# OGC GeoAPI 3.1/4.0

## Table of Contents

# Open Geospatial Consortium

Submission Date: <yyyy-mm-dd>

Approval Date: <yyyy-mm-dd>

Publication Date: <yyyy-mm-dd>

External identifier of this OGC® document: http://www.opengis.net/doc/is/geoapi/{m.n} (TODO: put numbers)

Internal reference number of this OGC® document: YY-nnnrx

Version: n.n

Category: OGC® Implementation Specification

Editor: Martin Desruisseaux

# OGC GeoAPI 3.1/4.0



## Warning

This document is not an OGC Standard. This document is distributed for review and comment. This document is subject to change without notice and may not be referred to as an OGC Standard.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

| | |
|---|---|
| Document type: | OGC® Standard |
| Document stage: | Draft |
| Document language: | English |

License Agreement

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD.

THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER' S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR' s sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR

standards or specifications. This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

# i. Abstract

The GeoAPI Implementation Standard defines application programming interfaces (API) in some programming languages (currently Java and Python) for geospatial applications. The API includes a set of types and methods which can be used for the manipulation of geographic information structured following the specifications adopted by the Technical Committee 211 of the International Organization for Standardization (ISO) and by the Open Geospatial Consortium (OGC). Those interfaces standardize the informatics contract between the client code, which manipulates normalized data structures of geographic information based on the published API, and the library code able both to instantiate and operate on these data structures according to the rules required by the published API and by the ISO and OGC standards.

This standard is a *realization* of OGC/ISO abstract specifications: it defines a language specific layers of normalization. By comparison, the Geographic Markup Language (GML) is another realization of the same abstract specifications (ignoring version numbers) but targeting a different language (XML). GeoAPI documents the mapping of types and methods from the abstract model into Java and Python programming languages, providing standard interfaces in the `org.opengis` or `opengis` namespaces and explaining the use of GeoAPI interfaces.

# ii. Keywords

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

ogcdoc, OGC document, GeoAPI, programming, Java, Python, interface, geospatial, metadata, referencing, feature

# iii. Preface

There is various libraries for helping developers to process geospatial data from programming languages like Java and Python. But the proliferation of API variations degrade interoperability. Since each library defines its own Application Programming Interface (API), the choice of a particular library result in a vendor lock-in situation even with open-source softwares. For example it is difficult for a Web Map Service (WMS) implementation to replace its map projection engine if all available engines use incompatible APIs. Standard API in programmatic language can reduce such vendor lock-in by providing a layer which separates client code, which call the API, from library code, which implements the API. This follows a similar pattern to the well known JDBC (in Java language) or ODBC (in C/C++ language) API which provides standardized interfaces to databases. Clients can use those APIs without concern for the particular implementation which they will use.

GeoAPI interfaces are derived from OGC/ISO conceptual models described by Unified Modeling Language (UML) diagrams. The XML schemas are generally not used (except when there is no UML diagrams describing the model) because they carry XML-specific constraints that do not apply to

programming languages. For example querying the coordinate system associated to a Coordinate Reference System (CRS) is a single method call in GeoAPI. But this single operation would have required more than 50 lines of code if the API was generated from Geographic Markup Language (GML) schema instead than from the UML diagrams of abstract models.

The interfaces described in this standard follow closely, without introducing new concepts, from the previously published standards of the Open Geospatial Consortium and the International Organization for Standardization. Nonetheless, attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

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

## iv. Submitting organizations

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

| Organization name(s) |
| --- |
| Geomatys |
| TODO |

## v. Submitters

All questions regarding this submission should be directed to the editor or the submitters:

| Name | Affiliation |
| --- | --- |
| Martin Desruisseaux | Geomatys |

# 1. Scope

GeoAPI contains a series of interfaces and classes in several packages which interpret into some programming languages (currently Java and Python) the data model and UML types of the ISO and OGC standards documents. The interfaces includes documentation which complement the injunctions of the OGC/ISO specifications by explaining particularities of GeoAPI: interpretations made of the specifications where there was room for choice, constraints specific to each programming language, or standard patterns of behavior expected by the community of a programming language. This document explains GeoAPI interfaces and defines its use by library code implementing the API and by client code calling the API. Jointly with the interface definitions, this work aims to provide:

- a carefully considered interpretation of the OGC specifications for Java and Python languages,
- a base structure to facilitate the creation of software libraries which implement OGC standards,
- a well defined, full documented binding reducing the programming effort of using the OGC abstract model,
- and to facilitate the portability of application code between different implementations.

The interfaces defined in this standard provide one way to structure the use of the Java and Python languages to implement softwares which follow the design and intents of the OGC/ISO specifications. The creators of the GeoAPI interfaces consider this approach as an effective compromise between the OGC specifications, the requirements of above-cited programming languages, and the tradition of the core libraries of those languages.

Version 3.1 and 4.0 of GeoAPI covers the base of the OGC/ISO Abstract Model for geographic information. GeoAPI provides utilities, base types, metadata structures, geo-referencing and a feature model. The geo-referencing data elements enable the creation of reference systems for spatial coordinates and mathematical operators to convert coordinates from one coordinate reference system to another. This version of the standard covers the specifications ISO 19103, ISO 19115, ISO 19111 (completed by some elements from the closely related OGC™ specification OGC 01-009), ISO 19109 and four elements from ISO 19107 necessary to the implementation of ISO 19111. Future versions of this specification are expected to expand this set of interfaces to cover more models of the OGC Abstract Specification series, including notably geometry data structures.

# 2. Conformance

This standard defines interfaces in Java and Python programming languages. The normative publication of the interfaces occurs in the ASCII or binary format specific to each target language. The interfaces are distributed in ZIP bundles along with the API documentation. An online version of the API documentation, which may contain fixes for errata discovered after publication of this specification, is available at the URLs listed below:

*Table 1. Distribution formats*

| | Java | Python |
|---|---|---|
| Interfaces: | Java Archive (JAR) binary | Python source files (.py) |
| Documentation: | Javadoc as HTML files | Sphinx generated pages as HTML files |
| Online version: | http://www.geoapi.org/snapshot/javadoc/ | http://www.geoapi.org/snapshot/python/ |

TODO: The `snapshot` elements in above URLs will be replaced by `3.1` and `4.0` after release.

Contrarily to other OGC/ISO standards, the UML diagrams in this specification are only informative. The reason is because those diagrams show only a small subset of all interfaces and properties provided by GeoAPI, and details such as naming conventions (§ 6.1.1) differ in the materialization to Java and Python languages. The complete and normative UML diagrams are provided by the abstract specifications listed in section 3.

This specification defines several conformance classes for implementations covering different packages of the API or providing different levels of complexity in their implementations. GeoAPI provides a test suite through which to establish conformance of GeoAPI implementations. Requirements for 2 standardization target types are considered:

- **Libraries** which provide software components for building geospatial applications.
- **Applications** which use the above-cited software components.

The second target type (applications) leaves more freedom than the first target type (libraries). In particular, applications are free to delete any types, methods or functionalities that they do not use. Requirements for those two target types are given in section 7.

Conformance with this standard shall be checked using all the relevant tests specified in annex A of this document. The framework, concepts, and methodology for testing, and the criteria to be achieved to claim conformance are specified in the OGC Compliance Testing Policies and Procedures

and the OGC Compliance Testing web site. In order to conform to this OGC® interface standard, a software implementation shall choose to implement:

- Any one of the conformance levels specified in annex B.
- Any one of the Distributed Computing Platform profiles specified in annex C through annex D (normative).

# 3. References

The following normative documents contain provisions that, through reference in this text, constitute provisions of this document, except for any departures from the listed specifications which are explicitly mentioned in this text. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. For undated references, the latest edition of the normative document referred to applies.

- ISO 19103:2015, Geographic information — Conceptual schema language (2015)
- ISO 19107:2003, Geographic information — Spatial schema (2003)
- ISO 19109:2015, Geographic information — Rules for application schema (2015)
- ISO 19111:2007, Geographic information — Spatial referencing by coordinates (2007)
- ISO 19115-1:2014, Geographic information — Metadata — Fundamentals (2014)
- ISO 19115-1:2014/Amd 1:2018, Metadata — Fundamentals — Amendment 1 (2018)
- ISO 19115-1:2014/Amd 1:2020, Metadata — Fundamentals — Amendment 2 (2020)
- ISO 19115-2:2019, Geographic information — Metadata — Extensions for acquisition and processing (2019)
- ISO 19141:2008, Geographic information — Schema for moving features (2008)
- ISO 19157:2013, Geographic information — Data quality (2013)
- ISO 19162, Geographic information — Well-known text representation of coordinate reference systems (2019)
- OGC 01-009, OpenGIS® Implementation Specification: Coordinate Transformation Services (2001)
- OGC 18-075, OGC® Moving Features Encoding Part I: XML Core (2019)
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Oracle: The Java Language Specification
- Python Software Foundation: The Python Language Reference

# 4. Terms and Definitions

This document uses the terms defined in Sub-clause 5.3 of [OGC 06-121r9], which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular:

- SHALL (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this standard.
- SHOULD is the verb form used to indicate desirable ability or use, without mentioning or excluding other possibilities.
- MAY is the verb form used to indicate an action permissible within the limits of this specification.
- CAN is the verb form used for statements of possibility.

For the purposes of this document, the following additional terms and definitions apply. Further discussion about *type*, *class*, *interface*, *property*, *attribute* and *implementation* terms can be found in convention section (§ 5.2).

### application programming interface (API)

a formally defined set of types and methods which establish a contract between client code which uses the API and implementation code which provides the API

### cardinality

number of elements in a set

Note 1 to entry: Contrast with *multiplicity*, which is the range of possible cardinalities a set can hold.

[source: ISO 19103]

### conceptual model

model that defines concepts of a universe of discourse

[source: ISO 19101-1]

### constraint

UML condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element

[source: ISO 19103]

### coordinate

one of a sequence of numbers designating the position of a point

Note 1 to entry: In a spatial coordinate reference system, the coordinate numbers are qualified by units.

[source: ISO 19111]

### coordinate operation

process using a mathematical model, based on a one-to-one relationship, that changes coordinates in a source coordinate reference system to coordinates in a target coordinate reference system, or that changes coordinates at a source coordinate epoch to coordinates at a target coordinate epoch within the same coordinate reference system

[source: ISO 19111]

### coordinate reference system

coordinate system that is related to an object by a datum

Note 1 to entry: Geodetic and vertical datums are referred to as reference frames.

Note 2 to entry: For geodetic and vertical reference frames, the object will be the Earth. In planetary applications, geodetic and vertical reference frames may be applied to other celestial bodies.

[source: ISO 19111]

### covariant type

a type that can be replaced by a more specialized type when the property is overridden in a subclass

Note 1 to entry: Some programming languages allow covariant return types in method declarations.

### coverage

feature that acts as a function to return values from its range for any direct position within its spatial, temporal, or spatiotemporal domain

[source: ISO 19123]

### dataset

identifiable collection of data

[source: ISO 19115-1]

### datatype

specification of a value domain with operations allowed on values in this domain

Examples: `Integer`, `Real`, `Boolean`, `String` and `Date`.

Note 1 to entry: Data types include primitive predefined types and user definable types.

[source: ISO 19103]

### dynamic attribute

characteristic of a feature in which its value varies with time

[source: OGC 16-140]

### feature

abstraction of a real world phenomena

Note 1 to entry: A feature can occur as a type or an instance. Feature type or feature instance should be used when only one is meant.

[source: ISO 19109]

### feature attribute

characteristic of a feature

Note 1 to entry: A feature attribute can occur as a type or an instance. Feature attribute type or feature attribute instance is used when only one is meant.

[source: ISO 19109]

### feature operation

operation that every instance of a feature type may perform

[source: ISO 19109]

### function

rule that associates each element from a domain (source, or domain of the function) to a unique element in another domain (target, co-domain, or range)

[source: ISO 19107:2003]

### geographic feature

representation of real world phenomenon associated with a location relative to the Earth

[source: ISO 19101-2]

### geometric object

spatial object representing a geometric set

[source: ISO 19107:2003]

### interface

UML classifier that represents a declaration of a set of coherent public UML features and obligations

Note 1 to entry: An interface specifies a contract; any classifier that realizes the interface must fulfil that contract. The obligations that can be associated with an interface are in the form of various kinds of constraints (such as pre- and post-conditions) or protocol specifications, which can impose ordering restrictions on interactions through the interface.

[source: UML 2]

### Java

trademark of Oracle used to refer to an object oriented, single inheritance programming language whose syntax derives from the C programming language and which is defined by the Java Language Specification

### literal value

constant, explicitly specified value

Note 1 to entry: This contrasts with a value that is determined by resolving a chain of substitution (e.g. a variable).

[source: ISO 19143]

### metadata

data about data

[source: ISO 19115-1]

### moving feature

feature whose location changes over time

Note 1 to entry: Its base representation uses a local origin and local coordinate vectors of a geometric object at a given reference time.

Note 2 to entry: The local origin and ordinate vectors establish an engineering coordinate reference system (ISO 19111), also called a local frame or a local Euclidean coordinate system.

### multiplicity

UML specification of the range of allowable cardinalities that a set may assume

Note 1 to entry: Contrast with *cardinality*, which is the number of elements in a set.

[source: ISO 19103]

### package

UML general purpose mechanism for organizing elements into groups

[source: ISO 19103]

### property

facet or attribute of an object referenced by a name

[source: ISO 19143]

### Python

an interpreted high-level programming language for general-purpose programming

[source: Wikipedia]

### realization

specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other representing an implementation of the latter (the client)

Note 1 to entry: Realization indicates inheritance of behaviour without inheritance of structure.

Note 2 to entry: GeoAPI and GML are two realizations of OGC/ISO abstract specifications.

[source: ISO 19103] (except note 2)

### trajectory

path of a moving point described by a one parameter set of points
[source: ISO 19141]

# 5. Conventions

This section provides details for conventions used in the document. All examples in this document illustrated by the (i) icon are informative only.

## 5.1. UML notation

Unified Modeling Language (UML) static structure diagrams appearing in this document are used as described in Subclause 5.2 of OGC Web Services Common [OGC 06-121r9].

## 5.2. Type terminology

The meaning of *type*, *class*, *interface*, *property* and *attribute* can vary depending on the programming language. This document follows the UML 2 definition of interface as a declaration of a set of coherent public operations, properties and obligations specifying a contract. UML interfaces are represented in programming languages by Java interfaces and Python abstract classes. The word *class* is generally not used in this document except in discussions specific to a programming language, in which case the word takes the meaning defined by the target language; in Java this is often (but not only) an implementation of an interface.

The word *type* is used as a generic term for *interface*, *class* (whatever it is in target programming languages), *code list*, *enumeration* or the description of a feature. Note that code lists, enumerations and feature types are not interfaces.

The word *property* (not attribute) is used for values or associations defined by an interface. This document reserves the word *attribute* for feature attributes or XML attributes.

The meaning of *implementation* depends on the context. From the perspective of OGC/ISO abstract specifications, Java interfaces or Python abstract classes are implementations of the abstract models, in the same way than XML Schema Definitions (XSD) are other implementations of the same abstract models. But from the perspective of programming languages, interfaces are not implementations; instead the word "implementation" is used for concrete classes. In this document, the types provided by GeoAPI are said to be an implementation of abstract models, while the concrete classes provided by vendors are said to be an implementations of GeoAPI.

## 5.3. Abbreviated terms

The following symbols and abbreviated terms are used in this specification:

*Table 2. Abbreviated terms*

| API | Application Program Interface |
|---|---|
| BBOX | Bounding Box |
| CRS | Coordinate Reference System |
| ISO | International Organization for Standardization |
| OGC | Open Geospatial Consortium |
| UML | Unified Modeling Language |
| URI | Uniform Resource Identifiers |
| UTC | Coordinated Universal Time |
| WKT | Well Known Text |
| XML | Extensible Markup Language |
| XSD | XML Schema Definition |
| 1D | One Dimensional |
| 2D | Two Dimensional |
| 3D | Three Dimensional |
| nD | Multi-Dimensional |
| λ | Geodetic longitude |
| ϕ | Geodetic latitude |

## 5.4. Identifiers

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

http://www.opengis.net/spec/geoapi/{m.n}/ (TODO: put numbers)

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

### 5.4.1. Package namespaces

This specification uses `opengis` in the text for denoting a package or module in OGC namespace, but the fully qualified name depends on the programming language. For example the metadata package is spelled `org.opengis.metadata` in Java but only `opengis.metadata` (without `org` prefix) in Python. Except in language-specific notes, this specification uses the shorter form in the text and lets readers adapt to their programming language of interest.

# 6. Geospatial API overview

The GeoAPI interfaces formalizes the handling of types defined in OGC/ISO abstract specifications. Whereas the specifications define types, operations and relationships using the general UML notation, the GeoAPI types implement those standards as programming language interfaces or simple classes. The structure of the GeoAPI library mirrors the packaging and separation of the different ISO and OGC specifications by grouping different types and functionality in separated Java and Python language packages.

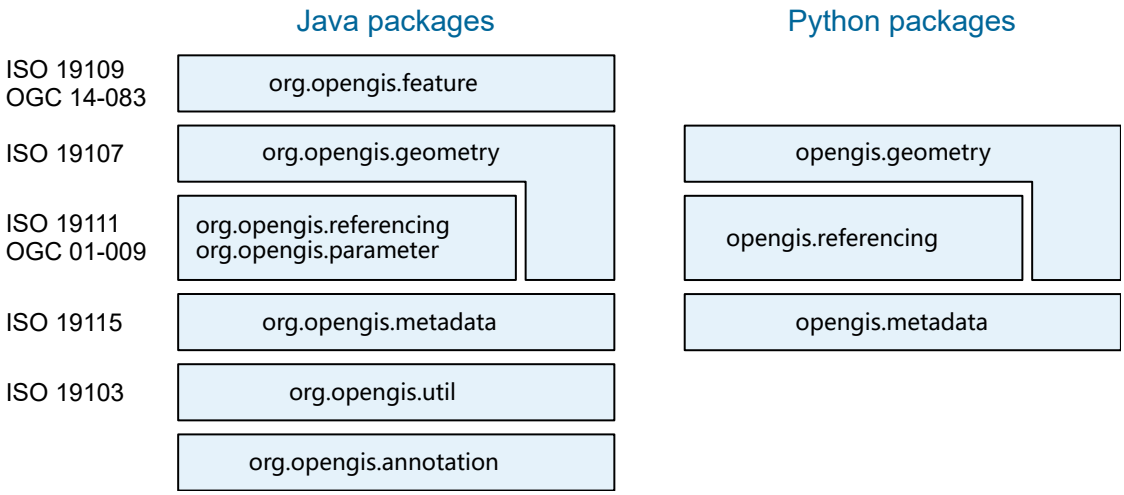


*Figure 1. OGC/ISO specifications and GeoAPI packages mapping*

The `opengis.annotation` package provides the annotation system used to document the origin and obligation level of all methods and types in GeoAPI. These annotations are available through introspection at runtime for any code which wishes to exploit this information. The `opengis.util` package provides some basic types shared by two or more standards. The packages in the `opengis.metadata` namespace cover the types defined in the ISO 19115 *Metadata* specification, which are data structures describing other data. The packages in the `opengis.parameter` and `opengis.referencing` namespaces implement the types from the ISO 19111 *Spatial Referencing by Coordinates* specification complemented by the mathematical operator types from the OGC 01-009 *Coordinate Transformation Services* implementation specification. The packages in the `opengis.geometry` namespace cover the types defined in the ISO 19107 *Spatial Schema* specification, although version 4.0 of the library only defines the elements from that specification needed by the geo-referencing types. Finally the `opengis.feature` package covers the meta-classes defined in the ISO 19109 *Rules for application schema* specification, completed by the dynamic attributes defined in OGC 18-075 *Moving Features* specification. The feature package is not needed for dynamic languages like Python.

