

FeatureType Survey

Table of contents

- [#Introduction](#)
- [#Goals](#)
- [#Schema mapping essentials](#)
- [#Current API overview](#)
 - [#Feature related architectural layers](#)
 - [#Feature API](#)
 - [#AttributeType class diagram](#)
- [#XML/GML Schema support requirements](#)
 - [#XML Schema simple types that needs to be supported](#)
 - [#Simple types breakdown](#)
 - [#List data types](#)
 - [#Union data types](#)
 - [#String data types](#)
 - [#Temporal data types](#)
 - [#Numeric data types](#)
 - [#Miscellaneous data types](#)
 - [#Types that does not needs support](#)
 - [#Complex type constructs that needs to be supported](#)
 - [#Multiple geometric attributes](#)
 - [#Complex content models](#)
 - [#choice](#)
 - [#sequence](#)
 - [#all](#)
 - [#Nested Features](#)
 - [#XLinking](#)
 - [#Multiple namespaces](#)
 - [FeatureCollection's Feature API](#)
 - [#Object Identity](#)
 - [#Association type](#)
 - [#Current requirements support report](#)
 - [#Simple types support report](#)
 - [#Restriction support](#)
 - [#Complex types support report](#)
 - [#Enhancement requirements](#)
 - [Minimum requirements: Level 0 Profile of GML3 for WFS](#)
 - [Requirements break down](#)
 - [#Enhancements proposal](#)

Introduction

The GeoTools FeatureType model has born with the concept of being mappable to GML schemas, in the hope of being directly usable by a Web Feature Service implementation. The concepts governing the FeatureType model was deeply inspired on the Simple Features for SQL implementation specification. That meant that for a while, this model served perfectly for the representation of simple, also called flat, feature types, which are those whose properties are all scalar.

But the notion of complex feature types, which includes multiple instances of a given property type, nested features, etc, was always there, mostly because of the need of a deeper support of GML. This lack of support for complex

constructs forced Rob A. and his partners to develop the `complex_sco` GeoServer branch to bypass this limitation through a customized mechanism for querying and encoding scientific data such as water quality and geochemistry datasets.

Then, Chris Holmes and David Zwiers joint together and had a try at extending the GeoTools feature type model for a wider synchronization of both schema models. As a result, we have now a potentially way more useful type hierarchy to model complex schemas and an outdated GeoServer branch with functionality needed in the core product.

In this document we will open up a discussion of the 2.1.0 status of the FeatureType model to identify the key aspects that still need to be enhanced, and hopefully outline a strategy for the achievement of the level of expressiveness needed to take a FeatureType instance and construct a to more meaningful GML schema.

Goals

- To evaluate the current coverage level of the FeatureType API regarding GML Schemas
- To propose the enhancements needed for a better, if not optimal, coverage of the GML Schema constructs

Schema mapping essentials.

The current FeatureType capabilities includes only the representation of flat schemas. A flat schema, or simple feature type, maps to a complex xml type whose attributes are only a sequence of scalar attributes:

```
<xs:element name="road">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name"
type="xs:string"/>
      <xs:element name="centerline"
type="gml:LineStringPropertyType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Simple attributes may also have restrictions (aka, facets):

```

<xs:element name="road">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name"
type="xs:string"/>
      <xs:element name="code">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern
value="[A-Z][A-Z][A-Z]"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="centerline"
type="gml:LineStringPropertyType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

The GeoTools AttributeType interface, which is the superinterface for all the different possible types of attributes, has an initial attempt of supporting restrictions, or facets, by using a Filter to express the restriction, and convenient methods for expressing multiplicity and nillability:

AttributeType.java

```
public interface AttributeType{
    ...
    Filter getRestriction();
    boolean isNillable();
    int getMinOccurs();
    int getMaxOccurs();
    ...
}
```

The drawback is that still no DataStore implementation is able of meaningfully using this abilities.

XML attributes

It is **not** the goal of the GeoTools FeatureType API to be able of reflecting each and every possible XML schema, but only those valid for GML. Better said, GML schemas.

GML schemas express the properties of a Feature as elements, never as attributes. This is, though, a GML recommendation, since nothing can prevent one from declaring a set of element attributes in a GML/XML schema. The point is that we're not going to worry, through this document, that the FeatureType API should be able to reflect if a given feature property maps to a XML attribute or to a XML element. We'll assume that feature properties always maps to a Feature property in the GML sense, following the Object/property pattern.

In the case that, for reasons of Application Schema design, a GeoTools AttributeType may need to be mapped to a XML attribute for Application Schema conformance, other mechanisms should exist to perform such a mapping, that have nothing to do with the way the GeoTools FeatureType API is or will be modeled.

There are, though, a few attributes that most GML instances use: `gml:id`, `xlink:href` and `srsName`. They're already being handled at Feature to GML serialization time.

Thus, the GeoTools FeatureType API does not needs to worry about any specific issue related to XML attributes, or to XML at all, beyond ensuring that it provides a straightforward mechanism of modeling a GML schema.

Related features

GML supports relationships between features via a simple "association" property that allows inline containment of related features, or use of `xlink` to reference them. The latter is perhaps the critical one to implement, since duplication of contained features with identical `gml:ids` is not legal.

We also need to expose the types of related features through the API because it is necessary to formulate requests - e.g. find roads that join junction B33454

Unable to render {include} The included page could not be found.

XML/GML Schema support requirements

To support the mapping of a GML schema, the FeatureType API needs at least to support the basic XML Schema types.

XML Schema types can be simple or complex:

- **Simple data types:** simple types defines XML elements that have no attributes. They can be:
 - **Atomic:** The - lexical space- of an - atomic- data type is a set of literals whose internal structure is specific to the datatype in question. For instance, they can be:
 - **Primitive** datatypes are those that are not defined in terms of other datatypes; they exist ab initio.
 - **Derived** datatypes are those that are defined in terms of other datatypes.
 - **List datatypes** are those having values each of which consists of a finite-length (possibly empty) sequence of values of an - atomic- datatype.
 - **Union datatypes** are those whose - value space- s and - lexical space- s are the union of the - value space- s and - lexical space- s of one or more other datatypes. Much like a union in C.
- **Complex data types** provides for:
 - Constraining element information item [children] to be empty, or to conform to a specified element-only or mixed content model,
 - Using the mechanisms of [Type Definition Hierarchy](#) to derive a complex type from another simple or complex type.
 - Controlling the permission to substitute, in an instance, elements of a derived type for elements declared in a content model to be of a given complex type.

