Open GIS Consortium Inc.

Date: 2004-05-14

Reference number of this OpenGIS® project document: **OGC 03-064r5**

Version: 0.5.0

Category: OpenGIS® Implementation Specification

Editor: Eric Bertel, Polexis, Inc. / OGC

GO-1 Application Objects

Document type: OpenGIS® Recommendation Paper
Document stage: Final
Document language: English

Contents

i. Preface

This document is the Open GIS Consortium Application Objects Implementation Specification. This specification is a result of the OGC Geographic Objects Initiative, which was established to develop an open set of common, lightweight, language-independent abstractions for describing, managing, rendering, and manipulating geometric and geographic objects within an application programming environment. This document defines that set of vendor-neutral, object-oriented geometric and geographic object abstractions for the application space. It provides both an abstract object specification (in UML) and a programming-language specific profile (in Java) to that specification. The language-specific bindings serve as an open Application Program Interface (API).

ii. Submitting organizations

The following organisations submitted this Implementation Specification to the Open GIS Consortium Inc. in response to the OGC Call for Participation (CFP) in the Geographic Objects Phase One (GO-1) Initiative:

a) Polexis

b) Northrop Grumman Information Technology

c) Pennsylvania State University

iii. Submission contact points

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

CONTACT	COMPANY	ADDRESS	PHONE/FAX	EMAIL
Eric Bertel	Polexis			eric@polexis.com
Greg Reynolds	Polexis			greynolds@polexis.com

CONTACT	COMPANY	ADDRESS	PHONE/FAX	EMAIL
				xis.com
John Davidson	Image Matters LLC/OGC			johnd@imagemattersllc.com
Phillip C. Dibner	Ecosystem Associates/OGC			pcd@ecosystem.com
Charles Heazel	OGC			cheazel@opengis.org
Ava Mann	Northrop Grumman IT			amann@northropgrumman.com
James MacGill	Penn. State Univ.			jmacgill@psu.edu

iv. Revision history

Date	Release	Author	Paragraph modified	Description
19 May 03	0.1.9	P. Dibner		First public draft
03 June 03	0.2.0	P. Dibner		Cleanup for June 2003 TC
17 Sept 03	0.3.0	E. Bertel		Second public draft
11 Mar04	0.3.5	E. Bertel	Added 6.3.4.5.1, 6.3.4.5.2, 6.3.4.5.3, 6.3.4.7. Modified vi, 6.1.1.4, 6.2.2.1, 6.2.2.2, 6.2.2.3, 6.2.2.4, 6.2.2.5, 6.3.4.2, 6.3.4.5, Bibliography.	Additional content resulting from GO-1 proof-of-concept implementation.
11 Mar 04	0.4.0	G. Reynolds	Added 6.1.1.6, 6.3.4.3.1, 6.5, 6.5.1, 6.5.2, 6.5.3, 6.5.4, 6.5.5, 7.2.1, Annex	Third public draft

			B. Modified 6.1.1.5, Bibliography.	
14 May 04	0.5.0	E. Bertel	Modified i, vi, xi, 1, 2, 6.1, 6.1.1.1, 6.1.1.4, 6.1.1.6, 6.1.1.7, 6.2.1.2, 6.2.1.8, 6.2.2 (each), 6.3, 6.3.1 (each), , 6.3.4 (each), 6.3.4.6, Bibliography, Figures 1 - 32. Removed 6.3.2, 6.3.2.1, 6.3.3, 6.3.4.5.3, 6.4, 6.4.1, 6.4.2, 6.4.3.	Fourth public draft, incorporating changes based on comments at the April 2004 Technical Conference GO-1 RFC presentation, modifications to the GeoAPI baseline, and numerous minor editorial corrections.

v. Changes to the OpenGIS® Abstract Specification

The OpenGIS® Abstract Specification does not require changes to accommodate this OpenGIS® standard.

vi. Future Work

The Application Objects specification defines a set of core packages that support a small set of Geometries, a basic set of renderable Graphics that correspond to those Geometries, 2D device abstractions (displays, mouse, keyboard, etc.), and supporting classes. Implementation of these APIs will support the needs of many users of geospatial and graphic information. These APIs support the rendering of geospatial datasets, provide fine-grained symbolization of geometries, and support dynamic, event and user driven animation of geo-registered graphics.

We anticipate the need for extensions to this specification to support more specialized applications. It is likely that the core packages will warrant some granular enhancements, which would constitute revisions to the specification. Some extensions, however, will constitute major new capability areas. Implementing these extensions as a revision to this Application Objects specification would not be advisable, especially if the extension introduces a capability that not all implementers would want to support. These new capability areas should be defined in separate "extension" specifications that include the core specification by reference. Implementations would be declared compliant with one or more of these extensions, and consumers could choose a product that meets their applications' need.

We recommend that future work on new Application Object-dependent specifications be considered for the following extensions:

- 3D - to support 3D Geometries and 3D Graphics for objects such as surfaces and solids, perhaps the integration of standard 3D models such as VRML, and other 3D concepts.

- Advanced 2D - to support the more advanced 2D Geometries and 2D Graphics including those defined by Topic 1 (ISO-19107).

- Immediate Mode Rendering - to add an optional "call back" method to allow the application programmer to render Graphics using lightweight, transient calls during the physical rendering process (which is useful to support the rendering of extensive amounts of graphical information, but not easily supported by some implementations, such as distributed or client/server map engines). This allows an application programmer to reuse Geometry and Graphics objects to render many similar items (e.g., thousands of CurveSegments) and avoid the overhead of modelling them in memory, prior to render time. In addition to the performance considerations, this also allows for scale and location-dependent rendering to be done by the application, such as rendering sparse representations of grid data, where application logic must be used to calculate the correct placement of the graphics.

- Additional data sources - GO-1 has been architected to accommodate non-geospatial data models. The integration of non-GIS information models (engineering, modelling and simulation, etc.) into the GO-1 framework should be pursued.

We recommend that future work on new Application Object core specification be considered in the following areas:

- A more extensive investigation into the differences in requirements and capabilities of graphical vs. analytic geometry descriptions.

- The API will eventually need to be extended to give the Canvas the ability to span multiple processes and correctly align its state between those processes (e.g. a Canvas that is served to multiple network clients). The X protocol is a good example of an architecture that handles this situation.

Furthermore, we recommend that the work from GO-1 be considered for inclusion in the following OGC work areas:

- Style Layer Descriptor (SLD) - The GO-1 `GraphicStyle` can express certain concepts not found in SLD (e.g. `Viewability`, `Editability`, `Highlight`, `ArrowStyle`, `FillStyle`, `FillPattern`, `Symbology`). The SLD specification should be expanded to express these concepts.