## 6.1. General mapping rules

This section gives high-level guidance in the mapping from UML to Java and Python API applying to all GeoAPI types. Other sections after this one will focus on specific type subsets (metadata, referencing, *etc*). Those guidance are not strict rules; departures exist on a case-by-case basis when the semantic justify them.

### 6.1.1. Naming conventions

The interface and property names defined in OGC/ISO standards may be modified for compliance with the conventions in use in target programming languages. The main changes are described below:

#### Interfaces

The two-letter prefixes are dropped. For example `MD_Metadata` and `CI_Citation` interfaces are named `Metadata` and `Citation` in Java and Python. The camel cases convention (for example `CoordinateSystemAxis`) is kept unchanged for interfaces.

#### Code lists and enumerations

The two-letter prefixes are dropped in the same way than for interfaces. Then if the type name ends with the `Code` suffix, that suffix is dropped too in strongly-typed languages like Java. For example the ISO 19115 `TopicCategoryCode` code list is named `TopicCategory` in Java classes. The `Code` suffix drop is not applied to more dynamic languages like Python, because the naming convention can be a compensation for the absence of compile-time type checks.

#### Properties

The name adaptations depend on the target programming language and on the multiplicity. In Java, accessor methods start with the `get` prefix and are followed by their property name in camel cases. But in Python, no prefix is added and the camel cases convention is replaced by the snake cases convention (see the `coordinateSystem` example below). If a property allows more than one value, then the plural form of its noun may be used. The plural form hints the developers that they may need to use indexes or iterators for accessing elements.

*Table 3. Example of name adaptations in programming elements*

| Metatype | Name in OGC/ISO | Name in Java | Name in Python |
|---|---|---|---|
| Interface | `CoordinateSystemAxis` | `CoordinateSystemAxis` | `CoordinateSystemAxis` |
| Interface | `CI_Citation` | `Citation` | `Citation` |
| Code list | `CI_TelephoneTypeCode` | `TelephoneType` | `TelephoneTypeCode` |
| Enumeration | `MD_TopicCategoryCode` | `TopicCategory` | `TopicCategoryCode` |
| Property [0…1] | `coordinateSystem` | `getCoordinateSystem()` | `coordinate_system` |

| Metatype | Name in OGC/ISO | Name in Java | Name in Python |
|---|---|---|---|
| Property [0…∞] | alternateTitle | getAlternateTitles() | alternate_title |

### 6.1.2. Multiplicity conventions

The UML diagrams may specify arbitrary multiplicities (minimum and maximum number of occurrences) for each property. But GeoAPI recognizes only the following four multiplicities, materialized in the API as annotations (§ 6.1.3) and in the method signatures. If a different multiplicity is needed, then [0 … ∞] should be used with a restriction documented in the text attached to the property.

- [0 … 0] — the property can not be set (this happen sometime in subtypes).
- [0 … 1] — the property is optional or conditional.
- [1 … 1] — the property is mandatory.
- [0 … ∞] — the property can appear an arbitrary number of times, including zero or one.

Some programming languages have an `Optional` construct for differentiating the [0 … 1] and [1 … 1] cases. This construct is used where appropriate, but shall be considered only as a hint. It may appear in a mandatory property if that property was optional in the parent interface. Conversely, absence of `Optional` construct is not a guarantee that the value will never be null. Some properties fall in gray area, where they are *usually* not null but may be null in some rare situations. For example the `ellipsoid` property of a Geodetic Reference Frame is mandatory when used in the context of geographic or projected coordinate reference systems, which are by far the most common cases. Even when used in other contexts, the ellipsoid is optional but still recommended. Consequently GeoAPI does not use the `Optional` construct for the `ellipsoid` property in order to keep the most common usages simpler, but robust applications should be prepared to handle a null value. Developers should refer to the API documentation for the policy on null values.

When the multiplicity is [0 … ∞], the property type is a collection such as a list. In such case an absent property is represented by an empty collection, not a null value. When the multiplicity is [1 … ∞] the collection shall never be empty, but there is no language construct in Java and Python for enforcing that requirement.

### 6.1.3. Annotated API

The `opengis.annotation` package allows GeoAPI to document the UML elements from the various specification documents used for defining the Java and Python constructs. Those annotations encode the source document, stereotype, original name, and obligation level of the various types, properties and operations published by GeoAPI. The source document may be completed by a version number when the GeoAPI construct is based on a different edition of a normative document than the dated references listed in section 3. GeoAPI defines two annotations in the Java language (no

annotation in Python): `@UML` which is applied on types and properties (fields or methods), and `@Classifier` which can be applied only on types. Those annotations are shown in the figure below:

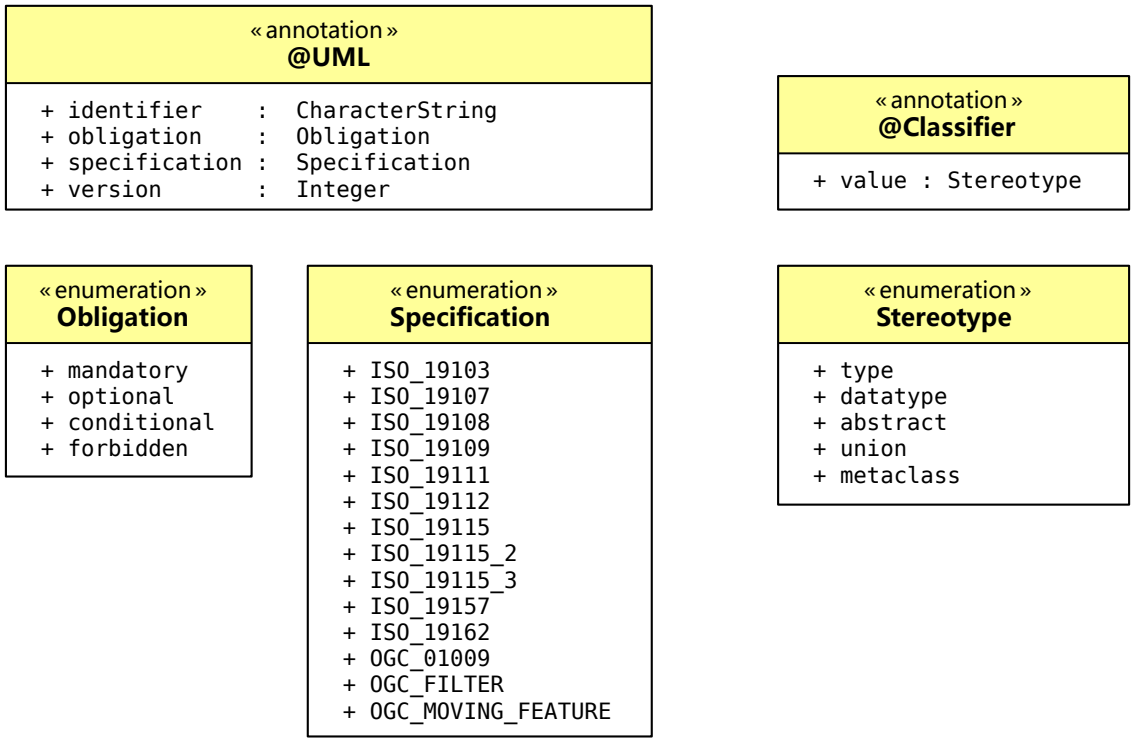


*Figure 2. Annotations reflecting UML elements used by GeoAPI*

> Those annotations are related to the ISO 19115-1 Metadata standard in the following way: the GeoAPI `Obligation` is the ISO `MD_ObligationCode` enumeration moved into the annotation package for use with other UML-related types. A `forbidden` enumeration value has been added for handling the cases where a property defined in a parent interface is inapplicable to a sub-interface ([0…0] multiplicity in abstract models). The GeoAPI `Stereotype` enumeration is a copy of the ISO `MD_DatatypeCode` code list retaining only the values relevant to annotation of GeoAPI programming elements. This duplication exists because the ISO 19115 standard defines a code list, while Java annotations require enumerations.

An example in § E.1.1 shows how these annotations are applied in the Java language and how they are available at runtime by introspection.

### 6.1.4. Derived methods

GeoAPI may define additional methods not explicitly specified in OGC/ISO abstract models, when the values returned by those methods can be derived from the values provided by standard OGC/ISO properties. Those extensions are enabled by the way properties are handled. In OGC/ISO abstract models each property could have its value stored verbatim, for example as a column in a database

table, an XML element in a file or a field in a class. For enabling efficient use of OGS/ISO models in relational databases or XML files, those models are generally non-redundant: each value is stored in exactly one property. By contrast in GeoAPI all properties are getter methods; no matter how implementations store property values, users can fetch them only through method calls. Since methods are free to compute values from other properties, GeoAPI uses this capability for making some information more easily accessible in situations where property values can be reached only indirectly in OGC/ISO models. Those additional methods introduce apparent duplications, but they should be thought as links to the real properties rather than copies of the property values. Those methods are added sparsely, in places where introducing them brings some harmonization by reducing the needs to perform special cases. Examples include fetching the head of an arbitrary `GenericName`, fetching the Geodetic Reference Frame indirectly associated to a `ProjectedCRS`, fetching axes of an arbitrary Coordinate Reference System (including compound ones), and more. Those additional methods can be recognized by the absence of `@UML` annotation.

## 6.2. Core data types mapping

The ISO 19103 specification (*Geographic Information - Conceptual schema language*) defines types which are used as building blocks by the other standards in the 19100 series. ISO 19103:2015 defines Primitive types (§7.2 of that standard), Collection types (§7.3), Enumerated types (§7.4), Name types (§7.5), Record types (§7.7) and Unit of Measure types (§C.4). GeoAPI maps these types either to existing types from the Java and Python standard libraries or, when needed, to types defined in the `opengis.util` package. That utility package is used by GeoAPI for types defined in the ISO 19103 specification for which no equivalence is already present in the Java and Python standard libraries.

For various practical reasons the mapping from ISO types to programming language types is not a one-to-one relationship. The mapping actually used is explained below. Furthermore not all of the types in ISO 19103 have a mapping defined because the need for these types has not yet appeared, since they have not yet appeared in any other specification for which GeoAPI defines interfaces. Such types are listed as "unimplemented" in the tables below.

### 6.2.1. Primitive types

*From ISO 19103:2015 §7.2.1 and 7.2.5 to 7.2.11*

Each primitive type of the OGC/ISO specifications maps to zero, one or two object structures in GeoAPI. Where the mapping can be made directly to a programming language primitive type, such as `int` and `float`, the language primitive is preferred. In languages where "primitives" and "wrappers" are distinct, wrappers may be used instead of primitives on a case-by-case basis. The following table shows the mapping used by GeoAPI to represent the primitive types in the ISO 19100 series.

*Table 4. Primitive types mapping*

| ISO 19103 interface | Java type | Python type |
|---|---|---|
| Boolean | boolean (1) | int |
| Number | java.lang.Number | int or float |
| Integer | int (1) (2) | int |
| Real | double (1) | float |
| Decimal (3) | java.math.BigDecimal | float |
| Vector | unimplemented | unimplemented |
| Sequence<Character> | java.lang.CharSequence | str |
| CharacterString | java.lang.String (4) | str |
| LanguageString (5) | org.opengis.util.InternationalString | str |
| LanguageCode (5) | java.util.Locale | |
| CharacterSetCode | java.nio.charset.Charset | |

(1) Wrapper types such as `java.lang.Integer` or `java.util.OptionalInt` may be used where appropriate.

(2) Sometime substituted by `long` or `java.lang.Long` where the value may exceed $2^{32}$.

(3) `Decimal` differs from `Real`, as `Decimal` is exact in base 10 while `Real` may not.

(4) Substituted by `org.opengis.util.InternationalString` where the string representation depends on the locale.

(5) Actually an *extension data type* defined in ISO 19103 annex C.2. See *internationalization* in § 6.2.8.

### 6.2.2. Date and time

*From ISO 19103:2015 § 7.2.2 to 7.2.4*

The ISO 19103 `Date` interface gives values for year, month and day while the `Time` interface gives values for hour, minute and second. `DateTime` is the combination of a date with a time, with or without timezone. GeoAPI maps the ISO date and time interfaces to the types provided in the standard library of target languages. In some cases like Java, this mapping forces GeoAPI to choose whether the time component shall include timezone information or not since the choices are represented by different types (e.g. `LocalDateTime`, `OffsetDateTime` and `ZonedDateTime`). The timezone information is often desired for geospatial data (for example in the acquisition time of a remote sensing image), but may be undesired for some other cases like office opening hours. In the later case, the decision to include timezone or not depends if the opening hours apply to one specific office or to all offices spanning the multiple timezones of a country. GeoAPI generally includes timezone information, but this policy may be adjusted on a case-by-case basis.

*Table 5. Date and time types mapping*

| ISO 19103 interface | Java class | Python type |
|---|---|---|
| Date | java.time.LocalDate (1) | |

| ISO 19103 interface | Java class | Python type |
|---|---|---|
| Time | java.time.OffsetTime | |
| DateTime | java.time.ZonedDateTime (1) | datetime |
| (none) | java.time.Instant (1) | |

(1) Some properties defined in GeoAPI 3.*x* use the legacy `java.util.Date` class for historical reasons.

`DateTime` is distinct from `Instant`. The former is expressed in the proleptic Gregorian calendar as described in ISO 8601, while the later is an instantaneous point on the selected time scale, astronomical or atomic. An `Instant` does not have year, month or day components. It is instead a duration elapsed since an epoch, and its conversion to a `DateTime` may be complicated. In GeoAPI, temporal objects in metadata are typically `DateTime` while coordinates in a temporal coordinate reference system are typically `Instant`.

### 6.2.3. Collections

*From ISO 19103:2015 §7.3*

GeoAPI implements ISO 19103 collection interfaces using the standard Collections Frameworks provided by Java and Python. A `Set` is a finite collection of objects where each object appears only once. A `Bag` is similar to a `Set` except that it may contain duplicated instances. The order of elements in a `Set` or a `Bag` is not specified. A `Sequence` is similar to a `Bag` except that elements are ordered.

*Table 6. Collections mapping*

| ISO 19103 interface | Java type | Python type |
|---|---|---|
| Collection | java.util.Collection | Sequence |
| Bag | java.util.Collection | |
| Set | java.util.Set | |
| Sequence | java.util.List | Sequence |
| Dictionary | java.util.Map | |

Unless otherwise required by the semantic of a property, GeoAPI preferably uses the `Collection` type in Java method signatures. This allows implementers to choose their preferred subtypes, usually `Set` or `Sequence`. The `Set` type is not the default type because enforcing element uniqueness may constraint implementations to use hash tables or similar algorithms, which is not always practical.

### 6.2.4. Controlled vocabulary

*From ISO 19103:2015 §6.5*

The feature models abstract specification (annex G) defines controlled vocabulary as an established list of standardized terminology (names, words or phrases) with associated definitions for use to identify, describe, index or retrieve information. A controlled vocabulary implementation commonly found in many programming languages is enumeration.

GeoAPI distinguishes two different types of controlled vocabularies: enumerations are *closed* controlled vocabularies: it is not possible to add new members (except by releasing new GeoAPI versions). By contrast, code lists are *open* vocabularies: they provide a basic set of members defined at compile-time, but users are free to add new members at runtime. Many programming languages provide an `enum` construct for the closed case. For the opened case, GeoAPI defines the `CodeList` abstract class in Java.

*Table 7. Enumerated types mapping*

| ISO 19103 type | Java type | Python type |
|---|---|---|
| CodeList | `org.opengis.util.CodeList` | `Enum` (1) |
| Enumeration | `java.lang.Enum` | `Enum` |
| `Bit` | unimplemented | |
| `Digit` | unimplemented | |
| `Sign` | unimplemented | |

(1) GeoAPI does not yet provide an extensible implementation of code list in the Python language, but this limitation may be addressed in a future version.

Code lists can be extended by calls to `valueOf(String)` methods in the Java language. Extensions should follow ISO 19103 recommendation: *Extensions of a code list use the existing code list values and merely add additional unique values. These additional values should not replace an existing code by changing the name or definition, or have the same definition as an existing value.*

The figure below shows one closed (on the left side) and one opened (on the right side) enumeration derived from ISO 19115. The `ControlledVocabulary` parent interface is a GeoAPI addition. Its indirect inheritance by `PixelOrientation` is because the `Enum` class is defined by programming language standard library and can not be modified directly. An example in § E.1.2 shows how code lists are used in the Java language.

