# Data Model for Quantity
## Version 0.2

## IVOA DM WG Internal Draft
## 2004-03-01

**Working Group:** http://www.ivoa.net/twiki/bin/view/IVOA/IvoaDataModel
**Editors:**
Jonathan McDowell, David Berry, Patrick Dowler, Brian Thomas
**Authors:**
IVOA Data Model Working Group

## Abstract

This document defines a Quantity data model.

## Status of this document

This is a Working Group Internal Draft only. It is inappropriate to reference this document.

## Acknowledgments

Members of the IVOA Data Model Working Group, including representatives of the US NVO, Astrogrid, Starlink, the Canadian VO, and the AVO have contributed to the present draft.

## Contents

# 1 Introduction and Scope

In this document we define a VO data model to describe the semantic content of sets of astronomical data values and their most closely associated metadata. The model may be used by aggregation or extension in higher level models describing astronomical datasets; the serializations of these models will allow data providers to consistently describe their data to the VO and the implementations of the data model classes in software will simplify the development of tools to manipulate VO data.

Astronomical data consists of string and numerical (floating point and integer) values and arrays of such values, arranged and related in complicated ways. Any numerical value must be associated with a physical concept (e.g. 'radial velocity of a spectral line'), which can be tagged as a UCD, and with a physical unit (e.g 'km/s'), by which we mean to include nonstandard cases such as 'dimensionless' and 'pixel'.

In this document we define a set of interfaces of modest complexity to an object called Quantity and to some related objects. Our intent is that the eventual VO data access layer will be coded in terms of these interfaces. Higher level objects will also be defined; we do not require Quantity to do everything. In particular, we are developing a model provisionally called Observation to describe a 'complete' (in a sense to be defined) dataset.

By choosing to define interfaces, we concentrate on what one can do and how one uses a Quantity, rather than on how they are or might be stored. Storage is, of course, more than just a detail of the numerous implementations that are sure to follow such a design - the external serializations are the core standards that give us interoperability. Nevertheless, by aiming at an interface of moderate complexity as our core definition, we enable applications software to have a relatively simple target to develop against, at the expense of quantity implementors, who will see some overhead cost when the data are simple and they must still implement methods that do nothing. The cost of leaving something out of Quantity is that applications which need that something will have to implement a heavier object that contains it and the extra feature will not be portable *as an instance of quantity.*

However we also present an interface to BasicQuantity, intended as a simple, lightweight version that is useful for many applications.

The goal is to define interfaces such that the most frequent uses need most of the features and don't need other objects. If every time you operate on a value, you will be checking for an error (even if it might not be there), then the code will be simpler if value and error are in the same object. The second trick is that if a dataset or an application deals with a set of different objects in much the same way, it may be worth seeing if these different objects can be generalized as cases of a single thing. This is the motivation for combining numeric and string quantities.

The Quantity interface describes a way to access a physical measurement which may be:

- A single datum with a UCD and units
- A single datum with errors, UCD and units
- An array of data values with errors and common units
- array data with alternative representations and coordinate systems.

## 2 Use Cases

This is a summary of the use cases that are directly relevant to the design of a quantity interface. The complete collection of quantity use cases will be collected in a separate document. There is a great deal of variety in the scope of use cases and in how they combine various usage patterns to accomplish the overall goal. All of the use cases we have looked at generally indicate one or more of the following use patterns, in some combination:

- exchange data across the network
- combine data from different sources
- enable search/exploration/discovery
- enable analysis

The use cases include:

- Describing a scalar value and its context
- Manipulating and combining arrays while preserving units and checking compatibility (in the sense of UCDs)
- Describing the position of a pixel (cell) on an array axis
- Accessing data which may be either an array of explicit values or the values of a parameterized function.
- Comparing the contexts of values separately from the values themselves

## 3 Requirements

The use patterns described above appear to correspond to the following requirements[1] on the quantity model:

- a quantity instance must be serializable and standardized so it can be exchanged
- quantity must be flexible enough to contain/describe data from different sources in a uniform way so quantities can be integrated
- tools (quantity consumers) must be able to interact with multiple sources (data producers) at the same time

[1]Requirements are not use cases. Requirements are derived from use cases and are thus the design constraints that are *deduced* from the use cases. Each use case should require one or more features. If a use case has no corresponding feature(s), it is not possible to accomplish that task. If a requirement has no corresponding use case, it is unnecessary and should be removed.

- quantity must have a uniform interface that can be referenced symbolically so that searching can constrain any/all components
- quantity must have a uniform interface that can be invoked on instances to extract data and metadata
- quantity should support both explicit values and at least a simple set of functionally defined values.
- quantity should describe arrays of values sharing the same physical description.
- quantity should support the ability to describe the same data in several different coordinate systems.
- quantity should provide a Frame object which describes the context of values without containing values themselves
- The descriptions of coordinate axes should allow explicit listing of axis coordinate values (rather than an algorithmically defined or regularly spaced axis) without excessive overhead
- coordinate axes should support the concept of mappings as proposed by the Starlink team
- Coordinate descriptions on compound axes (the RA,Dec case) should be supported.

## 4 Quantity Concepts

The following concepts are involved in our quantity model. They are essentially the various nouns that people use in describing use cases.

### 4.1 Phenomenon

The Phenomenon is the thing being measured, reported, predicted or discussed. People tend to use the terms *property* or *attribute* as well since they are typically talking about a property or attribute *of* something. For example, the brightness of a galaxy is a property or attribute (of the galaxy); brightness is a phenomenon. It is useful at the modelling stage to keep this distinction in mind, even though in practice (i.e. software) we rarely if ever deal with the phenomenon by itself. If one thinks of a quantity as a *name-value* pair, this is the *name* part and its semantics are given by the UCD system.

It is useful to be able to distinguish between the temperature of the source (the star is at 5000 degrees) and the temperature of the instrument (the CCD was at 220 K). The VO UCD group is developing a language in which to capture such distinctions, and will allow us to distinguish between the general phenomenon (temperature), the subject of the phenomenon (star), and the qualified phenomenon (temperature of the star).

## 4.2 Value

The concept of value is the core of any quantity model. This is what consumers and producers of quantities are (and have been) trying to convey. The task of a quantity model is to add just enough extra context to a value (by aggregation) that it is fully specified and can be fully understood. By value, we also include arrays of values.

However, there are use cases which involve the context without the value. We leave open for now the possibility of a quantity with an empty value.

## 4.3 Accuracy

We use the term accuracy to describe numbers which are needed to decide if the values in two compatible quantities are different. This includes errors (uncertainties), upper limits, and possibly pixelization size and instrumental resolution. There are many types of errors (ways to represent error): absolute error, relative error, systematic and random error, etc. Each type of error must have rules or operations that specify the correct way to use them in calculations.

## 4.4 Quality

Quality is similar in use to error or uncertainty, except that quality is generally not numeric. The general idea is that quality can be used to filter (include or exclude) quantities from further consideration, but it does not otherwise impact calculations.

