Open Geospatial Consortium, Inc.

OpenGIS® Implementation Specification: Coordinate Transformation Services

Revision 1.00

OpenGIS Project Document 01-009
Release Date: 12 January 2001

Copyright © Open Geospatial Consortium, Inc. (2005)

To obtain additional rights of use, visit http://www.opengeospatial.org/legal/

THIS PAGE LEFT INTENTIONALLY BLANK

Table of Contents

1 Preface 10
1.1 Submitting Companies 10
1.2 Submission Contact Point 10
1.3 Revision History 10
1.3.1 00-007r4 10
2 Overview 11
3 Architecture 12
3.1 Decomposition into Packages 12
3.2 Naming Conventions 12
3.3 Object Mutability 12
3.4 Error Handling 12
3.5 Mandatory Parts 13
3.6 Constraints on Operation Sequencing 13
4 Coordinate System Identity 14
5 Domains of Validity 15
6 Profiles 16
6.1 COM profile 16
6.1.1 Generating COM Type Libraries 16
6.1.2 COM Categories 16
6.2 CORBA profile 17
6.3 Java profile 17
7 Well-Known Text format 18
7.1 Math Transform WKT 18
7.2 Coordinate System WKT 18
7.3 Description of WKT keywords 19
7.3.1 AUTHORITY 19
7.3.2 AXIS 19
7.3.3 COMPD_CS 19
7.3.4 CONCAT_MT 19
7.3.5 DATUM 20
7.3.6 FITTED_CS 20
7.3.7 GEOCCS 20
7.3.8 GEOGCS 20
7.3.9 INVERSE_MT 20
7.3.10 LOCAL_CS 20
7.3.11 PARAMETER 20
7.3.12 PARAM_MT 20
7.3.13 PASSTHROUGH_MT 21
7.3.14 PRIMEM 21
7.3.15 PROJCS 21
7.3.16 PROJECTION 21
7.3.17 SPHEROID 21
7.3.18 TOWGS84 22
7.3.19 UNIT 22
7.3.20 VERT_DATUM 22
7.3.21 VERT_CS 22
7.4 WKT Example 23
8 Relationship to Simple Features 24
8.1 Compatibility of Well-Known Text 24
8.2 Name changes from SF-SRS to CS 24
8.3 Authority Codes 24
9 Processes 25
9.1 Create a NAD27 California Zone 1 CS 25

9.2 Create a NAD83 California Zone 1 CS 25
9.3 Store and recall CS in a Database....25
9.4 Merge California Zone 1 data in NAD27 and NAD83 26
9.5 Transform steps from NAD27 to NAD83 in California Zone 1 27
10 Parameterized Transforms 30
10.1 Ellipsoid_To_Geocentric....30
10.2 Geocentric_To_Ellipsoid....31
10.3 Abridged_Molodenski....32
10.4 Affine....32
10.5 Longitude_Rotation 33
10.6 Cartographic Projection Transforms....34
10.6.1 Cartographic Projection Transform Parameters 35
10.6.1.1 Transverse_Mercator Projection 35
10.6.1.2 Lambert_Conformal_Conic_1SP....35
10.6.1.3 Lambert_Conformal_Conic_2SP....35
11 Referencing objects by ID versus Value....36
11.1 Registered WKT Authorities 36
11.1.1 EPSG/POSC 36
12 UML Documentation 38
12.1 Packages....38
12.2 PT Logical Package: Positioning....39
12.2.1 PT_CoordinatePoint Class.... 40
12.2.1.1 ord (Attribute) :....40
12.2.2 PT_Envelope Class 41
12.2.2.1 maxCP (Attribute) : 41
12.2.2.2 minCP (Attribute) : 41
12.2.3 PT_Matrix Class.... 42
12.2.3.1 elt (Attribute) : 42
12.3 CS Logical Package: Co-ordinate Systems 43
12.3.1 CS_AngularUnit Class 48
12.3.1.1 radiansPerUnit (Attribute) : 48
12.3.2 CS_AxisInfo Class 49
12.3.2.1 name (Attribute) : 49
12.3.2.2 orientation (Attribute) : 49
12.3.3 CS_AxisOrientationEnum Class 50
12.3.3.1 CS_AO_Other (Attribute) : 0....50
12.3.3.2 CS_AO_North (Attribute) : 1....50
12.3.3.3 CS_AO_South (Attribute) : 2 50
12.3.3.4 CS_AO_East (Attribute) : 3 50
12.3.3.5 CS_AO_West (Attribute) : 4 50
12.3.3.6 CS_AO_Up (Attribute) : 5....50
12.3.3.7 CS_AO_Down (Attribute) : 6 50
12.3.4 CS_CompoundCoordinateSystem Class.... 51
12.3.4.1 headCS (Attribute) : 51
12.3.4.2 tailCS (Attribute) : 51
12.3.5 CS_CoordinateSystem Class 52
12.3.5.1 defaultEnvelope (Attribute) : 52
12.3.5.2 dimension (Attribute) :....52
12.3.5.3 getAxis (dimension:Integer) : CS_AxisInfo....52
12.3.5.4 getUnits (dimension:Integer) : CS_Unit....52
12.3.6 CS_CoordinateSystemAuthorityFactory Class.... 53
12.3.6.1 authority (Attribute) : 53
12.3.6.2 createAngularUnit (code:CharacterString) : CS_AngularUnit 53
12.3.6.3 createCompoundCoordinateSystem (code:CharacterString) : CS_CompoundCoordinateSystem 53
12.3.6.4 createEllipsoid (code:CharacterString) : CS_Ellipsoid 53

12.3.6.5 createGeographicCoordinateSystem (code:CharacterString) : CS_GeographicCoordinateSystem 53
12.3.6.6 createHorizontalCoordinateSystem (code:CharacterString) : CS_HorizontalCoordinateSystem 53
12.3.6.7 createHorizontalDatum (code:CharacterString) : CS_HorizontalDatum 53
12.3.6.8 createLinearUnit (code:CharacterString) : CS_LinearUnit 53
12.3.6.9 createPrimeMeridian (code:CharacterString) : CS_PrimeMeridian 53
12.3.6.10 createProjectedCoordinateSystem (code:CharacterString) : CS_ProjectedCoordinateSystem 53
12.3.6.11 createVerticalCoordinateSystem (code:CharacterString) : CS_VerticalCoordinateSystem 53
12.3.6.12 createVerticalDatum (code:CharacterString) : CS_VerticalDatum 53
12.3.6.13 descriptionText (code:CharacterString) : CharacterString 53
12.3.6.14 geoidFromWKTName (wkt:CharacterString) : CharacterString 54
12.3.6.15 wktGeoidName (geoid:Integer) : CharacterString 54
12.3.7 CS_CoordinateSystemFactory Class 55
12.3.7.1 createCompoundCoordinateSystem (name:CharacterString, head:CS_CoordinateSystem, tail:CS_CoordinateSystem) : CS_CompoundCoordinateSystem 55
12.3.7.2 createEllipsoid (name:CharacterString, semiMajorAxis:Double, semiMinorAxis:Double, linearUnit:CS_LinearUnit) : CS_Ellipsoid 55
12.3.7.3 createFittedCoordinateSystem (name:CharacterString, base:CS_CoordinateSystem, toBaseWKT:CharacterString, arAxes:Sequence<CS_AxisInfo>) : CS_FittedCoordinateSystem 55
12.3.7.4 createFlattenedSphere (name:CharacterString, semiMajorAxis:Double, inverseFlattening:Double, linearUnit:CS_LinearUnit) : CS_Ellipsoid 55
12.3.7.5 createFromWKT (wellKnownText:CharacterString) : CS_CoordinateSystem ..55
12.3.7.6 createFromXML (xml:CharacterString) : CS_CoordinateSystem....55
12.3.7.7 createGeographicCoordinateSystem (name:CharacterString, angularUnit:CS_AngularUnit, horizontalDatum:CS_HorizontalDatum, primeMeridian:CS_PrimeMeridian, axis0:CS_AxisInfo, axis1:CS_AxisInfo) : CS_GeographicCoordinateSystem 56
12.3.7.8 createHorizontalDatum (name:CharacterString, horizontalDatumType:CS_DatumType, ellipsoid:CS_Ellipsoid, toWGS84:CS_WGS84ConversionInfo) : CS_HorizontalDatum 56
12.3.7.9 createLocalCoordinateSystem (name:CharacterString, datum:CS_LocalDatum, unit:CS_Unit, arAxes:Sequence<CS_AxisInfo>) : CS_LocalCoordinateSystem 56
12.3.7.10 createLocalDatum (name:CharacterString, localDatumType:CS_DatumType) : CS_LocalDatum....56
12.3.7.11 createPrimeMeridian (name:CharacterString, angularUnit:CS_AngularUnit, longitude:Double) : CS_PrimeMeridian 56
12.3.7.12 createProjectedCoordinateSystem (name:CharacterString, gcs:CS_GeographicCoordinateSystem, projection:CS_Projection, linearUnit:CS_LinearUnit, axis0:CS_AxisInfo, axis1:CS_AxisInfo) : CS_ProjectedCoordinateSystem....56
12.3.7.13 createProjection (name:CharacterString, wktProjectionClass:CharacterString, parameters:Sequence<CS_ProjectionParameter>) : CS_Projection....56
12.3.7.14 createVerticalCoordinateSystem (name:CharacterString, verticalDatum:CS_VerticalDatum, verticalUnit:CS_LinearUnit, axis:CS_AxisInfo) : CS_VerticalCoordinateSystem 56
12.3.7.15 createVerticalDatum (name:CharacterString, verticalDatumType:CS_DatumType) : CS_VerticalDatum 57
12.3.8 CS_Datum Class 58
12.3.8.1 datumType (Attribute) : 58
12.3.9 CS_DatumType Class 59
12.3.9.1 CS_HD_Classic (Attribute) : 1001 59

12.3.9.2 CS_HD_Geocentric (Attribute) : 1002 59
12.3.9.3 CS_HD_Max (Attribute) : 1999 59
12.3.9.4 CS_HD_Min (Attribute) : 1000 59
12.3.9.5 CS_HD_Other (Attribute) : 1000 59
12.3.9.6 CS_LD_Max (Attribute) : 32767 59
12.3.9.7 CS_LD_Min (Attribute) : 10000 59
12.3.9.8 CS_VD_AltitudeBarometric (Attribute) : 2003 59
12.3.9.9 CS_VD_Depth (Attribute) : 2006 59
12.3.9.10 CS_VD_Ellipsoidal (Attribute) : 2002 59
12.3.9.11 CS_VD_GeoidModelDerived (Attribute) : 2005 60
12.3.9.12 CS_VD_Max (Attribute) : 2999 60
12.3.9.13 CS_VD_Min (Attribute) : 2000 60
12.3.9.14 CS_VD_Normal (Attribute) : 2004 60
12.3.9.15 CS_VD_Orthometric (Attribute) : 2001 60
12.3.9.16 CS_VD_Other (Attribute) : 2000 60
12.3.10 CS_Ellipsoid Class 61
12.3.10.1 axisUnit (Attribute) : 61
12.3.10.2 inverseFlattening (Attribute) : 61
12.3.10.3 ivfDefinitive (Attribute) : 61
12.3.10.4 semiMajorAxis (Attribute) : 61
12.3.10.5 semiMinorAxis (Attribute) : 61
12.3.11 CS_FittedCoordinateSystem Class 62
12.3.11.1 baseCoordinateSystem (Attribute) : 62
12.3.11.2 toBase (Attribute) : 62
12.3.12 CS_GeocentricCoordinateSystem Class 62
12.3.12.1 horizontalDatum (Attribute) : 62
12.3.12.2 linearUnit (Attribute) : 62
12.3.12.3 primeMeridian (Attribute) : 62
12.3.13 CS_GeographicCoordinateSystem Class 63
12.3.13.1 angularUnit (Attribute) : 63
12.3.13.2 numConversionToWGS84 (Attribute) : 63
12.3.13.3 primeMeridian (Attribute) : 63
12.3.13.4 getWGS84ConversionInfo (index:Integer) : CS_WGS84ConversionInfo 63
12.3.14 CS_HorizontalCoordinateSystem Class 64
12.3.14.1 horizontalDatum (Attribute) : 64
12.3.15 CS_HorizontalDatum Class 65
12.3.15.1 ellipsoid (Attribute) : 65
12.3.15.2 WGS84Parameters (Attribute) : 65
12.3.16 CS_Info Class 66
12.3.16.1 abbreviation (Attribute) : 66
12.3.16.2 alias (Attribute) : 66
12.3.16.3 authority (Attribute) : 66
12.3.16.4 authorityCode (Attribute) : 66
12.3.16.5 name (Attribute) : 66
12.3.16.6 remarks (Attribute) : 66
12.3.16.7 WKT (Attribute) : 66
12.3.16.8 XML (Attribute) : 67
12.3.17 CS_LinearUnit Class 68
12.3.17.1 metersPerUnit (Attribute) : 68
12.3.18 CS_LocalCoordinateSystem Class 69
12.3.18.1 localDatum (Attribute) : 69
12.3.19 CS_LocalDatum Class 70
12.3.20 CS_PrimeMeridian Class 71
12.3.20.1 angularUnit (Attribute) : 71
12.3.20.2 longitude (Attribute) : 71
12.3.21 CS_ProjectedCoordinateSystem Class 72

12.3.21.1 geographicCoordinateSystem (Attribute) : 72
12.3.21.2 linearUnit (Attribute) : 72
12.3.21.3 projection (Attribute) : 72
12.3.22 CS_Projection Class 73
12.3.22.1 className (Attribute) : 73
12.3.22.2 numParameters (Attribute) : 73
12.3.22.3 getParameter (index:Integer) : CS_ProjectionParameter 73
12.3.23 CS_ProjectionParameter Class 74
12.3.23.1 name (Attribute) : 74
12.3.23.2 value (Attribute) : 74
12.3.24 CS_Unit Class 75
12.3.25 CS_VerticalCoordinateSystem Class 76
12.3.25.1 verticalDatum (Attribute) : 76
12.3.25.2 verticalUnit (Attribute) : 76
12.3.26 CS_VerticalDatum Class 77
12.3.27 CS_WGS84ConversionInfo Class 78
12.3.27.1 areaOfUse (Attribute) : 78
12.3.27.2 dx (Attribute) : 78
12.3.27.3 dy (Attribute) : 78
12.3.27.4 dz (Attribute) : 78
12.3.27.5 ex (Attribute) : 78
12.3.27.6 ey (Attribute) : 78
12.3.27.7 ez (Attribute) : 78
12.3.27.8 ppm (Attribute) : 78
12.4 CT Logical Package: Co-ordinate Transformations 79
12.4.1 CT_CoordinateTransformation Class 80
12.4.1.1 areaOfUse (Attribute) : 80
12.4.1.2 authority (Attribute) : 80
12.4.1.3 authorityCode (Attribute) : 80
12.4.1.4 mathTransform (Attribute) : 80
12.4.1.5 name (Attribute) : 80
12.4.1.6 remarks (Attribute) : 80
12.4.1.7 sourceCS (Attribute) : 80
12.4.1.8 targetCS (Attribute) : 80
12.4.1.9 transformType (Attribute) : 80
12.4.2 CT_CoordinateTransformationAuthorityFactory Class 81
12.4.2.1 authority (Attribute) : 81
12.4.2.2 createFromCoordinateSystemCodes (sourceCode:CharacterString, targetCode: CharacterString) : CT_CoordinateTransformation 81
12.4.2.3 createFromTransformationCode (code:CharacterString) : CT_CoordinateTransformation 81
12.4.3 CT_CoordinateTransformationFactory Class 82
12.4.3.1 createFromCoordinateSystems (sourceCS:CS_CoordinateSystem, targetCS:CS_CoordinateSystem) : CT_CoordinateTransformation 82
12.4.4 CT_DomainFlags Class 83
12.4.4.1 CT_DF_Inside (Attribute) : 1 83
12.4.4.2 CT_DF_Outside (Attribute) : 2 83
12.4.4.3 CT_DF_Discontinuous (Attribute) : 4 83
12.4.5 CT_MathTransform Class 84
12.4.5.1 dimSource (Attribute) : 84
12.4.5.2 dimTarget (Attribute) : 84
12.4.5.3 identity (Attribute) : 84
12.4.5.4 WKT (Attribute) : 84
12.4.5.5 XML (Attribute) : 84
12.4.5.6 derivative (cp:PT_CoordinatePoint) : PT_Matrix 84
12.4.5.7 getCodomainConvexHull (ord:Sequence<Double>) : Sequence<Double> 84

12.4.5.8 getDomainFlags (ord:Sequence<Double>) : CT_DomainFlags....85
12.4.5.9 inverse () : CT_MathTransform....85
12.4.5.10 transform (cp:PT_CoordinatePoint) : PT_CoordinatePoint....85
12.4.5.11 transformList (ord:Sequence<Double>) : Sequence<Double>....85
12.4.6 CT_MathTransformFactory` Class....86
12.4.6.1 createAffineTransform (matrix:PT_Matrix) : CT_MathTransform....86
12.4.6.2 createConcatenatedTransform (transform1:CT_MathTransform, transform2:CT_MathTransform) : CT_MathTransform....86
12.4.6.3 createFromWKT (wellKnownText:CharacterString) : CT_MathTransform....87
12.4.6.4 createFromXML (xml:CharacterString) : CT_MathTransform....87
12.4.6.5 createParameterizedTransform (classification:CharacterString, parameters:Sequence<CT_Parameter>) : CT_MathTransform....87
12.4.6.6 createPassThroughTransform (firstAffectedOrdinate:Integer, subTransform:CT_MathTransform) : CT_MathTransform....87
12.4.6.7 isParameterAngular (parameterName:CharacterString) : Boolean....87
12.4.6.8 isParameterLinear (parameterName:CharacterString) : Boolean....87
12.4.7 CT_Parameter Class....88
12.4.7.1 name (Attribute) :....88
12.4.7.2 value (Attribute) :....88
12.4.8 CT_TransformType Class....89
12.4.8.1 CT_TT_Other (Attribute) : 0....89
12.4.8.2 CT_TT_Conversion (Attribute) : 1....89
12.4.8.3 CT_TT_Transformation (Attribute) : 2....89
12.4.8.4 CT_TT_ConversionAndTransformation (Attribute) : 3....89
13 Appendix A: Interface Definition Files....90
13.1 COM....90
13.1.1 OGC_PT.IDL....90
13.1.2 OGC_CS.IDL....91
13.1.3 OGC_CT.IDL....100
13.2 CORBA....103
13.2.1 OGC_PT.IDL....103
13.2.2 OGC_CS.IDL....104
13.2.3 OGC_CT.IDL....108
13.3 Java....110
14 Appendix B: Guidelines for development of Conformance Test....111
14.1 Ask vendors which profiles will be implemented....111
14.2 Ask TC members to list possible errors....111
14.3 List core mandatory transforms and projections....111
14.4 List coordinate systems in WKT and XML....111
14.5 Acquire reference implementation of projection code....112
14.6 Prepare list of known points in different coordinate systems....112
15 Appendix C: Informative....113
15.1 XML....113
15.1.1 XML Definition....113
15.1.2 XML Example....116

Table of Figures

Figure 1: Package interface dependencies 12
Figure 2: Classifications of parameterized math transforms 30
Figure 3: EPSG code ranges 36
Figure 4. Packages 38
Figure 5. Positioning (Class Diagram) 39
Figure 6. Coordinate System (Class Diagram) 43
Figure 7. Coordinate System Classes (Class Diagram) 44
Figure 8. Datums (Class Diagram) 45
Figure 9. Units (Class Diagram) 46
Figure 10. Coordinate System Factories (Class Diagram) 47
Figure 11. Coordinate Transformation (Class Diagram) 79

1 Preface

1.1 Submitting Companies

The following company is pleased to submit this specification in response to the OGC Request 9, "A Request for Proposals: OpenGIS Coordinate Transformation Services" (OpenGIS Project Document Number 99-057):

Computer Aided Development Corporation (Cadcorp) Ltd.

1.2 Submission Contact Point

All questions about the submission should be directed to:

Martin Daly,
Cadcorp Ltd,
Sterling Court,
Norton Road,
Stevenage,
Herts, UK
SG1 2JY

E-mail: martin@cadcorpdev.com

Thanks are due to Arliss Whiteside for his assistance in automatically generating much of this document from the accompanying UML model.

1.3 Revision History

1.3.1 00-007r4

Substantial changes have been made to this document in the interests of clarity. The significant changes to the specification are as follows:

- The XML specified in 00-007r3 has been moved to an informative Appendix.
- The CS_Info.code, and CT_CoordinateTransformation.code, attributes have been renamed authorityCode, and converted from a Long Integer to a String.

2 Overview

This Implementation Specification provides interfaces for general positioning, coordinate systems, and coordinate transformations.

Coordinates can have any number of dimensions. So this specification can handle 2D and 3D coordinates, as well as 4D, 5D etc.

In order to handle any number of dimensions, this specification provides a Coordinate System package that could eventually replace the Spatial Reference package contained in the Simple Features specifications. However, it has been designed to work in conjunction with Simple Features during any transition period.

This Implementation Specification anticipates the adoption of more advanced geometry interfaces. So these interfaces do not reference the IGeometry interface or the WKB format from Simple Features. Convex hulls are use where geometry is required (e.g. to define the domains of transformation functions). Convex hulls are defined from a list of points.

This document can be read in conjunction with the attached HTML files and UML model, which contain the same information in a more structured format.

The definitive statement of the DCP profiles is expressed in the attached Java source, and IDL files (COM and CORBA). Large parts of this document were generated from the same abstract model that was used to generate the DCP profiles. If the abstract model is revised this document may temporarily lag behind.

3 Architecture

3.1 Decomposition into Packages

The interfaces for Coordinate Transformation Services are split into three packages:

a) Positioning (PT)
b) Coordinate Systems (CS)
c) Coordinate Transformations (CT)

The interfaces in these packages have the following dependency relationships:

Package	Interfaces depend on…
PT	None
CS	PT
CT	PT CS

Figure 1: Package interface dependencies

3.2 Naming Conventions

All interfaces, structures and enumerations start with a two letter prefix indicating their package. For example, all interfaces in the Coordinate System package start with the prefix “CS_”. This convention both makes client code easier to understand, and resolves issues with re-using common names, e.g. Parameter.

All method, parameter and property names are consistent with the normal conventions for the DCP in question.

3.3 Object Mutability

All interfaced objects are immutable. This means that implementations must promise not to change an object’s internal state once they have handed out an interface pointer.

All mutable entities are implemented with structures in the COM and CORBA profiles, and classes in the Java profile. These language elements allow for return-by-value and pass-by-value semantics.

3.4 Error Handling

All errors are handled using the normal method for the DCP.

In the COM profile, an implementation that fails to perform an interface method can return a HRESULT value of E_FAIL. Successful method functions will return S_OK. In most COM client environments (e.g. Visual Basic and ATL C++) these return codes are processed by client-side stub code, and errors are converted into exceptions that the client application must catch. C client programmers should check all the HRESULT return values with the SUCCEEDED macro.

In the Java profile, an implementation that fails to perform an interface method should throw a runtime exception.

No exceptions have been specified in the CORBA profile.

3.5 Mandatory Parts

In order to make client software as easy as possible to write, *all* of the interfaces specified in each package are mandatory, with the exception of the 'Authority' factories. The reason for this is that if portions of the specification are optional, then, in practice, client software will be programmed to use only the mandatory parts.

For example, if an implementation were to implement the CS_CoordinateSystem interface, but not the CS_GeographicCoordinateSystem, CS_ProjectedCoordinateSystem, etc., interfaces, then the only way to establish the type of a CS_CoordinateSystem object would be to decode either the WKT or the XML.

This does not however mean that all portions of the specification *have* to be implemented. For example, an implementation could support the CS_CoordinateSystemFactory interface, but not support the createVerticalCoordinateSystem method.