- Coordinate Reference System (CRS) and Coordinate Transformation (CT) – The GO-1 API introduces the `Projection` object family, which extends the OGC `Conversion` object; the `MathTransform` object family as a decorator to the OGC `Operation` object family; and a pattern using a default `Factory` in concert with an `AuthorityFactory`. With the exception of Projection, all of these additions are implemented from the OGC 01-009 implementation specification (*Coordinate Transformation Services*).

-

Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open GIS Consortium Inc. shall not be held responsible for identifying any or all such patent rights.

This document consists of the following parts, under the main body:

• Clause 1: Scope

• Clause 2: Conformance

• Clause 3: Normative references

• Clause 4: Terms and definitions

• Clause 5: Conventions

• Clause 6: Design and Specification for Application Objects

• Clause 7 Behaviours

• Annex Documents: Detailed Implementation Specifications for Application Objects in External, Javadoc Documents

• Bibliography

Introduction

This document describes architectural and implementation issues concerning the development of a suite of software objects that facilitate the development of applications with geospatial content, as elucidated during the Geographic Objects Phase 1 Initiative (GO-1) conducted under the auspices of the Open GIS Consortium Interoperability program (OGC IP). The particular implementation focus of this initiative is interface definition and code organization in the Java programming language.

The bases of the interface definition are the object models defined in *The OpenGIS Abstract Specification Topic 1: Feature Geometry (ISO-19107 Spatial Schema) Version 5* (OGC 01-101r5) and *The OpenGIS Abstract Specification Topic 2: Spatial Referencing By Coordinates* (OGC 03-073r4). These models provide the architectural bridge between the OGC GO-1 *application-domain* specification and other OGC *service-domain* specifications.

OpenGIS© Interface — Application Objects

1 Scope

This OpenGIS document describes the specification for Application Objects. These are the Java implementations of objects and interfaces that can be used to implement geospatial applications.

Application Objects are oriented on the application domain (e.g. user-facing, localized processes and operations), and less so on the service domain (e.g. centralized processes and operations that are not necessarily exposed to the user).

2 Conformance

2.1 Types of Conformance

This document recognises two broad categories of conformance, *API conformance* and *functional conformance*. API conformance is the ability of an application to invoke all of the required operations without any unexpected returned values or states. API conformance does not require that the component actually do anything. Functional conformance mandates not only that the required operations can be invoked, but also that the component performs the operations in a standard and universally understood manner.

Because this API is intended to be used in a wide range of deployment environments, the primary focus of this document is upon API conformance. API conformance can be specified and tested in a manner that is implementation-neutral. When an operation is invoked, it either succeeds, or fails to produce the intended result. There is no ambiguity.

Functional conformance is more difficult and far more implementation-dependent. What is acceptable in one environment may not be adequate in another. For example, a high-performance, low-power display might be designed to render lines in only a few colours and styles. This would be inadequate for a more feature-rich unit used to develop cartographic imagery. Such differences in functionality should be invisible to a generic API. A rigid definition of functional conformance would limit component developers' ability to tailor their products to the requirements of their respective developer communities.

Even within the domain of API conformance, there is a wide spectrum of developer objectives and corresponding application types. Not all of these would benefit by incorporating every interface specified below. In the remainder of this section we describe various categories of conformance, and suggest the kinds of applications that might benefit most from each one.

Crucial to this notion are the object classes and interfaces that form natural suites of related functionality, or packages, that define the substance of the various conformance classes. Certain suites, like the Spatial Objects, can be implemented as compliant standalone object libraries. Others, like the Display Objects, are dependent upon one or more other frameworks, and compliant implementations of these must also comply with the specifications of the frameworks on which they depend. To utilize them implies a conformance to the object suites upon which they depend as well.

Even within a framework, there is variation among environments as to which operations and perhaps even which objects may be necessary or useful. Future versions of this specification may provide additional flexibility to implementers by defining different conformance profiles. Simpler profiles would offer less functionality, simpler implementation, and fewer resource requirements than the more extensive profiles.

2.2 Display Object Conformance

The Display Objects described in this document include the Canvas, Graphic, GraphicStyle, and the objects of the Event model. The Canvas is the rendering environment for the user. Graphics objects are the entities that a Canvas manipulates and renders according to the styling attributes of a GraphicStyle object. Events provide user input to the Canvas, and both control and notification between objects on the Canvas. Together, these constitute the display subsystem of an application.

Display system conformance confers a number of benefits upon applications that implement it. Some of these benefits are:

1. Implementations have a variety of architectural and design decisions already made for them. They implement patterns and benefit from best practices as identified by participants in the GO-1 initiative.

2. Among the patterns of interest are a consistent means of ingesting data from a variety of OGC-specified sources.

3. Users of these systems will find familiar user interaction paradigms and control semantics as they move between applications.

4. Applications loosely coupled to their display subsystems may connect with any of a number of local or remote displays, and may therefore provide a means to coordinate control or share information among a variety of distributed sites.

5. Thus display system conformance confers interoperability with respect to the display and user interface subsystem.

2.3 Spatial Object Conformance

Geometry, Coordinate Reference System, and related entities constitute the spatial objects defined by GO-1. These build upon the body of work that has resulted in the

OGC Abstract Specification Topic 1 (ISO-19107), OGC Abstract Specification Topic 2, and several OGC Discussion and Recommendation Papers.

Direct support of spatial objects confers interoperability with local conforming data sources and with remote services, like WFS, that provide an encoded stream of features per the definitions in these documents.

2.4 OGC Service Conformance

While this specification outlines certain service interfaces, it does not require the use of any particular service implementation. A conformal application may derive its data from one or more services as defined by any of the OGC service specifications. It may also act as a client to transformation or other processing services when they become available.

The OGC web services defined to date are effectively standalone. An application may conform to any one of them independently, without necessarily conforming to others.

3 Normative references

The following normative documents contain provisions that, through reference in this text, constitute provisions of this part of OGC 03-064. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of OGC 03-064 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies.

(Normative references are included in the Bibliography.)

4 Terms and definitions

For the purposes of this document, the terms and definitions given in Section 5.1 below apply.

5 Conventions

5.1 Symbols (and abbreviated terms)

API	Application Program Interface
COTS	Commercial Off The Shelf
CRS	Coordinate Reference System
CS	Coordinate System
GO-1	Geographic Objects, Phase 1

ISO	International Organisation for Standardisation
OGC	Open GIS Consortium
SLD	Styled Layer Descriptor
SRS	Spatial Reference System
UML	Unified Modelling Language
XML	eXtended Markup Language
1D	One Dimensional
2D	Two Dimensional
3D	Three Dimensional

5.2 UML Notation

The diagrams that appear in this standard are presented using the Unified Modelling Language (UML) static structure diagram. The UML notations used in this standard are described in the diagram below.