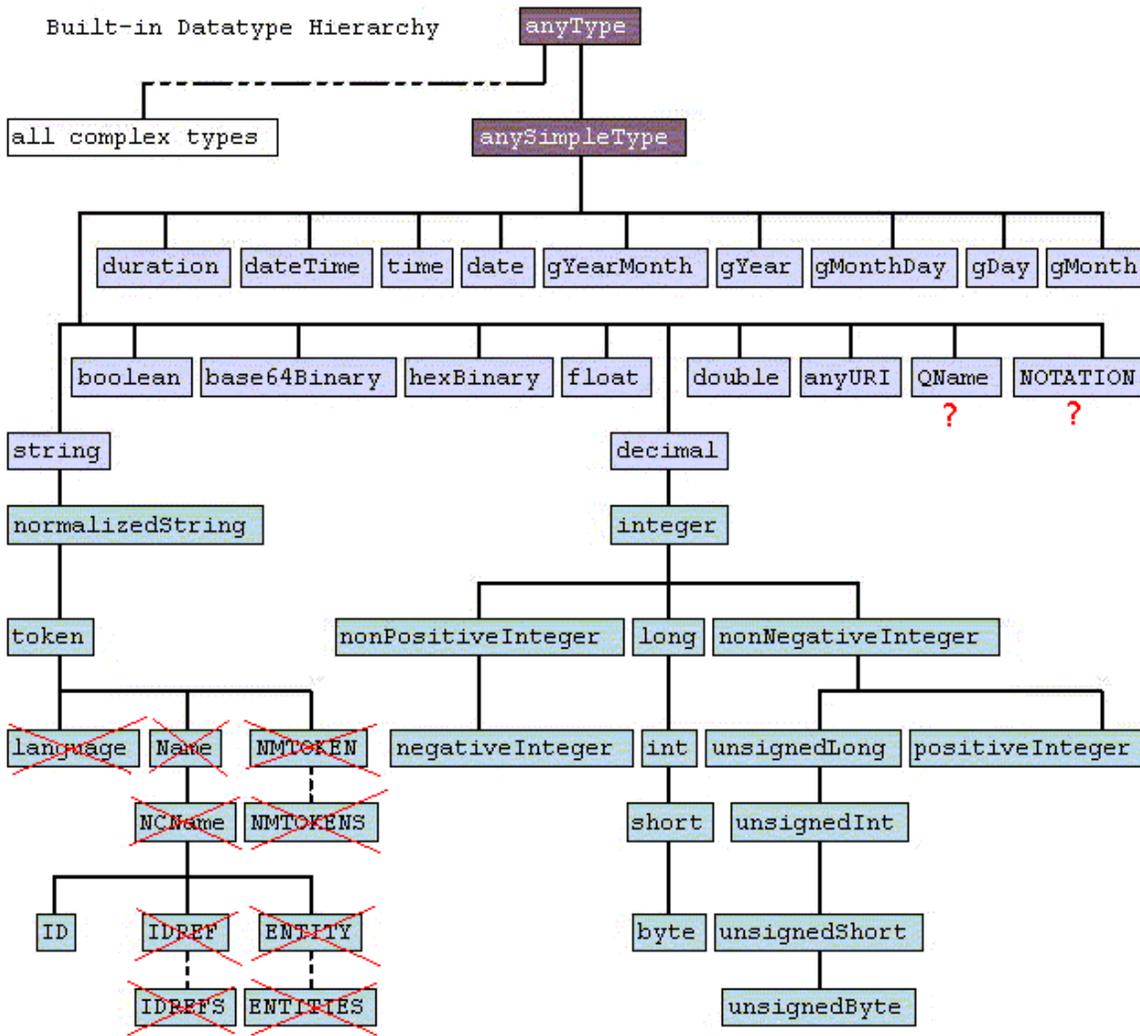
Simple types may have restrictions, also called **facets**, which are a limitation of its value space.

XML Schema simple types that needs to be supported

F.2.1.2.11 Predefined basic types (page 527)

The simple types from the XML Schema and GML namespace listed in the left hand column of Table 22 may be used in the GML application schema. All other simple types from these namespaces shall not be used in a GML application schema.

This sections contains a review of the relevant XML Schema types that the API should support in order to allow a bidirectional mapping of an XML Schema and a FeatureType instance.



- ur types
- built-in primitive types
- built-in derived types
- complex types
- derived by restriction
- derived by list
- derived by extension or restriction

This diagram shows primitive and derived built in data type of XML Schema that needs to be supported by the AttributeType model for strongly mapping a GML schema.

Simple types breakdown

List data types

List data types are simple types that are defined as an aggregation or collection of a given atomic data type. Thus, they're always derived from the concrete atomic type.

Example:

```

<xs:element name="byteList">
  <xs:simpleType>
    <xs:list itemType="xs:byte" />
  </xs:simpleType>
</xs:element>

```

this XML Schema fragment defines an element named `byteList` whose instances may hold only a space separated list of `byte` literals. If we had to represent it in Java, would do it simply with a `List<Byte>`. An instance of this element may look like:

```

<byteList>1 2 3 127</byteList>

```

The following facets apply to list data types:

- [length](#)
- [maxLength](#)
- [minLength](#)
- [enumeration](#)
- [pattern](#)
- [whiteSpace](#)

When evaluating a length type restriction, the unit of length is measured in number of list items.

Union data types

The - value space- and - lexical space- of a - union- datatype are the union of the - value space- s and - lexical space- s of its - memberTypes- . - union- datatypes are always - derived- . The member types of a union data type may be any combination of one or more **atomic or list** data types.

Example:

```

<xs:simpleType name="aUnionType">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="gml:booleanOrNull" />
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="gml:doubleOrNull" />
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token" />
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

```

this XML Schema fragment defines an type whose instances may hold either a `gml:booleanOrNull`, `gml:doubleOrNull`, or `xs:token`

Instances of this element may look like:

```
<unionInstance xsi:type="aUnionType">Null</unionInstance>
<unionInstance xsi:type="aUnionType">1.0</unionInstance>
<unionInstance xsi:type="aUnionType">JUST_A_TOKEN</unionInstance>
<unionInstance xsi:type="aUnionType">1</unionInstance>
```

i There are no facets defined for union data types. Instead, value instances may validate against one of the member data types of the union.

String data types

Type name	Description	Restrictions
ID	A string that represents the ID attribute in XML. The BNF for an ID attribute is as follows: NCName ::= (Letter '_') (NCNameChar)* /* An XML Name, minus the ":" */ NCNameChar ::= Letter Digit '.' '-' '_' CombiningChar Extender	<ul style="list-style-type: none">• enumeration• length• maxLength• minLength• pattern• whiteSpace
normalizedString	A string that does not contain line feeds, carriage returns, or tabs	"
string	A string	"
token	A string that does not contain line feeds, carriage returns, tabs, leading or trailing spaces, or multiple spaces	"

Temporal data types

Type name	Description	Restrictions
date	Defines a date value	<ul style="list-style-type: none">• enumeration• maxExclusive• maxInclusive• minExclusive• minInclusive• pattern• whiteSpace
dateTime	Defines a date and time value	"

duration	Defines a time interval	"
gDay	Defines a part of a date - the day (DD)	"
gMonth	Defines a part of a date - the month (MM)	"
gMonthDay	Defines a part of a date - the month and day (MM-DD)	"
gYear	Defines a part of a date - the year (YYYY)	"
gYearMonth	Defines a part of a date - the year and month (YYYY-MM)	"
time	Defines a time value	"

Numeric data types

Type name	Description	Restrictions
byte	A signed 8-bit integer	<ul style="list-style-type: none"> • enumeration • fractionDigits • maxExclusive • maxInclusive • minExclusive • minInclusive • pattern • totalDigits • whiteSpace
float	IEEE single-precision 32-bit floating point type	"
double	IEEE double-precision 64-bit floating point type	"
decimal	A decimal value	"
int	A signed 32-bit integer	"
integer	An integer value whose value space is the infinite set	"

long	A signed 64-bit integer. long is derived from integer by setting the value of - maxInclusive- to be 9223372036854775807 and - minInclusive- to be -9223372036854775808.	"
negativeInteger	An integer <= -1	"
nonNegativeInteger	An integer >= 0	"
nonPositiveInteger	An integer <= 0	"
positiveInteger	An integer <= 1	"
short	A signed 16-bit integer	"
unsignedLong	An unsigned 64-bit integer	"
unsignedInt	An unsigned 32-bit integer	"
unsignedShort	An unsigned 16-bit integer	"
unsignedByte	An unsigned 8-bit integer	"

Miscellaneous data types

Type name	Description	Restrictions
boolean	boolean has the - value space- required to support the mathematical concept of binary-valued logic: {true, false, 1, 0}.	<ul style="list-style-type: none"> • pattern • whiteSpace
base64Binary	represents Base64-encoded arbitrary binary data	<ul style="list-style-type: none"> • length (measured in octets (8 bits) of binary data) • minLength • maxLength • pattern • enumeration • whiteSpace
hexBinary	represents arbitrary hex-encoded binary data	"

anyURI	represents a Uniform Resource Identifier Reference (URI)	<ul style="list-style-type: none"> • length • minLength • maxLength • pattern • enumeration • whiteSpace
------------------------	--	--

Types that does not needs support

I think the following types needs not to have direct mappings at the AttributeType level, though I may be wrong with some of them:

QName, NOTATION, language, Name, NCName, NMTOKEN, NMTOKENS, IDREF, IDREFS, ENTITY, ENTITIES.

Complex type constructs that needs to be supported

One of the main goals of the "complex schema" support project is that the GeoTools FeatureType API becomes expressive enough as to seamlessly represent a GML3 schema. That is, one with complex features.

Complex features are those that have complex properties.

Complex properties may be as complex as any [XML Schema complex type definition](#)

A complex property may be complex by its own, or because it is a Feature association.

