:docsubtype: conceptual-model
:language: en
:status: draft
:committee: technical
:docnumber: 22-999
:received-date: 2023-01-01
:issued-date: 2023-01-01
:published-date: 2023-01-01
:external-id: http://www.opengis.net/doc/XXX/YYYYY
:keywords: ogcdoc, OGC document, MDA, model-driven
:mn-document-class: ogc
:imagesdir: images
:mn-output-extensions: xml,html,pdf,doc,rxl
```

8.2.1.2. Metanorma instructions

The following lines specify that this document is an OGC document, and it should render the various specified types of output, including XML, HTML, PDF, Word and RXL. RXL refers to the Relaton XML format which is used for encoding bibliographic information, and is required for the Metanorma site generation functionality.

The `:imagesdir:` attribute indicates that all images are located under that path, when using the `image::{path}[]` directive.

```
:mn-document-class: ogc
:mn-output-extensions: xml,html,pdf,doc,rxl
:imagesdir: images
```

Figure 21

8.2.1.3. Document type and sub-types

OGC has an extensive list of document types and some of them require specification of sub-types.

Please refer to Metanorma for a full list of these values. If there is no sub-type for the document type, do not specify a sub-type.

```
:doctype: standard
:docsubtype: conceptual-model
```

Figure 22

8.2.1.4. Document status

OGC document types are processed through different approval procedures, and this attribute encodes the status of a document.

Please refer to Metanorma for the list of statuses available for the particular document type. Invalid statuses will result in warnings during document generation.

```
:status: draft
```

Figure 23

8.2.1.5. Document identification

OGC documents are uniquely identified via two aspects:

- OGC document number. This is a unique number is obtained from the OGC portal through a reservation process, in a pattern of “nn-mmm”.

  NOTE”nn” refers to the year when the document number is reserved, and “mmm” is a sequential number reflecting the number of documents in that year prior to reservation.

- OGC unique identifier. This identifier is called the `external-id` in Metanorma. This identifier typically has the pattern like `xxx/yyy`, and is required to be unique across OGC.

```
:docnumber: 22-999
:external-id: http://www.opengis.net/doc/XXX/YYYYY
```

Figure 24

8.2.1.6. Document provenance

An OGC document is typically developed under the scope of the OGC Technical Committee.

```
:committee: technical
```

Figure 25

8.2.1.7. Document dates

The OGC standards development process specifies several approval related dates. These dates need to be encoded as they pass through those stages.

```
:received-date: 2023-01-01
:issued-date: 2023-01-01
:published-date: 2023-01-01
```

Figure 26

8.2.1.8. OGC keywords

OGC requires all documents to have keywords specified for the purpose of enabling user discovery.

```
:keywords: ogcdoc, OGC document, MDA, model-driven
```

Figure 27

8.2.2. Body

8.2.2.1. General

An OGC document has certain fixed and mandatory sections.

For a conceptual model document, it includes:

- Preface sections
- Clause 1: Scope
- Clause 2: Conformance
- Clause 3: Normative references
- Clause 4: Terms and definitions
- Clause 5 onwards: content body
- Annexes (optional)
- Bibliography

8.2.2.2. Preface sections

The preface sections are encoded as follows.

```
[abstract]
== Abstract

Enter the abstract for this document.

== Preface

Enter the preface for this document.

== Submitters

All questions regarding this document should be directed to the editor or the
contributors:

[options="header"]
|===
| Name | Organization | Role

| Given-name-1 Last-name-1 | Organization-1 | Editor
| Given-name-2 Last-name-2 | Organization-2 | Editor
| Given-name-3 Last-name-3 | Organization-3 | Editor

|===
```

Figure 28 — Preface sections in Metanorma AsciiDoc

8.2.2.3. Scope

The scope describes the purpose of the document in succinct terms.

```
== Scope
```

```
This OGC Standard provides...
```

Figure 29 — Scope in Metanorma AsciiDoc

8.2.2.4. Conformance

The conformance section describes the requirements stated by the document.

```
== Conformance

This OGC Standard provides the following requirements...
```

Figure 30 — Conformance in Metanorma AsciiDoc

8.2.2.5. Normative references

The normative references section describes information resources necessary for the implementation of the document.

```
== Normative references

* [[[OGC_08-131,OGC 08-131r3]]], OGC ModSpec
```

Figure 31 — Normative references in Metanorma AsciiDoc

8.2.2.6. Terms and definitions

The terms and definitions section defines the terms used in the document, which could be defined by the document or imported from other resources.

The terms and definitions section can encode complex concepts and relations, for detailed documentation please refer to the Metanorma website.

```
== Terms and definitions <1>

==== conceptual model <2>
alt:[CM] <3>

model that defines concepts of a universe of discourse <4>

[.source]
<<ISO_19101-1,clause=4.1.5>> <5>

==== logical model

model that implements a {{conceptual model}} at a logical level <6>
<1> Mandatory clause title
<2> Term for concept
<3> Alternate term for concept
<4> Definition of concept
<5> Source of concept
<6> Concept mention of a defined term in the same document
```

Figure 32 — Terms and definitions in Metanorma AsciiDoc

8.2.2.7. Content body

The content body is used to describe the conceptual model and is composed of one or more clauses.

In an OGC MDS document, it is necessary to utilize one or more section to describe the information model. Typically, the Metanorma LutaML plugin is used to render the conceptual model in XMI format. Information on how to use this automated process is described in Clause 10.3.

```
== Model overview

=== Design requirements

The development of MUDDI has been motivated by a number of specific design
requirements...
```

Figure 33 — Content body in Metanorma AsciiDoc

8.3. Building the document

8.3.1. Single document

The command to build a document is: `metanorma {filename}`.

Example — Example of running the `metanorma compile` command: This command compiles the Metanorma AsciiDoc file `my-ogc-standard.adoc` into an HTML document.

```
$ metanorma my-ogc-standard.adoc
```

8.3.2. Site

Metanorma supports a site build feature that is useful when multiple outputs are expected.

A site manifest needs to be created at `metanorma.yml`, where it internally specifies the component documents of this site. An example is shown in Figure 34.

```
---
metanorma:
  source:
    files:
      - sources/as21-dggs/20-040r3.adoc
      - sources/as21-dggs/iso-19170-1-is-en-sections.adoc

  collection:
    organization: "OGC"
```

```
    name: "OGC TB 17 D144 DGGS XMI model-driven standard"
```

Figure 34 — Example of generating both OGC and ISO flavors using a site manifest

Assuming that the `metanorma.yml` file exists at the current path, the command to generate a site is:

```
$ metanorma site generate
```

Figure 35

The resulting site will be built at `_site` which contains the entry point of `_site/index.html`.

9

SPECIFYING REQUIREMENTS

9 SPECIFYING REQUIREMENTS

9.1. General

This clause describes best practices on how OGC requirements are encoded adhering to the OGC Modular Specification (OGC 08-131r3), also called the "ModSpec", in an OGC deliverable.

OGC ModSpec specifies a requirements model scheme where requirements are expressed through a set of UML models, with description on how these models are to be treated and presented in OGC standards.

According to the OGC Policy Directives, OGC standards that contain requirements must have those requirements conform to OGC 08-131r3.

As OGC utilizes the Metanorma toolchain for publishing its standards, it is necessary for the OGC author to understand how ModSpec instances are encoded in the Metanorma format.

9.2. Background

Metanorma provides a special syntax for the encoding and embedding of requirements compliant to OGC ModSpec, for the exporting of machine-readable requirements as well as ModSpec-compliant rendering.

Specifically, the following models in ModSpec are supported in Metanorma:

- Conformance class
- Conformance test
- Requirements class
- Normative statements
    - Requirement
    - Recommendation
    - Permission (not specified in ModSpec but allowed in ISO 19105, see OGC 08-131r3, Clause 4.20)

NOTE 1The "Conformance suite", "Conformance module", "Requirements module" models are not yet supported in Metanorma. Please contact OGC DocTeam if support is required.

In this document, we refer to “recommendations”, “requirements” and “permissions” collectively using the generic term “requirement”.

NOTE 2 In some instances, the naming of terms that Metanorma uses in general is used in Metanorma markup instead of the nomenclature used in the ModSpec:

- Metanorma uses *target* to refer to what the requirement is about, rather than the more specific language of the ModSpec, to ensure that requirements are represented consistently within Metanorma.
- The different types of requirement expressed by Metanorma for ModSpec are about different things, and the more abstract types of requirement are about other requirements.

9.3. ModSpec models

9.3.1. General

A basic understanding of ModSpec is crucial in order to understand how to encode ModSpec-compliant models.

This clause describes ModSpec models in simplified terms (see OGC 08-131r3, Annex C).

9.3.2. Requirements class

A “Requirement class” consists of multiple “Requirements”.

All “Requirements” within a “Requirement class” are about the same standardization target type.

9.3.3. Requirement

A “Requirement” is a condition to be satisfied by a single standardization target type.

9.3.4. Conformance class

A “Conformance class” consists of multiple “Conformance tests”.

A “Conformance class” is associated with a single corresponding “Requirements class”.

Each “Conformance test” within the “Conformance class” corresponds to a set of “Requirements” within the corresponding “Requirements class”.

9.3.5. Conformance test

A “Conformance test” checks if a set of “Requirements” is met by a single standardization target (an entity).

A “Conformance test” has a many-to-many relation with “Requirements”.

A “Conformance test” is about a single standardization target.

A “Conformance test” can be “concrete” or “abstract” depending on the type of conformance test suite (see OGC 08-131r3, Clause 6.4). A concrete conformance test is typically called as a “conformance test”, while an abstract conformance test is called an “abstract test”.

9.3.6. Conformance test suite

A “Test suite” is “a collection of identifiable conformance classes” (see OGC 08-131r3, Clause 6.4)

A “Conformance test suite” contains only “Conformance classes” of the “concrete” kind. Such conformance class can only contain “concrete” conformance tests.

An “Abstract test suite” contains only “Conformance classes” of the “abstract” kind. Such conformance class can only contain Abstract tests.

9.4. ModSpec instantiation

ModSpec models are defined as classes. In order to create ModSpec models inside an OGC deliverable, it is necessary to “instantiate” them into ModSpec instances.

9.5. Encoding of ModSpec instances

9.5.1. General

A ModSpec instance is encoded in the Metanorma AsciiDoc markup language, via tagged blocks with definition lists, containing other tagged example blocks and open blocks.

NOTE 1 Metanorma also supports the OGC legacy “block attribute” syntax, but it is not described in this document since it is no longer recommended for the flexibility in the newer syntax.

This syntax requires specification of a `[%metadata]` definition list within a ModSpec instance, which provides the necessary information for the specified model. Values given in the definition list syntax can be fully-formatted Metanorma AsciiDoc text.

A ModSpec model instance is encoded with one of these block types:

- `[requirement]` for Requirement
- `[recommendation]` for Recommendation
- `[permission]` for Permission
- `[requirements_class]` for Requirements class
- `[conformance_test]` for Conformance test
- `[conformance_class]` for Conformance class
- `[abstract_test]` for Abstract test

NOTE 2These ModSpec types are available from [added in Metanorma OGC version v1.4.3]

In addition, if the Metanorma generic `[requirements]` block is used, these values are to be used in the `type` attribute.

The following two encodings are equivalent:

```
[conformance_test]
```

Figure 36

```
[requirement,type=conformance_test]
```

Figure 37

Attributes that can take rich textual input (Metanorma AsciiDoc input), such as `part`, `conditions`, and `guidance`, are components of requirements in Metanorma.

These can be encoded within the definition list, or in the block attributes syntax using the `[.component]` role within the ModSpec instance block, on open blocks or example blocks.

Example 1 — Example of encoding a ModSpec requirement "part" within the definition list

```
[requirement]
====
[%metadata]
identifier:: /req/world/hello
part:: Part A of the requirement.
====
```

Example 2 — Example of encoding a ModSpec requirement "part" in an open block syntax

```
[requirement]
====
[%metadata]
identifier:: /req/world/hello

[.component,class=part]
--
```

```
Part A of the requirement.
--
====
```

Example 3 — Example of encoding a ModSpec requirement "part" in an example block syntax

```
[requirement]
=====
[%metadata]
identifier:: /req/world/hello

[.component,class=part]
====
Part A of the requirement.
====
=====
```

The `%metadata` definition list may contain embedded levels [added in Metanorma OGC version v1.4.3]; this is needed specifically for steps embedded within a test method.

If you need to insert a cross-reference to a component, for example referencing a specific part of a requirement elsewhere, you can only use the block attributes sequence (as illustrated above).

