NIEM Naming and Design Rules (NDR) Version 6.0

Project Specification Draft 01

27 January 2025

This stage:

https://docs.oasis-open.org/niemopen/ndr/v6.0/psd01/ndr-v6.0-psd01.html

https://docs.oasis-open.org/niemopen/ndr/v6.0/psd01/ndr-v6.0-psd01.pdf (Authoritative)

Previous stage:

https://docs.oasis-open.org/niemopen/ndr/v6.0/psd01/ndr-v6.0-psd01.html

https://docs.oasis-open.org/niemopen/ndr/v6.0/psd01/ndr-v6.0-psd01.pdf (Authoritative)

Latest stage:

https://docs.oasis-open.org/niemopen/ndr/v6.0/ndr-v6.0.html

https://docs.oasis-open.org/niemopen/ndr/v6.0/ndr-v6.0.pdf (Authoritative)

Open Project:

OASIS NIEMOpen OP

Project Chair:

Katherine Escobar (katherine.b.escobar.civ@mail.mil), Joint Staff J6

NTAC Technical Steering Committee Chairs:

Brad Bolliger (brad.bolliger@ey.com), EY

James Cabral (jim@cabral.org), Individual

Scott Renner (sar@mitre.org), MITRE

Editors:

James Cabral (jim@cabral.org), Individual

Tom Carlson (Thomas.Carlson@gtri.gatech.edu), Georgia Tech Research Institute

Scott Renner (sar@mitre.org), MITRE

Related work:

This specification replaces or supersedes:

- *National Information Exchange Model Naming and Design Rules*. Version 5.0 December 18, 2020. NIEM Technical

Architecture Committee (NTAC). https://reference.niem.gov/niem/specification/naming-and-design-rules/5.0/niem-ndr-5.0.html.

This specification is related to:

- *NIEM Model Version 6.0*. Edited by Christina Medlin. Latest stage: https://docs.oasis-open.org/niemopen/niem-model/v6.0/niem-model-v6.0.html.
- Conformance Targets Attribute Specification (CTAS) Version 3.0. Edited by Tom Carlson. 22 February 2023. OASIS Project Specification 01. https://docs.oasis-open.org/niemopen/ctas/v3.0/ps01/ctas-v3.0-ps01.html. Latest stage: https://docs.oasis-open.org/niemopen/ctas/v3.0/ctas-v3.0.html.

Abstract:

Work in progress.

Status:

This document was last revised or approved by the Project Governing Board of the OASIS NIEMOpen OP on the above date. The level of approval is also listed above. Check the “Latest stage” location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Open Project (OP) are listed at http://www.niemopen.org/.

Comments on this work can be provided by opening issues in the project repository or by sending email to the project’s public comment list: niemopen@lists.oasis-open-projects.org. List information is available at https://lists.oasis-open-projects.org/g/niemopen.

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product’s prose narrative document(s), the content in the separate plain text file prevails.

Key words:

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in BCP 14 [RFC 2119] and [RFC 8174] when, and only when, they appear in all capitals, as shown here.

Citation format:

When referencing this specification the following citation format should be used:

[NIEM-NDR-v6.0]

NIEM Naming and Design Rules (NDR) Version 6.0. Edited by Scott Renner. 1 January 2025. OASIS Project Specification Draft 01. https://docs.oasis-open.org/niemopen/ndr/v6.0/psd01/ndr-v6.0-psd01.html. Latest stage: https://docs.oasis-open.org/niemopen/ndr/v6.0/ndr-v6.0.html.

Notices

Table of Contents

1. Introduction

NIEM, formerly known as the “National Information Exchange Model,” is a framework for exchanging information among public and private sector organizations. The framework includes a reference data model for objects, properties, and relationships; and a set of technical specifications for using and extending the data model in information exchanges. The NIEM framework supports developer-level specifications of data that form a contract between developers. The data being specified is called a *message* in NIEM. While a message is usually something passed between applications, NIEM works equally well to specify an information resource published on the web, an input or output for a web service or remote procedure, and so forth, basically, any package of data that crosses a system or organization boundary.

NIEM promotes scalability and reusability of messages between information systems, allowing organizations to share data and information more efficiently. It was launched in 2005 in response to the U.S. Homeland Security Presidential Directives to improve information sharing between agencies following 9/11. Until 2023, NIEM was updated and maintained in a collaboration between the U.S. federal government, state and local government agencies, private sector, and non-profit and international organizations, with new versions released around once per year. NIEM defines a set of common objects, the *NIEM Core*, and 17 sets of objects that are specific to certain government or industry verticals, the *NIEM Domains*.

In 2023, NIEM became the NIEMOpen OASIS Open Project. NIEMOpen welcomes participation by anyone irrespective of affiliation with OASIS. Substantive contributions to NIEMOpen and feedback are invited from all parties, following the OASIS rules and the usual conventions for participation in GitHub public repository projects.

NIEMOpen is the term generally used when referring to the organization such as Project Governing Board (PGB), NIEMOpen Technical Architecture Committee (NTAC), NIEMOpen Business Architecture Committee (NBAC), organization activities or processes. NIEM is the term used when directly referring to the model i.e. NIEM Domain, NIEM Model version.

This document specifies principles and enforceable rules for NIEM data components and schemas. Schemas and components that obey the rules set forth here are conformant to specific conformance targets. Conformance targets may include more than the level of conformance defined by this NDR, and may include specific patterns of use, additional quality criteria, and requirements to reuse NIEM release schemas.

1.1 Glossary

1.1.1 Definitions of terms

Term	Definition
Absolute URI	A Uniform Resource Identifier (URI) with scheme, hierarchical part, and optional query, but without a fragment; a URI matching the grammar syntax `<absoluteURI>` as defined by [RFC 3986].
Adapter class	A class that contains only properties from a single external namespace. [*see §4.4*]
Adapter type	An XSD type definition that encapsulates external components for use within NIEM. *(see §9.4)*
Appinfo namespace	A namespace defined by a schema document that provides additional semantics for components in the XSD representation of a model. *(see §9.7)*
Association class	A class that represents a specific relationship between objects. *(see §4.4)*
Attribute property	A data property represented in XSD as an attribute declaration. *(see §4.8)*

Term	Definition
Augmentation	The means by which a designer of one namespace adds properties to a class defined in a different namespace. *(see §3.7, §4.15)*
Augmentation element	An element in an XML message that is a container for one or more augmentation properties. *(see §4.15.2)*
Augmentation point element	An abstract element declaration that provides a place for augmentation properties within the XSD representation of an augmented class. *(see §4.15.2)*
Augmentation property	A property added by one namespace to an augmented class in another namespace. *(see §4.15)*
Augmentation type	An XSD type definition for an augmentation element. *(see §4.15.2)*
Cardinality	The number of times a property may/must appear in an object.
Class	A definition of an entity in a model; that is, a real-world object, concept, or thing *(see §3.4, §4.4)*
Code list datatype	A datatype in which each valid value is also a string in a code list. *(see §4.12)*
Code list	A set of string values, each having a known meaning beyond its value, each representing a distinct conceptual entity. *(see §4.12)*
Conforming namespace	A namespace that satisfies all of the applicable rules in this document; a reference namespace, extension namespace, or subset namespace. *(see §6.1)*
Conforming schema document	A schema document that satisfies all of the applicable rules in this document. *(see §6.1)*
Conforming schema document set	A schema document set that satisfies all of the applicable rules in this document. *(see §6.1)*
Data definition	A text definition of a component, describing what the component means.
Data property	Defines a relationship between an object and a literal value.
Datatype	Defines the allowed values of a corresponding literal value in a message.
Documented component	A CMF object or XSD schema component that has an associated data definition.
Element property	An object property, or a data property that is not an attribute property; represented in XSD by an element declaration. *(see §4.8)*
Extension namespace	A namespace defining components that are intended for reuse, but within a more narrow scope than those defined in a reference namespace. *(see §3.6)*
Extension schema document	A schema document that is the XSD representation of an extension namespace.

Term	Definition
External attribute	An attribute declaration in external schema document.
External component	A component defined by an external schema document. *(see §9.4)*
External namespace	Any namespace defined by a schema document that is not a conforming namespace, the structures namespace, or the XML namespace http://www.w3.org/XML/1998/namespace. *(see §3.6)*
External schema document	A schema document that defines an external namespace. *(see §3.6)*
Literal class	A class that contains no object properties, one or more attribute properties, and exactly one element property. *(see §4.4)*
Literal property	The element property in a literal class.
Local term	A word, phrase, acronym, or other string of characters that is used in the name of a namespace component, but that is not defined in OED, or that has a non-OED definition in this namespace, or has a word sense that is in some way unclear. *(see §4.16)*
Message	A package of data shared at runtime; a sequence of bits that convey information to be exchanged or shared; an instance of a message type. *(see §3.1.1)*
Message designer	A person who creates a message type and message format from an information requirement, so that an instance message at runtime will contain all the facts that need to be conveyed.
Message developer	A person who writes software to implement a message specification, producing or processing messages that conform to the message format.
Message format	A specification of the valid syntax of messages that conform to a message type. *(see §3.1.2)*
Message model	A data model intended to precisely define the mandatory and optional content of messages and the meaning of that content. *(see §3.1.3)*
Message object	The initial object in a message.
Message specification	A collection of related message formats and message types. *(see §3.1.4)*
Message type	A specification of the information content of messages. *(see §3.1.3)*
Model file	The CMF representation of a NIEM model; a message that conforms to the CMF message type. *(see §3.5, §6.1)*
Namespace	A collection of uniquely-named components, managed by an authoritative source. *(see §3.6)*
NCName	A non-colonized name, matching the grammar syntax `<NCName>` as defined by [XML Namespaces].
Object class	Represents a class of objects defined by a NIEM model. *(see §4.4)*

Term	Definition
Proxy type	An XSD complex type definition with simple content that extends one of the simple types in the XML Schema namespace with `structures:SimpleObjectAttributeGroup`. *(see §9.5)*
Relationship property	A property that provides information about the relationship between its parent and grandparent objects. *(see §4.6, §5.5)*
Reference attribute property	An attribute property that contains a reference to an object in a message.*(see §4.8)*
Reference namespace	A namespace containing components that are intended for the widest possible reuse.*(see §3.6)*
Reference schema document	The XSD representation of a reference namespace. *(see §9.8)*
Reuse model	A data model entirely comprised of reference namespaces and extension namespaces; a model intended to make the agreed definitions of a community available for reuse.
Schema	An artifact that can be used to assess the validity of a message; in XML Schema for XML messages, JSON Schema for JSON messages. *(see §3.1.2)*
Schema document set	A collection of schema documents that together are capable of validating an XML document.*(see §10.2)*
Serialization	*(Verb)* A process of converting a data structure into a sequence of bits that can be stored or transferred. *(Noun)* A standard for the output of serialization; for example, XML and JSON.
Structures namespace	A namespace that provides base types and attributes for the XSD representation of NIEM models.*(see §3.6)*
Subset namespace	A subset of the components in a reference or extension namespace.*(see §3.6)*
Subset rule	Any data that is valid for a subset namespace must also be valid for its reference namespace or extension namespace, and must have the same meaning.*(see §8.4)*
Subset schema document	A schema document for a subset namespace. *(see §9.10)*

Terms imported from *Extensible Markup Language (XML) 1.0 (Fourth Edition)* [XML]:

Term	Definition
Document element	An element, no part of which appears in the content of another element; preferred synonym for *root element*.
XML document	A data object is an XML document if it is well-formed, as defined in this specification. (Section 2, Documents)

Terms imported from *XML Information Set (Second Edition)* [XML Infoset]:

Term	Definition

Term	Definition
Attribute	An *attribute information item*, as defined by Section 2.3: *Attribute Information Items*.
Element	An *element information item*, as defined by Section 2.2, *Element Information Items*.

Terms imported from [XML Schema Structures]:

Term	Definition
Attribute declaration	As defined by *Section 2.2.2.3, Attribute Declaration*.
Base type definition	A type definition used as the basis for an extension or restriction.*(see Section 2.2.1.1, Type Definition Hierarchy)*
Complex type definition	As defined by *Section 2.2.1.3, Complex Type Definition**.
Element declaration	As defined by *Section 2.2.2.1, Element Declaration*.
Schema component	The generic term for the building blocks that comprise the abstract data model of the schema*(see Section 2.2, XML Schema Abstract Data Model)*
Schema document	As defined by *Section 3.1.2, XML Representations of Components*, which states, “A document in this form (i.e. a element information item) is a schema document.”
Simple type definition	As defined by *Section 2.2.1.2, Simple Type Definition*.
Valid	As defined by *Section 2.1, Overview of XML Schema*, which states, “The word valid and its derivatives are used to refer to clause 1 above, the determination of local schema-validity.”
XML Schema	A set of schema components. *(see Section 2.2, XML Schema Abstract Data Model)*
XML Schema definition language (XSD)	As defined by *Abstract*, which states, “XML Schema: Structures specifies the XML Schema definition language, which offers facilities for describing the structure and constraining the contents of XML 1.0 documents, including those which exploit the XML Namespace facility.”

Terms imported from NIEM Conformance Targets Attribute Specification [CTAS-v3.0]:

Term	Definition
Conformance target	A class of artifact, such as an interface, protocol, document, platform, process or service, that is the subject of conformance clauses and normative statements. *(see §6.1)*
Conformance target identifier	An internationalized resource identifier (IRI) that uniquely identifies a conformance target.

Term	Definition
Effective conformance targets attribute	The first occurrence of the attribute {https://docs.oasis-open.org/niemopen/ns/specification/conformanceTargets/6.0/}conformanceTargets, in document order.
Effective conformance target identifier	An internationalized resource identifier reference that occurs in the document's effective conformance targets attribute.

1.1.2 Acronyms and abbreviations

Term	Literal
APPINFO	Application Information
CCC	Complex type with Complex Content
CMF	Common Model Format
CSC	Complex type with Simple Content
CSV	Comma Separated Values
CTAS	Conformance Targets Attribute Specification
ID	Identifier
IEP	Information Exchange Package
IEPD	Information Exchange Package Documentation
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation Linked Data
NBAC	NIEMOpen Business Architecture Committee
NS	Namespace
NTAC	NIEMOpen Technical Architecture Committee
OED	Oxford English Dictionary
OP	Open Project
OWL	Web Ontology Language
PGB	Project Governing Board
QName	Qualified Name
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RFC	Request For Comments

Term	Literal
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
XML	Extensible Markup Language
XSD	XML Schema Definition

2. How To Read This Document

This document provides normative specifications for NIEM-conforming data models. It also describes the goals and principles behind those specifications. It includes examples and explanations to help users of NIEM understand the goals, principles, and specifications.

This document is not intended as a user guide. Training materials for message designers and developers will be available at www.niemopen.org.

The relevant sections of this document will depend on the role of the user.Figure 2-1 illustrates the relationships between these roles and NIEM activities.

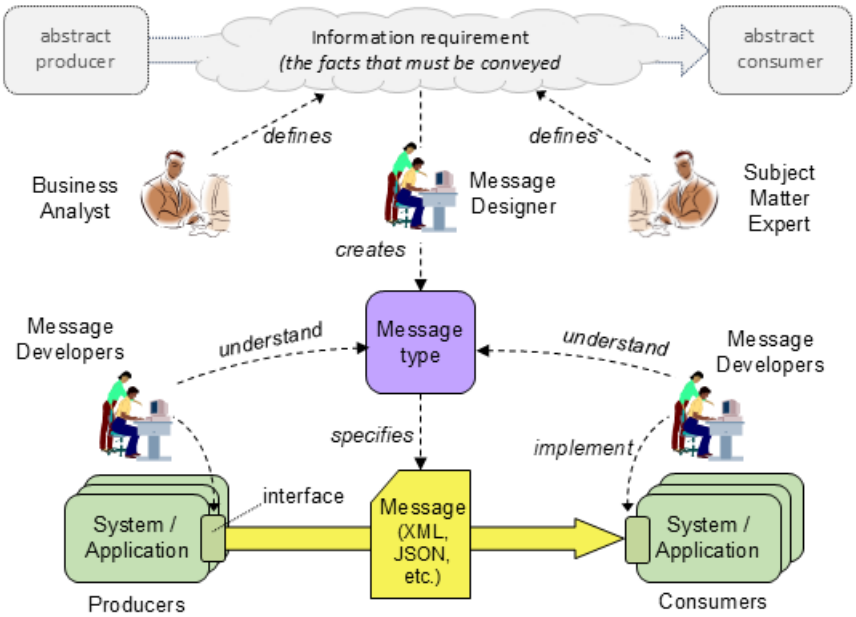


Figure 2-1: User roles and activities

The user roles in the above figure are:

- *Business analysts* and *subject matter experts*, who provide the requirements for information transfer. These requirements might describe an information resource available to all comers. They could describe an information exchange as part of a business process. They need not be tied to known producers and consumers.
- *Message designers*, who express those requirements as a[message type](), which specifies the syntax and semantics of the data that will convey the required information at runtime.

- *Message developers*, who write software to construct messages that contain the required information and follows the defined syntax, and who write software to parse and process such messages.

The remaining sections of this document most relevant to each of these roles are shown in the following table:

Section	Manager	Business Analyst	Message Designer	Message Developer
3. Overview of NIEM technical architecture	x	x	x	x
4. Data models in NIEM			x	
5. Data modeling patterns			x	
6. Conformance		x	x	x
7. Rules for model components		x	x	
8. Rules for namespaces		x	x	
9. Rules for schema documents			x	
10. Rules for models			x	
11. Rules for message types and message formats			x	x
12. Rules for XML messages			x	x
13. Rules for JSON messages			x	x
14. RDF interpretation of NIEM models and messages			x	

Table 2-2: Relevant document sections by user role

2.1 Document references

This document relies on references to many outside documents. Such references are noted by bold, bracketed inline terms. For example, a reference to RFC 3986 is shown as [RFC 3986]. All reference documents are recorded inAppendix A, References, below.

2.2 Clark notation and qualified names

This document uses both Clark notation and QName notation to represent qualified names.

QName notation is defined by XML Namespaces Section 4, Qualified Names. A QName for the XML Schema string datatype is xs:string. Namespace prefixes used within this specification are listed in Section 2.3, Use of namespaces and namespace prefixes, below.

This document sometimes uses Clark notation to represent qualified names in normative text. Clark notation is described by ClarkNS, and provides the information in a QName without the need to first define a namespace prefix, and then to reference that namespace prefix. A Clark notation representation for the qualified name for the XML Schema string datatype is {http://www.w3.org/2001/XMLSchema}string.

Each Clark notation value usually consists of a namespace URI surrounded by curly braces, concatenated with a local name. The exception to this is when Clark notation is used to represent the qualified name for an attribute with no namespace, which is ambiguous when represented using QName notation. For example, the element targetNamespace, which has no [namespace name] property, is represented in Clark notation as `{}targetNamespace`.

2.3 Use of namespaces and namespace prefixes

The following namespace prefixes are used consistently within this specification. These prefixes are not normative; this

document issues no requirement that these prefixes be used in any conformant artifact. Although there is no requirement for a schema or XML document to use a particular namespace prefix, the meaning of the following namespace prefixes have fixed meaning in this document.

- `xs`: The namespace for the XML Schema definition language as defined by XML Schema Structures and XML Schema Datatypes, http://www.w3.org/2001/XMLSchema.
- `xsi`: The XML Schema instance namespace, defined by XML Schema Structures Section 2.6, Schema-Related Markup in Documents Being Validated, for use in XML documents, http://www.w3.org/2001/XMLSchema-instance.
- `ct`: The namespace defined by CTAS for the conformanceTargets attribute, https://docs.oasis-open.org/niemopen/ns/specification/conformanceTargets/6.0/.
- `appinfo`: The namespace for the appinfo namespace, https://docs.oasis-open.org/niemopen/ns/model/appinfo/6.0/.
- `structures`: The namespace for the structures namespace, https://docs.oasis-open.org/niemopen/ns/model/structures/6.0/.
- `cmf`: The namespace for the CMF model representation, https://docs.oasis-open.org/niemopen/ns/specification/cmf/1.0/.

---

3. Overview of the NIEM Technical Architecture

This overview describes NIEM's design goals and principles, and introduces key features of the architecture. The major design goals are:

- *Shared understanding of data.* NIEM helps developers working on different systems to understand the data their systems share with each other.
- *Reuse of community-agreed data definitions.* NIEM reduces the cost of data interoperability by promoting shared data definitions — without requiring a single data model of everything for everyone.
- *Open standards with free-and-open-source developer tools.* NIEM does not depend on proprietary standards or the use of expensive developer tools.