The following are the requirements that our API must support in order to be capable of modeling complex features:

i Jody's feedback

In particular ensuring that the modeling power of FeatureType / AttributeType are capable of describing both FeatureCollections and Features. One way I have approached the problem is to ensure that the correct XPath expression can be generated based on the information available in the FeatureType/AttributeType, and that the Feature / FeatureCollection data structure is complete enough that the expression can be used successfully to process an XPath expression.

This leads me to the following simple QA tests:

- Can I make an SLD document based on FeatureType / AttributeType description?
- Using FeatureCollection.getFeatureType() can I figure out what the child features are?

i Is FeatureType an AttributeType?

[Jody Garnett](#)

The relationship between FeatureType (not present in your diagram), ListAttributeType and FeatureAttributeType leads me to think that FeatureType may better be served extending AttributeType.

[Chris Holmes](#)

We've actually talked about doing that every single time we've even thought about redoing the Feature stuff, and always decided against it.

IanS has many good arguments on why this is a bad thing, in the archives. It comes down to putting too much responsibility on the FeatureType if it extends AttributeType, it starts to do two things when it should just do one. I can dig up the arguments again if needed, as I always forget, but every time I read them I'm convinced again.

Multiple geometric attributes

A complex FeatureType may have zero to N geometric attributes.

Geometric attributes are those whose values are a Geometry primitive or a Geometry aggregation. For instance, Point, LineString, MultiPoint, etc.

They're always represented in GML as an association, where the property name is the place holder for the actual geometry element. For example:

```
<Road>
  <code>66</code>
  <the_geom>

  <gml:LineString><gml:coordinates>...</gml:co
ordinates></gml:LineString>
  </the_geom>
</Road>
```

is an instance of a Road feature type, with a geometric property named `the_geom`.

That a feature type supports multiple geometric attributes means that more than one of its properties are of a geometry type. Like in:

```

<Road>
  <code>66</code>
  <from_point>
    <gml:location>....</gml:location>
  </from_point>
  <to_point>
    <gml:location>....</gml:location>
  </to_point>
  <the_geom>

  <gml:LineString><gml:coordinates>...</gml:co
ordinates></gml:LineString>
  </the_geom>
</Road>

```

In this case, the Road feature type has three geometric attributes, `from_point`, `to_point`, and `the_geom`. The following considerations apply to features with multiple geometric properties:

- the bounds of a Feature instance is the union of the bounds of all its geometric attributes.
- one of the geometric attributes must be the *default* geometry.

i fortunately this issues are already well covered by the GeoTools Feature API:

- a FeatureType may have as many GeometryAttributes as it wants
- the bounds are calculated
- the default geometry must be explicitly set by the data provider, is obtained by `Feature::getDefaultGeometry():Geometry`, and which attribute type is the default geometric one is obtained through `FeatureType::getDefaultGeometry():GeometryAttributeType`

Complex content models

A complex property is one that contains other complex or scalar properties.

There are three different kinds of complex properties, mandated by the XML Schema order indicators: *choice*, *sequence*, and *all*.

These are *order indicators*, that group properties affecting how they may appear.

Thus, these *order indicators* specifies a *content model* or *schema* for the property they belongs to, independently of the *type* of values of the properties they allow as children.

Following is a brief explanation of how these three different content models for complex properties act:

choice

A *choice* is like a union in C. It contains another properties but a value instance of this type has to of one of the types in the choice:

choice

```
<xs:element name="the_geom">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="gml:Point"/>
      <xs:element ref="gml:Polygon"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

In this case, an instance of this property may consist of either a `gml:Point` or a `gml:Polygon` element, but not both.

sequence

A *sequence* indicator specifies that the nested attributes must appear in the specified order

sequence

```
<xs:element name="the_geom">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="gml:Point"
minoccurs="0" maxoccurs="unbounded"/>
      <xs:element ref="gml:Polygon"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

In this case, an instance of this property may consist of any number of `gml:Point` elements (see `minOccurs` and `maxOccurs`), followed by one `gml:Polygon` element, in that order.

all

The *all* indicator specifies that the child elements can appear in any order and that each child element must occur once and only once

When using the *all* indicator you can set `minOccurs` to 0 or 1 and `maxOccurs` can only be set to 1

all

```
<xs:element name="the_geom">
  <xs:complexType>
    <xs:all>
      <xs:element ref="gml:Point"
minOccurs="0" />
      <xs:element ref="gml:Polygon" />
    </xs:all>
  </xs:complexType>
</xs:element>
```

In this case, an instance of this property may consist of zero or one `gml:Point` elements and one `gml:Polygon` element, in any order.

Nested Features

In GML, a Feature property value may be indeed another Feature instance. We'll call them *nested Features*.

Nested Features is a special case of complex properties, governed by a GML convention: the schema of a nested feature is mandated to be of GML `FeaturePropertyType`, following the `gml:AssociationType` pattern.

schema	sample instance
<pre><xs:complexType name="RoadType"> <xs:complexContent> <xs:extension base="gml:AbstractFeatureType"> <xs:sequence> <xs:element name="roadName" type="xs:string" /> <xs:element name="centerLine"</pre>	

```

type="gml:LineStringPropertyType
"/>
  <xs:sequence>
  </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType
name="RoadPropertyType">
  <xs:annotation>
    <xs:documentation>Container
for a Road -
follow gml:AssociationType
pattern.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:restriction
base="gml:FeaturePropertyType">
      <xs:sequence>
        <xs:element ref="Road"
/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="CityType">
  <xs:complexContent>
    <xs:extension
base="gml:AbstractFeatureType">
      <xs:sequence>
        <xs:element
name="cityName"
type="xs:string"/>
        <xs:element
name="roadProperty"
maxOccurs="unbounded"
type="RoadPropertyType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name='Road'
type='RoadType'
substitutionGroup="gml:_Feature"
/>
<xs:element name='City'

```

```

<City gml:id="city.1">
  <cityName>Bilbao</cityName>
  <roadProperty>
    <Road gml:id="road.1">

<roadName>Kastrexana</roadName>
    <centerLine>

<gml:LineString>...</gml:LineSt
ring>
    </centerLine>
  </Road>
</roadProperty>
  <roadProperty>
    <Road gml:id="road.2">
      <roadName>A8</roadName>
      <centerLine>

<gml:LineString>...</gml:LineSt
ring>
    </centerLine>
  </Road>
</roadProperty>
</City>

```

```
type='CityType'  
substitutionGroup="gml:_Feature"  
</>
```

A feature association instance can be encoded by *value* or by *reference*. *By value* means that in the XML document instance, the feature property definition is inlined on the corresponding xml element body, and *by reference* means that it exists somewhere else in the document, and thus it is remotely referenced by its **unique identifier** by using a `xlink:href` element attribute.

XLinking

There is a restriction, though, on the way that nested features can be encoded in a GML document:

- Features has a FID
- FIDs are used to uniquely identify a feature instance through the `gml:id` attribute
- IDs must be unique on a GML document
- ⚠️ so you can't encode the same feature twice on the same GML document.

⚠️ Note that remotely referencing a Feature property is not optional: Features have ids, ids are unique, so if a Feature instance appears more than one time on a document, all but one occurrence **must** be referenced properties, and only one must be a valued property.

⚠️ Mad Idea

This makes me think ahead on the concept of a LazyFeature, one that has only FID. Actually it may be proxied, so if client code doesn't know about the existence of LazyFeatures, the feature instance could go fetch its contents to the back end, but if the client code knows about LazyFeatures, it may choose not to consume it, since it knows the Feature contents were previously acquired.

