1 Implementation: Coverage Core

This implementation work unit concerns most of Coverage Core. The classes which are defined in this implementation effort are the basis upon which special cases in other packages are built. These classes are:

- Coverage
- ContinuousCoverage
- ValueObject
- DomainObject
- AttributeValues
- GeometryValuePair

Some classes have been omitted from coverage core to be considered under separate cover. These are various subclasses of GeometryValuePair which are for use with specializations of DiscreteCoverage. They have been omitted because they are specified incorrectly and need to be fixed before they are implemented.

1.1 GeoAPI Interface Diagram

The interfaces shown in Figure 1 must be implemented by the coverage core work unit.

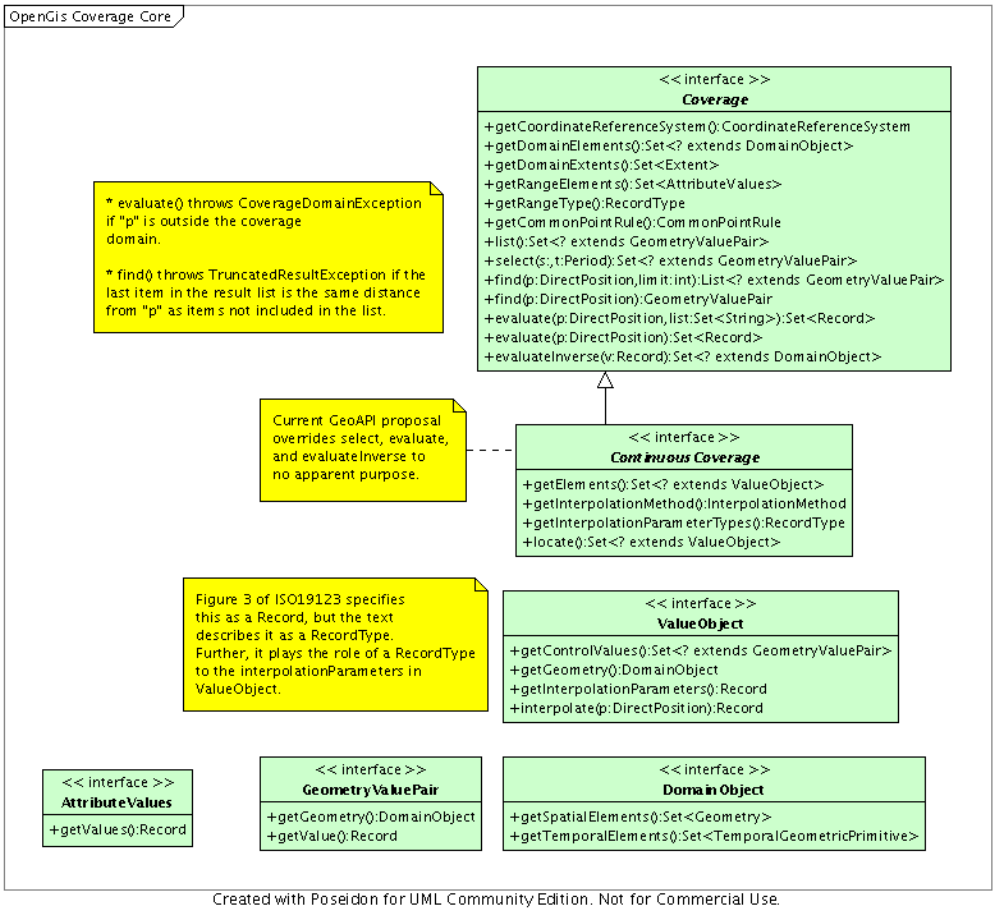

Figure 1: GeoAPI Interfaces for Coverage Core

This work unit includes a number of items (exceptions and code lists) which are supplied with GeoAPI and need

not be implemented in GeoTools. These items are shown in Figure 2.