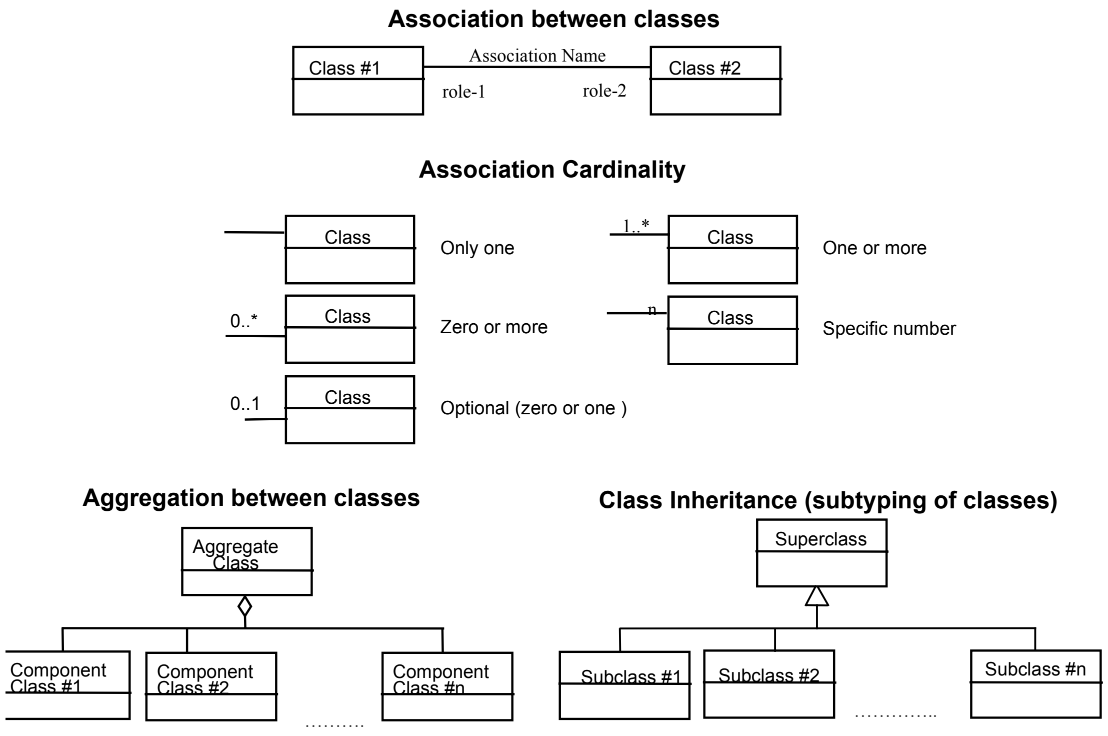


Figure 1 - UML notation

In this standard, the following three stereotypes of UML classes are used:

a) <<Interface>> A definition of a set of operations that is supported by objects having this interface. An Interface class cannot contain any attributes.

b) <<DataType>> A descriptor of a set of values that lack identity (independent existence and the possibility of side effects). A DataType is a class with no operations whose primary purpose is to hold the information.

c) <<CodeList>> is a flexible enumeration that uses string values for expressing a list of potential values.

In this standard, the following standard data types are used:

a) CharacterString – A sequence of characters

b) Integer – An integer number

c) Double – A double precision floating point number

d) Float – A single precision floating point number

6 Application Object Definitions

6.1 Factory

The GO-1 factory classes support a `getCapabilities()` operation that allows it to describe the features it supports. An application attempting to use a given implementation can invoke this method to determine whether an implementation is suitable for rendering its graphic information, or whether it would have to do extra work in order to use the implementation.

For example, the `CommonFactory.getCapabilities()` method returns an object that implements the `CommonCapabilities` interface. This object may then be queried about support for specific features. In order to ascertain display capabilities, the DisplayFactory.getCapabilities() method is used to obtain capabilities related to graphic primitives and styles.

Among the kinds of information an application may discover are various types of graphical rendering that the implementation is capable of doing, e.g., kinds of stroke and fill patterns available, support for blinking or backlighting, colour palette, line join styles and end caps, etc.

Graphic Object Creation

Graphic objects are created by invocation of the relevant creation method against a `DisplayFactory`. The Factory pattern, which is used extensively throughout GO-1, insulates client code from all details of the created class internals. `Graphic` creation methods may instantiate `Graphic` objects based on ISO-19107 geometries presented to them, but they may also be created using Shapefiles, or other formats for setting the geometry and geospatial location of a `Graphic`.

6.1 Display Objects

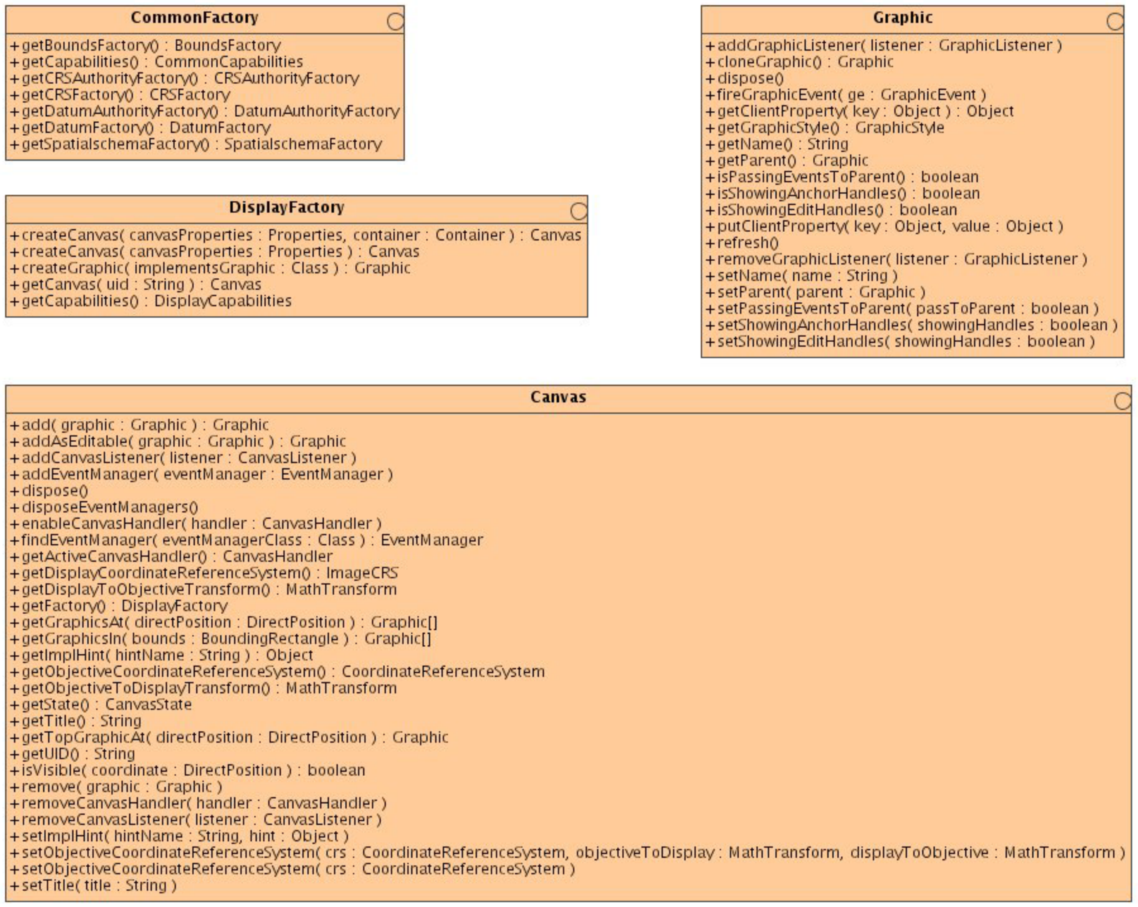
Display objects mediate the dynamic interactions of geospatial, graphical, or other data with the application. The particular role of such objects in the context of the present specification involves interaction with end users: displaying the data on a user-viewable device, and accepting user or programmatic input to control the application.