In this version of the document, the XML Definition is 'informative', and is therefore optional.

See also Parameterized Transforms.

3.6 Constraints on Operation Sequencing

Since all interfaced objects are specified to be immutable, there do not need to be any constraints on operation sequencing. This means that these interfaces can be used in parallel computing environments (e.g. internet servers).

4 Coordinate System Identity

This specification uses two character strings for coordinate system identity. The first string identifies the “authority” or “nameSpace” that specifies multiple standard reference systems, e.g. “EPSG”. The second string specifies the “authority code” or “name” of a particular reference system specified by that authority. In the case of “EPSG”, the “authority code” will be a string representation of an integer. See CS_Info Class and Well-Known Text format: AUTHORITY.

This specification does not currently support editions or versions of reference systems. That is, no explicit way is provided to represent the edition of a reference system “authority” or “authority code”. If multiple editions exist for a reference system, the interfaces assume that the latest edition is intended.

5 Domains of Validity

Every GIS coordinate system, math transform, and coordinate transformation has a domain of validity, which is known as its ‘domain’.

For a GIS coordinate system, the domain is the set of coordinates that correspond to a position in the real world, to an acceptable level of accuracy.

For a math transform, the domain is the set of coordinate positions that can be transformed without mathematical problems, such as dividing by zero.

For a coordinate transformation, the domain can be imagined as the intersection of three domains: two domains from the source and target coordinate systems, and the domain of the math transform. (In fact, it's not quite this simple, since the intersection cannot be formed until the three domain shapes are in the same coordinate system.)

While working inside a single coordinate system, the domain is usually not critical. Since many GIS systems (but not all) use floating-point coordinates, you can place points slightly outside the coordinate system domain without causing any immediate problem. So the interface method to get the domain of a coordinate system can be quite simple. (See CS_CoordinateSystem.getDefaultEnvelope.)

However, while transforming coordinates, the exact shape of these domains is often critical, since you cannot work outside them without problems, but you often want to work right up to the edges of them. Since math transform domains can, and often do have highly irregular shapes, it is not possible to have an interface method that returns the shape explicitly. Instead, the math transform domain is described implicitly. This means that you have to ask whether a particular ‘elementary shape’ is inside the domain, outside the domain, or partially inside and partially outside. The elementary shape that the CT_MathTransform interface uses is the convex hull. A convex hull is a shape in a coordinate system, where if two positions A and B are inside the shape, then all positions in the straight line between A and B are also inside the shape. So in 3D a cube and a sphere are both convex hulls. Other less obvious examples of convex hulls are straight lines, and single points. (A single point is a convex hull, because the positions A and B must both be the same - i.e. the point itself. So the straight line between A and B has zero length.)

Some examples of shapes that are NOT convex hulls are donuts, and horseshoes.
The simplest type of complex hull is a single point. So you can easily ask whether a point is inside, or outside the domain of a math transform.

6 Profiles

All packages have been profiled for COM, CORBA and Java.

In order to make the different DCP profiles as compatible as possible, we have avoided using too many constructs. We have assumed that all DCPs have constructs for the following:

a) Atomic types of CharacterString, Boolean, Integer and Double
b) Structures passed and returned by value
c) Fixed length arrays
d) Variable length sequences (including 2D)
e) Structures containing atomic types, arrays and structures

We have NOT used the following constructs:

a) Multiple Inheritance of interfaces
b) Inheritance of structures
c) Native DCP dictionaries
d) Pass-by-value of entities with behavior
e) Sequences of interfaces

6.1 COM profile

The COM profile is specified in the attached MIDL files (see Appendix A: Interface Definition Files: COM). The files are named OGC_*XX*.IDL, where *XX* indicates the package name. (See Naming Conventions.)

The dependencies between the packages are expressed using the MIDL keyword “import”.

6.1.1 Generating COM Type Libraries

Each MIDL file can be used to generate a type library (TLB) file by using the MIDL compiler as follows:

```
midl /D _OGC_PT_TLB_ OGC_PT.idl
midl /D _OGC_CS_TLB_ OGC_CS.idl
midl /D _OGC_CT_TLB_ OGC_CT.idl
```

These type libraries can be used to generate implementations, and they can be used by client-side application code that intends to discover server-side implementations at runtime.

6.1.2 COM Categories

This specification includes several *factory* interfaces, which are used to create all the other COM objects. But how do client applications create these factories?

There are three methods:

a) Use the GUID of the chosen implementation’s co-class
b) Use the fully formed name of the chosen implementation’s co-class.
c) Browse for available implementations of the category at runtime.

For each factory interface, there is a corresponding category GUID. Here are the details of the GUID-s for the categories for all the factory interfaces:

```
DEFINE_GUID(CATID_CS_CoordinateSystemFactory,
0x49937f54, 0x9921, 0x11d3, 0x81, 0x64, 0x0, 0xc0, 0x4f, 0x68, 0xf, 0xff);

DEFINE_GUID(CATID_CS_CoordinateSystemAuthorityFactory,
0x49937f53, 0x9921, 0x11d3, 0x81, 0x64, 0x0, 0xc0, 0x4f, 0x68, 0xf, 0xff);

DEFINE_GUID(CATID_CT_MathTransformFactory,
0x49937f52, 0x9921, 0x11d3, 0x81, 0x64, 0x0, 0xc0, 0x4f, 0x68, 0xf, 0xff);

DEFINE_GUID(CATID_CT_CoordinateTransformationFactory,
0x2ca0a8bd, 0xca5b, 0x11d3, 0x81, 0x98, 0x0, 0xc0, 0x4f, 0x68, 0xf, 0xff);

DEFINE_GUID(CATID_CT_CoordinateTransformationAuthorityFactory,
0x2ca0a8be, 0xca5b, 0x11d3, 0x81, 0x98, 0x0, 0xc0, 0x4f, 0x68, 0xf, 0xff);
```

6.2 CORBA profile

The CORBA profile is specified in the attached CORBA IDL files (see Appendix A: Interface Definition Files: CORBA).

Each package is mapped to a CORBA module of the same name.

Two-dimensional arrays (e.g. matrices) are modeled with a sequence of sequences. All 2D arrays in this specification should be regular, so each sub-sequence in the 2D array should have the same length.

6.3 Java profile

The Java profile is specified in the attached Java source files.

Each Java package is called "`org.opengis.xx`" where "`xx`" is the two-character package prefix in lower case.

The Java source includes tagged comments. These comments were used to generate the accompanying HTML documentation using the JavaDoc utility.

In the Java profile, structures are modeled with Java classes (e.g. PT_CoordinatePoint), and interfaces are used for entities that will have many different implementations (e.g. CS_ProjectedCoordinateSystem).

This specification says that structures used as return values should use "return-by-value" semantics. This means that whenever a Java method has an object return type, the implementation should create a new Java object. Since this is not very efficient, there are also pass-by-reference versions of heavily used functions (e.g. double[] **transformList**(double[] ord)).

Matrices are modeled with the Java type "`double[][]`". Although the Java language allows ragged multi-dimensional arrays, all arrays in this specification should be regular (i.e. rectangular in 2D).

7 Well-Known Text format

Many entities in this specification can be printed in a well-known text format. This allows objects to be stored in databases (persistence), and transmitted between interoperating computer programs.

Each entity has a keyword in upper case (for example, `DATUM` or `UNIT`) followed by the defining, comma-delimited, parameters of the object in brackets. Some objects are composed of objects so the result is a nested structure. Implementations are free to substitute standard brackets ( ) for square brackets [ ] and should be prepared to read both forms of brackets.

The definition for WKT is shown below using Extended Backus Naur Form (EBNF). The WKT for a math transform can be used inside a fitted coordinate system, so it is shown first.

7.1 Math Transform WKT

```
<math transform> = <param mt> | <concat mt> | <inv mt> | <passthrough mt>
<param mt> = PARAM_MT["<classification name>" {,<parameter>}* ]
<parameter> = PARAMETER["<name>", <value>]
<value> = <number>
<concat mt> = CONCAT_MT[<math transform> {,<math transform>}* ]
<inv mt> = INVERSE_MT[<math transform>]
<passthrough mt> = PASSTHROUGH_MT[<integer>, <math transform>]
```

7.2 Coordinate System WKT

```
<coordinate system> = <horz cs> | <geocentric cs> | <vert cs> | <compd cs>
      | <fitted cs> | <local cs>
<horz cs> = <geographic cs> | <projected cs>
<projected cs> = PROJCS["<name>", <geographic cs>, <projection>,
      {<parameter>,}* <linear unit> {,<twin axes>}{,<authority>}]
<projection> = PROJECTION["<name>" {,<authority>}]
<geographic cs> = GEOGCS["<name>", <datum>, <prime meridian>, <angular
      unit> {,<twin axes>} {,<authority>}]
<datum> = DATUM["<name>", <spheroid> {,<to wgs84>} {,<authority>}]
<spheroid> = SPHEROID["<name>", <semi-major axis>, <inverse flattening>
      {,<authority>}]
<semi-major axis> = <number>
<inverse flattening> = <number>
<prime meridian> = PRIMEM["<name>", <longitude> {,<authority>}]
<longitude> = <number>
<angular unit> = <unit>
<linear unit> = <unit>
<unit> = UNIT["<name>", <conversion factor> {,<authority>}]
<conversion factor> = <number>
<geocentric cs> = GEOCCS["<name>", <datum>, <prime meridian>, <linear unit> {,<axis>,
      <axis>, <axis>} {,<authority>}]
<authority> = AUTHORITY["<name>", "<code>"]
<vert cs> = VERT_CS["<name>", <vert datum>, <linear unit>, {<axis>,}
      {,<authority>}]
<vert datum> = VERT_DATUM["<name>", <datum type> {,<authority>}]
<datum type> = <number>
```

```
<compd cs> = COMPD_CS["<name>", <head cs>, <tail cs> {,<authority>}]
<head cs> = <coordinate system>
<tail cs> = <coordinate system>
<twin axes> = <axis>, <axis>
<axis> = AXIS["<name>", NORTH | SOUTH | EAST | WEST | UP | DOWN | OTHER]
<to wgs84s> = TOWGS84[<seven param>]
<seven param> = <dx>,  <dy>, <dz>, <ex>, <ey>, <ez>, <ppm>
<dx> = <number>
<dy> = <number>
<dz> = <number>
<ex> = <number>
<ey> = <number>
<ez> = <number>
<ppm> = <number>
<fitted cs> = FITTED_CS["<name>", <to base>, <base cs>]
<to base> = <math transform>
<base cs> = <coordinate system>
<local cs> = LOCAL_CS["<name>", <local datum>, <unit>, <axis>,
      {,<axis>}* {,<authority>}]
<local datum> = LOCAL_DATUM["<name>", <datum type> {,<authority>}]
```

7.3 Description of WKT keywords

7.3.1 AUTHORITY

This is an optional clause that allows an external authority to manage the definition of an entity. Please see Referencing objects by ID versus Value for more details.

7.3.2 AXIS

The name of the axis is for human consumption. The enumerated value that follows is to allow software to correctly overlay different coordinate systems.

If the optional AXIS terms are not present, then the default values are assumed. They are
Geographic Coordinate Systems: `AXIS["Lon",EAST],AXIS["Lat",NORTH]`
Projected Coordinate System: `AXIS["X",EAST],AXIS["Y",NORTH]`
Geocentric Coordinate System: `AXIS["X",OTHER],AXIS["Y",EAST],AXIS["Z",NORTH]`

However, if these terms are present, and have non-default values, then implementations must be prepared to swap and reverse the coordinates of geometry before attempting to overlay graphics.

7.3.3 COMPD_CS

This indicates a compound coordinate system, which combines the coordinate of two other coordinate systems. For example, a compound 3D coordinate system could be made up of a horizontal coordinate system and a vertical coordinate system.

7.3.4 CONCAT_MT

A transform defined by the concatenation of sub-transforms. The dimension of the output space of the first transform must match the dimension of the input space in the second transform (if defined), and so on for the remaining sub-transforms.

7.3.5 DATUM

This indicates the horizontal datum, which corresponds to the procedure used to measure positions on the surface of the Earth.

7.3.6 FITTED_CS

This indicates a fitted coordinate system.

The math transform is used to construct a map from the fitted coordinate system to the base coordinate system. The transform is often an affine map.

The math transform works from the fitted CS to the base CS so that the fitted CS can have a smaller dimension that the base CS. This is often quite useful. For example, a fitted coordinate system could be a 2D plane approximately tangential to the Earth, but based on a WGS84 geocentric 3D coordinate system.

7.3.7 GEOCCS

A 3D coordinate system, with its origin at the center of the Earth. The X axis points towards the prime meridian. The Y axis points East or West. The Z axis points North or South. By default the Z axis will point North, and the Y axis will point East (e.g. a right handed system), but you should check the axes for non-default values

7.3.8 GEOGCS

A coordinate system based on latitude and longitude. Some geographic coordinate systems are Lat/Lon, and some are Lon/Lat. You can find out which this is by examining the axes. You should also check the angular units, since not all geographic coordinate systems use degrees.

7.3.9 INVERSE_MT

A math transform defined as the inverse of another transform.

7.3.10 LOCAL_CS

This indicates a local, ungeoreferenced coordinate system. Such coordinate systems are often used in CAD systems. They can also be used for local surveys, where the relationship between the surveyed site and the rest of the world is not important.

The number of AXIS clauses indicates the dimension of the local coordinate system.

7.3.11 PARAMETER

A named projection parameter value.

The units of the parameter must be inferred from its context. If the parameter is inside a PROJCS, then its units will match the units of the PROJCS. If the parameter is inside a PARAM_MT, then its units will be meters and degrees for linear and angular values respectively.

7.3.12 PARAM_MT

A parameterized math transform.

All the linear parameters are expressed in meters, and all the angular parameters are expressed in degrees. Other parameters should use S.I. units where possible. (E.g. use Kg for mass, and seconds for time.)

The `<classification name>` is a coded value that specifies the formulae used by the math transform. See Parameterized Transforms for legal values, and the corresponding parameters.

7.3.13 PASSTHROUGH_MT

This is a math transform that passes through a subset of ordinates to another transform. This allows transforms to operate on a subset of ordinates. For example, if you have (Lat,Lon,Height) coordinates, then you may wish to convert the height values from meters to feet without affecting the (Lat,Lon) values. If you wanted to affect the (Lat,Lon) values and leave the Height values alone, then you would have to swap the ordinates around to (Height,Lat,Lon). You can do this with an affine map.

The `<integer>` argument is the index of the first affected ordinate. The `<math transform>` argument is the transform to pass coordinates onto.

7.3.14 PRIMEM

This defines the meridian used to take longitude measurements from. The units of the `<longitude>` must be inferred from the context. If the PRIMEM clause occurs inside a GEOGCS, then the longitude units will match those of the geographic coordinate system. If the PRIMEM clause occurs inside a GEOCCS, then the units will be in degrees.

The longitude value defines the angle of the prime meridian relative to the Greenwich Meridian. A positive value indicates the prime meridian is East of Greenwich, and a negative value indicates the prime meridian is West of Greenwich.

7.3.15 PROJCS

This indicates a projected coordinate system. The PROJECTION sub-clause contains the classification name used by the CT_MathTransformFactory::CreateParameterizedTransform method, and the PARAMETER clauses specify the parameters. However, the units used by CT_MathTransformFactory are always meters and degrees, and the units in the PARAMETER clauses are in the linear/angular units of the PROJCS/GEOGCS respectively. So if you are writing code to read or write WKT, then you must use the IsParameterAngular and IsParameterLinear methods to do the unit conversions - be careful!

(Notice that this handling of units is slightly different from the way the EPSG 4 database works. In the EPSG 4 database, each transformation parameter value defines its own units. However, 99% of the EPSG projection parameter units are the same as the units of the corresponding projected coordinate system.)

7.3.16 PROJECTION

This describes a projection from geographic coordinates to projected coordinates. It is used inside a PROJCS to define the parameters of the projection transform.

7.3.17 SPHEROID

This describes a spheroid, which is an approximation of the Earth's surface as a squashed sphere. In this document, the terms “spheroid” and “ellipsoid” are synonymous. The term

"SPHEROID" is used in WKT for compatibility with Simple Features. However, the term "ellipsoid" is preferred elsewhere in this specification.

7.3.18 TOWGS84

This indicates a list of up to 7 Bursa Wolf transformation parameters. These parameters can be used to approximate a transformation from the horizontal datum to the WGS84 datum. However, it must be remembered that this transformation is only an approximation. For a given horizontal datum, different Bursa Wolf transformations can be used to minimize the errors over different regions.

If the DATUM clause contains a TOWGS84 clause, then this should be its "preferred" transformation, which will often be the transformation which gives a broad approximation over the whole area of interest (e.g. the area of interest in the containing geographic coordinate system).

Sometimes, only the first three or six parameters are defined. In this case the remaining parameters must be zero. If only three parameters are defined, then they can still be plugged into the Bursa Wolf formulas, or you can take a short cut. The Bursa Wolf transformation works on geocentric coordinates, so you cannot apply it onto geographic coordinates directly. If there are only three parameters then you can use the Molodenski or abridged Molodenski formulas.

The DATUM clause may not contain a TOWGS84 clause in the following situations:

a) The writing application was using the Simple Features specification, which does not specify TOWGS84 as a valid keyword
b) The writing application did not have an available transformation.
c) There is no possible transformation. For example, the horizontal datum could be a surface that rotates relative to the Earth's surface.

In particular, if the DATUM does contain a TOWGS84 clause, and the parameter values are zero, then the receiving application can assume that the writing application believed that the datum is approximately equal to WGS84.

7.3.19 UNIT

This describes units used for values elsewhere within the parent WKT clause (sometimes including descendants of the parent clause).

The physical dimension (i.e. type) of the units is determined by context. For example, in a GEOGCS the type of the units is angular. In a VERT_CS the type of the units is linear.

Within a UNIT clause, the units are described by relating them to a fundamental unit of that type with a conversion factor. For linear units, the conversion factor is the scalar value that converts the described units into meters. For angular units, the conversion factor is the scalar value that converts the described units into radians.

7.3.20 VERT_DATUM

This indicates the vertical datum, or method used for vertical measurements.

7.3.21 VERT_CS

This indicates a vertical coordinate system.

7.4 WKT Example

The following example shows a 3D compound coordinate system, which is made by combining a projected coordinate system and a vertical coordinate system. This is the same coordinate system as used for the XML Example.

```
COMPD_CS[
    "OSGB36 / British National Grid + ODN",
    PROJCS[
        "OSGB 1936 / British National Grid",
        GEOGCS[
            "OSGB 1936",
            DATUM[
                "OSGB_1936",
                SPHEROID[
                    "Airy 1830",6377563.396,299.3249646,
                    AUTHORITY["EPSG","7001"]
                ],
                TOWGS84[375,-111,431,0,0,0,0],
                AUTHORITY["EPSG","6277"]
            ],
            PRIMEM[
                "Greenwich",0,
                AUTHORITY["EPSG","8901"]
            ],
            UNIT[
                "DMSH",
                0.0174532925199433,
                AUTHORITY["EPSG","9108"]
            ],
            AXIS["Lat",NORTH],
            AXIS["Long",EAST],
            AUTHORITY["EPSG","4277"]
        ],
        PROJECTION["Transverse_Mercator"],
        PARAMETER["latitude_of_origin",49],
        PARAMETER["central_meridian",-2],
        PARAMETER["scale_factor",0.999601272],
        PARAMETER["false_easting",400000],
        PARAMETER["false_northing",-100000],
        UNIT[
            "metre",1,
            AUTHORITY["EPSG","9001"]
        ],
        AXIS["E",EAST],
        AXIS["N",NORTH],
        AUTHORITY["EPSG","27700"]
    ],
    VERT_CS[
        "Newlyn",
        VERT_DATUM[
            "Ordnance Datum Newlyn",2005,
            AUTHORITY["EPSG","5101"]
        ],
        UNIT[
            "metre",1,
            AUTHORITY["EPSG","9001"]
        ],
        AXIS["Up",UP],
        AUTHORITY["EPSG","5701"]
    ],
    AUTHORITY["EPSG","7405"]
]
```

8 Relationship to Simple Features

"Simple Features" is an earlier interface specification adopted by OGC. The Simple Features specification includes two packages: Geometry and Spatial Reference Systems (SRS).

The CS package in this specification covers the same ground as the Simple Features SRS package. The SRS package was based on a subset of the coordinate systems in the EPSG abstract model. The CS package extends this abstract model to encompass all EPSG coordinate systems, including vertical coordinate systems and compound coordinate systems. CS then goes further to handle more than 3 dimensions and ungeoreferenced coordinate systems.

8.1 Compatibility of Well-Known Text

The CS package keeps the WKT serializing concept of Simple Features SRS. In fact, **all Simple Features SRS Well-Known Text strings are valid in CS, and have exactly the same meaning**. This ensures that all stored Simple Features SRS data will be compatible with the new CS package.

The Geographic Transform interface in the Simple Features SRS package has been absorbed into the CT package as a range of different math transforms (e.g. Molodenski). This does not affect WKT compatibility, since the WKT for geographic transformations was never defined.

8.2 Name changes from SF-SRS to CS

The CS package basically just renames all the Simple Features SRS interfaces by adding the "CS_" prefix, and then removes all the mutating methods. The functionality of the mutating methods (e.g. changing the longitude of a Prime Meridian) has been moved into the enlarged factory interfaces. A couple of the old Simple Features interfaces have been moved into other packages (see below).

Since most of the method names have been left unchanged from Simple Features, it is possible for objects to support both sets of interfaces. For example, a geographic coordinate system object (e.g. MyGCS) can easily support both CS_GeographicCoordinateSystem (from CS) and IGeographicCoordinateSystem (from SF). This means that the object MyGCS can be used with code that expects Simple Features interfaces or code that expects CS interfaces.

One notable name change is the method to get an entity's well-known text. For example, in COM Simple Features this property was "WellKnownText", whereas in CS the property is called "WKT". This is done deliberately, so a new object which supports both SF and CS interfaces can tell whether it is being asked for an SF compatible WKT string, or a CS compatible WKT string. If it is asked for an SF compatible string then it should not include the new optional clauses. If the object is not part of the SF abstract model (e.g. a 3D coordinate system) then it should either not support the SF interfaces, or it should fail the WellKnownText method.

8.3 Authority Codes

The CS and CT packages use a String for the Authority Code, whereas Simple Features used an Integer. This change was made for compatibility with those authorities that use String codes.

As a result, the Simple Features ISpatialReferenceInfo "Code" property will fail if the CS_Info "AuthorityCode" property cannot be represented as an integer.

9 Processes

Common processes are shown here as scenarios. Some scenarios refer to previous scenarios, so you may find it convenient to read them in order.

9.1 Create a NAD27 California Zone 1 CS

In this example, a client wishes to get a coordinate system for California zone 1, based on the NAD27 geoid. The units of the coordinate system will be in feet.

The client does not know (or care) what projection this CS uses, let alone the correct parameter values to use for that projection. However, they do know the EPSG code, which is 26741.

1) Create a CS_CoordinateSystemAuthorityFactory

EPSG is a coordinate system “authority”, so firstly the client needs to create a CS_CoordinateSystemAuthorityFactory.

In the COM DCP they could write a loop to iterate through all the available CS_CoordinateSystemAuthorityFactory implementations, testing each one to see if its authority name was “EPSG”.

In other DCPs, or if they knew the class name of the EPSG authority factory object within their implementation, they could dynamically create the object by name.

2) Use authority factory to create CS

In pseudo-code:

```
Dim csaf as  CS_CoordinateSystemAuthorityFactory
csaf = GetMyEpsgFactory()
dim cs as CS_CoordinateSystem
cs = csaf.CreateHorizontalCoordinateSystem(26741)
```

9.2 Create a NAD83 California Zone 1 CS

This process is identical to the process above, except that the EPSG code for a NAD83 projection is 26941. Also, the EPSG NAD83 coordinate system uses meters instead of feet.