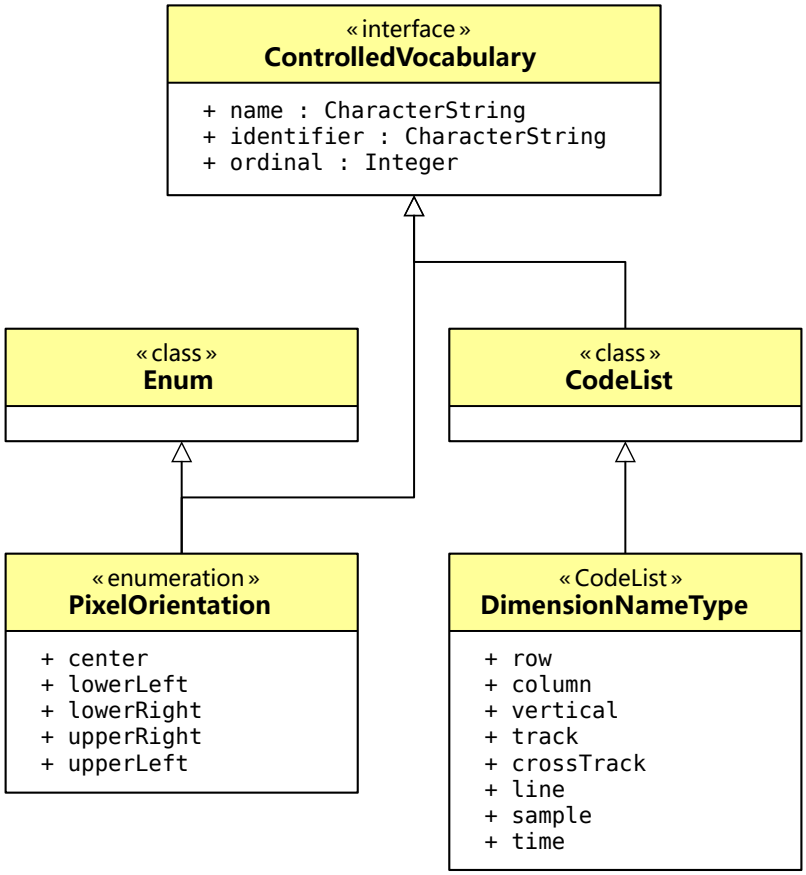


*Figure 3. Closed and opened enumeration examples*

## Departures from ISO model

### Harmonization

GeoAPI introduces a `ControlledVocabulary` interface as the parent type of enumerations and code lists. It provides methods for accessing properties common to both enumerated types: the name used in the target programming language (e.g. `"UPPER_LEFT"`), the identifier used in OGC/ISO specification (e.g. `"upperLeft"`), and its position (ordinal) in the enumeration. Note that the ordinal values may change in any future specification if new enumeration values are added, but this is not necessarily a problem because not all algorithms need stable values (e.g. hash tables).

### Renaming

In some specifications (for example ISO 19115), code list and enumeration names end with the `Code` suffix. Some other specifications (for example ISO 19111) do not use any particular suffix. The mapping to programmatic API may uniformize those type names to a single convention, depending on the target language. For the Java API, `Code` suffixes are omitted in class names. But for the Python API, class names are left unchanged. See naming conventions in § 6.1.1 for examples in both languages.

### 6.2.5. Name types

*From ISO 19103:2015 § 7.5*

A `GenericName` is a sequence of identifiers rooted within the context of a namespace. `NameSpace` defines a domain in which names can be mapped to objects. For example names could be primary keys in a database table, in which case the namespace is materialized by the table. Each storage (XML, shapefiles, netCDF, …) may have their own constraints for names in their namespaces.

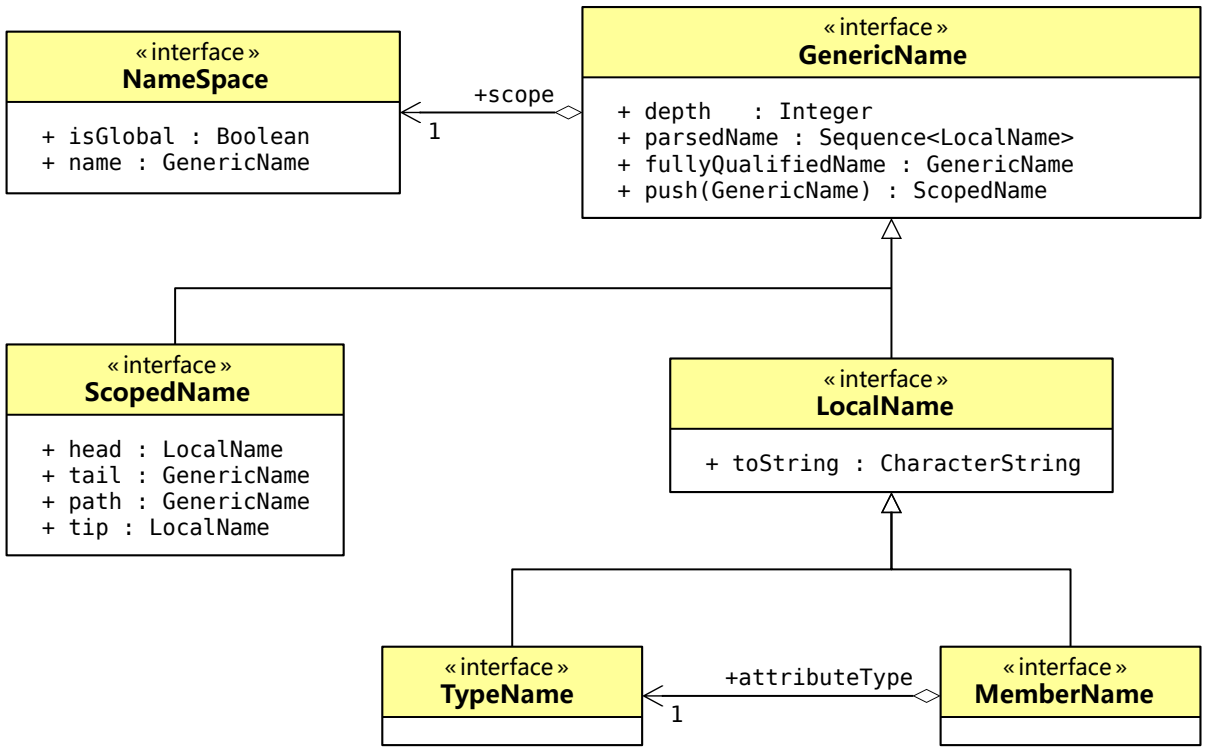


*Figure 4. Generic names derived from ISO 19103*

`GenericName` is the base interface for all names in a `NameSpace`. A generic name can be either a `LocalName`, or a `ScopedName` which is an aggregate of a `LocalName` (the *head*) for locating another `NameSpace` and a `GenericName` (the *tail*) valid in that name space. For example if `"urn:ogc:def:crs:EPSG:10:4326"` is a `ScopedName`, then `"urn"` is the *head* and `"ogc:def:crs:EPSG:10:4326"` is the *tail*. GeoAPI extends the model by allowing navigation in the opposite direction, with `"urn:ogc:def:crs:EPSG:10"` as the *path* and `"4326"` as the *tip*.

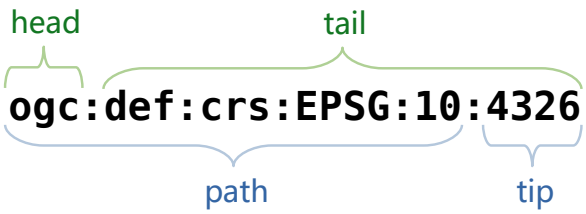


*Figure 5. Components of a generic name*

`TypeName` and `MemberName` are subtypes of `LocalName` for referencing a type (for example a class) and a member (for example a property in a class) respectively. All those types are mapped to Java and Python classes as below:

*Table 8. Name types mapping*

| ISO 19103 interface | Java interface | Python class |
|---|---|---|
| (constructors) | `org.opengis.util.NameFactory` | |
| `NameSpace` | `org.opengis.util.NameSpace` | `opengis.metadata.naming.NameSpace` |
| `GenericName` | `org.opengis.util.GenericName` | `opengis.metadata.naming.GenericName` |
| `ScopedName` | `org.opengis.util.ScopedName` | `opengis.metadata.naming.ScopedName` |
| `LocalName` | `org.opengis.util.LocalName` | `opengis.metadata.naming.LocalName` |
| `TypeName` | `org.opengis.util.TypeName` | `opengis.metadata.naming.TypeName` |
| `MemberName` | `org.opengis.util.MemberName` | `opengis.metadata.naming.MemberName` |

## Departures from ISO model

### Generalization

GeoAPI extends the ISO 19103 model by adding a (*path*, *tip*) pair in complement to the (*head*, *tail*) pair. While the *head* and *tip* properties carry non-trivial information only inside `ScopedName`, GeoAPI nevertheless makes them available from the parent `GenericName` interface (not shown in above UML diagram) with the restriction that they shall return `this` (Java) or `self` (Python) when the name is an instance of `LocalName`. This generalization makes common operations simpler without the need to check for the exact name interface.

### Renaming

The ISO 19103 `GenericName.aName` property appears as `toString` in above UML diagram, but this property should be mapped to the standard mechanism for representing an arbitrary object as a character string in the target programming language. In Java this is the `toString()` method; in Python this is `__str__` or `__repr__`. This specification uses the Java method name as it is more readable, but other languages should adapt. The `aName` and `scopedName` properties in sub-interfaces are replaced by inheritance of `toString` in above UML.

### Omissions

ISO 19103 defines mapping methods from a name to the object identified by that name: `getObject()` in `GenericName` and numerous methods in `NameSpace`. Those methods are not included in GeoAPI interfaces. Instead we left these mappings to other frameworks, for example *Java Naming and Directory Interface* (JNDI). An example in § E.1.3 shows how some omitted methods can be implemented by JNDI in the Java language.

### Additions

The `NameFactory` is an extension to allow the construction of instances of these name types (in GeoAPI, factories are realizations of constructors). The `GenericName.toFullyQualifiedName()` method is an addition for developer convenience. All GeoAPI additions on name types carry no new information compared to ISO model.

### 6.2.6. Record types

*From ISO 19103:2015 § 7.7*

Records define new data type as an heterogeneous aggregation of component data types (the fields). A `RecordType` defines dynamically constructed data type. It is identified by a `TypeName` and contains an arbitrary amount of fields as (*name*, *type*) pairs. A `Record` is an instance of `RecordType` containing the actual field values.

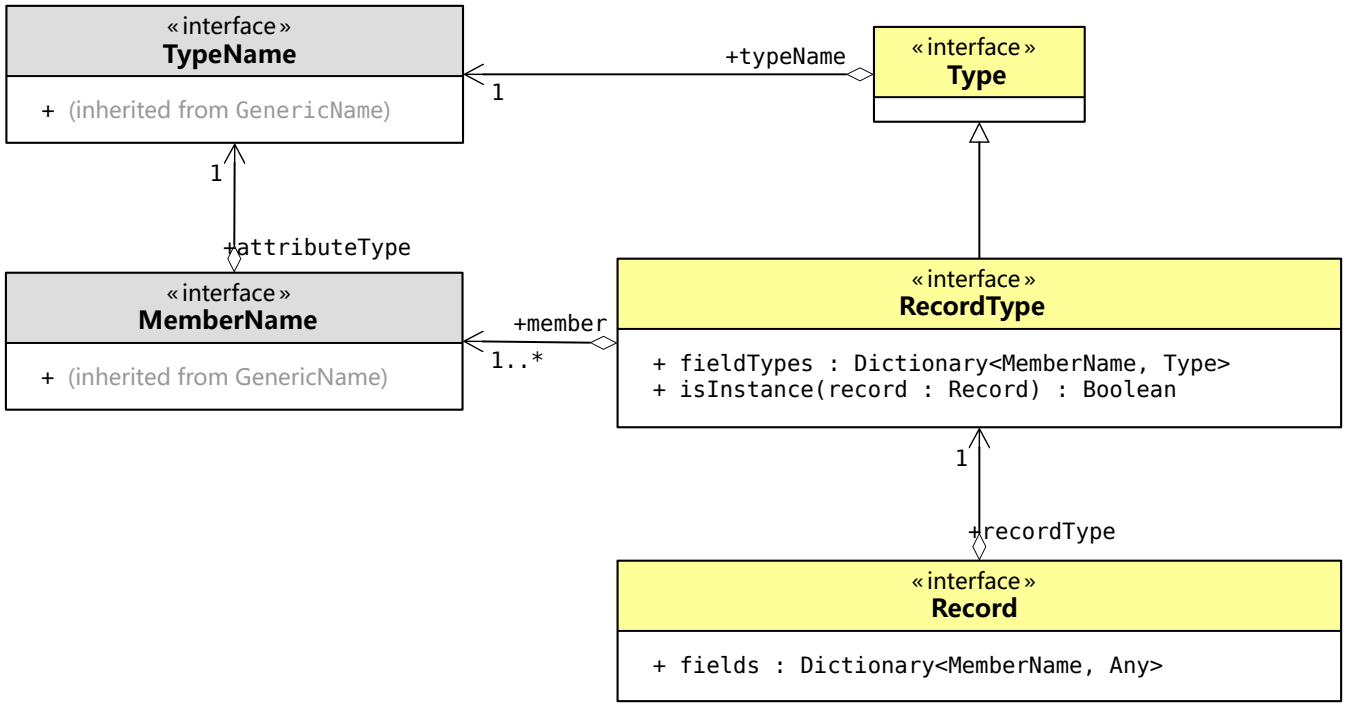


*Figure 6. Records derived and extended from ISO 19103*

`Record` and `RecordType` are lookup mechanisms that associate field names to *values* and *value types* respectively. Field names are locally mapped, and field types are most often primitives. Because the `RecordType` describes the structure of a set of records, it is essentially a metaclass for that set of records viewed as a class. In dynamic languages such as Python, the `Record` and `RecordType` interfaces are not really needed because those languages can handle dynamically constructed data types natively, but they can nevertheless be useful as marker interfaces.

*Table 9. Record types mapping*

| ISO 19103 interface | Java class or interface | Python class |
|---|---|---|
| Any | java.lang.Object | |

| ISO 19103 interface | Java class or interface | Python class |
|---|---|---|
| Type | org.opengis.util.Type | |
| Record | org.opengis.util.Record | opengis.metadata.naming.Record |
| RecordType | org.opengis.util.RecordType | opengis.metadata.naming.RecordType |
| Field | Map.Entry<MemberName, Type> | |
| FieldType | Map.Entry<MemberName, Object> | |

`Map.Entry` in above table means `java.util.Map.Entry`.

### Departures from ISO model

#### Additions

The `Type` interface and `typeName` property were defined in the older ISO 19103:2005 standard and are kept by GeoAPI despite their removal from ISO 19103:2015. The `isInstance(…)` method is a GeoAPI extension.

#### Omissions

The ISO 19103 `Field` and `FieldType` interfaces are omitted because the same functionality is achieved with `Dictionary<MemberName, Type>` and `Dictionary<MemberName, Any>` respectively. Dictionaries are realized by language-specific types such as `java.util.Map` (see collection types in § 6.2.3) which offer more flexibility. In particular dictionaries provide functionality equivalent to the ISO 19103:2005 `locate(name : MemberName)` method. That method was defined in older ISO standard but removed in ISO 19103:2015 revision. The use of dictionaries allows GeoAPI to keep the `locate` functionality despite the method removal.

## 6.2.7. Web types

*From ISO 19103:2015 § C.3*

ISO 19103 defines the following data types for use in World Wide Web environments. Those types are often found in XML documents. GeoAPI maps the ISO types to standard types of the target languages without introducing new interfaces.

*Table 10. Web types mapping*

| ISO 19103 interface | Java class or interface | Python class |
|---|---|---|
| Anchor | unimplemented | |
| FileName | java.nio.Path | |
| MediaType | unimplemented | |
| URI | java.net.URI | |

### Departures from ISO model

All ISO 19103 web types extend `CharacterString`. But it is not the case of equivalent objects provided by the standard Java library. Consequently a character string can not easily be substituted by an anchor, file name or URI in GeoAPI for Java.

## 6.2.8. Internationalization

*From ISO 19103:2015 § C.2*

The `InternationalString` interface is defined by GeoAPI to handle textual sequences which may potentially need to be translated for users of different locales. Conceptually this act as a `CharacterString` but may, depending on the implementation, provide access to locale specific representations of that string. GeoAPI `InternationalString` is closely related, but not identical, to ISO 19103 `LanguageString`. The main difference is that the later is a character string in one specific language, while `InternationalString` can be a collection of character strings in different locales. This is useful, for example, when an implementation is operating on a server that serves multiple languages simultaneously, to allow sending string representations in the locale of the client rather than the locale of the server running the GeoAPI implementation.

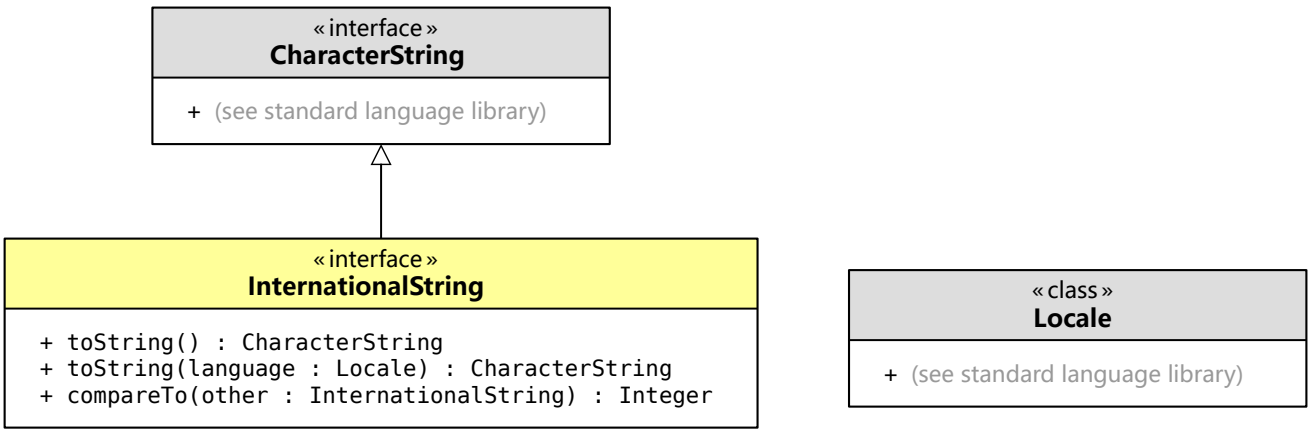


*Figure 7. Cultural and linguistic adaptability*

`InternationalString` is inspired by JSR-150 (Internationalization Service for J2EE) with support for different timezones omitted.

The `toString()` method (`__str__` in Python) returns the text in a language at implementer's choice. The `toString(Locale)` method tries to return the text in the specified language if available, and otherwise fallbacks on a language at implementer's choice. This specification makes no recommendation about the default or fallback languages because some programming languages provide their own mechanism (e.g. using *Language Range* as defined by RFC 4647 Matching of Language Tags (https://tools.ietf.org/html/rfc4647)). Consequently GeoAPI delegates most internationalization types to the target language standard library, as shown in the following table. An example in § E.1.4 shows the use of `InternationalString` in the Java language.

*Table 11. Linguistic types mapping*

| ISO 19103 interface | Java class or interface | Python class |
|---|---|---|
| CharacterString | java.lang.String (1) | str |
| LanguageString | org.opengis.util.InternationalString | str |
| LanguageCode | java.util.Locale | |
| Currency (2) | java.util.Currency | |
| CharacterSetCode (3) | java.nio.charset.Charset | |

(1) Sometime substituted by `CharSequence` or `InternationalString`.

(2) Defined as a unit of measurement in ISO 19103 §C.4.24.

(3) From ISO 19103 §7.2.10 (primitive types).