6.1.1 Canvas

6.1.1.1 General Description

The `Canvas` class defines a common abstraction for the display and user manipulation of geospatial information. It contains and manages a collection of `Graphic` objects that may be rendered as a map or represent features on a map, and maintains display context.

Instances of this class are created by the `DisplayFactory`. The Factory pattern, which is used extensively throughout GO-1, insulates client code from all details of the created class internals.



```
CommonFactory
+getBoundsFactory() : BoundsFactory
+getCapabilities() : CommonCapabilities
+getCRSAuthorityFactory() : CRSAuthorityFactory
+getCRSFactory() : CRSFactory
+getDatumAuthorityFactory() : DatumAuthorityFactory
+getDatumFactory() : DatumFactory
+getSpatialschemaFactory() : SpatialschemaFactory

DisplayFactory
+createCanvas( canvasProperties : Properties, container : Container ) : Canvas
+createCanvas( canvasProperties : Properties ) : Canvas
+createGraphic( implementsGraphic : Class ) : Graphic
+getCanvas( uid : String ) : Canvas
+getCapabilities() : DisplayCapabilities

Graphic
+addGraphicListener( listener : GraphicListener )
+cloneGraphic() : Graphic
+dispose()
+fireGraphicEvent( ge : GraphicEvent )
+getClientProperty( key : Object ) : Object
+getGraphicStyle() : GraphicStyle
+getName() : String
+getParent() : Graphic
+isPassingEventsToParent() : boolean
+isShowingAnchorHandles() : boolean
+isShowingEditHandles() : boolean
+putClientProperty( key : Object, value : Object )
+refresh()
+removeGraphicListener( listener : GraphicListener )
+setName( name : String )
+setParent( parent : Graphic )
+setPassingEventsToParent( passToParent : boolean )
+setShowingAnchorHandles( showingHandles : boolean )
+setShowingEditHandles( showingHandles : boolean )

Canvas
+add( graphic : Graphic ) : Graphic
+addAsEditable( graphic : Graphic ) : Graphic
+addCanvasListener( listener : CanvasListener )
+addEventManager( eventManager : EventManager )
+dispose()
+disposeEventManagers()
+enableCanvasHandler( handler : CanvasHandler )
+findEventManager( eventManagerClass : Class ) : EventManager
+getActiveCanvasHandler() : CanvasHandler
+getDisplayCoordinateReferenceSystem() : ImageCRS
+getDisplayToObjectiveTransform() : MathTransform
+getFactory() : DisplayFactory
+getGraphicsAt( directPosition : DirectPosition ) : Graphic[]
+getGraphicsIn( bounds : BoundingRectangle ) : Graphic[]
+getImplHint( hintName : String ) : Object
+getObjectiveCoordinateReferenceSystem() : CoordinateReferenceSystem
+getObjectiveToDisplayTransform() : MathTransform
+getState() : CanvasState
+getTitle() : String
+getTopGraphicAt( directPosition : DirectPosition ) : Graphic
+getUID() : String
+isVisible( coordinate : DirectPosition ) : boolean
+remove( graphic : Graphic )
+removeCanvasHandler( handler : CanvasHandler )
+removeCanvasListener( listener : CanvasListener )
+setImplHint( hintName : String, hint : Object )
+setObjectiveCoordinateReferenceSystem( crs : CoordinateReferenceSystem, objectiveToDisplay : MathTransform, displayToObjective : MathTransform )
+setObjectiveCoordinateReferenceSystem( crs : CoordinateReferenceSystem )
+setTitle( title : String )
```

Figure 2 - Canvas and related classes

6.1.1.2 Output Device

A `Canvas` is associated with an output device such as a window or a portion of a window on a display screen, or an image buffer. The `Canvas` is responsible for intelligent handling of the viewable area of the window, including panning, zooming, growing, and shrinking, repaints of "dirty" areas in the image due to external window changes, and visual changes in the `Graphics` due to editing, animation, or filtering.

6.1.1.3 Input Device

A `Canvas` may be associated with one or more input devices such as a mouse, keyboard, eye tracker, or gesture reader. These devices allow the user to manipulate the `Graphic` objects held by the `Canvas`. The `Canvas` manages the input events from these devices.

6.1.1.4 Coordinate Reference System

The `Canvas` maintains two coordinate reference systems (CRS):

1. The `Canvas` *display CRS* is associated with the geometry of the display device, and generally uses display coordinates such as *pixels*.

2. The `Canvas` *objective CRS* is associated with the data modelled by the Canvas, and is generally associated with model coordinates, such as *points*.

Most computer screens are a rectangular array of pixels, and would use a `CRS` backed by a `CartesianCS` for the display CRS. A planetarium or IMAX theatre is a spherical display, and might require a spherical coordinate reference system.

The `Canvas` objective CRS is typically a `ProjectedCRS` for a rendered map, but could be *a GeographicCRS if simple lat/lon rendering is desired, or* a non-georeferenced `CoordinateReferenceSystem`, such as an isometric projection of an `EngineeringCRS`.