```
[requirement]
.Encoding of logical models
====
[%metadata]
identifier:: /spec/waterml/2.0/req/xsd-xml-rules
subject:: system
part:: Metadata models faithful to the original UML model.
description:: Logical models encoded as XSDs should be faithful to the original
UML conceptual models.

test-method::
step::: Step 1
step::: Step 2
step:::: Step 2a
step:::: Step 2b
step::: Step 3
====
```

Figure 38 — ModSpec requirement with hierarchical test-method steps

When using ModSpec within other documents that, by default, uses another requirements model scheme (such as non-OGC flavors), it is necessary specify the instance with the `model` attribute.

Example 4 — Encoding a ModSpec instance within a document that uses another requirements model scheme

```
[requirement,model=ogc]
====
[%metadata]
identifier:: /req/iso-nnnnn/considerations

This is an OGC ModSpec requirement within an ISO document.
====
```

9.5.2. Instance attributes

Attributes accepted by a ModSpec instance are as follows:

`identifier` (mandatory) Identifier of the requirement, such as a URI or a URN. Plain text.

This must be unique in the document (as required by ModSpec), and is also used for referencing and cross-linking between ModSpec instances.

NOTE 1 The `identifier` was previously encoded as `label` until Metanorma OGC version v2.2.0 .

`subject` (optional) Subject that the model refers to. Plain text.

`obligation` (optional) Accepted values are one of:

`requirement`	(default) The instance is a requirement.
`recommendation`	The instance is a recommendation.
`permission`	The instance is a permission.

`description` (optional) The descriptive text for this instance.

NOTE 2 In a normative statement, the `description` key is treated as a synonym of `statement`, which forms the statement of compliance itself instead of informative, descriptive, text. [added in mn-requirements version v0.2.1].

`target` (conditional: only for conformance-related models) The "target" that is being tested against, specified with the identifier of the requirement or requirements class. (Replaces `subject` in that context.)

NOTE 3 The `target` is only supported in definition list syntax. [added in Metanorma OGC version v2.2.0]

- When in a conformance test (or an abstract test), specify the corresponding identifier of the requirement that is being tested.
- When in a conformance class, specify the corresponding identifier of the requirement class that is being tested.

Differentiated types of ModSpec models allow additional attributes.

9.5.3. Normative statement: requirement, recommendation, permission

Metanorma ModSpec supports the following normative statement types:

- Requirement (`requirement`)

- Recommendation (`recommendation`)
- Permission (`permission`)

The type of normative statement can be specified with using the above values as block types, or by setting the `type` attribute of a block.

It supports the following attributes in addition to base ModSpec attributes:

`statement` (mandatory) The statement to which compliance applies within this provision.

NOTE 1 Prior to mn-requirements v0.2.1, the key `description` is used. `description` is now a synonym for `statement` in a provision instance [added in mn-requirements version v0.2.1].

`conditions` (optional) Conditions on where this requirement applies. Accepts rich text.

`part` (optional) A requirement can contain multiple parts of sub-requirements. Accepts rich text. Labelled with a capital alphabetic letter.

NOTE 2A part is distinct from a step (as appears in Clause 9.5.6): a part is a component of a requirement, which is itself a requirement. A step is a stage in a process of testing a requirement: it only makes sense within a test method.

`guidance`	(optional) Guidance on how to apply the requirement. Used to avoid numbering of notes or examples as part of the overall document. Accepts rich text. Guidance is always rendered last in ModSpec. [added in mn-requirements version v0.1.4]
`inherit`	(optional) A requirement can inherit from one or more requirements (*direct dependency* in ModSpec terms). Accepts identifiers of other requirements: multiple values are semicolon-delimited. Can be repeated in definition list syntax.
`indirect-dependency`	(optional) A requirement can inherit indirectly from one or more requirement classes, which have a different standardization target from that of the requirement. That requirement class is used, produced, or associated with the current requirement, but its requirements are not inherited by this requirement. Only supported in definition list syntax. [added in Metanorma OGC version v2.2.1]
`implements`	(optional) A requirement can implement another requirement. Accepts identifiers of other requirements. Can be repeated in definition list syntax [added in mn-requirements version v0.1.9].
`classification`	(optional) Classification of this requirement. The `classification` attribute is marked up as in the rest of Metanorma: `key1=value1;key2=value2...`, where *value* is either a single string, or a comma-delimited list of values.
`requirement`, `permission`, `recommendation`	A requirement, permission, or recommendation contained within a requirement. The value of the element is its identifier. Only supported in definition list syntax.

`conformance-test`, `abstract-test`, `conformance-class`, `requirement-class` `recommendation-class`, `permission-class`:: A requirement, permission, or recommendation of those categories, contained within a requirement. The value of the element is its identifier. Only supported in definition list syntax. [added in mn-requirements version v0.1.6]

NOTE 3 `conditions`, `part` supported since [added in Metanorma OGC version v1.4.2].

NOTE 4 In the default rendering of ModSpec, the `statement` attribute, descriptions are labelled as *Statement* for requirements, recommendations, permissions. They are left as *Description* for all other kinds of ModSpec instances.

Example 1 — OGC CityGML 3.0 sample requirement with two parts (definition list)

```
[requirement]
====
[%metadata]
identifier:: /req/relief/classes
statement:: For each UML class defined or referenced in the Relief Package:
part:: The Implementation Specification SHALL contain an element which
represents the
same concept as that defined for the UML class.
part:: The Implementation Specification SHALL represent associations with the
same
source, target, direction, roles, and multiplicities as those of the UML class.
====
```

This renders as:

REQUIREMENT 1	
Identifier	`/req/relief/classes`
Statement	For each UML class defined or referenced in the Relief Package:
A	The Implementation Specification SHALL contain an element which represents the same concept as that defined for the UML class.
B	The Implementation Specification SHALL represent associations with the same source, target, direction, roles, and multiplicities as those of the UML class.

Example 2 — OGC CityGML 3.0 sample requirement with two parts (block attributes)

```
[requirement,identifier="/req/relief/classes"]
====
For each UML class defined or referenced in the Relief Package:

[.component,class=part]
--
The Implementation Specification SHALL contain an element which represents the
same concept as that defined for the UML class.
--

[.component,class=part]
--
The Implementation Specification SHALL represent associations with the same
source, target, direction, roles, and multiplicities as those of the UML class.
```

```
--
====
```

renders as:

Requirement 1	
/req/relief/classes	
For each UML class defined or referenced in the Relief Package:	
A	The Implementation Specification SHALL contain an element which represents the same concept as that defined for the UML class.
B	The Implementation Specification SHALL represent associations with the same source, target, direction, roles, and multiplicities as those of the UML class.

Figure 39

Example 3 — OGC CityGML 3.0 sample requirement with two parts

```
[requirement]
====
[%metadata]
identifier:: /req/core/encoding

All target implementations SHALL conform to the appropriate GroundWaterML2
Logical Model UML defined in Section 8.
====
```

OGC GroundWaterML 2.0 sample requirement

renders as:

Requirement 3
/req/core/encoding
All target implementations SHALL conform to the appropriate GroundWaterML2 Logical Model UML defined in Section 8.

Figure 40

9.5.4. Requirements class

A “Requirements class” is encoded as a block of `requirements_class` or using `type` equals to `requirements_class`.

A Requirements class is cross-referenced and captioned as a “{Requirement} class {N}” [added in Metanorma OGC version v0.2.11].

NOTE 1 Classes for Recommendations will be captioned as “Recommendations class {N}”, similarly for “Requirements class {N}” and “Permissions class {N}”.

Requirements classes allow the following attributes in addition to the base ModSpec attributes:

Name	(mandatory) Name of the requirements class should be specified as the block caption.
`subject`	(mandatory) The Target Type. Rendered as *Target Type*.
`inherit`	(optional) Dependent requirements classes. See Requirement, recommendation, permission.
`indirect-dependency`	(optional) Indirect dependent requirements classes. See Requirement, recommendation, permission.
`guidance`	(optional) Guidance on requirement class. See Requirement, recommendation, permission.
Embedded requirements (optional)	Requirements contained in a class are marked up as nested requirements.

Example 1 — Example from OGC CityGML 3.0

```
[requirements_class]
====
[%metadata]
identifier:: http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-building
subject:: Implementation Specification
inherit:: /req/req-class-core
inherit:: /req/req-class-construction
====
```

Renders as:

Requirements class 1	
http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-building	
Target type	Implementation Specification
Dependency	/req/req-class-core
Dependency	/req/req-class-construction

Figure 41

NOTE 2 In this example, both block attributes and definition list syntax is used; the `inherit` attribute has two values, which are expressed in the definition list.

A requirements class can contain multiple requirements, specified with embedded requirements.

The contents of these embedded requirements may be specified within the requirements class, or specified outside of the requirements class (referenced using the identifier). If the requirement is specified within a definition list, the definition list value is interpreted as the requirement identifier.

Example 2 — Example from OGC GroundWaterML 2.0

```
[requirements_class]
.GWML2 core logical model
====
[%metadata]
```

```
identifier:: http://www.opengis.net/spec/waterml/2.0/req/xsd-xml-rules[*req/
core*]
obligation:: requirement
subject:: Encoding of logical models
inherit:: urn:iso:dis:iso:19156:clause:7.2.2
inherit:: urn:iso:dis:iso:19156:clause:8
inherit:: http://www.opengis.net/doc/IS/GML/3.2/clause/2.4
inherit:: O&M Abstract model, OGC 10-004r3, clause D.3.4
inherit:: http://www.opengis.net/spec/SWE/2.0/req/core/core-concepts-used
requirement:: /req/core/encoding
requirement:: /req/core/quantities-uom
====
```

renders as:

Requirement Class 1 GWML2 core logical model	
req/core	
Obligation	Requirement
Target Type	Encoding of logical models
Dependency	urn:iso:dis:iso:19156:clause:7.2.2
Dependency	urn:iso:dis:iso:19156:clause:8
Dependency	http://www.opengis.net/doc/IS/GML/3.2/clause/2.4
Dependency	O&M Abstract model, OGC 10-004r3, clause D.3.4
Dependency	http://www.opengis.net/spec/SWE/2.0/req/core/core-concepts-used
Requirement	/req/core/encoding
Requirement	/req/core/quantities-uom

Embedded requirements (such as are found within Requirements classes) will automatically insert cross-references to the non-embedded requirements with the same identifier [added in Metanorma OGC version v1.0.8].

Example 3 — Example of specifying embedded requirements within a ModSpec instance