### Departures from ISO model

#### Renaming

All GeoAPI linguistic types have names and properties different than the ISO 19103 types. This is either because the ISO types are mapped to types provided by the target language standard library or because the GeoAPI type is inspired by a similar effort in the target language ecosystem.

#### Harmonization

`InternationalString` as defined by GeoAPI provides the same functionality than ISO 19115 `PT_FreeText`. It can be applied to all ISO 19115 elements who's data type is `CharacterString` and domain is "free text". Consequently GeoAPI uses the single `InternationalString` interface for the two ISO types `LanguageCode` (from ISO 19103) and `PT_FreeText` (from ISO 19115).

## 6.2.9. Units of measurement

*From ISO 19103:2015 §C.4*

ISO 19103 represents measurements and their units by two base interfaces: `Measure` for the result from performing the act of ascertaining the value of a characteristic of some entity, and `UnitOfMeasure` as a quantity adopted as a standard of measurement for other quantities of the same kind. Those two base interfaces have a parallel set of subtypes. For example `Length` as a `Measure` specialization for distances, accompanied by `UomLength` as an `UnitOfMeasure` specialization for length units. Likewise `Area` is accompanied with `UomArea`, `Time` is accompanied with `UomTime`, *etc.*

GeoAPI does not define any interface for the ISO 19103 `Measure` and `UnitOfMeasure` types because Java and Python already have their own library for units of measurement. For example Java has standardized a set of quantity interfaces in the Java Specification Request JSR-363 (TODO: upgrade to JSR-385). When such language-specific standard exists and provides equivalent functionality to ISO 19103, that external standard is used. See Java profile (§ C.1) or Java example code (§ E.1.5) for more information.

## 6.3. Metadata packages

The GeoAPI metadata packages use the `opengis.metadata` namespace and implement the types defined in the ISO 19115-1:2014 – *Metadata part 1: Fundamentals* specification along with the modifications of Amendments 1 and 2 from 2018 and 2020. They are completed or merged with the types defined in ISO 19115-2:2019 – *Metadata part 2: Extensions for acquisition and processing* specification.

The metadata packages of GeoAPI provide container types for descriptive elements which may be related to data sets or components. All of these data structures are essentially containers for strings, dates or numbers, and the interfaces consist almost exclusively of methods which provide read access to those types or a container. The API defines no methods which manipulate or modify the data structures. Implementers are free to provide a fully mutable implementation of GeoAPI interfaces, but users may need to cast to the implementation classes in order to modify a metadata.

There is almost 500 properties in more than 150 metadata interfaces. This specification does not describe them; see ISO 19115 together with GeoAPI Javadoc or Python-doc (section 2) for a description of each property. Implementers can support a subset of their choice. The following UML example shows a few frequently used interfaces with some of their properties. An example in § E.1.6 shows how to extract the individual name from a `Citation` in the Java language.

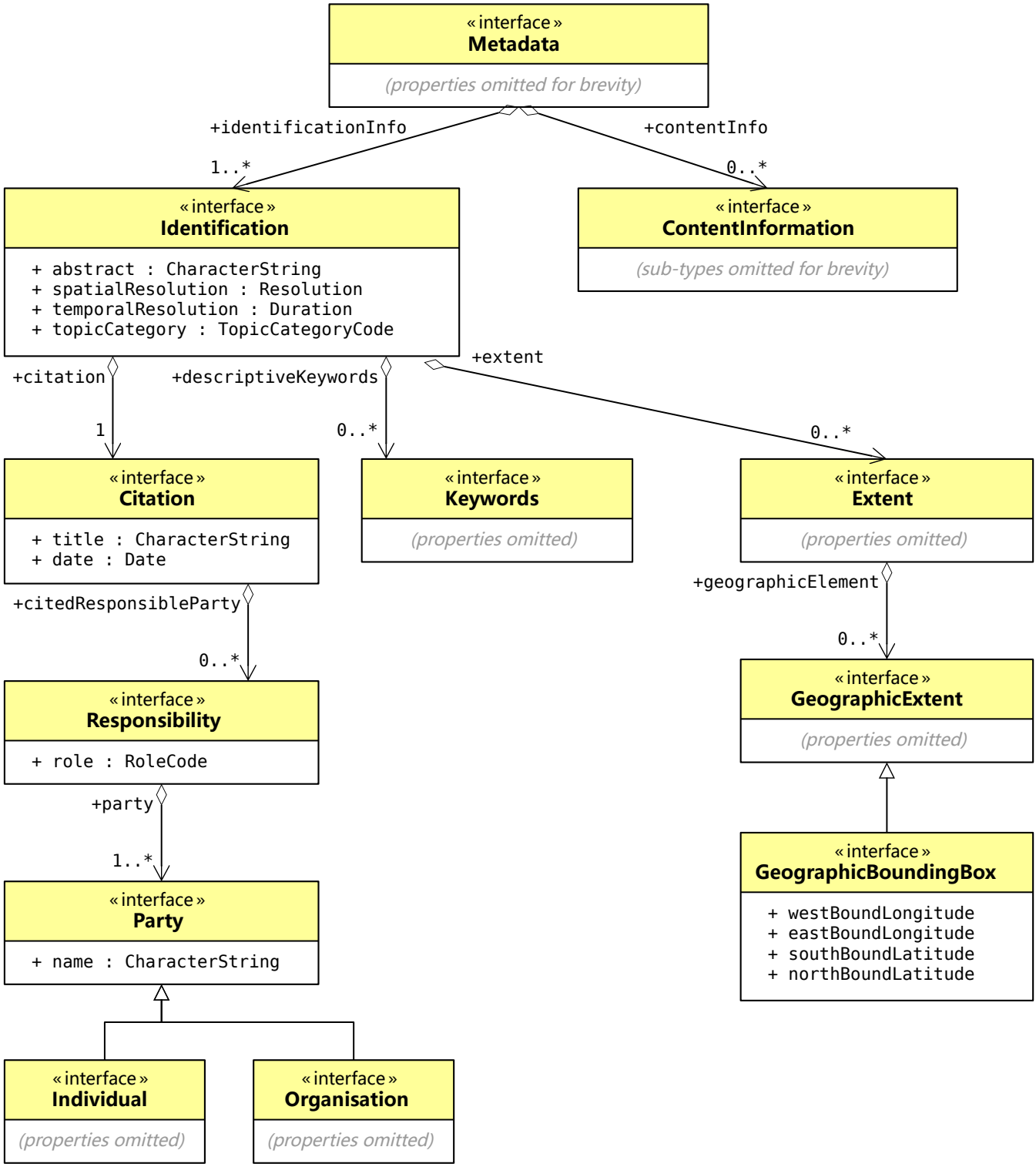


*Figure 8. Example of a few metadata interfaces with a subset of their properties*

### 6.3.1. Package mapping

*From ISO 19115-1:2014 §6.5.[2…14] and §6.6; ISO 19157:2013*

The mapping of ISO packages to GeoAPI packages follows a parallel naming scheme, shown in the table below. Some minor packages (for example *Portrayal catalogue* which contains only one interface) have been aggregated into another package. All packages are defined by ISO 19115

except *Data quality* which is defined by ISO 19157 but considered by GeoAPI as metadata for historical reasons, and *Reference system* which has been retrofitted in the ISO 19111 interfaces from the referencing package.

*Table 12. Metadata package mapping*

| ISO package | Java package | Python module |
|---|---|---|
| Metadata | `org.opengis.metadata` | `opengis.metadata.base` |
| Identification | `org.opengis.metadata.identification` | `opengis.metadata.identification` |
| Constraint | `org.opengis.metadata.constraint` | `opengis.metadata.constraints` |
| Lineage | `org.opengis.metadata.lineage` | `opengis.metadata.lineage` |
| Maintenance | `org.opengis.metadata.maintenance` | `opengis.metadata.maintenance` |
| Spatial representation | `org.opengis.metadata.spatial` | `opengis.metadata.representation` |
| Reference system | `org.opengis.referencing` | `opengis.referencing.crs` |
| Content | `org.opengis.metadata.content` | `opengis.metadata.content` |
| Portrayal catalogue | `org.opengis.metadata` | `opengis.metadata.base` |
| Distribution | `org.opengis.metadata.distribution` | `opengis.metadata.distribution` |
| Metadata extension | `org.opengis.metadata` | `opengis.metadata.extension` |
| Application schema | `org.opengis.metadata` | `opengis.metadata.extension` |
| Service metadata | `org.opengis.metadata.identification` | `opengis.metadata.identification` |
| Extent | `org.opengis.metadata.extent` | `opengis.metadata.extent` |
| Citation and party | `org.opengis.metadata.citation` | `opengis.metadata.citation` |
| Data quality | `org.opengis.metadata.quality` | `opengis.metadata.quality` |

### 6.3.2. Reference systems

*Derived from ISO 19115-1:2014 §6.5.8*

The way GeoAPI handles reference systems in metadata differs significantly from ISO 19115. Coordinate Reference Systems (CRS) are defined in details by the ISO 19111 standard. But the ISO 19115 metadata standards do not reference those CRS interfaces directly (except in one case), at the cost of some overlaps. By contrast GeoAPI does not enforce such separation between standards when a bidirectional dependency can bring harmonization. A bidirectional dependency does not imply that implementers have to support both ISO standards.

Reference Systems interfaces are defined in GeoAPI `referencing` packages. A Reference System may be a Coordinate Reference System (ISO 19111) or may use geographic identifiers (ISO 19112). GeoAPI supports both cases by defining `ReferenceSystem` as the common parent of *Coordinate Reference System* and *Reference System using Geographic Identifier*. This is done by inserting the ISO 19115 `ReferenceSystem` interface between `IdentifiedObject` and `CoordinateReferenceSystem` in ISO 19111 type hierarchy as shown below (note: this diagram does not show all types, properties and associations for brevity reasons):

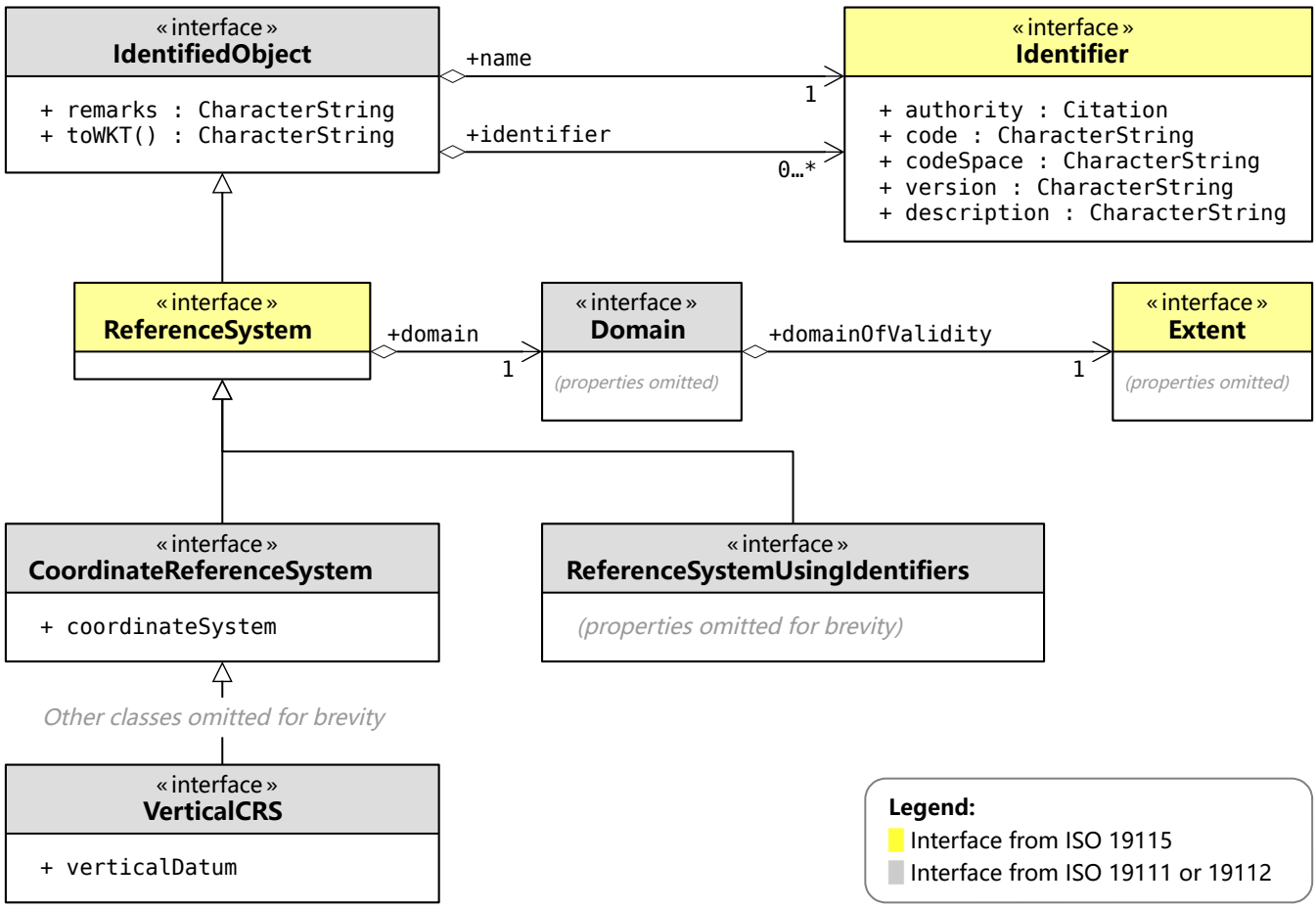


*Figure 9. Location of `ReferenceSystem` in the hierarchy of ISO 19111 types*

More information about the `CoordinateReferenceSystem` interface is given in the *Referencing packages* section (§ 6.5). The following table lists all association to a reference system from the metadata packages:

*Table 13. Associations from a metadata object to a reference system*

| Metadata interface | Property name | Property type |
|---|---|---|
| Metadata | referenceSystemInfo | ReferenceSystem |
| Source | sourceReferenceSystem | ReferenceSystem |
| VerticalExtent | verticalCRS | VerticalCRS |
| GCPCollection | coordinateReferenceSystem | ReferenceSystem |

### 6.3.3. Nil values

ISO 19115-3 – *XML schema implementation for fundamental concepts* allows property values to be nil even when the property is declared mandatory by ISO 19115-1 standard. In such case, ISO 19115-3 requires to specify why the property is nil. Nil reasons can be:

- **Template:** the metadata is only a template with values to be provided later.
- **Withheld:** the value is not divulged.
- **Unknown:** a correct value probably exists but is not known and not computable.
- **Missing:** the correct value is not readily available and may not exist.
- **Inapplicable:** there is no value.

GeoAPI does not provide a mechanism for specifying the reason why a property is nil. Instead, the GeoAPI rules of method return values have been relaxed for the metadata packages. Elsewhere in GeoAPI, methods which have a mandatory obligation in the specification must return an instance of the return type and cannot return the Java `null` or Python `None` reference. However, in the metadata package this rule is relaxed because data sets are encountered so frequently which have nil values for any of above-cited reasons. In the GeoAPI metadata packages, methods for mandatory properties should return a valid instance, but users should be prepared to receive `null` (Java), `None` (Python) or an empty collection. This modification has been adopted to allow implementations sufficient latitude about how to handle metadata records with nil values. Nonetheless, sophisticated implementations can determine if a metadata record conforms with the ISO 19115-1 specification by inspecting the annotations at runtime (§ E.1.1).

### 6.3.4. Departures from ISO 19115

#### Harmonization with ISO 19115-2

A departure in the GeoAPI metadata packages from the published ISO 19115 standard is in the way GeoAPI metadata package added the types and properties defined in the specification ISO 19115-2 – *Extensions for acquisition and processing*. The latter was forced to create a number of interfaces to hold elements which naturally could occur directly in the interfaces defined by ISO 19115-1. We integrated such interfaces directly into the existing interfaces rather than adding complexity to the API which exists by historical accident. For example ISO 19115-2 defines a `MI_Band` interface which extends the `MD_Band` interface defined by ISO 19115-1, with the addition of a `transferFunctionType` property (among others) for completing the `scaleFactor` and `offset` properties defined by ISO 19115-1. GeoAPI merges those two interfaces together, with annotations (§ 6.1.3) on each property for declaring the originating standard. The metadata interfaces merged in such way are:

*Table 14. Metadata ISO 19115-2 interfaces merged with ISO 19115-1 parent interfaces*

| ISO 19115-1 parent interface | ISO 19115-2 subclass merged with parent |
|---|---|
| LI_ProcessStep | LE_ProcessStep |
| LI_Source | LE_Source |

| ISO 19115-1 parent interface | ISO 19115-2 subclass merged with parent |
|---|---|
| MD_Band | MI_Band |
| MD_CoverageDescription | MI_CoverageDescription |
| MD_Georectified | MI_Georectified |
| MD_Georeferenceable | MI_Georeferenceable |
| MD_ImageDescription | MI_ImageDescription |
| MD_Metadata | MI_Metadata |

### Harmonization with ISO 19111

Coordinate Reference Systems (CRS) are defined in details by the ISO 19111 standard. But the ISO 19115 metadata standards do not reference those CRS interfaces directly (except `VerticalCRS`). Instead the metadata standards reference CRS by their identifier (for example an EPSG code), optionally accompanied by a code telling whether the CRS type is geographic, projected, temporal, a compound of the above, a geographic identifier, *etc.* The ISO 19115 standard combines those two information in a `ReferenceSystem` interface.

In order to have a more uniform way to handle reference systems, GeoAPI replaces (*identifier*, *type code*) tuples by associations to the actual *Reference System* objects. The ISO 19115 `ReferenceSystem` type is redefined as a subtype of ISO 19111 `IdentifiedObject` (see UML in § 6.3.2). The `referenceSystemIdentifier` property defined by ISO 19115 is replaced by inheritance of `identifier` property from `IdentifiedObject`. The `referenceSystemType` property value (geographic, projected, compound, geographic identifier, *etc.*) can be determined using language-specific instructions such as `instanceof` in Java.

In the same spirit than above replacement, `verticalCRSId` is omitted because redundant with `verticalCRS`.

## 6.4. Geometry packages

The GeoAPI geometry packages use the `opengis.geometry` namespace and are placeholder for types defined in the ISO 19107:2003 – *Spatial schema* specification. TODO: upgrade to ISO 19107:2019. The geometry specification provides a vector based spatial representation of elements. The geometry types defined in GeoAPI include only the two simplest types from ISO specification along with their abstract parent interfaces. Those types are defined because they are needed by the referencing package. Other types are expected to be added in future GeoAPI versions.