*Note: One could convert quality flags to weights and thus use them within some types of calculations. However we do not model this here.*

In this document our accuracy and quality models are stubs, developed only enough to define their interface to Quantity. At present we assume that Quality is handled as a kind of Accuracy.

## 4.5 Array Axes

If a quantity has more than one value, access to it may be described by a simple list, or by an n-dimensional array index (imposing an array structure on the set of values). The quantity values thus become a function of n integers. Each of these n mathematical dimensions is called an array axis. (There may be more than one such organization of the pixels for a given quantity; e.g. both 1D, 2D and 3D arrangements of the values.) The word 'pixels' here is taken to include 'samples' and 'cells', there is no intent to imply a regular array in an underlying physical (detector).

## 4.6 Coordinates

A array axis (dimension of the array index) or (e.g. in the case of projected spherical coordinates) a set of array axes, can be associated with a phenomenon (and a Quantity), in the sense that the quantity values now become a function of n parameters (independent variables), each of which can take any of the values a Quantity can take. These parameters are called coordinates. For example, a quantity containing an array of flux values might be associated with a time coordinate if it is a time series, or a wavelength coordinate if it is a spectrum.

More than one such set of associations can be applied to a quantity. For example, a 3D cube might have RA, Dec, time coordinates and also focal plane position/exposure number coordinates.

The array axes themselves constitute a special case of such coordinates. The function that takes an array index and returns the coordinate value is a special case of a mapping (see below).

For some types of quantities - specifically those using string data types (e.g. galaxy classification) the coordinate is more like an enumerated type that lists all the allowed values. Proper names of astronomical objects are also values taken from a standard list (for example, M31 is a name from the Messier Catalog). The same sort of pattern applies to positions (X,Y is a value in the J2000 equatorial coodinate system).

## 4.7 Frames

Frames give the context of values, either the quantity values or the values of the phenomena tied to axes. Frames include information about units, coordinate zero points, and identification of coordinate systems (some of which are parameterized, e.g equatorial systems parameterized by equinox, epoch and reference system). The frame would include the phenomenon, but not the values.

When there is more than one axis and each has an independent frame, we have a compound frame made of combining the individual frames.

## 4.8 Coordinate Systems

We use the phrase 'coordinate system' to indicate metadata needed to describe the context of values which go beyond simple units. Familiar examples of coordinate systems are celestial coordinates, and velocity and spectral coordinate systems.

## 4.9 Units

A unit is a component of a frame that defines the measuring rod used for a particular value. Units are about size rather than zero point (there are a few problematic cases like 'degrees Celsius'); for instance in a celestial coordinate system, choice of units (degrees or arcseconds) is orthogonal to choice of origin (barycenter, Earth, etc) and choice of orientation (J2000 ICRS).

## 4.10 Transformation

A transformation maps values expressed in one frame to values expressed in another frame. We consider only one-one and many-one transformations; one-many transformations are outside the scope of the present discussion. We don't define an explicit Transformation class, its properties are including in Mapping (below).

## 4.11 Mapping

A mapping is either the pairing of a transformation and its inverse, or just the transformation if its inverse would be one-many. It represents the full information about the connection between two frames.

A special case of Mapping is an explicit set of values - an array of real numbers is a look-up mapping from the integers to the reals. So, we can unify the concepts of Mapping and Values, allowing us to use one whereever the other might appear as a 'rule to get values'. In the case of a Quantity for which alternate represenations are provided - say, an array of Wavelengths which we also want to be accessible as a set of Frequencies - our overall Quantity might consist of two CoreQuantities (see below), one of which, WavelengthCQ, is the wavelength Values together with the wavelength Frame information, while the second, FrequencyCQ, is the wavelength-to-frequency Mapping togther with the frequency Frame information. The unification of the Values array with the Mapping idea lets us treat WavelengthCQ and FrequencyCQ as the same kind of thing, without us caring which of them holds the raw numbers.

## 4.12 Description

A description could be a phrase or sentence that describes the meaning of the quantity to a human, or it could be a UCD that either a human user or software could use to *understand* the quantity.

## 4.13 Data Type

Data types are used to represent values and errors. All data types are considered classes; there will be no *primitive types* in the model. There is no universally accepted place to draw the line between *simple* data types and complicated data types, so at this point we propose several places a line might be drawn. In principle the Quantity model will support arbitrary objects as data types, but we may restrict initial implementations to use only simple types. The Quantity model supports arrays of values; the Quantity data type is then the data type of a single element in the array (a Quantity containing an array of 2 doubles has data type "double", not "array of 2 doubles"; but one could have a Quantity contaning an single object of data type '2-tuple of doubles' - the semantics are different even if the data values are the same, and the operation 'get first value' will return different things in the two cases.).

The minimum set of allowed data types includes logical/booleans, integers, real numbers, dates, and strings. These are all what could be called *scalars* in the sense that there is a single represented value which does not require further parsing or deconstruction.

The next data type that could be added to the list is the interval, which is comprised of two fully ordered values of the same type. Thus, one can have intervals of numbers, strings, and dates, but not intervals of boolean values since they are not fully ordered (false is neither less than nor greater than true). The interval type would be useful in tables or axes describing binned data with different size bins, such as a list of observing time intervals. The *complex* type from mathematics is of similar complexity. However, semantically it is a point in a particular two-dimensional space, so it introduces the concepts of geometry.

Like the interval, there are many fixed-size data types that could be used, staring wit most simple geometry objects. Geometric types like point, line (segment), circle, ellipse, and rectangle are fixed size and well understood types. However, if we allow for geometry in general, one would immediately like to add polygon as well, but polygon a variable sized collection of points. Since string is also variable sized, and this appears to be the only slippery part of polygon, it is probably acceptable; polygon is certainly a very useful data type for representing regions, just as point is very useful in describing positions such as (RA, Dec) pairs.

Given that strings and polygons have variable length, it is tempting to consider other variable-length data types, such as lists, for use as the value of a quantity. Since such constructs are used for aggregation with none or minimal additional semantic value, we propose to aggregate quantities rather than allow such data types to be used within a quantity. (Note: Strings and polygons have considerable additional meaning beyond being a collection.)

Where to draw the line? There appear to be three places to draw the line. Minimally, we only allow scalars (string, number, date, and boolean). Alternatively, we allow for scalars, intervals, and tuples (points), but not general geometric types. It appears clear that beyond geometry and arrays there lies a whole mess of possible types... just look at the STL or the Java Collections API for a hint of the size and complexity of that can of worms.

Our proposal is to adopt the intermediate case: for the time being Quantities can be arrays of scalars, intervals or tuples, but not of polgyons, circles, or other objects. However, software should be prepared to handle unknown data types gracefully.

## 4.14 Associated Metadata

In our Quantity model, we promote extensibility by allowing other Quantities to be referenced as 'associated metadata'. The semantics of this association are not defined by the Quantity model, but may be defined by higher level models. For example, a SkyCoordinateSystem model based on Quantity might specify that if an Equinox quantity is present as associated metadata, it would be interpreted as the equinox of the coordinate system.

## 5 BasicQuantity Model

Here we summarize the BasicQuantity model before plunging into the more complicated StandardQuantity model.