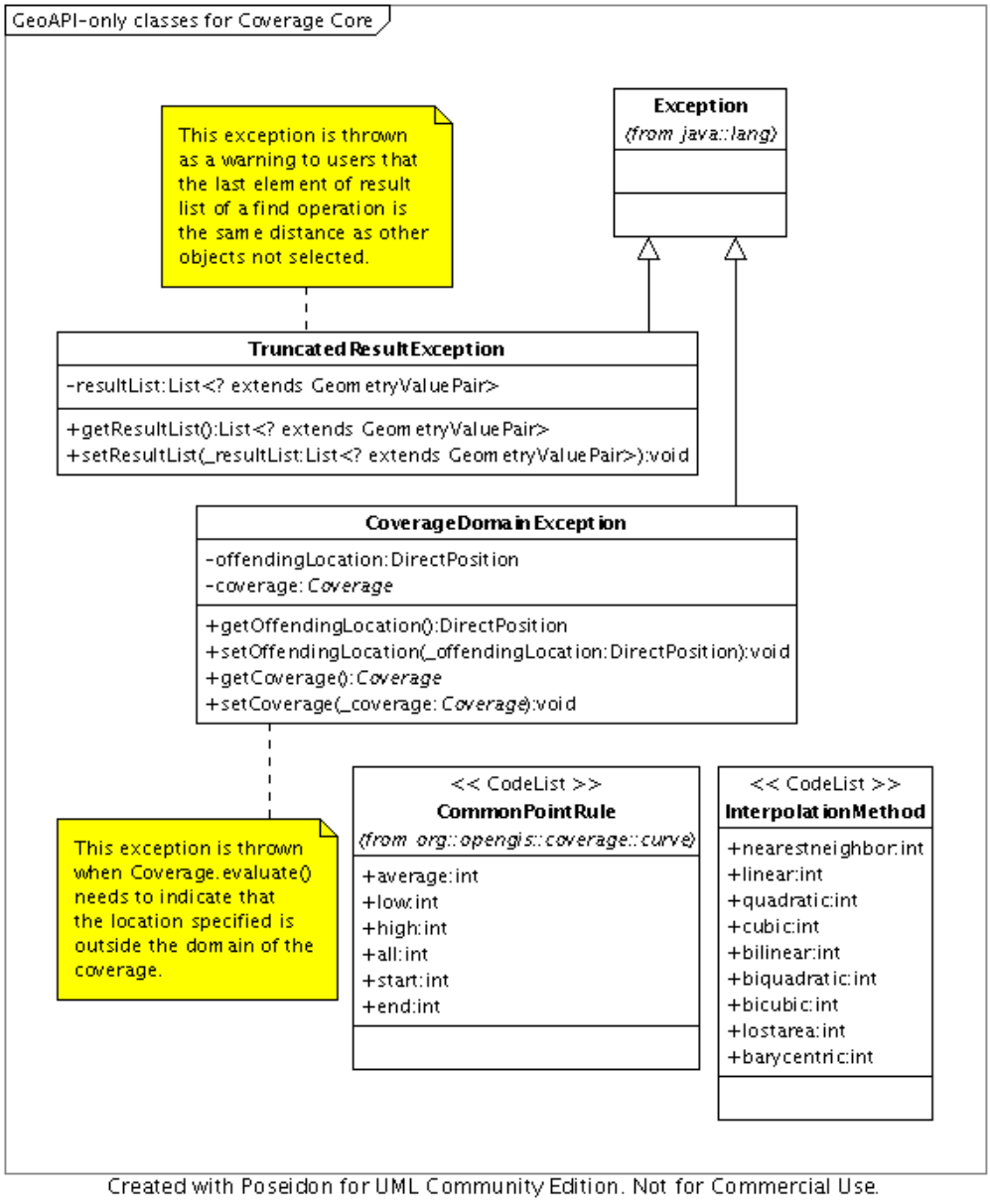

Figure 2: Coverage Core classes and interfaces only represented in GeoAPI.