The `Canvas` must provide accessors for two `MathTransform` objects, the first which specifies the particular transformation method from the objective CRS to the display CRS, and the second which specifies the transformation method from the display CRS to the objective CRS (note this latter transformation can be provided by `MathTransform.invert() method which is part of OGC 01-009 specification)`. This `MathTransform` shall be invertible, in order to get the transformation method from the display CRS to the objective CRS. For example if the ProjectedCRS defines objects in grid coordinates, the first transform can convert the grid coordinates of a ProjectedCRS to screen coordinates of the display CRS.
Alternately an implementation can choose to utilize as the objective CRS a non-projected `CoordinateReferenceSystem`, such as a `GeospatialCRS`, a `GeographicCRS`, or an `EngineeringCRS`.

Before adding a `Graphic` to a `Canvas` the user is responsible to ensure that the `CoordinateReferenceSystem` of the `Graphic` is supported by the implementation, by using `CommonCapabilities.getSupportedCoordinateReferenceSystems()`.

If the Graphic `CoordinateReferenceSystem` is not supported, then the client must transform the `Graphic` to an appropriate `CoordinateReferenceSystem` prior to adding it.

If the Graphic `CoordinateReferenceSystem` is supported, but is different than the objective CRS of the `Canvas`, the `Canvas` will transform the original `Graphic` object to a new `Graphic` object, discard the original `Graphic` object, and return a reference to the new `Graphic` object. The client is responsible to update its internal reference to the new `Graphic` object.

6.1.1.5 Z-Order and Rendering of Graphics

The `Canvas` controls the visual layering, or z-order, of the `Graphic` objects it contains. The z-order allows `Graphics` to overlap and occlude each other in a controllable way.

The Canvas may optimise its display by not rendering `Graphics` that are fully occluded.

Furthermore, when an input device selects a location on the display, the z-order allows the `Canvas` to designate the topmost `Graphic` (i.e., the highest z-order value for all `Graphics` at that coordinate location) as the object of interest.

In the general case of a distributed, asynchronous environment, the z-order cannot be designated deterministically by software external to the `Canvas`. To maximise the control of the situation, `GraphicStyle` objects have a z-order hint that the application can set, and the `Canvas` can read. When a `Graphic` is added to a `Canvas`, the `Canvas` gets the `Graphic's` z-order hint and attempts to place the `Graphic` at that z-order location. The z-order is defined as a double to permit a large range of values.

6.1.1.6 Canvas State

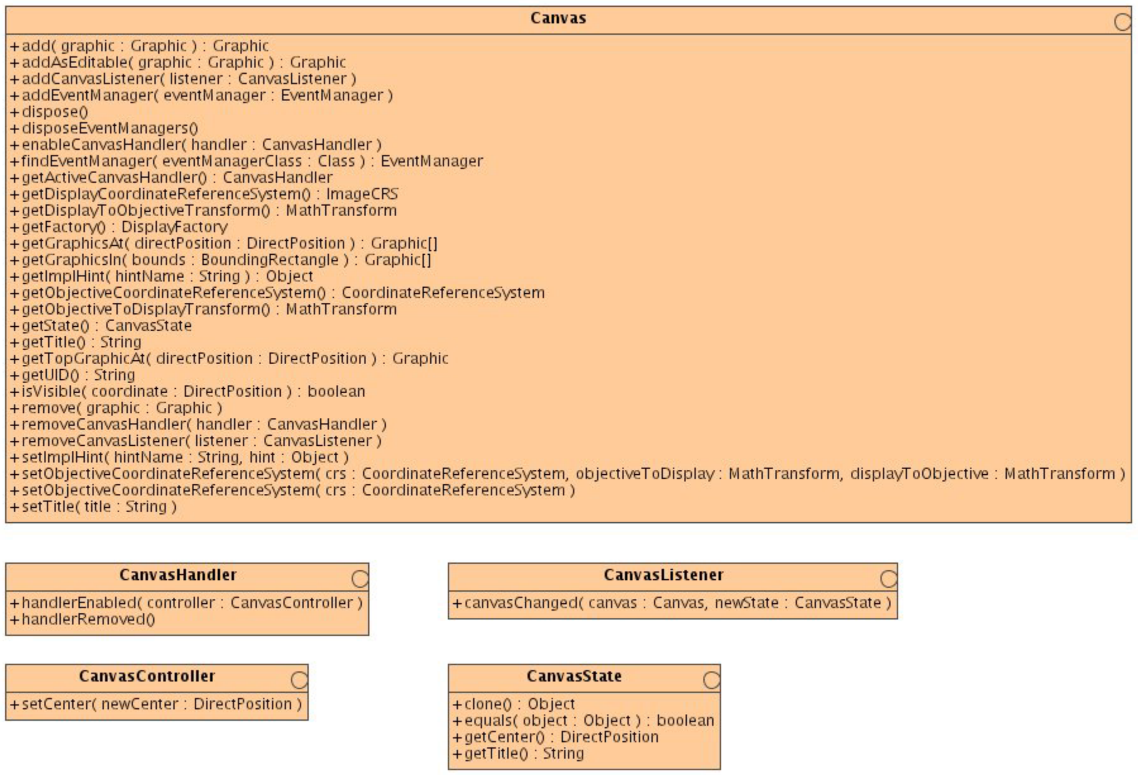

Canvas
+add(graphic : Graphic) : Graphic
+addAsEditable(graphic : Graphic) : Graphic
+addCanvasListener(listener : CanvasListener)
+addEventManager(eventManager : EventManager)
+dispose()
+disposeEventManagers()
+enableCanvasHandler(handler : CanvasHandler)
+findEventManager(eventManagerClass : Class) : EventManager
+getActiveCanvasHandler() : CanvasHandler
+getDisplayCoordinateReferenceSystem() : ImageCRS
+getDisplayToObjectiveTransform() : MathTransform
+getFactory() : DisplayFactory
+getGraphicsAt(directPosition : DirectPosition) : Graphic[]
+getGraphicsIn(bounds : BoundingRectangle) : Graphic[]
+getImplHint(hintName : String) : Object
+getObjectiveCoordinateReferenceSystem() : CoordinateReferenceSystem
+getObjectiveToDisplayTransform() : MathTransform
+getState() : CanvasState
+getTitle() : String
+getTopGraphicAt(directPosition : DirectPosition) : Graphic
+getUID() : String
+isVisible(coordinate : DirectPosition) : boolean
+remove(graphic : Graphic)
+removeCanvasHandler(handler : CanvasHandler)
+removeCanvasListener(listener : CanvasListener)
+setImplHint(hintName : String, hint : Object)
+setObjectiveCoordinateReferenceSystem(crs : CoordinateReferenceSystem, objectiveToDisplay : MathTransform, displayToObjective : MathTransform)
+setObjectiveCoordinateReferenceSystem(crs : CoordinateReferenceSystem)
+setTitle(title : String)
CanvasHandler
+handlerEnabled(controller : CanvasController)
+handlerRemoved()
CanvasListener
+canvasChanged(canvas : Canvas, newState : CanvasState)
CanvasController
+setCenter(newCenter : DirectPosition)
CanvasState
+clone() : Object
+equals(object : Object) : boolean
+getCenter() : DirectPosition
+getTitle() : String