*Table 15. Mapping of types from the geometry package*

| ISO 19107 interface | Java type | Python type |
|---|---|---|
| GM_Position | org.opengis.geometry.coordinate.Position | |
| DirectPosition | org.opengis.geometry.DirectPostion | |

| ISO 19107 interface | Java type | Python type |
|---|---|---|
| GM_Envelope | org.opengis.geometry.Envelope | |

The `DirectPosition` type represents a single location in the coordinate space defined by a `CoordinateReferenceSystem`. The `Envelope` type represents the lower and upper extreme values along each axis. This type is frequently conflated with a bounding rectilinear box but the two types differ conceptually. The `Envelope` type in ISO 19107 provides methods to obtain the "corners" of the envelope as `DirectPosition` instances. However, users should note that these positions might not have any meaning in physical space. For example the corners could be outside the Coordinate Reference System (CRS) domain of validity even if the feature itself is fully inside that domain. The corner `DirectPosition` instances are acting, for convenience, as data containers for tuples of coordinates but not as representations of an actual position.

### 6.4.1. Departures from ISO 19107

#### Convenience

GeoAPI has moved the `DirectPosition` and `Envelope` types from the coordinate sub-package where they are defined in the ISO 19107 specification up to the `opengis.geometry` package due to their importance and frequency of use.

#### Union replacement

ISO 19107 defines `Position` as the union of `DirectPosition` and `Point`. But unions are not allowed in Java. Instead, GeoAPI defines `Position` as the base interface of both types so a similar functionality can be achieved with an *"is instance of"* check.

#### Efficiency

GeoAPI adds the following shortcut methods in the `Envelope` interface. They are frequently requested information that implementers can often provide in a more efficient way than forcing users to compute them from lower and upper corners: `dimension`, `minimum`, `maximum`, `median` and `span`.

## 6.5. Referencing packages

The GeoAPI referencing packages use the `opengis.referencing` and `opengis.parameter` namespaces and implement the types defined in the ISO 19111:2007 - *Referencing by coordinates* specification. TODO: upgrade to ISO 19111:2019. The referencing package also includes the types describing object factories and mathematical transformation operators defined in the legacy

standard OGC 01-009 – *Coordinate Transformation Services* from 2003. Those types can be used to define various datums, define various coordinate systems, and combine those to define the coordinate referencing systems (CRS) generally encountered in geospatial science. The UML below (incomplete for brevity) shows a few commonly used GeoAPI types. Differences compared to ISO 19111 (notably inheritance from `ReferenceSystem` type, location of `coordinateSystem` association and inclusion of `MathTransform` interface) are discussed in following sections and in the departures section (§ 6.5.6).

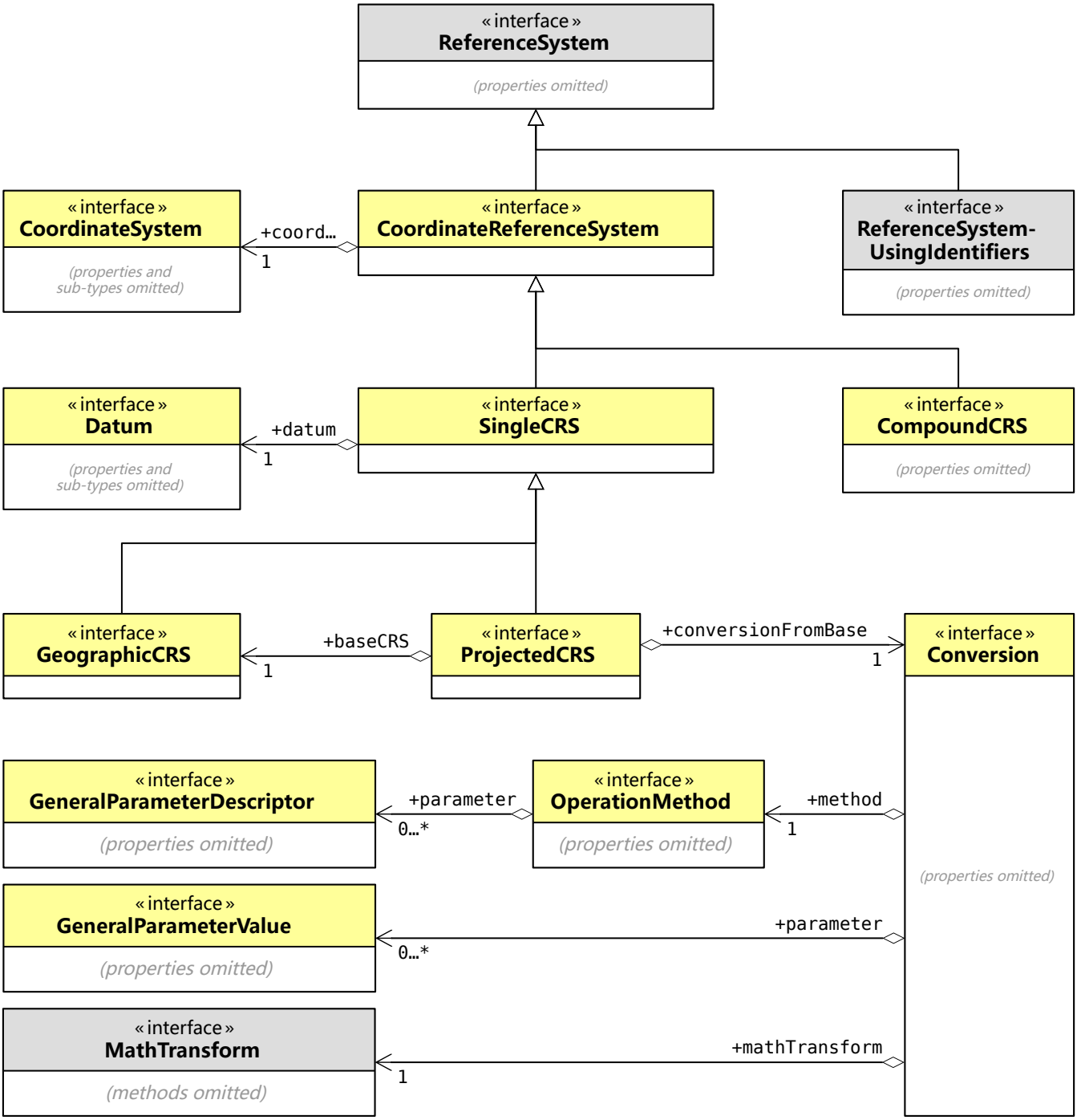


*Figure 10. Subset of interfaces derived from ISO 19111 and other standards*

More information about the `ReferenceSystem` parent interface is given in the *Metadata packages* section (§ 6.3.2). The mapping from ISO 19111 packages to GeoAPI packages is shown below:

*Table 16. Referencing package mapping*

| ISO package | Java package | Python module |
|---|---|---|
| Identified Object | `org.opengis.referencing` | `opengis.referencing.datum` |
| Reference System | `org.opengis.referencing` | `opengis.referencing.crs` |
| Coordinate Reference System | `org.opengis.referencing.crs` | `opengis.referencing.crs` |
| Coordinate System | `org.opengis.referencing.cs` | `opengis.referencing.cs` |
| Datum | `org.opengis.referencing.datum` | `opengis.referencing.datum` |
| Coordinate Operation | `org.opengis.referencing.operation` | `opengis.referencing.operation` |
| Coordinate Operation | `org.opengis.referencing.parameter` | `opengis.parameter` |

### 6.5.1. Coordinate systems

*From ISO 19111:2019 § 10 and § C.3*

A Coordinate System (CS) contains the set of axes that spans a given coordinate space. Each axis defines an approximate direction (north, south, east, west, up, down, port, starboard, past, future, *etc.*), units of measurement, minimal and maximal values, and what happen after reaching those extremum. For example in longitude case, after +180° the coordinate values continue at −180°.

In addition to axis definitions, another important coordinate system characteristic is their type (`CartesianCS`, `SphericalCS`, *etc.*). The CS type implies the set of mathematical rules for calculating geometric quantities such as angles, distances and surfaces. Usually the various CS subtypes do not define any new programmatic methods compared to the parent type, but are nevertheless important for type safety. For example many calculations or associations are legal only when all axes are perpendicular to each other. In such case the coordinate system type is restricted to `CartesianCS` in method signatures.

#### Associations with CRS

The coordinate system associated to a `SingleCRS` shall be an instance of one `CoordinateSystem` subtype. It should be a subtype defined by ISO 19111 when possible, or a user-defined subtype when no standard type describes the space geometry. The standard coordinate system types are Cartesian, affine, spherical, cylindrical, polar, ellipsoidal, vertical, temporal, ordinal and parametric.

The coordinate system associated to a `CompoundCRS` is of unspecified type. None of the standard types describe the full space geometry, and implementers do not need to define a public type. The `CoordinateSystem` in this context is only a list of axes defined as the concatenation of the

list of axes of all components, without saying anything about space geometry. Requesting elements in that list can be thought as redirection to the coordinate system of a CRS component.

*Example*

An implementation may know that the third coordinate system axis of a `CompoundCRS` is the first axis of the second CRS component, and redirects axis information requests to that component. In this scenario the `CoordinateSystem` implementations does the work that users would need to do themselves if they wanted information about axes, but in a more convenient and potentially more efficient way.

Since above paragraphs define coordinate system for both `SingleCRS` and `CompoundCRS` interfaces, GeoAPI provides the `coordinateSystem` association in the `CoordinateReferenceSystem` (CRS) parent interface. It makes easy for users to get information about the coordinate system axes of any CRS, with automatic redirection to `CompoundCRS` components when needed.

### 6.5.2. Factories

*From OGC 01-009 §12.3.[6…7] and §12.4.6*

The referencing packages include factory types defined originally in the OGC 01-009 specification. These factories define a normalized approach to object instantiation and, if used exclusively, simplify the work of switching between implementations. Subtypes of `ObjectFactory` instantiate objects by assembling components passed as arguments and subtypes of `AuthorityFactory` instantiate objects based on identifiers in some third party database, notably those in the EPSG geodetic dataset.

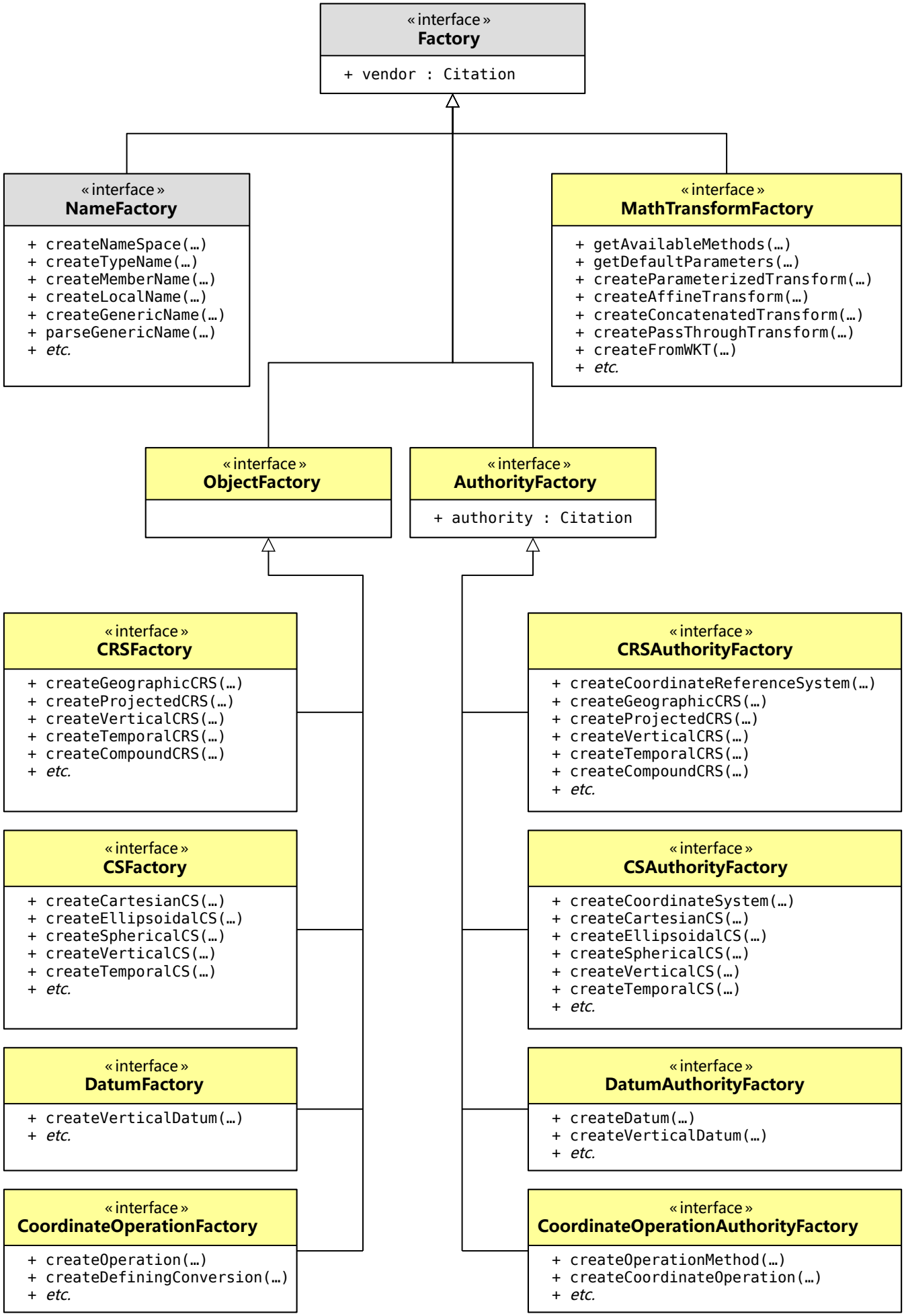


*Figure 11. Factory interfaces partially derived from OGC 01-009*

TODO: Some methods of `CRSAuthorityFactory` and `CoordinateOperationAuthorityFactory` to be replaced by ISO 19111:2019 `RegisterOperations`.

Code that needs to instantiate one of the objects defined in GeoAPI referencing packages should first obtain the factory in some platform dependent manner and then use the factory methods to instantiate the desired object instances. These instances can then be used through the interfaces defined in the GeoAPI library.

In Java, factory implementations are discovered by `java.util.ServiceLoader`. In Python, GeoAPI does not have a recommended mechanism yet.

### Axis order

The order of coordinate system axes in every objects obtained from an `AuthorityFactory` shall be as defined by the authority. The order depends on the Coordinate Reference System (CRS) type and the country defining the CRS. In the case of geographic CRS, this is often – but not always – the (*latitude*, *longitude*) axis order. In particular the following method call:

```java
GeographicCRS WGS84 = crsAuthorityFactory.createGeographicCRS("EPSG::4326");
```

shall return a coordinate reference system with (*latitude*, *longitude*) axis order, not (*longitude*, *latitude*).

## 6.5.3. Coordinate operations

*From ISO 19111:2019 §12 and §C.5*

A Coordinate Operation can *transform* or *convert* coordinate tuples from one Coordinate Reference System (CRS) to another CRS. There is four kinds of coordinate operations in GeoAPI:

- A *coordinate conversion* is the implementation of some mathematical formulas without empirically derived parameters. Conversions can be as accurate as floating point computations allow. Map projections are kinds of coordinate conversions.
- A *coordinate transformation* involves empirically derived parameters. Because those parameters have observational error and because transformation methods are only approximations of a complex reality, the coordinate transformation results also have errors. Furthermore several transformations may exist for the same pair of coordinate reference systems, differing in their method, parameter values, domain of validity and accuracy characteristics.
- TODO: Point motion operation.
- A *concatenated operation* defines a sequential execution of any of above operations.

GeoAPI provides three ways to create a coordinate operation:

### From a pair of CRS

Given a source coordinate reference system (CRS) in which existing coordinate values are expressed, and a target coordinate reference system in which coordinate values are desired, `CoordinateOperationFactory` can provide a coordinate operation performing the conversion or transformation work. An example is given in § E.1.8 for the Java language.

Note that when several possibilities exist for the same pair of CRS, the selected transformation is implementation-dependent. Implementations should select the "best" transformation based on criterion such as accuracy and domain of validity, but GeoAPI does not specify any rule. Users are encouraged to verify which transformation has been selected by invoking the `CoordinateOperation.toWKT()` method.

TODO: `RegisterOperations` returns a set of all operations.

### From an authority code

If a determinist operation is required and if that operation is identified by an authority code, then that operation can be obtained with `CoordinateOperationAuthorityFactory`. This approach avoids the implementation-dependent variability of the approach described in previous paragraph.

### From operation parameter values

The `ParameterValue` interface provides methods to set the value of the operation parameter. In the general use pattern for these types, a `ParameterValueGroup` containing all the named parameters for an operation method is first obtained from the `OperationMethod` or the `MathTransformFactory`, and then each `ParameterValue` instance is obtained in turn and its value set. This use pattern ensures that all the needed parameters for an operation method can be obtained as a single block. An example is given in § E.1.7 for the Java language.

## 6.5.4. Math transforms

*From OGC 01-009 § 12.4.5*

The `CoordinateOperation` object introduced in previous section provides high-level information (source and target CRS, domain of validity, positional accuracy, operation parameter values, *etc*). The actual mathematical work is performed by a separated object obtained by a call to `getMathTransform()`. At the difference of `CoordinateOperation` instances, `MathTransform` instances do not carry any metadata. They are kind of black box which know nothing about the source and target CRS, the domain or the accuracy (actually the same `MathTransform` can be used for different pairs of CRS if the mathematical work is the same), Furthermore `MathTransform` may be implemented in a very different way than what `CoordinateOperation` said. In particular many conceptually different coordinate operations such as unit conversions, axis swapping, *etc.*

can be implemented by `MathTransform` as affine transforms and concatenated for efficiency. The result may be a `MathTransform` doing in a single step a calculation described by `CoordinateOperation` as a chain of distinct operations. Having `MathTransform` separated from `CoordinateOperation` gives more flexibility to implementers for optimizations.