The BasicQuantity model associates metadata with a single scalar value. We present its interface below. It allows a single value to be grouped with a name, description, data type, unit, UCD and accuracies (errors etc.). It does not support associated metadata, alternate descriptions, or coordinate systems, handled by the more sophisticated StandardQuantity model described below. The BasicQuantity interface is a subset of the StandardQuantity interface, so that we may inherit StandardQuantity from BasicQuantity.

*There is some disagreement about whether BasicQuantity should in fact support errors and associated metadata. Other methods from CoreQuantity below could be added to BasicQuantity.*

In the full model described below, BasicQuantity implements Frame, and here we list the methods of Frame as well as those specific to BasicQuantity.

| <<interface>> org.ivoa::BasicQuantity |
|---|
| getDataType(): DataType |
| setDataType( d: DataType ): bool |
| getDescription(): string |
| setDescription(d: string): bool |
| getName(): string |
| setName(n: string): bool |
| getUCDString(): string |
| setUCDString( u: string): bool |
| getCoordinateSystem(): CoordinateSystem |
| setCoordinateSystem( c: CoordinateSystem ) |
| getUnit(): Unit |
| setUnit(u: Unit): bool |
| getValue(): Object |
| setValue(v: Object): bool |
| addAccuracy(a: Accuracy) : bool |
| removeAccuracy(a: Accuracy) : void |
| getAccuracyList(): List |

## 6 Proposed Interface Model

Here we briefly describe some general aspects of the model. In the following subsections we present the interfaces, then describe the interfaces, and finally present sample pseudocode to illustrate possible use of the interfaces.

The core idea of this model is that a Quantity object can be thought of as **a rule for getting values** together with **a frame (context) for those values**. The rule-for-values idea, allowing you to either list values explicitly or provide an algorithm to derive them, unifies the use cases of simple data values, theoretical model fit, and array axes (so that the axes of an array Quantity are themselves Quantity objects).

The requirement to support both alternate coordinate axis descriptions (say RA,Dec versus detector coords vs (l,b)) and alternate value frames (say flux expressed in observed, dereddened, and rest frame versions) is rather separate from the requirement to fully describe data and their frames. We therefore layer the model by introducing CoreQuantity, which provides a single description of either data or coordinate axes, and StandardQuantity, which provides the mechanism to describe alternative representations and to connect a description of coordinate axes with the description of the data. A CoreQuantity has a single phenomenon (UCD) while a StandardQuantity may be able to provide the data in terms of several related UCDs, and, if the data is an array, may have coordinates on the array axes each with their own UCDs.

A StandardQuantity consists of CoreQuantity objects describing single values or arrays of values, and optionally CoreQuantity objects describing the coordinate axes on those arrays. We distinguish these cases by referring to the CoreQuantity objects describing values as ValueQuantity, and to the CoreQuantity objects describing coordinate axes as CoordsQuantity. The concepts ValueQuantityand CoordsQuantityare not distinct data types (classes) - they are both examples of the CoreQuantity class with different usage contexts.

A CoreQuantity can contain either explicit data values, or a mapping which generates data values given input data. A CoreQuantity has a function "get the 1st, 2nd, 3rd... value", which has an obvious interpretation if the data values are explicit. If the CQ has a Mapping rather than explicit values, it also may have a Parent CoreQuantity representing the input values and frame; this allows mappings to be chained together.

As a use case, consider a 1-D spectrum which has three axis representations: the original detector channels, the calibrated wavelength, and the frequency. We can provide three CoreQuantities describing the axis: Channel, which has the trivial unit mapping and no Parent (the 2nd value of the Channel axis is 2, the channel number); Wavelength, with a polynomial mapping and Channel as the Parent; and Frequency, with a F(lambda)=(c/lambda) mapping and Wavelength as the Parent. To find the 2nd value of Frequency, we chain back through the Parents till we get to Channel, and then map forward to find that Frequency(2) is F(poly(2)), applying the wavelength and frequency mappings to obtain the frequency value for channel 2. (One could actually do without the semi-trivial Channel quantity, but it can be useful to have it explicitly present. In practice, the mapping from wavelength to frequency is so basic that it will probably be handled automatically by software handling photon spectral coordinates rather than having to be explicit in the Quantity, but the example illustrates the general idea.)

The use case of an RA, Dec image always drives complexity in astronomical data models. Because the RA, Dec coordinate systems are coupled to the X, Y axis pair and not separately to the X and Y axes, we are forced to models which can represent this non-separability. Most (but not all) other

2-and-higher-dimensional use cases involve separable axes. In this model one method handling this is to make the CoordsQuantityof such an image be a single Quantity with vector (RA,Dec) data type, rather than two Quantities (one for RA and one for Dec). Similarly, a position-velocity data cube could have two axis quantities, one for position (with the 2-tuple double data type) and one for velocity (assumed in this case to be separable.). However, it is not forbidden to simply model the object using separate quantities for RA and Dec - it just becomes harder to do the bookkeeping on their common coordinate system.

The requirement to have a simple BasicQuantity interface for simple use drives an inheritance relationship between Frame (whose methods are needed for BasicQuantity), BasicQuantity, and CoreQuantity.