```
[requirements_class,identifier="/req/conceptual"]
.GWML2 core logical model
====

[requirement,identifier="/req/core/encoding"]
======
======
```

```
====

[requirement,identifier="/req/core/encoding"]
====
Encoding requirement
====
```

renders as:

Requirement Class 3: GWML2 core logical model /req/conceptual	
Requirement 1	/req/core/encoding

Requirement 1 /req/core/encoding
Encoding requirement

9.5.5. Conformance class

Specified by setting the block as `conformance_class` or by using `type` as `conformance_class`.

A Conformance class is cross-referenced and captioned as “Conformance class {N}”, and is otherwise rendered identically to a “Requirements class” [added in Metanorma OGC version v1.0.4].

Conformance classes support the following attributes in addition to base ModSpec attributes:

`target` (mandatory) Associated Requirements class. Populated with the identifier of the Requirements class. Rendered as *Requirements Class*.

`inherit` (optional) Dependencies of the conformance class. Accepts multiple values, which are the identifiers of other requirements. See Requirement, recommendation, permission.

`indirect-dependency` (optional) Indirect dependent requirements classes. See Requirement, recommendation, permission.

Conformance classes also feature:

Name (optional) Specified as the block caption.

Nesting (optional) Conformance tests contained in a conformance class are encoded as conformance tests within the conformance class block, marked as `conformance-test`. See Requirements class.

NOTE Conformance classes do not have a Target Type (as specified in ModSpec). If one must be encoded, it should be encoded as a classification key-value pair.

Example — Example of encoding a conformance class

```
[conformance_class]
====
[%metadata]
identifier:: http://www.opengis.net/spec/ogcapi-features-2/1.0/conf/crs
target:: http://www.opengis.net/spec/CityGML-1/3.0/req/req-class-building
indirect-dependency:: http://www.opengis.net/doc/IS/ogcapi-features-1/1.0#ats_
core
classification:: Target Type:Web API
====
```

renders as:

CONFORMANCE CLASS 1	
http://www.opengis.net/spec/ogcapi-features-2/1.0/conf/crs	
Requirements Class	*Requirements Class 'Coordinate Reference Systems by Reference'*
Dependency	http://www.opengis.net/doc/IS/ogcapi-features-1/1.0#ats_core
Target Type	Web API

9.5.6. Conformance test and Abstract test

A "Conformance test" can be "concrete" or "abstract" depending on the type of conformance test suite (see OGC 08-131r3, 6.4).

The OGC author should identify whether a standard requires an "Abstract test suite" or a "Conformance test suite" in order to decide the encoding of "Conformance tests" (concrete tests) versus "Abstract tests".

- A conformance test is specified by creating a `conformance_test` block or using `type` as `conformance_test`. It is cross-referenced as "Conformance test {N}"
- An abstract test is specified by creating an `abstract_test` block or using `type` as `abstract_test`, or `conformance_test` together with `abstract=true`. It is cross-referenced as "Abstract test {N}" [added in Metanorma OGC version v1.0.4].

Conformance tests support the following attributes and components in addition to base ModSpec attributes:

`target` The associated requirement. Populated with the identifier of the requirement. Multiple semicolon-delimited values may be provided. Rendered as *Requirement*.

`inherit` (optional) Dependencies. Accepts multiple values, which are the identifiers of other requirements. See Requirement, recommendation, permission.

- `indirect-dependency` (optional). Indirect dependent requirements classes. See Requirement, recommendation, permission.

Components (optional) Components of the conformance test. Accepts rich text. [added in Metanorma OGC version v1.4.0]. Allows the following classes:

`test-purpose` (optional) Purpose of the test. Rich text. Presented as *Test Purpose* [added in Metanorma OGC version v1.4.2]

`test-method` (optional) Method of the test. Rich text. Presented as *Test Method* [added in Metanorma OGC version v1.4.2]

`step` (optional) Step of the test method. Is expected to be embedded within `test-method`, and may contain substeps of its own. Rich text. Presented as a numbered list. added in Metanorma OGC version v1.4.2].

Steps can be nested, the nested list order is: *arabic*, then *alphabetic*, then *roman*.

`test-method-type` (optional) Method of the test. Rich text. Presented as *Test Method Type* [added in Metanorma OGC version v1.4.3]

`reference` (optional) Purpose of the test. Rich text. Presented as *Reference*.

Test type The test type of a Conformance test is encoded as a `classification` key-value pair.

Conformance tests also feature:

- Name (optional). Specified as the requirement's block caption.

NOTEConformance Tests are excluded from the "Table of Requirements" in Word output [added in Metanorma OGC version v0.2.10].

Example 1 — Example of Abstract test from CityGML 3.0

```
[abstract_test]
====
[%metadata]
identifier:: /conf/core/classes

target:: /req/core/classes

test-purpose:: To validate that the Implementation Specification correctly
implements the UML Classes defined in the Conceptual Model.

test-method-type:: Manual Inspection

description:: For each UML class defined or referenced in the Core Package:

part:: Validate that the Implementation Specification contains a data element
which represents the same concept as that defined for the UML class.

part:: Validate that the data element has the same relationships with other
elements as those defined for the UML class. Validate that those relationships
have the same source, target, direction, roles, and multiplicities as those
documented in the Conceptual Model.
====
```

renders as:

Abstract test 1	
/ats/core/classes	
Requirement	/req/core/classes
Test purpose	To validate that the Implementation Specification correctly implements the UML Classes defined in the Conceptual Model.
Test-method-type	Manual Inspection
For each UML class defined or referenced in the Core Package:	
A	Validate that the Implementation Specification contains a data element which represents the same concept as that defined for the UML class.
B	Validate that the data element has the same relationships with other elements as those defined for the UML class. Validate that those relationships have the same source, target, direction, roles, and multiplicities as those documented in the Conceptual Model.

Figure 42

Example 2 — Example of Abstract test from DGGS

```
[abstract_test]
====
[%metadata]
identifier:: /conf/crs/crs-uri
target:: /req/crs/crs-uri
target:: /req/crs/fc-md-crs-list-A
target:: /req/crs/fc-md-storageCrs
target:: /req/crs/fc-md-crs-list-global
classification:: Test Type:Basic
test-purpose:: Verify that each CRS identifier is a valid value
test-method::
+
--
For each string value in a `crs` or `storageCrs` property in the collections
and collection objects,
validate that the string conforms to the generic URI syntax as specified by
https://tools.ietf.org/html/rfc3986#section-3[RFC 3986, section 3].

. For http-URIs (starting with `http:`) validate that the string conforms to
the syntax specified by RFC 7230, section 2.7.1.

. For https-URIs (starting with `https:`) validate that the string conforms to
the syntax specified by RFC 7230, section 2.7.2.
--
reference:: <<ogc_07_147r2,clause=15.2.2>>
====
```

renders as:

ABSTRACT TEST 1	
/conf/crs/crs-uri	
Requirement	*/req/crs/crs-uri*, */req/crs/fc-md-crs-list A*, */req/crs/fc-md-storageCrs*, */req/crs/fc-md-crs-list-global*
Test Purpose	Verify that each CRS identifier is a valid value
Test Method	For each string value in a `crs` or `storageCrs` property in the collections and collection objects, validate that the string conforms to the generic URI syntax as specified by RFC 3986, section 3. 1. For http-URIs (starting with `http:`) validate that the string conforms to the syntax specified by RFC 7230, section 2.7.1. 2. For https-URIs (starting with `https:`) validate that the string conforms to the syntax specified by RFC 7230, section 2.7.2.
Reference	OGC-07-147r2: cl. 15.2.2
Test Type	Basic

9.6. Cross-referencing ModSpec instances

9.6.1. General

Similar to when specifying attributes for ModSpec instances, it is preferred to refer to other instances using identifiers, rather than the numbered labels allocated by default.

Example: In OGC, it is preferred to show the identifier of a ModSpec instance in a cross-reference, like `http://www.example.com/req/crs/crs-uri` instead of *Requirement Class 6*.

9.6.2. Referencing using predefined anchors

This can be extended to cross-references. If the anchor of the requirement is known, a normal cross-reference can be marked up, as shown below.

Example — Cross-reference to a ModSpec instance using a predefined anchor

```
<<id1,http://www.example.com/req/crs/crs-uri>>
```

Renders (assuming that this is the 10th Requirement):

Requirement 10

9.6.3. Referencing using instance identifiers

However, not all ModSpec instances are assigned predefined anchors, especially when using model-based generation. It also precludes automated manipulation of the identifier base path.

For that reason, Modspec in Metanorma supports anchor aliasing: the identifier of the requirement can be used in cross-references as an alias of the anchor.

Metanorma will automatically map the anchor it allocates to requirements to identifiers, to that end: users do not need to supply the anchor alias mappings manually.

So for a requirement such as:

```
[[id1]]
[requirement]
====
identifier:: http://www.example.com/req/crs/crs-uri
====
```

Figure 43

It is possible to reference a ModSpec instance using its identifier instead of the anchor, as follows.

Example 1 — Cross-reference to a ModSpec instance using its identifier, displaying the instance's name

```
xref:http://www.example.com/req/crs/crs-uri[]
```

Renders (assuming that this is the 10th Requirement):

Requirement 10

Metanorma treats them as fully equivalent, and will render them in the same way, as a numbered label (*Requirement Class 6*).

NOTE As a limitation of syntax, URIs cannot be processed correctly within `[_..]`. The `xref:…[]` command needs to be used instead.

To make the cross-reference render the identifier value of the instance itself, while still hyperlinking to the correct identifier, you can specify `style=id%` as the cross-reference text, as follows.

Example 2 — Cross-reference to a ModSpec instance using its identifier, displaying the instance's identifier