GML encoding sketch

```
FeatureIterator features = ...
while(features.hasNext()){
    Feature f = features.next();
    encodeFeature(f);
}
features.close();
...

private void encodeFeature(Feature f){
    FeatureType schema = f.getFeatureType();

    if( f instanceof LazyFeature ){
        //just writes <propertyName xlink:href="id"/>
        encodeByReferenceFeature(schema, f.getID());
    }else{
        startFeature(schema, f.getID());
        AttributeType []types = schema.getAttributeTypes();
        encodeAttributes(f.getAttributes(), types);
        endFeature(schema);
    }
}

private void encodeAttributes(Object[] values, FeatureType
schema){
    for(int i = 0; i < values.length; i++){
        if(schema.getAttribute(i) instanceof FeatureAttributeType){
            encodeFeature((Feature)f);
        }else{
            //.... encode simple types, etc, etc
        }
    }
}
```

Output

```
<wfs:FeatureCollection>
  <gml:boundedBy>...

  <featureMember>
    <City gml:id="city.1">
      <name>Rosario</name>
      <intersects>
        <Road gml:id="road.1">
          <name>Ruta 9</name>
        </Road>
      </intersects>
    </City>
  </featureMember>

  <featureMember>
    <City gml:id="city.2">
      <name>Buenos Aires</name>
      <intersects xlink:href="#road.1"/>
    </City>
  </featureMember>

  <featureMember>
    <City gml:id="city.3">
      <name>Cordoba</name>
      <intersects xlink:href="#road.1"/>
    </City>
  </featureMember>

</wfs:FeatureCollection>
```

Benefit: such an approach may ease the job of cross DataStore joins, by not having to do a full join at all, if you can guarantee or force that the "foreign key" on your master table is the FID of the features on the slave table.

Multiple namespaces

As in XML Schema, namespaces are used in GML schemas to relate to externally defined types. This is extremely important for building *community schemas*, which aims for *semantic* interoperability instead of just common protocols.

Suppose in your country's *transportation* organization has elaborated a schema for defining inter-agency interchange of transportation information. By the other hand, your province admin urbanism department has a schema modeling the urban infrastructures, which includes information of the roads from the transportation schema, for each city:

```

<urb:City>
  <urb:cityName>Bilbao</urb:cityName>
  <tr:roadProperty>
    <tr:Road>
      <tr:roadName>Kastrexana</tr:roadName>
      <tr:centerLine>
        <gml:LineString>....</gml:LineString>
      </tr:centerLine>
    </tr:Road>
  </tr:roadProperty>
  <tr:roadProperty>
    <tr:Road>
      <tr:roadName>A8</tr:roadName>
      <tr:centerLine>
        <gml:LineString>....</gml:LineString>
      </tr:centerLine>
    </tr:Road>
  </tr:roadProperty>
</urb:City>

```

In this example, the `urb` namespace prefix belongs to the feature types defined in a hipotetical urbanism schema, and `tr` namespace prefix to the ones defined in the hipotetical transportation schema.

The urbanism schema may look something like this:

hipotetical urbanism schema

```

<xs:schema
targetNamespace="http://myprov.org/urbanism"
  xmlns:urb="http://myprov.org/urbanism"

```

```
xmlns:tr="http://mycountry.org/transportation"
```

```
xmlns:gml="http://www.opengis.net/gml"
```

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="1.0">
```

```
  <xs:import
namespace="http://www.opengis.net/gml"
schemaLocation="../schemas.opengis.net/gml/3
.1.1/base/feature.xsd" />
```

```
  <xs:import
namespace="http://mycountry.org/transportation"
schemaLocation="transportation.xsd" />
```

```
  <xs:complexType name="CityType">
    <xs:complexContent>
      <xs:extension
base="gml:AbstractFeatureType">
        <xs:sequence>
          <xs:element name="cityName"
type="xs:string"/>
          <xs:element name="roadProperty"
minOccurs="0" maxOccurs="unbounded"
type="tr:RoadPropertyType"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

```
<xs:element name='City'
```

```
type='urb:CityType'  
substitutionGroup="gml:_Feature" />  
</xs:schema>
```

There is a problem, though, that arises from an attribute being able of belonging to a different namespace: ⚠️ two attributes may have the same "local" name, and different namespaces.

For example:

```
<element name="nameAttribute">  
  <complexType>  
    <choice>  
      <element name="name" type="xs:string"/>  
      <element ref="gml:name"/>  
    </choice>  
  </complexType>  
</element>
```

Despite its aparent unusefulness, it's a completely valid example.

An instance of this schema fragment can be either

```
<myNs:nameAttribute>  
  <myNs:name>theName</myNs:name>  
</myNs:nameAttribute>
```

or

```
<myNs:nameAttribute>  
  <gml:name>theName</gml:name>  
</myNs:nameAttribute>
```

now try to do `feature.setAttribute("nameAttribute", "theName")....`

💡 it may be the case that a single attribute name is not enough to set an attribute value, we might need QNames (qualified names). So it may worth taking a look at GeoAPI's [GenericName](#)

By the other hand, 🤔 what would be really consistent:

```
1. feature.setAttribute("nameAttribute", "theName")
```

```
2. feature.setAttribute("nameAttribute/gml:name", "theName")
```

```
3. feature.setAttribute("nameAttribute", Arrays.asList( new String[]{null,
    "theName"} ) )
```

```
4. AttributeType type =
    feature.getFeatureType().getAttributeType("nameAttribute").getAttributeType("http://www.opengis.org/gml", "name");
    feature.setAttribute(type, "theName" )
```

Example #1 does not solve the problem, it's ambiguous.

Example #2 may be enough, provided that at least [relative location paths](#) are directly supported.

i Example #3 exemplifies the current approach for dealing with complex types: **every** complex type value is a `java.util.List`. Note it has some inconveniences, though: It *partially* addresses the ambiguity described above, since attribute values are *order dependant* of the declaration of child types, **BUT**: this order dependence is only applicable to the semantics of a **sequence** construct. For a choice, the value might be a single Object (though it is managed as being always a singleton List?), and for an **all** construct there is no way, since the semantics of *all* means its children can appear in *any* order; so how do you assign a value to one of the name attributes in the example above?

Example #4 might be safer: it doesn't require intrinsic XPath support and is namespace aware, but there is a strong dependence of the internal data structure.

Now let's try the inverse case: getting the attribute value:

```
1. Object value = feature.getAttribute("nameAttribute")
```

```
2. Object value1 = feature.getAttribute("nameAttribute/gml:name")
   Object value2 = feature.getAttribute("nameAttribute/myNs:name")
```

```
3. List values = (List) feature.getAttribute("nameAttribute")
```

```

4. AttributeType type1 =
   feature.getFeatureType().getAttributeType("nameAttribute").getAttributeType("http://www.opengis.org/gml", "name");
   AttributeType type2 =
   feature.getFeatureType().getAttributeType("nameAttribute").getAttributeType("http://www.myNs.org", "name");
   Object val1 = feature.getAttribute(type1);
   Object val2 = feature.getAttribute(type2);

```

To evaluate the usefulness of those constructs, let's suppose we have to encode a GML document based on the previous reading examples:

Example #1: you got the value. Which one?

Example #2: eh... I'm sure I would not use that API.

Example #3 (aka, how things work right now): again, ok for order dependant attributes, but what for *all*? you got List[null, "theName"], what's null? gml:name or myNs:name?

Example #4: apart of looking a bit burdened, it may work, except that you need a deep pre-knowledge of the whole structure, and it may make a lot more of sense to be able of just asking for the value of "nameAttribute" and being able of knowing which of its sub elements the value corresponds to.

It seems, like in the XML world, that a *value* has no sense outside its context (i.e. its element name, on our case, its attribute).

✓ What about....

```

//a tuple for handling attribute values
interface AttributeValue{
  /** which attribute type the value belongs to */
  AttributeType getType();

  /**
   * Returns the value of the attribute instance.
   * The value of the attribute instance is defined to be:
   * - a List of actual values if getType().getMaxOccurs() > 1
   * - an actual value otherwise.
   * An actual value is defined to be:
   * - an AttributeValue if the value's type is complex
   * - a value literal if the value's type is simple
   */
  Object getValue();

  void setValue(Object value);
}

```

Feature.java

```
interface Feature{
    ...
    /** value.getType() may address a leaf attribute (aka, a
SimpleAttributeType),
    * or a complex attribute, in which case value.getValue() will
be another
    * AttributeValue up to the nesting level desired.
    */
    void setAttribute(AttributeValue value);

    /** attName must address a leaf attribute */
    void setAttribute(GenericName attName, Object value);

    AttributeValue getAttribute(GenericName attName);

    /** returns a literal if att is simple, or an AttributeValue
    * if type is complex
    */
    Object getAttribute(AttributeType att);
}
```

Though this example is far from being a complete model, its intention is to show that:

- A Feature instance, as a complex structure, actually holds a sequence of tree like attribute values
- Those attribute values have to be *typed*
- The `AttributeValue` tuple defines by itself a tree structure of typed values that:
 - uniquely identifies a node in the hierarchy
 - avoids ambiguity in the interpretation of the value's type

FeatureCollection's Feature API

A `FeatureCollection` is defined in GML as a **Feature** of type `FeatureCollectionType`, which in turn extends `AbstractFeatureType`.