Figure 3 - Canvas state and controls

To interact with the `Canvas`, outside entities must be aware of certain properties that provide context for graphical operations. Collectively, these properties comprise the `Canvas` state, and are contained in instances of `CanvasState`. This object describes only the viewing area or volume of the `Canvas`, not any state or other information about the data contained within it. When an instance of `CanvasState` is returned from `Canvas` methods, it contains a "snapshot" of the current state of the canvas. Its values never change, even if the state of the `Canvas` itself does.

For example, in an `XY` display CRS, a class implementing `CanvasState` would provide access to the following properties:

- `pixelWidth`
- `pixelHeight`
- `center`
- `width`
- `scale`
- `boundingRectangle`

Entities that are interested in reading `Canvas` state must implement the `CanvasListener` interface. `CanvasListener` includes the `canvasChanged()` method, which is called by a Canvas when its state has changed. The `Canvas` passes a populated `CanvasState` data object to the `canvasChanged()` method.

If an entity needs to change the state of a `Canvas`, it must implement the `CanvasHandler` interface. This interface provides a mechanism for multiple entities to change `Canvas` properties without contention or deadlock. The `Canvas` enables exactly one `CanvasHandler` at a time. When a `CanvasHandler` is enabled, the `Canvas` passes it a `CanvasController`, through which the entity can modify `Canvas` state values. The `CanvasController` remains active until another `CanvasHandler` is enabled.

This architecture assumes a `Canvas` that is in a single process. However, if the `Canvas` spans multiple processes, then a state alignment issue occurs, where a process may not detect a change initiated by another process. This specification does ***not*** address this latter scenario.

6.1.2 Events

6.1.2.1 Model and Rationale

The general paradigm for control by input devices is based on the Java Event model, and works as follows:

For each control device (e.g. a mouse or keyboard), there is specialized `Event` object type. When the device changes state (e.g., a mouse button is pressed), the system sends an instance of that `Event` to objects that have implemented and registered an `EventListener` interface for that device.

The event-handling mechanism for each device includes a stack of `EventHandlers`. When the `EventListener` receives an Event, it passes it to the first Handler on the stack. Each Handler implements specialized functionality – the system's response to the

Event - that it executes when it receives an Event. Then it can either consume the `Event`, or by not consuming it, allow it to be passed to the next handler in the stack. Thus the response of the system to a control input depends directly on the flow of `Events` through the Handler stack.

An `EventManager` interface allows the application to control which Handlers are on the stack, and in which order. This flexible arrangement allows the application to establish different modal responses for different states of the system, such as selection mode vs. editing mode.

There is also a `ManagerSupport` object that actually implements several of these interfaces, and does the real work that allows this input model to function.

This design is explained in greater detail in the following sections.

6.1.2.2 GO-1 Event Management

The GO-1 model for responding to user-actuated controls, programmatic state changes, and other asynchronous events is mediated through a general-purpose framework based on the Java Event model. In the Java model, a physical or programmatic change constitutes an event, which is represented by an Event object that contains information about the event and identifies the source of the event. Objects can implement an appropriate `EventListener` interface and register with the event source in order to receive events generated by that source. Some event sources, such as those that generate mouse or keyboard events, are present by default in the underlying system. Others may be implemented in the application or in library packages. Event sources *per se* are not a part of the response system documented here, but they motivate one important aspect of its organisation: for each source in a GO-1 implementation there is an event handling subsystem whose structure is described by the following paragraphs.

6.1.2.3 ManagerSupport Object

There is one instantiable class dedicated to each event management chain: a `ManagerSupport` class, typically named after the control that it supports: `MouseManagerSupport`, `KeyManagerSupport`, etc.

Each `ManagerSupport` object implements an `EventListener` subinterface appropriate to its event source, and registers as a listener for that source. Consequently, it receives `Events` for that source, but it does not respond to them directly. Instead, it manages a stack of `EventHandlers` (through its `EventManager` interface; see below) and passes `Events` it receives to one or more Handlers on the stack.

6.1.2.4 EventHandler Stack

For each event chain, there is at least one `EventHandler` object. The Handlers do the actual work of mediating the system's response to an event. For example, a

MouseHandler implements a mouseClicked() operation that may cause an object to be selected or highlighted.

Like the ManagerSupport object, a Handler implements the EventListener interface appropriate to its source, but it does not register as a listener. Instead, it implements the interface in order to inherit (and perhaps override) the relevant event handling methods.

As noted above, each Support object has a stack of Handlers. When it receives an event, the Support object invokes the appropriate action method against the top Handler on the stack. The Handler performs whatever specialized function this method implements, and then optionally consumes the event. If the event is not consumed, the Support object invokes the method against the next handler on the stack, and so on until the event is consumed. Thus an event may trigger a series of responses that varies according to the arrangement of EventHandlers on the stack. This mechanism may be used to implement modal behaviours in response to input events, such as a change from selection behaviours to editing behaviours in the application's response to mouse gestures.

6.1.2.5 EventManager

The ManagerSupport object also implements a subclass of the EventManager interface. EventManagers are concerned primarily with maintaining the stack of Handlers. They have methods to enable, push, pop, find, remove, and replace Handlers on the stack.

Canvas objects are loosely coupled with EventManager objects. The EventManager pattern is extensible to accommodate input devices beyond the traditional keyboard and mouse (such as eye tracker, gesture reader, etc.). Canvas support of particular EventManager implementations is determined through Canvas.findEventManager(Class eventManagerClass), where eventManagerClass is a class that extends EventManager, and whose implementation satisfies the various Event operations for that device.

Much of the user input will be processed by the Canvas, which manages its own event managers. The Graphic class, described under Graphical Data Objects below, also have event manages similar to the Canvas.

6.2 Graphical Data Objects

A graphical rendering environment differs from a general geospatial processing environment in several respects. For one thing, due to their inherently limited resolution and other physical constraints, raster display devices can only accurately depict a limited set of geometries. For another, each display device and corresponding software system may have its own notion of how to style the objects that it renders.