The key architecture features mentioned in this section:

- *The NIEM metamodel* — an abstract, technology-neutral data model for NIEM data models
- *Two equivalent model representations* — One is a profile of XML Schema (XSD) that has been used in every version of NIEM. The other is itself a NIEM-based data specification, suitable for XML and many other data technologies.
- *Model namespaces* — for model configuration management by multiple authors working independently.

3.1 Machine-to-machine data specifications

NIEM is a framework for developer-level specifications of data. A NIEM-based data specification — which is built*using* NIEM and in *conformance* to NIEM, but is not itself a*part* of NIEM — describes data to the developers of producing and consuming systems. This data may be shared via:

- a message passed between applications
- an information resource published on the web
- an API for a system or service

NIEM is potentially useful for any data sharing mechanism that transfers data across a system or organization boundary. (Within a system, NIEM may be useful when data passes between system components belonging to different developer teams.)

The primary purpose of a NIEM-based data specification is to establish a common understanding among developers, so that they can write software that correctly handles the shared data, hence "machine-to-machine". (NIEM-conforming data may also be directly presented to human consumers, and NIEM can help these consumers understand what they see, but that is not the primary purpose of NIEM.)

Data sharing in NIEM is implemented in terms of messages, message formats, and message types. These are illustrated in

figure 3-1.

- message — a package of data shared at runtime; an instance of a message format and of a message type
- message format — a definition of a syntax for the messages of a message type
- message type — a definition of the information content in equivalent message formats

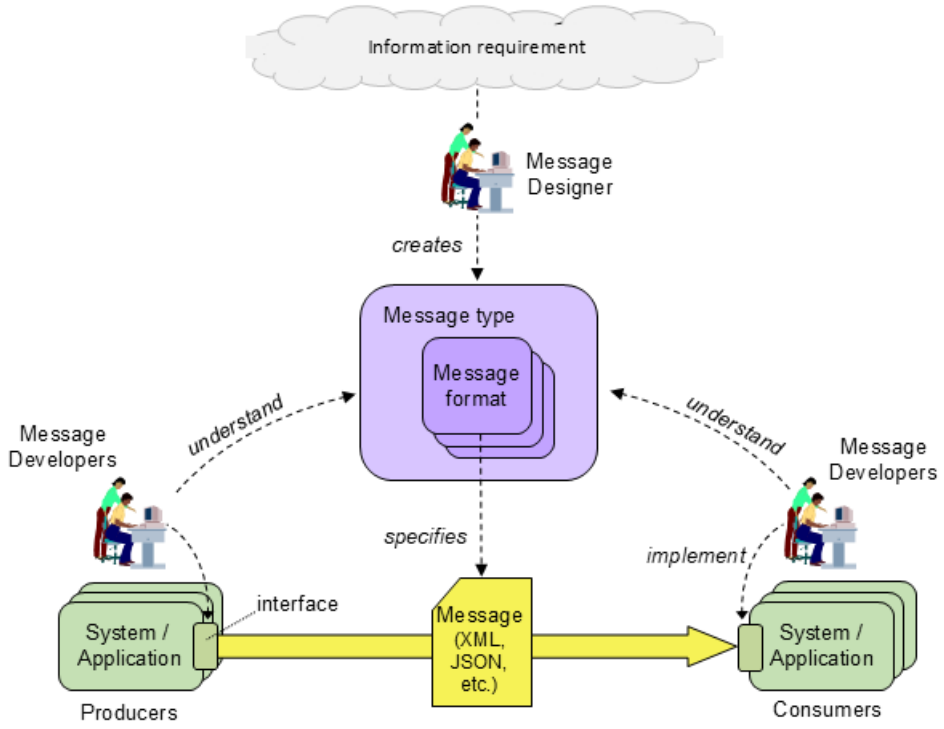


Figure 3-1: Message types, message formats, and messages

A message designer turns information requirements into a message type, then turns a message type into one or more message formats. Message developers then use the message type and message format to understand how to implement software that produces or consumes conforming messages.

3.1.1 Messages

In NIEM terms, the package of data shared at runtime is a message. This data is arranged according to a supported serialization. The result is a sequence of bits that represents the information content of the message. Example 3-2 shows two messages representing the same information, one serialized in XML, the other in JSON. Each message in this example is a request for a quantity of some item. (In all examples, closing tags and brackets may be omitted, long lines may be truncated, and some portions omitted and/or replaced with ellipses (…).)

```
<msg:Request                                                    | {
 xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem | "@context": {
 xmlns:msg="http://example.com/ReqRes/1.0/">                  |   "nc": "https://docs.oasis-open.org/niemopen/ns/model/n
  <msg:RequestID>RQ001</msg:RequestID>                        |   "msg": "http://example.com/ReqRes/1.0/"
  <msg:RequestedItem>                                         | },
    <nc:ItemName>Wrench</nc:ItemName>                         | "msg:Request": {
    <nc:ItemQuantity>10</nc:ItemQuantity>                     |   "msg:RequestID" : "RQ001",
  </msg:RequestedItem>                                        |   "msg:RequestedItem": {
</msg:Request>                                                |     "nc:ItemName": Wrench",
                                                              |     "nc:ItemQuantity": 10
                                                              |   }
                                                              | }
```


Example 1 - Markdown with rows

```
| **XML**                                                  | **JSON**
|----------------------------------------------------------|-----------------------------------------------------------------
| `<msg:Request`                                           | `{`
| ` xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem"` | `  "@context": {`
| ` xmlns:msg="http://example.com/ReqRes/1.0/">`           | `    "nc": "https://docs.oasis-open.org/niemopen/ns/model/ni
| `  <msg:RequestID>RQ001</msg:RequestID>`                 | `    "msg": "http://example.com/ReqRes/1.0/"`
| `  <msg:RequestedItem>`                                  | `  },`
| `    <nc:ItemName>Wrench</nc:ItemName>`                  | `  "msg:Request": {`
| `    <nc:ItemQuantity>10</nc:ItemQuantity>`              | `    "msg:RequestID" : "RQ001",`
| `  </msg:RequestedItem>`                                 | `    "msg:RequestedItem": {`
| `</msg:Request>`                                         | `      "nc:ItemName": "Wrench",`
|                                                          | `      "nc:ItemQuantity": 10`
|                                                          | `    }`
|                                                          | `  }`
|                                                          | `}`
```

Examle 2 - HTML embedded in markdown

```
<table style="border-collapse: collapse; width: 100%;">
  <tr>
    <td style="vertical-align: top; padding-right: 20px; border: none;">
<pre><code>&lt;msg:Request
 xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem"
 xmlns:msg="http://example.com/ReqRes/1.0/"&gt;
  &lt;msg:RequestID&gt;RQ001&lt;/msg:RequestID&gt;
  &lt;msg:RequestedItem&gt;
    &lt;nc:ItemName&gt;Wrench&lt;/nc:ItemName&gt;
    &lt;nc:ItemQuantity&gt;10&lt;/nc:ItemQuantity&gt;
  &lt;/msg:RequestedItem&gt;
&lt;/msg:Request&gt;
</code></pre>
    </td>
    <td style="vertical-align: top; border: none;">
<pre><code>{
  "@context": {
    "nc": "https://docs.oasis-open.org/niemopen/ns/model/niem-",
    "msg": "http://example.com/ReqRes/1.0/"
  },
  "msg:Request": {
    "msg:RequestID": "RQ001",
    "msg:RequestedItem": {
      "nc:ItemName": "Wrench",
      "nc:ItemQuantity": 10
    }
  }
}
</code></pre>
    </td>
  </tr>
</table>
```

XML vs JSON Message Example

```
[XML]                                                                | [JSON]
<msg:Request                                                         | {
 xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem        |   "@context": {
 xmlns:msg="http://example.com/ReqRes/1.0/">                         |     "nc": "https://docs.oasis-open.org/niemopen/ns/model/n
  <msg:RequestID>RQ001</msg:RequestID>                               |     "msg": "http://example.com/ReqRes/1.0/"
  <msg:RequestedItem>                                                |   },
    <nc:ItemName>Wrench</nc:ItemName>                                |   "msg:Request": {
    <nc:ItemQuantity>10</nc:ItemQuantity>                            |     "msg:RequestID" : "RQ001",
  </msg:RequestedItem>                                               |     "msg:RequestedItem": {
</msg:Request>                                                       |       "nc:ItemName": "Wrench",
                                                                     |       "nc:ItemQuantity": 10
                                                                     |     }
                                                                     |   }
                                                                     | }
```

```
| **XML**                                                                  | **JSON**
|--------------------------------------------------------------------------|-----------------------------------------------
| `<msg:Request                                                    `       | `{`
| ` xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem`          | `  "@context": {`
| ` xmlns:msg="http://example.com/ReqRes/1.0/">`                           | `    "nc": "https://docs.oasis-open.org/niemope
| `  <msg:RequestID>RQ001</msg:RequestID>`                                 | `    "msg": "http://example.com/ReqRes/1.0/"`
| `  <msg:RequestedItem>`                                                  | `  },`
| `    <nc:ItemName>Wrench</nc:ItemName>`                                  | `  "msg:Request": {`
| `    <nc:ItemQuantity>10</nc:ItemQuantity>`                              | `    "msg:RequestID" : "RQ001",`
| `  </msg:RequestedItem>`                                                 | `    "msg:RequestedItem": {`
| `</msg:Request>`                                                         | `      "nc:ItemName": "Wrench",`
|                                                                          | `      "nc:ItemQuantity": 10`
|                                                                          | `    }`
|                                                                          | `  }`
|                                                                          | `}`
```

Example 3-2: Example of messages in XML and JSON syntax

The data structure of a NIEM message appears to be a tree with a root node. It is actually a directed graph with an initial node called the message object. For example, the message object in example 3-2 is the `msg:Request` element in the XML message. In the JSON message it is the value for the `msg:Request` key.

Every NIEM serialization has a mechanism for references; that is, a way for one object in the serialized graph to point to an object elsewhere in the graph. This mechanism supports cycles and avoids duplication in the graph data structure. (See section 5.2.)

Every message is an instance of a message format. A conforming message must satisfy the rules in section 12 and section 13. In particular, it must be valid according to the schema of its message format.

> A NIEM message was originally known as an *information exchange package (IEP)*, a term that found its way into the U.S. Federal Enterprise Architecture (2005). A message specification was originally known as an *information exchange package documentation (IEPD).* These terms are in widespread use within the NIEM community today, and will not go away soon (if ever).

3.1.2 Message format

A message format specifies the syntax of valid messages. This provides message developers with an exact description of the messages to be generated or processed by their software.

A message format includes a schema that can be used to assess the validity of a message. This schema is expressed in XML Schema (XSD) for XML message formats, and JSON Schema for JSON message formats. Example 3-3 shows a portion of the schemas for the two example messages in example 3-2.

```
<xs:complexType name="RequestType">
  <xs:sequence>
    <xs:element ref="msg:RequestID"/>
    <xs:element ref="msg:RequestedItem"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="Request" type="msg:RequestType"/>
```

```
{
  "msg:RequestType": {
    "type": "object",
    "properties": {
      "msg:RequestID": {"$ref": "#/properties/msg:RequestID"},
      "msg:RequestedItem": {"$ref": "#/properties/msg:RequestedItem"}
    },
    "required": [
      "msg:RequestID",
      "msg:RequestedItem"
    ]
  },
  "msg:Request": {
    "$ref": "#/definitions/msg:RequestType"
  }
}
```

Example 3-3: Example message format schemas

Producing and consuming systems may use the message format schema to validate the syntax of messages at runtime, but are not obligated to do so. Message developers may also use the schema during development for software testing. The schemas may also be used by developers for data binding; for example, Java Architecture for XML Binding (JAXB).

A message format belongs to exactly one message type. A conforming message format must satisfy the rules in section 11; in particular, it must be constructed so that every message that is valid according to the format also satisfies the information content constraints of its message type.

3.1.3 Message type

One important feature of NIEM is that every message has an equivalent message in every other supported serialization. These equivalent messages have a different message format, but have the same message type. For example, the XML message and the JSON message in example 3-2 above are equivalent. They represent the same information content, and can be converted one to the other without loss of information.

A message type specifies the information content of its messages without prescribing their syntax. A message type includes a message model, which is the means through which the message designer precisely defines the mandatory and optional content of conforming messages and the meaning of that content. This model is expressed in either of NIEM's two model representations, which are described in section 3.4 and section 3.5, and fully defined in section 4. Example 3-4 shows a portion of the message model for the two message formats in example 3-3.

```
<xs:complexType name="ItemType" appinfo:referenceCode="NONE">
  <xs:annotation>
    <xs:documentation>A data type for an article or thing.
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="structures:ObjectType">
      <xs:sequence>
        <xs:element ref="nc:ItemName"/>
        <xs:element ref="nc:ItemQuantity"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="ItemName" type="nc:TextType">
  <xs:annotation>
    <xs:documentation>A name of an item.</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="RequestedItem" type="nc:ItemType">
  <xs:annotation>
    <xs:documentation>A specification of an item request.</xs
  </xs:annotation>
</xs:element>
```

```
<Class structures:id="nc.ItemType">
  <Name>ItemType</Name>
  <Namespace structures:ref="nc" xsi:nil="true"/>
  <DocumentationText>A data type for an article or th
  <ReferenceCode>NONE</ReferenceCode>
  <ChildPropertyAssociation>
    <DataProperty structures:ref="nc.ItemName" xsi:nil="true"/>
    <MinOccursQuantity>1</MinOccursQuantity>
    <MaxOccursQuantity>1</MaxOccursQuantity>
  </ChildPropertyAssociation>
  <ChildPropertyAssociation>
    <DataProperty structures:ref="nc.ItemQuantity"
    <MinOccursQuantity>1</MinOccursQuantity>
    <MaxOccursQuantity>1</MaxOccursQuantity>
  </ChildPropertyAssociation>
</Class>
<DataProperty structures:id="nc.ItemName">
  <Name>ItemName</Name>
  <Namespace structures:ref="nc" xsi:nil="true"/>
  <DocumentationText>A name of an item.
  <Datatype structures:ref="nc.TextType" xsi:nil="true"/>
</DataProperty>
<ObjectProperty structures:id="msg.RequestedItem">
  <Name>RequestedItem</Name>
  <Namespace structures:ref="msg" xsi:nil="true"/>
  <DocumentationText>A specification of an item
  <Class structures:ref="nc.ItemType" xsi:nil="true"/>
  <ReferenceCode>NONE</ReferenceCode>
</ObjectProperty>
```

Example 3-4: Example message model in XSD and CMF

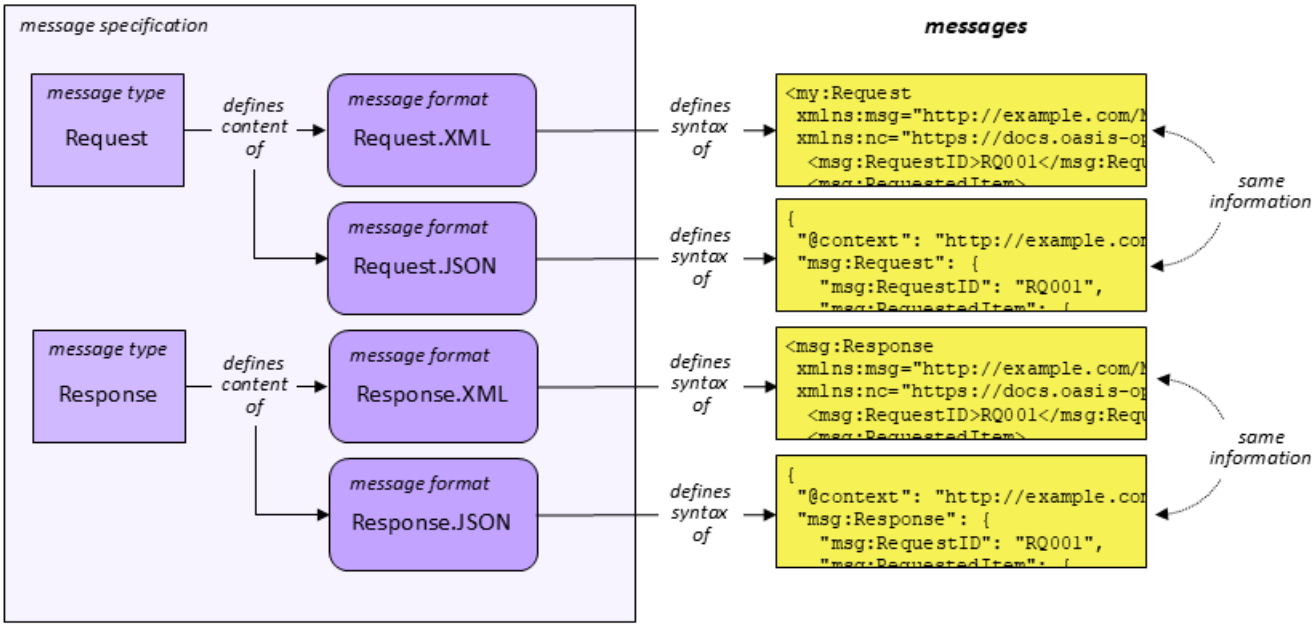
In addition to the message model, a message type also declares the initial property of conforming messages. In a conforming message, the message object is always the value of the initial property. For example, the message type for the message in example 3-2 declares that the initial property is `msg:Request`.

A message type provides all of the information needed to generate the schema for each message format it specifies. NIEMOpen provides free and open-source software tools to generate these schemas from the message model. (Message designers may also compose these schemas by hand, if desired.)

A conforming message type must satisfy all of the rules in section 11.

3.1.4 Message specification

A message specification is a collection of related message types. For instance, a Request message type might be paired with a Response message type as part of a request/response protocol. Those two message types could be collected into a message specification for the protocol, as illustrated below in example 3-5.



Example 3-5: Message specifications, types, and formats

Summary:

- A message specification defines one or more message types; a message type belongs to one message specification
- A message type defines one or more message formats; a message format belongs to one message type
- A message format defines the syntax of valid messages
- A message type defines the semantics of valid messages, plus their mandatory and optional content
- A message is an instance of a message format and of that format's message type

3.2 Reuse of community-agreed data models

NIEM is also a framework allowing communities to create reuse models for concepts that are useful in multiple data specifications. These community models are typically not *complete* for any particular specification. Instead, they reflect the community's judgement on which definitions are *worth the trouble of agreement* The NIEM core model contains definitions found useful by the NIEM community as a whole. NIEM domain models reuse the core, extending it with definitions found useful by the domain community. The core model plus the domain models comprise the "NIEM model". Figure 3-6 below illustrates the relationships between domain communities and community models.

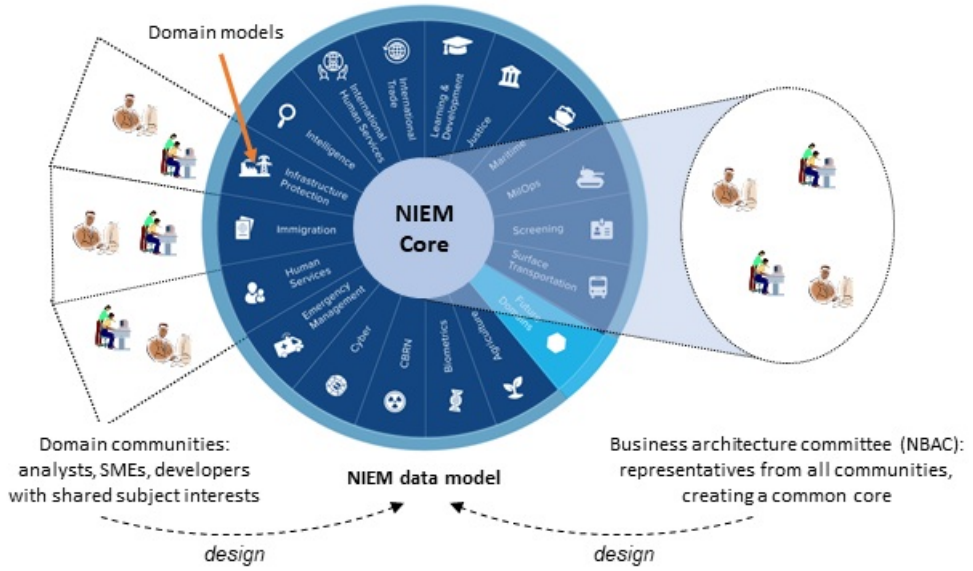


Figure 3-6: NIEM communities and data models

Message designers reuse definitions from the NIEM model, selecting a (usually small) subset of definitions that express a part of their information requirement. Message designers then create model extensions, adding components that do not yet exist in the NIEM model. These local extensions could be useful to others in the community beyond the scope of the original message, and may be submitted for potential adoption into the NIEM model (see https://github.com/niemopen/niem-model/issues).