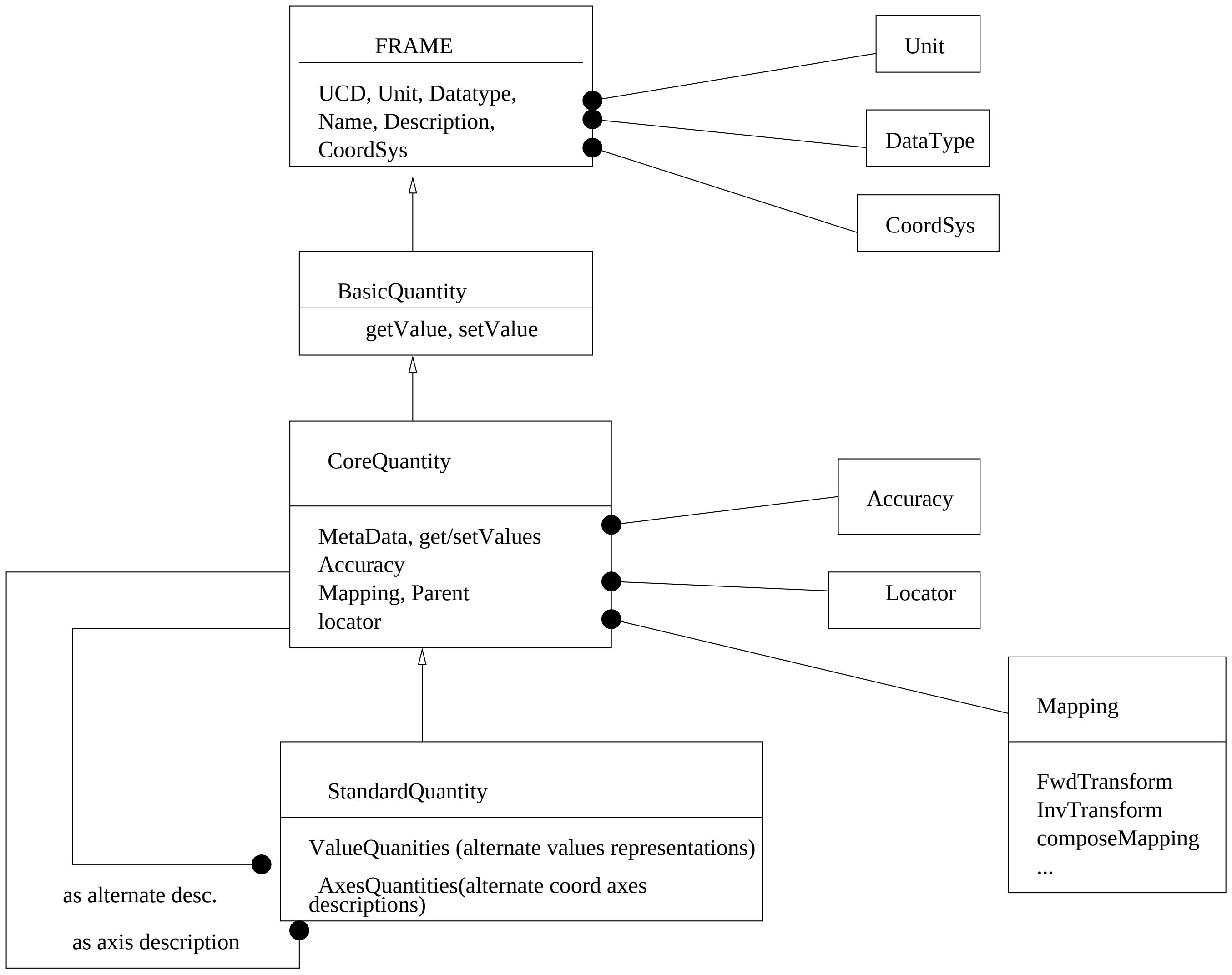

Figure 1: Relation between Quantity interfaces (which might be an inheritance diagram for Quantity classes). On the right are shown subsidiary interfaces which are incompletely worked out. The StandardQuantity extends CoreQuantity, and also aggregates CoreQuantities as both alternate value descriptions and as coordinate axis descriptions.

## 6.1 Interfaces

| <<interface>> org.ivoa::CoreQuantity |
|---|
| *extends BasicQuantity, and:* |
| addMetaData(m: CoreQuantity): bool |
| removeMetaData( m: CoreQuantity): bool |
| getMetaDataList(): List |
| getValue(locator: Locator): Object |
| setValue(locator: Locator, value: Object): bool |
| getValues(r: Region): List |
| setValues(r: Region, l: List ): bool |
| getNumberOfValues(): long |
| setNumberOfValues( n: long ): bool |
| getMapping(): Mapping |
| setMapping(m: Mapping, parent: CoreQuantity): bool |
| getParent(): CoreQuantity |
| CoreQuantity( list: List(CoreQ)): CoreQuantity |
| getMembers(): List |
| createLocator(): Locator |

| <<interface>> org.ivoa::StandardQuantity |
|---|
| *extends CoreQuantity plus:* |
| addCoordsQuantity( a: CoreQuantity ): bool |
| removeCoordsQuantity( a: CoreQuantity ): bool |
| getCoordsQuantityList(): List |
| addValueQuantity( q: CoreQuantity ) |
| removeValueQuantity( q: CoreQuantity ) |
| getValueQuantityList(): List ) |
| setCurrentCoordsQuantity( a: CoreQuantity ): bool |
| setCurrentValueQuantity( q: CoreQuantity ) |

| <<interface>> org.ivoa::BasicQuantity |
|---|
| *extends Frame, and:* |
| getValue(): Object |
| setValue(v: Object): bool |
| addAccuracy(a: Accuracy) : bool |
| removeAccuracy(a: Accuracy) : void |
| getAccuracyList(): List |

<<interface>>
org.ivoa::Unit

get(): string
set( u: string ): bool

<<interface>>
org.ivoa::DataType

get(): string
set( u: string ): bool

<<interface>>
org.ivoa::Region

*TBD*

<<interface>>
org.ivoa::Locator

setArrayIndices( p: List ): bool
setCoords( p: List ): bool
offset( o: long ): bool
next(): bool
prev(): bool
getParent(): CoreQuantity
forward(n: long): bool
back(n: long): bool
reset(): void
all(): Region
getDimensions(): long List
setDimensions( l: long List ): bool
setAxisIndices( l: List )
getAxisIndices( ): List
setRegion( r: Region ): bool

<<interface>>
org.ivoa::Accuracy

(TBD)

<<interface>>
org.ivoa::Frame

setCoordinateSystem( c: CoordinateSystem): bool
getCoordinateSystem(): CoordinateSystem
getUCDString(): string
setUCDString( u: string): bool
getUnit(): Unit
setUnit(u: Unit): bool
getDataType(): DataType
setDataType( d: DataType ): bool
getDescription(): string
setDescription(d: string): bool
getName(): string
setName(n: string): bool
Frame( framelist: List ): Frame
getFrameMembers(): List

<<interface>>
org.ivoa::Mapping

fwdTransform( in: List ): List
invTransform( in: List ): List
getMappingType(): MappingType
getMappingParams(): List
composeMapping( m: Mapping ): Mapping
decomposeMapping(): List
Mapping( maplist: List ): Mapping
getMappingMembers(): List

<<interface>>
org.ivoa::CoordinateSystem

(TBD)

## 6.2 Interface descriptions

### 6.2.1 Frame

The Frame carries the basic contextual information: name, description, units, UCD, description, coordinate system.

The Frame( framelist: List ) constructor creates a compound frame from individual frames. The getFrameMembers() returns the individual frames.

### 6.2.2 Mapping

The Mapping transforms values. It is described in more detail in the Mapping Data Model document.

Similar to the Frame methods, the Mapping( maplist ) constructor and the getMappingMembers() routine handle compound maps. A compound map is just a way of grouping maps into an object of higher dimensionality: the (exterior) product of a spatial axes map and a velocity axis map, for example. It should not be confused with composing maps in series, which is handled by the composeMapping and decomposeMapping routines.

### 6.2.3 CoreQuantity

CoreQuantity is a Frame which can return a finite set of values and accuracies, and may have other generic associated metadata.

The **getValue** methods return a single object whose data type is that of the CoreQuantity. The various polymorphic versions let you ask for the nth value, the value at array index coords (i,j..) or if coordinate axes are defined, the value at particular coordinates. This may imply an interpolation if the coordinates are not on an integer array index.

The **add/get/removeMetaData** methods allow us to attach other CoreQuantities (or BasicQuantities) as associated metadata.

The **getMapping** method returns the Mapping (see below) associated with the rule to get values. The **getParent** method returns a Quantity which provides the input frame for the Mapping.