```
xref:http://www.example.com/req/crs/crs-uri[style=id%]
```

Renders as:

http://www.example.com/req/crs/crs-uri

This will also highlight the URI text as subject to truncation, with reference to identifier bases.

9.6.4. Identifier base pattern

NOTE 1 This functionality is first implemented in [added in mn-requirements version v0.2.1].

A ModSpec instance can be cross-referenced from other parts of the document, with the reference text used to identify the ModSpec instance named either according to its:

- instance label (e.g. “Requirement 3”); or
- identifier (e.g. `http://www.opengis.net/spec/waterml/2.0/req/xsd-xml-rules`)

ModSpec instances need to be assigned unique identifiers, which are typically either URIs, URNs or URLs.

These identifier types utilize a hierarchical pattern. If two identifiers share a common prefix, it means that the two identifiers can be grouped semantically at some level.

In well-structured standards (in OGC and others), ModSpec instances often share a common identifier prefix. For example, a defined, document-wide identifier prefix is used as the “base” for all ModSpec identifiers.

Example 1 — Document-wide identifier prefix with ModSpec instances using that prefix: OGC WaterML 2.0 applies a document identifier prefix:

- document identifier prefix: `http://www.opengis.net/spec/waterml/2.0`
- sample of a ModSpec instance identifier in the document: `http://www.opengis.net/spec/waterml/2.0/req/xsd-xml-rules`

When cross-referencing a ModSpec instance using its identifier, the references can be lengthy to read.

If a document-wide identifier “base prefix” is defined, Metanorma will omit the base prefix in the rendering of ModSpec instances when using the identifier as reference text.

There are the following ways of specifying an identifier base prefix:

Document-wide	The document attribute `:modspec-identifier-base:` is used to specify the identifier base prefix for the entire document.
ModSpec class instance	An identifier base prefix can be defined inside a ModSpec class instance (e.g. Requirements class), using the definition list tag `identifier-base`.
ModSpec instance	An identifier base prefix can be defined inside a ModSpec instance (e.g. Requirement), using the definition list tag `identifier-base`.

The behavior is specified as follows:

- If an identifier base prefix is specified document-wide:
  - When a ModSpec instance or class instance is cross-referenced using its identifier, the identifier base prefix will be removed from the identifier in the reference text.

- If an identifier base prefix is specified on a ModSpec class instance (e.g. Requirements class):
  - This identifier base prefix overrides any value specified in `:modspec-identifier-base:`, if any.
  - The identifier base prefix specified will apply to all its ModSpec instances (e.g. Requirements in the Requirements class) unless overridden.
  - When a ModSpec class instance is cross-referenced using its identifier, the identifier base prefix will be removed from the identifier in the reference text.

- If an identifier base prefix is specified on a ModSpec instance (e.g. Requirement):
  - The identifier base prefix specified on the instance overrides all higher level identifier base prefixes;
  - The identifier base prefix specified on the instance's class (e.g. Requirements class) overrides any value specified in `:modspec-identifier-base:`, if any;
  - When the instance is cross-referenced using its identifier, the identifier base prefix will be removed from the identifier in the reference text.

NOTE 2An identifier base specified on a requirement applies to all ModSpec requirement cross-references rendered within that requirement. The identifier base truncation is applied to cross-references rendered as just the identifier (`style=id%`), but it is also applied to the identifiers incorporated inside of normal cross-references, and to the identifier labels of requirements.

Example 2 — Setting a document-wide identifier base prefix

```
:modspec-identifier-base: http://www.example.com

Refer to
xref:http://www.example.com/req/class1[] and
xref:http://www.example.com/req/class1/req1[style=id%].

[requirements_class]
```

identifier http://www.example.com/req/class1

requirement http://www.example.com/req/class1/req1

description Some description.

Example 3:

identifier http://www.example.com/req/class1/req1

statement A requirement.

Example 4

```
Renders as:

____

Refer to
/req/class1 and /req/class1/req1.

|===
2+| Requirements class 1

h| Identifier          | `/req/class1/`
h| Normative statement | Requirement 1: `/req/class1/req1`
h| Description         | Some description.
|===

|===
2+| Requirement 1

h| Identifier  | `/req/class1/req1`
h| Included in | Requirement class 1: `/req/class1`
h| Statement   | A requirement.
|===
____
```

Example 5 — Setting a identifier base prefix at a class instance

```
[requirement,type=requirements_class]
```

identifier http://www.example.com/req/class1

identifier-base http://www.example.com/req

requirement http://www.example.com/req/class1/req1

description Some description.

Example 6:

identifier http://www.example.com/req/class1/req1

statement A requirement.

Example 7

```
Renders as:

____
|===
2+| Requirements class 1

h| Identifier          | `/class1/`
h| Normative statement | Requirement 1: `/class1/req1`
h| Description         | Some description.
|===
```

```
|===
2+| Requirement 1

h| Identifier  | `/class1/req1`
h| Included in | Requirements class 1: `/class1`
h| Statement   | A requirement.
|===
----
```

Example 8 — Setting identifier base prefixes for document-wide and at the class instance level

```
:modspec-identifier-base: http://www.example.com

[requirement-class]
----
identifier:: http://www.example.com/req/class1
identifier-base:: http://www.example.com/req
requirement:: http://www.example.com/req/class1/req1
----

[requirement-class]
----
identifier:: http://www.example.com/req/class2
requirement:: http://www.example.com/req/class2/req2
----

[requirement]
----
identifier:: http://www.example.com/req/class1/req1
statement:: See also xref:http://www.example.com/req/class2/req2[style=id%].
----

[requirement]
----
identifier:: http://www.example.com/req/class2/req2
statement:: See also xref:http://www.example.com/req/class1/req1[].
----
```

Renders as:

REQUIREMENT CLASS 1	
IDENTIFIER	`/class1`
NORMATIVE STATEMENT	Requirement 1: `/class1/req1`

REQUIREMENT CLASS 2	
IDENTIFIER	`/req/class2`
NORMATIVE STATEMENT	Requirement 2: `/req/class2/req2`

REQUIREMENT 1	
IDENTIFIER	`/class1/req1`
INCLUDED IN	Requirements class 1: `/class1`
STATEMENT	See also /class2/req2

REQUIREMENT 2	
IDENTIFIER	`/req/class2/req2`
INCLUDED IN	Requirements class 2: `/req/class2`
STATEMENT	See also Requirement 1: `/req/class1/req1`

9.7. Rendering of ModSpec instances

ModSpec instances are rendered in a table format.

NOTE 1 This rendering method is consistent with prior OGC ModSpec practice.

- For HTML rendering, the CSS class of the ModSpec specification table is the `type` attribute of the requirement.

  The following types are recognised:

  - No value for Requirements
  - `conformance_test` for Conformance tests
  - `abstract_test` for Abstract tests
  - `requirements_class` for Requirements classes
  - `conformance_class` for Conformance classes

  NOTE 2The default CSS class currently assigned for HTML rendering is `recommend`.

- The heading of the table (spanning two columns) is its name (the role or style of the requirement, e.g. `[permission]` or `[.permission]`), optionally followed by its title (the caption of the requirement, e.g. `.Title`).

- The title of the table (spanning two columns) is its `identifier` attribute.
- The initial rows of the body of the table give metadata about the requirement. They include:
    - The `obligation` attribute of the requirement, if given: *Obligation* followed by the attribute value
    - The `subject` attribute of the requirement, if given: *Subject*, followed by the attribute. The subject attribute can be marked up as a cross-reference to another requirement given in the same document. If there are multiple values of the subject, they are semicolon delimited [added in https://github.com/metanorma/metanorma-standoc/releases/tag/v1.10.4].
    - The `inherit` attribute of the requirement, if given: *Dependency* followed by the attribute value. If there are multiple values of the attribute, they are semicolon delimited.
    - The `indirect-dependency` attribute of the requirement, if given: *Indirect Dependency* followed by the attribute value. If there are multiple values of the attribute, they are semicolon delimited.
    - The `classification` attributes of the requirement, if given: the classification tag (in capitals), followed by the classification value.

- The remaining rows of the requirement are the remaining components of the requirement, encoded as table rows instead of as a definition table (as they are by default in Metanorma).
    - These include the explicit `component` components of the requirement [added in Metanorma OGC version v1.4.0], which capture internal components of the requirement defined in ModSpec.

      These are divided into two categories:

        - Components with a `class` attribute other than `part` are extracted in order, with the class name normalised (title case), followed by the component contents. So a component with a `class` attribute of `conditions` will be rendered as *Conditions* followed by the component contents. In the foregoing, we have seen components defined in ModSpec: `test-purpose`, `test-method`, `test-method-type`, `conditions`, `reference`. However the block attribute syntax allows open-ended component names.
        - Components with the `class` attribute `part` are extracted and presented in order: each Part is rendered as an incrementing capital letter (*A*, *B*, *C* and so on), followed by the component contents. Any cross-references to part components will automatically be labelled with the identifier of their parent requirement, followed by their ordinal letter.

- Components can include descriptive text (`description`), which is interleaved with other components.
- Components can include open blocks marked with role attributes. That includes the legacy Metanorma components:
  - `[.specification]`
  - `[.measurement-target]`
  - `[.verification]`
  - `[.import]`

10

RENDER UML MODELS

10 RENDER UML MODELS

10.1. Render UML models with LutaML

OGC uses the Metanorma toolchain for publishing standards. The steps involved in transforming UML models into an MDS can be simplified as the conversion from UML models into Metanorma syntax. This clause describes in detail how this conversion step is performed.

OGC (through Testbed-17) has developed an automated workflow that provides a default UML rendering template set to render each UML class and package in the same way. This workflow uses the LutaML plugin to render the UML model's contents into document elements, called the `[lutaml_uml_datamodel_description]` block.

The `[lutaml_uml_datamodel_description]` block is used to iterate through a sequence of UML packages, rendering each in a consistent way. The rendering template for each type of UML element is predefined. Users do not have to supply their own template text unless overriding is needed.

NOTE LutaML uses Liquid as its templating language.

10.2. Exporting an MDS-readable model from EA

In order to make its information accessible to the MDA process, the UML models and associated information needs to be exported into an interoperable format.

Enterprise Architect version 16 onwards uses a proprietary binary format called `qea`, which is not readable outside of the application itself.

The interoperable format used in the OGC MDS process is the OMG UML format exported as OMG XMI (XML Model Interchange) (OMG XMI 2.5) format, as an XML file with the extension of `xmi`.

To export a UML Package (top-level package or one of the packages), first select the UML Package to be exported, then click on "Publish As..." as shown in Figure 44.

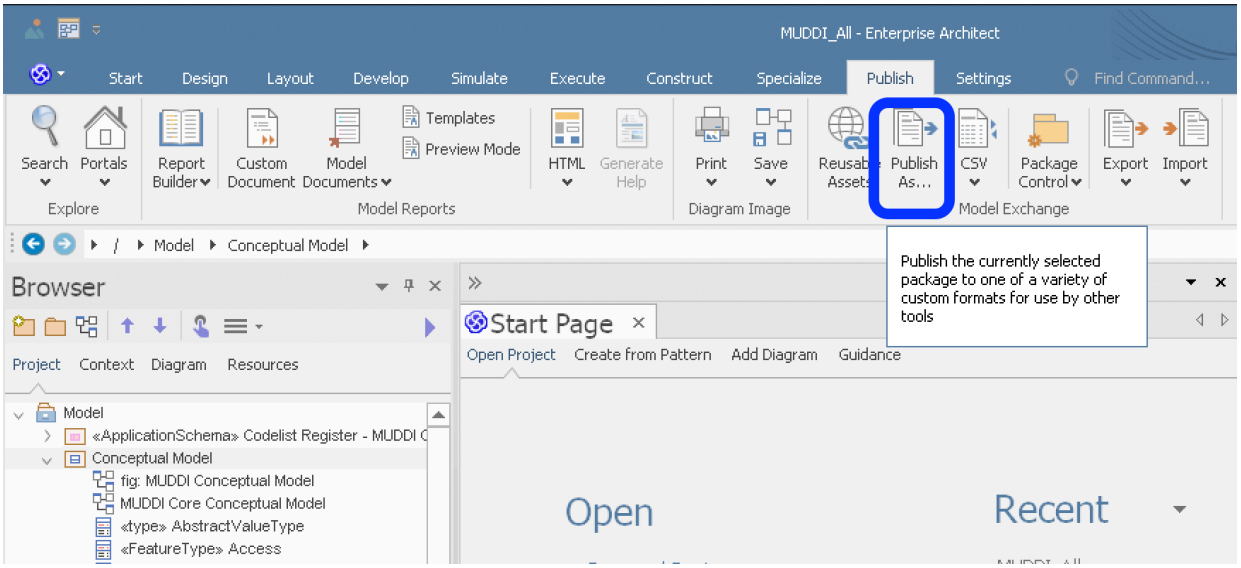


Figure 44 — Location of the "Publish As..." button

Clicking on the “Publish As...” button opens a dialog box with the options shown in Figure 45.

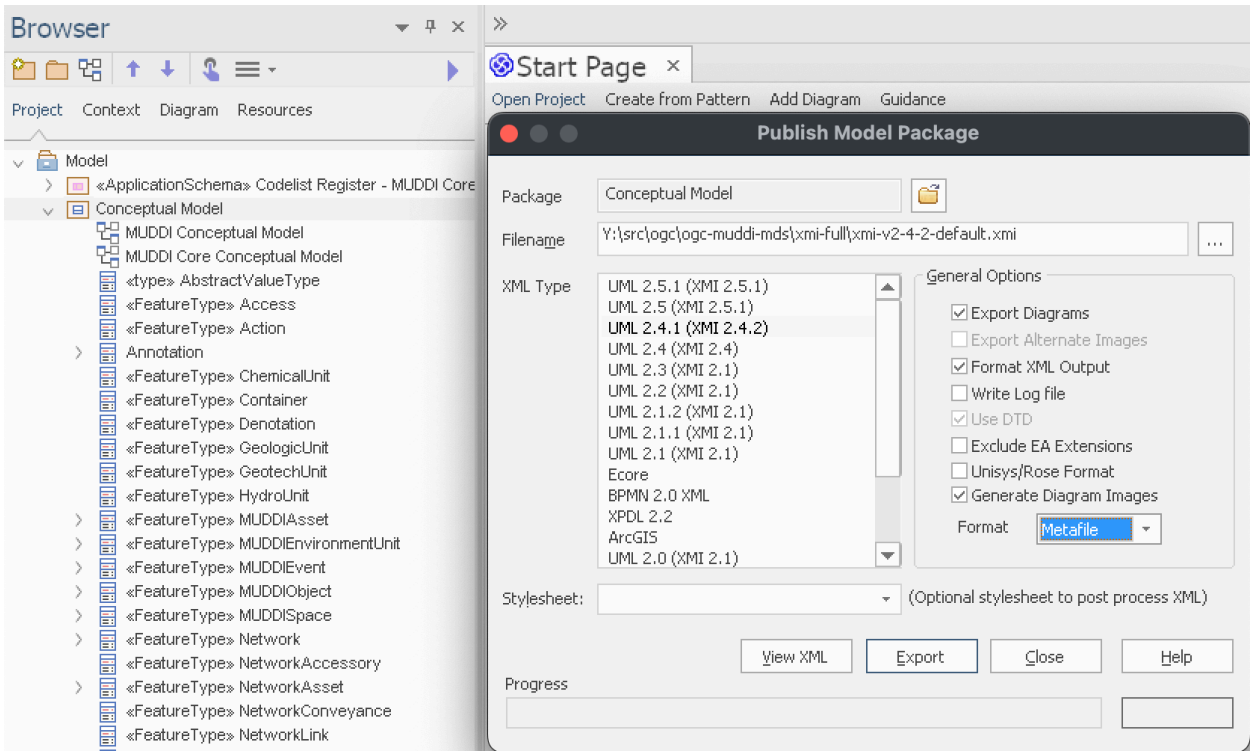


Figure 45 — Generation options for an XMI that works with Metanorma

The user will need to export the file with the following configuration set:

- Filename change the file extension to use `.xmi` in the “...” dialog box
- XML Type set to “UML 2.4.1 (XMI 2.4.2)”

- Check the following boxes in “General Options”:
  - Export Diagrams
  - Format XML Output
  - Generate Diagram Images, set Format to “SVG”
- Click on “Export”

NOTEThe Format “SVG” option is supported from EA version 16.1. Prior to 16.1, EMF was the only vector image format.

When these steps are followed the exported XMI will be at the path specified, ready to serve as input for the MDS process.

The resulting output will be placed in the selected directory as seen in Example 1. Note that the UML diagrams will be exported under a new directory called `Images/` under the selected directory.

Example 1 — Example of EA-exported XMI with EMF images