The most significant differences are more general, and incorporate the above particulars. Displays are often compact, high-performance, and necessarily specialized devices that raise issues familiar from the earlier days of general-purpose computing. Very robust, immensely flexible, and therefore large object systems intended to meet every possible functional requirement are both irrelevant and overly expensive in terms of memory requirements and processing overhead. Items that constitute the primary focus of functionality in a general context, such as a map, may be nothing more than a graphical background in a display system.

The classes described here are therefore lighter weight and less general than the ISO Geometry classes described in Section 6.3.1 (from the OGC Topic, i.e. ISO-19107). Nonetheless, they seek to retain the semantics and many of the behaviours of objects already defined by published or existing OGC standards. Where appropriate, they are defined as restrictions of the more general objects, and are typically instantiated via factory objects that take corresponding general-purpose spatial objects as arguments.

6.2.1 Graphic

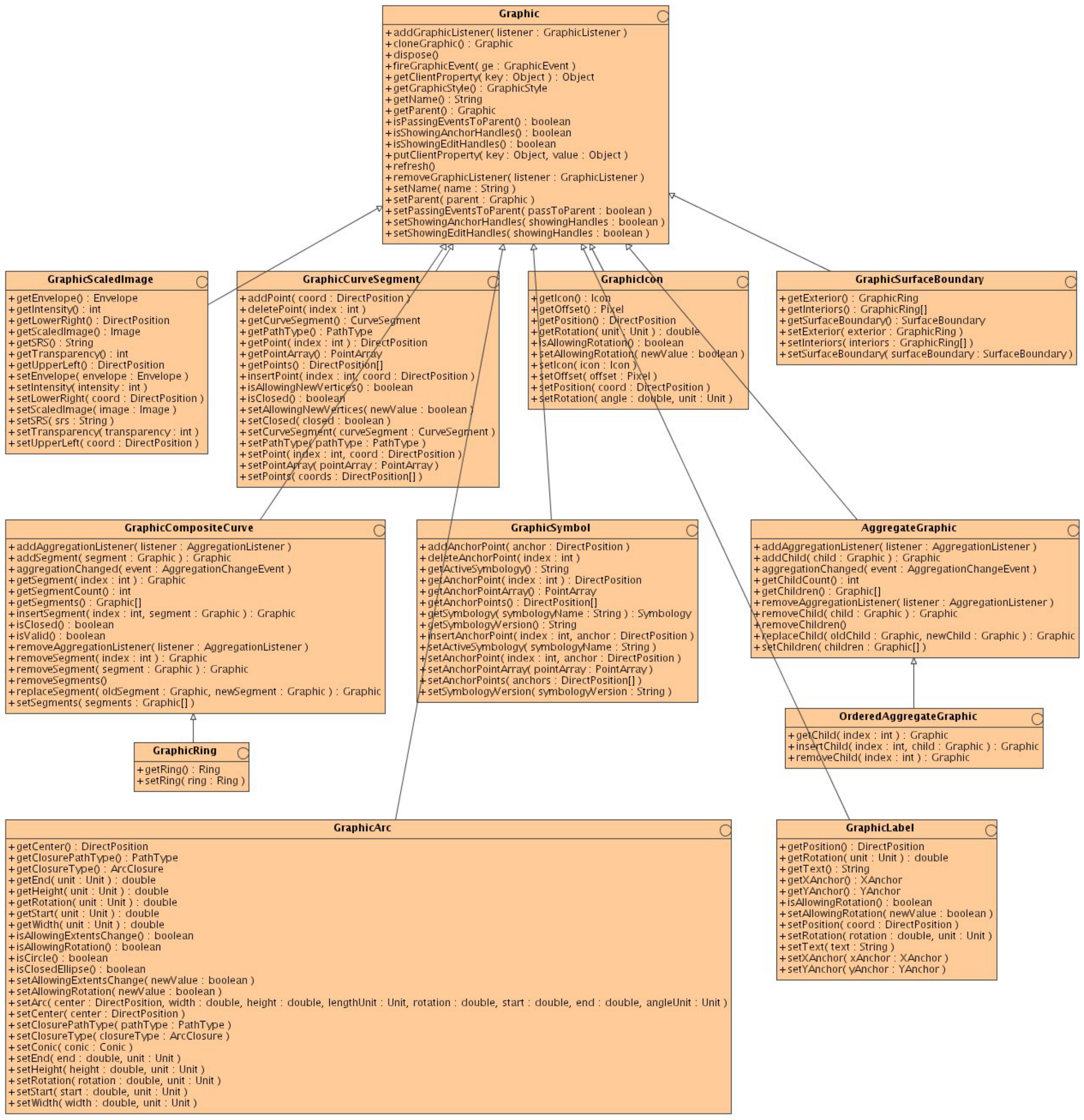

Graphic
+addGraphicListener(listener : GraphicListener)
+cloneGraphic() : Graphic
+dispose()
+fireGraphicEvent(ge : GraphicEvent)
+getClientProperty(key : Object) : Object
+getGraphicStyle() : GraphicStyle
+getName() : String
+getParent() : Graphic
+isPassingEventsToParent() : boolean
+isShowingAnchorHandles() : boolean
+isShowingEditHandles() : boolean
+putClientProperty(key : Object, value : Object)
+refresh()
+removeGraphicListener(listener : GraphicListener)
+setName(name : String)
+setParent(parent : Graphic)
+setPassingEventsToParent(passToParent : boolean)
+setShowingAnchorHandles(showingHandles : boolean)
+setShowingEditHandles(showingHandles : boolean)
GraphicScaledImage
+getEnvelope() : Envelope
+getIntensity() : int
+getLowerRight() : DirectPosition
+getScaledImage() : Image
+getSRS() : String
+getTransparency() : int
+getUpperLeft() : DirectPosition
+setEnvelope(envelope : Envelope)
+setIntensity(intensity : int)
+setLowerRight(coord : DirectPosition)
+setScaledImage(image : Image)
+setSRS(srs : String)
+setTransparency(transparency : int)
+setUpperLeft(coord : DirectPosition)
GraphicCurveSegment
+addPoint(coord : DirectPosition)
+deletePoint(index : int)
+getCurveSegment() : CurveSegment
+getPathType() : PathType
+getPoint(index : int) : DirectPosition
+getPointArray() : PointArray
+getPoints() : DirectPosition[]
+insertPoint(index : int, coord : DirectPosition)
+isAllowingNewVertices() : boolean
+isClosed() : boolean
+setAllowingNewVertices(newValue : boolean)
+setClosed(closed : boolean)
+setCurveSegment(curveSegment : CurveSegment)
+setPathType(pathType : PathType)
+setPoint(index : int, coord : DirectPosition)
+setPointArray(pointArray : PointArray)
+setPoints(coords : DirectPosition[])
GraphicIcon
+getIcon() : Icon
+getOffset() : Pixel
+getPosition() : DirectPosition
+getRotation(unit : Unit) : double
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean)
+setIcon(icon : Icon)
+setOffset(offset : Pixel)
+setPosition(coord : DirectPosition)
+setRotation(angle : double, unit : Unit)
GraphicSurfaceBoundary
+getExterior() : GraphicRing
+getInteriors() : GraphicRing[]
+getSurfaceBoundary() : SurfaceBoundary
+setExterior(exterior : GraphicRing)
+setInteriors(interiors : GraphicRing[])
+setSurfaceBoundary(surfaceBoundary : SurfaceBoundary)
GraphicCompositeCurve
+addAggregationListener(listener : AggregationListener)
+addSegment(segment : Graphic) : Graphic
+aggregationChanged(event : AggregationChangeEvent)
+getSegment(index : int) : Graphic
+getSegmentCount() : int
+getSegments() : Graphic[]
+insertSegment(index : int, segment : Graphic) : Graphic
+isClosed() : boolean
+isValid() : boolean
+removeAggregationListener(listener : AggregationListener)
+removeSegment(index : int) : Graphic
+removeSegment(segment : Graphic) : Graphic
+removeSegments()
+replaceSegment(oldSegment : Graphic, newSegment : Graphic) : Graphic
+setSegments(segments : Graphic[])
GraphicSymbol
+addAnchorPoint(anchor : DirectPosition)
+deleteAnchorPoint(index : int)
+getActiveSymbology() : String
+getAnchorPoint(index : int) : DirectPosition
+getAnchorPointArray() : PointArray
+getAnchorPoints() : DirectPosition[]
+getSymbology(symbologyName : String) : Symbology
+getSymbologyVersion() : String
+insertAnchorPoint(index : int, anchor : DirectPosition)
+setActiveSymbology(symbologyName : String)
+setAnchorPoint(index : int, anchor : DirectPosition)
+setAnchorPointArray(pointArray : PointArray)
+setAnchorPoints(anchors : DirectPosition[])