1.2 Implementation Design

The proposed design for GeoTools coverage core is presented in Figure 3. GeoAPI interfaces are green, and GeoTools classes are yellow. Note that all associations are unidirectional, and every association relates a GeoTools class with a GeoAPI interface. Where possible, attributes are specified without accessor methods. This has been done for readability. Translating these UML diagrams directly into classes (without adding accessor methods) will of course fail to implement the GeoAPI classes in Figure 1. The UML diagram in Figure 3 has not factored in the need for or presence of gateway attributes.

Note that Coverage and ContinuousCoverage are abstract.

Distinctly absent from Figure 3 is any means of setting the various attributes and references. The implementor is intended to consider the means of setting these items.

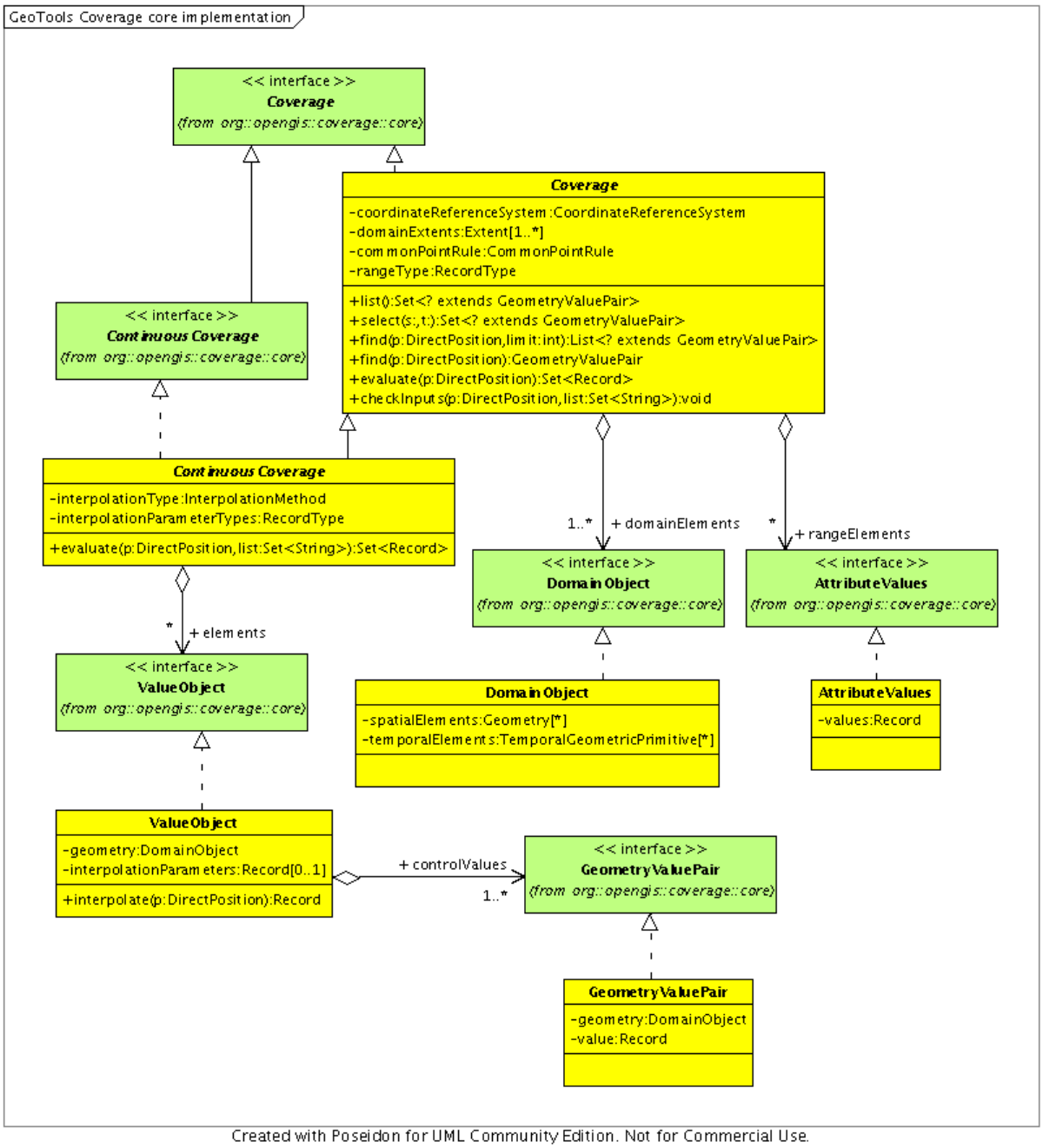

Figure 3: Implementation design for Coverage Core.