Said that, the notably thing is that a `FeatureCollection` does not differs from any other derived feature type that you may define in order to allow the holding of any number of other `Feature` instances.

But if you need such a thing, the recommendation is to use a `FeatureCollection` instead of reinventing the wheel, since that's why there are so many predefined elements and types in GML.

So, as `FeatureCollection` is already well defined, it has a well defined schema:

```

abstract AbstractFeatureCollectionType
extends AbstractFeatureType
  featureMember (0..N)
  featureMembers (0..1)

FeatureCollectionType extends
AbstractFeatureCollectionType

```

where featureMember is the FeatureCollection's AttributeType (following the gml:AssociationType pattern) that allows it to hold *any* number of features of *any* FeatureType, and featureMember*s* may be used to hold any number of features without enclosing each one on a featureMember element (since it is an array of features).

The difference between this two ways of holding features on a feature collection is not just the saving of space in encoding (one featureMember element for each Feature vs just one featureMembers element for the whole document), but the ability to reference a remote feature using XLinking that featureMember has against the featureMembers array.

With all this small background in mind, what worths being said is that you can use a FeatureCollection the same way as you use a Feature instance, that a FeatureCollection's member may indeed be another FeatureCollection, and so on.

💡 The other notably thing is, as long as GML is intended as a basis for defining specific application domain schemas, its obvious that you can extend (actually restrict) the value space of a FeatureCollection so its schema decalres what exactly are the allowable Feature types for the collection.

📘 So keep in mind the previous statement for further discussion, as it seems there is some confusion (it could be me, of course):

- a FeatureCollection, as defined, allows for the containment of **any** type of Feature instances
- you **can** restrict which ones to allow, but this requires the definition of your own restricted FeatureCollectionType, deriving from the base FeatureCollection type

What about GeoTools FeatureCollections ?

Well, obviously it would be desirable to treat a GeoTools FeatureCollection instance as a single Feature in a number of situations, *though it is possible nowadays*, the following is a list of current situation:

1. 🌟 FeatureCollection properly requires that its schema is of gml:AbstractFeatureCollectionType type
2. 🌟 FeatureCollection is thought on the basis of GML < 3.1.1, where gml:AbstractFeatureCollectionType descends from gml:BoundedFeatureType, making boundedBy property mandatory. **This is no more the case** as for GML 3.1.1, which we should target, as it is the first version that completely validates and the one that will drive future developments in the short term.
3. 🌟 it may act as a derived FeatureCollection, due to the ability of knowing the FeatureType of its contained Feature instances (note the distinction between the [#FeatureCollection schema](#) and the schema of its contained features). This ability is useful since most the time you will make a query over a DataStore's FeatureType and obtain a FeatureCollection as result, all of which are of the same type. 📘 So in this case we're gracefully extending gml:FeatureCollection adding this extra behavior: *restricting its value space*. The

way to notice the existence of this restriction is if `getSchema():FeatureType` does not returns null. \(\on) So a more usefull extention may be the ability to restrict the allowable features to more than one `FeatureType`, by replacing

```
getSchema():FeatureType
```

by

```
getAllowedMemberTypes():Set<FeatureType>
```

4. ★ The current Feature API implementation for `FeatureCollection` **does not** respects the association type pattern:
 - a. It acts like a `gml:FeatureArrayPropertyType` itself, by allowing to query a member as `getAttribute("typeName")`, where "typeName" is the ame of a child `FeatureType`.
 - b. In order to complain with the `FeatureCollectionType` definition, it should be: `getAttribute("featureMember")`, which may return a `java.util.List<Feature>`, since the `featureMember` as sociation multiplicity is (0..N).
 - c. ⚠️ **BUT** a derived `FeatureCollection` is not mandated to have `featureMember(s)` properties. The point of being able to derive them is, apart of being able of restricting its members to a given type(s), **to be able of calling the member property as you like**. For example, you can define a `RiverCollection` that derives from `FeatureCollection`, whose association attribute is called `riverMember` instead of `fetureMember`.

Conclusion

- It is clear that for simplicity of use, programmers should be able of using a `FeatureCollection` instance through a convenient interface, like `FeatureCollection.features():FeatureIterator`
- But, if we're going to be able of using a `FeatureCollection` through its Feature API, we'll need:
 1. 💡 a `FeatureCollectionType`, as well as we have a `FeatureType`
 2. 💡 a convenient way of restricting its members to one or more `Feature` types
 3. 💡 a convenient way of redefining the name of its feature association attribute name (like for replacing `featureMember` by `riverMember`)

Object Identity

From GML 3.1.1 spec, page 23: (paragraph numbers are mine)

"(1) A GML object is an XML element of a type derived directly or indirectly from AbstractGMLType. From this derivation, a GML object may have a gml:id attribute.

(2) A GML property may not be derived from AbstractGMLType, may not have a gml:id attribute, or any other attribute of XML type ID.

(3) An element is a GML property if and only if it is a child element of a GML object.

(4) No GML object may appear as the immediate child of a GML object.

(5) Consequently, no element may be both a GML object and a GML property.

(6) NOTE In this version of GML, the use of additional XML attributes in a GML application schema is discouraged."

Implications:

(1) Identity: apparently our type system may be able of dealing with identity beyond Feature. Any complex type that you define may inherit from AbstractGMLType or not. If it does, it *may* have identity, as well as metadataProperty, description and name.

(2): that's what a simple Attribute and a Complex one means

(4): that's where the gml:AssociationType pattern comes from. Currently, when we encode a GeometryAttribute to GML, we respect that rule just because we *already know* how geometries should be encoded. 💡 Adding the ability to know if a complex attribute "*is identified*" makes it more explicit and allows for user defined types to derive from AbstractGMLType

(6) that's consistent with the Object/property rule.

(5) the following is a real world example:

```
<sco:wq_plus gml:id="_41010901">
  <sco:sitename>BALRANALD
WEIR</sco:sitename>

<sco:anzlic_no>ANZNS0359100023</sco:anzlic_n
o>
  <sco:location>
    <gml:Point
srsName="http://www.opengis.net/gml/srs/epsg
.xml#4283">
      <gml:pos>22 143.53399658</gml:pos>
    </gml:Point>
  </sco:location>
  <sco:measurement
gml:id="_16JAN94002001002003000000">

<sco:determinand_description>16/JAN/94</sco:
determinand_description>
  <sco:result>Turbidity</sco:result>
</sco:measurement>
  <sco:measurement
gml:id="_24JAN94002001002003000000">

<sco:determinand_description>24/JAN/94</sco:
determinand_description>
  <sco:result>Turbidity</sco:result>
</sco:measurement>
  <sco:project_no>RWWQ0004</sco:project_no>
</sco:wq_plus>
```

Given (5), `<sco:measurement>` is apparently wrong, since it is both a property and a GML Object (its type derives from `AbstractGMLType`, or its badly defined, because of (1) and (6)).

💡 So that's the requirement, adding a identity capable complex attribute, since its clear that not only features may have id, there are plenty of them in the GML spec (for example, `Geometry`), and a user should be able of defining its own complex type with identity.

Association type

From the above points, we had learned that the `gml:AssociationType` pattern is widely used. It is a container property for a complex attribute that's generally defined externally to the `FeatureType` itself, like a `Feature`, a `Geometry`, etc.

More than that, **every time** you have to refer to an extenally defined entity (those than in GML sense derives from `AbstractGMLType`, and thus *may* have identity), you **should** use a container (association) property to refer to such an externally defined entity.

By the way, we do have a small set of well known association types that we've already treating as such: `FeatureAttributeType` and `GeometryAttributeType`.