9.3 Store and recall CS in a Database

The client may have lots of point data that uses coordinates in NAD27 California Zone 1. They can easily store the points in a two-column table, but how can they store the coordinate system and retrieve it later?

One solution would be to store the EPSG code of the CS (26741). But if they do this, then other software which does not know about EPSG, or which will not have an EPSG factory, will not be able to georeference the points correctly.

An alternative would be to store the CS in Well-Known Text format. (This is what the SQL92 schema of the Simple Feature specification does.)

To do this, the client would get the WKT property from the CS object. The WKT property is a text string, so it is easy to store in a database.

Later, other software could read the WKT string and create their own CS object by using a CS_CoordinateSystemFactory as follows:

```
Dim csf as CS_CoordinateSystemFactory
csf = GetMyCoordsysFactory()
Dim wkt as String
wkt = ReadWellKnownTextFromDB()
dim cs as CS_CoordinateSystem
cs = csf.CreateFromWKT(wkt)
```

Please notice that the above example uses a normal factory, and not an “authority” factory.

9.4 Merge California Zone 1 data in NAD27 and NAD83

If a client has an old table of California zone 1 points in NAD27 feet, and a new table of California zone 1 points in NAD83 meters, then they may want to merge the old table into the new table. To do this they will need to perform a coordinate transformation.

Firstly, the client can get WKT strings for the two coordinate systems. (If the WKT strings for the coordinate systems are stored with the point data, then they can skip this step.)

Secondly, the client can get a CT_CoordinateTransformation between the two coordinate systems. This will get details of the required transformation, but will not actually do any transforming operations.

Thirdly, the client can retrieve the CT_MathTransform from the CT_CoordinateTransformation.

Finally, the client can use the CT_MathTransform to transform the old point data into the NAD83 coordinate system, and then add the transformed points to the new table.

In pseudo code:

```
Rem Firstly get WKT of tables
Dim wktOld as String
Dim wktNew as String
wktOld = GetOldTableCS()
wktNew = GetNewTableCS()

Rem Secondly get a transformation
Dim ctf as new CT_CoordinateTransformationFactory
Dim ct as CT_CoordinateTransformation
ct = ctf.CreateFromCoordinateSystems( wktOld, wktNew )

Rem Thirdly retrieve the math transform
Dim mt as CT_MathTransform
mt = ct.MathTransform

Rem Finally transform and merge the old points
While (OldPointAvailable())
    Dim ptOld as PT_CoordinatePoint
    ptOld.Ord.resize(2)
    ptOld.Ord[0] = ReadPointX()
    ptOld.Ord[1] = ReadPointY()
    Dim ptNew as PT_CoordinatePoint
    ptNew = mt.Transform( ptOld )
```

```
          StoreNewPoint (ptNew.Ord[0], ptNew.Ord[1] )
      End while
```

9.5 Transform steps from NAD27 to NAD83 in California Zone 1

The math transform used in the previous scenario contains many steps.

It is the job of the CT_CoordinateTransformationFactory to analyze the source and target coordinate systems to find a series of transform steps that will make up the complete transformation.

Here is a possible math transform for the previous scenario in Well-Known Text format: (Embedded comments are shown in bold. These comments are not valid in WKT format, and are only included here to explain what the steps are doing.)

```
// Use concatenated transform
CONCAT_MT[
    // Use affine map to convert feet into meters.
    PARAM_MT["Affine",
        PARAMETER["num_row",3],
        PARAMETER["num_col",3],
        PARAMETER["elt_0_0",0.3048006096012192],
        PARAMETER["elt_0_1",0],
        PARAMETER["elt_0_2",0],
        PARAMETER["elt_1_0",0],
        PARAMETER["elt_1_1",0.3048006096012192],
        PARAMETER["elt_1_2",0],
        PARAMETER["elt_2_0",0],
        PARAMETER["elt_2_1",0],
        PARAMETER["elt_2_2",1]
    ],
    // Undo the California state plane projection.
    INVERSE_MT[
        // Projection converts metres to (lon,lat).
        PARAM_MT["Lambert_Conformal_Conic_2SP",
            PARAMETER["latitude_of_origin",41.66666666666662],
            PARAMETER["central_meridian",39.99999999999996],
            PARAMETER["standard_parallel_1",39.3333333333333],
            PARAMETER["standard_parallel_2",-121.9999999999999],
            PARAMETER["false_easting",609601.2192024385],
            PARAMETER["false_northing",0],
            PARAMETER["semi_major",6378206.4],
            PARAMETER["semi_minor",6356583.8]
        ]
    ],
    // Use affine map to change (lon,lat) to (lon,lat,0).
    PARAM_MT["Affine",
        PARAMETER["num_row",4],
        PARAMETER["num_col",3],
        PARAMETER["elt_0_0",0],
        PARAMETER["elt_0_1",1],
        PARAMETER["elt_0_2",0],
        PARAMETER["elt_1_0",1],
        PARAMETER["elt_1_1",0],
        PARAMETER["elt_1_2",0],
        PARAMETER["elt_2_0",0],
```

```
        PARAMETER["elt_2_1",0],
        PARAMETER["elt_2_2",0],
        PARAMETER["elt_3_0",0],
        PARAMETER["elt_3_1",0],
        PARAMETER["elt_3_2",1]
    ],
    // Convert (lon,lat,0) to geocentric (X,Y,Z).
    PARAM_MT["Ellipsoid_To_Geocentric",
        PARAMETER["semi_major",6378206.4],
        PARAMETER["semi_minor",6356583.8]
    ],
    // Use affine map for Bursa Wolf transformation.
    PARAM_MT["Affine",
        PARAMETER["num_row",4],
        PARAMETER["num_col",4],
        PARAMETER["elt_0_0",1],
        PARAMETER["elt_0_1",0],
        PARAMETER["elt_0_2",0],
        PARAMETER["elt_0_3",3],
        PARAMETER["elt_1_0",0],
        PARAMETER["elt_1_1",1],
        PARAMETER["elt_1_2",0],
        PARAMETER["elt_1_3",-142],
        PARAMETER["elt_2_0",0],
        PARAMETER["elt_2_1",0],
        PARAMETER["elt_2_2",1],
        PARAMETER["elt_2_3",-183],
        PARAMETER["elt_3_0",0],
        PARAMETER["elt_3_1",0],
        PARAMETER["elt_3_2",0],
        PARAMETER["elt_3_3",1]
    ],
    // Convert (X,Y,Z) to (lon,lat,hgt).
    PARAM_MT["Geocentric_To_Ellipsoid",
        PARAMETER["semi_major",6378137],
        PARAMETER["semi_minor",6356752.314140356]
    ],
    // Discard height, converting (lon,lat,hgt) into (lon,lat).
    PARAM_MT["Affine",
        PARAMETER["num_row",3],
        PARAMETER["num_col",4],
        PARAMETER["elt_0_0",0],
        PARAMETER["elt_0_1",1],
        PARAMETER["elt_0_2",0],
        PARAMETER["elt_0_3",0],
        PARAMETER["elt_1_0",1],
        PARAMETER["elt_1_1",0],
        PARAMETER["elt_1_2",0],
        PARAMETER["elt_1_3",0],
        PARAMETER["elt_2_0",0],
        PARAMETER["elt_2_1",0],
        PARAMETER["elt_2_2",0],
        PARAMETER["elt_2_3",1]
    ],
    // Reapply state plane projection.
    PARAM_MT["Lambert_Conformal_Conic_2SP",
        PARAMETER["latitude_of_origin",41.66666666666662],
```

```
        PARAMETER["central_meridian",39.99999999999996],
        PARAMETER["standard_parallel_1",39.3333333333333],
        PARAMETER["standard_parallel_2",-121.9999999999999],
        PARAMETER["false_easting",2000000],
        PARAMETER["false_northing",500000],
        PARAMETER["semi_major",6378137],
        PARAMETER["semi_minor",6356752.314140356]
    ]
]
```

10 Parameterized Transforms

Many of the math transforms in the CT package are completely characterized by their classification, or transform, name, e.g. “Transverse_Mercator”, and their parameter values. This allows these transforms to be implicitly serialized in foreign WKT strings (e.g. the PROJECTION clause in the CS WKT definition).

We expect that the list of classification names will grow to incorporate new transforms, as this Interface Specification is used in future specifications, or by different GIS/CAD market sectors. We encourage OGC to create and maintain a registry of transform classifications, and their parameters, and ideally their formulas.

Here is an initial list of math transform classifications:

Classification Name	EPSG Code	Inception	Mandatory
Longitude_Rotation	9601	CT	Yes
Molodenski	9604	CT	
Abridged_Molodenski	9605	CT	Yes
Geocentric_To_Ellipsoid	9602	CT	Yes
Ellipsoid_To_Geocentric	9602	CT	Yes
Affine		CT	Yes
Lambert_Conformal_Conic_1SP	9801	SF	Yes
Lambert_Conformal_Conic_2SP	9802	SF	Yes
Lambert_Conformal_Conic_2SP_Belgium	9803	SF	
Mercator_1SP	9804	SF	
Mercator_2SP	9805	SF	
Cassini_Soldner	9806	SF	
Transverse_Mercator	9807	SF	Yes
Transverse_Mercator_South_Orientated	9808	SF	
Oblique_Stereographic	9809	SF	
Polar_Stereographic	9810	SF	
New_Zealand_Map_Grid	9811	SF	
Hotine_Oblique_Mercator	9812	SF	
Laborde_Oblique_Mercator	9813	SF	
Swiss_Oblique_Cylindrical	9814	SF	
Oblique_Mercator	9815	SF	
Tunisia_Mining_Grid	9816	SF	

Figure 2: Classifications of parameterized math transforms

The inception column shows when these classification names were first specified by OGC. So an inception of SF indicates that OpenGIS Simple Features first used the classification, and an inception of CT indicates that the classification was first used by this specification document.

10.1 Ellipsoid_To_Geocentric

Parameter	Description
Semi_major	Equatorial radius in meters
semi_minor	Polar radius in meters

This transform operates on 3D coordinates. The first input ordinate is the longitude in degrees, and the second input ordinate is the latitude in degrees. The third input ordinate is the height above the ellipsoid in meters.

(These formulas and equations are copied from the EPSG 4.3 database.)

Latitude, P, and Longitude, L, in terms of Geographic Coordinate System (GCS) A may be expressed in terms of a geocentric (earth centered) Cartesian coordinate system X, Y, Z with the Z axis corresponding with the Polar axis positive northwards, the X axis through the intersection of the Greenwich meridian and equator, and the Y axis through the intersection of the equator with longitude 90 degrees E. If the GCS's prime meridian is not Greenwich, longitudes must first be converted to their Greenwich equivalent. If the earth's spheroidal semi major axis is a, semi minor axis b, and inverse flattening 1/f, then

XA= (nu + hA) cos P cos L
YA= (nu + hA) cos P sin L
ZA= ((1 - e^2) nu + hA) sin P

where nu is the prime vertical radius of curvature at latitude P and is equal to
nu = a /(1 - e^2*sin^2(P))^0.5,
P and L are respectively the latitude and longitude (related to Greenwich) of the point
h is height above the ellipsoid, (topographic height plus geoidal height), and
e is the eccentricity of the ellipsoid where e^2 = (a^2 -b^2)/a^2 = 2f -f^2

10.2 Geocentric_To_Ellipsoid

Parameter	Description
semi_major	Equatorial radius in meters
semi_minor	Polar radius in meters

(These formulas and equations are copied from the EPSG 4.3 database.)

Cartesian coordinates in geocentric coordinate system B may be used to derive geographical coordinates in terms of geographic coordinate system B by:
P = arctan (ZB + e^2* nu*sin P) / (XB^2 + YB^2)^0.5 by iteration
L = arctan YB/XB
hB = XB sec L sec P - nu

where LB is relative to Greenwich. If the geographic system has a non Greenwich prime meridian, the Greenwich value of the local prime meridian should be applied to longitude.

(Note that h is the height above the ellipsoid. This is the height value which is delivered by Transit and GPS satellite observations but is not the topographic height value which is normally used for national mapping and leveling operations. The topographic height is usually the height above mean sea level or an alternative level reference for the country. If one starts with a topographic height, it will be necessary to convert it to an ellipsoid height before using the above transformation formulas. h = N + H, where N is the geoid height above the ellipsoid at the point and is sometimes negative, and H is the height of the point above the geoid. The height above the geoid is often taken to be that above mean sea level, perhaps with a constant correction applied. Geoid heights of points above the nationally used ellipsoid may not be readily available. For the WGS84 ellipsoid the value of N, representing the height of the geoid relative to the ellipsoid, can vary between

values of -100m in the Sri Lanka area to +60m in the North Atlantic.)

10.3 Abridged_Molodenski

Parameter	Description
dim	Dimension of points (i.e. 2 or 3).
dx	X shift in meters
dy	Y shift in meters
dz	Z shift in meters
src_semi_major	Source equatorial radius in meters
src_semi_minor	Source polar radius in meters
tgt_semi_major	Target equatorial radius in meters
tgt_semi_minor	Target polar radius in meters

This transform can operate on 2D or 3D coordinates. In either case the first ordinate is the longitude in degrees, and the second ordinate is the latitude in degrees. In the 3D form the third ordinate is the height above the ellipsoid in meters.

As an alternative to the computation of the new latitude, longitude and height above ellipsoid in discrete steps through geocentric coordinates, the changes in these coordinates may be derived directly by formulas derived by Molodenski. Abridged versions of these formulas, which are quite satisfactory for three parameter transformations, are as follows:

(These formulas and equations are copied from the EPSG 4.3 database.)

Dlat " = [(-dX*sin(lat)*cos(lon)) - (dY*sin(lat)*sin(lon)) + (dZ*cos(lat)) + (((a*Df) + (f*Da))*sin(2*lat))] / (rho * sin(1"))

Dlon " = (-dX*sin(lon) + dY*cos(lon)) / ((nu*cos(lat)) * sin(1"))

Dh = (dX*cos(lat)*cos(lon)) + (dY*cos(lat)*sin(lon)) + (dZ*sin(lat)) + ((a*Df + f*Da)*(sin(lat)^2)) - Da

where the dX, dY and dZ terms are as before, and rho and nu are the meridian and prime vertical radii of curvature at the given latitude (lat) on the first ellipsoid (see section on Geographic/Geocentric formulas), Da is the difference in the semi-major axes (a1 - a2) of the first and second ellipsoids and Df is the difference in the flattening of the two ellipsoids.

The formulas for Dlat and Dlon indicate changes in latitude and longitude in arc-seconds.

10.4 Affine

Affine maps are commonly used in coordinate transformations. An affine map can perform translation, rotations, scaling and shearing. In fact, any transformation that transforms all straight lines into straight lines is an affine map.

However, a classical matrix/vector product can only perform rotations – it cannot do translations. In order to do translations, the follow convention must be followed:

For a 2D affine map, the matrix is actually a 3 by 3 matrix, laid out as follows:

| Axx Ayx Sx |

Axy	Ayy	Sy
0	0	1

The 2D position (X,Y) will be transformed to (Axx*X + Ayx*Y + Sx, Axy*X + Ayy*Y + Sy).

You can think of this working as a normal matrix multiplication on the column vector (X, Y, 1). Notice that an extra vector element of 1 is added, to pick up the affine map's transformation of (Sx,Sy). This means that the dimensions of the matrix are always one bigger than the dimensions of the source/target coordinate systems.

This method of using matrices to define affine maps is extended to all dimensions.

Affine maps can also be used to define a Bursa Wolf transformation. The Bursa Wolf transformation is applied to geocentric coordinates to model a change of datum. The formula is as follows:

```
S=(1+ppm*1000000)

(X')        (  1    -ez     +ey)    (X)    (dx)
(Y') = S *  ( +ez    1      -ex) *  (Y) +  (dy)
(Z')        ( -ey   +ex      1 )    (Z)    (dz)
```

Parameter	Description
Num_row	Number of rows in matrix
Num_col	Number of columns in matrix
elt_<r>_<c>	Element of matrix

For the element parameters, <r> and <c> should be substituted by printed decimal numbers. The values of r should be from 0 to (num_row-1), and the values of s should be from 0 to (num_col-1).

Any undefined matrix elements are assumed to be zero for `(r!=c)`, and one for `(s==t)`. This corresponds to the identity transformation when the number of rows and columns are the same. The number of columns corresponds to one more than the dimension of the source coordinates and the number of rows corresponds to one more than the dimension of target coordinates. This means that the affine map can be applied to a point by multiplying its matrix on the left-hand side of the point's vector. (There is another math convention where the matrix is used to the right of the vector, but here we are using the left-hand side convention.) The extra dimension in the matrix is used to let the affine map do a translation.

10.5 Longitude_Rotation

Parameter	Description
dim	Dimension of coordinates
rotation	Rotation angle in degrees

This transform adds a constant onto longitude in (Lon,Lat) or (Lon,Lat,Hgt) coordinates. This is useful for handling changes in Prime Meridian.

- In the real world, a longitude rotation is reversible – you just rotate in the opposite direction. So we want to make sure that the Longitude_Rotation transform has a well-defined inverse transform. To do this, we must be very precise about the domain of this transform, and make sure that all points on the Earth have a unique (Lon,Lat) coordinate. So all longitudes should

be in degrees, in the range [-180,180). The latitude values should be in the range [-90,90]. If the latitude value is not in the range (-90,90), then the longitude should be 0. (For example, you should always use a longitude of zero for the North and South poles.) These rules apply to input and output longitude and latitude values.

- The longitude is assumed to be the first ordinate, and the latitude is assumed to be the second ordinate. Any subsequent ordinates are ignored, and should be left unchanged.
- This transform's derivative should be an identity matrix, except at points where the transform is discontinuous (i.e. the poles, and along two meridians).

10.6 Cartographic Projection Transforms

Cartographic projection transforms are used by projected coordinate systems to map geographic coordinates (e.g. Longitude and Latitude) into (X,Y) coordinates. These (X,Y) coordinates can be imagined to lie on a plane, such as a paper map or a screen.

All cartographic projection transforms will have the following properties:

- Converts from (Longitude, Latitude) coordinates to (X,Y)
- All angles are assumed to be degrees, and all distances are assumed to be meters.
- The domain should be a subset of {[-180,180)x(-90,90), (0,-90), (0,90)}.

Although all cartographic projection *transforms* must have the properties listed above, many projected *coordinate systems* have different properties. For example, in Europe some projected coordinate systems use grads instead of degrees, and often the base geographic coordinate system is (Latitude, Longitude) instead of (Longitude, Latitude). This means that the cartographic projected transform is often used as a single step in a series of transforms, where the other steps change units and swap ordinates.

10.6.1 Cartographic Projection Transform Parameters

The valid parameters for the mandatory cartographic projection transforms are listed below. These include parameters for the semi-major and semi-minor axis of their ellipsoid in meters. These ellipsoid parameters can be skipped when the projection WKT is being embedded inside a Projected Coordinate System (see Well-Known Text format details), since the PCS ellipsoid will override the projection transform ellipsoid.

10.6.1.1 Transverse_Mercator Projection

Parameter	Description of units
semi_major	Meters
semi_minor	Meters
latitude_of_origin	Degrees
central_meridian	Degrees
scale_factor	No units
false_easting	Meters
false_northing	Meters

10.6.1.2 Lambert_Conformal_Conic_1SP

Parameter	Description of units
semi_major	Meters
semi_minor	Meters
latitude_of_origin	Degrees
central_meridian	Degrees
scale_factor	No units
false_easting	Meters
false_northing	Meters

10.6.1.3 Lambert_Conformal_Conic_2SP

Parameter	Description of units
semi_major	Meters
semi_minor	Meters
latitude_of_origin	Degrees
central_meridian	Degrees
standard_parallel1	Degrees
standard_parallel2	Degrees
false_easting	Meters
false_northing	Meters

11 Referencing objects by ID versus Value

External authorities may be used to manage definitions of objects used in this Interface Specification. The definitions of these objects are referenced using code strings. The codes and authority names can be embedded in WKT strings or XML, using an AUTHORITY clause.

If there is an AUTHORITY clause, and the reader recognizes the authority name, then the reader is entitled to use the authority's definition, and ignore all the other values at that level in the definition. (How the reader consults the authority is outside the scope of this Interface Specification.)

To improve interoperability, writers of WKT and XML must always include a copy of all the values, as well as any AUTHORITY clause.

Readers will use the values (as opposed to the authority code) in the following cases:

1) There is no AUTHORITY clause
2) The reader does not recognize the authority
3) The reader fails to create an object using the authority code

The last case (where the reader recognizes the authority name, but fails to create an object using the code) can happen if the reader is using a different version of the authority from the writer.

In order to avoid misinterpretations, the following guidelines for authorities are recommended:

a) The authority name (or namespace) must be registered by OGC
b) The authority must not reuse old code strings for "different" entities.

11.1 Registered WKT Authorities

We expect this part of the specification to be expanded as more authority names and name-spaces are registered with OGC.

11.1.1 EPSG/POSC

The European Petroleum Survey Group maintains a database of spatial reference objects (see http://www.petroconsultants.com/products/geodetic.html). The IDs of these objects are currently used in the GeoTIFF specification, and the OGC Web Mapping Testbed interfaces.

WKT key word	Authority name	EPSG table name	Code min	Code max
SPHEROID	"EPSG"	ELLIPSOID	7000	7299
COMPD_CS	"EPSG"	COMPD_CS	7400	7999
DATUM	"EPSG"	GEOD_DATUM	6000	6999
PRIMEM	"EPSG"	P_MERIDIAN	8900	8999
GEOGCS	"EPSG"	HORIZ_CS	4000	4999
PROJCS	"EPSG"	HORIZ_CS	200	32766
CT	"EPSG"	GEOD_TRF	8000	8399
UNIT	"EPSG"	UOM_ANGLE, UOM_LENGTH	9000	9199
VERT_CS	"EPSG"	VERT_CS	5600	5999
VERT_DATUM	"EPSG"	VERT_DATUM	5000	5399

Figure 3: EPSG code ranges

The table above shows that the authority name “EPSG” is registered for the WKT clauses SPHEROID, COMPD_CS, DATUM, PRIMEM, GEOGCS, PROJCS, CT, UNIT, VERT_CS and VERT_DATUM. The table names and code ranges are shown for interest only – since they are an external authority, EPSG/POSC may change these ranges without notice at any time.

12 UML Documentation

The specified interfaces have been modeled in the Unified Modeling Language in a DCP independent manner. If you have software for reading UML files then you may wish to use it to examine the attached UML model file (with file extension MDL).

Please note that the UML model is automatically generated from the Java interface source files. The conversion program follows some simple rules that define, for example, how to model the dependencies between classes. Although this approach is very good for keeping the UML model and all of the profiles consistent (the COM and CORBA Interface Definition Files are also automatically generated from the Java source files), the resulting UML model is less sophisticated than one created by hand.