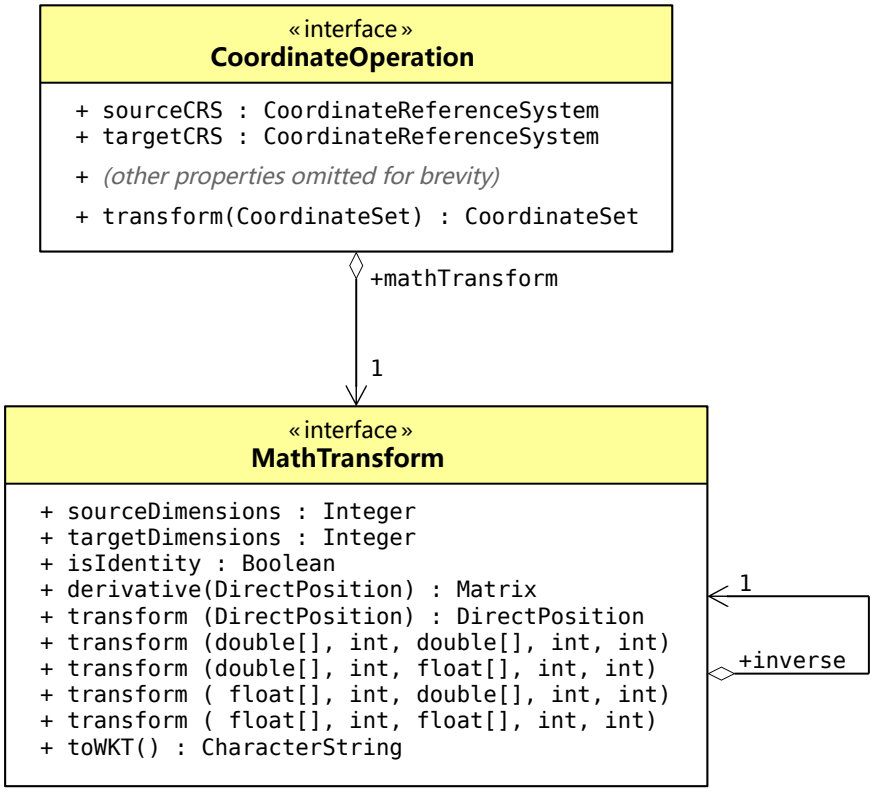


*Figure 12. Association between Coordinate Operation and Math Transform*

`MathTransform` has a method taking a `DirectPosition` (the coordinates in source CRS) in input and returning a `DirectPosition` (the coordinates in target CRS) in output. But `MathTransform` provides also various methods operating on an arbitrary amount of coordinate tuples packed in arrays of `float` or `double` types. If there is many points to transform, the methods operating on arrays will generally be much more efficient than the method operating on `DirectPosition`. The example in § E.1.8 shows how to transform in Java the coordinates of 6 cities.

## Partial derivatives

*From OGC 01-009 § 12.4.5.6*

`MathTransform` can also provide the derivative of the transform function at a point. The derivative is the matrix of the non-translating portion of the approximate affine map at the point. For example if $P$ is a map projection converting degrees of latitude and longitude $(\phi, \lambda)$ into projected coordinates $(x, y)$ in metres, then the derivative $P'$ can be represented by a Jacobian matrix as below:

$$P'(\phi, \lambda) = \begin{bmatrix} \partial x/\partial \phi & \partial x/\partial \lambda \\ \partial y/\partial \phi & \partial y/\partial \lambda \end{bmatrix}$$

### 6.5.5. Well-Known Text (WKT)

*From ISO 19162 and OGC 01-009 §7.1*

Most objects in the `opengis.referencing` packages can be imported and exported in *Well-Known Text* (WKT) format. This format allows the exchange of CRS definitions with implementations of other OGC standards such as web services. GeoAPI provides two `toWKT()` methods for producing a WKT representation of an object:

- One method is defined in the `IdentifiedObject` parent interface and is inherited notably by `CoordinateReferenceSystem`. But that method can also be used with `CoordinateOperation` and various components such as `Datum`. Well Known Text (WKT) character strings produced by that method shall be compliant with the format defined in the ISO 19162 standard.
- A second `toWKT()` method is defined in the `MathTransform` interface, which is not an object defined by ISO standards. Character strings produced by that method shall be compliant with the format defined in OGC 01-009 §7.1. Other sections such as OGC 01-009 §7.2 should be ignored since they are replaced by ISO 19162.

Difference between the two WKT representations is illustrated below. For a very simple operation doing only a swapping of latitude and longitude axes, the ISO 19162 representation of the `CoordinateOperation` object can be as below:

```
COORDINATEOPERATION["Axis order change (2D)",
  SOURCECRS[GEODCRS["RGF93", …definition omitted for brevity…, ID["EPSG", 4171]]],
  TARGETCRS[GEODCRS["RGF93 (lon-lat)", …definition omitted for brevity…, ID["EPSG", 7084]]],
  METHOD["Axis Order Reversal (2D)", ID["EPSG", 9843]],
  AREA["World."], BBOX[-90.00, -180.00, 90.00, 180.00]]
```

The OGC 01-009 §7.1 representation of the `MathTransform` associated to above coordinate operation can be as below. All metadata information (source and target CRS, domain of validity, *etc.*) are lost, and the human-readable *"Axis Order Reversal"* operation is replaced by a more mathematical *"Affine parametric transformation"* operation.

```
PARAM_MT["Affine",
  PARAMETER["num_row", 3],
  PARAMETER["num_col", 3],
  PARAMETER["elt_0_0", 0.0],
  PARAMETER["elt_0_1", 1.0],
  PARAMETER["elt_1_0", 1.0],
  PARAMETER["elt_1_1", 0.0]]
```

Those two representations are complementary. `CoordinateOperation.toWKT()` provides high-level information about the operation together with metadata for understanding the context. `MathTransform.toWKT()` said more information about how the operation is implemented, which is useful for analyzing the validity of transformation results.

The converse operation – creating an object from a WKT definition – is done by `createFromWKT(…)` methods defined in `CRSFactory` and `MathTransformFactory` interface.

### 6.5.6. Departures from ISO 19111

The main departures of GeoAPI from the ISO 19111 standards are inspired by the legacy OGC 01-009 standard published in 2001. That legacy standard included COM, CORBA and Java profiles. In support for those profiles, some aspects of the legacy model were better suited to programming languages compared to the more web-focused standards published the following years. GeoAPI retains some OGC 01-009 aspects when appropriate and adapt them to the ISO 19111 interfaces. References to OGC 01-009 clauses are given in sub-sections below.

#### Separation between coordinate operation metadata and execution

*From OGC 01-009 § 12.4.[1,5] and figure 11*

A major extension of GeoAPI compared the ISO 19111 standard comes from the inclusion, directly in the `CoordinateOperation` interface, of a method providing access to the `MathTransform` interface from the older OGC 01-009 specification. That interface performs the actual computation of coordinates in target CRS from the coordinates in source CRS. This separation – between a type providing operation metadata and a type doing the actual work – existed in OGC 01-009 and has been kept by GeoAPI for giving to developers more flexibility regarding optimizations. The operations that are really executed by `MathTransform` may be very different than the operations described by `CoordinateOperation`, provided that they are mathematically equivalent. An example is given in § 6.5.5.

Since the 2019 revision of ISO 19111 specification, the `CoordinateOperation` interface has a `transform(…)` method too. But that new method can be implemented as a convenience method redirecting the work to `MathTransform`.

#### Coordinate system association

*From OGC 01-009 § 12.3.5.3 and figure 7*

In the ISO 19111 specification, only `SingleCRS` has an association to `coordinateSystem`. GeoAPI moves this association to the `CoordinateReferenceSystem` parent interface for user convenience, because coordinate system (CS) dimension and axes are frequently requested information and will always be available, directly or indirectly, even for `CompoundCRS`. However only coordinate systems associated to `SingleCRS` can be instances of ISO 19111 concrete subtypes (`CartesianCS`, `SphericalCS`, `VerticalCS`, *etc.*); it is not possible to assign a standard concrete type to the coordinate system of a `CompoundCRS`. For the later case, users see only the abstract `CoordinateSystem` type which serves merely as an axis container. The concrete CS subtype can be hidden in implementation details.

*Rational*

This generalization would not be appropriate for a database schema or for formats such as *Well Known Text* because axes listed with a `CompoundCRS` would be redundant with axes listed with each CRS component. But this consideration does not apply to programming languages: the axes do not need to be repeated because each method such as `CoordinateSystem.getAxis(int)` contains code, not necessarily data structure. Implementations can redirect requests for `CoordinateSystem` axes to corresponding `CompoundCRS` elements.

### Type and property names

*From OGC 01-009 § 12.3.11.2*

The ISO 19111 `CRS` type name has been expanded to the `CoordinateReferenceSystem` name in GeoAPI for compatibility with previous GeoAPI versions. It is also common practice in Java language to avoid abbreviations.

In the `GeneralDerivedCRS` interface, the `conversion` association to `Conversion` has been renamed `conversionFromBase` for making clear that the source CRS is the `baseCRS`. This naming is similar to the one used in OGC 01-009 with direction reversed.

### Ellipsoid second defining parameter

*From OGC 01-009 § 12.3.10*

ISO 19111 defines the union named `SecondDefiningParameter` as being either `semiMinorAxis` or `inverseFlattening`. The union construct (defined in some languages like C/C++) does not exist in Java. GeoAPI changed the interface to require both ellipsoidal parameters (in addition to the `semiMajorAxis` parameter which is mandatory in any case), as was done in OGC 01-009. However, implementors could readily permit users to only provide one of the two parameters by creating a class which calculates the second parameter from the first. For precision, GeoAPI imports the `isIvfDefinitive` boolean property from OGC 01-009 to enable the user to establish which of the two parameters was used to define the instance.

### Omitted unions

ISO 19111 defines `GeodeticCS`, `EngineeringCS` and `DerivedProjectedCS` unions for type safety. For example the `GeodeticCS` union ensures that a `GeodeticCRS` can only be associated to a `CartesianCS`, an `EllipsoidalCS` or a `SphericalCS`. Those unions have been omitted from GeoAPI because unions are not supported in Java language, and the Python language does not enforce type safety.

#### Shared parameters

ISO 19111 defines seven types for representing coordinate operation parameters, but those types are generic enough for use in other contexts. GeoAPI moves those types in a separated `parameter` package and rename `OperationParameter` as `ParameterDescriptor`.

> The removal of the "Operation" prefix is for extending the use of these parameter interfaces to a more general use rather than only for referencing operation types. The addition of the "Descriptor" suffix is for making more apparent that this type is an abstract definition of parameters – not their actual values. It is also consistent with usage in other libraries (e.g. `ParameterListDescriptor` in *Java Advanced Imaging*).

## 6.6. Feature packages

The GeoAPI feature package uses the `opengis.feature` namespace and implements the types defined in the ISO 19109:2015 – *Rules for application schema* specification. The main UML materialized by GeoAPI is ISO 19109 figure 5 (TODO: verify). That figure is also reproduced in ISO 19103:2015 figure E.6.

TODO: provide UML and explain the mapping.

### 6.6.1. Moving feature

*From OGC 18-075 figure 3*

Features often have an attribute of type `Geometry` (ISO 19107). A sub-type of `Geometry` is `Trajectory` (ISO 19141). A feature where the geometry is a trajectory is a moving feature. In addition of time-dependent positions defined by the trajectory, a moving feature may also have time-dependent attribute values. Those attributes are represented by the `DynamicAttribute` sub-type.

## 6.7. Filter packages

The GeoAPI filter packages use the `opengis.filter` namespace and implement the types defined in the ISO 19143:2010 – *Filter encoding* specification. GeoAPI ignores the XML encoding aspects and retains only the types defined in UML diagrams.

The two most fundamental types are `Expression` and `Filter`, shown in next sections. All expressions are identified by a `ScopedName` and all filters are identified by a `CodeList`. This is an extension to ISO 19143 specification where only some specific sub-types have those identifications. This generalization allows, in some cases, to execute generic code without the need to check the filter or expression sub-types.

### 6.7.1. Expression

*From ISO 19143:2010 § 7. [3…6]*

An expression is a function that receives a resource (typically a `Feature`) in input and produces a value (typically a character string, a number or a geometry) in output. The output is restricted to `Boolean` if the expression is used as an operand of a `LogicalOperator` (§ 6.7.2).

Expressions extend the mechanism provided by the target platform for defining functions. For example in the Java language, `Expression` extends the `java.util.function.Function` interface (§ C.2).

An expression can be a literal, a reference to the values of a particular property in resources, or a named procedure (for example arithmetic operations between the results of two sub-expressions). An expression can have zero or more parameters and generates a single result. The parameters themselves are in-turn expressions and shall appear in the order in which they are defined in the `FilterCapabilities` (§ 6.7.3).

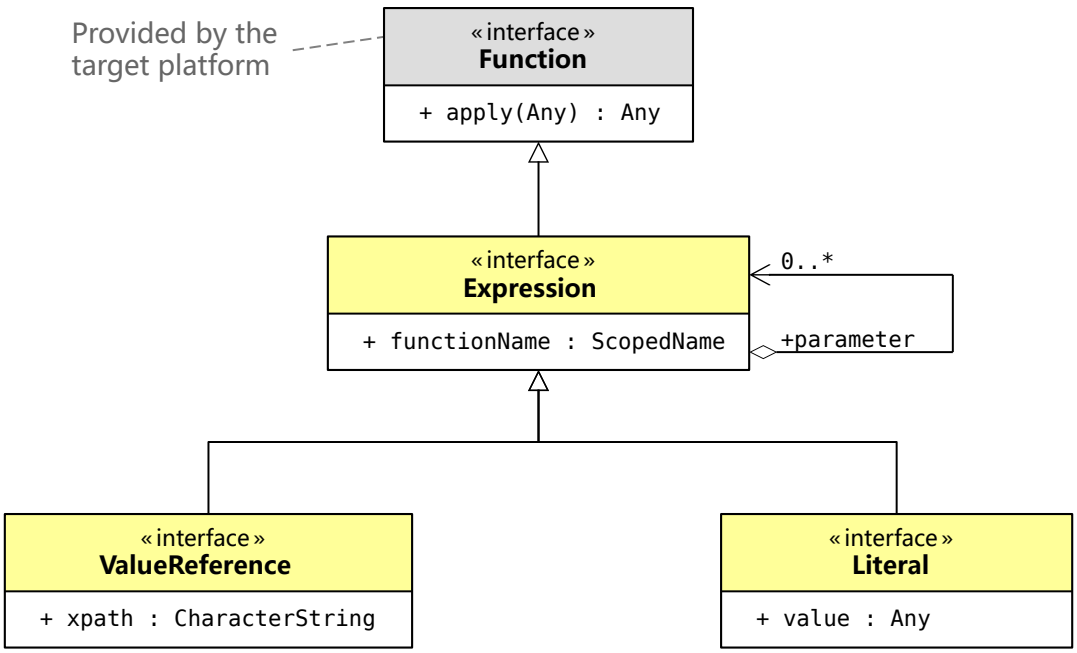


*Figure 13. Partial UML of expressions*

`ValueReference` is an expression whose value is computed by retrieving the value indicated by a name or a XPath in a resource (typically a property in `Feature` instances), and `Literal` is an expression whose value is a constant.

Parameters are specified at `Expression` creation time. For example an expression computing $x+1$ where $x$ is the value of a `Feature` property can be created with two parameters: a `ValueReference` retrieving the $x$ value for each feature and a `Literal` whose value is the integer 1. When `apply(f)` is invoked on that "Add" expression where $f$ is a `Feature` instance, the "Add" expression invokes in turn `apply(f)` on the two above-cited parameters. The values computed by those parameters are added, then returned by the "Add" expression.

### 6.7.2. Filter

*From ISO 19143:2010 §7.[7…11]*

A filter is a predicate that identifies a subset of resources from a collection of resources. Each resource instance is evaluated against a filter, which always evaluates to `true` or `false`. If the filter evaluates to `true`, the resource instance is included in the result set. If the filter evaluates to `false`, the resource instance is ignored. Roughly speaking, a filter encodes the information present in the `WHERE` clause of a SQL statement.

There are various sub-interfaces of this interface that represent many types of filters, such as simple property comparisons or spatial queries. The following diagram shows the 4 basic types of filters together with the code lists identifying which operation is applied. More specialized sub-types such as `BinaryComparisonOperator` and `DistanceOperator` are not shown in this diagram.

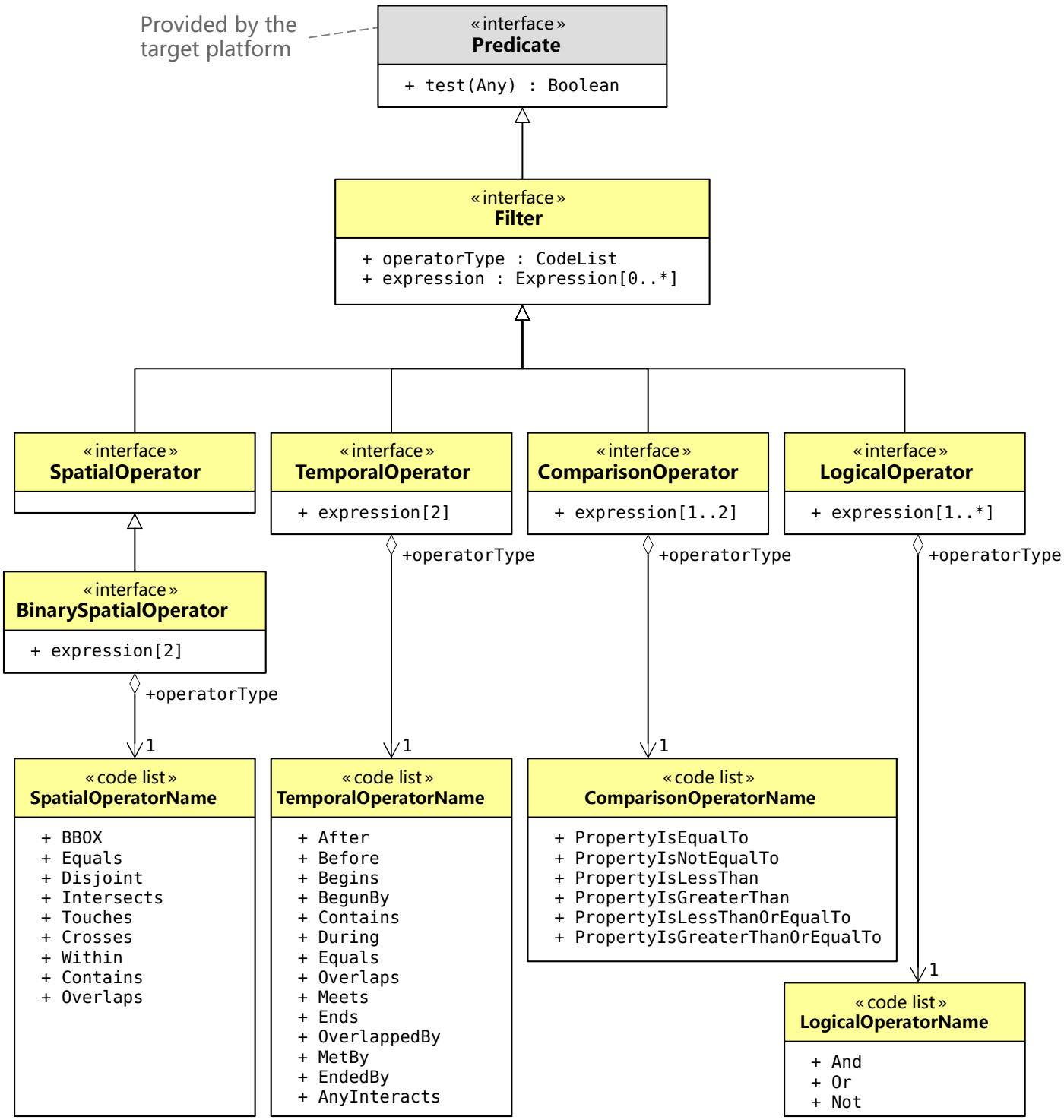


*Figure 14. Partial UML of filters*

In above diagram, the `operatorType` property defined in the `Filter` parent interface is overridden by more specialized types (shown as associations) in each sub-interface. This is type covariance. Likewise `expression` is defined in parent interface with unconstrained multiplicity, but that multiplicity is restrained in sub-interfaces.