The **Accuracy** methods associate a list of Accuracies of different kinds with the quantity. We haven't worked out the details yet, but the idea is that you might attach different kinds of accuracy (quality, uncertainties, array index cell sizes) to the data, and you get the list of them all and look for the kind of accuracy you want.

If the CoreQuantity was serialized as a set of explicit values, the resulting Mapping is a special type we will call LookupMapping, and the getParent method returns null.

### 6.2.4 StandardQuantity

The **ValueQuantity** methods manage alternate representations of the values. The **CoordsQuantity** methods manage alternate representations of the coordinate axes, if any.

### 6.2.5 Locator

The Locator object is provided to give a way of specifying to the Core-Quantity's GetValue method which data element, elements, or interpolation between elements, is wanted. The locator's array_index and coords methods return pixel (array index) locations in the object, possibly not at integer array index locations. We also provide iterator methods, although it's not clear if these are needed within the model.

## 6.3 Examples

As a moderately difficult use case, suppose we consider a position-velocity (RA, Dec, V) data cube of size 512 x 512 x 30, and we wish to evaluate the following cases: (1) Value of array index 128,20,3; (2) Interpolated value at coordinates (121.32 deg,-22.12 deg,120 km/s).

The psuedocode below shows how to do this; please note we have avoided defining convenience functions in our interface, and these operations would be much simpler in a production interface.

```
/* First we create an Axes collection describing the axes
of the array in RA, Dec, velocity space. */

/* The initial step is to define the frames */

Pos = new CoreQuantity();
Pos.setCoordinateSystem( "J2000/FK5" );
Pos.setUCDString( "POS_EQ" );
Pos.setUnit( "deg" );


/* Now define the spatial mapping. We haven't elaborated the specific
   subclasses of Mapping interface yet so take these as illustrative */

Mapping M1 =  new SphericalTanProj2( 121.30, 21.02, ... ), NULL );
Pos.setMapping( M1 );
Pos.getLocator().setDimensions( { 512, 512 } );
/* Pos has datatype "vector of 2 doubles" */
Pos.setDataType( VO_Vector( VO_DOUBLE, 2 ));


Vel = new CoreQuantity();
Vel.setCoordinateSystem( "Heliocentric" );
Vel.setUCDString( "RAD_VELOCITY" );
Vel.setUnit( "km s^-1" );
/*  For the velocity axis, suppose we have a set of fixed values for each channel
    rather than a continuous mapping:
 */

Vel.setNoValues( 30 );
Vel.setValues( List { 10.0, 20.0, 25.0, 28.0, 32.0 ... } );
/* automatically implies Vel.setMapping( "Explicit" ) */
Vel.setDataType( VO_DOUBLE );

/* Now collect these axes into an Axes collection */

CoreQuantity AxisCollection1 = new CoreQuantity( { Pos, Vel } );


/* Next make the value (flux) quantity: */
```

```
CoreQuantity Flux = new CoreQuantity( );
Flux.setCoordinateSystem( "Baars_et_al_1977_scale" );
Flux.setUnit( Unit("Jy Hz"));
Flux.setUCDString( "phot.flux" );
Flux.setDataType( VO_DOUBLE );

/* Set the values for the array using some method or other */
Flux.setValues( List readFITS(foo));

/* Now make the overall quantity */

StandardQuantity Q = new StandardQuantity();
Q.addValueQuantity( Flux );
Q.addCoordsQuantity( AxisCollection1 );

/* Just to emphasize there can be more than one AxisCollection,
  let's do a second one: */
AxisCollection2 = CoreQuantity( { Galactic, Wavelength } );
Q.addCoordsQuantity( AxisCollection2 );



...
/* Note that A is the base Axes, so L knows to expect 3 values */
L = Q.getLocator();
double V1 = Q.getValue( L.setArrayIndices( 128, 20, 3 ));
double V2 = Q.getValue( L.setCoords( 121.32, -22.12, 120.0 ));
```

Recall that we can define alternate CoreQs which are transformations of other CoreQs, allowing chained mappings. Hence:

```
Frame frame1 =new Frame( ObservedFrame, UCD_PHOT_FLUX, "Jy Hz");
Frame frame2 =new Frame( RestFrame, UCD_PHOT_LUM, "erg s^-1" );
Mapping map1 =new Mapping("Dereddening");
Mapping map2 =new Mapping("RestFrame");
CoreQuantity DeredFlux = new CoreQuantity( frame1 );
DeredFlux.SetMapping( map1, Flux );
CoreQuantity RestLum = new CoreQuantity( frame2 );
RestLum.SetMapping( map2, DeredFlux );
Q.addValueQuantity( DeredFlux );
Q.addValueQuantity( RestLum );
```

How might the first part of this look in a production interface with convenience functions? Perhaps a little more compact, for example:

```
Mapping M1 =  new SphericalTanProj2( 121.30, 21.02, ... ), NULL );
Frame  F1 = new SkyFrame( "J2000/FK5", POS_EQ, "deg" );
CoreQuantity Pos = new CoreQuantity( F1, M1, { 512, 512 }, PositionType );
CoreQuantity Vel = new CoreQuantity( F2, NULL, {30}, VO_DOUBLE );
Vel.setFrame() = new VelFrame( "Heliocentric", RAD_VELOCITY, "km s^-1" );
Vel.setValues( List { 10.0, 20.0, 25.0, 28.0, 32.0 ... } );
CoreQuantity AxisCollection1 = new CoreQuantity( { Pos, Vel } );
```

```
Frame Flux = new PhotFrame("Baars_et_al_1977_scale", phot.flux, "Jy Hz" );
StandardQuantity Q = new StandardQuantity( Flux, VO_DOUBLE, AxisCollection1 );
Q.setValues( List readFITS(foo));
```

# 7 XML Serialization

Since an interface does not specify the storage format of the data content, we will specify rules for how one serializes an instance (object) that implements an interface. As an initial cut without such rules, the interfaces described above are used by hand to derive an XML Schema that allows for the serialization of instances (data) and its storage or transamission in a fashion that is independent of the class library (implementation) used by any application.

The proposed XML serialization of the model may be summarized as follows:

## 7.1 Basic Quantity

1. The BasicQuantity interface is serialized with a `<basicQuantity>` tag.

2. The `<basicQuantity>` tag has attributes of `name`, and `description` and `ucd`. Each of these attributes may instead be represented as an element tag.

3. The value of BasicQuantity is held within a `<value>` tag.

4. Within each BasicQuantity, the data type is specified by a different tag for each supported data type: `<integer>`,`<string>`,`<float>`. The tags has an attribute of `width` (for suggested display field width). The float tag has the additional attribute `precision` for number of decimal places). The integer tag has the additional attributes `signed=yes|no` and `type=octal|decimal|hexadecimal` (the defaults are signed=yes and type=decimal). These types do not specify a binary representation. We expect to later define derived types which give a recommended binary representation compatible with the FITS data types.

5. Within each BasicQuantity or Frame, the Name, Description, UCD, Unit and CoordSys values may appear as `<name>`, `<description>`, `<ucd>`, `<units>` and `<coordSystem>` tags. The special tag `<unitless/>` is equivalent to `<units></units>`. The name, description and ucd tags may appear as attributes of the basicQuantity tag instead.