12.1 Packages

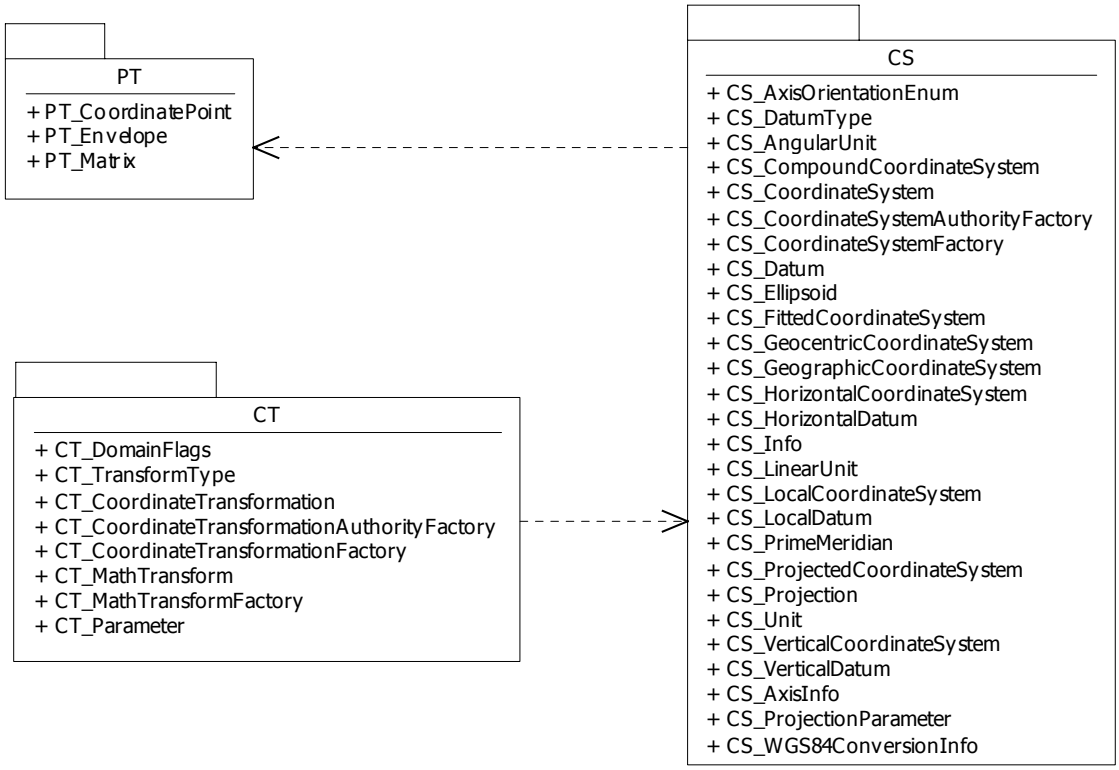

Figure 4. Packages

12.2 PT Logical Package: Positioning

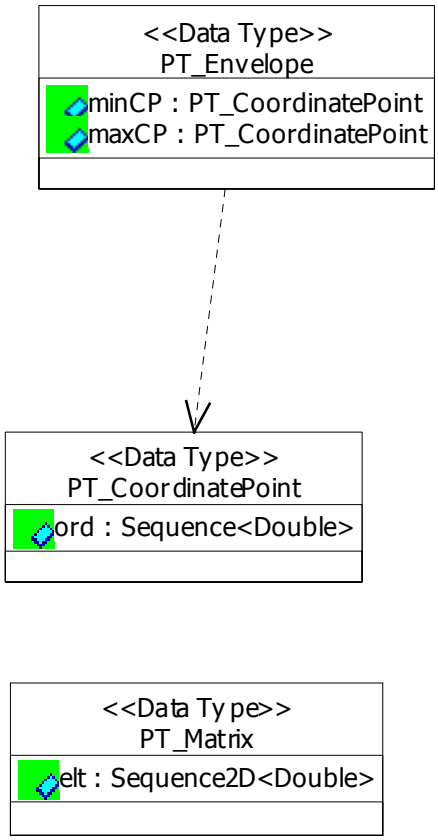

Figure 5. Positioning (Class Diagram)

12.2.1 PT_CoordinatePoint Class

A position defined by a list of numbers. The ordinate values are indexed from 0 to (NumDim-1), where NumDim is the dimension of the coordinate system the coordinate point belongs in.

12.2.1.1 ord (Attribute) :

The ordinates of the coordinate point.

12.2.2 PT_Envelope Class

A box defined by two positions. The two positions must have the same dimension. Each of the ordinate values in the minimum point must be less than or equal to the corresponding ordinate value in the maximum point. Please note that these two points may be outside the valid domain of their coordinate system. (Of course the points and envelope do not explicitly reference a coordinate system, but their implicit coordinate system is defined by their context.)

12.2.2.1 maxCP (Attribute) :

Point containing maximum ordinate values.

12.2.2.2 minCP (Attribute) :

Point containing minimum ordinate values.

12.2.3 PT_Matrix Class

A two dimensional array of numbers.

12.2.3.1 elt (Attribute) :

Elements of the matrix. The elements should be stored in a rectangular two dimensional array. So in Java, all double[] elements of the outer array must have the same size. In COM, this is represented as a 2D SAFEARRAY.

12.3 CS Logical Package: Co-ordinate Systems

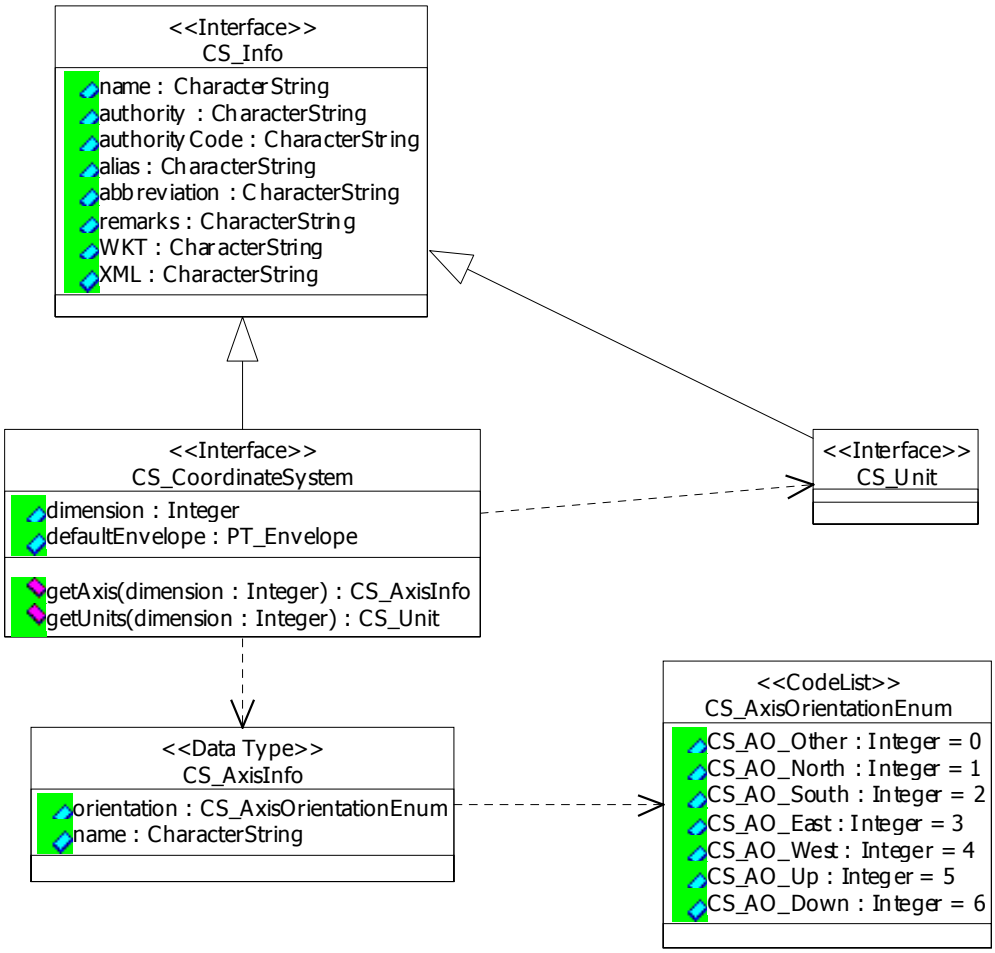

Figure 6. Coordinate System (Class Diagram)

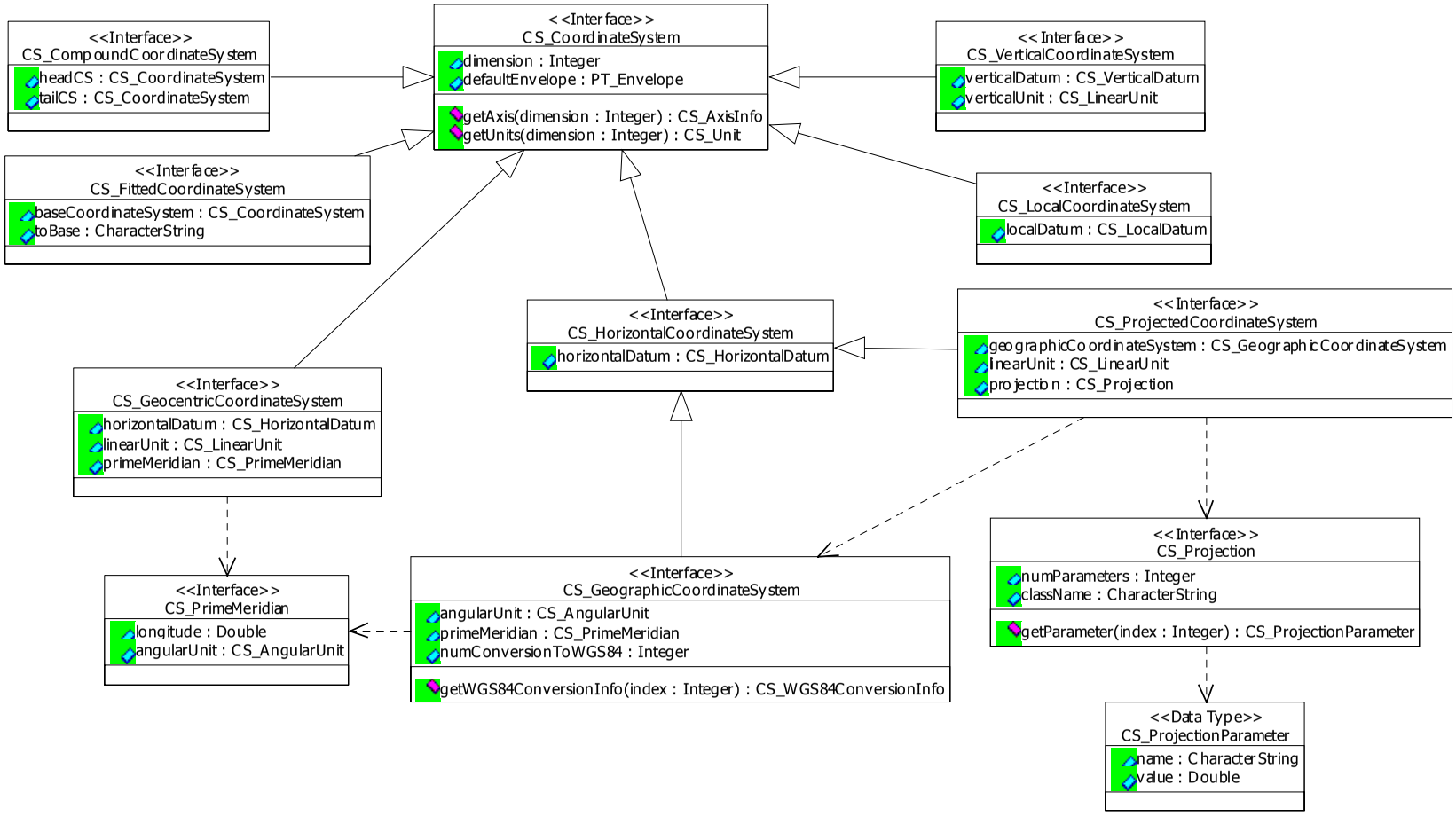

Figure 7. Coordinate System Classes (Class Diagram)

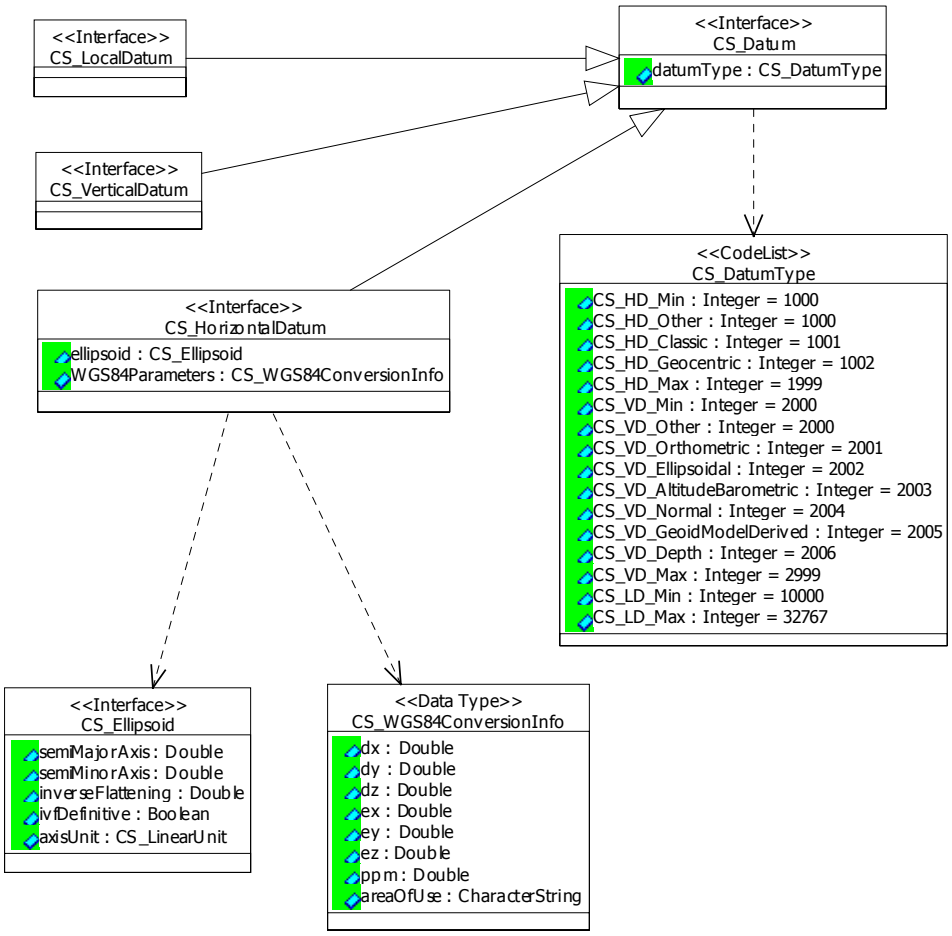

Figure 8. Datums (Class Diagram)

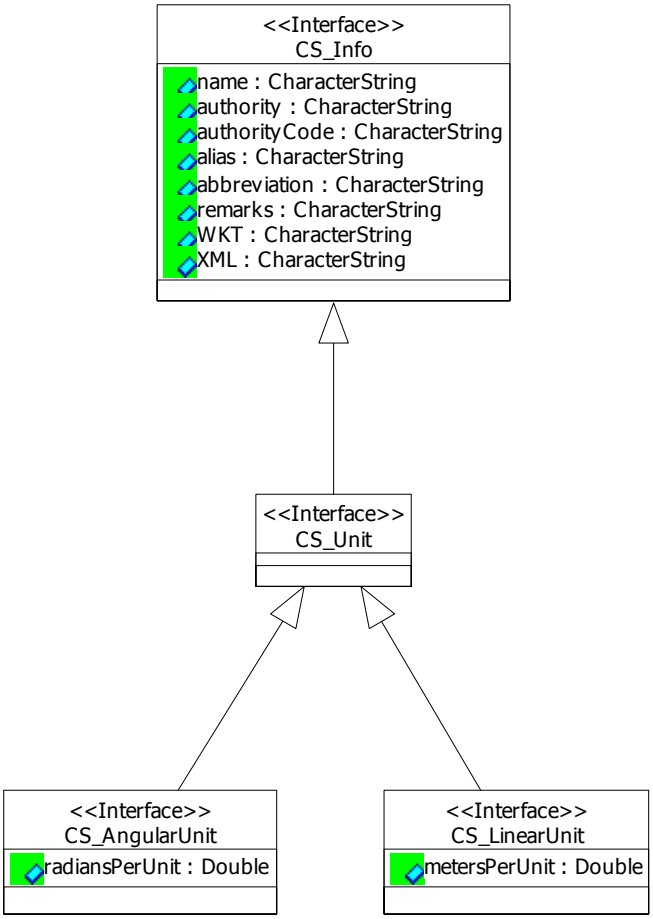

Figure 9. Units (Class Diagram)

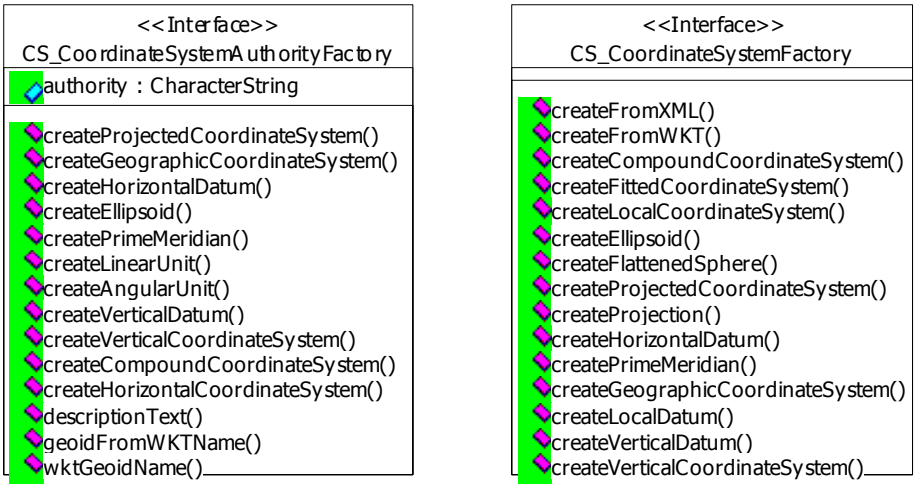

Figure 10. Coordinate System Factories (Class Diagram)

12.3.1 CS_AngularUnit Class

Definition of angular units.

12.3.1.1 radiansPerUnit (Attribute) :

Returns the number of radians per AngularUnit.

12.3.2 CS_AxisInfo Class

Details of axis. This is used to label axes, and indicate the orientation.

12.3.2.1 name (Attribute) :

Human readable name for axis. Possible values are X, Y, Long, Lat or any other short string.

12.3.2.2 orientation (Attribute) :

Gets enumerated value for orientation.

12.3.3 CS_AxisOrientationEnum Class

Orientation of axis. Some coordinate systems use non-standard orientations. For example, the first axis in South African grids usually points West, instead of East. This information is obviously relevant for algorithms converting South African grid coordinates into Lat/Long.

12.3.3.1 CS_AO_Other (Attribute) : 0

Unknown or unspecified axis orientation. This can be used for local or fitted coordinate systems.

12.3.3.2 CS_AO_North (Attribute) : 1

Increasing ordinates values go North. This is usually used for Grid Y coordinates and Latitude.

12.3.3.3 CS_AO_South (Attribute) : 2

Increasing ordinates values go South. This is rarely used.

12.3.3.4 CS_AO_East (Attribute) : 3

Increasing ordinates values go East. This is rarely used.

12.3.3.5 CS_AO_West (Attribute) : 4

Increasing ordinates values go West. This is usually used for Grid X coordinates and Longitude.

12.3.3.6 CS_AO_Up (Attribute) : 5

Increasing ordinates values go up. This is used for vertical coordinate systems.

12.3.3.7 CS_AO_Down (Attribute) : 6

Increasing ordinates values go down. This is used for vertical coordinate systems.

12.3.4 CS_CompoundCoordinateSystem Class

An aggregate of two coordinate systems (CRS). One of these is usually a CRS based on a two dimensional coordinate system such as a geographic or a projected coordinate system with a horizontal datum. The other is a vertical CRS which is a one-dimensional coordinate system with a vertical datum.

12.3.4.1 headCS (Attribute) :

Gets first sub-coordinate system.

12.3.4.2 tailCS (Attribute) :

Gets second sub-coordinate system.

12.3.5 CS_CoordinateSystem Class

Base interface for all coordinate systems.

A coordinate system is a mathematical space, where the elements of the space are called positions. Each position is described by a list of numbers. The length of the list corresponds to the dimension of the coordinate system. So in a 2D coordinate system each position is described by a list containing 2 numbers.

However, in a coordinate system, not all lists of numbers correspond to a position - some lists may be outside the domain of the coordinate system. For example, in a 2D Lat/Lon coordinate system, the list (91,91) does not correspond to a position.

Some coordinate systems also have a mapping from the mathematical space into locations in the real world. So in a Lat/Lon coordinate system, the mathematical position (lat, long) corresponds to a location on the surface of the Earth. This mapping from the mathematical space into real-world locations is called a Datum.

12.3.5.1 defaultEnvelope (Attribute) :

Gets default envelope of coordinate system. Coordinate systems which are bounded should return the minimum bounding box of their domain. Unbounded coordinate systems should return a box which is as large as is likely to be used. For example, a (lon,lat) geographic coordinate system in degrees should return a box from (-180,-90) to (180,90), and a geocentric coordinate system could return a box from (-r,-r,-r) to (+r,+r,+r) where r is the approximate radius of the Earth.

12.3.5.2 dimension (Attribute) :

Dimension of the coordinate system.

12.3.5.3 getAxis (dimension:Integer) : CS_AxisInfo

Gets axis details for dimension within coordinate system. Each dimension in the coordinate system has a corresponding axis.

12.3.5.4 getUnits (dimension:Integer) : CS_Unit

Gets units for dimension within coordinate system. Each dimension in the coordinate system has corresponding units.

12.3.6 CS_CoordinateSystemAuthorityFactory Class

Creates spatial reference objects using codes The codes are maintained by an external authority. A commonly used authority is EPSG, which is also used in the GeoTIFF standard.

12.3.6.1 authority (Attribute) :

Returns the authority name.

12.3.6.2 createAngularUnit (code:CharacterString) : CS_AngularUnit

Returns an AngularUnit object from a code.

12.3.6.3 createCompoundCoordinateSystem (code:CharacterString) : CS_CompoundCoordinateSystem

Creates a 3D coordinate system from a code.

12.3.6.4 createEllipsoid (code:CharacterString) : CS_Ellipsoid

Returns an Ellipsoid object from a code.

12.3.6.5 createGeographicCoordinateSystem (code:CharacterString) : CS_GeographicCoordinateSystem

Returns a GeographicCoordinateSystem object from a code.

12.3.6.6 createHorizontalCoordinateSystem (code:CharacterString) : CS_HorizontalCoordinateSystem

Creates a horizontal co-ordinate system from a code. The horizontal coordinate system could be geographic or projected.

12.3.6.7 createHorizontalDatum (code:CharacterString) : CS_HorizontalDatum

Returns a HorizontalDatum object from a code.

12.3.6.8 createLinearUnit (code:CharacterString) : CS_LinearUnit

Returns a LinearUnit object from a code.

12.3.6.9 createPrimeMeridian (code:CharacterString) : CS_PrimeMeridian

Returns a PrimeMeridian object from a code.

12.3.6.10 createProjectedCoordinateSystem (code:CharacterString) : CS_ProjectedCoordinateSystem

Returns a ProjectedCoordinateSystem object from a code.

12.3.6.11 createVerticalCoordinateSystem (code:CharacterString) : CS_VerticalCoordinateSystem

Create a vertical coordinate system from a code.

12.3.6.12 createVerticalDatum (code:CharacterString) : CS_VerticalDatum

Creates a vertical datum from a code.

12.3.6.13 descriptionText (code:CharacterString) : CharacterString

Gets a description of the object corresponding to a code.

12.3.6.14 geoidFromWKTName (wkt:CharacterString) : CharacterString

Gets the Geoid code from a WKT name. In the OGC definition of WKT horizontal datums, the geoid is referenced by a quoted string, which is used as a key value. This method converts the key value string into a code recognized by this authority.

12.3.6.15 wktGeoidName (geoid:Integer) : CharacterString

Gets the WKT name of a Geoid. In the OGC definition of WKT horizontal datums, the geoid is referenced by a quoted string, which is used as a key value. This method gets the OGC WKT key value from a geoid code.

12.3.7 CS_CoordinateSystemFactory Class

Builds up complex objects from simpler objects or values.

CS_CoordinateSystemFactory allows applications to make coordinate systems that cannot be created by a CS_CoordinateSystemAuthorityFactory. This factory is very flexible, whereas the authority factory is easier to use.