1.3 Detailed Discussion

1.3.1 Coverage

The coverage class is the abstract base class from which all coverage types inherit. All coverages act as spatial

functions which determine values for a predetermined set of parameters based on the input location. The precise manner in which this is accomplished is delegated to the children. Some coverages merely manage collections of spatio-temporal objects for later retrieval. Others interpolate values between the data samples they manage. Still others calculate values based solely on time and position.

Some basic characteristics of all coverages:

1. Coverages have a defined extent, which need not be contiguous.
2. Coverages overlay a single coordinate reference system on their constituent data for the purposes of querying.
3. A particular coverage returns the same set of parameters regardless of the location queried. Only the values for these parameters is different.
4. Coverages permit user control over how to handle “multiplicity”. For instance, if the coverage could return more than one value at a particular point, which one should it return?
5. Coverages provide access to their defining data (if they are based on, and therefore manage, sampled data). The defining data may be queried by relationship to a point (find()), or may be queried with respect to a spatio-temporally defined geometry (select()).

1.3.1.1 domainExtents

The extent of a coverage in space, time or space-time. The formulation of this attribute as a set of extents allows for discontinuous coverages.

1.3.1.2 coordinateReferenceSystem

The coordinate reference system specifies the CRS in which the coverage expects to receive DirectPositions in the evaluate and find methods. It also specifies the CRS in which geometry objects should be provided to the select method. This CRS is not necessarily related to the constituent objects which make up the coverage. However, it is likely that an efficient implementation will convert all constituent objects (as well as all queries) into this CRS to provide a common reference system.

1.3.1.3 rangeType

This attribute specifies a RecordType which determines the parameter set handled by this coverage. All records returned from the evaluate() method must correspond to this RecordType. This rangeType also applies to every GeometryValuePair.value attribute returned from select(), find(), and list().

1.3.1.4 commonPointRule

This attribute specifies how a coverage is to handle the situation when multiple answers are possible for a given evaluate() call. In general, commonPointRule specifies simple decisions, like return the low, high, or average value. This attribute applies only to the evaluate method, as the syntax for find and select implies that all relevant constituent data are returned. The possible values for this CodeList are defined in Figure 2.

1.3.1.5 domainElements and rangeElements

This role is how all coverages represent (separately) the location and value portions of their constituent data. There is no relation between elements of these two collections. The multiplicities on these two roleNames are even different. (domainElements is [1..*]; rangeElements is [0..*]) To obtain a 1:1 relation between location and value, use the list() method.

The difference in multiplicities may help formulate an example of when this might be used. Consider a network of weather stations. The locations are known independently of the values. Locations are relatively static, whereas the values receive regular updates. The locations could potentially be located in a text file, whereas the values might need to be retrieved from a website containing the current readings. The coverage would then be required to map instrument ID to location to form GeometryValuePairs prior to supporting any queries on this newly combined dataset.

To continue this example, the weather stations may take data at different rates, or deliver data via different mechanisms. At any given time, the number of domain objects does not necessarily equal the number of range objects.

1.3.1.6 list()

Returns the set of constituent data which has been paired in a 1:1 relationship between location and value. This set is the result of the join operation described in section 1.3.1.5. In the event that the coverage actually maintains a set of GeometryValuePairs, the reverse is true: the domainElements and rangeElements are derived from the maintained set.

If this is an analytical coverage, this method shall return the empty set.

1.3.1.7 select()

This method allows the user to select those constituent elements which “lie within” a given spatio-temporal geometry, which are specified separately. Either the space part or the time part may be specified. It is also legal to specify both a space part and a time part. If both are specified, then the returned elements must “lie within” the space part and the time part.