6. An `<accuracy>` tag is provided to encapsule accuracy information. Detailed modelling of accuracy has been deferred.

7. We provide a special serialization `<trivialQuantity>` to cover the special case of a BasicQuantity of string data type, which has no units, etc.

8. There is also a `<frame>` tag to serialize a Frame, which is essentially a quantity without values.

9. We provide a special serialization `<refQuantity>` which acts as a pointer to quantities elsewhere in the same document. This is useful with the `<metaData>` tag (see below).

10. The order of tags within a BasicQuantity is: name, description, ucd, coordSystem, units, value

## 7.2 CoreQuantity

1. The CoreQuantity interface is serialized with a `<coreQuantity>` tag with attributes of `name` and `description`, `ucd`, and also `size`, the latter being integer-valued and corresponding to the getNumberofValues() method.

2. The element tags defined for use within BasicQuantity also apply to CoreQuantity.

3. A special data type `<vector>` allows the construction of vectors of the basic types, with the elements described by extra Frames within the `<vector>` tag. This corresponds to the getFrameMembers() method.

4. A `<metaData>` tag encloses Quantity tags describing associated metadata, corresponding to the getMetaDataList() method.

5. For an algorithmic mapping to get values, we use a `<mapping>` tag. Within this tag, the specific mapping has tags to define its parameters; these will be discussed in a separate document on mapping serialization.

6. For the case where the mapping from sample number to value is given by explicit values, a `<values>` tag is used instead of `<mapping>`.

7. For cases where there is more than one value, the individual values within `<values>` may be enclosed within `<value>...</value>` tags. Alternatively the values may simply be space-separated, except in the case of strings which might contain spaces. Examples:

```
<values>2 -30 -85 74 16 57 65 -3 87 81</values>
<values>String1 String2 </values>
<values><value>First string</value><value>Second string</value></values>
```

8. In the case of only one value, the `<values>` and `<value>` tags are equivalent.

9. For the special case where the data type is an arbitrary object, we do not provide a specific data type `<object>` tag. Instead, we replace the `<values>` tag with a `<members>` tag, which should be read as '`<values>` and by the way the datatype is `<object>`'.

10. The order of tags within a CoreQuantity is: metadata, name, description, ucd, coordSystem, units, (data type) or vector or members, values or mapping, accuracy.

## 7.3 StandardQuantity

1. The StandardQuantity interface is serialized with a `<standardQuantity>` tag with attributes `name`, `description` and `size`.

2. The StandardQuantity uses `<axesList>` and `<altValues>` to delimit the alternate axes and value representations of CoordsQuantityList and ValueQuantityList respectively; i.e. the method getCoordsQuantityList() maps to `<axesList>` and getValueQuantityList() maps to `<altValues>`.

3. Within an `<axesList>`, each single representation of the axes is given within an `<axes>` tag. Within that tag, a sequence of `<coreQuantity>` tags describe the axes. The `<axes>` tag corresponds to the addCoordsQuantity() method, and represents a compound coreQuantity (a coreQuantity with 'members', see below) aggregating the individual axes.

4. If there is only one representation of the axes, the `<axesList>` tag may be omitted and the single `<axes>` tag appears directly within `<standardQuantity>`.

5. The `<altValues>` tag, contains a sequence of `<coreQuantity>` tags describing the different representations of the StandardQuantity values.

6. If there is only one representation of the values, the `<altValues>` tag is omitted.

7. A StandardQuantity contains a default value (or mapping) representation with the properties of CoreQuantity. This default does not appear within the `<altValues>` tag - instead its elements (ucd, values, mapping, metadata etc) appear at the StandardQuantity level with no `<coreQuantity>` tag - making use of the idea that a StandardQuantity "is a" CoreQuantity. These elements apply only to the default ValueQuantity and not to the alternate representations, which have their own elements.

8. For multidimensional arrays, the dimensions of the axes are inferred from the `size` attribute of the individual axis core quantities. The ordering convention is consistent with FITS so that

```
<quantity>
    <axes>
      <coreQuantity name="x" size="3"/>
      <coreQuantity name="y" size="2"/>
    </axes>
```

```
          <values> A B C D E F</values>
     </quantity>
```

implies a 3 x 2 array ordered so that

```
v(x=1,y=1) = A
v(x=2,y=1) = B
v(x=3,y=1) = C
v(x=1,y=2) = D
v(x=2,y=2) = E
v(x=3,y=2) = F
```

9. The order of tags within a StandardQuantity is: metadata, name, description, ucd, coordSystem, units, axesList or axes, (data type, including vector or members, values or mapping, altValues, accuracy.

## 7.4 Notes on XML aspects of the serialization

There are 3 types of quantity, "basic", "core" and "standard", and in that order are extensions of the prior type node structure. Extension is via accretion of XML components (whether child nodes or attributes).

Thus, as components are optional/have sensible defaults, you should be able to "upcast" a more primitive type of quantity into a higher one. For example, the serialization of a coreQuantity describes both a coreQuantity, and a standard quantity (with some implicit defaults).

Conversely, IF a "advanced" level quantity is fairly simple (such as a core quantity that held only one scalar number) then its representation in XML will look exactly like its parent realization, the basic quantity.

XML serializations of quantities may have the following valid XML node names (excepting extensions the user may make to the schema) with the corresponding XML schema types:

| XML Node Name | XML Schema Complex Types allowed |
|---|---|
| basicQuantity | basicQuantityType |
| coreQuantity | coreQuantityType |
| stdQuantity | stdQuantityType |
| quantity | stdQuantityType |

A user should be able to create XML schemata from these types. The new nodes may have limits placed on the quantities such that units, ucd, etc are limited to a preferred settings. For example, the "velocity" element constucted from a "stdQuantityType" may be limited to having units of type "`cm/sec^-1`" and a UCD for "velocity". The new schema would give this "extended" quantity the node name "velocity" in order to identify it in the XML document. In all other aspects, it behaves as an unrestricted quantity. Examples on how to do this in XML schemata will be given in another document.

Within a type of quantity, there are a number of simplifying defaults which, for the largest fraction of data, will serve to compress the XML which need be passed; some possible simplifications are noted below. Thus, when some attributes are not specified, they are assumed to fall to some default. For example, if dataType isn't specified, it is assumed to be "a scalar string"; if "units" are not specified, they are assumed to be "unitless". To show a *possible* choice, he following examples *could* be considered equivalent:

```
<coreQuantity name="example">
  <unitless/>
  <string>
  <values>My data</values>
</coreQuantity>
```

and

```
<coreQuantity name="example">
   <values>My data</values>
</coreQuantity>
```

and

```
<coreQuantity name="example">
   <value>My data</value>
</coreQuantity>
```

## 7.5 Strings and string arrays in XML

Note that in XML, white space in normal string values (called PCDATA in XML) is not preserved. If number of spaces or presence of leading and trailing spaces is important, we would need to do

```
<coreQuantity name="example">
     <values><![CDATA[ My   data  ]]></values>
</coreQuantity>
```

If we want an array of strings, we would do

```
<coreQuantity name="example" size=2>
        <values>
          <value><![CDATA[ My   data  ]]></value>
          <value><![CDATA[ Your   data  ]]></value>
        </values>
</coreQuantity>
```

Note that white space **is** preserved in attribute values. Hence, in

```
<basicQuantity name="Silly   Name ">Fred</basicQuantity>
```

the white space in the name is preserved, while in

```
<basicQuantity>
 <name>Silly    Name </name>
 <value>Fred</value>
</basicQuantity>
```

it is not.