Now we found that the concept is extensible to *any* entity defined externally to a given `FeatureType`. Note this is different from having a `FeatureType` property that has a complex structure by itself.

💡 So there exists the need to be able of explicitly modeling a property which acts as a container of an entity whose type is defined externally to a given `FeatureType`, wether such an entity is or not a `Feature`, it could be any kind of complex type (like a topology type).

Current requirements support report

Simple types support report

The goal of the following table is to expose if the current API is able of expressing the needed simple types.

This evaluation does **not** takes in count if they check or not that the restrictions applied through a `Filter` adheres to each type allowed restrictions, since for all the current `AttributeTypes` there is not this kind of validation. Thus, adding this kind of validation will be the subject of another chapter by itself.

What this table does exposes is if there are a current `AttributeType` mapping for each primitive type.

Type	Level of support	Description
list	★	Not supported. Passing <code>List.class</code> as type argument for <code>DefaultAttributeType</code> will not be enough. list data types needs to know the type of its elements, and the type may be a derived atomic one, so it has to deal with the item type restrictions, etc. 💡 list data types may need their own <code>AttributeType</code>

union	★	<p>Not supported</p> <p>💡 union will definitely need its own AttributeType.</p> <p>⚠️ it may be too difficult to parse a literal to the correct member type! (think of a union of nonNegativeInteger and decimal, you receive a Integer(0), how to decide?)</p>
string	★	TextualAttributeType fully handles plain strings
boolean	★	<p>When a Boolean attribute is needed, we simply use DefaultAttributeType passing it Boolean.class as its type. Since we're going stronger,</p> <p>💡 boolean may need its own AttributeType implementation, at least to deal with allowed restrictions?</p>
decimal	★	it may be specified by passing BigDecimal.class as the type of a NumericAttributeType
float	★	passing Float.class as the type of a NumericAttributeType
double	★	passing Double.class as the type of a NumericAttributeType
duration	★	<p>duration can be stored as an instance of java.util.Date, thus mapping to TemporalAttributeType.</p> <p>💡 But we need a way of distinguishing the exact XML Schema type given an instance of TemporalAttributeType. Otherwise we'll not be able of correctly encoding a GML schema from a FeatureType instance</p>
dateTime	★	its the current exact match for TemporalAttributeType
time	★	same as duration
date	★	same as duration

gYearMonth	★	same as duration
gYear	★	same as duration
gMonthDay	★	same as duration
gDay	★	same as duration
gMonth	★	same as duration
hexBinary	★	has no current mapping
base64Binary	★	has no current mapping
anyURI	★	has no current mapping, though it may be inferred by using a <code>DefaultAttributeType</code> with <code>URI.class</code> as its type
normalizedString	★	may map to <code>TextualAttributeType</code> , but it has no way of enforcing or validating a <code>normalizedString</code>
token	★	same as above
ID	★	same as above
integer	★	supported by passing <code>BigInteger.class</code> as the type of a <code>NumericAttributeType</code>
nonPositiveInteger	★	<code>NumericAttributeType</code> has no way of validating the type value space, and in such a case, there is no way of differentiating a positive from a negative type since there is no direct equivalences to Java classes and <code>NumericAttributeType</code> relies on the <code>Class</code> passed as its constructor argument to set the type. 💡 We need to extend <code>NumericAttributeType</code> in a way that all the XML Schema basic type derived from decimal could be mapped with no ambiguities
negativeInteger	★	same as above

long	★	Exact match for NumericAttributeType with Long.class as type argument
int	★	Exact match for NumericAttributeType with Integer.class as type argument
short	★	Exact match for NumericAttributeType with Short.class as argument
byte	★	Exact match for NumericAttributeType with Byte.class as argument
nonNegativeInteger	★	same as nonPositiveInteger
unsignedLong	★	same as nonPositiveInteger
unsignedInt	★	same as nonPositiveInteger
unsignedShort	★	same as nonPositiveInteger
unsignedByte	★	same as nonPositiveInteger
positiveInteger	★	same as nonPositiveInteger

Level of support legend:

- ★ *unsupported*
- ★ *partially supported*
- ★ *supported*

Restriction support

As each simple type has its own set of allowed restrictions, the first step is to ensure that all the possible restriction types can be represented. The current approach to define a restriction on an AttributeType is indeed very smart, since there exists already a powerful API to represent constraints on a data set: the Filter API, which is the GeoTools implementation of the Open Geospatial Consortium Filter 1.0 implementation specification.

The following is a table of the XML Schema [constraining facets](#) and how they could be represented by a GeoTools Filter. The power of this is such that, even if there is no predefined filter expression types to match a given facet, there is a huge chance that it still can be used by a Function expression.

Facet	Description	Restriction example	Comments
length			

enforces the length of value to be equal to the declared length. length is the number of units of length, where units of length varies depending on the type that is being - derived- from (#of chars for a string type, #of octets for a binary type, etc)

schema

```
<xs:simpleType
  name="_name">
  <xs:restriction
    base="xs:string">
    <xs:length
      value="10"
    />
  </xs:restriction>
</xs:simpleType>
```

that works for string based types. 💡 A new function expression should have to be created to check the length of binary data types

**matching
filter**

```
<Filter>  
  <Property  
    IsEqualTo>  
    <Function  
      name="str  
        Length">  
        <Property  
          Name>att_  
            name</Pro  
              pertyName  
                >  
        </Function  
          >  
        <Literal>  
          10</Liter  
            al>  
        </Proper  
          tyIsEqual  
            To>  
      </Filter>
```

[minLength](#)

enforces the length of attribute value to be lower than or equal to the non negative integer specified as the value of the restriction

schema

```
<xs:simpleType  
  name="att_  
    _name">  
  <xs:rest  
    riction  
      base="xs:  
        string">  
        <xs:minLe  
          ngth  
            value="10  
              "/>  
        </xs:res  
          triction>  
      </xs:simp  
        leType>
```

same as above

**matching
filter**

```
<Filter>
  <PropertyIsGreaterThanOrEqualTo>

  <Function name="StringLength" >

  <PropertyName>att_name</PropertyName>
  >

  </Function>
  >

  <Literal>
  10</Literal>
  </PropertyIsGreaterThanOrEqualTo>
</Filter>
```

[maxLength](#)

enforces the length of attribute value to be greater than or equal to the non negative integer specified as the value of the restriction

same as above

schema

```
<xs:simpleType
  name="attribute_name">
  <xs:restriction
    base="xs:string">

    <xs:maxLength
      value="10"
    />
  </xs:restriction>
</xs:simpleType>
```

**matching
filter**

```
<Filter>
  <PropertyIsLessThanOrEqualTo>

    <Function
      name="StringLength" >

      <PropertyName>att_
        name</PropertyName>
    >

  </Function>

  <Literal>
    10</Literal>
  </PropertyIsLessThanOrEqualTo>
</Filter>
```

[pattern](#)

specifies a regular expression that the attribute value must match to. **Note:** It is a consequence of the schema representation constraint [Multiple patterns](#) and of the rules for - restriction- that - pattern- facets specified on the same step in a type derivation are **OR** ed together, while - pattern- facets specified on different steps of a type derivation are **AND** ed together.

pattern applies to all basic types (temporal, string, boolean, numeric, binary and anyURI)
 may we assume that the string representation of each of type should be used to test pattern matching?
 verify that the XML Schema regular expressions are compatible with Java1.4 ones, or find a regular expression evaluation engine that does

schema

```
<xs:simpleType
  name="attribute_name">
  <xs:restriction
    base="xs:string">

    <xs:pattern
      value="[A-Z]*/>
    </xs:restriction>
  </xs:simpleType>
```

**matching
filter**

```
<Filter>
  <PropertyIsEqualTo>
    <Function
      name="strMatches">
        <PropertyName>att_name</PropertyName>
        <Literal>[A-Z]*</Literal>
      </Function>
      <Literal>true</Literal>
    </PropertyIsEqualTo>
  </Filter>
```

[enumeration](#)

constrains the - value space- to a specified set of values.