To “lie within” is the wording of the standard. If the coverage maintains a list of points, then this meaning is clear. For all other component geometries, the meaning is open to interpretation. There is a well known set of topological relations, a subset of which could apply to this situation (e.g., “lie within” could mean “contained”, or “intersects”, but could not mean “disjoint”.) For the purposes of GeoTools, “lie within” shall be interpreted to mean “contained within”.

This method returns the empty set if the coverage is an analytical coverage.

1.3.1.8 find()

Find allows the client to query the coverage with a point and determine the nearest constituent objects to that point. The default is to return the single nearest object. However, the user may specify a “limit” parameter to retrieve a List of objects which are increasingly more distant from the query point. The maximum size of this list is the value of the “limit” parameter. The distance metric is not returned.

The introduction of the limit parameter leads to the possibility that two objects which are the same distance from

the query point, might not both be included in the returned List. In this case, a TruncatedResultException is thrown. The “resultList” property is set to the list which would have been returned had an exception not been thrown. The intent is to warn the user that more data exist at the same distance as the last element in the list, so the selection of the last element in the list was arbitrary.

For non-point objects, “distance” is an ambiguous term. It could mean “distance to the centroid”, or “distance to the closest point on the object.” For the purposes of GeoTools, distance shall be interpreted to mean “distance to the centroid of the geometric object.”

This method returns the empty set (or null, in the case of the “default” limit of 1) if the coverage is an analytical coverage.

Additionally, in the case of spatio-temporal coverages, it is unclear how to weight distances in space with respect to distances in time. For instance, consider a coverage with two constituent objects: one has a spatial location exactly equal to the DirectPosition, but occurs an hour later. The other occurs at the same time, but is three meters distant. Which has a smaller distance from the DirectPosition, therefore first in the returned list? For the purposes of GeoTools, we will consider the find() operation on spatio-temporal coverages to be unsupported. (It remains supported on pure-spatial or pure-temporal coverages.) The find() operation on spatio-temporal coverages shall throw an UnsupportedOperationException.

1.3.1.9 evaluate()

The evaluate() method allows the user to query the coverage for a set of values at a particular location. If more than one potential answer is available, the behavior is controlled by the “commonPointRule” attribute. If the user supplies a “list” of attributes, then only these attributes are returned. It is an error to request an attribute not supported by this coverage. This error will result in a java.lang.NoSuchFieldException. If the user does not supply a list of desired attributes, then all supported attributes are returned.

An error may occur if the client requests values for a location not in the domain of the coverage (as indicated by the domainExtents property.) This error will result in a CoverageDomainException.

The evaluate method is not implemented in the coverage class because Coverage does not know enough about how the results are produced. The convenience method version of evaluate (e.g. without the “list” attribute, and without the NoSuchFieldException) is provided because it does no work particular to any subclass.

1.3.1.10 evaluateInverse()

This is an ill-defined method. It is simple, in theory: given a particular “range” value, return all locations which hold this value. The example given in ISO 19123 is that this method on an elevation coverage could return a contour of given a particular elevation. The Coverage would, of course, interpolate a contour to the desired elevation given the sampled data.

This process becomes complicated if the RecordType of the coverage contains more than one attribute. The units problems noticed in the example about computing a single distance with both space and time components (section 1.3.1.8) do not apply in this case, as a separate interpolation is performed on each field of the Record. However, because the interpolation is performed separately, and the input variables are considered to be independent, two different contours will result. Attempting to generate a contour for more than one property at a time should cause the method to throw an UnsupportedOperationException.

Additionally, contouring is only one interpretation of what this method could do. DiscreteCoverages could

perform a reverse-lookup on their constituent objects, returning DomainObjects associated with Records which are an exact match to the provided data. One might also want a DiscreteCoverage to return a contour. Attempting to define all legitimate functionality of this method is not wise. Concrete coverage impelementations, therefore, should provide decorator objects to provide a nominal set of desired functionality. Because this method largely concerns a query, it is conceivable that the same Coverage could be decorated once with a Contouring and once with a ReverseLookup decorator, each referring to the same copy of data.