NIEM's policy of easy model extension supports easy reuse of community data models. Because a community model does not need to be complete for the union of all needs, each community may focus its effort on its common needs, where the effort of agreement has the highest value. Data definitions that are not common, that are needed only for a particular message appear only as extensions in that message type, and need be learned only by the message developers who implement it. Model extensions are further described in section 3.7.

Data model reuse is especially useful in a large enterprise. Its value grows with the number of developer teams, and with the degree of commonality in the shared data. NIEM was originally designed for data sharing among federal, state, and local governments — where commonality and number of developer teams is large indeed.

3.3 Reuse of open standards

NIEM is built on a foundation of open standards, primarily:

- XML and XSD — message serialization and validation; also a modeling formalism
- JSON and JSON-LD — message serialization and linked data
- JSON Schema — message validation
- RDF, RDFS, and OWL — formal semantics
- ISO 11179 — conventions for data element names and documentation

One of NIEM's principles is to reuse well-known information technology standards when these are supported by free and open-source software. NIEM avoids reuse of standards that effectively depend on proprietary software. When the NIEMOpen project defines a standard of its own, it also provides free and open-source software to support it.

3.4 The NIEM metamodel

A data model in NIEM is either a message model, defining the information content of a message type, or a reuse model, making the agreed definitions of a community available for reuse. The information required for those purposes can itself be modeled. The model of that information is the *NIEM metamodel* – an abstract model for NIEM data models. The metamodel is expressed in UML, and is described in detail in section 4. At a high level, the major components of the metamodel are properties, classes, datatypes, namespaces, and models. Figure 3-7 provides an illustration.

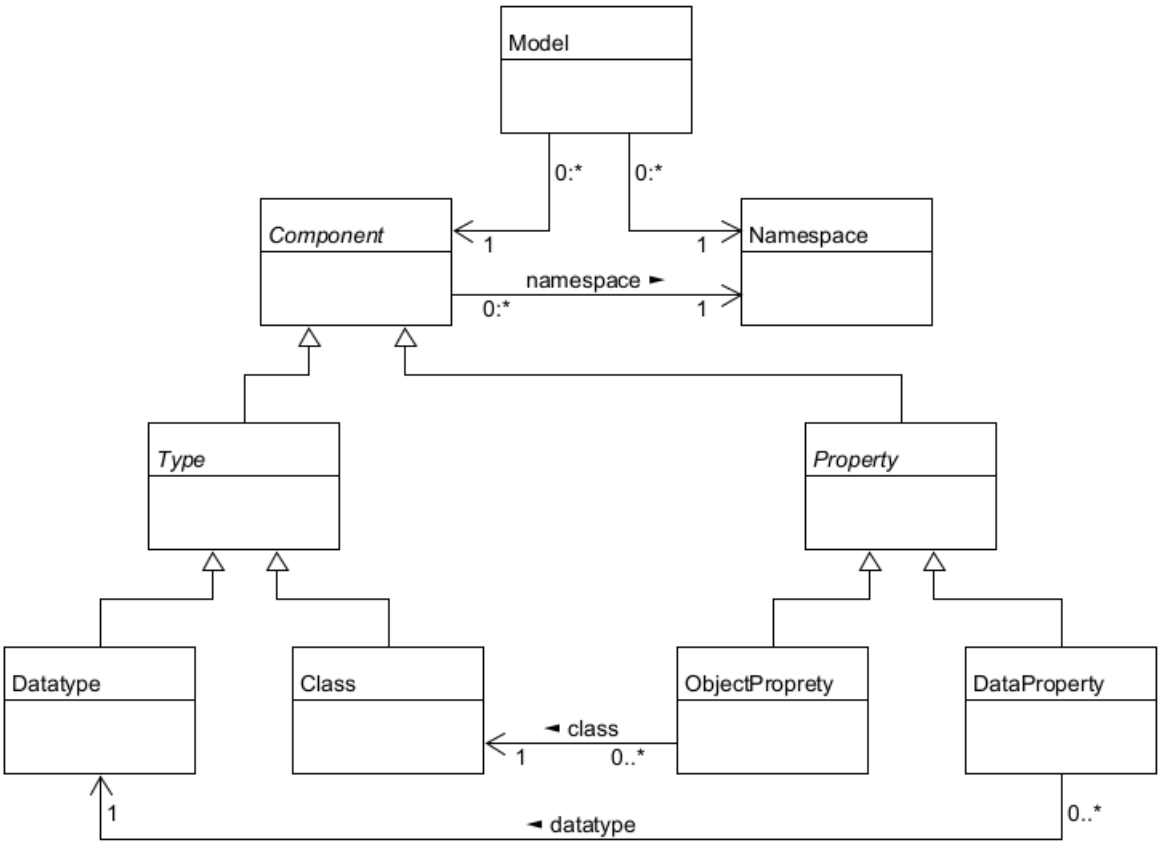


Figure 3-7: High-level view of the NIEM metamodel

- A *property* is a concept, idea, or thing. It defines a field that may appear in a message and can contain subfields (for objects / object properties) or a value (for literals / data properties). For example, in example 3-4, `req:RequestedItem` and `nc:ItemName` are names of properties. `req:RequestedItem` is an object property for the requested item; `nc:ItemName` is a data property for the name of the item. The meaning of these properties is captured in the documentation text.

- A *class* defines the properties that may appear in the content of a corresponding *object* in a message. A class has one or more *properties*. An *object property* in a class defines a subject-property-value relationship between two objects. A *data property* defines a relationship between an object and a literal value. In example 3-4, `nc:ItemType` is the name of a class.

- A *datatype* defines the allowed values of a corresponding *literal value* in a message. In example 3-4, `nc:TextType` is the name of a datatype.

- Classes and datatypes are the two kinds of *type* in the metamodel. For historical reasons, the name of every class and datatype in the NIEM model ends in “Type”. This is why the high-level view of the metamodel includes the abstract Type UML class.

- Classes, datatypes, and properties are the three kinds of metamodel *component*. (All of the common properties of classes and datatypes are defined in the Component class, which is why the abstract Type class is not needed in the detailed metamodel diagram in section 4.)

- A *namespace* is a collection of uniquely-named components defined by an authority. (See section 3.6)

- A *model* is a collection of components (organized into namespaces) and their relationships.

Figure 3-8 below illustrates the relationships among metamodel components, NIEM model components, and the corresponding message objects and values.

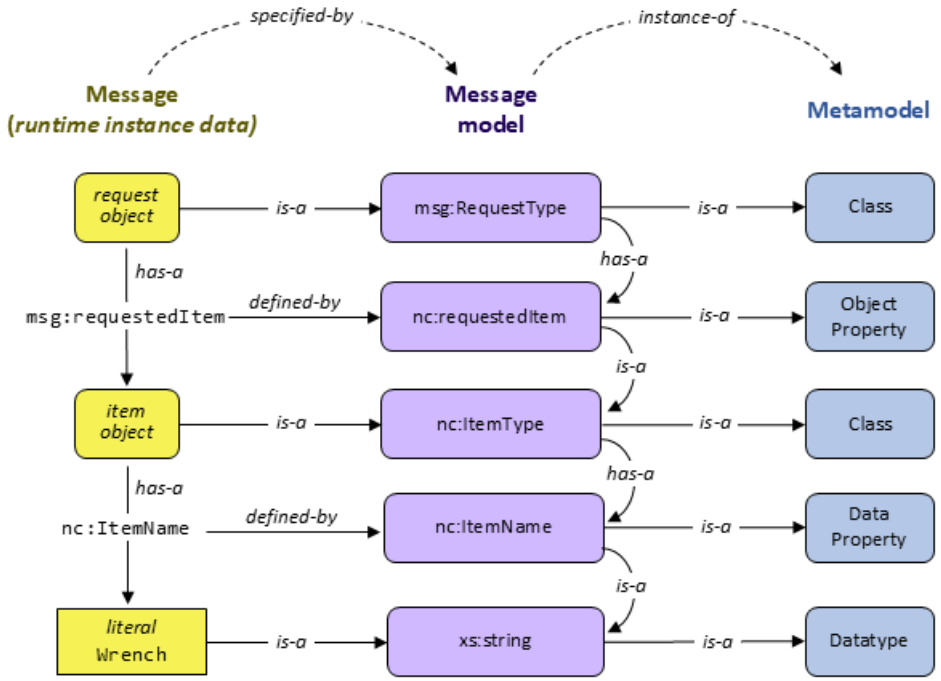


Figure 3-8: Message, message model, and metamodel relationships

A NIEM message contains properties which are based on objects or literal values. These are specified by the class, property, and datatype objects in a NIEM message model, which defines the content of a conformingmessage and also defines the meaning of that content. For example, in figure 3-8, the *item object* is defined by the `nc:ItemType` Class object; the *literal value* (`Wrench`) is defined by the `xs:string` Datatype object, and the property relationship between the two is defined by the `nc:ItemName` DataProperty object.

3.5 NIEM model representations: XSD and CMF

The abstract metamodel has two concrete representations: NIEM XSD and NIEM CMF. These are equivalent representations and may be converted from one to the other without loss. (NIEMOpen provides free and open-source software tools that perform the conversion; see software tools.)

Every version of NIEM uses a profile of XML Schema (XSD) as a NIEM model representation. In XSD, a NIEM model is represented as a schema assembled from a collection of schema documents. Every aspect of the metamodel is represented in some way by a schema component.

XSD as a model representation directly supports conformance testing of NIEM XML messages through schema validation. However, JSON developers (and developers working with other formats) cannot use XSD to validate their messages. Nor do they want to read XSD specifications of message content. For this reason, NIEM 6 introduces the Common Model Format (CMF), which is a NIEM model representation intended to support all developers.

CMF is the result of applying the NIEM framework to the information requirements in the metamodel. That result is a NIEM-based message type, which is part of a message specification, which is published in CMF. In CMF, a model is represented as an instance of that message type; that is, a CMF message, also known as a model file.

CMF is a technology-neutral model representation, because:

- A CMF model can be transformed into XSD for validation of XML messages, and into JSON Schema for validation of JSON messages.
- A CMF model can itself be represented in XML or JSON, according to developer preference. That is, like any other NIEM message, the CMF representation of a model can be serialized in either XML or JSON. For example, example 3-9 shows a portion of the message model from example 3-4 in both XML and JSON syntax.

```
<Class structures:id="nc.ItemType">                                  | {
  <Name>ItemType</Name>                                              |   "cmf:Class": {
  <Namespace structures:ref="nc" xsi:nil="true"/>                    |     "cmf:Name": "ItemType",
  <DocumentationText>A data type for an article or thing.</Docum     |     "cmf:Namespace": { "@id": "#nc" },
  <ReferenceCode>NONE</ReferenceCode>                                |     "cmf:DocumentationText": "A data type for an article
  <ChildPropertyAssociation>                                         |     "cmf:ReferenceCode": "NONE",
    <DataProperty structures:ref="nc.ItemName" xsi:nil="true"/>      |     "cmf:PropertyAssociation": {
    <MinOccursQuantity>1</MinOccursQuantity>                         |       "cmf:DataProperty": { "@id": "#nc.ItemName" },
    <MaxOccursQuantity>1</MaxOccursQuantity>                         |       "cmf:MinOccursQuantity": 1,
  </ChildPropertyAssociation>                                        |       "cmf:MaxOccursQuantity": 1
  <ChildPropertyAssociation>                                         |     },
    <DataProperty structures:ref="nc.ItemQuantity" xsi:nil="true     |     "cmf:PropertyAssociation": {
    <MinOccursQuantity>1</MinOccursQuantity>                         |       "cmf:DataProperty": { "@id": "#nc.ItemQuantity" },
    <MaxOccursQuantity>1</MaxOccursQuantity>                         |       "cmf:MinOccursQuantity": 1,
  </ChildPropertyAssociation>                                        |       "cmf:MaxOccursQuantity": 1
</Class>                                                             |     }
                                                                     |   }
                                                                     | }
```

Example 3-9: CMF model in XML and JSON syntax

Section 4 defines the mappings between the metamodel, NIEM XSD, and CMF.

> While NIEM uses JSON Schema to validate JSON messages, there is no JSON Schema representation of the metamodel, because JSON Schema does not have all of the necessary features to represent NIEM models.

3.6 Namespaces

The components of a NIEM model are partitioned into *namespaces.* This prevents name clashes among communities or domains that have different business perspectives, even when they choose identical data names to represent different data concepts.

Each namespace has an author, a person or organization that is the authoritative source for the namespace definitions. A namespace is the collection of model components for concepts of interest to the namespace author. Namespace cohesion is important: a namespace should be designed so that its components are consistent, may be used together, and may be updated at the same time.

Each namespace must be uniquely identified by a URI. The namespace author must also be the URI's owner, as defined by [webarch]. Both URNs and URLs are allowed. It is helpful, but not required, for the namespace URI to be accessible, returning the definition of the namespace content in a supported model format.

NIEM defines two categories of authoritative namespace: reference namespace and extension namespace.

- *Reference namespace:* The NIEM model is a reuse model comprised entirely of reference namespaces. The components in these namespaces are intended for the widest possible reuse. They provide names and definitions for concepts, and relations among them. These namespaces are characterized by "optionality and over-inclusiveness". That is, they define more concepts than needed for any particular data exchange specification, without cardinality constraints, so it is easy to select the concepts that are needed and omit the rest. They also omit unnecessary range or length constraints on property datatypes.

  A reference namespace is intended to capture the meaning of its components. It is not intended for a complete definition of any particular message type. Message designers are expected to subset, profile, and extend the components in reference namespaces as needed to match their information exchange requirements.

- *Extension namespace:* The components in an extension namespace are intended for reuse within a more narrow scope than those defined in a reference namespace. These components express the additional vocabulary required for an information exchange, above and beyond the vocabulary available from the NIEM model. The intended scope is often a particular message specification. Sometimes a community or organization will define an extension namespace for components to be reused in several related message specifications. In this case, the namespace components may also omit cardinality and datatype constraints, and may be incomplete for any particular message type.

  Message designers are encouraged to subset, profile, and extend the components in extension namespaces created by

another author when these satisfy their modeling needs, rather than create new components.

Namespaces are the units of model configuration management. Once published, the components in a reference namespace or extension namespace may not be removed or changed in meaning. A change of that nature may only be made in a new namespace with a different URI.

As a result of this rule, once a specific version of a namespace is published, it can no longer be modified. Updates must go into a new version of the namespace. All published versions of a namespace continue to be valid in support of older exchanges.

In addition, note that a message specification contains its own copy of the schemas that they depend upon. Therefore new versions of a model or a namespace do not affect existing exchanges. Exchange partners may decide to upgrade to a new version of NIEM if they decide it suits their needs, but only if they choose to do so, and only on their own timeline. The NIEM release schedule does not force adopters to keep in sync.

Message designers almost never require *all* the components in the NIEM model, and so NIEM defines a third namespace category:

- *Subset namespace:* Technically, this is a “namespace subset”, which contains only some of the components of a reference namespace or extension namespace. It provides components for reuse, while enabling message designers and developers to:
  - Omit optional components in a reference namespace or extension namespace that they do not need.
  - Provide cardinality and datatype constraints that precisely define the content of one or more message types.

  All message content that is valid for a subset namespace must also be valid for the reference namespace or extension namespace with the same URI. Widening the value space of a component is not allowed. Adding components is not allowed. Changing the documentation of a component is not allowed.

NIEM has a fourth namespace category, for namespaces containing components from standards or specifications that are based on XML but not based on NIEM.

- *External namespace:* Any namespace defined by a schema document that is not:
  - a reference namespace
  - an extension namespace
  - a subset namespace
  - the structures namespace, https://docs.oasis-open.org/niemopen/ns/model/structures/6.0/
  - the XML namespace, http://www.w3.org/XML/1998/namespace.

  XML attributes defined in an external namespace may be part of a NIEM model. XML elements defined in an external namespace are not part of a NIEM model, but may be used as properties of an adapter type *(see §9.4)*.

Three special namespaces do not fit into any of the four categories:

- The structures namespace is not part of any NIEM model. It provides base types and attributes that are used in the XSD representation of NIEM models.
- The XML namespace is not considered to be an external namespace. It defines the `xml:lang` attribute, which may be a component in a NIEM model.
- The XSD namespace (http://www.w3.org/2001/XMLSchema) defines the primitive datatypes (`xs:string`, etc.) This namespace appears explicitly in CMF model representations, and is implicitly part of every XSD representation.

3.7 Model extensions

Reuse of a community data model typically supplies some but not all of the necessary data definitions. Model extension allows a model designer to supply the missing definitions. NIEM has two forms of model extension: subclassing and augmentation.

In a *subclass*, a namespace designer creates a new class in his own namespace to represent a special kind of thing. The new class shares all of the properties of its parent class, and adds properties belonging only to the new class. For example, in the NIEM model, `nc:Vehicle` is a subclass of `nc:Conveyance`. Like any Conveyance, a Vehicle may have the `nc:ConveyanceEngineQuantity` property, but only Vehicles have the `nc:VehicleSeatingQuantity` property; other Conveyances do not.

In an *augmentation*, a namespace designer creates additional properties for a class that is defined in a different namespace. Here the designer is not creating a new class for a new kind of thing. Instead, he is providing properties which could have been defined by the original class designer, but in fact were not. For example, the designers of the NIEM Justice domain have augmented `nc:PersonType` with the `j:PersonSightedIndicator` property, because for the members of the Justice domain it is useful to record whether a person is able to see, even though to the NIEM community as a whole, adding this property to NIEM Core is not worth the trouble.

In general, augmentations are preferred over subclassing. At present the NIEM metamodel does not support multiple inheritance. If several domains were to create a subclass of `nc:PersonType`, there would be no way for a message designer to combine in his message model the properties of a person from NIEM Justice, NIEM Immigration, etc. That combination is easily done with augmentations.

---

4. Data models in NIEM

The NIEM metamodel is an abstract model that specifies the content of a NIEM data model. It is described by the UML diagram in figure 4-1 below.