Thanks to Dave:
According to OGC Filter implementation, <Functions> **must** have a pre-defined number of arguments. You can always implement "in" with a set of "<Or>...<PropertyIsEqualTo>...".

schema

```
<xs:simpleType
  name="_name">
  <xs:restriction
    base="xs:string">

    <xs:enumeration
      value="option 1"/>

    <xs:enumeration
      value="option 2"/>
  </xs:restriction>
</xs:simpleType>
```

**matching
filter**

```
<Filter>
  <Or>

    <Property
      IsEqualTo
    >

      <Property
        Name>att_
        name</Pro
        pertyName
      >

        <Literal>
          option
          1</Litera
          l>

        </Propert
        yIsEqualT
        o>

      <Property
        IsEqualTo
      >

        <Property
          Name>att_
          name</Pro
          pertyName
        >

          <Literal>
            option
            2</Litera
            l>

          </Propert
          yIsEqualT
          o>

        </Or>
      </Filter>
```

whiteSpace

constrains the - value space- of types - derived- from string such that the various behaviors specified in [Attribute Value Normalization in XML 1.0 \(Second Edition\)](#) are realized. The value of whiteSpace must be one of {preserve, replace, collapse}

schema

```
<xs:simpleType
  name="_name">
  <xs:restriction
    base="xs:string">
    <!--
    one of
    collapse|
    preserve|
    replace
    -->

    <xs:whiteSpace
      value="collapse"/>
  </xs:restriction>
</xs:simpleType>
```

 no matching filter construct

maxInclusive

is the - inclusive upper bound- of the - value space. for a datatype with the - ordered- property

schema

```
<xs:simpleType
  name="date">
  <xs:restriction
    base="xs:date">

    <xs:maxInclusive
      value="2005-08-09"
    />
  </xs:restriction>
</xs:simpleType>
```

matching filter

```
<Filter>
  <PropertyIsLessThanOrEqualTo>

  <PropertyName>date_att</PropertyName>

  <Literal>2005-08-09</Literal>
</PropertyIsLessThanOrEqualTo>
</Filter>
```

applies to all numeric and temporal types. Fully supported for numeric types.

⚠ proper support for temporal types may need that the comparisons be aware of the value space. i.e., avoiding to compare the "fields" that does not apply to the concrete temporal type, for example, comparing only the month and day parts of a gMonthDay type, ignoring the year and time. This could be easily done with `java.util.Calendar` and knowledge of the concrete temporal type

[maxExclusive](#)

is the - exclusive upper bound- of the - value space- for a datatype with the - ordered- property

schema

```
<xs:simpleType
  name="date_att">
  <xs:restriction
    base="xs:date">

    <xs:maxExclusive
      value="2005-08-09"
    />
  </xs:restriction>
</xs:simpleType>
```

matching filter

```
<Filter>
  <PropertyIsLessThan>

    <PropertyName>date_att</PropertyName>

    <Literal>2005-08-09</Literal>
  </PropertyIsLessThan>
</Filter>
```

same as above

[minExclusive](#)

is the - exclusive lower bound- of the - value space- for a datatype with the - ordered- property

schema

```
<xs:simpleType
  name="date_att">
  <xs:restriction
    base="xs:date">
    <xs:minExclusive
      value="2005-08-09"
    />
  </xs:restriction>
</xs:simpleType>
```

matching filter

```
<Filter>
  <PropertyIsGreaterThan>
    <PropertyName>date_att</PropertyName>
    <Literal>2005-08-09</Literal>
  </PropertyIsGreaterThan>
</Filter>
```

same as above

[minInclusive](#)

is the - inclusive lower bound- of the - value space- for a datatype with the - ordered- property

schema

```
<xs:simpleType
  name="date_att">
  <xs:restriction
    base="xs:date">
    <xs:minInclusive
      value="2005-08-09"
    />
  </xs:restriction>
</xs:simpleType>
```

matching filter

```
<Filter>
  <PropertyIsGreaterThanOrEqualTo>
    <PropertyName>date_att</PropertyName>
    <Literal>2005-08-09</Literal>
  </PropertyIsGreaterThanOrEqualTo>
</Filter>
```

same as above

<p>totalDigits</p>	<p>controls the maximum number of values in the - value space- of datatypes - derived- from decimal, by restricting it to numbers that are expressible as $i \times 10^n$ where i and n are integers such that $i < 10^{\text{totalDigits}}$ and $0 \leq n \leq \text{totalDigits}$. The value of totalDigits must be a positiveInteger.</p>	<div style="border: 1px dashed blue; padding: 10px;"> <p style="text-align: center;">schema</p> <pre><xs:simpleType name="decimal_att" > <xs:restriction base="xs:string"> <xs:totalDigits value="10" /> </xs:restriction> </xs:simpleType></pre> </div>	<p> no matching filter construct.</p> <p> a new function expression should be easily created following the facet rules</p>
<p>fractionDigits</p>	<p>controls the size of the minimum difference between values in the - value space- of datatypes - derived- from decimal, by restricting the - value space- to numbers that are expressible as $i \times 10^n$ where i and n are integers and $0 \leq n \leq \text{fractionDigits}$. The value of fractionDigits must be a nonNegativeInteger.</p>	<div style="border: 1px dashed blue; padding: 10px;"> <p style="text-align: center;">schema</p> <pre><xs:simpleType name="decimal_att" > <xs:restriction base="xs:string"> <xs:fractionDigits value="10" /> </xs:restriction> </xs:simpleType></pre> </div>	<p> no matching filter construct.</p> <p> a new function expression should be easily created following the facet rules</p>

Complex types support report

Requirement	Level of support	Description
Multiple geometric attributes		

support for multiple geometric attributes was in there from the inception of the GeoTools Feature API

💡 Though we should make sure that a complex attribute (not a Feature association) that contains a GeometricAttribute shall be treated as well?? For example:

```
<xs:complexType  
name="MyFeatureTy  
pe">
```

```
<xs:complexContent  
>
```

```
  <xs:extension  
base="gml:Abstrac  
tFeatureType">
```

```
<xs:sequence>
```

```
<xs:element  
name="the_geom"  
type="gml:PointPr  
opertyType"/>
```

```
<xs:element  
name="aComplexAtt  
">
```

```
<xs:complexType>
```

```
<xs:sequence>
```

```
<xs:element  
name="complexAttN  
ame"  
type="xs:string"/  
>
```

```
<xs:element  
name="comlexAttGe  
om"  
type="gml:LineStr  
ingPropertyType"/  
>
```

```
</xs:sequence>
```

```
</xs:complexType>
```

```
</xs:element>
```

```
</xs:sequence>
```

```
</xs:extension>
```

```
</xs:complexConte  
nt>
```

```
</xs:complexType>
```

		<p>❓ have both MyFeatureType/the_geom and MyFeatureType/aComplexAtt/complexAttGeom to be taken in count?</p>
choice	★	<p>Current API has initial support for the different content models defined in XML Schema. Of the three, <i>choice</i> is the one that is more tested, though the following considerations should be noted for all of them:</p> <p>(!)The current AttributeType model is reaching its limit for modeling a complex feature class. The property values of a complex feature instance are structured in a tree like fashion, where each "node" should be addressable by a expression language (like an XPath's <i>location path</i>), and its value, either <i>simple</i> or <i>complex</i> must not only respect the type's value space, but also must be structured in a way that respects its <i>content model</i>.</p> <p>💡 We have found that explicitly separating both concerns, type system and content model (or <i>schema</i>) may lead to a simpler and more powerful API</p>
sequence	★	<p>Currently, ListAttributeType is meant to represent a <i>sequence</i> content model and define the type of the allowed content, though the same considerations than for <i>choice</i> support applies.</p>
all	★	<p>The current SetAttributeType is intended to reflect the <i>all</i> content model, though its implementation still has order dependant code that makes difficult to ensure the content model is respected by a value instance.</p>

Nested Features	★	There exists a <code>FeatureAttributeType</code> to denote a nested feature property, though the lack of <code>DataStore</code> implementations that can make use of it limits the ability to evaluate its appliance to real world use cases.
Multiple namespaces	★	currently there are no support for a Feature property belonging to a different namespace than the Feature's <code>FeatureType</code> one
FeatureCollection's Feature API	★	if we're going to be able of using a <code>FeatureCollection</code> through its Feature API, we'll need: <ol style="list-style-type: none"> 💡 <code>FeatureCollectionType</code>, as well as we have a <code>FeatureType</code> 💡 convenient way of restricting its members to one or more Feature types 💡 convenient way of redefining the name of its feature association attribute name (like for replacing <code>featureMember</code> by <code>riverMember</code>)
Object Identity	★	The current API has no knowledge of object identity beyond <code>Feature</code>
Association type	★	Currently there exist a <code>NestedAttributeType</code> which is intended to reflect that it is referring to an externally defined object type, but: <ol style="list-style-type: none"> ⚠️ it has no knowledge of identity ⚠️ it derives from <code>ListAttributeType</code>, which limits its content model to the equivalence of a XML Schema sequence It seems it's not being used across the GeoTools code base

Enhancement requirements

Supporting all the features mentioned above seems like a lot of work. We have, though, a driven specification that will help limit scope. Actually it's an OGC discussion paper, but is what the project stakeholders are interested in for the short time: [Level 0 Profile of GML3 for WFS](#).