1.3.1.11 checkInputs()

This method is not defined by the ISO 19123 specification or by the GeoAPI interfaces. It exists solely as an aid to implementing the evaluate() method in the child classes. It takes the inputs to the evaluate method (position and list of attributes to return) and throws an exception if either error condition documented in section 1.3.1.9 exists. Children may use this method as the first statement in their evaluate() implementation. If no exception is thrown, children may assume that they are responsible for generating a record of feature attribute values. A null list value indicates that all feature attributes are considered to be specified, and therefore, no error results.

1.3.2 Analytical Coverages

Analytical coverages do not appear on either the GeoAPI or GeoTools UML diagrams. This is because Analytical Coverages do not require support in terms of managing a collection of sampled data, and they also do not require support for an interpolation scheme. As such, analytical coverages belong directly under Coverage in the class tree, as a sibling to ContinuousCoverage and DiscreteCoverage. The Coverage class has facilities to support the collection, storage, and retrieval GeometryValuePairs. Left alone, these facilities will behave correctly when inherited by an analytical coverage. There is, therefore, no additional support remaining to give to individual analytical coverages. The only thing remaining is to implement the equation as the evaluate() method and decide how to handle evaluateInverse().

Therefore, every analytical coverage will be a sibling of DiscreteCoverage and ContinuousCoverage, with no parent “AnalyticalCoverage” class.

1.3.3 ContinuousCoverage

A continuous coverage is one of the two main categories of coverage, the alternate category being a discrete coverage. ISO 19123 defines a continuous coverage to be a coverage that returns a “distinct record of feature attribute values for any record within it's domain[1]”. Technically, this definition would include analytical coverages as well, but analytical coverages are not in need of (and may well be hindered by) support for interpolation. Therefore analytical coverages are siblings to ContinuousCoverage and DiscreteCoverage. ContinuousCoverage may then be considered a synonym for an “InterpolatedCoverage”.

A continuous coverage is distinguished by it's domain. Consider again the example of the weather stations. A discrete coverage would consider it's domain to be the locations of the weather stations themselves. The intervening spaces would not be considered part of the domain. On the other hand, a continuous coverage based on those same weather stations would consider it's domain to be at least the convex hull of the weather station locations, and might consider it's domain to be the smallest rectangle in some coordinate system capable

1 As opposed to a discrete coverage, which returns the *same* record of feature attribute values within a *single* domain object in it's domain—in effect returning a constant value everywhere inside a particular constituent object.

of containing all the stations. Given the same input data, the domain of the two coverages is different.

As discussed in section 1.3.1.9, the evaluate() method is obliged to throw a CoverageDomainException if the requested position is outside the domain. The corollary to this is: if the requested position is inside the domain, the coverage must produce and return a record of feature attribute values. There are only two outcomes to a call to evaluate(): a record is returned, or an exception is thrown.

A second defining characteristic of a ContinuousCoverage is that it is considered to be comprised of one or more interpolation objects (e.g. ValueObjects), each of which contains samples between which the interpolation is performed. This differs from a DiscreteCoverage in that the DiscreteCoverage is considered to be directly comprised of the data itself. At their base, however, both types of coverage are defined by a collection of GeometryValuePairs; ContinuousCoverage merely has an extra layer of processing between the raw data and the values which comprise the coverage. This accounts for the fact that most of the methods on the Coverage class deal with “fundamental data units” of GeometryValuePairs, as every child which relies on data to define their values must express their data in that form.

In spite of this conceptual similarity, the impact is that the CoverageFunction relation and the locate() method are defined twice: once in ContinuousCoverage and once in DiscreteCoverage. CoverageFunction and locate()may not be promoted to the Coverage class because the types are different: for DiscreteCoverage, the CoverageFunction represents a set of GeometryValuePairs; for ContinuousCoverage, it represents a set of ValueObjects (which can produce values based on an associated set of GeometryValuePairs.)