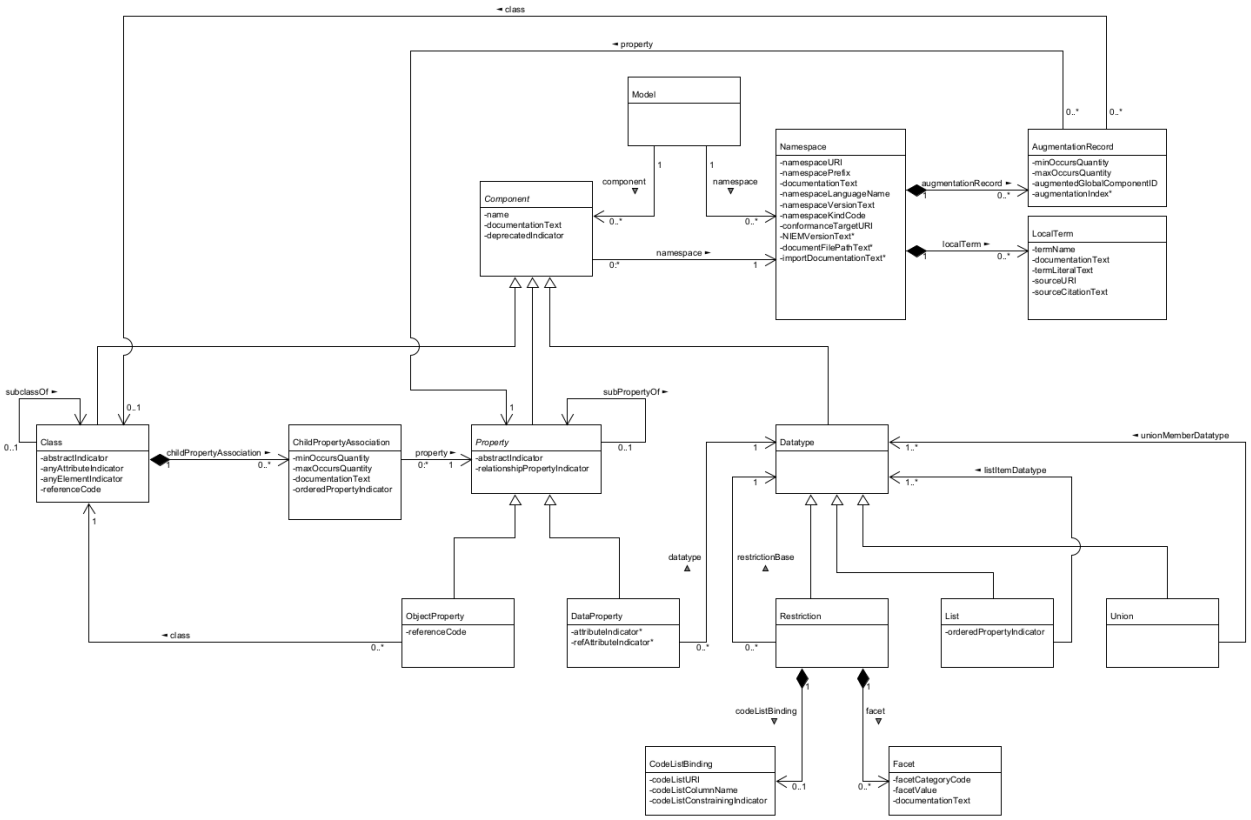


Figure 4-1:The NIEM metamodel

This section specifies:

- the meaning of the classes, attributes, and relationships in the metamodel
- the meaning of the classes, datatypes, and properties in CMF, which implements the metamodel

- the XSD constructs that correspond to CMF classes, datatypes, and properties, and which also implement the metamodel

In addition to the UML diagram, this section contains several tables that document the classes, attributes, and relationships in the metamodel. These tables have the following columns:

Column	Definition
Name	the name of the class, attribute, or relationship
Definition	the definition of the object or property
Card	the number of times this property may/must appear in an object
Ord	true when the order of the instances of a repeatable property in an object is significant
Range	the class or datatype of a property

Table 4-2: Definition of columns in metamodel property tables

Classes, attributes, and relationships have the same names in the metamodel and in CMF. (Attributes and relationship names have lower camel case in the diagram and tables, following the UML convention. The tables and the CMF specification use the same names in upper camel case, following the NIEM convention.)

The definitions in these tables follow NIEM rules for documentation (which are described in section 7.2). As a result, the definition of each metamodel class begins with “A data type for…” instead of “A class for…”. (For historical reasons, the name of every class and datatype in the NIEM model ends in “Type”, and this is reflected in the conventions for documentation; see section 3.4.)

Names from CMF and the metamodel do not appear in the XSD representation of a model. Instead, NIEM defines special interpretations of XML Schema components, making the elements and attributes in an XSD schema document equivalent to CMF model components. The mapping between CMF components and XSD schema components is provided by a table in each section below, with these columns:

Column	Definition
CMF	CMF component name
XSD	XSD equivalent

Table 4-3: Definition of columns in CMF-XSD mapping tables

4.1 Model

A Model object represents a NIEM model.

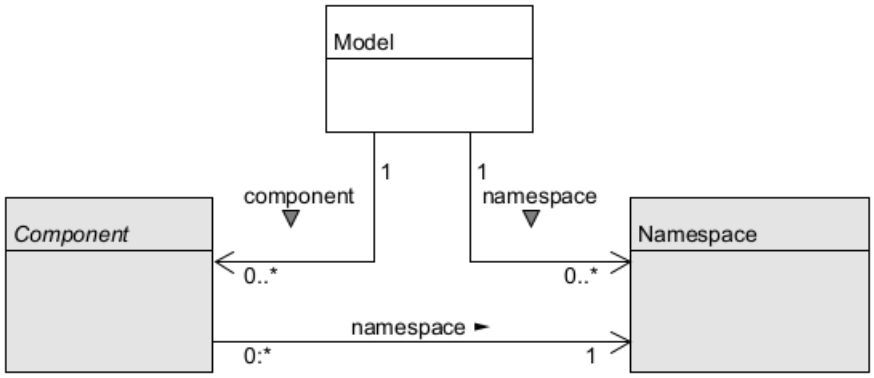


Figure 4-4: Model class diagram

Name	Definition	Card	Ord	Range
Model	A data type for a NIEM data model.			
Component	A data concept for a component of a NIEM data model.	0..*	-	ComponentType
Namespace	A namespace of a data model component	0..*	-	NamespaceType

Table 4-5: Properties of the Model object class

In XSD, an instance of the Model class is represented by a schema document set.

4.2 Namespace

A Namespace object represents a namespace in a model. For example, the namespace with the URI https://docs.oasis-open.org/niemopen/ns/model/niem-core/6.0/ is a namespace in the NIEM 6.0 model.

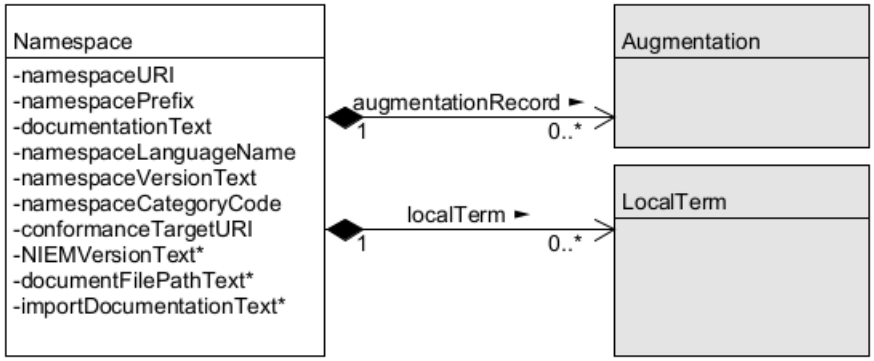


Figure 4-6: Namespace class diagram

Name	Definition	Card	Ord	Range
Namespace	A data type for a namespace.			
NamespaceURI	A URI for a namespace.	1	-	xs:anyURI
NamespacePrefixText	A namespace prefix name for a namespace.	1	-	xs:NCName

Name	Definition	Card	Ord	Range
DocumentationText	A human-readable text documentation of a namespace.	1..*	Y	TextType
NamespaceLanguageName	A name of a default language of the terms and documentation text in a namespace.	1	-	xs:language
NamespaceVersionText	A version of a namespace; for example, used to distinguish a namespace subset, bug fix, documentation change, etc.	1	-	xs:token
NamespaceCategoryCode	A kind of namespace in a NIEM model (external, core, domain, etc.).	1	-	NamespaceCategoryCodeType
ConformanceTargetURI	A conformance target identifier.	0..*	-	xs:anyURI
NIEMVersionText	A NIEM version number of the builtin schema components used in a namespace; e.g. “5” or “6”.	0..1	-	xs:token
DocumentFilePathText	A relative file path from the top schema directory to a schema document for this namespace.	0..1	-	xs:string
ImportDocumentationText	Human-readable documentation from the first `xs:import` element importing this namespace.	0..1	-	xs:string
AugmentationRecord	An augmentation of a class with a property by a namespace.	0..*	-	AugmentationType
LocalTerm	A data type for the meaning of a term that may appear within the name of a model component.	0..*	-	LocalTermType

Table 4-7: Properties of the Namespace object class

In XSD, an instance of the Namespace class is represented by the `<xs:schema>` element in a schema document. Example 4-8 shows the representation of a Namespace object in CMF and in the corresponding XSD.

```
<Namespace>
  <NamespaceURI>https://docs.oasis-open.org/niemopen/ns/model/niem-core/6.0/</NamespaceURI>
  <NamespacePrefixText>nc</NamespacePrefixText>
  <DocumentationText>NIEM Core.</DocumentationText>
  <ConformanceTargetURI>
    https://docs.oasis-open.org/niemopen/ns/specification/NDR/6.0/#ReferenceSchemaDocument
  </ConformanceTargetURI>
  <NamespaceVersionText>ps02</NamespaceVersionText>
  <NamespaceLanguageName>en-US</NamespaceLanguageName>
</Namespace>
---------------
<xs:schema
  targetNamespace="https://docs.oasis-open.org/niemopen/ns/model/niem-core/6.0/"
  xmlns:ct="https://docs.oasis-open.org/niemopen/ns/specification/conformanceTargets/6.0/"
  xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem-core/6.0/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  ct:conformanceTargets="https://docs.oasis-open.org/niemopen/ns/specification/NDR/6.0/#ReferenceSchemaDocument"
  version="ps02"
  xml:lang="en-US">
  <xs:annotation>
    <xs:documentation>NIEM Core.</xs:documentation>
  </xs:annotation>
</xs:schema>
```

Example 4-8: Namespace object in CMF and XSD

The following table shows the mapping between Namespace object representations in CMF and XSD.

CMF	XSD
NamespaceURI	`xs:schema/@targetNamespace`
NamespacePrefixText	The prefix in the first namespace declaration of the target namespace
DocumentationText	`xs:schema/xs:annotation/xs:documentation`
ConformanceTargetURI	Each of the URIs in the list attribute `xs:schema/@ct:conformanceTargets`
NamespaceVersionText	`xs:schema/@version`
NamespaceLanguageName	`xs:schema/@xml:lang`

Table 4-9: Namespace object properties in CMF and XSD

4.3 Component

A Component is either a Class object, a Property object, or a Datatype object in a NIEM model. This abstract class defines the common properties of those three concrete subclasses.

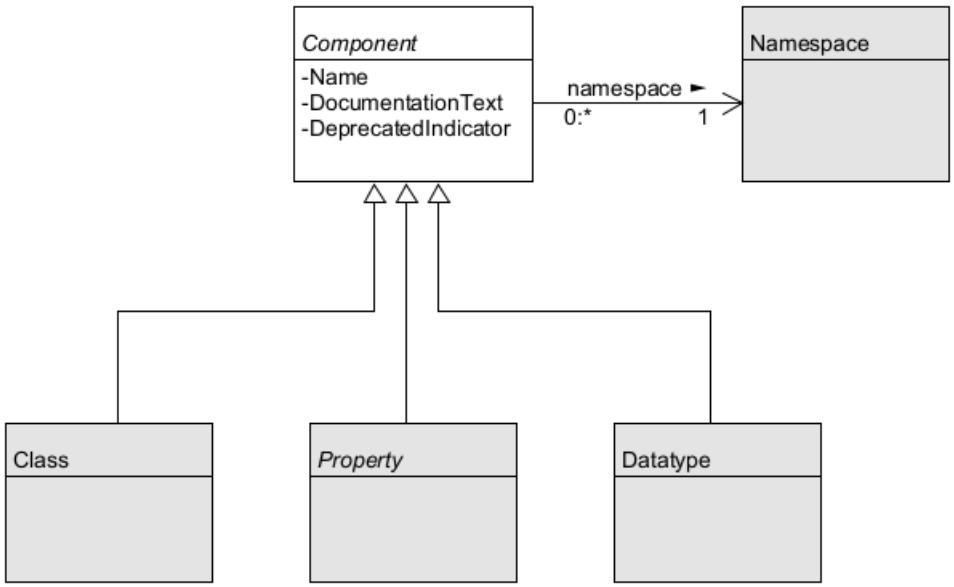


Figure 4-10: Component class diagram

Name	Definition	Card	Ord	Range
Component	A data type for common properties of a data model component in NIEM.			
Name	The name of a data model component.	1	-	xs:NCName
DocumentationText	A human-readable text definition of a data model component.	0..*	Y	TextType
DeprecatedIndicator	True for a deprecated schema component; that is, a component that is provided, but the use of which is not recommended.	0..1	-	xs:boolean
Namespace	The namespace of a data model component.	1	-	NamespaceType

Table 4-11: Properties of the Component abstract class

In XSD, the common properties of a Component object are represented by a complex type definition or an element or attribute declaration. Example 4-12 shows the representation of those common properties in CMF and XSD.

```
<DataProperty>
  <Name>ActivityCompletedIndicator</Name>
  <Namespace structures:ref="nc"/>
  <DocumentationText>True if an activity has ended; false otherwise.</DocumentationText>
  <DeprecatedIndicator>false</DeprecatedIndicator>
---------------
<xs:element name="ActivityCompletedIndicator" type="niem-xs:boolean" appinfo:deprecated="false">
  <xs:annotation>
    <xs:documentation>True if an activity has ended; false otherwise.</xs:documentation>
  </xs:annotation>
</xs:element>
```

Example 4-12: Component object (abstract) in CMF and XSD

The following table shows the mapping between Component object properties in CMF and XSD.

CMF	XSD

CMF	XSD
Name	`@name` of element or attribute declaration
NamespaceURI	`@targetNamespace` of schema document
DocumentationText	`xs:annotation/xs:documentation` of element or attribute declaration
DeprecatedIndicator	'@appinfo:deprecated` of element or attribute declaration

Table 4-13: Component object properties in CMF and XSD

4.4 Class

A Class object represents a class of message objects defined by a NIEM model. For example, `nc:ItemType` is a Class object in the NIEM Core model.

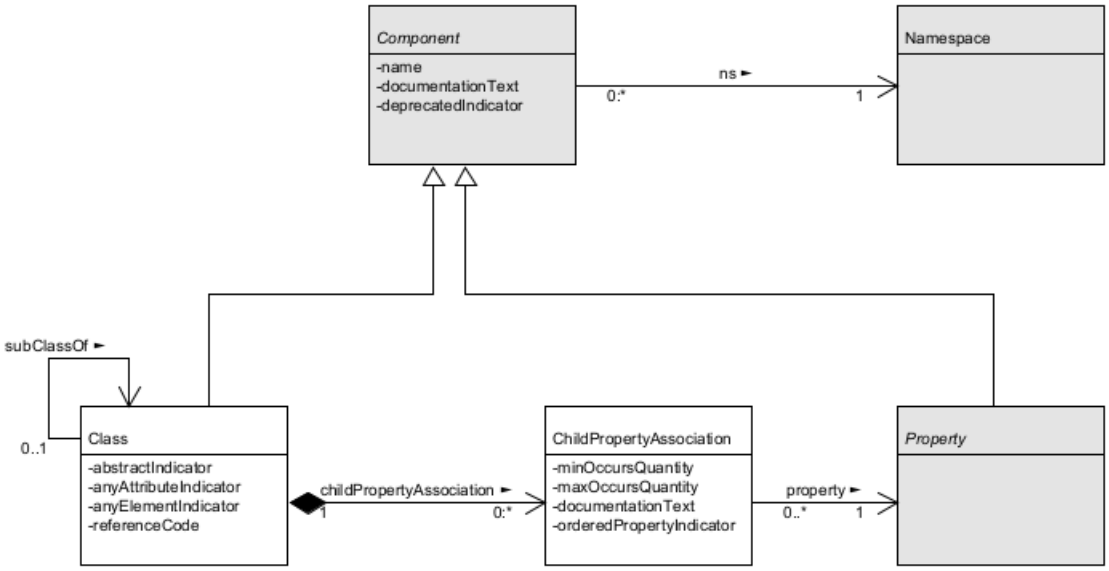


Figure 4-14: Class and ChildPropertyAssociation class diagram

Name	Definition	Card	Ord	Range
Class	A data type for a class.			
AbstractIndicator	True if a class is a base for extension, and must be specialized to be used directly; false if a class may be used directly.	0..1	-	xs:boolean
AnyAttributeIndicator	True when instances of a class may have arbitrary attribute properties in addition to those specified by ChildPropertyAssociation.	0..1	-	xs:boolean
AnyElementIndicator	True when instances of a class may have arbitrary element properties in addition to those specified by ChildPropertyAssociation.	0..1	-	xs:boolean

Name	Definition	Card	Ord	Range
ReferenceCode	A code describing how a property may be referenced (or must appear inline).	0..1	-	ReferenceCodeType
SubClassOf	A base class of a subclass.	0..1	-	ClassType
ChildPropertyAssociation	An association between a class and a child property of that class.	0..*	Y	ChildPropertyAssociationType

Table 4-15: Properties of the Class object class

The range of the `ReferenceCode` property is a code list with the following codes and meanings:

Code	Definition
REF	A code for a property that may be referenced by an IDREF (in XML) or NCName (in JSON).
URI	A code for a property that may be referenced by a URI.
ANY	A code for a property that may be reference by IDREF/NCName or URI.
NONE	A code for a property that my not be referenced and must appear inline.

Table 4-16: ReferenceCode code list

Class objects may be categorized into four groups, as follows:

- An object class contains one or more properties from a conforming namespace. An object class has a name ending in "Type". Most class objects fall into this category.
- An adapter class contains only properties from a single external namespace. It acts as a conformance wrapper around data components defined in standards that are not NIEM conforming. An adapter class has a name ending in "AdapterType". (See section 9.4.)
- An association class represents a specific relationship between objects. Associations are used when a simple NIEM property is insufficient to model the relationship clearly, or to model properties of the relationship itself. An association class has a name ending in "AssociationType".
- A literal class contains no object properties, at least one attribute property, and exactly one element property. A literal class has a name ending in "Type".

The instances of most classes (including adapter and association classes) are represented in XML as an element with complex content; that is, with child elements, and sometimes with attributes. For example, example 4-17 shows an XML element with complex content, and also the equivalent in a JSON message.

```
<ex:ItemWeightMeasure>                                  | {
  <ex:MassUnitCode>KGM</unece:MassUnitCode>             |   "ex:ItemWeightMeasure": {
  <ex:MeasureDecimalValue>22.5</ex:MeasureDecimalValue> |     "ex:MassUnitCode": "KGM",
</ex:ItemWeightMeasure>                                 |     "ex:MeasureDecimalValue": 22.5
                                                        |   }
                                                        | }
```

Example 4-17: Instance of a class in XML and JSON

These classes are represented in XSD as a complex type with complex content ("CCC type"); that is, a type with child elements. Example 4-18 below shows a ordinary Class object defining the class of the `ItemWeightMeasure` property in the example above, represented first in CMF, and then in XSD as a complex type with child elements.

```
<Class structures:id="ex.WeightMeasureType">
  <Name>WeightMeasureType</Name>
  <Namespace structures:ref="ex" xsi:nil="true"/>
  <ChildPropertyAssociation>
    <DataProperty structures:ref="ex.MassUnitCode" xsi:nil="true"/>
    <MinOccursQuantity>1</MinOccursQuantity>
    <MaxOccursQuantity>1</MaxOccursQuantity>
  </PropertyAssociation}>
  <ChildPropertyAssociation>
    <DataProperty structures:ref="ex.MeasureDecimalValue" xsi:nil="true"/>
    <MinOccursQuantity>1</MinOccursQuantity>
    <MaxOccursQuantity>1</MaxOccursQuantity>
  </PropertyAssociation>
</Class>
---------------
<xs:complexType name="WeightMeasureType">
  <xs:complexContent>
    <xs:extension base="structures:ObjectType">
      <xs:sequence>
        <xs:element ref="ex:MassUnitCode"/>
        <xs:element ref="ex:MeasureDecimalValue"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Example 4-18: A Class object in CMF and XSD (CCC type)

The following table shows the mapping between Class object representations in CMF and XSD.

CMF	XSD
AbstractIndicator	`xs:complexType/@abstract`
AnyAttributeIndicator	`xs:anyAttribute`
AnyElementIndicator	`xs:any`
ReferenceCode	`xs:complexType/@appinfo:referenceCode`
SubClassOf	`xs:complexType/xs:complexContent/xs:extension/@base`
ChildPropertyAssociation	`xs:complexType/xs:complexContent/xs:extension/xs:sequence/xs:element` or `xs:complexType/xs:complexContent/xs:extension/xs:attribute`

Table 4-19: Class object object properties in CMF and XSD

Instances of a literal class are represented as an element with simple content and attributes in XML. Example 4-20 below shows an XML and JSON instance of a literal class.

```
<ex:ItemWeightMeasure ex:massUnitCode="KGM">
  22.5
</ex:ItemWeightMeasure>
```

```
| {
|   "ex:ItemWeightMeasure": {
|     "ex:massUnitCode": "KGM",
|     "ex:WeightMeasureLiteral": 22.5
|   }
| }
```

Example 4-20: Instance of a literal class in XML and JSON

A literal class is represented in XSD as a complex type with simple content (“CSC type”) and attributes. This is illustrated in example 4-21 below, which shows a literal class defining the class of the `ItemWeightMeasure` property in example 4-20 above.