Minimum requirements: Level 0 Profile of GML3 for WFS

The intent of that document is to specify the encoding of application schemas sufficiently so that WFS client implementations do not need to deal with the entire scope of XML-Schema and GML but only need to understand a restricted subset of both specifications in order to be able interpret schema documents generated in response to a DescribeFeatureType request.

Among other things, it defines the essential geometry and simple types that a system must support to achieve a minimum level of interoperability.

From the mentioned discussion paper, table 4, page 21, the following are the allowed geometry types:

GML Geometric Property Type	Defined in GML Schema File	Restrictions
<code>gml:PointPropertyType</code>	<code>geometryBasic0d1d.xsd</code>	none
<code>gml:CurvePropertyType</code>	<code>geometryBasic0d1d.xsd</code>	only <code>LineString</code> allowed as value
<code>gml:SurfacePropertyType</code>	<code>geometryBasic2d.xsd</code>	only <code>Polygon</code> allowed as value
<code>gml:MultiCurvePropertyType</code>	<code>geometryAggregates.xsd</code>	only <code>MultiLineString</code> allowed as value
<code>gml:MultiSurfacePropertyType</code>	<code>geometryAggregates.xsd</code>	<code>MultiPolygon</code> and <code>MultiSurface</code> allowed as value; <code>MultiSurface</code> can use only linear (sub)geometries
<code>gml:LinearRingPropertyType</code>	<code>geometryBasic2d.xsd</code>	this was missing in GML- 2 and has justifiably been added to GML- 3
<code>gml:RingPropertyType</code>	<code>geometryPrimitives.xsd</code>	only <code>LinearRing</code> or <code>Ring</code> with <code>LineStrings</code> can appear as value

That's good, we already handle most of them.

As for simple types, section 7.5.2 Basic Data Types, page 26, limits the set of available basic types to a smaller subset. The rationale being that a smaller common set of supported basic data types is likely to be more interoperable.

The list of supported basic data types is:

1. Integers, limited to *integer* data type with no limit on applicable facets
2. Reals, limited to *decimal*, *float* and *double* and *totalDigits* and *fractionDigits* facets
3. Character strings, to *string* type and *maxLength* facet
4. Date, limited to *date* and *dateTime* data types with no restriction on facets
5. *boolean* data type
6. Binary data, both *base64Binary* and *hexBinary*, with no restriction on facets, and the aggregate of the attributes *url*, *mimeType*, and *role*, as specified in the following fragment:

```

<xs:element name="propertyName" minOccurs="0|N" maxOccurs="0|N|unbounded">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:base64Binary|xs:hexBinary">
        <xs:attribute name="url" type="xs:anyURI" use="optional"/> <!-- when
referenced from an external URI -->
        <xs:attribute name="mimeType" type="xs:string" use="required"/> <!--
must be specified to indicate the type or format of binary data that is
being
referenced -->
        <xs:attribute name="role" type="xs:string" use="optional"/> <!-- can
be used to assign a user-defined
role to the data. The role attribute allows complex binary format s like
HDF/EOS,
which contains multiple independent binary component s, to be supported.
-->
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

```

💡 so we may be able of coming back with a flexible design that addresses this essential requirements and leaves place for the wider implementation of the rest.

⚠️ Be careful

GML Level 0 profile is about **simple features**, so the explanation above applies only to the **simple attributes** support requirements. We **still need** to support multiple geometric attributes, related features, and complex attribute types.

Requirements break down:

Simple support for simple types

It is **not** a requirement (at least for this phase of the project) to cover all the mappings between simple types to the predefined XML Schema simple types.

It is enough to count with a well known set of java to xml bindings, like Integer->xs:int, String->xs:string, Date->xs:date, etc.

There is no need of having mappings to XML Schema `list` and `union` simple types.

Keep it simple.

The majority of available data sources are producers of Simple Features. So the API should be kept simple enough that it adds no significant complexity for simple cases.

Separation of concerns

We have discovered that the current approach has difficulties to express, in a single type hierarchy, what the valid

structure of a type instance is, and what's the exact type of each of the components of such an instance. The new model must be able of:

1. specify the type (name, value range, nullability) of a property
2. state the content model of a type (multiplicity and tree structure)
3. provide an unambiguous containment model

Multiple namespaces

A type always belongs to a namespace. A nested object type may belong to a different namespace than its parent.

Identity beyond Feature

Not only Feature instances has identity (for Feature, it is its FID). It should be possible to define a type whose values has a unique identifier inside its namespace. These are always complex types.

A common case are Geometries. Two or more Features may share the same Geometry. In GML, Geometries are allowed to have an id. So both Features may share the same geometry value, by reference.

Make explicit the association type pattern.

A Feature property follows the association type pattern. A Geometry property too. In fact, all Feature attributes whose type is defined externally to the feature's type, or internally but may have identity, should be referenced by an association element. That's what will allow to reference a remote value instance.

Derived types

All the types that can be defined are derived from another one. If it is a simple type, it derives from a primitive type. If it is a complex type, either it derives from `gml:_Object`, or if it may have identity, derives directly or indirectly from `gml:AbstractGMLType`.

There are some well known abstract supertypes for well known type hierarchies, like for Geometries and Features.

Type inheritance is limited to single inheritance.

Setting a requirement for a type to return its super type (i.e. `getParent():Type`) may be overkill, since due to cross namespace inheritance, it would be the task of a **repository** of types to obtain a type instance. **But** due to the use of qualified names to identify types, a given type **may be able of returning the qualified name of its parent**.

Keep expressions out, make an easily navigable model.

Do not stick the modeling power of the API to XPath. XPath is an expression language and there can exist others. We must provide an API that makes easy the implementation of an expression language against it.

Give FeatureCollection its place

FeatureCollection is a well known construct, with a well known type, as well as Geometry and its derivatives are. The same way you can define your own type to hold features, you could define your own type to hold a geometry. Instead of having to do that, GML provides these predefined constructs to ease the job of GIS users, so lets integrate FeatureCollection in the modeling system in a way that it is not needed to have a special treatment to deal with them, but simply using its Feature API.

Make restrictions implementation independent

The restrictions on the value space of a simple type is given by its "facets".

The restrictions on the value space of a complex type is given by its "content model".

By the other hand, the idea of directly expressing these restrictions through the Filter API may or may not satisfy the whole spectrum of needs. It does, predefined function names should be defined to treat facets, and the Filter API may need to be extended to deal with validation of the content model for complex types. More than that, it would be desirable to validate a type instance without having a complete Feature instance, which the current Filter API does not allow.

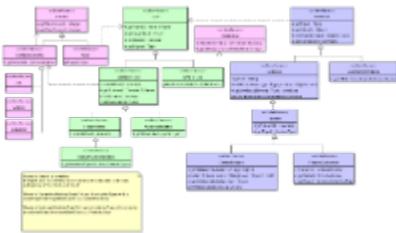
This may impose an unacceptable burden to the API making it very hard to implement.

Instead, the restrictions on a type's value space could be expressed by a simple API and give implementations the freedom to validate them as they like.

Enhancements proposal

This is an API change proposal to address the previously stated needs.

Feature API architecture proposal



Refer to the [Feature Model Design Discussion](#) document, and specially to the [Discussion](#) section for an insight on the process the lead to this proposal.