So CS_CoordinateSystemAuthorityFactory can be used to make 'standard' coordinate systems, and CS_CoordinateSystemAuthorityFactory can be used to make 'special' coordinate systems.

For example, the EPSG authority has codes for USA state plane coordinate systems using the NAD83 datum, but these coordinate systems always use meters. EPSG does not have codes for NAD83 state plane coordinate systems that use feet units. This factory lets an application create such a hybrid coordinate system.

12.3.7.1 createCompoundCoordinateSystem (name:CharacterString, head:CS_CoordinateSystem, tail:CS_CoordinateSystem) : CS_CompoundCoordinateSystem

Creates a compound coordinate system.

12.3.7.2 createEllipsoid (name:CharacterString, semiMajorAxis:Double, semiMinorAxis:Double, linearUnit:CS_LinearUnit) : CS_Ellipsoid

Creates an ellipsoid from radius values.

12.3.7.3 createFittedCoordinateSystem (name:CharacterString, base:CS_CoordinateSystem, toBaseWKT:CharacterString, arAxes:Sequence<CS_AxisInfo>) : CS_FittedCoordinateSystem

Creates a fitted coordinate system. The units of the axes in the fitted coordinate system will be inferred from the units of the base coordinate system. If the affine map performs a rotation, then any mixed axes must have identical units. For example, a (lat_deg,lon_deg,height_feet) system can be rotated in the (lat,lon) plane, since both affected axes are in degrees. But you should not rotate this coordinate system in any other plane.

12.3.7.4 createFlattenedSphere (name:CharacterString, semiMajorAxis:Double, inverseFlattening:Double, linearUnit:CS_LinearUnit) : CS_Ellipsoid

Creates an ellipsoid from an major radius, and inverse flattening.

12.3.7.5 createFromWKT (wellKnownText:CharacterString) : CS_CoordinateSystem

Creates a coordinate system object from a Well-Known Text string.

12.3.7.6 createFromXML (xml:CharacterString) : CS_CoordinateSystem

Creates a coordinate system object from an XML string.

12.3.7.7 createGeographicCoordinateSystem (name:CharacterString, angularUnit:CS_AngularUnit, horizontalDatum:CS_HorizontalDatum, primeMeridian:CS_PrimeMeridian, axis0:CS_AxisInfo, axis1:CS_AxisInfo) : CS_GeographicCoordinateSystem

Creates a GCS, which could be Lat/Lon or Lon/Lat.

12.3.7.8 createHorizontalDatum (name:CharacterString, horizontalDatumType:CS_DatumType, ellipsoid:CS_Ellipsoid, toWGS84:CS_WGS84ConversionInfo) : CS_HorizontalDatum

Creates horizontal datum from ellipsoid and Bursa-World parameters. Since this method contains a set of Bursa-Wolf parameters, the created datum will always have a relationship to WGS84. If you wish to create a horizontal datum that has no relationship with WGS84, then you can either specify a horizontalDatumType of CS_HD_Other, or create it via WKT.

12.3.7.9 createLocalCoordinateSystem (name:CharacterString, datum:CS_LocalDatum, unit:CS_Unit, arAxes:Sequence<CS_AxisInfo>) : CS_LocalCoordinateSystem

Creates a local coordinate system. The dimension of the local coordinate system is determined by the size of the axis array. All the axes will have the same units. If you want to make a coordinate system with mixed units, then you can make a compound coordinate system from different local coordinate systems.

12.3.7.10 createLocalDatum (name:CharacterString, localDatumType:CS_DatumType) : CS_LocalDatum

Creates a local datum.

12.3.7.11 createPrimeMeridian (name:CharacterString, angularUnit:CS_AngularUnit, longitude:Double) : CS_PrimeMeridian

Creates a prime meridian, relative to Greenwich.

12.3.7.12 createProjectedCoordinateSystem (name:CharacterString, gcs:CS_GeographicCoordinateSystem, projection:CS_Projection, linearUnit:CS_LinearUnit, axis0:CS_AxisInfo, axis1:CS_AxisInfo) : CS_ProjectedCoordinateSystem

Creates a projected coordinate system using a projection object.

12.3.7.13 createProjection (name:CharacterString, wktProjectionClass:CharacterString, parameters:Sequence<CS_ProjectionParameter>) : CS_Projection

Creates a projection.

12.3.7.14 createVerticalCoordinateSystem (name:CharacterString, verticalDatum:CS_VerticalDatum, verticalUnit:CS_LinearUnit, axis:CS_AxisInfo) : CS_VerticalCoordinateSystem

Creates a vertical coordinate system from a datum and linear units.

12.3.7.15 createVerticalDatum (name:CharacterString, verticalDatumType:CS_DatumType) : CS_VerticalDatum

Creates a vertical datum from an enumerated type value.

12.3.8 CS_Datum Class

A set of quantities from which other quantities are calculated. For the OGC abstract model, it can be defined as a set of real points on the earth that have coordinates. EG. A datum can be thought of as a set of parameters defining completely the origin and orientation of a coordinate system with respect to the earth. A textual description and/or a set of parameters describing the relationship of a coordinate system to some predefined physical locations (such as center of mass) and physical directions (such as axis of spin). The definition of the datum may also include the temporal behavior (such as the rate of change of the orientation of the coordinate axes).

12.3.8.1 datumType (Attribute) :

Gets the type of the datum as an enumerated code.

12.3.9 CS_DatumType Class

Type of the datum expressed as an enumerated value. The enumeration is split into ranges which indicate the datum's type. The value should be one of the predefined values, or within the range for local types. This will allow OGC to coordinate the addition of new interoperable codes.

12.3.9.1 CS_HD_Classic (Attribute) : 1001

These datums, such as ED50, NAD27 and NAD83, have been designed to support horizontal positions on the ellipsoid as opposed to positions in 3-D space. These datums were designed mainly to support a horizontal component of a position in a domain of limited extent, such as a country, a region or a continent.

12.3.9.2 CS_HD_Geocentric (Attribute) : 1002

A geocentric datum is a "satellite age" modern geodetic datum mainly of global extent, such as WGS84 (used in GPS), PZ90 (used in GLONASS) and ITRF. These datums were designed to support both a horizontal component of position and a vertical component of position (through ellipsoidal heights). The regional realizations of ITRF, such as ETRF, are also included in this category.

12.3.9.3 CS_HD_Max (Attribute) : 1999

Highest possible value for horizontal datum types.

12.3.9.4 CS_HD_Min (Attribute) : 1000

Lowest possible value for horizontal datum types.

12.3.9.5 CS_HD_Other (Attribute) : 1000

Unspecified horizontal datum type. Horizontal datums with this type should never supply a conversion to WGS84 using Bursa Wolf parameters.

12.3.9.6 CS_LD_Max (Attribute) : 32767

Highest possible value for local datum types.

12.3.9.7 CS_LD_Min (Attribute) : 10000

Lowest possible value for local datum types.

12.3.9.8 CS_VD_AltitudeBarometric (Attribute) : 2003

The vertical datum of altitudes or heights in the atmosphere. These are approximations of orthometric heights obtained with the help of a barometer or a barometric altimeter. These values are usually expressed in one of the following units: meters, feet, millibars (used to measure pressure levels), or theta value (units used to measure geopotential height).

12.3.9.9 CS_VD_Depth (Attribute) : 2006

This attribute is used to support the set of datums generated for hydrographic engineering projects where depth measurements below sea level are needed. It is often called a hydrographic or a marine datum. Depths are measured in the direction perpendicular (approximately) to the actual equipotential surfaces of the earth's gravity field, using such procedures as echo-sounding.

12.3.9.10 CS_VD_Ellipsoidal (Attribute) : 2002

A vertical datum for ellipsoidal heights that are measured along the normal to the ellipsoid used in the definition of horizontal datum.

12.3.9.11 CS_VD_GeoidModelDerived (Attribute) : 2005

A vertical datum of geoid model derived heights, also called GPS-derived heights. These heights are approximations of orthometric heights (H), constructed from the ellipsoidal heights (h) by the use of the given geoid undulation model (N) through the equation: H=h-N.

12.3.9.12 CS_VD_Max (Attribute) : 2999

Highest possible value for vertical datum types.

12.3.9.13 CS_VD_Min (Attribute) : 2000

Lowest possible value for vertical datum types.

12.3.9.14 CS_VD_Normal (Attribute) : 2004

A normal height system.

12.3.9.15 CS_VD_Orthometric (Attribute) : 2001

A vertical datum for orthometric heights that are measured along the plumb line.

12.3.9.16 CS_VD_Other (Attribute) : 2000

Unspecified vertical datum type.

12.3.10 CS_Ellipsoid Class

An approximation of the Earth's surface as a squashed sphere.

12.3.10.1 axisUnit (Attribute) :

Returns the LinearUnit. The units of the semi-major and semi-minor axis values.

12.3.10.2 inverseFlattening (Attribute) :

Returns the value of the inverse of the flattening constant. The inverse flattening is related to the equatorial/polar radius by the formula ivf=re/(re-rp). For perfect spheres, this formula breaks down, and a special IVF value of zero is used.

12.3.10.3 ivfDefinitive (Attribute) :

Is the Inverse Flattening definitive for this ellipsoid? Some ellipsoids use the IVF as the defining value, and calculate the polar radius whenever asked. Other ellipsoids use the polar radius to calculate the IVF whenever asked. This distinction can be important to avoid floating-point rounding errors.

12.3.10.4 semiMajorAxis (Attribute) :

Gets the equatorial radius. The returned length is expressed in this object's axis units.

12.3.10.5 semiMinorAxis (Attribute) :

Gets the polar radius. The returned length is expressed in this object's axis units.

12.3.11 CS_FittedCoordinateSystem Class

A coordinate system which sits inside another coordinate system. The fitted coordinate system can be rotated and shifted, or use any other math transform to inject itself into the base coordinate system.

12.3.11.1 baseCoordinateSystem (Attribute) :

Gets underlying coordinate system.

12.3.11.2 toBase (Attribute) :

Gets Well-Known Text of a math transform to the base coordinate system. The dimension of this fitted coordinate system is determined by the source dimension of the math transform. The transform should be one-to-one within this coordinate system's domain, and the base coordinate system dimension must be at least as big as the dimension of this coordinate system.

12.3.12 CS_GeocentricCoordinateSystem Class

A 3D coordinate system, with its origin at the centre of the Earth. The X axis points towards the prime meridian. The Y axis points East or West. The Z axis points North or South. By default the Z axis will point North, and the Y axis will point East (e.g. a right handed system), but you should check the axes for non-default values.

12.3.12.1 horizontalDatum (Attribute) :

Returns the HorizontalDatum. The horizontal datum is used to determine where the centre of the Earth is considered to be. All coordinate points will be measured from the centre of the Earth, and not the surface.

12.3.12.2 linearUnit (Attribute) :

Gets the units used along all the axes.

12.3.12.3 primeMeridian (Attribute) :

Returns the PrimeMeridian.

12.3.13 CS_GeographicCoordinateSystem Class

A coordinate system based on latitude and longitude. Some geographic coordinate systems are Lat/Lon, and some are Lon/Lat. You can find out which this is by examining the axes. You should also check the angular units, since not all geographic coordinate systems use degrees.

12.3.13.1 angularUnit (Attribute) :

Returns the AngularUnit. The angular unit must be the same as the CS_CoordinateSystem units.

12.3.13.2 numConversionToWGS84 (Attribute) :

Gets the number of available conversions to WGS84 coordinates.

12.3.13.3 primeMeridian (Attribute) :

Returns the PrimeMeridian.

12.3.13.4 getWGS84ConversionInfo (index:Integer) : CS_WGS84ConversionInfo

Gets details on a conversion to WGS84. Some geographic coordinate systems provide several transformations into WGS84, which are designed to provide good accuracy in different areas of interest. The first conversion (with index=0) should provide acceptable accuracy over the largest possible area of interest.

12.3.14 CS_HorizontalCoordinateSystem Class

A 2D coordinate system suitable for positions on the Earth's surface.

12.3.14.1 horizontalDatum (Attribute) :

Returns the HorizontalDatum.

12.3.15 CS_HorizontalDatum Class

Procedure used to measure positions on the surface of the Earth.

12.3.15.1 ellipsoid (Attribute) :

Returns the Ellipsoid.

12.3.15.2 WGS84Parameters (Attribute) :

Gets preferred parameters for a Bursa Wolf transformation into WGS84. The 7 returned values correspond to (dx,dy,dz) in meters, (ex,ey,ez) in arc-seconds, and scaling in parts-per-million.

12.3.16 CS_Info Class

A base interface for metadata applicable to coordinate system objects.

The metadata items ‘Abbreviation’, ‘Alias’, ‘Authority’, ‘AuthorityCode’, ‘Name’ and ‘Remarks’ were specified in the Simple Features interfaces, so they have been kept here.

This specification does not dictate what the contents of these items should be. However, the following guidelines are suggested:

When CS_CoordinateSystemAuthorityFactory is used to create an object, the ‘Authority’ and ‘AuthorityCode’ values should be set to the authority name of the factory object, and the authority code supplied by the client, respectively. The other values may or may not be set. (If the authority is EPSG, the implementer may consider using the corresponding metadata values in the EPSG tables.)

When CS_CoordinateSystemFactory creates an object, the ‘Name’ should be set to the value supplied by the client. All of the other metadata items should be left empty.

12.3.16.1 abbreviation (Attribute) :

Gets the abbreviation.

12.3.16.2 alias (Attribute) :

Gets the alias.

12.3.16.3 authority (Attribute) :

Gets the authority name.

An Authority is an organization that maintains definitions of Authority Codes. For example the European Petroleum Survey Group (EPSG) maintains a database of coordinate systems, and other spatial referencing objects, where each object has a code number ID. For example, the EPSG code for a WGS84 Lat/Lon coordinate system is ‘4326’.

12.3.16.4 authorityCode (Attribute) :

Gets the authority-specific identification code.

The AuthorityCode is a compact string defined by an Authority to reference a particular spatial reference object. For example, the European Survey Group (EPSG) authority uses 32 bit integers to reference coordinate systems, so all their code strings will consist of a few digits. The EPSG code for WGS84 Lat/Lon is ‘4326’.

12.3.16.5 name (Attribute) :

Gets the name.

12.3.16.6 remarks (Attribute) :

Gets the provider-supplied remarks.

12.3.16.7 WKT (Attribute) :

Gets a Well-Known text representation of this object.

12.3.16.8 XML (Attribute) :

Gets an XML representation of this object.

12.3.17 CS_LinearUnit Class

Definition of linear units.

12.3.17.1 metersPerUnit (Attribute) :

Returns the number of meters per LinearUnit.

12.3.18 CS_LocalCoordinateSystem Class

A local coordinate system, with uncertain relationship to the world. In general, a local coordinate system cannot be related to other coordinate systems. However, if two objects supporting this interface have the same dimension, axes, units and datum then client code is permitted to assume that the two coordinate systems are identical. This allows several datasets from a common source (e.g. a CAD system) to be overlaid. In addition, some implementations of the Coordinate Transformation (CT) package may have a mechanism for correlating local datums. (E.g. from a database of transformations, which is created and maintained from real-world measurements.)

12.3.18.1 localDatum (Attribute) :

Gets the local datum.

12.3.19 CS_LocalDatum Class

Local datum. If two local datum objects have the same datum type and name, then they can be considered equal. This means that coordinates can be transformed between two different local coordinate systems, as long as they are based on the same local datum.

12.3.20 CS_PrimeMeridian Class

A meridian used to take longitude measurements from.

12.3.20.1 angularUnit (Attribute) :

Returns the AngularUnits.

12.3.20.2 longitude (Attribute) :

Returns the longitude value relative to the Greenwich Meridian. The longitude is expressed in this objects angular units.

12.3.21 CS_ProjectedCoordinateSystem Class

A 2D cartographic coordinate system.

12.3.21.1 geographicCoordinateSystem (Attribute) :

Returns the GeographicCoordinateSystem.

12.3.21.2 linearUnit (Attribute) :

Returns the LinearUnits. The linear unit must be the same as the CS_CoordinateSystem units.

12.3.21.3 projection (Attribute) :

Gets the projection.

12.3.22 CS_Projection Class

A projection from geographic coordinates to projected coordinates.

12.3.22.1 className (Attribute) :

Gets the projection classification name (e.g. ‘Transverse_Mercator’).

12.3.22.2 numParameters (Attribute) :

Gets number of parameters of the projection.

12.3.22.3 getParameter (index:Integer) : CS_ProjectionParameter

Gets an indexed parameter of the projection.

12.3.23 CS_ProjectionParameter Class

A named projection parameter value. The linear units of parameters' values match the linear units of the containing projected coordinate system. The angular units of parameter values match the angular units of the geographic coordinate system that the projected coordinate system is based on. (Notice that this is different from CT_Parameter, where the units are always meters and degrees.)

12.3.23.1 name (Attribute) :

The parameter name.

12.3.23.2 value (Attribute) :

The parameter value.

12.3.24 CS_Unit Class

Base interface for defining units.

12.3.25 CS_VerticalCoordinateSystem Class

A one-dimensional coordinate system suitable for vertical measurements.

12.3.25.1 verticalDatum (Attribute) :

Gets the vertical datum, which indicates the measurement method.

12.3.25.2 verticalUnit (Attribute) :

Gets the units used along the vertical axis. The vertical units must be the same as the CS_CoordinateSystem units.

12.3.26 CS_VerticalDatum Class

Proceedure used to measure vertical distances.

12.3.27 CS_WGS84ConversionInfo Class

Parameters for a geographic transformation into WGS84. The Bursa Wolf parameters should be applied to geocentric coordinates, where the X axis points towards the Greenwich Prime Meridian, the Y axis points East, and the Z axis points North.

12.3.27.1 areaOfUse (Attribute) :

Human readable text describing intended region of transformation.

12.3.27.2 dx (Attribute) :

Bursa Wolf shift in meters.

12.3.27.3 dy (Attribute) :

Bursa Wolf shift in meters.

12.3.27.4 dz (Attribute) :

Bursa Wolf shift in meters.

12.3.27.5 ex (Attribute) :

Bursa Wolf rotation in arc seconds.

12.3.27.6 ey (Attribute) :

Bursa Wolf rotation in arc seconds.

12.3.27.7 ez (Attribute) :

Bursa Wolf rotation in arc seconds.

12.3.27.8 ppm (Attribute) :

Bursa Wolf scaling in parts per million.

12.4 CT Logical Package: Co-ordinate Transformations

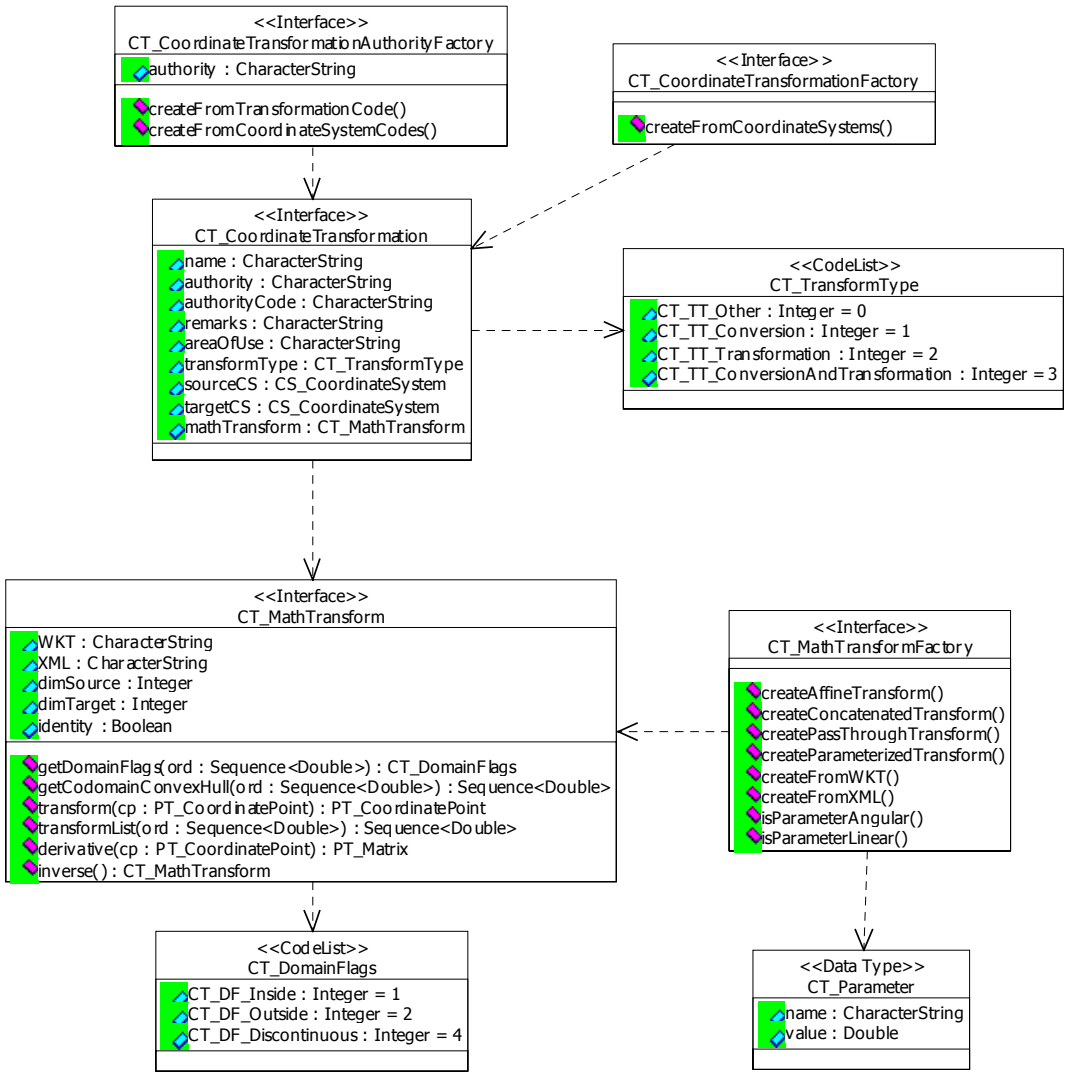

Figure 11. Coordinate Transformation (Class Diagram)

12.4.1 CT_CoordinateTransformation Class

Describes a coordinate transformation. This interface only describes a coordinate transformation, it does not actually perform the transform operation on points. To transform points you must use a math transform.

The math transform will transform positions in the source coordinate system into positions in the target coordinate system.

12.4.1.1 areaOfUse (Attribute) :

Human readable description of domain in source coordinate system.

12.4.1.2 authority (Attribute) :

Authority which defined transformation and parameter values.

An Authority is an organization that maintains definitions of Authority Codes. For example the European Petroleum Survey Group (EPSG) maintains a database of coordinate systems, and other spatial referencing objects, where each object has a code number ID. For example, the EPSG code for a WGS84 Lat/Lon coordinate system is ‘4326’.

12.4.1.3 authorityCode (Attribute) :

Code used by authority to identify transformation. An empty string is used for no code.

The AuthorityCode is a compact string defined by an Authority to reference a particular spatial reference object. For example, the European Survey Group (EPSG) authority uses 32 bit integers to reference coordinate systems, so all their code strings will consist of a few digits. The EPSG code for WGS84 Lat/Lon is ‘4326’.

12.4.1.4 mathTransform (Attribute) :

Gets math transform.

12.4.1.5 name (Attribute) :

Name of transformation.

12.4.1.6 remarks (Attribute) :

Gets the provider-supplied remarks.

12.4.1.7 sourceCS (Attribute) :

Source coordinate system.

12.4.1.8 targetCS (Attribute) :

Target coordinate system.

12.4.1.9 transformType (Attribute) :

Semantic type of transform. For example, a datum transformation or a coordinate conversion.

12.4.2 CT_CoordinateTransformationAuthorityFactory Class

Creates coordinate transformation objects from codes. The codes are maintained by an external authority. A commonly used authority is EPSG, which is also used in the GeoTIFF standard

12.4.2.1 authority (Attribute) :

The name of the authority.