```
  <Class structures:id="ex.WeightMeasureType">
    <Name>WeightMeasureType</Name>
    <Namespace structures:ref="ex" xsi:nil="true"/>
    <ChildPropertyAssociation>
      <DataProperty structures:ref="ex.massUnitCode" xsi:nil="true"/>
      <MinOccursQuantity>1</MinOccursQuantity>
      <MaxOccursQuantity>1</MaxOccursQuantity>
    </ChildPropertyAssociation>
    <ChildPropertyAssociation>
      <DataProperty structures:ref="ex.WeightMeasureLiteral" xsi:nil="true"/>
      <MinOccursQuantity>1</MinOccursQuantity>
      <MaxOccursQuantity>1</MaxOccursQuantity>
    </ChildPropertyAssociation>
  </Class>
---------------
<xs:complexType name="WeightMeasureType">
  <xs:simpleContent>
    <xs:extension base="xs:decimal">
      <xs:attribute ref="ex:massUnitCode" use="required"/>
      <xs:attributeGroup ref="structures:SimpleObjectAttributeGroup"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Example 4-21: A literal class object in CMF and XSD (CSC type)

A literal class always has one DataProperty that is not an attribute property. This property is named after the class, with “Type” replaced by “Literal” It does not appear in the XSD representation of the literal class, or as a separate element in the XML message.

A literal class always has at least one attribute property. In XSD, a complex type with simple content and no attributes represents a Datatype, not a Class.

4.5 ChildPropertyAssociation

An instance of the ChildPropertyAssociation class represents an association between a class and a child property of that class. For example, `nc:PersonMiddleName` property and `nc:personNameCommentText` are two child properties of the ’nc:PersonType` class.

Name	Definition	Card	Ord	Range
ChildPropertyAssociation	A data type for an occurrence of a property as content of a class.			
MinOccursQuantity	The minimum number of times a property may occur within an object of a class.	1	-	xs:integer
MaxOccursQuantity	The maximum number of times a property may occur within an object of a class.	1	-	MaxOccursType
DocumentationText	A human-readable documentation of the association between a class and a child property content of that class.	0..*	Y	TextType
OrderedPropertyIndicator	True if the order of a repeated property within an object is significant.	0..1	-	xs:boolean
Property	The property that occurs in the class.	1	-	PropertyType

Table 4-22: Properties of the ChildPropertyAssociation object class

A ChildPropertyAssociation object is represented in XSD as an element or attribute reference within a complex type definition. Example 4-23 shows the representation of two PropertyAssociation objects, first in CMF, and then in XSD.

```
<ChildPropertyAssociation>
  <ObjectProperty structures:ref="nc.PersonMiddleName" xsi:nil="true"/>
  <MinOccursQuantity>0</MinOccursQuantity>
  <MaxOccursQuantity>unbounded</MaxOccursQuantity>
  <DocumentationText>
    Documentation here is unusual; it refers to the association between the object and this property.
  </DocumentationText>
  <OrderedPropertyIndicator>true</OrderedPropertyIndicator>
</ChildPropertyAssociation>
<ChildPropertyAssociation>
  <DataProperty structures:ref="nc:personNameCommentText" xsi:nil="true"/>
  <MinOccursQuantity>0</MinOccursQuantity>
  <MaxOccursQuantity>1</MaxOccursQuantity>
</ChildPropertyAssociation>
---------------
<xs:sequence>
  <xs:element ref="nc:PersonMiddleName"
    minOccurs="0" maxOccurs="unbounded" appinfo:orderedPropertyIndicator="true">
    <xs:annotation>
      <xs:documentation>
        Documentation here is unusual; it refers to the relationship between the object and this property.
      </xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:sequence>
<xs:attribute ref="nc:personNameCommentText" use="optional"/>
```

Example 4-23: PropertyAssociation object in CMF and XSD

The following table shows the mapping between PropertyAssociation representations in CMF and XSD.

CMF	XSD
Property	The property object for `xs:element/@ref` or `xs:attribute/@ref`.
MinOccursQuantity	`xs:element/@minOccurs` or `xs:attribute/@use`
MaxOccursQuantity	`xs:element/@maxOccurs`
DocumentationText	`xs:element/xs:annotation/xs:documentation` or `xs:attribute/xs:annotation/xs:documentation`
OrderedPropertyIndicator	`xs:element/@appinfo:orderedPropertyIndicator`
AugmentingNamespace	`xs:element/@appinfo:augmentingNamespace` or `xs:attribute/@appinfo:augmentingNamespace`

Table 4-24: ChildPropertyAssociation object properties in CMF and XSD

4.6 Property

A Property object is either an ObjectProperty or a DataProperty in a NIEM model. This abstract class defines the common properties of those two concrete subclasses.

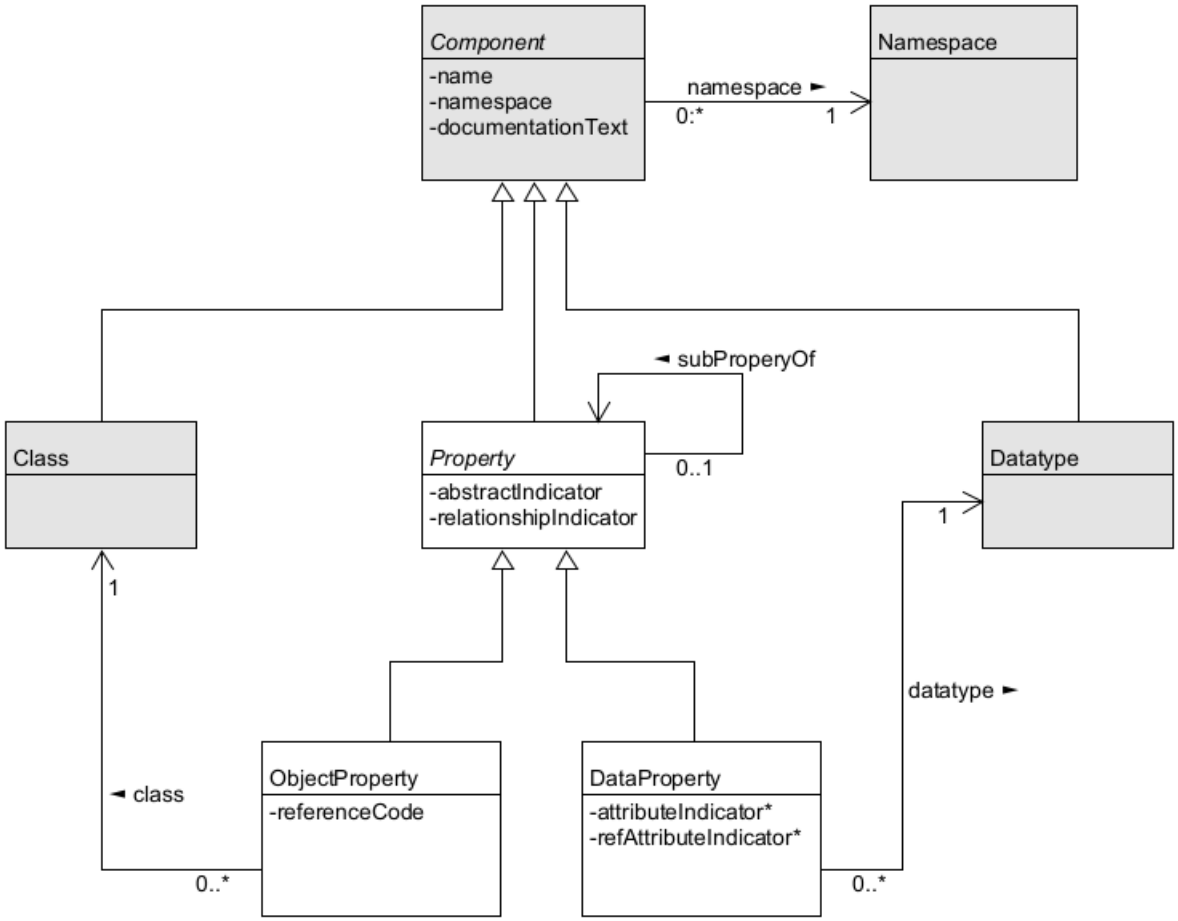


Figure 4-25: Property class diagram

Name	Definition	Card	Ord	Range
Property	A data type for a property.			
AbstractIndicator	True if a property must be specialized; false if a property may be used directly.	0..1	-	xs:boolean
RelationshipIndicator	True for a relationship property, a property that applies to the relationship between its parent and grandparent objects.	0..1	-	xs:boolean
SubPropertyOf	A property of which a property is a subproperty.	0..1	-	PropertyType

Table 4-26: Properties of the Property abstract class

Apart from the message object, every object in a message is a child property of another object, and typically provides information about that object. A relationship property instead provides information about the relationship between its parent and grandparent objects. Section 5.5 provides an example.

The examples of a Property object in CMF and XSD, and the table showing the mapping between the CMF and XSD representations, are shown below in the definitions of the concrete subclasses, ObjectProperty and DataProperty.

4.7 ObjectProperty

An instance of the ObjectProperty class represents a property in a NIEM model with a range that is a class. For example, the

`nc:PersonMiddleName` object in the NIEM core model is an object property with a range of the `nc:PersonNameTextType` class.

Name	Definition	Card	Ord	Range
ObjectProperty	A data type for an object property.			
ReferenceCode	A code describing how a property may be referenced (or must appear inline).	0..1	-	ReferenceCodeType
Class	The class of this object property.	1	-	ClassType

Table 4-27: Properties of the ObjectProperty object class

An ObjectProperty object is represented in XSD as an element declaration with a type that is a Class object.Example 4-28 shows an ObjectProperty object, represented first in CMF, and then in XSD.

```
<ObjectProperty structures:id="ex.ExampleObjectProperty">
  <Name>ExampleObjectProperty</Name>
  <Namespace structures:ref="ex" xsi:nil="true"/>
  <DocumentationText>Documentation text for ExampleObjectProperty.</DocumentationText>
  <DeprecatedIndicator>false</DeprecatedIndicator>
  <AbstractIndicator>true</AbstractIndicator>
  <ReferenceCode>URI</ReferenceCode>
  <Class structures:ref="ex.ExType" xsi:nil="true"/>
</ObjectProperty>
---------------
<xs:element name="ExampleObjectProperty" type="ex:ExType" abstract="true" appinfo:referenceCode="URI">
  <xs:annotation>
    <xs:documentation>Documentation text for ExampleObjectProperty.</xs:documentation>
  </xs:annotation>
</xs:element>
```

Example 4-28: ObjectProperty object in CMF and XSD

The following table shows the mapping between ObjectProperty object representations in CMF and XSD.

CMF	XSD
Namespace	The namespace object for the containing schema document.
Name	`xs:complexType/@name`
DocumentationText	`xs:complexType/xs:annotation/xs:documentation`
DeprecatedIndicator	`xs:complexType/@appinfo:deprecated`
AbstractIndicator	`xs:complexType/@abstract`
SubPropertyOf	The property object for `xs:element/@substitutionGroup`
RelationshipPropertyIndicator	`xs:element/@appinfo:relationshipPropertyIndicator`
Class	The class object for `xs:element/@type`
ReferenceCode	`xs:complexType/@appinfo:referenceCode`

Table 4-29: ObjectProperty object properties in CMF and XSD

4.8 DataProperty

An instance of the DataProperty class represents a property in a NIEM model with a range that is a datatype. For example, the `nc:personNameCommentText` property in the NIEM core model is a data property with a range of the `xs:string` datatype.

Name	Definition	Card	Ord	Range
DataProperty	A data type for a data property.			
AttributeIndicator	True for a property that is represented as attributes in XML.	0..1	-	xs:boolean
RefAttributeIndicator	True for a property that is an reference attribute property.	0..1	-	xs:boolean
Datatype	The datatype of this data property.	1	-	DatatypeType

Table 4-30: Properties of the DataProperty object class

An attribute property is a data property in which `AttributeIndicator` is true. These are represented in XSD as an attribute declaration.

A reference attribute property is an attribute property that contains one or more identifiers for message objects of a known class. It is interpreted as an [object reference] to each object thus identified. Object references and identifiers are described in section 5.3, and reference attribute properties in section 5.3.6.

A DataProperty object is represented in XSD as an attribute declaration, or as an element declaration with a type that is a Datatype object. Example 4-31 shows the representations of two DataProperty objects, first in CMF, and then in the corresponding XSD.

```
<DataProperty structures:id="ex.ExampleDataProperty">
  <Name>ExampleDataProperty</Name>
  <Namespace structures:ref="ex" xsi:nil="true"/>
  <DocumentationText>Documentation text for ExampleDataProperty.</DocumentationText>
  <DeprecatedIndicator>true</DeprecatedIndicator>
  <AbstractIndicator>true</AbstractIndicator>
  <SubPropertyOf structures:ref="ex.PropertyAbstract" xsi:nil="true"/>
  <Datatype structures:ref="ex.ExType" xsi:nil="true"/>
</DataProperty>
<DataProperty structures:id="ex.exampleAttributeProperty">
  <Name>exampleAttributeProperty</Name>
  <Namespace structures:ref="ex" xsi:nil="true"/>
  <DocumentationText>Documentation text for AttributeProperty.</DocumentationText>
  <DeprecatedIndicator>true</DeprecatedIndicator>
  <Datatype structures:ref="xs.string" xsi:nil="true"/>
  <AttributeIndicator>true</AttributeIndicator>
  <RefAttributeIndicator>true</RefAttributeIndicator>
</DataProperty>
---------------
<xs:element name="ExampleDataProperty" type="ex:ExType" substitutionGroup="ex:PropertyAbstract" appinfo:deprecated="true">
  <xs:annotation>
    <xs:documentation>Documentation text for ExampleDataProperty.</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:attribute name="exampleAttributeProperty" type="xs:string" appinfo:referenceAttributeIndicator="true">
  <xs:annotation>
    <xs:documentation>Documentation text for ExampleDataProperty.</xs:documentation>
  </xs:annotation>
</xs:attribute>
```

Example 4-31: DataProperty object in CMF and XSD

The following table shows the mapping between DataProperty representations in CMF and XSD.

CMF	XSD
Namespace	The namespace object for the containing schema document.
Name	`xs:complexType/@name`

CMF	XSD
DocumentationText	`xs:complexType/xs:annotation/xs:documentation`
DeprecatedIndicator	`xs:complexType/@appinfo:deprecated`
AbstractIndicator	`xs:complexType/@abstract`
SubPropertyOf	The property object for `xs:element/@substitutionGroup`
RelationshipPropertyIndicator	`xs:element/@appinfo:relationshipPropertyIndicator`
Datatype	The datatype object for `xs:element/@type`
AttributeIndicator	True for an attribute declaration.
RefAttributeIndicator	`xs:attribute/@appinfo:referenceAttributeIndicator`

Table 4-32: DataProperty object properties in CMF and XSD

4.9 Datatype

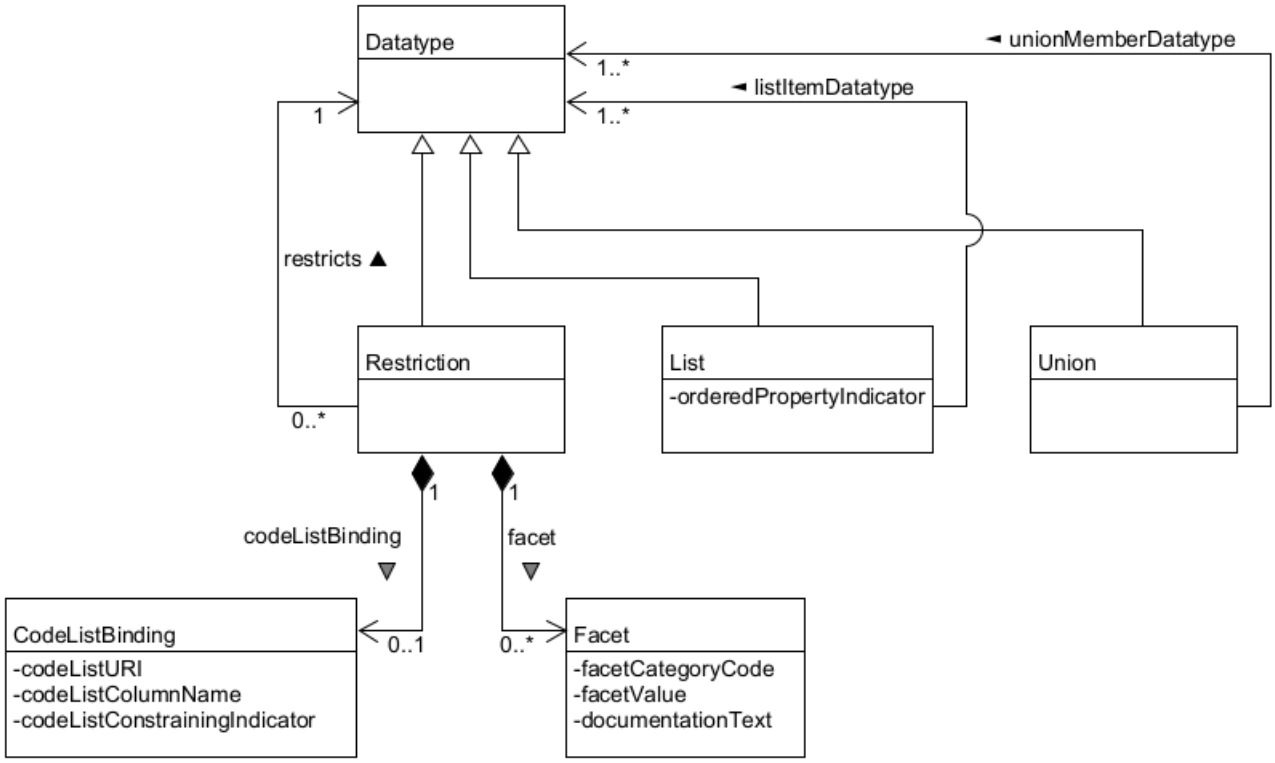


Figure 4-33: Datatype classes

An instance of the Datatype class defines the allowed values of a data property in a message. Objects for primitive data types, corresponding to the XSD data types, have only the *name*, *namespace*, and *documentation* properties inherited from the Component class. For example, example 4-34 shows the CMF representation of the `xs:string` primitive data type. All other datatypes are represented by either a Restriction, List, or Union object.

```
<Datatype>
  <Name>string</Name>
  <Namespace structures:ref="xs" xsi:nil="true"/>
</Datatype>
```

Example 4-34: Plain CMF datatype object for `xs:string`

4.10 List

An instance of the List class represents a NIEM model datatype with values that are a whitespace-separated list of literal values.

Name	Definition	Card	Ord	Range
List	A data type for a NIEM model datatype that is a whitespace-separated list of literal values.			
OrderedPropertyIndicator	True if the order of a repeated property within an object is significant.	0..1	-	xs:boolean
ListItemDatatype	The datatype of the literal values in a list.	1	-	DatatypeType

Table 4-35: Properties of the List object class

A List object is represented in XSD as a complex type definition that extends a simple type definition that has an `xs:list` element. Example 4-36 shows the CMF and XSD representation of a List object.

```
<List structures:id="ex.ExListType">
  <Name>ExListType</Name>
  <Namespace structures:ref="ex" xsi:nil="true"/>
  <DocumentationText>A data type for a list of integers.</DocumentationText>
  <ListItemDatatype structures:ref="xs.integer" xsi:nil="true"/>
  <OrderedPropertyIndicator>true</OrderedPropertyIndicator>
</List>
---------------
<xs:simpleType name="ExListSimpleType">
  <xs:list itemType="xs:integer"/>
</xs:simpleType>
<xs:complexType name="ExListType" appinfo:orderedPropertyIndicator="true">
  <xs:annotation>
    <xs:documentation>A data type for a list of integers.</xs:documentation>
  </xs:annotation>
  <xs:simpleContent>
    <xs:extension base="ex:ExListSimpleType">
      <xs:attributeGroup ref="structures:SimpleObjectAttributeGroup"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Example 4-36: List object in CMF and XSD

The following table shows the mapping between List object representations in CMF and XSD.

CMF	XSD
Namespace	The namespace object for the containing schema document.
Name	`xs:complexType/@name`
DocumentationText	`xs:complexType/xs:annotation/xs:documentation`
DeprecatedIndicator	`xs:complexType/@appinfo:deprecated`

CMF	XSD
ListItemDatatype	`xs:simpleType/xs:list/@itemType`
OrderedPropertyIndicator	`xs:complexType/@appinfo:orderedPropertyIndicator`

Table 4-37: List object properties in CMF and XSD

4.11 Union

An instance of the Union class represents a NIEM model datatype that is the union of one or more datatypes.

Name	Definition	Card	Ord	Range
Union	A data type for a NIEM model datatype that is a union of datatypes.			
UnionMemberDatatype	A NIEM model datatype that is a member of a union datatype.	1..*	-	DatatypeType

Table 4-38: Properties of the Union object class

A Union object is represented in XSD as a complex type definition that extends a simple type definition that has ar `xs:union` element. Example 4-39 shows the XSD and CMF representations of a Union object.