## 7.6 XML instance examples

1. Single Scalar Value.

   Here we present a single string valued quantity:

   ```
   <quantity name="Observer" description="Obs. name">Johannes H. Kepler</quantity>
   ```

   An equivalent and more verbose version:

   ```
           <!-- a completely explicit specification (stdQuantity) -->
           <quantity size="1">
                   <name>Observer</name>
                   <desc>name of Observer</desc>
                   <ucd/>
                   <coordSystem/>
                   <axesList/>
                   <unitless/>
                   <string length="14"/>
                   <values>Johannes H. Kepler</values>
                   <altValues/>
           </quantity>
   ```

   A full XML document might have

   ```
   <?xml version="1.0"?>
   <!-- test one : basic quantity with ucd, dataType and value -->
   <ivoa:basicQuantity
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:ivoa="http://ivoa.org"
         xsi:schemaLocation="http://ivoa.org BasicQuantity.xsd"
         name="Observer"
         description="name of Observer"
   >
      <ivoa:ucd>human.observer</ivoa:ucd>
      <ivoa:string width="14"/>
      <ivoa:value>Johannes H. Kepler</ivoa:value>
   </ivoa:basicQuantity>
   ```

2. Here is a serialization of a Quantity containing three photometry points, with an enumerated wavelength axis and several alternate represenations of the values. Note that this illustration of the use of Quantity isn't a recommended way to represent this kind of data; the Observation model will provide a full spec for that.

   ```
   <?xml version="1.0"?>
   <standardQuantity
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   ```

```
        xmlns="http://ivoa.org"
        xsi:schemaLocation="http://ivoa.org StandardQuantity.xsd"
        name="Photometry"      size="3"
  >
     <metaData>
          <trivialQuantity name="Observer">Alex Szalay</trivialQuantity>
          <basicQuantity name="Exposure"><units>s</units><float/>
                         <value>1480.2</value></basicQuantity>
          <coreQuantity name="ChipIDs" size="3"><integer/>
                         <values>2 7 9</values></coreQuantity>
     </metaData>
     <ucd>phot.opt</ucd>
     <axes>
        <coreQuantity name="wavelength" size="3"><float/>
           <values>4400.0 5500.0 9700.0</values><units>Angstrom</units>
        </coreQuantity>
     </axes>
     <units>mJy</units>
     <float/>
     <values>4.3 8.2 1.3</values>
     <altValues>
        <coreQuantity name="Magnitudes" size="3">
         <ucd>phot.mag</ucd>
         <values>12.1 14.3 9.8</values>
         </coreQuantity>
        <coreQuantity name="F-lambda"  size="3">
          <ucd>phot.flambda</ucd>
          <units>erg cm^-2 s^-1 Angstrom^-1</units>
          <values>1.31E-12 1.28E-13 1.88E-13</values>
        </coreQuantity>
     </altValues>
  </standardQuantity>
```

3. Here is an example of a StandardQuantity with an array and coordinate mappings on the axes.

   The `<axesList>` tag encloses two `<axes>` tags, each with its own representation of the axes. The first `<axes>` tag has 'x,y,time' axes; the second has 'pos,utctime' axes. The 'pos' axis has vector data type, and the `<vector>` tag is used to specify the two individual frames `RA,Dec` making up the vector. The vector has a single mapping taking (x,y) to (RA,Dec); since the mapping is intrinsically two-dimensional we attach it to the vector concept 'pos' rather than the separate RA and Dec concepts. After the `<axesList>` tag the values of the actual image are given in `<values>`. In actual use we will probably replace this with a reference to part of a FITS file; the syntax for this has not yet been discussed.

```
 <quantity name="Multiple time exposure image" size="18">
```

```
<ucd>phot.flux</ucd>
<units>erg cm^-2 sec^-1</units>
<axesList>
   <axes>
      <quantity name="x" size="1024">
         <integer width="2"/>
         <mapping><unit-mapping/></mapping>
      </quantity>
      </quantity name="y" size="1024">
         <integer width="2"/>
         <mapping><unit-mapping/></mapping>
      </quantity>
      <quantity name="time" size="3">
        <coordSystem>
          <timescale>TDB</timescale>
          <timezerojd>2454123.1</timezerojd>
         </coordSystem>
         <units>s</units>
         <mapping>
           <linear-map ref=1.0 value=1483212.3 step=600.0/>
         </mapping>
      </quantity>
    </axes>
    <axes>
      <quantity name="pos">
        <ucd>POS.EQ</ucd>
          <coordSystem>J2000/ICRS</coordSystem>
          <units>deg</units>
          <vector>
              <frame name="RA" size=1024/>
              <frame name="Dec" size=1024/>
          </vector>
          <mapping>
            <wcsmap type="TAN">
              <refvals>131.2181 -31.1284</refvals>
              <refpos>512.1 512.1</refpos>
              <scales>-0.0016 0.0016</scales>
              <rotation>48.3121</rotation>
            </wcsmap>
          </mapping>
        </quantity>
        <quantity name="utctime" size="3">
           <coordSystem>UTC</coordSystem>
           <units>d</units>
           <mapping>
              <polynomial nparams="3">
                <param>131281.4</param>
```

```
                    <param>-.00013</param>
                    <param>4.823</param>
                  </polynomial>
                </mapping>
            </quantity>
         </axes>
      </axesList>
      <float width="5" precision="2"/>
      <values>10.12 12.34 20.34 13.87 24.76  5.67 6.80 .7 12.8
                               ......
                0.12 12.34 20.34 13.87 24.76  5.67 6.80 .7 12.8
      </values>
   </quantity>
```

## 7.7 Advanced XML instance examples: Quantities with members

Here we illustrate a possible use of Quantity to describe a more complicated objects.

1. A table represented as a Quantity. Here we use the idea that the data type of Quantity can be an arbitrary object. The `<members>` tag allows us to aggregate simple quantities to make such an object. Access by row is indicated by having the parent quantity have a common row axis specified, and all of the member quantities refer to this axis (this is done explicitly here but it should probably be OK to omit this in the serialization). ıThis illustrates the possible uses of Quantity; we are not proposing here to replace VOTABLE as the standard way to represent tables.

   The data are:

| | | | | | |
|---|---|---|---|---|---|
| 1 | Berkeley58 | 41 | 7 | 2 | Florov,B.H.,Izv. |
| 2 | IC1805 | 354 | 135 | -30 | Vasilevskis.S.et al., |
| 3 | IC4665 | 275 | 87 | -85 | Sanders,W.L., |
| 4 | IC4756 | 464 | 166 | 74 | Herzog,A.D.et al., |
| 5 | NGC129(A) | 70 | 18 | 16 | Lenham,A.P., |
| 6 | NGC1664 | 222 | 135 | 57 | Kerridge,S.J.et al., |
| 7 | NGC1817 | 752 | 265 | 65 | Tian K.P.et al. |
| 8 | NGC188 | 228 | 136 | -3 | Upgren,A.R., |
| 9 | NGC1912 | 998 | 172 | 87 | Mills,G.A. |
| 10 | NGC2099(A) | 243 | 216 | 81 | Jefferys,W.H., |

   taken from Catalog 1215, Zhao & Tian, ”Tables of membership for 43 open clusters (1994 Version) (1995)”. Here is the StandardQuantity serialization using `<members>`.