12.4.2.2 createFromCoordinateSystemCodes (sourceCode:CharacterString, targetCode: CharacterString) : CT_CoordinateTransformation

Creates a transformation from coordinate system codes.

12.4.2.3 createFromTransformationCode (code:CharacterString) : CT_CoordinateTransformation

Creates a transformation from a single transformation code.

The ‘Authority’ and ‘AuthorityCode’ values of the created object will be set to the authority of this object, and the code supplied by the client, respectively. The other metadata values may or may not be set.

12.4.3 CT_CoordinateTransformationFactory Class

Creates coordinate transformations.

12.4.3.1 createFromCoordinateSystems (sourceCS:CS_CoordinateSystem, targetCS:CS_CoordinateSystem) : CT_CoordinateTransformation

Creates a transformation between two coordinate systems. This method will examine the coordinate systems in order to construct a transformation between them. This method may fail if no path between the coordinate systems is found, using the normal failing behavior of the DCP (e.g. throwing an exception).

12.4.4 CT_DomainFlags Class

Flags indicating parts of domain covered by a convex hull. These flags can be combined. For example, the value 3 corresponds to a combination of CT_DF_Inside and MF_DF_Outside, which means that some parts of the convex hull are inside the domain, and some parts of the convex hull are outside the domain.

12.4.4.1 CT_DF_Inside (Attribute) : 1

At least one point in a convex hull is inside the transform's domain.

12.4.4.2 CT_DF_Outside (Attribute) : 2

At least one point in a convex hull is outside the transform's domain.

12.4.4.3 CT_DF_Discontinuous (Attribute) : 4

At least one point in a convex hull is not transformed continuously. As an example, consider a "Longitude_Rotation" transform which adjusts longitude coordinates to take account of a change in Prime Meridian. If the rotation is 5 degrees east, then the point (Lat=175,Lon=0) is not transformed continuously, since it is on the meridian line which will be split at +180/-180 degrees.

12.4.5 CT_MathTransform Class

Transforms muti-dimensional coordinate points.

If a client application wishes to query the source and target coordinate systems of a transformation, then it should keep hold of the CT_CoordinateTransformation interface, and use the contained math transform object whenever it wishes to perform a transform.

12.4.5.1 dimSource (Attribute) :

Gets the dimension of input points.

12.4.5.2 dimTarget (Attribute) :

Gets the dimension of output points.

12.4.5.3 identity (Attribute) :

Tests whether this transform does not move any points.

12.4.5.4 WKT (Attribute) :

Gets a Well-Known text representation of this object.

12.4.5.5 XML (Attribute) :

Gets an XML representation of this object.

12.4.5.6 derivative (cp:PT_CoordinatePoint) : PT_Matrix

Gets the derivative of this transform at a point. If the transform does not have a well-defined derivative at the point, then this function should fail in the usual way for the DCP. The derivative is the matrix of the non-translating portion of the approximate affine map at the point. The matrix will have dimensions corresponding to the source and target coordinate systems. If the input dimension is M, and the output dimension is N, then the matrix will have size [M][N]. The elements of the matrix {elt[n][m] : n=0..(N-1)} form a vector in the output space which is parallel to the displacement caused by a small change in the m'th ordinate in the input space.

12.4.5.7 getCodomainConvexHull (ord:Sequence<Double>) : Sequence<Double>

Gets transformed convex hull. The supplied ordinates are interpreted as a sequence of points, which generates a convex hull in the source space. The returned sequence of ordinates represents a convex hull in the output space. The number of output points will often be different from the number of input points. Each of the input points should be inside the valid domain (this can be checked by testing the points' domain flags individually). However, the convex hull of the input points may go outside the valid domain. The returned convex hull should contain the transformed image of the intersection of the source convex hull and the source domain.

A convex hull is a shape in a coordinate system, where if two positions A and B are inside the shape, then all positions in the straight line between A and B are also inside the shape. So in 3D a cube and a sphere are both convex hulls. Other less obvious examples of convex hulls are straight lines, and single points. (A single point is a convex hull, because the positions A and B must both be the same - i.e. the point itself. So the straight line between A and B has zero length.)

Some examples of shapes that are NOT convex hulls are donuts, and horseshoes.

12.4.5.8 getDomainFlags (ord:Sequence<Double>) : CT_DomainFlags

Gets flags classifying domain points within a convex hull. The supplied ordinates are interpreted as a sequence of points, which generates a convex hull in the source space. Conceptually, each of the (usually infinite) points inside the convex hull is then tested against the source domain. The flags of all these tests are then combined. In practice, implementations of different transforms will use different short-cuts to avoid doing an infinite number of tests.

12.4.5.9 inverse () : CT_MathTransform

Creates the inverse transform of this object. This method may fail if the transform is not one to one. However, all cartographic projections should succeed.

12.4.5.10 transform (cp:PT_CoordinatePoint) : PT_CoordinatePoint

Transforms a coordinate point. The passed parameter point should not be modified.

12.4.5.11 transformList (ord:Sequence<Double>) : Sequence<Double>

Transforms a list of coordinate point ordinal values. This method is provided for efficiently transforming many points. The supplied array of ordinal values will contain packed ordinal values. For example, if the source dimension is 3, then the ordinals will be packed in this order (x0,y0,z0,x1,y1,z1 ...). The size of the passed array must be an integer multiple of DimSource. The returned ordinal values are packed in a similar way. In some DCPs. the ordinals may be transformed in-place, and the returned array may be the same as the passed array. So any client code should not attempt to reuse the passed ordinal values (although they can certainly reuse the passed array). If there is any problem then the server implementation will throw an exception. If this happens then the client should not make any assumptions about the state of the ordinal values.

12.4.6 CT_MathTransformFactory` Class

Creates math transforms.

CT_MathTransformFactory is a low level factory that is used to create CT_MathTransform objects. Many high level GIS applications will never need to use a CT_MathTransformFactory directly; they can use a CT_CoordinateTransformationFactory instead. However, the CT_MathTransformFactory interface is specified here, since it can be used directly by applications that wish to transform other types of coordinates (e.g. color coordinates, or image pixel coordinates).

The following comments assume that the same vendor implements the math transform factory interfaces and math transform interfaces.

A math transform is an object that actually does the work of applying formulae to coordinate values. The math transform does not know or care how the coordinates relate to positions in the real world. This lack of semantics makes implementing CT_MathTransformFactory significantly easier than it would be otherwise.

For example CT_MathTransformFactory can create affine math transforms. The affine transform applies a matrix to the coordinates without knowing how what it is doing relates to the real world. So if the matrix scales Z values by a factor of 1000, then it could be converting meters into millimeters, or it could be converting kilometers into meters.

Because math transforms have low semantic value (but high mathematical value), programmers who do not have much knowledge of how GIS applications use coordinate systems, or how those coordinate systems relate to the real world can implement CT_MathTransformFactory.

The low semantic content of math transforms also means that they will be useful in applications that have nothing to do with GIS coordinates. For example, a math transform could be used to map color coordinates between different color spaces, such as converting (red, green, blue) colors into (hue, light, saturation) colors.

Since a math transform does not know what its source and target coordinate systems mean, it is not necessary or desirable for a math transform object to keep information on its source and target coordinate systems.

12.4.6.1 createAffineTransform (matrix:PT_Matrix) : CT_MathTransform

Creates an affine transform from a matrix. If the transform's input dimension is M, and output dimension is N, then the matrix will have size [N+1][M+1]. The +1 in the matrix dimensions allows the matrix to do a shift, as well as a rotation. The [M][j] element of the matrix will be the j'th ordinate of the moved origin. The [i][N] element of the matrix will be 0 for i less than M, and 1 for i equals M.

12.4.6.2 createConcatenatedTransform (transform1:CT_MathTransform, transform2:CT_MathTransform) : CT_MathTransform

Creates a transform by concatenating two existing transforms. A concatenated transform acts in the same way as applying two transforms, one after the other.

The dimension of the output space of the first transform must match the dimension of the input space in the second transform. If you wish to concatenate more than two transforms, then you

can repeatedly use this method.

12.4.6.3 createFromWKT (wellKnownText:CharacterString) : CT_MathTransform

Creates a math transform from a Well-Known Text string.

12.4.6.4 createFromXML (xml:CharacterString) : CT_MathTransform

Creates a math transform from XML.

12.4.6.5 createParameterizedTransform (classification:CharacterString, parameters:Sequence<CT_Parameter>) : CT_MathTransform

Creates a transform from a classification name and parameters. The client must ensure that all the linear parameters are expressed in meters, and all the angular parameters are expressed in degrees. Also, they must supply "semi_major" and "semi_minor" parameters for cartographic projection transforms.

12.4.6.6 createPassThroughTransform (firstAffectedOrdinate:Integer, subTransform:CT_MathTransform) : CT_MathTransform

Creates a transform which passes through a subset of ordinates to another transform. This allows transforms to operate on a subset of ordinates. For example, if you have (Lat,Lon,Height) coordinates, then you may wish to convert the height values from meters to feet without affecting the (Lat,Lon) values. If you wanted to affect the (Lat,Lon) values and leave the Height values alone, then you would have to swap the ordinates around to (Height,Lat,Lon). You can do this with an affine map.

12.4.6.7 isParameterAngular (parameterName:CharacterString) : Boolean

Tests whether parameter is angular. Clients must ensure that all angular parameter values are in degrees.

12.4.6.8 isParameterLinear (parameterName:CharacterString) : Boolean

Tests whether parameter is linear. Clients must ensure that all linear parameter values are in meters.

12.4.7 CT_Parameter Class

A named parameter value.

12.4.7.1 name (Attribute) :

The parameter name.

12.4.7.2 value (Attribute) :

The parameter value.

12.4.8 CT_TransformType Class

Semantic type of transform used in coordinate transformation.

12.4.8.1 CT_TT_Other (Attribute) : 0

Unknown or unspecified type of transform.

12.4.8.2 CT_TT_Conversion (Attribute) : 1

Transform depends only on defined parameters. For example, a cartographic projection.

12.4.8.3 CT_TT_Transformation (Attribute) : 2

Transform depends only on empirically derived parameters. For example a datum transformation.

12.4.8.4 CT_TT_ConversionAndTransformation (Attribute) : 3

Transform depends on both defined and empirical parameters.

13 Appendix A: Interface Definition Files

13.1 COM

13.1.1 OGC_PT.IDL

```
// PT package, COM profile.
// Copyright (c) OpenGIS Consortium Thursday, October 19, 2000.

import "ocidl.idl";

// Forward declarations.

struct PT_CoordinatePoint;
struct PT_Envelope;
struct PT_Matrix;

typedef
[
    helpstring("A position defined by a list of numbers."),
    uuid(AE27FB39-983F-11d3-8163-00C04F680FFF)
]
struct PT_CoordinatePoint
{
    [helpstring("The ordinates of the coordinate point.")]
    SAFEARRAY(DOUBLE) Ord;
} PT_CoordinatePoint;

typedef
[
    helpstring("A box defined by two positions."),
    uuid(AE27FB3A-983F-11d3-8163-00C04F680FFF)
]
struct PT_Envelope
{
    [helpstring("Point containing minimum ordinate values.")]
    PT_CoordinatePoint MinCP;
    [helpstring("Point containing maximum ordinate values.")]
    PT_CoordinatePoint MaxCP;
} PT_Envelope;

typedef
[
    helpstring("A two dimensional array of numbers."),
    uuid(AE27FB38-983F-11d3-8163-00C04F680FFF)
]
struct PT_Matrix
{
    [helpstring("Elements of the matrix.")]
    SAFEARRAY(DOUBLE) Elt;
} PT_Matrix;

#ifdef _OGC_PT_TLB_
[
    version(1.0),
    uuid(AE27FB36-983F-11d3-8163-00C04F680FFF),
    helpstring("OpenGIS OGC_PT Type Library")
]
library OGC_PT
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    struct PT_CoordinatePoint;
    struct PT_Envelope;
    struct PT_Matrix;
};
#endif
```

13.1.2 OGC_CS.IDL

```
// CS package, COM profile.
// Copyright (c) OpenGIS Consortium Thursday, October 19, 2000.

import "ocidl.idl";
import "OGC_PT.idl";

// Forward declarations.

enum CS_AxisOrientationEnum;
enum CS_DatumType;

struct CS_AxisInfo;
struct CS_ProjectionParameter;
struct CS_WGS84ConversionInfo;

interface CS_Info;
interface CS_Unit;
interface CS_CoordinateSystem;
interface CS_CoordinateSystemAuthorityFactory;
interface CS_LinearUnit;
interface CS_Datum;
interface CS_Ellipsoid;
interface CS_FittedCoordinateSystem;
interface CS_HorizontalDatum;
interface CS_HorizontalCoordinateSystem;
interface CS_AngularUnit;
interface CS_LocalDatum;
interface CS_CompoundCoordinateSystem;
interface CS_PrimeMeridian;
interface CS_LocalCoordinateSystem;
interface CS_GeographicCoordinateSystem;
interface CS_Projection;
interface CS_ProjectedCoordinateSystem;
interface CS_VerticalDatum;
interface CS_GeocentricCoordinateSystem;
interface CS_VerticalCoordinateSystem;
interface CS_CoordinateSystemFactory;


typedef
[
    helpstring("Orientation of axis."),
    uuid(57B0C0FE-967D-11d3-8161-00C04F680FFF)
]
enum CS_AxisOrientationEnum
{
    CS_AO_Other=0,
    CS_AO_North=1,
    CS_AO_South=2,
    CS_AO_East=3,
    CS_AO_West=4,
    CS_AO_Up=5,
    CS_AO_Down=6
} CS_AxisOrientationEnum;

typedef
[
    helpstring("Type of the datum expressed as an enumerated value."),
    uuid(57B0C0FF-967D-11d3-8161-00C04F680FFF)
]
enum CS_DatumType
{
    CS_HD_Min=1000,
    CS_HD_Other=1000,
    CS_HD_Classic=1001,
    CS_HD_Geocentric=1002,
    CS_HD_Max=1999,
    CS_VD_Min=2000,
```

```
    CS_VD_Other=2000,
    CS_VD_Orthometric=2001,
    CS_VD_Ellipsoidal=2002,
    CS_VD_AltitudeBarometric=2003,
    CS_VD_Normal=2004,
    CS_VD_GeoidModelDerived=2005,
    CS_VD_Depth=2006,
    CS_VD_Max=2999,
    CS_LD_Min=10000,
    CS_LD_Max=32767
} CS_DatumType;

typedef
[
    helpstring("Details of axis."),
    uuid(57B0C0F7-967D-11d3-8161-00C04F680FFF)
]
struct CS_AxisInfo
{
    [helpstring("Gets enumerated value for orientation.")]
    CS_AxisOrientationEnum Orientation;
    [helpstring("Human readable name for axis.")]
    BSTR Name;
} CS_AxisInfo;

typedef
[
    helpstring("A named projection parameter value."),
    uuid(57B0C100-967D-11d3-8161-00C04F680FFF)
]
struct CS_ProjectionParameter
{
    [helpstring("The parameter name.")]
    BSTR Name;
    [helpstring("The parameter value.")]
    DOUBLE Value;
} CS_ProjectionParameter;

typedef
[
    helpstring("Parameters for a geographic transformation into WGS84."),
    uuid(57B0C0DD-967D-11d3-8161-00C04F680FFF)
]
struct CS_WGS84ConversionInfo
{
    [helpstring("Bursa Wolf shift in meters.")]
    DOUBLE Dx;
    [helpstring("Bursa Wolf shift in meters.")]
    DOUBLE Dy;
    [helpstring("Bursa Wolf shift in meters.")]
    DOUBLE Dz;
    [helpstring("Bursa Wolf rotation in arc seconds.")]
    DOUBLE Ex;
    [helpstring("Bursa Wolf rotation in arc seconds.")]
    DOUBLE Ey;
    [helpstring("Bursa Wolf rotation in arc seconds.")]
    DOUBLE Ez;
    [helpstring("Bursa Wolf scaling in parts per million.")]
    DOUBLE Ppm;
    [helpstring("Human readable text describing intended region of
transformation.")]
    BSTR AreaOfUse;
} CS_WGS84ConversionInfo;

[
    helpstring("A base interface for metadata applicable to coordinate system
objects."),
    uuid(57B0C0E7-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_Info : IUnknown
```

```
{
    [propget,helpstring("Gets the name.")]
    HRESULT Name([out, retval] BSTR* val);
    [propget,helpstring("Gets the authority name.")]
    HRESULT Authority([out, retval] BSTR* val);
    [propget,helpstring("Gets the authority-specific identification code.")]
    HRESULT AuthorityCode([out, retval] BSTR* val);
    [propget,helpstring("Gets the alias.")]
    HRESULT Alias([out, retval] BSTR* val);
    [propget,helpstring("Gets the abbreviation.")]
    HRESULT Abbreviation([out, retval] BSTR* val);
    [propget,helpstring("Gets the provider-supplied remarks.")]
    HRESULT Remarks([out, retval] BSTR* val);
    [propget,helpstring("Gets a Well-Known text representation of this object.")]
    HRESULT WKT([out, retval] BSTR* val);
    [propget,helpstring("Gets an XML representation of this object.")]
    HRESULT XML([out, retval] BSTR* val);
};

[
    helpstring("Base interface for defining units."),
    uuid(57B0C0DF-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_Unit : CS_Info
{
};

[
    helpstring("Base interface for all coordinate systems."),
    uuid(57B0C0F4-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_CoordinateSystem : CS_Info
{
    [propget,helpstring("Dimension of the coordinate system.")]
    HRESULT Dimension([out, retval] LONG* val);
    [propget,helpstring("Gets axis details for dimension within coordinate
system.")]
    HRESULT Axis([in] LONG Dimension,[out, retval] CS_AxisInfo* val);
    [propget,helpstring("Gets units for dimension within coordinate system.")]
    HRESULT Units([in] LONG Dimension,[out, retval] CS_Unit** val);
    [propget,helpstring("Gets default envelope of coordinate system.")]
    HRESULT DefaultEnvelope([out, retval] PT_Envelope* val);
};

[
    helpstring("Creates spatial reference objects using codes"),
    uuid(57B0C0F3-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_CoordinateSystemAuthorityFactory : IUnknown
{
    [propget,helpstring("Returns the authority name.")]
    HRESULT Authority([out, retval] BSTR* val);
    [helpstring("Returns a ProjectedCoordinateSystem object from a code.")]
    HRESULT CreateProjectedCoordinateSystem([in] BSTR Code,[out, retval]
CS_ProjectedCoordinateSystem** val);
    [helpstring("Returns a GeographicCoordinateSystem object from a code.")]
    HRESULT CreateGeographicCoordinateSystem([in] BSTR Code,[out, retval]
CS_GeographicCoordinateSystem** val);
    [helpstring("Returns a HorizontalDatum object from a code.")]
    HRESULT CreateHorizontalDatum([in] BSTR Code,[out, retval]
CS_HorizontalDatum** val);
    [helpstring("Returns an Ellipsoid object from a code.")]
    HRESULT CreateEllipsoid([in] BSTR Code,[out, retval] CS_Ellipsoid** val);
    [helpstring("Returns a PrimeMeridian object from a code.")]
    HRESULT CreatePrimeMeridian([in] BSTR Code,[out, retval] CS_PrimeMeridian**
val);
    [helpstring("Returns a LinearUnit object from a code.")]
    HRESULT CreateLinearUnit([in] BSTR Code,[out, retval] CS_LinearUnit** val);
```

```
    [helpstring("Returns an AngularUnit object from a code.")]
    HRESULT CreateAngularUnit([in] BSTR Code,[out, retval] CS_AngularUnit** val);
    [helpstring("Creates a vertical datum from a code.")]
    HRESULT CreateVerticalDatum([in] BSTR Code,[out, retval] CS_VerticalDatum**
val);
    [helpstring("Create a vertical coordinate system from a code.")]
    HRESULT CreateVerticalCoordinateSystem([in] BSTR Code,[out, retval]
CS_VerticalCoordinateSystem** val);
    [helpstring("Creates a 3D coordinate system from a code.")]
    HRESULT CreateCompoundCoordinateSystem([in] BSTR Code,[out, retval]
CS_CompoundCoordinateSystem** val);
    [helpstring("Creates a horizontal co-ordinate system from a code.")]
    HRESULT CreateHorizontalCoordinateSystem([in] BSTR Code,[out, retval]
CS_HorizontalCoordinateSystem** val);
    [helpstring("Gets a description of the object corresponding to a code.")]
    HRESULT DescriptionText([in] BSTR Code,[out, retval] BSTR* val);
    [helpstring("Gets the Geoid code from a WKT name.")]
    HRESULT GeoidFromWKTName([in] BSTR Wkt,[out, retval] BSTR* val);
    [helpstring("Gets the WKT name of a Geoid.")]
    HRESULT WktGeoidName([in] BSTR Geoid,[out, retval] BSTR* val);
};

[
    helpstring("Definition of linear units."),
    uuid(57B0C0E6-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_LinearUnit : CS_Unit
{
    [propget,helpstring("Returns the number of meters per LinearUnit.")]
    HRESULT MetersPerUnit([out, retval] DOUBLE* val);
};

[
    helpstring("A set of quantities from which other quantities are calculated."),
    uuid(57B0C0FC-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_Datum : CS_Info
{
    [propget,helpstring("Gets the type of the datum as an enumerated code.")]
    HRESULT DatumType([out, retval] CS_DatumType* val);
};

[
    helpstring("An approximation of the Earth's surface as a squashed sphere."),
    uuid(57B0C0F1-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_Ellipsoid : CS_Info
{
    [propget,helpstring("Gets the equatorial radius.")]
    HRESULT SemiMajorAxis([out, retval] DOUBLE* val);
    [propget,helpstring("Gets the polar radius.")]
    HRESULT SemiMinorAxis([out, retval] DOUBLE* val);
    [propget,helpstring("Returns the value of the inverse of the flattening
constant.")]
    HRESULT InverseFlattening([out, retval] DOUBLE* val);
    [propget,helpstring("Returns the LinearUnit.")]
    HRESULT AxisUnit([out, retval] CS_LinearUnit** val);
    [helpstring("Is the Inverse Flattening definitive for this ellipsoid?")]
    HRESULT IsIvfDefinitive([out, retval] VARIANT_BOOL* val);
};

[
    helpstring("A coordinate system which sits inside another coordinate
system."),
    uuid(57B0C0F0-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_FittedCoordinateSystem : CS_CoordinateSystem
```

```
{
    [propget,helpstring("Gets underlying coordinate system.")]
    HRESULT BaseCoordinateSystem([out, retval] CS_CoordinateSystem** val);
    [propget,helpstring("Gets Well-Known Text of a math transform to the base
coordinate system.")]
    HRESULT ToBase([out, retval] BSTR* val);
};

[
    helpstring("Procedure used to measure positions on the surface of the
Earth."),
    uuid(57B0C0E8-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_HorizontalDatum : CS_Datum
{
    [propget,helpstring("Returns the Ellipsoid.")]
    HRESULT Ellipsoid([out, retval] CS_Ellipsoid** val);
    [propget,helpstring("Gets preferred parameters for a Bursa Wolf transformation
into WGS84.")]
    HRESULT WGS84Parameters([out, retval] CS_WGS84ConversionInfo* val);
};

[
    helpstring("A 2D coordinate system suitable for positions on the Earth's
surface."),
    uuid(57B0C0E9-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_HorizontalCoordinateSystem : CS_CoordinateSystem
{
    [propget,helpstring("Returns the HorizontalDatum.")]
    HRESULT HorizontalDatum([out, retval] CS_HorizontalDatum** val);
};

[
    helpstring("Definition of angular units."),
    uuid(57B0C0F8-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_AngularUnit : CS_Unit
{
    [propget,helpstring("Returns the number of radians per AngularUnit.")]
    HRESULT RadiansPerUnit([out, retval] DOUBLE* val);
};

[
    helpstring("Local datum."),
    uuid(57B0C0FD-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_LocalDatum : CS_Datum
{
};

[
    helpstring("An aggregate of two coordinate systems (CRS)."),
    uuid(57B0C0F5-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_CompoundCoordinateSystem : CS_CoordinateSystem
{
    [propget,helpstring("Gets first sub-coordinate system.")]
    HRESULT HeadCS([out, retval] CS_CoordinateSystem** val);
    [propget,helpstring("Gets second sub-coordinate system.")]
    HRESULT TailCS([out, retval] CS_CoordinateSystem** val);
};

[
    helpstring("A meridian used to take longitude measurements from."),
    uuid(57B0C0E2-967D-11d3-8161-00C04F680FFF),
```

```
    object
]
interface CS_PrimeMeridian : CS_Info
{
    [propget,helpstring("Returns the longitude value relative to the Greenwich
Meridian.")]
    HRESULT Longitude([out, retval] DOUBLE* val);
    [propget,helpstring("Returns the AngularUnits.")]
    HRESULT AngularUnit([out, retval] CS_AngularUnit** val);
};

[
    helpstring("A local coordinate system, with uncertain relationship to the
world."),
    uuid(57B0C0E5-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_LocalCoordinateSystem : CS_CoordinateSystem
{
    [propget,helpstring("Gets the local datum.")]
    HRESULT LocalDatum([out, retval] CS_LocalDatum** val);
};

[
    helpstring("A coordinate system based on latitude and longitude."),
    uuid(57B0C0EB-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_GeographicCoordinateSystem : CS_HorizontalCoordinateSystem
{
    [propget,helpstring("Returns the AngularUnit.")]
    HRESULT AngularUnit([out, retval] CS_AngularUnit** val);
    [propget,helpstring("Returns the PrimeMeridian.")]
    HRESULT PrimeMeridian([out, retval] CS_PrimeMeridian** val);
    [propget,helpstring("Gets the number of available conversions to WGS84
coordinates.")]
    HRESULT NumConversionToWGS84([out, retval] LONG* val);
    [propget,helpstring("Gets details on a conversion to WGS84.")]
    HRESULT WGS84ConversionInfo([in] LONG Index,[out, retval]
CS_WGS84ConversionInfo* val);
};

[
    helpstring("A projection from geographic coordinates to projected
coordinates."),
    uuid(57B0C101-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_Projection : CS_Info
{
    [propget,helpstring("Gets number of parameters of the projection.")]
    HRESULT NumParameters([out, retval] LONG* val);
    [propget,helpstring("Gets an indexed parameter of the projection.")]
    HRESULT Parameter([in] LONG Index,[out, retval] CS_ProjectionParameter* val);
    [propget,helpstring("Gets the projection classification name (e.g.
'Transverse_Mercator').")]
    HRESULT ClassName([out, retval] BSTR* val);
};

[
    helpstring("A 2D cartographic coordinate system."),
    uuid(57B0C0E1-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_ProjectedCoordinateSystem : CS_HorizontalCoordinateSystem
{
    [propget,helpstring("Returns the GeographicCoordinateSystem.")]
    HRESULT GeographicCoordinateSystem([out, retval]
CS_GeographicCoordinateSystem** val);
    [propget,helpstring("Returns the LinearUnits.")]
    HRESULT LinearUnit([out, retval] CS_LinearUnit** val);
```

```
    [propget,helpstring("Gets the projection.")]
    HRESULT Projection([out, retval] CS_Projection** val);
};

[
    helpstring("Procedure used to measure vertical distances."),
    uuid(57B0C0FA-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_VerticalDatum : CS_Datum
{
};

[
    helpstring("A 3D coordinate system, with its origin at the center of the
Earth."),
    uuid(57B0C0EF-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_GeocentricCoordinateSystem : CS_CoordinateSystem
{
    [propget,helpstring("Returns the HorizontalDatum.")]
    HRESULT HorizontalDatum([out, retval] CS_HorizontalDatum** val);
    [propget,helpstring("Gets the units used along all the axes.")]
    HRESULT LinearUnit([out, retval] CS_LinearUnit** val);
    [propget,helpstring("Returns the PrimeMeridian.")]
    HRESULT PrimeMeridian([out, retval] CS_PrimeMeridian** val);
};

[
    helpstring("A one-dimensional coordinate system suitable for vertical
measurements."),
    uuid(57B0C0DE-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_VerticalCoordinateSystem : CS_CoordinateSystem
{
    [propget,helpstring("Gets the vertical datum, which indicates the measurement
method.")]
    HRESULT VerticalDatum([out, retval] CS_VerticalDatum** val);
    [propget,helpstring("Gets the units used along the vertical axis.")]
    HRESULT VerticalUnit([out, retval] CS_LinearUnit** val);
};

[
    helpstring("Builds up complex objects from simpler objects or values."),
    uuid(57B0C0F2-967D-11d3-8161-00C04F680FFF),
    object
]
interface CS_CoordinateSystemFactory : IUnknown
{
    [helpstring("Creates a coordinate system object from an XML string.")]
    HRESULT CreateFromXML([in] BSTR Xml,[out, retval] CS_CoordinateSystem** val);
    [helpstring("Creates a coordinate system object from a Well-Known Text
string.")]
    HRESULT CreateFromWKT([in] BSTR WellKnownText,[out, retval]
CS_CoordinateSystem** val);
    [helpstring("Creates a compound coordinate system.")]
    HRESULT CreateCompoundCoordinateSystem([in] BSTR Name,[in]
CS_CoordinateSystem* Head,[in] CS_CoordinateSystem* Tail,[out, retval]
CS_CompoundCoordinateSystem** val);
    [helpstring("Creates a fitted coordinate system.")]
    HRESULT CreateFittedCoordinateSystem([in] BSTR Name,[in] CS_CoordinateSystem*
Base,[in] BSTR ToBaseWKT,[in] long Count, [in, size_is(Count)] CS_AxisInfo
*ArAxes,[out, retval] CS_FittedCoordinateSystem** val);
    [helpstring("Creates a local coordinate system.")]
    HRESULT CreateLocalCoordinateSystem([in] BSTR Name,[in] CS_LocalDatum*
Datum,[in] CS_Unit* Unit,[in] long Count, [in, size_is(Count)] CS_AxisInfo
*ArAxes,[out, retval] CS_LocalCoordinateSystem** val);
    [helpstring("Creates an ellipsoid from radius values.")]
```

```
    HRESULT CreateEllipsoid([in] BSTR Name,[in] DOUBLE SemiMajorAxis,[in] DOUBLE
SemiMinorAxis,[in] CS_LinearUnit* LinearUnit,[out, retval] CS_Ellipsoid** val);
    [helpstring("Creates an ellipsoid from an major radius, and inverse
flattening.")]
    HRESULT CreateFlattenedSphere([in] BSTR Name,[in] DOUBLE SemiMajorAxis,[in]
DOUBLE InverseFlattening,[in] CS_LinearUnit* LinearUnit,[out, retval]
CS_Ellipsoid** val);
    [helpstring("Creates a projected coordinate system using a projection
object.")]
    HRESULT CreateProjectedCoordinateSystem([in] BSTR Name,[in]
CS_GeographicCoordinateSystem* Gcs,[in] CS_Projection* Projection,[in]
CS_LinearUnit* LinearUnit,[in] CS_AxisInfo* Axis0,[in] CS_AxisInfo* Axis1,[out,
retval] CS_ProjectedCoordinateSystem** val);
    [helpstring("Creates a projection.")]
    HRESULT CreateProjection([in] BSTR Name,[in] BSTR WktProjectionClass,[in] long
Count, [in, size_is(Count)] CS_ProjectionParameter *Parameters,[out, retval]
CS_Projection** val);
    [helpstring("Creates horizontal datum from ellipsoid and Bursa-Wolf
parameters.")]
    HRESULT CreateHorizontalDatum([in] BSTR Name,[in] CS_DatumType
HorizontalDatumType,[in] CS_Ellipsoid* Ellipsoid,[in] CS_WGS84ConversionInfo*
ToWGS84,[out, retval] CS_HorizontalDatum** val);
    [helpstring("Creates a prime meridian, relative to Greenwich.")]
    HRESULT CreatePrimeMeridian([in] BSTR Name,[in] CS_AngularUnit*
AngularUnit,[in] DOUBLE Longitude,[out, retval] CS_PrimeMeridian** val);
    [helpstring("Creates a GCS, which could be Lat/Lon or Lon/Lat.")]
    HRESULT CreateGeographicCoordinateSystem([in] BSTR Name,[in] CS_AngularUnit*
AngularUnit,[in] CS_HorizontalDatum* HorizontalDatum,[in] CS_PrimeMeridian*
PrimeMeridian,[in] CS_AxisInfo* Axis0,[in] CS_AxisInfo* Axis1,[out, retval]
CS_GeographicCoordinateSystem** val);
    [helpstring("Creates a local datum.")]
    HRESULT CreateLocalDatum([in] BSTR Name,[in] CS_DatumType LocalDatumType,[out,
retval] CS_LocalDatum** val);
    [helpstring("Creates a vertical datum from an enumerated type value.")]
    HRESULT CreateVerticalDatum([in] BSTR Name,[in] CS_DatumType
VerticalDatumType,[out, retval] CS_VerticalDatum** val);
    [helpstring("Creates a vertical coordinate system from a datum and linear
units.")]
    HRESULT CreateVerticalCoordinateSystem([in] BSTR Name,[in] CS_VerticalDatum*
VerticalDatum,[in] CS_LinearUnit* VerticalUnit,[in] CS_AxisInfo* Axis,[out,
retval] CS_VerticalCoordinateSystem** val);
};

#ifdef _OGC_CS_TLB_
[
    version(1.0),
    uuid(57B0C0DC-967D-11d3-8161-00C04F680FFF),
    helpstring("OpenGIS OGC_CS Type Library")
]
library OGC_CS
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    importlib("OGC_PT.tlb");

    enum CS_AxisOrientationEnum;
    enum CS_DatumType;
    struct CS_AxisInfo;
    struct CS_ProjectionParameter;
    struct CS_WGS84ConversionInfo;
    interface CS_Info;
    interface CS_Unit;
    interface CS_CoordinateSystem;
    interface CS_CoordinateSystemAuthorityFactory;
    interface CS_LinearUnit;
    interface CS_Datum;
    interface CS_Ellipsoid;
    interface CS_FittedCoordinateSystem;
    interface CS_HorizontalDatum;
    interface CS_HorizontalCoordinateSystem;
    interface CS_AngularUnit;
```

```
        interface CS_LocalDatum;
        interface CS_CompoundCoordinateSystem;
        interface CS_PrimeMeridian;
        interface CS_LocalCoordinateSystem;
        interface CS_GeographicCoordinateSystem;
        interface CS_Projection;
        interface CS_ProjectedCoordinateSystem;
        interface CS_VerticalDatum;
        interface CS_GeocentricCoordinateSystem;
        interface CS_VerticalCoordinateSystem;
        interface CS_CoordinateSystemFactory;
    };
    #endif
```