ContinuousCoverages use interpolation to fabricate data corresponding to locations not present in its data set. A continuous coverage is affected by the type of interpolation (e.g., linear, bilinear, bicubic, etc.) and by the assumptions made about the data. In the general case, of course, no assumption may be made about the arrangement of the data samples. However, in many cases, great efficiency gains may be realized if the data are structured in some manner.

The children of this class defined by ISO 19123 differ primarily in the assumptions they make about how data samples are arranged. Each child takes different shortcuts to speed the calculation of data, and each shortcut is based on the structure of the data samples.

1.3.3.1 interpolationType

This is a parameter which controls the type of interpolation performed on the data to produce an answer. The allowable interpolation types may be dependent on the arrangement of data samples. The InterpolationType codelist is presented in Figure 2. ISO 19123 specifies that this parameter is optional in the case of an analytical coverage. However, since analytical coverages are handled separately, this parameter should be considered mandatory.

1.3.3.2 interpolationParameterTypes

Given the type of interpolation specified by the interpolationType attribute, this RecordType specifies the parameters which are necessary to perform the interpolation.

1.3.3.3 locate()

The locate method performs a function similar to Coverage.select(), but uses a DirectPosition as the specified location, and returns the set of interpolator objects (section 1.3.4) which interpolate to the desired location. If the

continuous coverage does not maintain a persistent set of interpolator objects, then one must be constructed to be returned. This method must always return an interpolator object if the specified position is within the extent of the coverage. If the position is outside the coverage domain, locate returns null.

This method is present for all ContinuousCoverage objects, but the location of a particular subset of coverage values and the instantiation of an interpolator represents specialized knowledge that subclasses possess about the data structure. The implementation, therefore, is deferred to specific subclasses.

1.3.3.4 evaluate()

The ContinuousCoverage class does not know anything about the particular arrangement of data samples, nor is it acquainted with any of the particular interpolation implementation objects. However, the child class does know these things and a call to evaluate(p) on any ContinuousCoverage should be shorthand for a call to locate(p).interpolate(p). If a list of feature attributes is supplied, of course, the appropriate attributes should be copied to the record which is returned.

1.3.3.5 elements

The elements role relates the continuous coverage to the set of interpolation objects used to evaluate the coverage. These ValueObjects need not be persistent (section 1.3.4), and so the association is optional. However, if it is possible to produce a small number of ValueObjects (or even a single one) which is associated with many (or all) of the constituent data, it may be advantageous to initialize this component once and maintain a reference to it.

1.3.4 ValueObject

A ValueObject maintains references to data samples and provides for interpolation between those samples. A ContinuousCoverage is composed of one or more of these objects. If a coverage is composed of many ValueObjects, efficiency or memory constraints may dictate that they should be generated on the fly. If a few ValueObjects cover the entire domain of the coverage, then creating a single object may be called for.

The basic implementation provides methods to handle a reference to a geometry, interpolationParameters, and the controlValues. However, implementation of the interpolate method is deferred to the particular implementation.

1.3.4.1 geometry

This parameter indicates the domain over which this ValueObject is willing to interpolate data. This domain will depend on the extent of the raw data which backs this interpolator.

1.3.4.2 interpolationParameters

The interpolation parameters control the process of interpolation. The specific parameters present must correspond to the interpolationParameterTypes in the associated ContinuousCoverage. Note that this cannot be verified by the ValueObject, since there is no reference to the containing coverage. The ContinuousCoverage must ensure this.

1.3.4.3 controlValues

This set of GeometryValuePairs represents the data behind the interpolator object. This could be as simple as the two points which bound a linear interpolation, or as complex as a grid of points within which a gridded interpolator provides bilinear, bicubic, etc. interpolation.

1.3.4.4 interpolate

The interpolate function calculates a value anywhere within the domain defined in the “geometry” attribute. If the supplied location is outside the defined domain, it throws a CoverageDomainException.

1.3.5 AttributeValues

An attribute value represents a single element of the range of a coverage. This range value is dissociated from the corresponding domain value, as per the discourse in section 1.3.1.5. This class appears to be superfluous, as there is a single contained object which does all the work. When the GeometryValuePair class combines the “domainElement” and “rangeElement” into a single data structure, it does use the AttributeValue structure, preferring to just use the contained Record directly.