Filter operands are expressions. For example a filter named "PropertyIsEqualTo" uses two expressions. The first expression may be a `ValueReference` fetching the value of a property and the second expression may be a `Literal` with the desired value.

### 6.7.3. Capabilities

*From ISO 19143:2010 § 7.13*

`FilterCapabilities` is the entry point for listing which expressions and filter operators are available. It capabilities are separated in the following categories:

- `IdCapabilities` lists names that represent the resource identifier elements that the service supports.
- `ScalarCapabilities` advertises which logical, comparison and arithmetic operators the service supports.
- `SpatialCapabilities` advertises which spatial operators and geometric operands the service supports.
- `TemporalCapabilities` advertises which temporal operators and temporal operands the service supports.
- `AvailableFunction` describes functions that may be used in filter expressions.
- `ExtendedCapabilities` advertises any additional operators added to the filter syntax.

The enumeration of scalar, spatial and temporal capabilities use de code lists shown in § 6.7.2.

### 6.7.4. Departures from ISO 19143

The ISO 19143 model is slightly modified in GeoAPI for retrofitting with existing interfaces in Java and Python. Some modifications are also applied for generalization when non-encoded property values can be computed. The aim is to facilitate computational operations without preventing the encoding of ISO 19143 compliant XML documents, as omitted types (for example) can be inferred.

#### Omissions relative to expressions

`Function` interface is omitted (actually retrofitted into the base `Expression` interface) for avoiding confusion with function interfaces provided natively by target platforms (such as the `java.util.function` package). Instead the `Function` properties are moved into the `Expression` parent interface because from a programming language point of view, all expressions – including `Literal` – can be viewed as a kind of function. The following partial UML diagrams illustrate the difference.

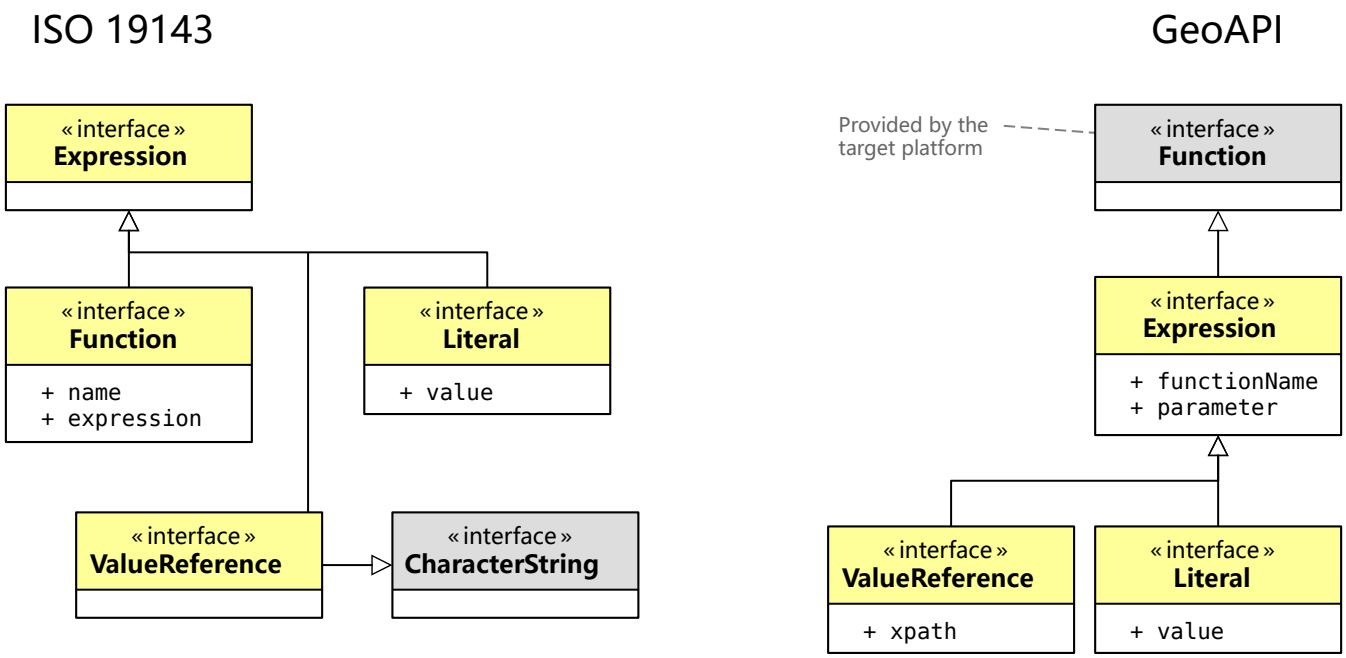


*Figure 15. Role of function interface in ISO versus GeoAPI model*

## Omissions relative to filters

`Operator` interface is omitted for simplicity reason. Instead operators defined by GeoAPI extends directly the `Filter` interface. This approach avoid an indirection level.

`NonIdOperator` and `ExtensionOperator` interfaces are omitted for semantic reasons. GeoAPI generally avoids interfaces with definition of the kind *"is not…"*. Users should extend directly the most appropriate interface instead.

`UnaryLogicOperator` and `BinaryLogicOperator` interfaces are omitted for simplicity. The `LogicalOperator` parent interface can handle both cases, with arity determined by the length of the operands list. If 1, the operator is an `UnaryLogicOperator`. If 2 or more, the operator is a `BinaryLogicOperator`. The length may be more than 2 if the AND or OR operation is repeated for all operands.

`SpatialDescription` and `TemporalOperand` unions are omitted for language constraint reasons. Unions are not supported in Java and not really needed in Python. They are replaced by recommendations in the API documentation.

## Omissions relative to capabilities

`SpatialOperatorDescription` and `TemporalOperatorDescription` in the capabilities section are omitted. The are replaced by the dictionary of target platforms (for example `Map.Entry` in Java). It reduces the number of GeoAPI defined interfaces by leveraging platform API and also makes easier to get the operands for a specific operator.

## Renaming

- The `expression` association in `Function` is renamed `parameter` for making clearer that they are function inputs.
- The `name` property in `Function` is renamed `functionName` for making its purpose clearer, since GeoAPI declares that property in the `Expression` parent interface.
- The `BinaryComparisonName` type is renamed `ComparisonOperatorName` for consistency with other operator names and for making possible to name other kinds of comparison than binary operators.

### Generalization of filter metadata

The ISO 19143 specification identifies *some* filters by a code list. That identification is provided by an `operatorType` property defined in `UnaryLogicOperator`, `BinaryLogicOperator`, `BinaryComparisonOperator`, `BinarySpatialOperator` and `DistanceOperator` interfaces. GeoAPI generalizes by declaring a `getOperatorType()` method in the base `Filter` interface. It provides a single access point for type information without the need to check for each specialized interface.

Likewise ISO 19143 specification provides different properties for getting the expressions used by a filter. Those properties have different names depending on the sub-type: `expression` (with a cardinality of 1, 2 or unlimited), `operand1`, `operand2`, or `valueReference`. GeoAPI provides a `getExpressions()` method in the parent `Filter` interface as a way to access those expressions without the need to make special cases for each sub-interfaces.

> The operand names vary in ISO 19143 specification because their types vary. But those types are often unions, which are materialized only by documentation in GeoAPI. Consequently the base `Expression` type is used almost everywhere, which make less useful to have heterogynous properties in sub-interfaces.

# 7. Requirements

This specification defines requirements for two target types: *libraries* and *applications*. A library is a software that exposes the `opengis` packages for use by independent parties. An application is a software that encapsulates the `opengis` packages for its internal working, but without exposing them to end users. Applications have less requirements than libraries.

For example compliant libraries shall obey to method signatures declared in published OGC interfaces, otherwise other developers could not base their developments on a common set of API. However applications are free to modify, add or remove methods as they see fit; if the `opengis` API of the application is not invoked by any external user, then changes to that API has no impact on inter-operability.

## 7.1. Library requirements

This section describes requirements for ensuring source compatibility or binary compatibility (when applicable) of libraries compliant with this specification. Those requirements apply to the *libraries* made available for use by other developers. The requirements usually do not apply to applications distributed to end users.

| **Requirements Class: Library** | |
|---|---|
| http://www.opengis.net/spec/ABCD/m.n/req/req-class-a | |
| Target type | Library |
| Dependency | Requirements class B: Application |
| **Requirement 1** | http://www.opengis.net/spec/ABCD/m.n/req/req-class-a/req-signatures<br>Source and binary compatibility |
| **Requirement 2** | http://www.opengis.net/spec/ABCD/m.n/req/req-class-a/req-behavior<br>Behavioral compatibility |

| | |
|---|---|
| Requirement 3 | http://www.opengis.net/spec/ABCD/m.n/req/req-class-a/req-factory-methods<br>Factory exception |
| Requirement 4 | http://www.opengis.net/spec/ABCD/m.n/req/req-class-a/req-setters<br>Setter methods |
| Requirement 5 | http://www.opengis.net/spec/ABCD/m.n/req/req-class-a/req-getter-mandatory<br>Mandatory getter methods |
| Requirement 6 | http://www.opengis.net/spec/ABCD/m.n/req/req-class-a/req-getter-optional<br>Optional getter methods |

### 7.1.1. Source and binary compatibility

| | |
|---|---|
| Requirement 1 | /req/req-class-a/req-signatures<br><br>Redistributed modules in OGC namespace shall contain the exact same set of types, methods and properties as listed in the API documentation published by OGC at the following locations:<br><br>Java: http://www.geoapi.org/snapshot/javadoc/ TODO: update URL.<br>Python: http://www.geoapi.org/snapshot/python/ |

Note that this requirement does not mean that vendors must implement all types and methods, or can not implement their own API in addition of GeoAPI. This requirement only means that modules or packages inside the `org.opengis` or `opengis` namespaces shall contain the exact same set of types as published at above links, and each of those types shall contain the exact same set of properties as published. But vendors are still free to implement only a subset of their choice and throw exception for unimplemented types and methods. Vendors can also add new types and methods, provided that those additions are in a namespace different than `org.opengis` and `opengis`. Finally, this requirement apply only to libraries redistributed for use by other developers. Final applications are free to remove any unused types or methods if such removal is invisible to other developers.

### 7.1.2. Behavioral compatibility

| | |
|---|---|
| Requirement 2 | /req/req-class-a/req-behavior<br><br>Libraries which provide code implementations of the GeoAPI interfaces shall follow the dictates of the Java or Python API documentation. |

### 7.1.3. Factory exception

| | |
|---|---|
| Requirement 3 | /req/req-class-a/req-factory-methods<br><br>Methods which create new instances, such as `Factory` methods, shall return the desired value or throw the exception documented in the API, such as `FactoryException` or a sub-type. |

### 7.1.4. Setter methods

| | |
|---|---|
| Requirement 4 | /req/req-class-a/req-setters<br><br>"Setter" methods (methods which set the value of an object) shall either succeed or throw an exception such as `UnsupportedOperationException` if the method is not implemented, `IllegalArgumentException` if the value is illegal in that implementation, or `IllegalStateException` if some context makes the use of that method illegal. This list of exception types is not exclusive. |

*Example*

A call to `parameter.setValue(-10.25)` may have the first of following results which is applicable:

- The method is supported and the -10.25 value is valid:
  The setter method returns silently.
- The method is not implemented by the library:
  `UnsupportedOperationException` is thrown.
- Values for that parameter are restricted to positive numbers:
  `IllegalArgumentException` is thrown.
- The `ParameterValue` instance has been declared unmodifiable:
  `IllegalStateException` or `UnsupportedOperationException` is thrown.

### 7.1.5. Mandatory getter methods

| Requirement 5 | /req/req-class-a/req-getter-mandatory<br><br>Unless otherwise specified in this specification or in the API documentation, mandatory "getter" methods shall return the requested value unless the value is missing in which case they shall throw an exception such as `IllegalStateException`. |
|---|---|

"Getter" methods (methods which obtain a value from an object) are documented through annotations in the Javadoc as mandatory or optional. Mandatory "getter" methods are expected to return the requested value unless the value is missing in which case they shall throw an exception. An exception is made to this requirement in the metadata packages because of the extensive existence of incomplete metadata (§ 6.3.3).

### 7.1.6. Optional getter methods

| Requirement 6 | /req/req-class-a/req-getter-optional<br><br>Optional "getter" methods shall return the requested value unless the value is missing or the method is not implemented in which case they shall return an empty collection if possible, or `null` or `None` otherwise. |
|---|---|

## 7.2. Application requirements

This section describes requirements for applications implementing GeoAPI. Those requirements apply also to libraries (§ 7.1) since applications are built from libraries.

| Requirements Class: Application | |
|---|---|
| http://www.opengis.net/spec/ABCD/m.n/req/req-class-b | |
| Target type | Application |
| Dependency | |
| **Requirement 1** | http://www.opengis.net/spec/ABCD/m.n/req/req-class-a/req-authority<br>Conformance with authority definitions |
| **Requirement 2** | http://www.opengis.net/spec/ABCD/m.n/req/req-class-a/req-wkt<br>Conformance with WKT syntax |

TODO: Describe requirements.

# Annex A: Conformance Class Test Suite (Normative)

## A.1. Conformance Class A

### A.1.1. Types and method signatures

| | |
|---|---|
| Test id: | conf/conf-class-a/req-signatures |
| Requirement: | req/req-class-a/req-signatures |
| Test purpose: | Verify that all types and properties in OGC namespace are as published. |
| Test method: | TODO: see below. |

TODO: write a Java program verifying signature of all classes and methods in the `org.opengis` namespace. This verification would be done for the "libraries" target type only, not the "applications" target type.

### A.1.2. Object invariants

| | |
|---|---|
| Test id: | conf/conf-class-a/req-invariants |
| Requirement: | req/req-class-a/req-behavior |
| Test purpose: | All the instances of GeoAPI interfaces shall be valid according to the test validator, whenever a validator exists for the instance type. |
| Test method: | TODO: describe validators. |

This does not require that all instances be tested but merely that if the instances were tested, they would validate.

# Annex B: Conformance Level (Normative)

This standard provides several levels of conformance for libraries. The levels are determined mostly by the capabilities of `referencing` or `feature` packages. This standard does not define conformance level for the `util` and `metadata` packages because those interfaces are typically implemented on a case-by-case basis in support of other packages.

All implementations must necessarily provide a fully functional implementation of the two-dimensional case of the `GeographicCRS` interface for the WGS84 ensemble of reference frames. "Fully functional" means that the implementation must support all mandatory properties of geographic CRS, notably `GeodeticReferenceFrame`, `Ellipsoid`, `PrimeMeridian`, `EllipsoidalCS` and `CoordinateSystemAxis` together with the "degree" and "metre" units of measurement. Optional properties can be omitted.

## B.1. Level A: referencing base

The first level identifies implementations capable to provide coordinate reference systems of various types (geographic, projected, vertical, temporal, compound…) identified by authority codes. This standard does not define which CRS types and which authority codes should be supported; the only requirement is that an implementation of the `RegisterOperations` interface shall be provided, potentially completed by a `CRSAuthorityFactory` for each supported authority. Implementations at this level are not required to support CRS that are not declared on their list of supported authority codes.

Coordinate operations for (source, target) pairs of CRS may be listed in a database such as the EPSG geodetic dataset. Implementations at this level are not required to support coordinate operations for (source, target) CRS pairs that are not in their database.

## B.2. Level B: referencing from components

The second level identifies implementations capable to create coordinate reference systems from their components (datum, axes, projection parameters…). Implementations at this level shall be able to create a CRS from a Well-Known Text (WKT) character string, a GML document or from CRS components provided in a programmatic way. It generally implies the support of `CRSFactory`, `CSFactory` and `DatumFactory` interfaces.

Implementations at this level should be able to create `CoordinateOperation` instances for arbitrary pairs of source and target CRS. A database may be used for known pairs of CRS, but implementations should be prepared to handle CRS not defined in the database.

# Annex C: Java Profile (Normative)

In addition to this document, this specification includes the GeoAPI Java Archive file `geoapi.jar`. That file can be downloaded using the following Maven coordinates:

```xml
<dependency>
  <groupId>org.opengis</groupId>
  <artifactId>geoapi</artifactId>
  <version>SNAPSHOT</version>
</dependency>
```

TODO: replace snapshot by final version number.

Libraries shall implement at least some interfaces defined in that archive file. It is not required to implement all interfaces. The contract for each interface and each method is specified in the javadoc.

## C.1. Units of measurement

The Java profile of GeoAPI does not materialize any of the units of measurement types (§ 6.2.9) defined by ISO 19103. Instead, unit modeling is entirely delegated to the Java Specification Request JSR-363 (TODO: upgrade to JSR-385). The ISO 19103 `Measure` type is represented by the JSR-363 `Quantity` interface. The Java standard defines various quantity subtypes in the same way than ISO 19103 does, often with the same names (`Angle`, `Length`, `Area`, `Volume`, *etc*). But contrarily to ISO 19103, JSR-363 does not define a parallel set of subtypes for units of measurement. Instead, it defines only one unit type, `javax.measure.Unit<Q extends Quantity<Q>>`, which is parameterized by the quantity type `Q`. For example instead of defining a `UomLength` subtype, developers use `Unit<Length>` to qualify the type of Unit or Measure being used. More examples are given in § E.1.5.

The mapping for ISO 19103 measurement types is shown below. Unless otherwise specified, all Java types listed below are in the `javax.measure` or `javax.measure.quantity` package.