13.1.3 OGC_CT.IDL

```
// CT package, COM profile.
// Copyright (c) OpenGIS Consortium Thursday, October 19, 2000.

import "ocidl.idl";
import "OGC_PT.idl";
import "OGC_CS.idl";

// Forward declarations.

enum CT_DomainFlags;
enum CT_TransformType;

struct CT_Parameter;

interface CT_MathTransform;
interface CT_CoordinateTransformationAuthorityFactory;
interface CT_CoordinateTransformationFactory;
interface CT_CoordinateTransformation;
interface CT_MathTransformFactory;

typedef
[
    helpstring("Flags indicating parts of domain covered by a convex hull."),
    uuid(459B806D-9912-11d3-8164-00C04F680FFF)
]
enum CT_DomainFlags
{
    CT_DF_Inside=1,
    CT_DF_Outside=2,
    CT_DF_Discontinuous=4
} CT_DomainFlags;

typedef
[
    helpstring("Semantic type of transform used in coordinate transformation."),
    uuid(459B806B-9912-11d3-8164-00C04F680FFF)
]
enum CT_TransformType
{
    CT_TT_Other=0,
    CT_TT_Conversion=1,
    CT_TT_Transformation=2,
    CT_TT_ConversionAndTransformation=3
} CT_TransformType;

typedef
[
    helpstring("A named parameter value."),
    uuid(459B806C-9912-11d3-8164-00C04F680FFF)
]
struct CT_Parameter
{
    [helpstring("The parameter name.")]
    BSTR Name;
    [helpstring("The parameter value.")]
    DOUBLE Value;
} CT_Parameter;

[
    helpstring("Transforms multi-dimensional coordinate points."),
    uuid(459B806F-9912-11d3-8164-00C04F680FFF),
    object
]
interface CT_MathTransform : IUnknown
{
    [propget,helpstring("Gets flags classifying domain points within a convex
hull.")]
```

```
    HRESULT DomainFlags([in] SAFEARRAY(DOUBLE)* Ord,[out, retval] CT_DomainFlags*
val);
    [propget,helpstring("Gets transformed convex hull.")]
    HRESULT CodomainConvexHull([in] SAFEARRAY(DOUBLE)* Ord,[out, retval]
SAFEARRAY(DOUBLE)* val);
    [propget,helpstring("Gets a Well-Known text representation of this object.")]
    HRESULT WKT([out, retval] BSTR* val);
    [propget,helpstring("Gets an XML representation of this object.")]
    HRESULT XML([out, retval] BSTR* val);
    [propget,helpstring("Gets the dimension of input points.")]
    HRESULT DimSource([out, retval] LONG* val);
    [propget,helpstring("Gets the dimension of output points.")]
    HRESULT DimTarget([out, retval] LONG* val);
    [helpstring("Transforms a coordinate point.")]
    HRESULT Transform([in] PT_CoordinatePoint* Cp,[out, retval]
PT_CoordinatePoint* val);
    [helpstring("Transforms a list of coordinate point ordinal values.")]
    HRESULT TransformList([in] SAFEARRAY(DOUBLE)* Ord,[out, retval]
SAFEARRAY(DOUBLE)* val);
    [helpstring("Gets the derivative of this transform at a point.")]
    HRESULT Derivative([in] PT_CoordinatePoint* Cp,[out, retval] PT_Matrix* val);
    [helpstring("Creates the inverse transform of this object.")]
    HRESULT Inverse([out, retval] CT_MathTransform** val);
    [helpstring("Tests whether this transform does not move any points.")]
    HRESULT IsIdentity([out, retval] VARIANT_BOOL* val);
};

[
    helpstring("Creates coordinate transformation objects from codes."),
    uuid(459B8069-9912-11d3-8164-00C04F680FFF),
    object
]
interface CT_CoordinateTransformationAuthorityFactory : IUnknown
{
    [propget,helpstring("The name of the authority.")]
    HRESULT Authority([out, retval] BSTR* val);
    [helpstring("Creates a transformation from a single transformation code.")]
    HRESULT CreateFromTransformationCode([in] BSTR Code,[out, retval]
CT_CoordinateTransformation** val);
    [helpstring("Creates a transformation from coordinate system codes.")]
    HRESULT CreateFromCoordinateSystemCodes([in] BSTR SourceCode,[in] BSTR
TargetCode,[out, retval] CT_CoordinateTransformation** val);
};

[
    helpstring("Creates coordinate transformations."),
    uuid(459B805E-9912-11d3-8164-00C04F680FFF),
    object
]
interface CT_CoordinateTransformationFactory : IUnknown
{
    [helpstring("Creates a transformation between two coordinate systems.")]
    HRESULT CreateFromCoordinateSystems([in] CS_CoordinateSystem* SourceCS,[in]
CS_CoordinateSystem* TargetCS,[out, retval] CT_CoordinateTransformation** val);
};

[
    helpstring("Describes a coordinate transformation."),
    uuid(459B805F-9912-11d3-8164-00C04F680FFF),
    object
]
interface CT_CoordinateTransformation : IUnknown
{
    [propget,helpstring("Name of transformation.")]
    HRESULT Name([out, retval] BSTR* val);
    [propget,helpstring("Authority which defined transformation and parameter
values.")]
    HRESULT Authority([out, retval] BSTR* val);
    [propget,helpstring("Code used by authority to identify transformation.")]
    HRESULT AuthorityCode([out, retval] BSTR* val);
    [propget,helpstring("Gets the provider-supplied remarks.")]
```

```
    HRESULT Remarks([out, retval] BSTR* val);
    [propget,helpstring("Human readable description of domain in source coordinate
system.")]
    HRESULT AreaOfUse([out, retval] BSTR* val);
    [propget,helpstring("Semantic type of transform.")]
    HRESULT TransformType([out, retval] CT_TransformType* val);
    [propget,helpstring("Source coordinate system.")]
    HRESULT SourceCS([out, retval] CS_CoordinateSystem** val);
    [propget,helpstring("Target coordinate system.")]
    HRESULT TargetCS([out, retval] CS_CoordinateSystem** val);
    [propget,helpstring("Gets math transform.")]
    HRESULT MathTransform([out, retval] CT_MathTransform** val);
};

[
    helpstring("Creates math transforms."),
    uuid(459B806E-9912-11d3-8164-00C04F680FFF),
    object
]
interface CT_MathTransformFactory : IUnknown
{
    [helpstring("Creates an affine transform from a matrix.")]
    HRESULT CreateAffineTransform([in] PT_Matrix* Matrix,[out, retval]
CT_MathTransform** val);
    [helpstring("Creates a transform by concatenating two existing transforms.")]
    HRESULT CreateConcatenatedTransform([in] CT_MathTransform* Transform1,[in]
CT_MathTransform* Transform2,[out, retval] CT_MathTransform** val);
    [helpstring("Creates a transform which passes through a subset of ordinates to
another transform.")]
    HRESULT CreatePassThroughTransform([in] LONG FirstAffectedOrdinate,[in]
CT_MathTransform* SubTransform,[out, retval] CT_MathTransform** val);
    [helpstring("Creates a transform from a classification name and parameters.")]
    HRESULT CreateParameterizedTransform([in] BSTR Classification,[in] long Count,
[in, size_is(Count)] CT_Parameter *Parameters,[out, retval] CT_MathTransform**
val);
    [helpstring("Creates a math transform from a Well-Known Text string.")]
    HRESULT CreateFromWKT([in] BSTR WellKnownText,[out, retval] CT_MathTransform**
val);
    [helpstring("Creates a math transform from XML.")]
    HRESULT CreateFromXML([in] BSTR Xml,[out, retval] CT_MathTransform** val);
    [helpstring("Tests whether parameter is angular.")]
    HRESULT IsParameterAngular([in] BSTR ParameterName,[out, retval] VARIANT_BOOL*
val);
    [helpstring("Tests whether parameter is linear.")]
    HRESULT IsParameterLinear([in] BSTR ParameterName,[out, retval] VARIANT_BOOL*
val);
};

#ifdef _OGC_CT_TLB_
[
    version(1.0),
    uuid(459B805B-9912-11d3-8164-00C04F680FFF),
    helpstring("OpenGIS OGC_CT Type Library")
]
library OGC_CT
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    importlib("OGC_PT.tlb");
    importlib("OGC_CS.tlb");

    enum CT_DomainFlags;
    enum CT_TransformType;
    struct CT_Parameter;
    interface CT_MathTransform;
    interface CT_CoordinateTransformationAuthorityFactory;
    interface CT_CoordinateTransformationFactory;
    interface CT_CoordinateTransformation;
    interface CT_MathTransformFactory;
};
#endif
```

13.2 CORBA

13.2.1 OGC_PT.IDL

```
// PT package, CORBA profile.
// Copyright (c) OpenGIS Consortium Thursday, October 19, 2000.

module pt {
    typedef sequence<double> DoubleSeq;
    typedef sequence<DoubleSeq> DoubleSeqSeq;
    struct PT_CoordinatePoint {
        DoubleSeq ord;
    };
    struct PT_Envelope {
        PT_CoordinatePoint minCP;
        PT_CoordinatePoint maxCP;
    };
    struct PT_Matrix {
        DoubleSeqSeq elt;
    };
};
```