1.3.5.1 values

This is the record of feature attribute values for this particular coverage element. It is not mapped to a corresponding location.

1.3.6 DomainObject

A DomainObject is the spatio-temporal position which can be associated with a value in a GeometryValuePair. The domain object separates spatial information from temporal information. A DomainObject is a single element in the domain of a coverage, able to handle whatever meaning is imparted to the word “element.”

At least one of spatialElements or temporalElements must be defined. Both may be defined if required. If both are defined, then the domain object refers to the aggregate spatial location and-ed with the aggregate temporal location.

1.3.6.1 spatialElements

The spatial component of a domain element. The aggregate spatial location referred to by this object is the logical “or” of all specified spatial elements.

1.3.6.2 temporalElements

The temporal component of a domain element. The aggregate temporal location referred to by this object is the logical “or” of all specified temporal elements.

1.3.7 GeometryValuePair

The GeometryValuePair is a 1:1 mapping between domain and range elements of a coverage. In function, it is

very similar to a geospatial java.util.Map.Entry class.

1.3.7.1 geometry

The domain part of the domain/range association. The term “geometry” is misleading because the DomainObject to which it refers has the capability of referring to both a spatial and a temporal element.

1.3.7.2 value

The value part of the of the domain/range association. This record must contain attributes which correspond to the rangeType property of the associated Coverage.

1.4 Observations

The representation of the defining coverage data as collections of GeometryValuePairs is certain to cause memory problems. First, the defining data is unlikely to be in that format, so it must be converted. Second, this format is extremely inefficient as each value has an explicitly associated location. Such an arrangement negates any memory efficiency associated with structured storage schemes, like grids.

Some means must be found, for efficiency's sake, to write middleware which interacts directly with common, efficient memory storage formats and represents this data as the correct collection of GeometryValuePairs. Perhaps an iterator could create these GeometryValuePairs on demand, where (hopefully) the client will perform some minimal processing on the object, then discard it. Until such a method is developed, certain methods with a high potential to always fail (like Coverage.list() on a DiscreteGridPointCoverage) should probably throw an UnsupportedOperationException. If they do not, they will be forced to convert the entire image to location/value pairs, add them to a set, and return them to the caller.

1.5 Modeling Pre-work

1.5.1 Classification of GeoAPI methods

TODO: Use the following spreadsheet object to classify the GeoAPI methods. Double-click the spreadsheet (and not the frame) to edit..

GeoAPI Package: org.opengis.xxx

GeoAPI Class/Interface:		
Method	Classification (A, R, O)	Type

Table 1: GeoAPI Method Classification for Coverage Core.

1.5.2 Interface and Data Type Proxies

TODO: Use the following spreadsheet object to list the Interface and Data Type proxies required for this implementation effort.

GeoAPI Package:

Interface Proxies Data Type Proxies

Table 2: Proxy types for Coverage Core

1.6 Test cases

TODO: Describe potential test cases and indicate whether they have been implemented...

Name	***Done?***	***Description***
Evaluate convenience method returns correct value	y/n	does the Coverage.evaluate(p:DirectPosition) method correctly relay the return value of the Coverage.evaluate(p:DirectPosition, null) method.
checkInputs method throws correct exceptions.	y/n	Does Coverage.checkInputs(p:DirectPosition, list:Set<String>) behave correctly under the following conditions: 1. p is inside extent, list contains only legal values 2. p is inside extent, list is null 3. p is outside extent, list contains only legal values 4. p is outside extent, list is null 5. p is null (should throw NullPointerException) 6. p is inside or outside extent, list contains illegal value.
Evaluate() method of ContinuousCoverage	y/n	Does ContinuousCoverage.evaluate(p:DirectPosition, list:Set<String>) correctly relay return value from this.locate(p).interpolate(p)? Does it filter for only the requested values?

Table 3: Potential Test Cases for Coverage Core