```
working-directory/
+- xmi-v2-4-2-default.xmi
+- UML_EA.dtd
+- Images/
   +- EAID_40625194_4483_46b2_80CF_2756F08865D8.svg
   +- EAID_76FDCDFB_19E5_47b6_9D21_E6450814059F.svg
   +- EAID_9499129E_BD74_4df2_9AC5_680582E4CD47.svg
```

For typical UML diagrams, the “SVG” format exports into `*.svg` files, and work best since they are vector images. SVG images allow for perfect scaling in PDF output and in HTML web browsers.

However, the EA SVG export functionality can occasionally fail to produce accurate results, especially for complex UML diagrams that involve custom relationships and lines.

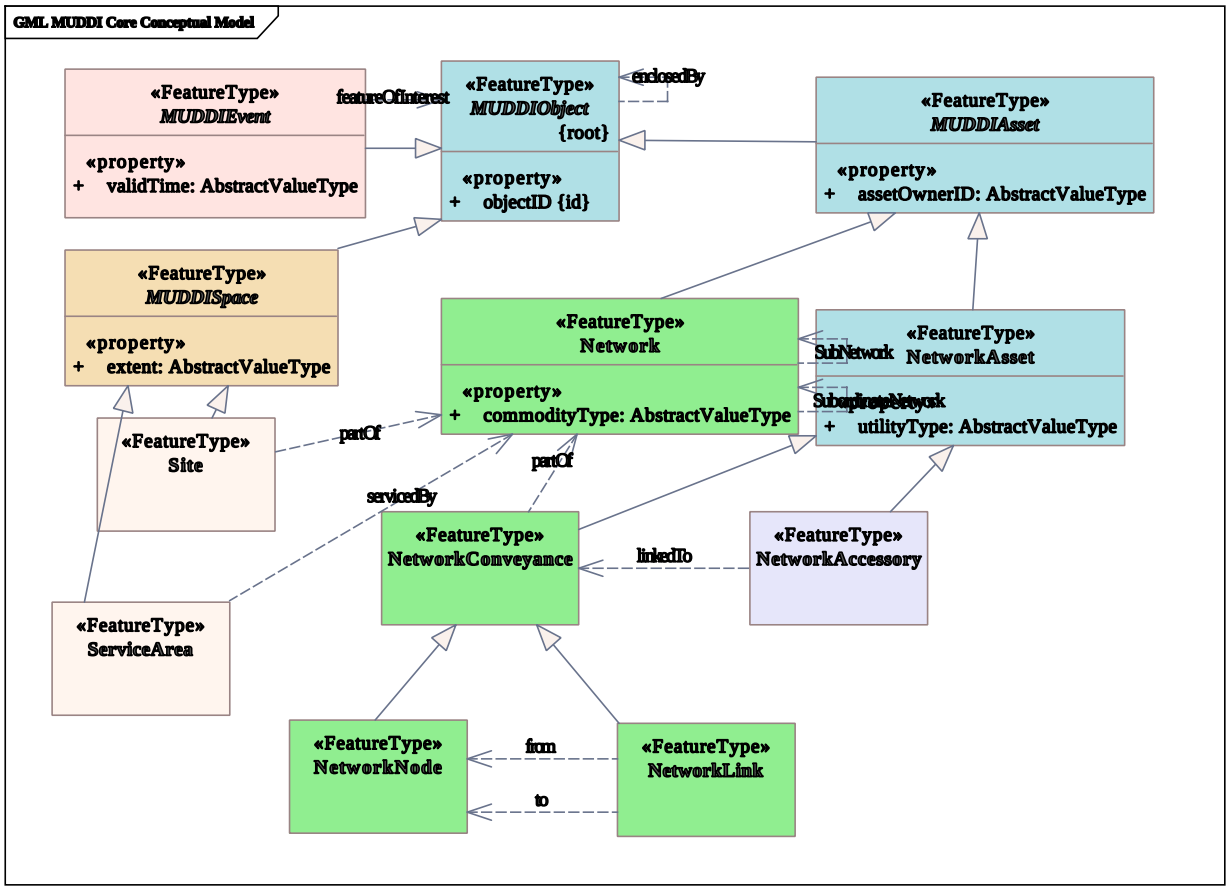


Figure 46-1 — EA-generated SVG file containing inaccurate layout

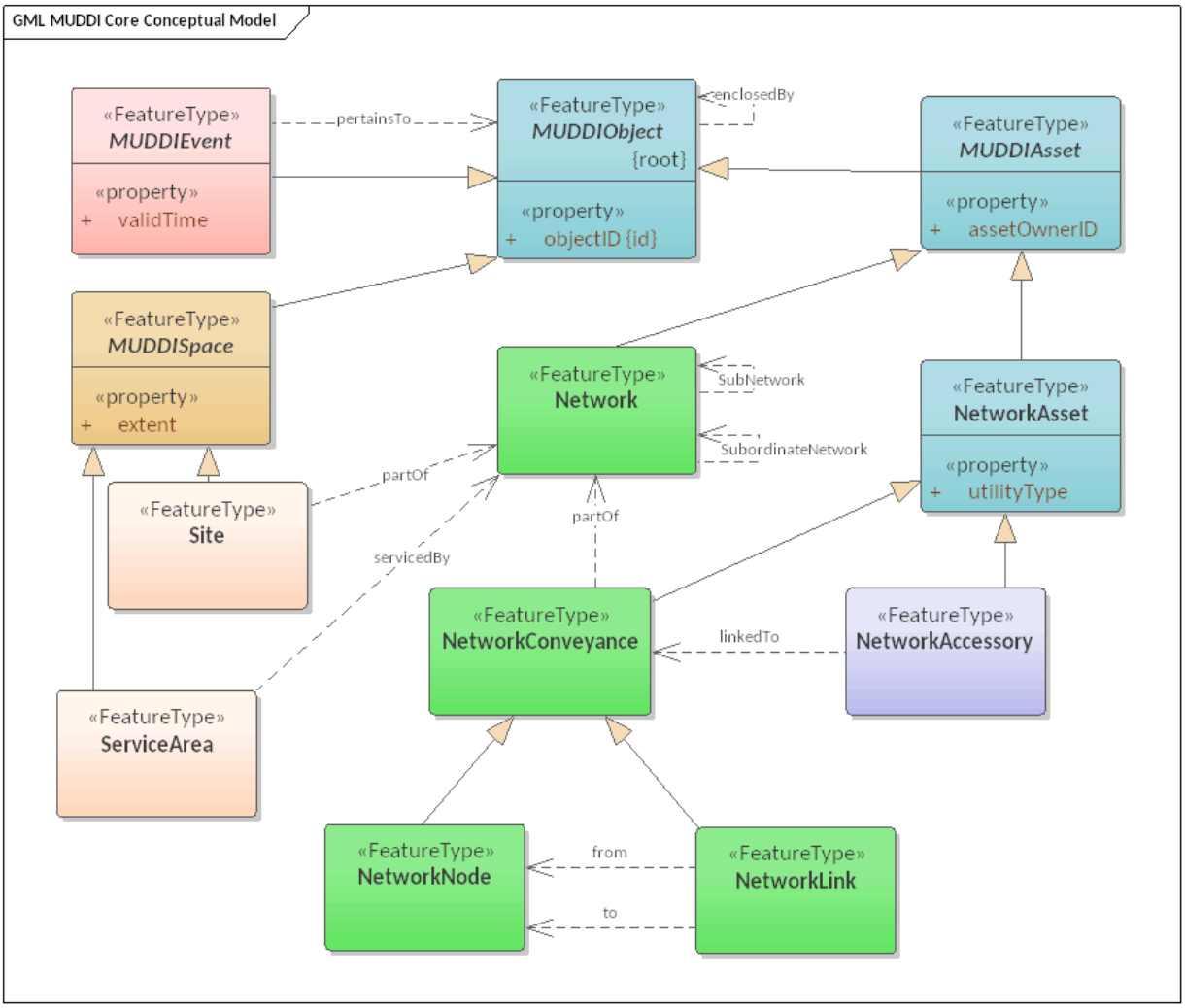


Figure 46-2 — EA-generated PNG file with correct layout

Figure 46 — Example of failed EA exported SVG

In this case, the follow additional steps will also export PNG images in the same directory:

- Filename change the file extension to use `.xmi` in the "..." dialog box
- XML Type set to "UML 2.4.1 (XMI 2.4.2)"
- Check the following boxes in "General Options":
  - Export Diagrams
  - Format XML Output
  - Generate Diagram Images, set Format to "PNG"
- Click on "Export"

If you specify the same location for exporting PNG images, they will be placed alongside the previously generated SVG images as shown in Example 2

Example 2 — Example of EA-exported XMI with EMF and PNG images