```
<Union structures:id="ex.UnionType">
  <Name>UnionType</Name>
  <Namespace structures:ref="test" xsi:nil="true"/>
  <DocumentationText>A data type for a union of integer and float datatypes.</DocumentationText>
  <UnionMemberDatatype structures:ref="xs.integer" xsi:nil="true"/>
  <UnionMemberDatatype structures:ref="xs.float" xsi:nil="true"/>
</Union>
---------------
<xs:simpleType name="UnionSimpleType">
  <xs:union memberTypes="xs:integer xs:float"/>
</xs:simpleType>
<xs:complexType name="UnionType">
  <xs:annotation>
    <xs:documentation>A data type for a union of integer and float datatypes.</xs:documentation>
  </xs:annotation>  <xs:simpleContent>
    <xs:extension base="ex:UnionSimpleType">
      <xs:attributeGroup ref="structures:SimpleObjectAttributeGroup"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Example 4-39: Union object in CMF and XSD

The following table shows the mapping between UnionDatatype object representations in CMF and XSD.

CMF	XSD
Namespace	The namespace object for the containing schema document.
Name	`xs:complexType/@name`
DocumentationText	`xs:complexType/xs:annotation/xs:documentation`
DeprecatedIndicator	`xs:complexType/@appinfo:deprecated`
UnionMemberDatatype	`xs:simpleType/xs:union/@memberTypes`

Table 4-40: Union object properties in CMF and XSD

4.12 Restriction

An instance of the Restriction class represents a NIEM model datatype as a base datatype plus zero or more constraining facets.

Name	Definition	Card	Ord	Range
Restriction	A data type for a restriction of a data type.			
RestrictionBase	The NIEM model datatype that is restricted by this datatype.	1	-	DatatypeType
Facet	A constraint on an aspect of a data type.	0..*	-	FacetType
CodeListBinding	A property for connecting literal values defined by a data type to a a column of a code list.	0..1	-	CodeListBindingType

Table 4-41: Properties of the Restriction object class

A Restriction object is represented in XSD as a complex type with simple content that contains an `xs:restriction` element. Example 4-42 shows the CMF and XSD representations of a Restriction object.

```
<Restriction structures:id="test.RestrictionType">
  <Name>RestrictionType</Name>
  <Namespace structures:ref="test" xsi:nil="true"/>
  <DocumentationText>Exercise code list binding</DocumentationText>
  <RestrictionBase structures:ref="xs.token" xsi:nil="true"/>
  <Facet>
    <FacetCategoryCode>enumeration</FacetCategoryCode>
    <FacetValue>GB</StringValue>
  </Facet>
  <Facet>
    <FacetCategoryCode>enumeration</FacetCategoryCode>
    <FacetValue>US</StringValue>
  </Facet>
  <CodeListBinding>
    <CodeListURI>http://api.nsgreg.nga.mil/geo-political/GENC/2/3-11</CodeListURI>
    <CodeListColumnName>foo</CodeListColumnName>
    <CodeListConstrainingIndicator>true</CodeListConstrainingIndicator>
  </CodeListBinding>
</Restriction>
---------------
<xs:complexType name="RestrictionType">
  <xs:annotation>
    <xs:appinfo>
      <clsa:SimpleCodeListBinding codeListURI="http://api.nsgreg.nga.mil/geo-political/GENC/2/3-11"
        columnName="foo" constrainingIndicator="true"/>
    </xs:appinfo>
  </xs:annotation>
  <xs:simpleContent>
    <xs:restriction base="niem-xs:token">
      <xs:enumeration value="GB"/>
      <xs:enumeration value="US"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

Example 4-42: Restriction object in CMF and XSD

The following table shows the mapping between Restriction object representations in CMF and XSD.

CMF	XSD
Namespace	The namespace object for the containing schema document.
Name	`xs:complexType/@name`

CMF	XSD
DocumentationText	`xs:complexType/xs:annotation/xs:documentation`
DeprecatedIndicator	`xs:complexType/@appinfo:deprecated`
RestrictionBase	The datatype object for `xs:complexType/xs:simpleContent/xs:restriction/@base`
Facet	`xs:complexType/xs:simpleContent/xs:restriction/` *facet-element*
CodeListBinding	`xs:complexType/xs:annotation/xs:appinfo/clsa:SimpleCodeListBinding`

Table 4-43: Restriction object properties in CMF and XSD

A code list is a set of string values, each having a known meaning beyond its value, each representing a distinct conceptual entity. These code values may be meaningful text or may be a string of alphanumeric identifiers that represent abbreviations for literals.

A code list datatype is a Restriction in which each value that is valid for the datatype corresponds to a code value in a code list.

Many code list datatypes have an XSD representation composed of `xs:enumeration` values. Code list datatypes may also be constructed using the *NIEM Code Lists Specification* [Code Lists], which supports code lists defined using a variety of methods, including CSV spreadsheets; these are represented by a CodeListBinding object, described below.

4.13 Facet

An instance of the Facet class specifies a constraint on the base datatype of a Restriction object.

Name	Definition	Card	Ord	Range
Facet	A data type for a constraint on an aspect of a data type.			
FacetCategoryCode	A kind of constraint on a restriction datatype.	1	-	FacetCategoryCodeType
FacetValue	A value of a constraint on a restriction datatype.	1	-	xs:string
DocumentationText	A human-readable documentation of a constraint on a restriction datatype.	0..*	Y	TextType

Table 4-44: Properties of the Facet object class

The range of the `FacetCategoryCode` property is a code list. The twelve codes correspond to the twelve constraining facets in XML Schema Structures; that is, the code `length` corresponds to the `xs:length` constraining facet in XSD, and constrains the valid values of the base datatype in the same way as the XSD facet.

A Facet object is represented in XSD as a constraining facet on a simple type.Example 4-45 shows the representation of two Facet objects, first in CMF, then in XSD:

```
<Facet>
  <FacetCategoryCode>minInclusive</FacetCategoryCode>
  <FacetValue>0</FacetValue>
</Facet>
<Facet>
  <FacetCategoryCode>maxExclusive</FacetCategoryCode>
  <FacetValue>360</FacetValue>
</Facet>
---------------
<xs:restriction base="niem-xs:decimal">
  <xs:minInclusive value="0"/>
  <xs:maxExclusive value="360"/>
</xs:restriction>
```


Example 4-45: Facet object in CMF and XSD

The following table shows the mapping between Facet representations in CMF and XSD:

CMF	XSD
FacetCategoryCode	*the local name of the facet element; e.g.* `minInclusive`
FacetValue	`@value`
DocumentationText	`xs:annotation/xs:documentation`

Table 4-46: Facet object properties in CMF and XSD

4.14 CodeListBinding

An instance of the CodeListBinding class establishes a relationship between a Restriction object and a code list specification. The detailed meaning of the object properties is provided in [Code Lists].

Name	Definition	Card	Ord	Range
CodeListBinding	A data type for connecting simple content defined by an XML Schema component to a a column of a code list.			
CodeListURI	A universal identifier for a code list.	1	-	xs:anyURI
CodeListColumnName	A local name for a code list column within a code list.	0..1	-	xs:string
CodeListConstrainingIndicator	True when a code list binding constrains the validity of a code list value, false otherwise.	0..1	-	xs:boolean

Table 4-47: Properties of the CodeListBinding object class

A CodeListBinding object is represented in XSD as a `clsa:SimpleCodeListBinding` element in an `xs:appinfo` element. Example 4-48 shows the representation of a CodeListBinding object, first in CMF, then in XSD.

```
<CodeListBinding>
  <CodeListURI>http://api.nsgreg.nga.mil/geo-political/GENC/2/3-11</CodeListURI>
  <CodeListConstrainingIndicator>false</CodeListConstrainingIndicator>
</CodeListBinding>
---------------
<xs:simpleType name="CountryAlpha2CodeSimpleType">
  <xs:annotation>
    <xs:documentation>A data type for country codes.</xs:documentation>
    <xs:appinfo>
      <clsa:SimpleCodeListBinding codeListURI="http://api.nsgreg.nga.mil/geo-political/GENC/2/3-11"constrainingIndicator="
    </xs:appinfo>
```

Example 4-48: CodeListBinding object in CMF and XSD

The following table shows the mapping between CodeListBinding representations in CMF and XSD.

CMF	XSD
CodeListURI	`clsa:SimpleCodeListBinding/@codeListURI`
CodeListColumnName	`clsa:SimpleCodeListBinding/@columnName`
CodeListConstrainingIndicator	`clsa:SimpleCodeListBinding/@constrainingIndicator`

Table 4-49: CodeListBinding object properties in CMF and XSD

4.15 Augmentation class

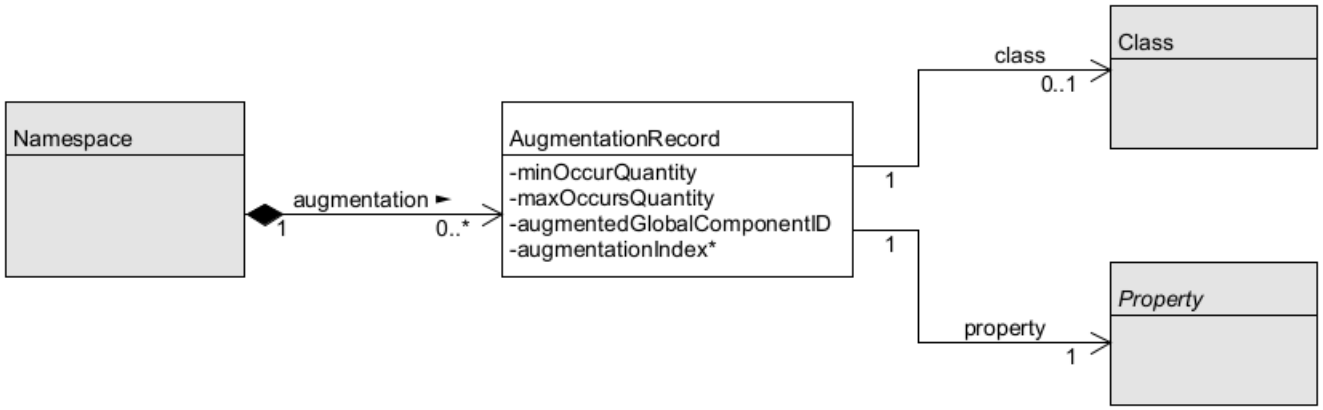


Figure 4-50: Augmentation class diagram

Augmentation is the NIEM mechanism allowing the author of one namespace (the*augmenting namespace*) to add a property to a class in another namespace (the *augmented namespace*) — without making any change to the augmented namespace. For example, the model designers for the NIEM Justice domain have augmented the `nc:PersonType` class with the `j:PersonSightedIndicator` property. Then:

- https://docs.oasis-open.org/niemopen/ns/model/domains/justice/6.0/ is the augmenting namespace
- https://docs.oasis-open.org/niemopen/ns/model/niem-core/6.0/ is the augmented namespace
- `j:PersonSightedIndicator` is an *augmentation property*
- `nc:PersonType` is an *augmented class*

The XSD representation of an augmentation is complex and is explained below. In CMF, an augmentation is represented as an AugmentationRecord object belonging to the augmenting namespace. In this way, each namespace object contains a complete list of all the augmentations it makes.

Name	Definition	Card	Ord	Range
AugmentationRecord	A data type for a class that is augmented with a property by a namespace.			
MinOccursQuantity	The minimum number of times a property may occur within an object of a class.	1	-	xs:integer
MaxOccursQuantity	The maximum number of times a property may occur within an object of a class.	1	-	MaxOccursType
AugmentationIndex	The ordinal position of an augmentation property that is part of an augmentation type.	0..1	-	xs:integer
GlobalClassCode	A code for a kind of class (object, association, or literal), such that every class in a model of that kind is augmented with a property	0..1	-	GlobalClassCodeType
Class	An augmented class.	0..1	-	ClassType
Property	An augmentation property .	1	-	PropertyType

Table 4-51: Properties of the Augmentation object class

For example, augmentation of `nc:PersonType` with `j:PersonAdultIndicator` and `j:PersonSightedIndicator` by the justice

namespace results in the following CMF for the augmenting namespace.

```
<Namespace>
  <NamespaceURI>https://docs.oasis-open.org/niemopen/ns/model/domains/justice/6.0/</NamespaceURI>
  <NamespacePrefix>j</NamespacePrefix>
  <AugmentationRecord>
    <Class structures:ref="nc.PersonType" xsi:nil="true"/>
    <Property structures:ref="j.PersonAdultIndicator" xsi:nil="true"/>
    <MinOccursQuantity>0</MinOccursQuantity>
    <MaxOccursQuantity>unbounded</MaxOccursQuantity>
    <AugmentationIndex>0</AugmentationIndex>
  </AugmentationRecord>
  <AugmentationRecord>
    <Class structures:ref="nc.PersonType" xsi:nil="true"/>
    <Property structures:ref="j.PersonSightedIndicator" xsi:nil="true"/>
    <MinOccursQuantity>0</MinOccursQuantity>
    <MaxOccursQuantity>unbounded</MaxOccursQuantity>
    <AugmentationIndex>1</AugmentationIndex>
  </AugmentationRecord>
</Namespace>
```

Example 4-52: Augmentation object in CMF

A *global augmentation* adds a property to every class of a specified kind in the model. In CMF, a global augmentation is represented by an AugmentationRecord object with a GlobalClassCode property and no Class property. For example, a global augmentation adding `my:PrivacyCode` to every every object class results in the following CMF for the augmenting namespace.

```
<Namespace>
  <NamespaceURI>http://example.com/MyNamespace/</NamespaceURI>
  <NamespacePrefix>my</NamespacePrefix>
  <AugmentationRecord>
    <Property structures:ref="my.PrivacyCode"/>
    <MinOccursQuantity>1</MinOccursQuantity>
    <MaxOccursQuantity>1</MaxOccursQuantity>
    <AugmentationIndex>0</AugmentationIndex>
    <GlobalClassCode>OBJECT</GlobalClassCode>
  </AugmentationRecord>
</Namespace>
```

Example 4-53: Global augmentation in CMF

A global AugmentationRecord object has no Class property (because it applies to every class). The range of the `GlobalClassCode` property is a code list with the following codes and meanings:

Code	Definition
OBJECT	A code for an augmentation property that applies to all object classes.
ASSOCIATION	A code for an augmentation property that applies to all association classes in the model.
LITERAL	A code for an augmentation property that applies to all datatypes and literal classes in the model. *(see §4.15.5)*

Table 4-54: GlobalClassCode code list

4.15.1 Augmentations in NIEM XSD

The XSD representation of an augmentation is complex, and varies based on two factors:

1. Whether the augmentation property is an attribute property or an element property
2. Whether the model is a message model In a message model, attribute augmentations appear in the schema documents

for both the augmenting namespace and the augmented namespace. (See section 4.15.4: *Attribute augmentations in message models*)

4.15.2 Augmenting a class with an element property in XSD

In XSD, a class with element properties is represented by a complex type definition with complex content (a “CCC type”). For example, `nc:PersonType` is represented as the following CCC type definition (some properties are omitted for simplicity):

```
<xs:complexType name="PersonType">
  <xs:annotation>
    <xs:documentation>A data type for a human being.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="structures:ObjectType">
      <xs:sequence>
        <xs:element ref="nc:PersonBirthDate" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="nc:PersonName" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="nc:PersonAugmentationPoint" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Example 4-55: Example complex type definition with complex content (CCC type)

Every CCC type contains an augmentation point element. This is an abstract element declaration in the same namespace, having the same name as the type which contains it, with the final “Type” replaced with “AugmentationPoint”. Because it is abstract, an augmentation point element cannot appear in a message; it is only a placeholder for element substitution. For example, `nc:PersonAugmentationPoint` is the augmentation point element for `nc:PersonType`.

```
<xs:element name="PersonAugmentationPoint" abstract="true">
  <xs:annotation>
    <xs:documentation>An augmentation point for PersonType</xs:documentation>
  </xs:annotation>
</xs:element>
```

Example 4-56: Example augmentation point element declaration

In the XSD representation of a model, a namespace augments a CCC type with an element property by defining an augmentation type and an augmentation element. Together these define a container element for the desired augmentation properties that is substitutable for the augmentation point element. For example, example 4-56 shows the XSD for the NIEM Justice namespace augmenting `nc:PersonType` with two properties, and example 4-57 shows an XML message with that augmentation. (The CMF corresponding to the XSD is shown in example 4-52.)

```
<xs:complexType name="PersonAugmentationType">
  <xs:complexContent>
    <xs:extension base="structures:AugmentationType">
      <xs:sequence>
        <xs:element ref="j:PersonAdultIndicator" minOccurs="0"/>
        <xs:element ref="j:PersonSightedIndicator" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="PersonAugmentation" type="j:ExampleAugmentationType" substitutionGroup="nc:PersonAugmentationPoint"/>
```

Example 4-57: Augmenting a class with an augmentation type and element in XSD

```
<nc:Person>
  <nc:PersonBirthDate>
    <nc:Date>2021-09-11</nc:Date>
  </nc:PersonBirthDate>
  <nc:PersonName>
    <nc:PersonFullName>John Doe</nc:PersonFullName>
  </nc:PersonName>
  <j:PersonAugmentation>
    <j:PersonAdultIndicator>true</j:PersonAdultIndicator>
    <j:PersonSightedIndicator>true</j:PersonSightedIndicator>
  </j:PersonAugmentation>
</nc:Person>
```

Example 4-58: Example message with an augmentation element

All of the augmentations in the XSD representation of the NIEM model use the above approach. There is an alternative approach, in which a namespace augments a CCC type without defining an augmentation type. This is done by making an element property substitutable for the augmentation point element. For example, the namespace http://example.com/Characters could augment `nc:PersonType` with a `PersonFictionalCharacterIndicator` property via the XSD in example 4-59.

```
<xs:element name="PersonFictionalCharacterIndicator" type="niem-xs:boolean"
    substitutionGroup="nc:PersonAugmentationPoint">
  <xs:annotation>
    <xs:documentation>True if a person is a character in a work of fiction.</xs:documentation>
  </xs:annotation>
</xs:element>
```

Example 4-59: Augmenting a class with an element property in XSD

```
<nc:Person>
  <nc:PersonBirthDate>
    <nc:Date>2021-09-11</nc:Date>
  </nc:PersonBirthDate>
  <nc:PersonName>
    <nc:PersonFullName>John Doe</nc:PersonFullName>
  </nc:PersonName>
  <chars:PersonFictionalCharacterIndicator>true</nc:PersonFictionalCharacterIndicator>
</nc:Person>
```

Example 4-60: Example message showing augmentation with an element property

The CMF corresponding to the XSD in example 4-59 is shown below. Since there is no augmentation type in the XSD, the AugmentationRecord object does not have an AugmentationIndex property to show the position of the augmentation property within that type.

```
<Namespace>
  <NamespaceURI>http://example.com/Characters/1.0</NamespaceURI>
  <NamespacePrefix>chars/NamespacePrefix>
  <DocumentationText>Example namespace for NDR6.</DocumentationText>
  <AugmentationRecord>
    <Class structures:ref="nc.PersonType" xsi:nil="true"/>
    <DataProperty structures:ref="chars.PersonFictionalCharacterIndicator" xsi:nil="true"/>
    <MinOccursQuantity>0</MinOccursQuantity>
    <MaxOccursQuantity>1</MaxOccursQuantity>
  </AugmentationRecord>
</Namespace>
```

Example 4-61: CMF for an element property augmentation

4.15.3 Augmenting a literal class or datatype with an element property in XSD

In the XSD representation of a model, a complex type definition with simple content (“CSC type”) can represent either a literal class or a datatype. It is not possible to directly augment either kind of CSC type with an element property, because element

properties are only possible within a CCC type. The desired effect is instead accomplished by augmenting the literal class or datatype with a reference attribute property. These are described in section 5.3.6. Note that augmenting a datatype with an attribute necessarily converts it into a literal class; see section 5.1.)