```
   <quantity name="Catalog 1215"
    description="Tables of membership for 43 open clusters (1994 Version)"
```

```
          size="60">
          <axes>
               <!-- declaring one or more axes with members means that that axis
                    is held in common between all the members, and can be used to
                    access the quantity. In this case, "rows" describes the common
                    row-axis exsists between the all the columns
                 -->
            <quantity qid="rows" name="rows" description="rows in document" size="10">
            <!-- linear algorithm for generation of row numbers -->
            <mapping><polynomial><param>0</param><param>1</param></polynomial></mapping>
            </quantity>
          </axes>
          <members>
       <!-- these are the columns of data.. all have unifiying reference to rows axis of p
           <quantity name="Seq" description="[1/43]?+ Order number" size="10">
              <axes><refQuantity qidRef="rows"/></axes>
              <integer type="decimal" width="3" noDataValue="-99"/>
                   <values>1 2 3 4 5 6 7 8 9 10</values>
           </quantity>
                <quantity name="Cluster" description="Name of open cluster" size="10">
                   <axes><refQuantity qidRef="rows"/></axes>
                   <string width="13" noDataValue="             "/>
                   <values>Berkeley58 IC1805 IC4665 NGC129(A) NGC1664
                           NGC1817 NGC188 NGC1912 NGC2099(A)</values>
                </quantity>
                <quantity name="Nstars"
                   description="? Total number of stars (1st line)" size="10">
                   <axes><refQuantity qidRef="rows"/></axes>
                   <integer type="decimal" width="5" noDataValue="-9999"/>
                   <values>41 354 275 464 70 222 752 228 998 243</values>
                </quantity>
                <quantity name="Nmembers" description="Number of cluster member (1)" size=
                   <axes><refQuantity qidRef="rows"/></axes>
                   <integer type="decimal" width="5" noDataValue="-9999"/>
                   <values>7 135 87 166 18 135 265 136 172 216</values>
                </quantity>
                <quantity name="PA"
                    description="[-90/90]? Major axis rotation angle" size="10">
                   <axes><refQuantity qidRef="rows"/></axes>
                   <units>deg</units>
                   <integer type="decimal" width="4" noDataValue="-999"/>
                   <values>2 -30 -85 74 16 57 65 -3 87 81</values>
                </quantity>
                <quantity name="Ref" description="Reference" size="10">
                   <axes><refQuantity qidRef="rows"/></axes>
                   <string width="26" noDataValue="                          "/>
                    <values>
                    <value>Florov,B.H.,Izv.</value>
                    <value>Vasilevskis.S.et al.,</value>
                    <value>Sanders,W.L.,</value>
                    <value>Herzog,A.D.et al.,</value>
                    <value>Lenham,A.P.,</value>
                    <value>Kerridge,S.J.et al.,</value>
                    <value>Tian K.P.et al., </value>
                    <value>Upgren,A.R.,</value>
                    <value>Mills,G.A.,  </value>
                    <value>Jefferys,W.H.,</value>
                </values>
            </quantity>
        </members>
    </quantity>
```

# 8 Appendix: related models

## 8.1 Data Types and Representations

Implementations should assume that integers may be signed 4-byte or 8-byte (i.e. int and long in Java and C#). Real numbers may be 4-byte or 8-byte IEEE floating point values, including positive and negative infinity and NaN values. By *date* we mean date and time with precision of at least one second.

A related issue is that of data representation. In astronomy two particularly delicate cases are those of time and celestial position, which frequently have string representations as, e.g. ISO dates and sexagesimal positions. The case of celestial position may be handled as a serialization issue, since there is a one to one mapping between such positions and a numerical representation as 2-tuples of decimal degrees. However, UTC dates are not so easy: the UTC timescale is a series of labelled instants and does not map to a continuous numerical value, because of the presence of leap seconds. (Although UTC times are frequently expressed as JD dates, for the highest precision work it is an error to subtract two UTC JD dates and use the result as the intervening time). You could convert to a continuous timescale such as TT, but that might be regarded as a change of science content. This example suggests the need to support special types, although more work is needed to see if we need to do this at the Quantity level or if it can be handled as a string as far as Quantity is concerned and the specialness handled at a higher level which cares more about the astronomical concepts.

## 8.2 Errors and Quality

Not all astronomical quantities have errors. Fundamental constants, definitions, integer-valued instrumental settings or data processing choices, and many classifications, names and other string-valued quantities do not have errors. Further, for many datasets that should have errors, the errors are unknown or poorly characterized. Nevertheless, the handling of errors is critical to much astronomical analysis and the VO must eventually support it. The most basic data manipulation operations will therefore include tests for the presence of errors and so it makes sense for Quantity to handle this.

The VO Error Model will be elaborated separately. At a minimum it will handle statistical fractional relative errors and systematic zero-point absolute errors. Errors can be on a per-sample basis or constant for the whole set of data samples. Some applications may require the calculation of errors on functions of a subset of data samples, and such calculations may not be a simple function on the per-sample errors of each sample. Conversely, the per-sample error may be originally computed based on a model involving surrounding data values (consider the nonlinearities involved in the grain density on photographic plates). Upper (non-detection) and lower (saturation) limits as well as dead-time errors also need to be handled.

Closely related to errors are quality masks. Quality masks are frequently used in astronomy to indicate the need for caution in using a data value. Conceptually they are a set of flags, each taking a finite set of values (not necessarily just true/false) and each indicating a different possible issue with the data. (Sometimes these are combined into a single integer interpreted as a bit-mask, but we will not insist on the implementation here.) The presence of quality flags does not necessarily indicate the need to throw out the data; it usually depends on the science question being asked. Often but not always, a quality flag represents an error value whose magnitude is unknown - examples such as 'uncertain astrometry', 'line flux poorly measured', and so on.

Some possible interfaces to Accuracy (or some Accuracy subclasses):

| <<interface>><br>org.ivoa::Accuracy |
|---|
| getError(): Object<br>getUpperError(): Object<br>getLowerError(): Object<br>getProbError( level: SigLevel ): Object<br>getAccuracyType(): AccuracyType<br>getLimit( bool upper_or_lower ): Object<br>isUpperLimit(): bool<br>isLowerLimit(): bool<br>getQuality(): ? |