```
working-directory/
+- xmi-v2-4-2-default.xmi
+- UML_EA.dtd
+- Images/
   +- EAID_40625194_4483_46b2_80CF_2756F08865D8.svg
   +- EAID_40625194_4483_46b2_80CF_2756F08865D8.png
   +- EAID_76FDCDFB_19E5_47b6_9D21_E6450814059F.svg
   +- EAID_76FDCDFB_19E5_47b6_9D21_E6450814059F.png
   +- EAID_9499129E_BD74_4df2_9AC5_680582E4CD47.svg
   +- EAID_9499129E_BD74_4df2_9AC5_680582E4CD47.png
```

10.3. Basic usage

Basic usage of the `[lutaml_uml_datamodel_description]` command is given in Figure 47.

```
[lutaml_uml_datamodel_description,path/to/example.xmi]
--
--
```

Figure 47 — Basic usage of the `lutaml_uml_datamodel_description` block

`lutaml_uml_datamodel_description` declares the type of this block; `path/to/example.xmi` is the path to the OMG XMI file.

This command generates a Metanorma representation of the UML elements contained in the XMI file `path/to/example.xmi`.

By default, this block will iterate through the entire XMI file:

- Including all diagrams as figures in the MDS
- Rendering all UML elements hierarchically in the order of Package, Classes, Attributes, Associations, etc.

10.4. Configuration file

10.4.1. General

The behavior of the `lutaml_uml_datamodel_description` can be customized through providing a configuration file in YAML, as shown in Figure 48.

```
[lutaml_uml_datamodel_description,path/to/example.xmi,config.yaml]
--
--
```

Figure 48 — Configuring behavior of the `lutaml_uml_datamodel_description` block

`config.yaml` is the path to a YAML config file for the `lutaml_uml_datamodel_description` block.

The `config.yaml` parameter is optional. The nominated YAML file specifies which packages to process in the command, in which order; rendering style instructions; and the location of the root package.

The syntax of the YAML file is described in Figure 49.

```
---
packages: <1>
  # includes these packages
  - "Package *"
  - two*
  - three
  # skips these packages
  - skip: four
render_style: data_dictionary <2>
section_depth: 2 <3>
package_root_level: 2 <4>
<1> The packages key.
<2> The render_style key.
<3> The section_depth key.
<4> The package_root_level key.
```

Figure 49 — YAML configuration for `lutaml_uml_datamodel_description` command

All keys in the configuration files are optional.

10.4.2. Package inclusion

The `packages` key accepts an array of package name specifications that describes which packages to be included or excluded. The filter execution order is in the sequence of specification.

Specifically, any package that matches the given pattern (supporting regular expression matches) will be included in output.

Example 1: The regular expression “three” will only match the package name “three”, which will be included in the rendered output.

Example 2: The regular expression `Package *` will match “Package 1”, “Package X” and “Package This-And-That”.

To exclude packages, a syntax of `skip: {name}` is used for the package name specification. If a package was included in one of the matches, a skip rule that matches will cause that package to be skipped.

Example 3: The specification “skip: four” will specifically skip the package named “four” even if it was included in one of the matches prior to the skip rule.

10.4.3. Rendering style

10.4.3.1. General

The `render_style` value indicates the automated generation style to be used.

The generation style affects:

- the clause hierarchical structure;
- the content rendered from the generated UML models.

There are 3 types of UML rendering styles:

- `default`: the default manner to render UML packages and classes;

  NOTEThe default style is used for OGC 20-040r3.

- `entity_list`: the entity list style which is used in CityGML;
- `data_dictionary`: the data dictionary style which is used in CityGML.

10.4.3.2. Default style

The default style is considered the style to use for new MDS documents as it provides an OGC accepted order and rendering of UML components.

In the default style, the following steps are taken:

1. For every UML package (“package-name”):

    a) Render an overview subclause for the package, titled “{package-name} overview”, with the following content:

        i) If this package contains subpackages, render the following:

            - “The {package-name} package is organized into {sub-package-count} packages:”
            - Each sub-package is then listed out

        ii) Figures included in the top-most level of the package are rendered.

    b) For every sub-packages, recurse as per step 1.

    c) Render defining tables for every element in this package (in the order of Class / Interface / Union / DataType) according to this list of information:

        - Name
        - Definition
        - Stereotype
        - Inheritance from (optional)
        - Generalization of (optional)
        - Abstract
        - Associations: Association with; Obligation; Maximum occurrence; Provides
        - Public attributes: Name; Definition; Derived; Obligation; Maximum occurrence; Data type
        - Constraints

An example of the `default` style is shown in Figure 50.

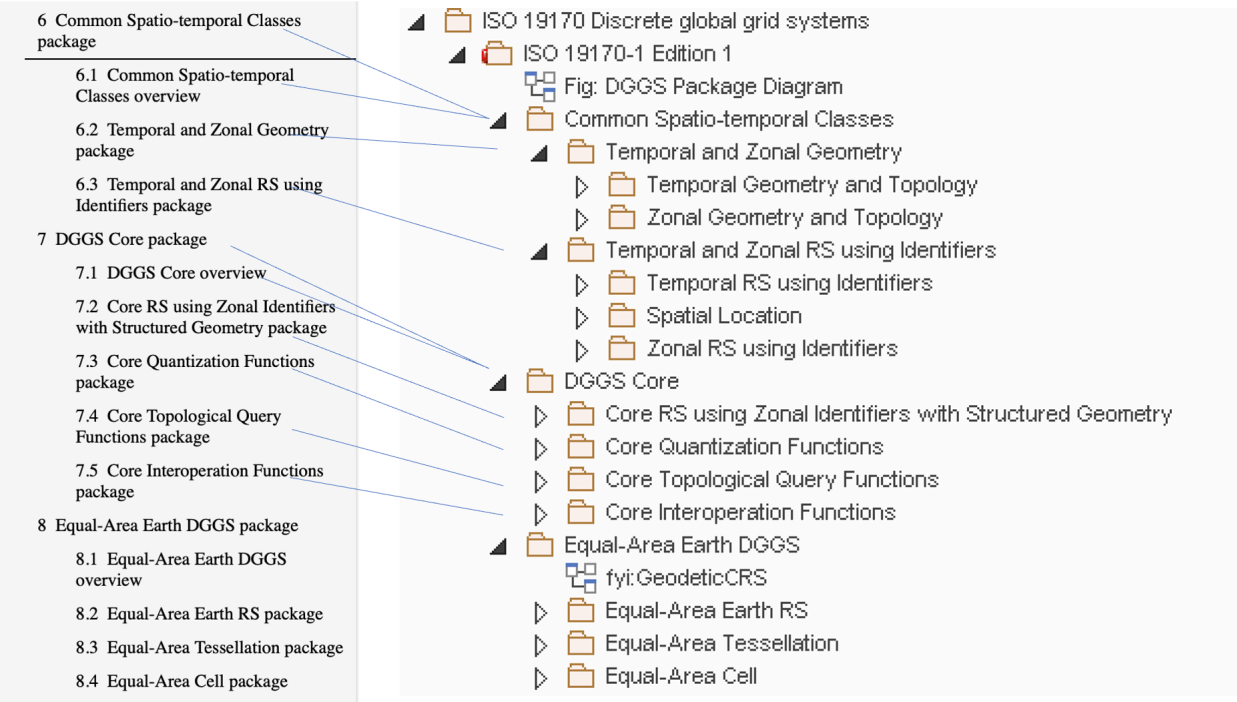


Figure 50 — Rendering style `default` used in OGC 20-040r3 (ISO 19170)

10.4.3.3. Entity list style

The entity list style provides an overview listing of all UML components within a UML package. It provides a high-level overview of the UML package and is meant to be used together with the data dictionary style.

This style was originally developed from OGC 20-010 and is only recommended for MDS experts to tailor the MDS experience.

An example of the `entity_list` style is shown in Figure 51 and Figure 52.

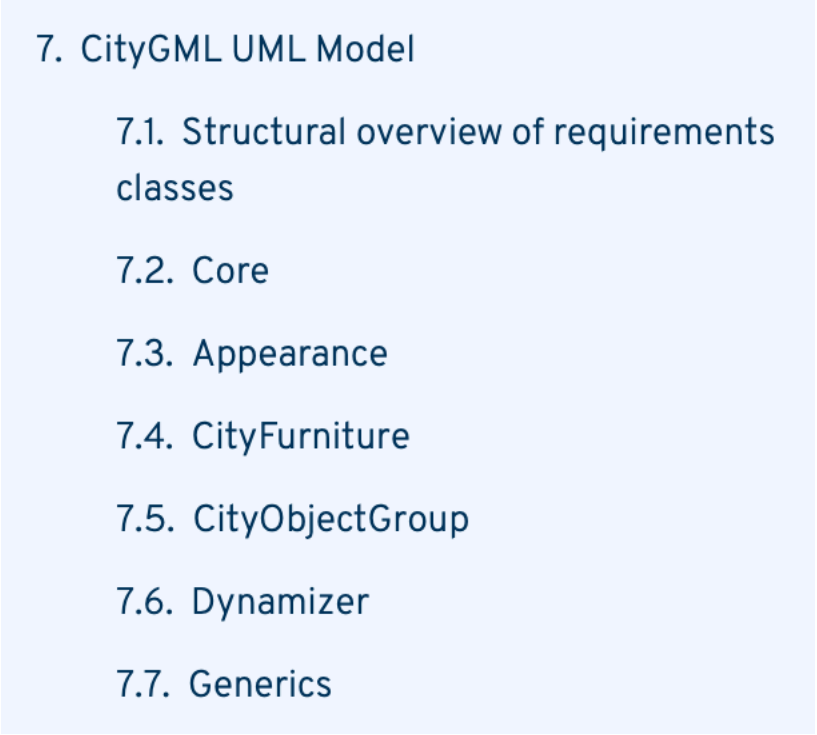


Figure 51 — Rendering style `entity_list` table of contents used in OGC 20-010

Table 5 — Space Classes used in Core

CLASS	DESCRIPTION
AbstractLogicalSpace «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 «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 «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 «Feature Type»	AbstractSpace is the abstract superclass for all types of spaces. A space is an entity of volumetric extent in the real world.
AbstractSpaceBoundary «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 «FeatureType»	AbstractThematicSurface is the abstract superclass for all types of thematic surfaces.

Figure 52 — Rendering style `entity_list` body contents used in OGC 20-010

10.4.3.4. Data dictionary style

The data dictionary style provides a detailed listing of all UML components within a UML package. It provides a detailed-level inspection of the UML package and is meant to be used together with the entity list style.

This style was originally developed from OGC 20-010 and is only recommended for MDS experts to tailor the MDS experience.

An example of the `data_dictionary` style is shown in Figure 53, Figure 54, Figure 55.

Figure 53 — Rendering style `data_dictionary` table of contents used in OGC 20-010

8.2. Core

Table 103 — Metadata of Core (ApplicationSchema)

DESCRIPTION:	The Core module defines the basic components of the CityGML conceptual model. This includes abstract base classes that define the core properties of more specialized thematic classes defined in other modules as well as concrete classes that are common to other modules, for example basic data types.
PARENT PACKAGE:	CityGML
STEREOTYPE:	«ApplicationSchema»

Figure 54 — Rendering style `data_dictionary` body content part 1 used in OGC 20-010

8.2.1. Classes

8.2.1.1. AbstractAppearance

Table 104 — Metadata of AbstractAppearance (FeatureType)

DEFINITION:	AbstractAppearance is the abstract superclass to represent any kind of appearance objects.
SUBCLASS OF:	AbstractFeatureWithLifespan
STEREOTYPE:	«FeatureType»

Table 105 — Attributes of AbstractAppearance (FeatureType)

ATTRIBUTE	VALUE TYPE AND MULTIPLICITY	DEFINITION
adeOfAbstract Appearance	ADEOfAbstract Appearance [0..*]	Augments AbstractAppearance with properties defined in an ADE.

NOTE: Unless otherwise specified, all attributes and role names have the stereotype «Property».

Figure 55 — Rendering style `data_dictionary` body content part 2 used in OGC 20-010

10.4.4. Section depth

The `section_depth` value specifies the clause depth intended for the automated rendering to occur in Metanorma.

Example: The `section_depth` value of 2 specifies that the location of the `lutaml_uml_datamodel_description` command is at the second level of depth, used to maintain the hierarchy of generated AsciiDoc sections.

10.4.5. Package root depth

The `package_root_depth` value indicates the depth of the automated inclusion process from the root UML package block (an OMG XMI file starts with a UML package as root).

Example: The `package_root_level` value of 2 specifies that the automatic inclusion iterative process starts with UML packages at depth 2 of the XMI.

10.5. Customization options

10.5.1. General

The `[lutaml_uml_datamodel_description]` block allows specification of multiple overriding hooks for users to insert content within the automated rendering process.

10.5.2. Diagrams

The `[.diagram_include_block]` block inside `[lutaml_uml_datamodel_description]` is used to import images generated from EA into the automated rendering process.

The process described Clause 10.2 allows extraction of UML diagrams directly from EA. These UML diagrams however exported into image files named according to an EA-proprietary unique ID, such as `EAID_40625194_4483_46b2_80CF_2756F08865D8.svg`, which are difficult to work with.

The `[.diagram_include_block]` block allows `[lutaml_uml_datamodel_description]` to find the correct figure files through this syntax:

```
[.diagram_include_block, base_path="working-directory/Images", format="svg"]
<1>
....
Text <2>
```

```
....
<1> base_path specifies the path of the EA-generated images, format
specifies the file extension of the EA-generated images.
<2> Metanorma AsciiDoc text prior to appearance of the image.
```

Figure 56 — Including diagrams in the `lutaml_uml_datamodel_description` block

The `base_path` parameter is a mandatory value that specifies the path of the EA-generated images. The path here is relative to the source file location where the `[lutaml_uml_datamodel_description]` block is defined.

For example, this is how a typical OGC MDS directory looks like:

Example 1 — Example of OGC MDS document directory with SVG images

```
+- sources/
  +- images/
  +- document.adoc
  +- model/
    +- export.xmi
    +- UML_EA.dtd
    +- Images/
      +- EAID_40625194_4483_46b2_80CF_2756F08865D8.svg
      +- EAID_76FDCDFB_19E5_47b6_9D21_E6450814059F.svg
```

Using this OGC MDS directory, the following block specification will include the EA-generated images.

Example 2 — Example to include EA-generated SVG images in the `lutaml_uml_datamodel_description` block

```
[.diagram_include_block, base_path="model/Images", format="svg"]
....
....
```

If the EA-generated SVG images are generated with undesired artefacts, the `png` format option can be used. Simply re-generate the images using the “PNG” output format in EA in the same directory.

Example 3 — Example of OGC MDS document directory with PNG images

```
+- sources/
  +- images/
  +- document.adoc
  +- model/
    +- export.xmi
    +- UML_EA.dtd
    +- Images/
      +- EAID_40625194_4483_46b2_80CF_2756F08865D8.png
      +- EAID_76FDCDFB_19E5_47b6_9D21_E6450814059F.png
```

Using this OGC MDS directory, the following block specification will include the EA-generated images.

Example 4 — Example to include EA-generated PNG images in the `lutaml_uml_datamodel_description` block

```
[.diagram_include_block, base_path="model/Images", format="png"]
....
....
```

10.5.3. Before and after blocks

The `[.before]` and `[.after]` blocks in the `[lutaml_uml_datamodel_description]` block allows specifying text before or after every described UML class.

When used by itself, it means that this block applies before or after all packages have been iterated through (`[.before]`, `[.after]`).

A `package` parameter can be given to this block to specify that the block only applies to before or after a particular package in the loop (`[.before, package="Another"]`, `[.after, package="CityGML"]`).

Example — Example of using before and after blocks

```
[.before]
....
Text applies before first package is inspected.
....

[.before, package="CityGML"]
....
Text applies immediately before the CityGML package is inspected.
....

[.after, package="CityGML"]
....
Text applies immediately after the CityGML package is inspected.
....

[.after]
....
Text applies after all packages have been inspected.
....
```

10.5.4. Include block

The include block allows dynamic insertion of an external file according to the UML package name being inspected.

The syntax is:

```
[.include_block, position="before", base_path="requirements/requirements_class_
"]
--
--
```

Figure 57

Where,

`base_path` specifies where to find the dynamic file being inserted;

`position=` (optional) specifies either `before` or `after`;

`package=` (optional) specifies the UML package match condition.

NOTE Only `before` and `after` are currently defined as values for `position`.

To use the include block it is necessary to know how the package name is translated into a file name, which file is to be included.

The package name to file name conversion takes these steps:

1. The package name is lowercased;
2. The symbols –, `:` and `` ` `` (whitespace) is converted into _;
3. The resulting name is prefixed with an underscore (_), appended with the `.adoc` or `.liquid` extension, and the specified `base_path=`.

For example, the UML package name of "MUDDI Core: packages" will be transformed into `{base_path}_muddi_core__packages.[adoc|liquid]`.

The `include_block` is useful for including per UML package content, such as ModSpec requirements, conformance tests and structured content.

```
This is additional text that will be included after the inclusion
of the `spec/fixtures/{include_package_name}` file for every UML
package evaluated.
```

Figure 58-1

```
This is additional text that will be included after the inclusion of
the `spec/fixtures/{include_package_name}` file before the `Another`
package.
```

Figure 58-2

10.5.5. Package block

The `[.package_text]` block in the `[lutaml_uml_datamodel_description]` block allows specifying a block to insert at a particular clause index.

The `[.package_text]` block can take the following forms.

To specify text to be interpolated in predefined positions within each package, use the `position=` and `package=` parameters (`[.package_text, position="after", package= "Another"]`).

The syntax is:

```
[.package_text, index="1", position="before", package="Another"]
--
--
```

Figure 59

Where,

`package=` specifies the UML package match condition;

`position=` (optional) specifies either `before` or `after`;

`index=` (optional) if there are multiple `package_text` blocks, define the order of the inserted blocks.

NOTE Only `before` and `after` are currently defined as values for `position`.

The `package_block` is useful for injecting particular texts or files to the automatically generated content.

Example

```
[.package_text, index="1", position="before", package="Common Spatio-temporal
Classes"]
....
Unresolved directive in document.adoc - include::/Users/mulgogi/src/ogc/ogc-
muddi-mds/sources/mds-guide/clause_7_1_common.adoc[]
....

[.package_text, index="2", position="before", package="Temporal and Zonal
Geometry"]
....
Unresolved directive in document.adoc - include::/Users/mulgogi/src/ogc/ogc-
muddi-mds/sources/mds-guide/clause_7_2_temporal.adoc[]
....

[.package_text, index="1", position="after", package="Temporal and Zonal
Geometry"]
....
=== Defining tables

Unresolved directive in document.adoc - include::/Users/mulgogi/src/ogc/ogc-
muddi-mds/sources/mds-guide/../tables/TAB_cc-st-g-t-i.adoc[]

Unresolved directive in document.adoc - include::/Users/mulgogi/src/ogc/ogc-
muddi-mds/sources/mds-guide/../tables/TAB_cc-st-g-t.adoc[]

The following requirement applies:

Unresolved directive in document.adoc - include::/Users/mulgogi/src/ogc/ogc-
muddi-mds/sources/mds-guide/../requirements/REQ_cc-temporal-geometry.adoc[]
....
```

10.6. Manual rendering (advanced)

For the advanced user who wishes to access data elements beyond the automated process, LutaML provides the `[lutaml]` command that can be used individually to build up an MDS.

NOTE Using the `[lutaml]` command for MDS will be highly repetitive and require in-depth understand of Liquid templating.

Figure 60 shows an instance of the `[lutaml]` command in Metanorma, which instructs LutaML to process the file in `path/to/file.xmi`, and pass the results of the parse into the object `package`.

The body of the command then iterates through the contents of `package`, and generates Metanorma AsciiDoc using values from the variable.