4.15.4 Augmenting a class with an attribute property in XSD

In the XSD representation of a model, a namespace augments a class with an attribute property by writing application information into the namespace schema document. For example, example 4-62 shows the XSD for the *Characters* namespace augmenting `nc:PersonType` with the attribute property `chars:genre`, and example 4-63 shows an XML message with that augmentation.

```
<xs:schema
  targetNamespace="http://example.com/Characters/1.0/"
  xmlns:myChars="http://example.com/Characters/1.0/"
  xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem-core/6.0/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  ct:conformanceTargets="https://docs.oasis-open.org/niemopen/ns/specification/NDR/6.0/#ExtensionSchemaDocument"
  version="1.0"
  xml:lang="en-US">
  <xs:annotation>
    <xs:documentation>Example Characters namespace for NDR6.</xs:documentation>
    <xs:appinfo>
      <appinfo:Augmentation class="nc:PersonType" property="myChars:genre"/>
    </xs:appinfo>
  </xs:annotation>
  <xs:attribute name="genre" type="xs:token">
    <xs:annotation>
      <xs:documentation>A name of a genre of fiction applicable to a fictional character.</xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:schema>
```

Example 4-62: Augmenting a class with an attribute property in XSD

```
<nc:Person myChars:genre="mystery">
  <nc:PersonBirthDate>
    <nc:Date>1890-10-15</nc:Date>
  </nc:PersonBirthDate>
  <nc:PersonName>
    <nc:PersonFullName>Peter Death Bredon Wimsey</nc:PersonFullName>
  </nc:PersonName>
  <chars:PersonFictionalCharacterIndicator>true</nc:PersonFictionalCharacterIndicator>
</nc:Person>
```

Example 4-63: Example message showing an attribute property augmentation

4.15.5 Global augmentations in XSD

Global augmentation with an element property is represented in XSD by creating an *augmentation element* substitutable for `structures:ObjectAugmentationPoint` or `structures:AssociationAugmentationPoint`. For example, example 4-64 shows the XSD for the *Privacy* namespace augmenting all object classes with the `priv:Restriction` element property; example 4-65 shows part of an XML message with that augmentation.

```
<xs:complexType name="ObjectAugmentationType">
  <xs:annotation>
    <xs:documentation>A data type for additional information about an object.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="structures:AugmentationType">
      <xs:sequence>
        <xs:element ref="priv:Restriction"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="ObjectAugmentation" type="priv:ObjectAugmentationType substitutionGroup="structures:ObjectAugmentationPoint">
  <xs:annotation>
    <xs:documentation>Additional information about an object.</xs:documentation>
  </xs:annotation>
</xs:element>
```

Example 4-64: Global augmentation with an element property in XSD

```
<nc:Person>
  <priv:ObjectAugmentation>
    <priv:Restriction>PII</priv:Restriction>
  </priv:ObjectAugmentation>
  <nc:PersonName>
    <nc:PersonFullName>John Doe</nc:PersonFullName>
  </nc:PersonName>
</nc:Person>
```

Example 4-65: Global augmentation with an element property in XSD

Global augmentation with an attribute property is represented in XSD by writing application information into the augmenting namespace schema document. Instead of specifying the augmented class, this appinfo provides a code from `GlobalAugmentationCodeType`. For example, example 4-66 shows the XSD for the *Privacy* namespace augmenting all object classes with the `priv:classification` attribute property.

```
<xs:schema
  targetNamespace="http://example.com/Privacy/1.0/"
  xmlns:priv="http://example.com/Privacy/1.0/"
  xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem-core/6.0/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  ct:conformanceTargets="https://docs.oasis-open.org/niemopen/ns/specification/NDR/6.0/#ExtensionSchemaDocument"
  version="1.0"
  xml:lang="en-US">
  <xs:annotation>
    <xs:documentation>Example Privacy namespace for NDR6.</xs:documentation>
    <xs:appinfo>
      <appinfo:Augmentation property="priv:classification" globalClassCode="OBJECT"/>
    </xs:appinfo>
  </xs:annotation>
</xs:schema>
```

Example 4-66: Global augmentation with an attribute property in XSD

4.15.6 Attribute augmentations in message models

The XSD representation of a message model must successfully validate all conforming messages. This means the augmented type definition has to include the augmenting attribute property. For example, the highlighted line in example 4-67 shows how the type definition of `nc:PersonType` would include the augmentation property `chars:genre`.

```
<xs:complexType name="PersonType"></code>
  <xs:annotation>
    <xs:documentation>A data type for a human being.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="structures:ObjectType">
      <xs:sequence>
        <xs:element ref="nc:PersonBirthDate" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="nc:PersonName" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="nc:PersonAugmentationPoint" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute ref="myChars:genre" appinfo:augmentingNamespace="chars"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Example 4-67: Example complex type definition with complex content (CCC type)

The `appinfo:augmentingNamespace` attribute is required; it declares that this attribute reference is an augmentation. The value of the attribute may be either the namespace prefix or URI.

4.16 LocalTerm

A local term is a word, phrase, acronym, or other string of characters that is used in the name of a namespace component, but that is not defined in OED, or that has a non-OED definition in this namespace, or has a word sense that is in some way unclear. An instance of the LocalTerm class captures the namespace author's definition of such a local term. For example, the Justice domain namespace in the NIEM model has a LocalTerm object defining the name "CLP" with documentation "Commercial Learners Permit".

Name	Definition	Card	Ord	Range
LocalTerm	A data type for the meaning of a term that may appear within the name of a model component.			
TermName	The name of the local term.	1	-	xs:token
DocumentationText	A human-readable text definition of a data model component or term, or the documentation of a namespace.	0..1	-	TextType
TermLiteralText	A meaning of a local term provided as a full, plain-text form.	0..1	-	xs:string
SourceURI	A URI that is an identifier or locator for an originating or authoritative document defining a local term.	0..*	-	xs:anyURI
SourceCitationText	A plain text citation of, reference to, or bibliographic entry for an originating or authoritative document defining a local term.	0..*	-	xs:string

Table 4-68: Properties of the LocalTerm object class

A LocalTerm object is represented in XSD by a `appinfo:LocalTerm` element within `xs:appinfo` element in the `xs:schema` element. Example 4-69 shows the representation of a LocalTerm object in CMF and XSD.

```
<LocalTerm>
  <TermName>2D</TermName>
  <TermLiteralText>Two-dimensional</TermLiteralText>
</LocalTerm>
<LocalTerm>
  <TermName>3D</TermName>
  <DocumentationText>Three-dimensional</DocumentationText>
</LocalTerm>
<LocalTerm>
  <TermName>Test</TermName>
  <DocumentationText>only for test purposes</DocumentationText>
  <SourceURI>http://example.com/1 http://example.com/2</SourceURI>
  <SourceCitationText>citation #1</SourceCitationText>
  <SourceCitationText>citation #2</SourceCitationText>
</LocalTerm>
---------------
<xs:appinfo>
  <appinfo:LocalTerm term="2D" literal="Two-dimensional"/>
  <appinfo:LocalTerm term="3D" definition="Three-dimensional"/>
  <appinfo:LocalTerm term="Test" definition="only for test purposes" sourceURIs="http://example.com/1 http://example.com/2
    <appinfo:SourceText>citation #1</appinfo:SourceText>
    <appinfo:SourceText>citation #2</appinfo:SourceText>
  </appinfo:LocalTerm>
</xs:appinfo>
```

Example 4-69: Example LocalTerm objects in CMF and XSD

The following table shows the mapping between LocalTerm object representations in CMF and XSD.

CMF	XSD
TermName	`appinfo:LocalTerm/@term`
DocumentationText	`appinfo:LocalTerm/@definition`
TermLiteralText	`appinfo:LocalTerm/@literal`
SourceURI	Each URI in the `appinfo:LocalTerm/@sourceURIs` list
SourceCitationText	`appinfo:LocalTerm/appinfo:SourceText`

Table 4-70: LocalTerm object properties in CMF and XSD

4.17 TextType

An instance of the TextType class combines a string property with a language property.

Name	Definition	Card	Ord	Range
TextType	A data type for a character string with a language code.			
TextLiteral	A literal value that is a character string.	1	-	xs:string
lang	A name of the language of a character string.	0..1	-	xs:language

Table 4-71: Properties of the TextType object class

5. Data modeling patterns

This section is informative. It explains common patterns in NIEM models and messages.

5.1 Datatypes and literal classes

A model component can be a datatype in one message model and a class in another. This occurs when a message designer creates a subset of a reused literal class, or augments a reused datatype.

Removing attribute properties from a reused literal class can turn it into a datatype. For example, `nc:NumericType` is a literal class in the NIEM model, but in a subset can become a datatype in a message model. In the NIEM model, `nc:NumericType` has one element property and one attribute property. Example 5-1 shows the class representation in CMF and XSD;example 5-2 shows an object of the class in an XML and JSON message.

```
<Class structures:id="nc.NumericType">                      | <xs:complexType name="NumericType">
  <Name>NumericType</Name>                                  |   <xs:simpleContent>
  <Namespace structures:ref="nc"                            |     <xs:extension base="niem-xs:decimal">
  <ChildPropertyAssociation>                                |       <xs:attribute ref="nc:toleranceNumeric" use="optional"/>
    <DataProperty structures:ref="nc.NumericLiteral"/>      |     </xs:extension>
    <MinOccursQuantity>1</MinOccursQuantity>                |   </xs:simpleContent>
    <MaxOccursQuantity>1</MaxOccursQuantity>                | </xs:complexType>
  </ChildPropertyAssociation>                               |
  <ChildPropertyAssociation>                                |
    <DataProperty structures:ref="nc.toleranceNumeric"/>    |
    <MinOccursQuantity>0</MinOccursQuantity>                |
    <MaxOccursQuantity>1</MaxOccursQuantity>                |
  </ChildPropertyAssociation>                               |
</Class>                                                    |
```

Example 5-1: A literal class in CMF and XSD

```
<my:Message>                                                | "my:Message": {
  <my:MaximumNumber nc:toleranceNumeric="2">7<my:MaximumNumber>     |   "my:MaximumNumber": {
</my:Message>                                               |     "nc:NumericLiteral": "7",
                                                            |     "nc:toleranceNumeric": "2"
                                                            |   }
                                                            | }
```

Example 5-2: Objects of a literal class in an XML and JSON message

If a message designer decides to reuse `nc:NumericType`, and to remove `nc:toleranceNumeric` from the class in his model subset, then `nc:NumericType` becomes a datatype in the subset.Example 5-3 shows the CMF and XSD representations of that subset; example 5-4 shows the resulting data property in an XML and JSON message.

```
<Restriction structures:id="nc.NumericType">                | <xs:complexType name="NumericType">
  <Name>NumericType</Name>                                  |   <xs:simpleContent>
  <Namespace structures:ref="nc"                            |     <xs:extension base="niem-xs:decimal"/>
  <RestrictionBase structures:ref="xs:decimal"/>            |   </xs:simpleContent>
</Restriction>                                              | </xs:complexType>
```

Example 5-3: A restriction datatype in a CMF and XSD model subset

```
<my:Message>                                                | "my:Message": {
  <my:MaximumNumber>7<my:MaximumNumber>                     |   "my:MaximumNumber": "7"
</my:Message>                                               | }
```

Example 5-4: A data property in an XML and JSON message

Going the other way, augmenting a reused datatype turns it into a literal class. For example, `nc:PersonUnionCategoryCodeType` is a datatype in the NIEM model, and `nc:PersonUnionCategoryCode` is a data property with that datatype.Example 5-5 shows the datatype representation in CMF and XSD; example 5-6 shows the data property in an XML and JSON message.

```
<Restriction structures:id="nc.PersonUnionCategoryCodeType">
  <Name>PersonUnionCategoryCodeType</Name>
  <Namespace structures:ref="nc" xsi:nil="true"/>
  <RestrictionBase structures:ref="xs.token" xsi:nil="true"/>
  <Enumeration>
    <StringValue>civil union</StringValue>
  </Enumeration>
  <Enumeration>
    <StringValue>common law</StringValue>
  </Enumeration>
  <Enumeration>
    <StringValue>domestic partnership</StringValue>
  </Enumeration>
  <Enumeration>
    <StringValue>married</StringValue>
  </Enumeration>
  <Enumeration>
    <StringValue>unknown</StringValue>
  </Enumeration>
</Restriction>
```

```
<xs:complexType name="PersonUnionCategoryCodeType">
  <xs:simpleContent>
    <xs:restriction base="niem-xs:token">
      <xs:enumeration value="civil union"/>
      <xs:enumeration value="common law"/>
      <xs:enumeration value="domestic partnership"/>
      <xs:enumeration value="married"/>
      <xs:enumeration value="unknown"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

Example 5-5: A datatype in CMF and XSD

```
<nc:Person>
  <nc:PersonUnionCategoryCode>married</nc:PersonUnionCategoryCode>
</nc:Person>
```

```
"nc:Person": {
  "nc:PersonUnionCategoryCode": "married"
}
```

Example 5-6: A data property in an XML and JSON message

A message designer might decide to augment `nc:PersonUnionCategoryCodeType` with metadata to indicate this information is sometimes privileged. Doing so turns it into a literal class in his model subset. Example 5-7 shows the CMF and XSD representations of that subset; example 5-8 shows the resulting object in an XML and JSON message.

```
<Restriction structures:id="nc.PersonUnionCategoryCodeSimple
  <Name>PersonUnionCategoryCodeSimpleType</Name>
  <Namespace structures:ref="nc"/>
  <RestrictionBase structures:ref="xs.token" xsi:nil="true"/
  <Enumeration>
    <StringValue>civil union</StringValue>
  </Enumeration>
  <Enumeration>
    <StringValue>common law</StringValue>
  </Enumeration>
  <Enumeration>
    <StringValue>domestic partnership</StringValue>
  </Enumeration>
  <Enumeration>
    <StringValue>married</StringValue>
  </Enumeration>
  <Enumeration>
    <StringValue>unknown</StringValue>
  </Enumeration>
</Restriction>
<DataProperty structures:id="nc.PersonCategoryCodeLiteral">
  <Name>PersonUnionCategoryCodeLiteral</Name>
  <Namespace structures:ref="nc"/>
  <Datatype structures:ref="nc.PersonUnionCategoryCodeSimple
</DataProperty>
<Class>
  <Name>PersonUnionCategoryCodeType</Name>
  <Namespace structures:ref="nc"/>
  <ChildPropertyAssociation>
    <DataProperty structures:ref="nc.PersonCategoryCodeLiter
    <MinOccursQuantity>1</MinOccursQuantity>
    <MaxOccursQuantity>1</MaxOccursQuantity>
  </ChildPropertyAssociation>
  <ChildPropertyAssociation>
    <DataProperty structures:ref="my.privileged"/>
    <MinOccursQuantity>0</MinOccursQuantity>
    <MaxOccursQuantity>1</MaxOccursQuantity>
    <AugmentingNamespace>my</AugmentingNamespace>
  </ChildPropertyAssociation>
</Class>
```

```
<xs:simpleType name="PersonUnionCategoryCodeSimpleType">
  <xs:restriction base="xs:token">
    <xs:enumeration value="civil union"/>
    <xs:enumeration value="common law"/>
    <xs:enumeration value="domestic partnership"/>
    <xs:enumeration value="married"/>
    <xs:enumeration value="unknown"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="PersonUnionCategoryCodeType">
  <xs:simpleContent>
    <xs:extension base="nc:PersonUnionCategoryCodeSimpleType">
      <xs:attribute ref="my:privileged"
          appinfo:augmentingNamespace="my"/>
      <xs:attributeGroup ref="structures:SimpleObjectAttributeG
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Example 5-7: A literal class in a CMF and XSD model subset

```
<nc:Person>
  <nc:PersonUnionCategoryCode
      my:privileged="true">married</nc:PersonUnionCategoryCode>
</nc:Person>
```

```
"nc:Person": {
  "nc:PersonUnionCategoryCode": {
    "nc:PersonUnionCategoryCodeLiteral": "married",
    "my:privileged": "true"
  }
}
```

Example 5-8: An object property with a code list class in an XML and JSON message

The representation of a literal class is complex when compared to the datatype. The JSON message is likewise complicated. Best practice is therefore to avoid augmenting a datatype whenever possible.

5.2 Meaning of NIEM data

The meaning of NIEM data is partly expressed through the hierarchy of nested objects in a message, and partly through the message model's definition of those objects. For example, the meaning of the two equivalent messages in example 3-2 (reproduced below) is described in table 5-9.

```
<msg:Request                                                    | {
 xmlns:nc="https://docs.oasis-open.org/niemopen/ns/model/niem | "@context": {
 xmlns:msg="http://example.com/ReqRes/1.0/">                  |   "nc": "https://docs.oasis-open.org/niemopen/ns/model/n
  <msg:RequestID>RQ001</msg:RequestID>                        |   "msg": "http://example.com/ReqRes/1.0/"
  <msg:RequestedItem>                                         | },
    <nc:ItemName>Wrench</nc:ItemName>                         | "msg:Request": {
    <nc:ItemQuantity>10</nc:ItemQuantity>                     |   "msg:RequestID" : "RQ001",
  </msg:RequestedItem>                                        |   "msg:RequestedItem": {
</msg:Request>                                                |     "nc:ItemName": Wrench",
                                                              |     "nc:ItemQuantity": 10
                                                              |    }
                                                              | }
                                                              | }
```

Message data	Description	Meaning
`<msg:Request>` *or* `"msg:Request":{...}`	The initial property is `msg:Request`. The message model defines the range of this property as the `msg:RequestType` class.	There is an object that is a request for a specified quantity of a named item.
`<msg:RequestID>` *or `"msg:RequestID":...`	The next property is `msg:RequestID`. The message model defines the range of this data property as the `xs:token` datatype.	There is a relationship between the object of `msg:RequestType` and the literal value `RQ001`.
`<msg:RequestedItem>` *or* `"msg:RequestedItem":{...}`	The next property is `msg:RequestedItem`. The message model defines the range of this object property as the `nc:ItemType` class.	There is a relationship between the object of `msg:RequestType` and the object of `nc:ItemType`.
`<nc:ItemName>` *or* `"nc:ItemName":...`	The next property is `nc:ItemName`. The message model defines the range of this data property as the `nc:TextType` datatype.	There is a relationship between the object of `nc:ItemType` and the literal value `Wrench`.
`<nc:ItemQuantity>` *or* `nc:ItemQuantity":...`	The next property is `nc:ItemQuantity`. The message model defines the range of this data property as the `nc:QuantityType` datatype.	There is a relationship between the object of `nc:ItemType` and the literal value `10`.

Table 5-9: Meaning of NIEM data

NIEM is designed so that NIEM data is a form of RDF data. For example, the message data above corresponds to the RDF shown in example 5-10

```
@prefix nc: <https://docs.oasis-open.org/niemopen/ns/model/adapters/niem-xs/6.0/> .
@prefix msg: <http://example.com/ReqRes/1.0/> .
_:n1 a msg:RequestType .
_:n1 msg:RequestID "RQ001".
_:n1 msg:RequestedItem _:n2 .
_:n2 a nc:ItemType .
_:n2 nc:ItemName "Wrench" .
_:n2 nc:ItemQuantity "10" .
```

Example 5-10: RDF interpretation of NIEM data (Turtle syntax)

That RDF data expresses a graph, illustrated by the diagram infigure 5-11.

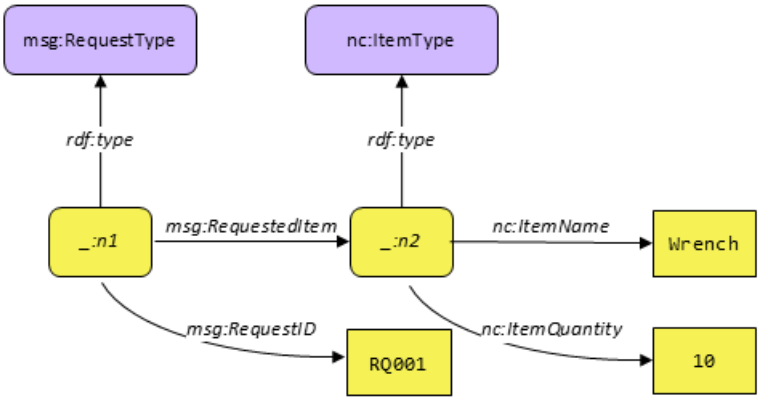


Figure 5-11: Diagram showing meaning of NIEM data

In a NIEM message, that which is not stated is not implied. If data says a person's name is John, it is not implicitly saying that he does not have other names, or that John is his legal name, or that he is different from a person known as Bob. The only assertion being made is that one of the names by which this person is known is John.

Likewise, nothing may be inferred from data that is not present in a NIEM message. It may be absent due to lack of availability, lack of knowledge, or deliberate withholding of information. These cases should be modeled explicitly, if they are required.

5.3 Identifiers and references in NIEM messages

A hierarchy of nested objects (illustrated above) is sufficient to represent simple data that takes the form of a tree. However, this simple representation has limitations, and is not capable of expressing all relationships among objects. Situations that cause problems include:

- Cycles: some object has a relationship that, when followed, eventually circles back to itself. For example, suppose that Bob has a sister relationship to Sue, who has a brother relationship back to Bob. These relationships do not form a tree, and require a data structure that is a graph, rather than a simple hierarchy of objects.
- Reuse: multiple objects have a relationship to a common object. For example, suppose Bob and Sue both have a mother relationship to Sally. Expressed via nested objects, this would result in a duplicate representation of Sally.

NIEM solves these problems through object identifiers and object references. Any object may have an identifier. An object reference can take the place of any object in a message, and is interpreted as if the object with the same identifier actually appeared in that place. The resulting data structure is a graph, not a tree.

For example, in example 5-12 below, there is only one Person object in the message; it has the identifier `JD`, and is a child property of `nc:PersonLocationAssociation`. The `nc:Person` property of the `nc:PersonOrganizationAssociation` object is an object reference. The interpretation is that the person located at the Pentagon is also the person associated with the US Army.