13.2.2 OGC_CS.IDL

```
// CS package, CORBA profile.
// Copyright (c) OpenGIS Consortium Thursday, October 19, 2000.

#include "ogc_pt.idl"

module cs {
    interface CS_Info;
    interface CS_CoordinateSystem;
    interface CS_CompoundCoordinateSystem;
    interface CS_CoordinateSystemAuthorityFactory;
    interface CS_Unit;
    interface CS_Datum;
    interface CS_LinearUnit;
    interface CS_FittedCoordinateSystem;
    interface CS_Ellipsoid;
    interface CS_HorizontalDatum;
    interface CS_HorizontalCoordinateSystem;
    interface CS_AngularUnit;
    interface CS_LocalDatum;
    interface CS_PrimeMeridian;
    interface CS_LocalCoordinateSystem;
    interface CS_GeographicCoordinateSystem;
    interface CS_Projection;
    interface CS_ProjectedCoordinateSystem;
    interface CS_VerticalDatum;
    interface CS_GeocentricCoordinateSystem;
    interface CS_VerticalCoordinateSystem;
    interface CS_CoordinateSystemFactory;

    typedef long CS_AxisOrientationEnum;
    const CS_AxisOrientationEnum CS_AO_Other=0;
    const CS_AxisOrientationEnum CS_AO_North=1;
    const CS_AxisOrientationEnum CS_AO_South=2;
    const CS_AxisOrientationEnum CS_AO_East=3;
    const CS_AxisOrientationEnum CS_AO_West=4;
    const CS_AxisOrientationEnum CS_AO_Up=5;
    const CS_AxisOrientationEnum CS_AO_Down=6;

    typedef long CS_DatumType;
    const CS_DatumType CS_HD_Min=1000;
    const CS_DatumType CS_HD_Other=1000;
    const CS_DatumType CS_HD_Classic=1001;
    const CS_DatumType CS_HD_Geocentric=1002;
    const CS_DatumType CS_HD_Max=1999;
    const CS_DatumType CS_VD_Min=2000;
    const CS_DatumType CS_VD_Other=2000;
    const CS_DatumType CS_VD_Orthometric=2001;
    const CS_DatumType CS_VD_Ellipsoidal=2002;
    const CS_DatumType CS_VD_AltitudeBarometric=2003;
    const CS_DatumType CS_VD_Normal=2004;
    const CS_DatumType CS_VD_GeoidModelDerived=2005;
    const CS_DatumType CS_VD_Depth=2006;
    const CS_DatumType CS_VD_Max=2999;
    const CS_DatumType CS_LD_Min=10000;
    const CS_DatumType CS_LD_Max=32767;

    struct CS_AxisInfo {
        CS_AxisOrientationEnum orientation;
        string name;
    };
    struct CS_ProjectionParameter {
        string name;
        double value;
    };
    struct CS_WGS84ConversionInfo {
        double dx;
        double dy;
```

```
            double dz;
            double ex;
            double ey;
            double ez;
            double ppm;
            string areaOfUse;
        };
        typedef sequence<CS_AxisInfo> CS_AxisInfoSeq;
        typedef sequence<CS_ProjectionParameter> CS_ProjectionParameterSeq;
        interface CS_Info
        {
            readonly attribute string name;
            readonly attribute string authority;
            readonly attribute string authorityCode;
            readonly attribute string alias;
            readonly attribute string abbreviation;
            readonly attribute string remarks;
            readonly attribute string WKT;
            readonly attribute string XML;
        };
        interface CS_CoordinateSystem : CS_Info
        {
            readonly attribute long dimension;
            readonly attribute pt::PT_Envelope defaultEnvelope;
            CS_AxisInfo getAxis(in long dimension);
            CS_Unit getUnits(in long dimension);
        };
        interface CS_CompoundCoordinateSystem : CS_CoordinateSystem
        {
            readonly attribute CS_CoordinateSystem headCS;
            readonly attribute CS_CoordinateSystem tailCS;
        };
        interface CS_CoordinateSystemAuthorityFactory
        {
            readonly attribute string authority;
            CS_ProjectedCoordinateSystem createProjectedCoordinateSystem(in string
    code);
            CS_GeographicCoordinateSystem createGeographicCoordinateSystem(in string
    code);
            CS_HorizontalDatum createHorizontalDatum(in string code);
            CS_Ellipsoid createEllipsoid(in string code);
            CS_PrimeMeridian createPrimeMeridian(in string code);
            CS_LinearUnit createLinearUnit(in string code);
            CS_AngularUnit createAngularUnit(in string code);
            CS_VerticalDatum createVerticalDatum(in string code);
            CS_VerticalCoordinateSystem createVerticalCoordinateSystem(in string
    code);
            CS_CompoundCoordinateSystem createCompoundCoordinateSystem(in string
    code);
            CS_HorizontalCoordinateSystem createHorizontalCoordinateSystem(in string
    code);
            string descriptionText(in string code);
            string geoidFromWKTName(in string wkt);
            string wktGeoidName(in string geoid);
        };
        interface CS_Unit : CS_Info
        {
        };
        interface CS_Datum : CS_Info
        {
            readonly attribute CS_DatumType datumType;
        };
        interface CS_LinearUnit : CS_Unit
        {
            readonly attribute double metersPerUnit;
        };
        interface CS_FittedCoordinateSystem : CS_CoordinateSystem
        {
            readonly attribute CS_CoordinateSystem baseCoordinateSystem;
            readonly attribute string toBase;
        };
```

```
    interface CS_Ellipsoid : CS_Info
    {
        readonly attribute double semiMajorAxis;
        readonly attribute double semiMinorAxis;
        readonly attribute double inverseFlattening;
        readonly attribute boolean ivfDefinitive;
        readonly attribute CS_LinearUnit axisUnit;
    };
    interface CS_HorizontalDatum : CS_Datum
    {
        readonly attribute CS_Ellipsoid ellipsoid;
        readonly attribute CS_WGS84ConversionInfo WGS84Parameters;
    };
    interface CS_HorizontalCoordinateSystem : CS_CoordinateSystem
    {
        readonly attribute CS_HorizontalDatum horizontalDatum;
    };
    interface CS_AngularUnit : CS_Unit
    {
        readonly attribute double radiansPerUnit;
    };
    interface CS_LocalDatum : CS_Datum
    {
    };
    interface CS_PrimeMeridian : CS_Info
    {
        readonly attribute double longitude;
        readonly attribute CS_AngularUnit angularUnit;
    };
    interface CS_LocalCoordinateSystem : CS_CoordinateSystem
    {
        readonly attribute CS_LocalDatum localDatum;
    };
    interface CS_GeographicCoordinateSystem : CS_HorizontalCoordinateSystem
    {
        readonly attribute CS_AngularUnit angularUnit;
        readonly attribute CS_PrimeMeridian primeMeridian;
        readonly attribute long numConversionToWGS84;
        CS_WGS84ConversionInfo getWGS84ConversionInfo(in long index);
    };
    interface CS_Projection : CS_Info
    {
        readonly attribute long numParameters;
        readonly attribute string className;
        CS_ProjectionParameter getParameter(in long index);
    };
    interface CS_ProjectedCoordinateSystem : CS_HorizontalCoordinateSystem
    {
        readonly attribute CS_GeographicCoordinateSystem
geographicCoordinateSystem;
        readonly attribute CS_LinearUnit linearUnit;
        readonly attribute CS_Projection projection;
    };
    interface CS_VerticalDatum : CS_Datum
    {
    };
    interface CS_GeocentricCoordinateSystem : CS_CoordinateSystem
    {
        readonly attribute CS_HorizontalDatum horizontalDatum;
        readonly attribute CS_LinearUnit linearUnit;
        readonly attribute CS_PrimeMeridian primeMeridian;
    };
    interface CS_VerticalCoordinateSystem : CS_CoordinateSystem
    {
        readonly attribute CS_VerticalDatum verticalDatum;
        readonly attribute CS_LinearUnit verticalUnit;
    };
    interface CS_CoordinateSystemFactory
    {
        CS_CoordinateSystem createFromXML(in string xml);
        CS_CoordinateSystem createFromWKT(in string wellKnownText);
```

```
        CS_CompoundCoordinateSystem createCompoundCoordinateSystem(in string
name,in CS_CoordinateSystem head,in CS_CoordinateSystem tail);
        CS_FittedCoordinateSystem createFittedCoordinateSystem(in string name,in
CS_CoordinateSystem base,in string toBaseWKT,in CS_AxisInfoSeq arAxes);
        CS_LocalCoordinateSystem createLocalCoordinateSystem(in string name,in
CS_LocalDatum datum,in CS_Unit unit,in CS_AxisInfoSeq arAxes);
        CS_Ellipsoid createEllipsoid(in string name,in double semiMajorAxis,in
double semiMinorAxis,in CS_LinearUnit linearUnit);
        CS_Ellipsoid createFlattenedSphere(in string name,in double
semiMajorAxis,in double inverseFlattening,in CS_LinearUnit linearUnit);
        CS_ProjectedCoordinateSystem createProjectedCoordinateSystem(in string
name,in CS_GeographicCoordinateSystem gcs,in CS_Projection projection,in
CS_LinearUnit linearUnit,in CS_AxisInfo axis0,in CS_AxisInfo axis1);
        CS_Projection createProjection(in string name,in string
wktProjectionClass,in CS_ProjectionParameterSeq parameters);
        CS_HorizontalDatum createHorizontalDatum(in string name,in CS_DatumType
horizontalDatumType,in CS_Ellipsoid ellipsoid,in CS_WGS84ConversionInfo toWGS84);
        CS_PrimeMeridian createPrimeMeridian(in string name,in CS_AngularUnit
angularUnit,in double longitude);
        CS_GeographicCoordinateSystem createGeographicCoordinateSystem(in string
name,in CS_AngularUnit angularUnit,in CS_HorizontalDatum horizontalDatum,in
CS_PrimeMeridian primeMeridian,in CS_AxisInfo axis0,in CS_AxisInfo axis1);
        CS_LocalDatum createLocalDatum(in string name,in CS_DatumType
localDatumType);
        CS_VerticalDatum createVerticalDatum(in string name,in CS_DatumType
verticalDatumType);
        CS_VerticalCoordinateSystem createVerticalCoordinateSystem(in string
name,in CS_VerticalDatum verticalDatum,in CS_LinearUnit verticalUnit,in
CS_AxisInfo axis);
    };
};
```

13.2.3 OGC_CT.IDL

```
// CT package, CORBA profile.
// Copyright (c) OpenGIS Consortium Thursday, October 19, 2000.

#include "ogc_pt.idl"
#include "ogc_cs.idl"

module ct {
    interface CT_MathTransform;
    interface CT_CoordinateTransformationAuthorityFactory;
    interface CT_CoordinateTransformationFactory;
    interface CT_CoordinateTransformation;
    interface CT_MathTransformFactory;

    typedef long CT_DomainFlags;
    const CT_DomainFlags CT_DF_Inside=1;
    const CT_DomainFlags CT_DF_Outside=2;
    const CT_DomainFlags CT_DF_Discontinuous=4;

    typedef long CT_TransformType;
    const CT_TransformType CT_TT_Other=0;
    const CT_TransformType CT_TT_Conversion=1;
    const CT_TransformType CT_TT_Transformation=2;
    const CT_TransformType CT_TT_ConversionAndTransformation=3;

    struct CT_Parameter {
        string name;
        double value;
    };
    typedef sequence<double> DoubleSeq;
    typedef sequence<CT_Parameter> CT_ParameterSeq;
    interface CT_MathTransform
    {
        readonly attribute string WKT;
        readonly attribute string XML;
        readonly attribute long dimSource;
        readonly attribute long dimTarget;
        readonly attribute boolean identity;
        CT_DomainFlags getDomainFlags(in DoubleSeq ord);
        DoubleSeq getCodomainConvexHull(in DoubleSeq ord);
        pt::PT_CoordinatePoint transform(in pt::PT_CoordinatePoint cp);
        DoubleSeq transformList(in DoubleSeq ord);
        pt::PT_Matrix derivative(in pt::PT_CoordinatePoint cp);
        CT_MathTransform inverse();
    };
    interface CT_CoordinateTransformationAuthorityFactory
    {
        readonly attribute string authority;
        CT_CoordinateTransformation createFromTransformationCode(in string code);
        CT_CoordinateTransformation createFromCoordinateSystemCodes(in string
sourceCode,in string targetCode);
    };
    interface CT_CoordinateTransformationFactory
    {
        CT_CoordinateTransformation createFromCoordinateSystems(in
cs::CS_CoordinateSystem sourceCS,in cs::CS_CoordinateSystem targetCS);
    };
    interface CT_CoordinateTransformation
    {
        readonly attribute string name;
        readonly attribute string authority;
        readonly attribute string authorityCode;
        readonly attribute string remarks;
        readonly attribute string areaOfUse;
        readonly attribute CT_TransformType transformType;
        readonly attribute cs::CS_CoordinateSystem sourceCS;
        readonly attribute cs::CS_CoordinateSystem targetCS;
        readonly attribute CT_MathTransform mathTransform;
```

```
    };
    interface CT_MathTransformFactory
    {
        CT_MathTransform createAffineTransform(in pt::PT_Matrix matrix);
        CT_MathTransform createConcatenatedTransform(in CT_MathTransform
transform1,in CT_MathTransform transform2);
        CT_MathTransform createPassThroughTransform(in long
firstAffectedOrdinate,in CT_MathTransform subTransform);
        CT_MathTransform createParameterizedTransform(in string classification,in
CT_ParameterSeq parameters);
        CT_MathTransform createFromWKT(in string wellKnownText);
        CT_MathTransform createFromXML(in string xml);
        boolean isParameterAngular(in string parameterName);
        boolean isParameterLinear(in string parameterName);
    };
};
```

13.3 Java

As there are so many Java files (one for each interface as opposed to one for each package), the Java listings have not been included here. Please see the attached Java source files and JavaDoc HTML documentation.

14 Appendix B: Guidelines for development of Conformance Test

The guidelines are listed below as sub-headings:

14.1 Ask vendors which profiles will be implemented

Conformance tests may need to be developed COM, CORBA and Java. In order to develop conformance tests, it would help to know which of these profiles will really be implemented.

14.2 Ask TC members to list possible errors

The technical purpose of the conformance tests is to give vendors and users a degree of confidence that implementations will successfully inter-operate. One approach therefore is to ask the OGC Technical Committee to collate a list of things that might go wrong, and then we can design meaningful tests.

For example:

a) Projection parameters using non-standard units
b) Coordinate systems using non-standard orientation of axes
c) Geographic coordinate systems using grads instead of degrees
d) Coordinate systems not using Greenwich prime meridian
e) Ellipsoid and geoid heights being mixed up

14.3 List core mandatory transforms and projections

One of the strengths of the Coordinate Transformation Profile is that the list of parameterized transforms (e.g. cartographic projections) is left open. However, we should agree a set of core mandatory transforms, and nail down their parameters. Then conformance-testing scripts can ask for operations to be performed with these known transforms, and the results can be checked for correctness.

The suggested core set of parameterized transforms are the following:

a) Lambert_Conformal_Conic_1SP
b) Lambert_Conformal_Conic_2SP
c) Transverse_Mercator
d) Affine
e) Abridged_Molodenski
f) Geocentric_To_Ellipsoid
g) Ellipsoid_To_Geocentric
h) Longitude_Rotation

14.4 List coordinate systems in WKT and XML

OGC could keep a list of all the EPSG projections in both WKT and XML formats.

This could probably not be a definitive list of WKT or XML coordinate systems, since EPSG may periodically adjust their parameters, and different implementations may validly differ (e.g. change number of decimal places). However, this exercise could help vendors flush out some bugs. It would also help to clarify the WKT and XML specifications.

14.5 Acquire reference implementation of projection code

If OGC had a library of reference code for performing projections (and other transforms), then vendors would probably be happy to use this for their internal testing. By having an agreed reference implementation, all vendors can ensure that they get the same (or similar) answers when transforming coordinates.

Preferably, the reference implementation should be provided by a non-vendor organization (e.g. a national mapping agency), and should be available in source code.

14.6 Prepare list of known points in different coordinate systems

OGC could maintain a database of known points in different coordinate systems. For example the location of the Eiffel tower could be recorded in WGS84 (Lat,Lon,Hgt), and then the same point in a variety of coordinate systems, with changes made by permuting the following:

a) Projection (e.g. TM versus LCC)
b) Prime Meridian (e.g. Greenwich versus Paris)
c) Horizontal datum (e.g. ED50 versus NTF)
d) Angular units (e.g. grads versus degrees)

This should be repeated for many points around the Earth.

15 Appendix C: Informative

15.1 XML

Some of the interfaces specified in this document have methods for returning the XML definition of the object. In addition, some of the 'factory' interfaces have methods for creating a new object from an XML definition. These methods have been left in the interfaces in anticipation of a future 'normative' specification. It is recommended that implementations of this version of the specification use this XML definition.

For all entities that can be printed in WKT format, there is a corresponding encoding in XML format.

The benefit of WKT is that it already exists as part of the Simple Features specification, it has been proved to work in practice, it has already been released in commercial products, and OGC retains control over its development.

The benefit of XML is that there are widely available software tools for creating, parsing and processing it. OGC does not have so much control over its development.

Both formats will be mandatory for all entity interfaces that have exporting methods using these formats, and all factory interfaces that have creation methods using these formats.

15.1.1 XML Definition

```
<!DOCTYPE CT_MathTransform [
    <!ELEMENT CT_MathTransform (
        CT_ConcatenatedTransform |
        CT_InverseTransform |
        CT_ParameterizedMathTransform |
        CT_PassThroughTransform) >

    <!ELEMENT CT_ParameterizedMathTransform (CT_Parameter*)>
    <!ATTLIST CT_ParameterizedMathTransform
        ClassName           CDATA    #REQUIRED
        >

    <!ELEMENT CT_PassThroughTransform (CT_MathTransform)>
    <!ATTLIST CT_PassThroughTransform
        FirstAffectedOrdinate   CDATA    #REQUIRED
        >

    <!ELEMENT CT_ConcatenatedTransform (CT_MathTransform*)>
    <!ELEMENT CT_InverseTransform (CT_MathTransform)>

    <!ELEMENT CT_Parameter EMPTY>
    <!ATTLIST CT_Parameter
        Name                CDATA    #REQUIRED
        Value               CDATA    #REQUIRED
        >
]>

<!DOCTYPE CS_CoordinateSystem [
    <!ELEMENT CS_CoordinateSystem (
        CS_CompoundCoordinateSystem |
        CS_FittedCoordinateSystem |
        CS_GeocentricCoordinateSystem |
        CS_GeographicCoordinateSystem |
        CS_ProjectedCoordinateSystem |
```

```
    CS_LocalCoordinateSystem |
    CS_VerticalCoordinateSystem) >
<!ATTLIST CS_CoordinateSystem
    Dimension           CDATA    #REQUIRED
    >

<!ELEMENT CS_Info EMPTY>
<!ATTLIST CS_Info
    AuthorityCode       CDATA    #IMPLIED
    Abbreviation        CDATA    #IMPLIED
    Alias               CDATA    #IMPLIED
    Authority           CDATA    #IMPLIED
    Name                CDATA    #IMPLIED
>

<!ELEMENT CS_AxisInfo EMPTY>
<!ATTLIST CS_AxisInfo
    Name                CDATA    #REQUIRED
    Orientation         CDATA    #REQUIRED
    >

<!ELEMENT CS_HorizontalDatum (CS_Info, CS_Ellipsoid, CS_WGS84ConversionInfo) >
<!ATTLIST CS_HorizontalDatum
    DatumType           CDATA    #REQUIRED
    >

<!ELEMENT CS_Ellipsoid (CS_Info, CS_LinearUnit) >
<!ATTLIST CS_Ellipsoid
    SemiMajorAxis       CDATA    #REQUIRED
    SemiMinorAxis       CDATA    #REQUIRED
    InverseFlattening   CDATA    #REQUIRED
    IvfDefinitive       CDATA    #REQUIRED
    >

<!ELEMENT CS_LinearUnit (CS_Info) >
<!ATTLIST CS_LinearUnit
    MetersPerUnit       CDATA    #REQUIRED
    >

<!ELEMENT CS_AngularUnit (CS_Info) >
<!ATTLIST CS_AngularUnit
    RadiansPerUnit      CDATA    #REQUIRED
    >

<!ELEMENT CS_WGS84ConversionInfo EMPTY>
<!ATTLIST CS_WGS84ConversionInfo
    Dx                  CDATA    #REQUIRED
    Dy                  CDATA    #REQUIRED
    Dz                  CDATA    #REQUIRED
    Ex                  CDATA    #REQUIRED
    Ey                  CDATA    #REQUIRED
    Ez                  CDATA    #REQUIRED
    Ppm                 CDATA    #REQUIRED
    >

<!ELEMENT CS_PrimeMeridian (CS_Info, CS_AngularUnit) >
<!ATTLIST CS_PrimeMeridian
    Longitude           CDATA    #REQUIRED
    >

<!ELEMENT CS_ProjectionParameter EMPTY >
<!ATTLIST CS_ProjectionParameter
    Name                CDATA    #REQUIRED
    Value               CDATA    #REQUIRED
    >

<!ELEMENT CS_Projection (CS_Info, CS_ProjectionParameter*) >
<!ATTLIST CS_Projection
    ClassName           CDATA    #REQUIRED
    >
```

```
    <!ELEMENT CS_VerticalDatum (CS_Info) >
    <!ATTLIST CS_VerticalDatum
        DatumType           CDATA    #REQUIRED
        >

    <!ELEMENT CS_LocalDatum (CS_Info) >
    <!ATTLIST CS_LocalDatum
        DatumType           CDATA    #REQUIRED
        >

    <!ELEMENT CS_FittedCoordinateSystem (CS_Info, CS_AxisInfo*, CS_CoordinateSystem)>
    <!ATTLIST CS_FittedCoordinateSystem
        ToBase              CDATA    #REQUIRED
        >

    <!ELEMENT CS_GeographicCoordinateSystem (CS_Info, CS_AxisInfo*, CS_HorizontalDatum,
        CS_AngularUnit, CS_PrimeMeridian)>
    <!ELEMENT CS_ProjectedCoordinateSystem (CS_Info, CS_AxisInfo*,
        CS_GeographicCoordinateSystem, CS_LinearUnit, CS_Projection)>
    <!ELEMENT CS_GeocentricCoordinateSystem (CS_Info, CS_AxisInfo*, CS_HorizontalDatum,
        CS_LinearUnit, CS_PrimeMeridian)>
    <!ELEMENT CS_VerticalCoordinateSystem (CS_Info, CS_AxisInfo*, CS_VerticalDatum,
        CS_LinearUnit)>
    <!ELEMENT CS_CompoundCoordinateSystem (CS_Info, CS_AxisInfo*, CS_CoordinateSystem,
        CS_CoordinateSystem)>
    <!ELEMENT CS_LocalCoordinateSystem (CS_Info, CS_AxisInfo*, CS_LocalDatum,
        (CS_AngularUnit | CS_LinearUnit))>
]>
```

15.1.2 XML Example

The following example shows a 3D compound coordinate system, which is made by combining a projected coordinate system and a vertical coordinate system. This is the same coordinate system as used for the WKT Example.

```
<CS_CoordinateSystem Dimension="3">
    <CS_CompoundCoordinateSystem>
        <CS_Info AuthorityCode="7405" Abbreviation="GB NatGrid + ODN" Authority="EPSG"
               Name="OSGB36 / British National Grid + ODN"/>
        <CS_AxisInfo Name="E" Orientation="EAST"/>
        <CS_AxisInfo Name="N" Orientation="NORTH"/>
        <CS_AxisInfo Name="Up" Orientation="UP"/>
        <CS_CoordinateSystem Dimension="2">
            <CS_ProjectedCoordinateSystem>
                <CS_Info AuthorityCode="27700" Abbreviation="British National Grid"
                      Authority="EPSG" Name="OSGB 1936 / British National Grid"/>
                <CS_AxisInfo Name="E" Orientation="EAST"/>
                <CS_AxisInfo Name="N" Orientation="NORTH"/>
                <CS_GeographicCoordinateSystem>
                    <CS_Info AuthorityCode="4277" Authority="EPSG" Name="OSGB 1936"/>
                    <CS_AxisInfo Name="Lat" Orientation="NORTH"/>
                    <CS_AxisInfo Name="Long" Orientation="EAST"/>
                    <CS_HorizontalDatum DatumType="1001">
                        <CS_Info AuthorityCode="6277" Authority="EPSG" Name="OSGB_1936"/>
                        <CS_Ellipsoid SemiMajorAxis="6377563.396"
                              SemiMinorAxis="6356256.90923729"
                              InverseFlattening="299.3249646" IvfDefinitive="1">
                            <CS_Info AuthorityCode="7001"
                                     Authority="EPSG" Name="Airy 1830"/>
                            <CS_LinearUnit MetersPerUnit="1">
                                <CS_Info AuthorityCode="9001"
                                      Abbreviation="m" Authority="EPSG"
                                      Name="metre"/>
                            </CS_LinearUnit>
                        </CS_Ellipsoid>
                        <CS_WGS84ConversionInfo Dx="375" Dy="-111" Dz="431" Ex="0" Ey="0"
                              Ez="0" Ppm="0"/>
                    </CS_HorizontalDatum>
                    <CS_AngularUnit RadiansPerUnit="1.74532925199433E-02">
                        <CS_Info AuthorityCode="9108" Authority="EPSG" Name="DMSH"/>
                    </CS_AngularUnit>
                    <CS_PrimeMeridian Longitude="0">
                        <CS_Info AuthorityCode="8901" Authority="EPSG" Name="Greenwich"/>
                        <CS_AngularUnit RadiansPerUnit="1.74532925199433E-02">
                            <CS_Info AuthorityCode="9110"
                              Authority="EPSG" Name="DDD.MMSSsss"/>
                        </CS_AngularUnit>
                    </CS_PrimeMeridian>
                </CS_GeographicCoordinateSystem>
                <CS_LinearUnit MetersPerUnit="1">
                    <CS_Info AuthorityCode="9001" Abbreviation="m"
                       Authority="EPSG" Name="metre"/>
                </CS_LinearUnit>
                <CS_Projection ClassName="Transverse_Mercator">
                    <CS_Info/>
                    <CS_ProjectionParameter Name="latitude_of_origin" Value="49"/>
                    <CS_ProjectionParameter Name="central_meridian" Value="-2"/>
                    <CS_ProjectionParameter Name="scale_factor" Value="0.999601272"/>
                    <CS_ProjectionParameter Name="false_easting" Value="400000"/>
                    <CS_ProjectionParameter Name="false_northing" Value="-100000"/>
                </CS_Projection>
            </CS_ProjectedCoordinateSystem>
        </CS_CoordinateSystem>
        <CS_CoordinateSystem Dimension="1">
            <CS_VerticalCoordinateSystem>
                <CS_Info AuthorityCode="5701"
                      Abbreviation="ODN"
                      Authority="EPSG" Name="Newlyn"/>
```

```
                <CS_AxisInfo Name="Up" Orientation="UP"/>
                <CS_VerticalDatum DatumType="2005">
                    <CS_Info AuthorityCode="5101" Abbreviation="ODN" Authority="EPSG"
                      Name="Ordnance Datum Newlyn"/>
                </CS_VerticalDatum>
                <CS_LinearUnit MetersPerUnit="1">
                    <CS_Info AuthorityCode="9001"
                      Abbreviation="m"
                      Authority="EPSG" Name="metre"/>
                </CS_LinearUnit>
            </CS_VerticalCoordinateSystem>
        </CS_CoordinateSystem>
    </CS_CompoundCoordinateSystem>
</CS_CoordinateSystem>
```