```
[lutaml,path/to/filelocation.xmi,package]
--
{% for diagram in package.diagrams %}
[[figure-{{ diagram.xmi_id }}]]
.{{ diagram.name }}
image::{{ base_path }}/{{ diagram.xmi_id }}.{{ format | default: 'png' }}[]

{% if diagram.definition %}
{{ diagram.definition | html2adoc }}
{% endif %}
{% endfor %}
--
```

Figure 60 — Rendering of a UML package under LutaML

- The directives in `{% ... %}` are Liquid processing directives, including loops and conditionals.
- The variables referenced in the directives, and invoked through `{{ ... }}`, are attributes parsed by LutaML from the given source files. For example, `package.diagrams` is the list of all diagrams under the current package, and `diagram` is a loop variable containing the parsed information for one such diagram.
- The variable `diagram` contains attributes of its own which LutaML has parsed; the XMI ID attribute for the diagram,
  - `{{ diagram.xmi_id }}` is used in conjunction with the LutaML parameter `{{ image_base_path }}` in order to define the file location of the associated image file.
  - `{{ diagram.xmi_id }}` is also used with the prefix `figure-` to define the anchor for the image (`[[...]]`), to be used in cross-references.
  - The markup `.{{ diagram.name }}` is used to insert the name attribute of the diagram as the image caption.

11

UML MODEL ELEMENTS USED IN THE MDS PROCESS

11 UML MODEL ELEMENTS USED IN THE MDS PROCESS

11.1. Package

11.1.1. Name

The package should have a unique name.

11.1.2. Description

The package description should be filled in.

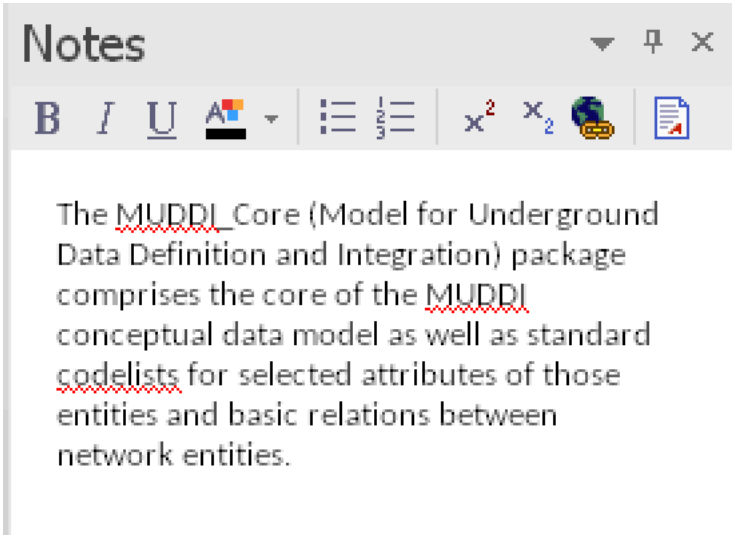


Figure 61

11.2. Class

11.2.1. Name

The class should have a unique name within the package it belongs to.

11.2.2. Description

The class description should be filled in in the Notes pane.

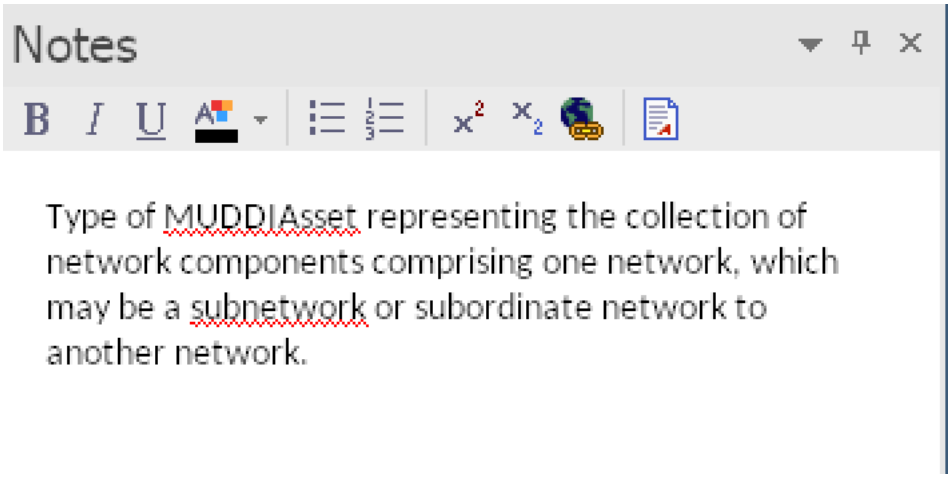


Figure 62

11.2.3. Relationships

A class can be set with multiple relationships.

These relationships are described in the MDS process.

Generalization	the target class will be described as a “superclass”, and the source class will be listed as a “subclass” of the target class.
Dependency	the source class will be listed as a “dependency” of the target.
Realization	EA creates Realization relationships from every UML class to the class itself, and these are not rendered in the MDA process.

11.3. Property

11.3.1. Name

The property should have a unique name within the class it belongs to.

11.3.2. Description

The property description is entered in the Notes pane.

11.3.3. Unspecified value type

If there is no value type specified for an property, create an “AbstractValueType” data type with the GML Stereotype “Type”, and assign it to the property (see Figure 63).

Table 14 – Attributes of MUDDIAsset (FeatureType)

Attribute	Value type and multiplicity	Definition
assetOwnerID	AbstractValueType [1..1]	

Figure 63 – Assignment of AbstractValueType to represent an unspecified value type (from: MUDDI Conceptual Model)

11.4. Data type

11.4.1. Name

The data type should have a unique name within the package it belongs to.

11.4.2. Description

The data type description is entered in the Notes pane.

11.5. Enumeration

11.5.1. Name

The enumeration should have a unique name within the package it belongs to.

11.5.2. Description

The enumeration description is entered in the Notes pane.

11.6. Enumeration values

11.6.1. Name

The enumeration value should have a unique name within the enumeration it belongs to.

11.6.2. Description

The enumeration value description is entered in the Notes pane.

11.7. Figure

Figures are automatically included in the package description.

A

ANNEX A (INFORMATIVE) CHECKLISTS TO COMPLETE

A ANNEX A (INFORMATIVE) CHECKLISTS TO COMPLETE

Table A.1

DESCRIPTION	DONE?
Have you filled in the Notes section for all classes?	...
Have you filled in the Notes section for all classes?	...

B

ANNEX B (INFORMATIVE) EXAMPLE OGC MDS DOCUMENT

ANNEX B
(INFORMATIVE)
EXAMPLE OGC MDS DOCUMENT

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Robert Gibb: OGC 20-040r3, *Topic 21 — Discrete Global Grid Systems — Part 1 Core Reference system and Operations and Equal Area E*. Open Geospatial Consortium (2021). https://docs.ogc.org/as/20-040r3/20-040r3.html.

[2] Josh Lieberman: OGC 17-090r1, *Model for Underground Data Definition and Integration (MUDDI) Engineering Report*. Open Geospatial Consortium (2019). https://docs.ogc.org/per/17-090r1.html.

[3] Policy SWG: OGC 08-131r3, *The Specification Model — Standard for Modular specifications*. Open Geospatial Consortium (2009). https://portal.ogc.org/files/?artifactid=34762&version=2.

[4] Clemens Portele, Panagiotis (Peter) A. Vretanos, Charles Heazel: OGC 17-069r3, *OGC API — Features — Part 1: Core*. Open Geospatial Consortium (2019). https://docs.ogc.org/is/17-069r3/17-069r3.html.

[5] Thomas H. Kolbe, Tatjana Kutzner, Carl Stephen Smyth, Claus Nagel, Carsten Roensdorf, Charles Heazel: OGC 20-010, *OGC City Geography Markup Language (CityGML) Part 1: Conceptual Model Standard*. Open Geospatial Consortium (2021). https://docs.ogc.org/is/20-010/20-010.html.

[6] Josh Lieberman, Andy Ryan: OGC 17-048, *OGC Underground Infrastructure Concept Study Engineering Report*. Open Geospatial Consortium (2017). https://docs.ogc.org/per/17-048.html.

[7] ISO: ISO 690, *Information and documentation — Guidelines for bibliographic references and citations to information resources*. International Organization for Standardization, Geneva https://www.iso.org/standard/72642.html.

[8] ISO: ISO 704, *Terminology work — Principles and methods*. International Organization for Standardization, Geneva https://www.iso.org/standard/79077.html.

[9] ISO: ISO 8601-1, *Date and time — Representations for information interchange — Part 1: Basic rules*. International Organization for Standardization, Geneva https://www.iso.org/standard/70907.html.

[10] ISO: ISO 10241-1, *Terminological entries in standards — Part 1: General requirements and examples of presentation*. International Organization for Standardization, Geneva https://www.iso.org/standard/40362.html.

[11] ISO: ISO 10303-11, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual*. International Organization for Standardization, Geneva https://www.iso.org/standard/38047.html.

[12] ISO: ISO 19101-1, *Geographic information — Reference model — Part 1: Fundamentals*. International Organization for Standardization, Geneva https://www.iso.org/standard/59164.html.

[13] ISO: ISO 19103:2015, *Geographic information — Conceptual schema language*. International Organization for Standardization, Geneva (2015). https://www.iso.org/standard/56734.html.

[14] ISO: ISO/TS 19103:2005, *Geographic information — Conceptual schema language*. International Organization for Standardization, Geneva (2005). https://www.iso.org/standard/37800.html.

[15] ISO: ISO 19105, *Geographic information — Conformance and testing*. International Organization for Standardization, Geneva https://www.iso.org/standard/76457.html.

[16] ISO: ISO 19109:2015, *Geographic information — Rules for application schema*. International Organization for Standardization, Geneva (2015). https://www.iso.org/standard/59193.html.

[17] ISO: ISO 19118:2011, *Geographic information — Encoding*. International Organization for Standardization, Geneva (2011). https://www.iso.org/standard/44212.html.

[18] ISO/IEC: ISO/IEC 19501, *Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2*. International Organization for Standardization, International Electrotechnical Commission, Geneva https://www.iso.org/standard/32620.html.

[19] ISO: ISO/AWI 36100, *Standardization documents — Document metamodel*. International Organization for Standardization, Geneva https://www.iso.org/standard/77056.html.

[20] ISO/PWI: ISO/PWI 36200, *Standardization documents — Metadata. International Organization for Standardization, Geneva*. ISO, PWI

[21] ISO/PWI: ISO/PWI 36300, *Standardization documents — Representation in XML. International Organization for Standardization, Geneva*. ISO, PWI

[22] OMG UML 2.5, *Unified Modeling Language*. (2015). https://www.omg.org/spec/UML/2.5/About-UML.

[23] OMG XMI 2.5, *XML Metadata Interchange*. Object Management Group (2015). https://www.omg.org/spec/XMI/2.5.1/About-XMI/

[24] W3C xmlschema-2, *XML Schema Part 2: Datatypes Second Edition*. https://www.w3.org/TR/xmlschema-2/.

[25] Ribose Inc. *Metanorma*. https://www.metanorma.org

[26] Ribose Inc. *Metanorma for OGC*. https://www.metanorma.org/author/ogc/

[27] Sparx Systems, *Enterprise Architect*. https://sparxsystems.com/products/ea/