```
<nc:PersonLocationAssociation>                          | "nc:PersonLocationAssociation": {
  <nc:Person structures:id="JD">                        |   "nc:Person": {
    <nc:PersonName>                                     |     "@id": "#JD",
      <nc:PersonFullName>4R</nc:PersonFullName>         |     "nc:PersonName": {
    </nc:PersonName>                                    |       "nc:PersonFullName": "John Doe"
  </nc:Person>                                          |     }
  <nc:Location>                                         |   },
    <nc:LocationName>Pentagon</nc:LocationName>         |   "nc:Location": {
  </nc:Location>                                        |     "nc:LocationName": "Pentagon"
</nc:PersonLocationAssociation>                         |   }
<nc:PersonOrganizationAssociation>                      | },
  <nc:Person structures:ref="JD" xsi:nil="true"/>       | "nc:PersonOrganizationAssociation": {
  <nc:Organization>                                     |   "nc:Person": {
    <nc:OrganizationName>US Army</nc:OrganizationName>  |     "@id": "#JD"
  </nc:Organization>                                    |   },
</nc:PersonOrganizationAssociation>                     |   "nc:Organization": {
                                                        |     "nc:OrganizationName": "US Army"
                                                        |   }
                                                        | }
```

Example 5-12: Example of object references in NIEM XML and JSON

5.3.1 Object references in NIEM XML using structures:id and structures:ref

[XML] defines ID and IDREF attributes; these act as references in XML data. NIEM XML uses ID and IDREF as one way to reference data across data objects.

- `structures:id` is an ID attribute. Its value is an identifier for the object in which it appears. For example, in example 5-12 the value `JD` is an identifier for the `nc:Person` object. According to the rules of XML, an ID value must be unique within the XML document.

  An ID attribute is a *fragment identifier* within the XML document. For example, if the message as a whole has the URI http://example.com/MSG/, then the object identifier `JD` is equivalent to http://example.com/MSG/#JD.

- `structures:ref` is an IDREF attribute. An object with this attribute is a reference to the object with that identifier. For example, in example 5-12, the element `<nc:Person structures:ref="JD" xsi:nil="true"/>` is a reference to the `<nc:Person>` object that has the identifier `JD`.

  The `structures:ref` attribute has type `xs:IDREF`, so according to the rules of XML the message must contain an ID attribute with the same value. This means a `structures:ref` reference can only link to an object within the same message.

  Object references using `structures:ref` must not have content. If the object type has mandatory content, then `xsi:nil="true"` is required.

5.3.2 Object references in NIEM XML using structures:uri

NIEM introduced support for linked data through the use of uniform resource identifiers (URIs), expressed in NIEM XML through the attribute `structures:uri`. In linked data, anything modeled or addressed by an information system may be called a *resource*: people, vehicles, reports, documents, relationships, ideas: anything that is talked about and modeled in an information system is a resource. In NIEM, the objects in a message are the resources; an object identifier is a resource identifier.

The `structures:uri` attribute assigns an object identifier to the element in which it appears. All of the elements having the same identifier refer to a single object, and all of those elements provide property values for that one object. For example, in example 5-13 below, there is only one Person object in the message. The person located at the Pentagon is also the person associated with the US Army; that person is named “John Doe” and also has red hair.

```
<nc:PersonLocationAssociation>                          | "nc:PersonLocationAssociation": {
  <nc:Person structures:uri="#JD">                      |   "nc:Person": {
    <nc:PersonName>                                     |     "@id": "#JD",
      <nc:PersonFullName>John Doe</nc:PersonFullName>   |     "nc:PersonName": {
    </nc:PersonName>                                    |       "nc:PersonFullName": "John Doe"
  </nc:Person>                                          |     }
  <nc:Location>                                         |   },
    <nc:LocationName>Pentagon</nc:LocationName>         |   "nc:Location": {
  </nc:Location>                                        |     "nc:LocationName": "Pentagon"
</nc:PersonLocationAssociation>                         |   }
<nc:PersonOrganizationAssociation>                      | },
  <nc:Person structures:uri="#JD">                      | "nc:PersonOrganizationAssociation": {
    <nc:PersonHairColorText>Red</nc:PersonHairColorText>|   "nc:Person": {
  </nc:Person>                                          |     "@id": "#JD",
  <nc:Organization>                                     |     "nc:PersonHairColorText": "Red"
    <nc:OrganizationName>US Army</nc:OrganizationName>  |   },
  </nc:Organization>                                    |   "nc:Organization": {
</nc:PersonOrganizationAssociation>                     |     "nc:OrganizationName": "US Army"
                                                        |   }
                                                        | }
```

Example 5-13: Example of URI object references in NIEM XML and JSON

The `structures:uri` attribute has the type `xs:anyURI`. Values can be either a *URI-reference* or a *relative-ref*, as defined by [RFC 3986]. A URI-reference can be a URN or URL; for example:

```
  <nc:Person structures:uri="urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6"/>
  <nc:Person structures:uri="http://example.com/PersonID/B263-1655-2187"/>
```

If the message as a whole has a URI, then a relative reference is interpreted according to the rules for reference resolution in [RFC 3986]. For example, if the message URI is `http://example.com/MSG/`, then the relative reference `JD` resolves to `http://example.com/MSG/#JD`.

A relative resource in `structures:uri` is the same thing as a fragment identifier in `structures:id`, but with a leading `#` character. For example, `structures:uri="#JD"` and `structures:id="JD"` denote the same resource identifier.

5.3.3 Comparison of object references in NIEM XML

- `structures:ref` and `structures:id` must appear within the same message.
- `structures:ref` requires and provides type safety, in that the type of an object pointed to by structures:ref must be consistent with the referencing element's type declaration.
- The value of `structures:id` must be unique for IDs within the XML document.
- The value of `structures:ref` must appear within the document as the value of an attribute `structures:id`.
- A `structures:uri` can reference any `structures:id` in the same message, or in another message.
- Any `structures:uri` may reference any other `structures:uri`, in the same message, or in another message.

5.3.4 Object references in NIEM JSON using @id

Object references in NIEM JSON use JSON-LD's `@id` keyword. This is equivalent to `structures:uri` in NIEM XML. For example, the following NIEM XML and JSON references mean the same thing and are interpreted in the same way. (There is no JSON equivalent to XML's ID/IDREF attributes.)

```
<nc:Person structures:uri="#JD">
"nc:Person": { "@id": "#JD" }
```

The two JSON objects in example 5-13 that are values of a `nc:Person` key have the same `#JD` value for `@id`. That means the two JSON objects contain properties of a single NIEM message object, representing a person named "John Doe" who has

red hair.

5.3.5 Meaning of inline objects and object references

An important aspect of all of the object reference mechanisms (`structures:ref`, `structures:uri`, and `@id`) is that they all have the same meaning. There is also no difference in meaning between an object that appears inline and an object that appears through a reference.

Any claim that inline objects represent composition, while object references represent aggregation is incorrect. No life cycle dependency is implied by either method. Similarly, any claim that inline objects are intrinsic (i.e., a property inherent to an object), while object references are extrinsic (i.e., a property derived from a relationship to other things), is false. A property represented as an inline object has the exact same meaning as that property represented by a reference.

5.3.6 Reference attribute properties

A reference attribute property contains a list of object identifiers, and is interpreted as an object reference to each of the objects thus identified, each a child property of the object containing the reference attribute property. For example, the two XML messages in example 5-14 have the same meaning.

```
<my:Thing nc:metadataRef="m6 m7">                    | <my:Thing>
  <my:ThingName>The Snark</my:ThingName>             |   <my:ThingName>The Snark</my:ThingName>
  <my:ThingLocation>Dismal Valley</my:ThingLocation>  |   <my:ThingLocation>Dismal Valley</my:ThingLocation>
</my:Thing>                                          |   <nc:Metadata>
<nc:Metadata structures:id="m6">                     |     <nc:ConfidencePercent>75</nc:ConfidencePercent>
  <nc:ConfidencePercent>75</nc:ConfidencePercent>    |   </nc:Metadata>
</nc:Metadata>                                       |   <nc:Metadata>
<nc:Metadata structures:id="m7">                     |     <nc:SourceIDText>Bingo-7</nc:SourceIDText>
  <nc:SourceIDText>Bingo-7</nc:SourceIDText>         |   </nc:Metadata>
</nc:Metadata>                                       | </my:Thing>
```

Example 5-14: Reference attribute property and equivalent message in XML

Example 5-15 shows the equivalent JSON message.

```
"my:Thing": {
  "my:ThingName": "The Snark",
  "my:ThingLocation": "Dismal Valley",
  "nc:metadataRef": [
    { "@id": "#m6"},
    { "@id": "#m7"}
  ]
},
"nc:Metadata": [
  {
    "nc:ConfidencePercent": "75",
    "@id": "#m6"
  },
  {
    "nc:SourceIDText": "Bingo-7",
    "@id": "#m7"
  }
]
```

Example 5-15: Reference attribute property in JSON message

The class of these objects is determined by the name of the reference attribute property. For example, an object reference belonging to `nc:metadataRef` must have the class `nc:MetadataType`, or a derived class.*(see rule 12-11.)*

5.4 Metadata and augmentation

Metadata is data about data. The distinction is created by intended use. To the person editing an image, the creation timestamp is metadata, something he does not need. To the person writing software to sort photos into creation order, the timestamp is the data for his code. One man's metadata is another man's data.

The NIEM model contains a number of classes and properties that are suitable for metadata representations, and any model designer is free to invent new components for this purpose, as needed. A message designer may use these components in his message model, in the same way as any other component. For example, a message designer might, within the components he creates, use `nc:Metadata` to represent a source of information and the level of confidence in that information. Figure 5-16 shows an example of a message in which the designer chose to use `nc:Metadata` as a property within his own `my:ThingType` class.

```
<my:Thing>
  <my:ThingName>The Snark</my:ThingName>
  <my:ThingLocation>Dismal Valley</my:ThingLocation>
  <nc:Metadata>
    <nc:ConfidencePercent>75</nc:ConfidencePercent>
    <nc:SourceIDText>Bingo-7</nc:SourceIDText>
```

Example 5-16: Metadata properties used in a designer's own class

A message designer might also want to record source and confidence in a class reused from another namespace. This is done through augmentation, following one of two patterns. The first is to augment the class with an object property. Example 5-17 shows a message example in which `nc:PersonType` is augmented with `nc:Metadata`.

```
<nc:Person>                                          | "nc:Person": {
  <nc:PersonBirthDate>                               |   "nc:PersonBirthDate": {
    <nc:Date>2021-09-11</nc:Date>                    |     "nc:Date": "2021-09-11"
  </nc:PersonBirthDate>                              |   },
  <nc:PersonName>                                    |   "nc:PersonName": {
    <nc:PersonFullName>John Doe</nc:PersonFullName>  |     "nc:PersonFullName": "John Doe"
  </nc:PersonName>                                   |   },
  <my:PersonAugmentation>                            |   "nc:Metadata": {
    <nc:Metadata">                                   |     "nc:SourceIDText": "Tango-7"
      <nc:SourceIDText>Tango-7</nc:SourceIDText>     |   }
    </nc:Metadata>                                   | }
  </my:PersonAugmentation>                           |
</nc:Person>                                         |
```

Example 5-17: Metadata object property augmenting a reused class

The above augmentation pattern only works for a class with element properties. To add metadata properties to a literal class, the message designer must augment the class with a reference attribute property. Example 5-18 shows a message example in which `nc:PersonNameTextType` is augmented with `nc:metadataRef`.

```
<nc:Person>                                                          | "nc:Person": {
  <nc:PersonBirthDate>                                               |   "nc:PersonBirthDate": {
    <nc:Date>2021-09-11</nc:Date>                                    |     "nc:Date": "2021-09-11"
  </nc:PersonBirthDate>                                              |   },
  <nc:PersonName>                                                    |   "nc:PersonName": {
    <nc:PersonFullName nc:metadataRef="m2">John Doe</nc:PersonFullName>  |     "nc:PersonFullName": "John Doe",
  </nc:PersonName>                                                   |     "nc:metadataRef": "#m2"
</nc:Person>                                                         |   },
<nc:Metadata structures:id="m2">                                     |   "nc:Metadata": {
  <nc:ConfidencePercent>75</nc:ConfidencePercent>                    |     "@id": "#m2",
&[illegible]
```

Rule 9-74: Appinfo attribute annotates schema
c


```
&&
```

&&


```
&&
```


&&

&&

&&

&&

&&

&&


```
&&
```

```
5i55i5
```


&&

55

&&

r555r

&&

r555r

&&

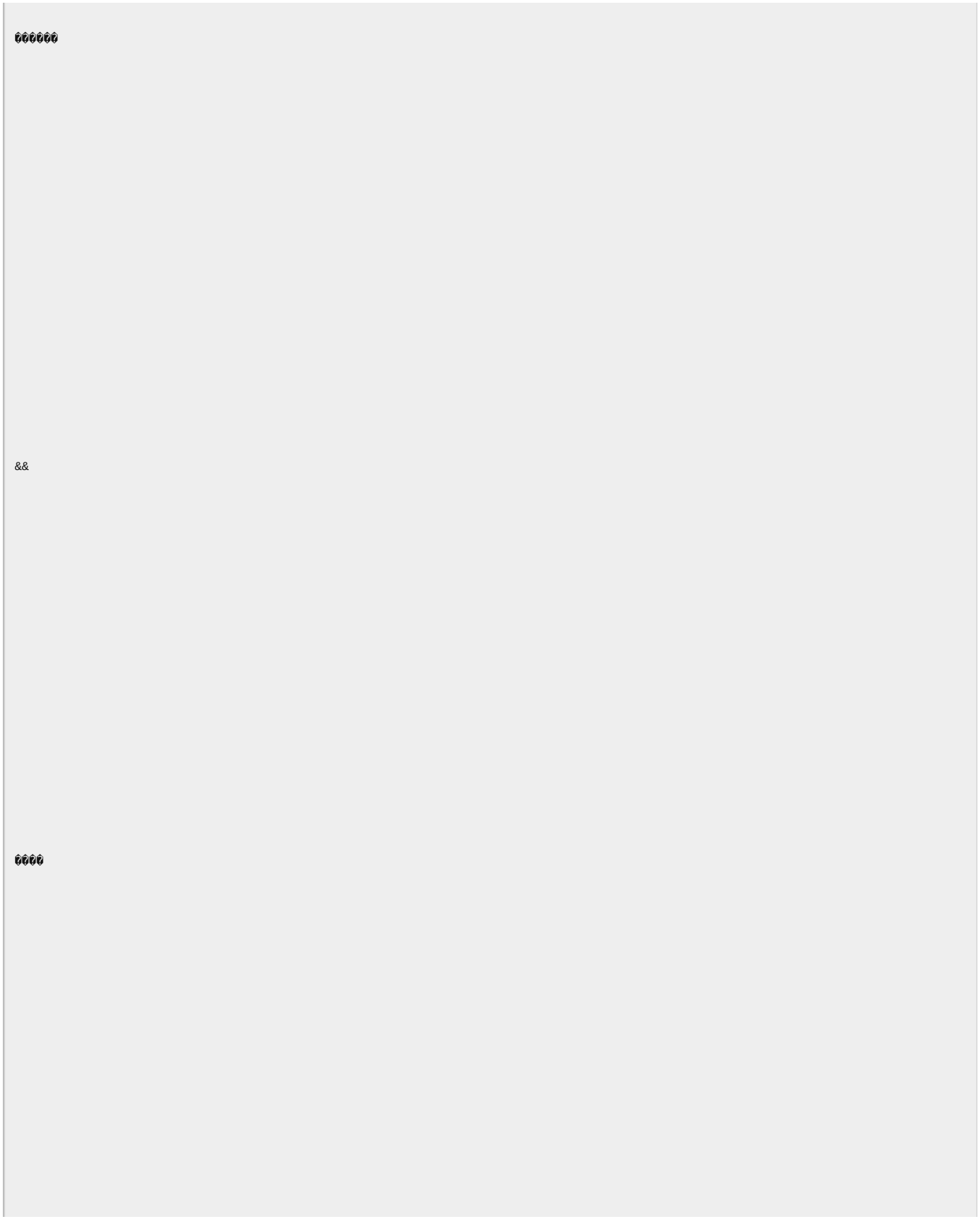
####5###r

```
&&

        
```



```
&&

``

&&
```

&&

```
&&

&&
```

&&

```
&&

&&
```

&&

```
&&
```


12-17, Attribute structures:relationshipMetadata references metadata element
no matching NIEM6 rule

Rule 12-18, Metadata is applicable to element
no matching NIEM6 rule

Appendix E. Table of examples

Appendix F. Table of figures

Appendix G. Table of tables

Appendix H. Acknowledgments

H.1 Participants

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Project-name OP Members:

First Name	Last Name	Company
Aubrey	Beach	JS J6
Brad	Bollinger	Ernst & Young
James	Cabral	Individual
Tom	Carlson	GTRI
Chuck	Chipman	GTRI
Mike	Douklias	JS J6
Katherine	Escobar	JS J6
Lavdjola	Farrington	JS J6
Dave	Hardy	JS J6
Mike	Hulme	Unisys

First Name	Last Name	Company
Dave	Kemp	NSA
Vamsi	Kondannagari	Integral Fed
Shunda	Louis	JS J6
Peter	Madruga	GTRI
Christina	Medlin	GTRI
Joe	Mierwa	Mission Critical Partners
April	Mitchell	FBI
Carl	Nelson	RISS
Scott	Renner	MITRE
Beth	Smalley	JS J6
Duncan	Sparrell	sFractal
Jennifer	Stathakis	NIST
Stephen	Sullivan	JS J6
Josh	Wilson	FBI

Appendix I. Notices

be essential to implementing this specification, and any offers of patent licensing terms, please refer to the NIEMOpen IPR Statement page.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Open Project (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. OASIS AND ITS MEMBERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THIS DOCUMENT OR ANY PART THEREOF.

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Project Specifications, OASIS Standards, or Approved Errata).

[OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Standards Final Deliverable, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Open Project that produced this deliverable.]

[OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this OASIS Standards Final Deliverable by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Open Project that produced this OASIS Standards Final Deliverable. OASIS may include such claims on its website, but disclaims any obligation to do so.]

[OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this OASIS Standards Final Deliverable or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Open Project can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Standards Final Deliverable, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any

```
claims in such list are, in fact, Essential Claims.]


The name “OASIS” is a trademark of OASIS, the owner and developer of
this specification, and should be used only to refer to the organization
and its official outputs. OASIS welcomes reference to, and
implementation and use of, specifications, while reserving the right to
enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark/
for above guidance.
```