*Table 17. Units of measurement mapping*

| Measure in ISO 19103 | Unit in ISO 19103 | Measure in Java | Unit in Java |
|---|---|---|---|
| Measure | UnitOfMeasure | Quantity<?> | Unit<?> |
| - Angle | UomAngle | Angle | Unit<Angle> |

| Measure in ISO 19103 | Unit in ISO 19103 | Measure in Java | Unit in Java |
|---|---|---|---|
| - AngularSpeed | UomAngularSpeed | Quantity<?> | Unit<?> |
| - Area | UomArea | Area | Unit<Area> |
| - Currency | UomCurrency | | java.util.Currency |
| - Length | UomLength | Length | Unit<Length> |
| └ Distance | UomLength | Length | Unit<Length> |
| - Scale | UomScale | Dimensionless | Unit<Dimensionless> |
| - Speed | UomSpeed | Speed | Unit<Speed> |
| - TimeMeasure | UomTime | Time | Unit<Time> |
| - Volume | UomVolume | Volume | Unit<Volume> |
| - Weight | UomWeight | Force | Unit<Force> |
| DirectedMeasure | | | |
| - AngularAcceleration | UomAngularAcceleration | | Unit<?> |
| - AngularVelocity | UomAngularSpeed | | Unit<?> |
| - Acceleration | UomAcceleration | | Unit<Acceleration> |
| - Velocity | UomVelocity | | Unit<Speed> |

*Differences between ISO 19103 and JSR-363:*

- Angular speed has no direct mapping to JSR-363, but some implementations may allow the use of generic type `Quantity<?>`.
- `Currency` is not considered a unit of measurement in this profile.
- The JSR-363 types do not distinguish `Distance` from `Length`.
- JSR-363 does not have a type equivalent to `DirectedMeasure`. But it still possible to represent their unit of measurement.
- `SubUnitsPerUnit`, `UnitsList` and `StandardUnits` defined in ISO 19103 are not materialized in this profile.

## C.2. Filters and Java functions

Java `Stream` provides an efficient way to deliver a potentially large amount of features. The GeoAPI interfaces for filters and expressions have been designed for working smoothly with the `java.util.stream` interfaces. In particular:

- The GeoAPI `Expression` interface extends the Java `Function` interface. Consequently expressions can be used with the Java standard `Stream.map(Function)` method.

- The GeoAPI `Filter` interface extends the Java `Predicate` interface. Consequently filters can be used with the Java standard `Stream.filter(Predicate)` method.
- The GeoAPI `SortBy` interface extends the Java `Comparator` interface. Consequently sort order can be used with the Java standard `Stream.sorted(Comparator)` method.

Note that GeoAPI aims for interoperability with streams, but does not mandate their use. implementations are free to ignore streams.

## C.3. Tests suite

The `geoapi-conformance` module contains a number of validators which can be used in JUnit test cases to test compliance of the objects created in an implementation. The GeoAPI validators can establish that certain instances are invalid and therefore can readily be integrated into the test suite of any implementation library. The following code demonstrates an example which uses the validators to evaluate an instance object created by the implementation within a unit test. This test requires the JUnit library, version 4 or later, on the Java Classpath.

```java
import static org.opengis.test.Validators.*;

public class ValidationTests {
    @Test
    public void testCRS() {
        // The implementation would build this CRS
        CoordinateReferenceSystem crs = ...;
        validate(crs);
    }
}
```

If the validation fails, the JUnit library would throw an `AssertionError`.

TODO: explain coordinate operation tests, GIGS tests.

# Annex D: Python Profile (Normative)

TODO

# Annex E: Examples

This appendix provides code snippets in the Java and Python programming languages for aspects discussed in the Geospatial API overview (section 6).

## E.1. Java examples

This section provides code snippets in the Java programming languages for aspects discussed in the Geospatial API overview (section 6).

### E.1.1. UML at runtime

The annotations in API (§ 6.1.3) are available at runtime by introspection. This is useful, for example, when code needs to marshall data using the names defined by the ISO standard rather than the GeoAPI names. In the following example, the top-level annotations indicate that the `ProjectedCRS` interface was derived from a type also named `ProjectedCRS` in the ISO 19111 standard. The enclosed annotation indicates that the `getCoordinateSystem()` method was derived from a property named `coordinateSystem` in the ISO 19111 standard, and that a non-null value must be provided by every `ProjectedCRS` instance:

JAVA

```
@Classifier(Stereotype.TYPE)
@UML(identifier = "ProjectedCRS", specification = ISO_19111)
public interface ProjectedCRS extends GeneralDerivedCRS {
    /**
     * Returns the coordinate system.
     */
    @UML(identifier    = "coordinateSystem",
         obligation    = MANDATORY,
         specification = ISO_19111)
    CartesianCS getCoordinateSystem();
}
```

At runtime, the annotation of a reference to a GeoAPI interface can be obtained as follows, taking as an example the method `getCoordinateSystem()` in the `ProjectedCRS` interface:

JAVA

```
Class<?>      type          = ProjectedCRS.class;
Method        method        = type.getMethod("getCoordinateSystem");
UML           annotation    = method.getAnnotation(UML.class);
String        identifier    = annotation.identifier();          // = "coordinateSystem"
Specification specification = annotation.specification();       // = ISO 19111
Obligation    obligation    = annotation.obligation();          // = mandatory
```

Java provides a class instance like the `ProjectedCRS.class` instance used here for every type, either interface or class, defined in the runtime. The `getMethod(…)` call uses introspection to obtain a reference to the method from which the annotation can then be obtained. The annotation system therefore provides access, at runtime, to the original definition of the element.

### E.1.2. Code list at runtime

Controlled vocabulary (§ 6.2.4) can take the form of enumerations or code lists. Those two types are implemented by `java.lang.Enum` and `org.opengis.util.CodeList` respectively. The use of `CodeList` classes includes accessing statically defined elements, defining new elements and retrieving any element defined for the code list. Considering, for example, `org.opengis.referencing.cs.AxisDirection`, the following codes could be used:

```java
AxisDirection north;
north = AxisDirection.NORTH;                                  // Compile-time value (safest).
north = AxisDirection.valueOf("NORTH");                       // Runtime value (more dynamic).
north = CodeList.valueOf(AxisDirection.class, "NORTH");       // Runtime type and value.
```

where the second and third locutions will create a new value if it does not exist. Special care should be taken to keep such calls consistent throughout the code since the `CodeList` will create a new element if there are any difference in the `String` parameters. The list of all elements (including new elements created by `valueOf(…)`) can be obtained as below:

```java
AxisDirection[] elements = AxisDirection.values();
```

### E.1.3. Names in JNDI context

The ISO 19103 name types (§ 6.2.5) define mapping methods from a name to the object identified by that name. But the mapping methods defined by ISO 19103 are not part of the `NameSpace` interface defined by GeoAPI. Instead, GeoAPI leaves that functionality to frameworks such as the Java Naming and Directory Interface™ (JNDI). Java applications which need such mapping may use the methods in the `javax.naming.Context` interface:

*Table 18. Java Naming and Directory Interface equivalences*

| ISO 19103 `NameSpace` member | `org.opengis.util.NameSpace` | `javax.naming.Context` |
|---|---|---|
| `isGlobal` | `isGlobal()` | |
| `acceptableClassList` | | |
| `generateID(Any)` | | |
| `locate(LocalName)` | | `lookup(Name)` |
| `name` | `name()` | `getNameInNamespace()` |
| `registerID(LocalName, Any)` | | `bind(Name, Object)` |
| `select(GenericName)` | | `lookup(Name)` |

| ISO 19103 `NameSpace` member | `org.opengis.util.NameSpace` | `javax.naming.Context` |
|---|---|---|
| `unregisterID(LocalName, Any)` | | `unbind(Name)` |

### E.1.4. Multilingual string

Internationalization (§ 6.2.8) is handled using objects provided by the standard Java library such as `java.util.Locale`, with the addition of one GeoAPI interface. The `org.opengis.util.InternationalString` interface provides a container for multiple versions of the same text, each for a specific `Locale` — the identifier used in Java for a specific language, possibly in a named territory.

```JAVA
NameFactory factory =  ...                        // Implementation dependent.
InternationalString multiLingual = factory.createInternationalString(Map.of(
        Locale.ENGLISH, "My documents",
        Locale.FRENCH,  "Mes documents"));

System.out.println(localized);         // Language at implementation choice.
System.out.println(localized.toString(Locale.FRENCH));
```

The method to obtain factories is not specified by this standard and therefore depends on the design of the library implementation. Also, the locale used by default depends on the choice of the implementation so the result of the call `toString()` without parameters will depend on the implementation.

### E.1.5. Parameterized units

Units of measurement are represented in Java by JSR-363 parameterized interfaces (see § C.1 for an overview). Units of the same parameterized type can be used in unit conversions like below (the `Units` class must be provided by a JSR-363 implementation):

```JAVA
Unit<Length> source = Units.NAUTICAL_MILE;
Unit<Length> target = Units.KILOMETRE;
UnitConverter converter = source.getConverterTo(target);
double distance = converter.convert(124);
System.out.println(distance);               // Prints 229.648
```

where the initial calls define units of length and then a converter is used to obtain the equivalent length in a new unit. The next example below creates a "kPa" unit of measurement from the "Pa" unit.

```JAVA
Unit<Pressure> kPa = Units.PASCAL.multiply(1000);
System.out.println( kPa );      // May print "kPa" or "1000*Pa" depending on implementation.
```

The next example creates a new quantity derived from the operand types. It may print "8 mW" – the exact number and unit shown depend on the implementation:

```java
Force  f = Quantities.create(4, Units.NEWTON);
Length d = Quantities.create(6, Units.MILLIMETRE);
Time   t = Quantities.create(3, Units.SECOND);
Quantity<?> e = f.multiply(d).divide(t);
System.out.println(e);
```

### E.1.6. Metadata

The interfaces in the GeoAPI metadata packages (§ 6.3) are primarily containers of primitive types and other metadata types. Metadata elements will be encountered for example from interfaces in the referencing packages. The metadata interfaces enable users to decompose a given element into smaller elements. As an example, the following code prints a list of all individuals (ignoring organizations) for a document starting with a `Citation` element:

```java
Citation citation = ...;    // We assume this instance is already available

for (Responsibility rp : citation.getCitedResponsibleParties()) {
    if (rp.getRole() == Role.AUTHOR) {
        for (Party party : rp.getParties()) {
            if (party instanceof Individual) {
                InternationalString author = rp.getName();
                System.out.println(author);
            }
        }
    }
}
```

The remainder of the metadata packages work in similar ways, where client code must decompose an instance to obtain the elements needed.

#### Write operations

The GeoAPI metadata interfaces provide no methods to set the values of the types. Furthermore, because the way that wildcards for Java Generics have been used in the interfaces, the collection instances are constrained to be read only. Implementers are free to provide a fully mutable implementation of GeoAPI interfaces, but users may need to cast to the implementation classes in order to modify a metadata.

### E.1.7. Coordinate reference system

A Coordinate Reference System (§ 6.5) can be constructed on its own or can be derived from other systems. This example mixes both cases by creating a `ProjectedCRS` derived from a `GeographicCRS` with the Mercator projection. Here we use an authority which has already defined the geographic CRS, the coordinate system and the projection method that we need for this example. Then the example creates a *defining conversion* from the parameter values and combines it with above-cited predefined components for creating the projected CRS.

TODO: Get factories from ServiceLoader.

```java
/*
 * Factories for obtaining predefined components from a geodetic registry
 * (in this example, from EPSG geodetic dataset).
 */
CSAuthorityFactory csRegistry = ...;
CRSAuthorityFactory crsRegistry = ...;
CoordinateOperationAuthorityFactory opRegistry = ...;
/*
 * Factories for creating objects from programmatic parameters.
 */
CRSFactory crsFactory = ...;
CoordinateOperationFactory opFactory = ...;
/*
 * Predefined components used in this example:
 *
 *   - EPSG::4326   —   WGS 84
 *   - EPSG::4440   —   Cartesian 2D. Orientations: east, north. UoM: m
 *   - EPSG::9804   —   Mercator (variant A)
 */
GeographicCRS   baseCRS = crsRegistry.createGeographicCRS  ("EPSG::4326");
CartesianCS     projCS  =  csRegistry.createCartesianCS    ("EPSG::4400");
OperationMethod method  =  opRegistry.createOperationMethod("EPSG::9804");
/*
 * Get the parameters initialized to their default values,
 * then set parameter values.
 */
ParameterValueGroup pg = method.getParameters().createValue();
pg.parameter("Latitude of natural origin").setValue(0.0);
pg.parameter("Longitude of natural origin").setValue(110.0);
pg.parameter("Scale factor at natural origin").setValue(0.997);
pg.parameter("False easting").setValue(3900000.0);
pg.parameter("False northing").setValue(900000.0);
/*
 * Create the defining conversion from above parameters.
 */
Conversion definition = opFactory.createDefiningConversion(
        Map.of(Conversion.NAME_KEY, "Makassar / NEIEZ"), method, pg);
/*
 * Create the projected CRS using conversion defined by parameters
 * and predefined components fetched from EPSG geodetic dataset.
 */
ProjectedCRS makassar = crsFactory.createProjectedCRS(
        Map.of(Conversion.NAME_KEY, "Makassar / NEIEZ"),
        baseCRS, definition, projCS);
```

Above example created the "Makassar / NEIEZ" coordinate reference system in a hard way for showing the use of factories. But when an authority code is available (which is the case of above example), the use of an authority factory is recommended not only because it is easier, but also because it generally provides more metadata information such as scope and domain of validity. The following example creates the same CRS than above example:

```java
CRSAuthorityFactory crsRegistry = ...;
ProjectedCRS makassar = crsRegistry.createProjectedCRS("EPSG::3002");
```

### E.1.8. Coordinate operation

This usage examples lets implementation infers a coordinate operation (§ 6.5.3) for a pair of coordinate reference systems. The `CoordinateOperationFactory` does all the work of establishing which parameters should be used and correctly instantiating the operation.

JAVA
```
// We assume these instances are already available.
CoordinateReferenceSystem sourceCRS = ...;
CoordinateReferenceSystem targetCRS = ...;

// Implementation infers an operation.
CoordinateOperationFactory opFactory = ...;
CoordinateOperation op = opFactory.createOperation(sourceCRS, targetCRS);
```

The following example uses the operation we just created for transforming coordinate tuples. It is possible to transform `DirectPosition` instances, but if there is many points to transform then packing them in a single array is generally more efficient. For simplicity reasons, this example performs the coordinate operation in-place (i.e. transformed coordinates overwrite source coordinates) and assumes that both source and target CRS are two-dimensional. But GeoAPI is not restricted to these assumptions.

JAVA
```
// We assume this instance is already available.
CoordinateOperation op = ...;

// We assume that the source CRS is geographic.
double[] coordinates = {
    49.250, -123.100,           // Vancouver
    37.783, -122.417,           // San-Francisco
    45.500,  -73.567,           // Montreal
    40.713,  -74.006,           // New-York
    48.801,    2.351,           // Paris
    35.666,  139.772};          // Tokyo

MathTransform mt = op.getMathTransform();
mt.transform(coordinates, 0, coordinates, 0, 6);    // 6 is the number of points.
```

# Annex F: Revision History

This GeoAPI standard evolved from an effort at the Open Geospatial Consortium (OGC) and in the free software community focused on developing a library of interfaces defining a coherent data model for the manipulation of geospatial data based on the data model defined in the OGC Abstract Specifications. GeoAPI interfaces originates with the publication in January 2001 of the implementation specification OGC 01-009 *Coordinate Transformation Services* Revision 1.00 (Martin Daly, ed.) which included a set of interfaces written in different programming languages and in the `org.opengis` namespace. The GeoAPI project started in 2003 as an effort from several contributors to develop a set of Java language interfaces which could be shared between several projects. The GeoAPI project subsequently considered the interfaces of OGC 01-009 as version 0.1 of GeoAPI and started working on GeoAPI 1.0 in collaboration with developers writing the OGC specification *Geographic Objects*. Subsequently, the Open Geospatial Consortium jettisoned its own Abstract Specifications and adopted, as the basis for further work, the standards developed by the Technical Committee 211 of the International Organization for Standardization (ISO) in its ISO 19100 series. The GeoAPI project therefore realigned its interfaces with those standards. In 2003, version 1.0 of GeoAPI interfaces was released to match the release of the first public draft of the implementation specification OGC 03-064 *GO-1 Application Objects* Version 1.0 (Greg Reynolds, ed.). The standardization effort of GO-1 took a couple of years during which extensive work was made on GeoAPI interfaces. Release 2.0 of GeoAPI was made at the time of the final publication of the GO-1 specification in 2005. GO-1 has been retired later, but a new working group has been formed in 2009 for continuing GeoAPI development with a more restricted scope: to provide interfaces for existing OGC standards only, without defining new conceptual models. GeoAPI 3.0.0 has been released in 2011 and GeoAPI 3.0.1 in 2017.

| Date | Release | Editor | Clauses modified | Description |
|---|---|---|---|---|
| 2009-04-08 | 3.0.0-Draft | Adrian Custer | All | Initial public draft |
| 2009-09-06 | 3.0.0-Draft-r1 | Martin Desruisseaux | Annex E | List of departures |
| 2010-02-11 | 3.0.0-Draft-r2 | Martin Desruisseaux | 8.1.1, 10.1, annex F | Clarifications |
| 2016-11-07 | 3.0.1 | Martin Desruisseaux | 3, 8.1.6, 8.2, annex G | Replaced JSR-275 by JSR-363 |

| Date | Release | Editor | Clauses modified | Description |
|---|---|---|---|---|
| 2021-XX-XX | 3.1 / 4.0 | Martin Desruisseaux | All | Rewrite |

TODO: update references in above table. The old sections are listed below. They need to be replaced by new section numbers.

- 3: Normative references
- 8.1.1: Primitive types (numeric, text, date & time, boolean, enumeration)
- 8.1.6: Derived types (units of measurement)
- 8.2: Use of the utility types (examples with NameFactory and unit conversion)
- 10.1: Geometry packages - Defined types (Position, DirectPosition, Envelope).
- annex E: list of departures
- annex F: Comparison with legacy OGC specifications
- annex G: Reference implementation

## F.1. Future work

This version of the standard does not propose a complete set of interfaces covering the entire set of OGC/ISO abstract standards, but focuses on an initial group of interfaces only. This initial group of interfaces covers enough of the abstract model to permit the description of geospatial metadata, reference systems, features, and to enable operations on coordinate tuples. The work writing interfaces matching other OGC specifications is done in the “pending” directory of the GeoAPI project. It is expected that these other interfaces will be proposed for standardization in subsequent revisions of this specification but the interfaces must first have been implemented, ideally several times, and then tested extensively by use.

# Annex G: Bibliography

Additional information and definitions were taken from the following sources.


- IOGP. The EPSG Geodetic Parameter Dataset, https://epsg.org/
- ISO 31. Quantities and units (1992)
- ISO 1000. SI units and recommendations for the use of their multiples and of certain other units (1992)
- ISO 19115-3. Metadata — XML schema implementation for fundamental concepts (2016)
- JUnit team. The Junit framework, https://junit.org/
- OGC. Features and geometry — Part 1: Feature models (2020), OGC 17-087r13 (https://docs.ogc.org/as/17-087r13/17-087r13.html)
- OGC. Open Geospatial APIs — White Paper (2016), OGC 16-019r4 (https://docs.opengeospatial.org/wp/16-019r4/16-019r4.html)
- UCUM. The Unified Code for Units of Measure, https://unitsofmeasure.org/


Last updated 2021-12-06 17:26:37 +0800