+setSymbologyVersion(symbologyVersion : String)
AggregateGraphic
+addAggregationListener(listener : AggregationListener)
+addChild(child : Graphic) : Graphic
+aggregationChanged(event : AggregationChangeEvent)
+getChildCount() : int
+getChildren() : Graphic[]
+removeAggregationListener(listener : AggregationListener)
+removeChild(child : Graphic) : Graphic
+removeChildren()
+replaceChild(oldChild : Graphic, newChild : Graphic) : Graphic
+setChildren(children : Graphic[])
OrderedAggregateGraphic
+getChild(index : int) : Graphic
+insertChild(index : int, child : Graphic) : Graphic
+removeChild(index : int) : Graphic
GraphicRing
+getRing() : Ring
+setRing(ring : Ring)
GraphicArc
+getCenter() : DirectPosition
+getClosurePathType() : PathType
+getClosureType() : ArcClosure
+getEnd(unit : Unit) : double
+getHeight(unit : Unit) : double
+getRotation(unit : Unit) : double
+getStart(unit : Unit) : double
+getWidth(unit : Unit) : double
+isAllowingExtentsChange() : boolean
+isAllowingRotation() : boolean
+isCircle() : boolean
+isClosedEllipse() : boolean
+setAllowingExtentsChange(newValue : boolean)
+setAllowingRotation(newValue : boolean)
+setArc(center : DirectPosition, width : double, height : double, lengthUnit : Unit, rotation : double, start : double, end : double, angleUnit : Unit)
+setCenter(center : DirectPosition)
+setClosurePathType(pathType : PathType)
+setClosureType(closureType : ArcClosure)
+setConic(conic : Conic)
+setEnd(end : double, unit : Unit)
+setHeight(height : double, unit : Unit)
+setRotation(rotation : double, unit : Unit)
+setStart(start : double, unit : Unit)
+setWidth(width : double, unit : Unit)
GraphicLabel
+getPosition() : DirectPosition
+getRotation(unit : Unit) : double
+getText() : String
+getXAnchor() : XAnchor
+getYAnchor() : YAnchor
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean)
+setPosition(coord : DirectPosition)
+setRotation(rotation : double, unit : Unit)
+setText(text : String)
+setXAnchor(xAnchor : XAnchor)
+setYAnchor(yAnchor : YAnchor)

Figure 4 - Graphic

6.2.1.1 General Description

Graphic objects contain the information needed by a `Canvas` to create a visual display. Similar in some respects to a Java 2 `Shape`, they contain geometric data, styling information (See `GraphicStyle`, Section 6.2.2), and geospatial coordinate location.

There are two broad categories of `Graphics`: primitives and aggregates. Primitive types are based on a simple rendered object (or an Icon, Text, or an Image) or are based on a primitive 2D ISO `Geometry` object, and include `GraphicCurveSegment`, `GraphicScaledImage`, `GraphicIcon`, `GraphicArc`, `GraphicLabel`, `GraphicCompositeCurve`, `GraphicRing`, and

GraphicSurfaceBoundary. Aggregates are collections of primitives, and include AggregateGraphic and OrderedAggregateGraphics.

Each primitive graphic object uses ISO-19107 Geometry to define its parameters. For example, GraphicScaledImage uses an Envelope to define its geographic extents. To the maximum extent the relationship allows, parameters for the Graphic object are stored in the underlying Geometry object fields. In cases where the underlying Geometry class does not have the same accessor and mutator methods, the Graphic must execute a conversion before storing the result in the underlying Geometry object. If the underlying Geometry object parameters change, the client is responsible for calling Graphic.refresh().

When a Graphic object changes or receives mouse or keyboard interaction it fires a GraphicEvent. A GraphicEvent can be received by objects that implement the GraphicListener interface. A client may receive GraphicEvents by creating an object that implements the GraphicListener interface and registering it with a Graphic via its addGraphicListener() method.

Graphic objects that are aggregations (i.e. AggregateGraphic and GraphicCompositeCurve, and their inheriting classes) can register AggregationListeners to listen for AggregationChangeEvents. These events are notifications when elements are added, removed, or (if applicable) reordered within the aggregation.

A Canvas knows how to read the attributes and geometric data from each Graphic type, and how to apply the styling information in the Graphic to create a visual representation. Graphics also contain a z-order hint, which the Canvas uses to help manage visual layering of the Graphics it displays.

Geometry objects are portable between implementations of the GO-1 specification. For example, an external program shall be able to create Geometry objects in one implementation but apply those Geometry objects to Graphic objects in any implementation of Canvas or Graphic.

Graphic objects are instantiated with a Factory pattern.

6.2.1.2 Primitives

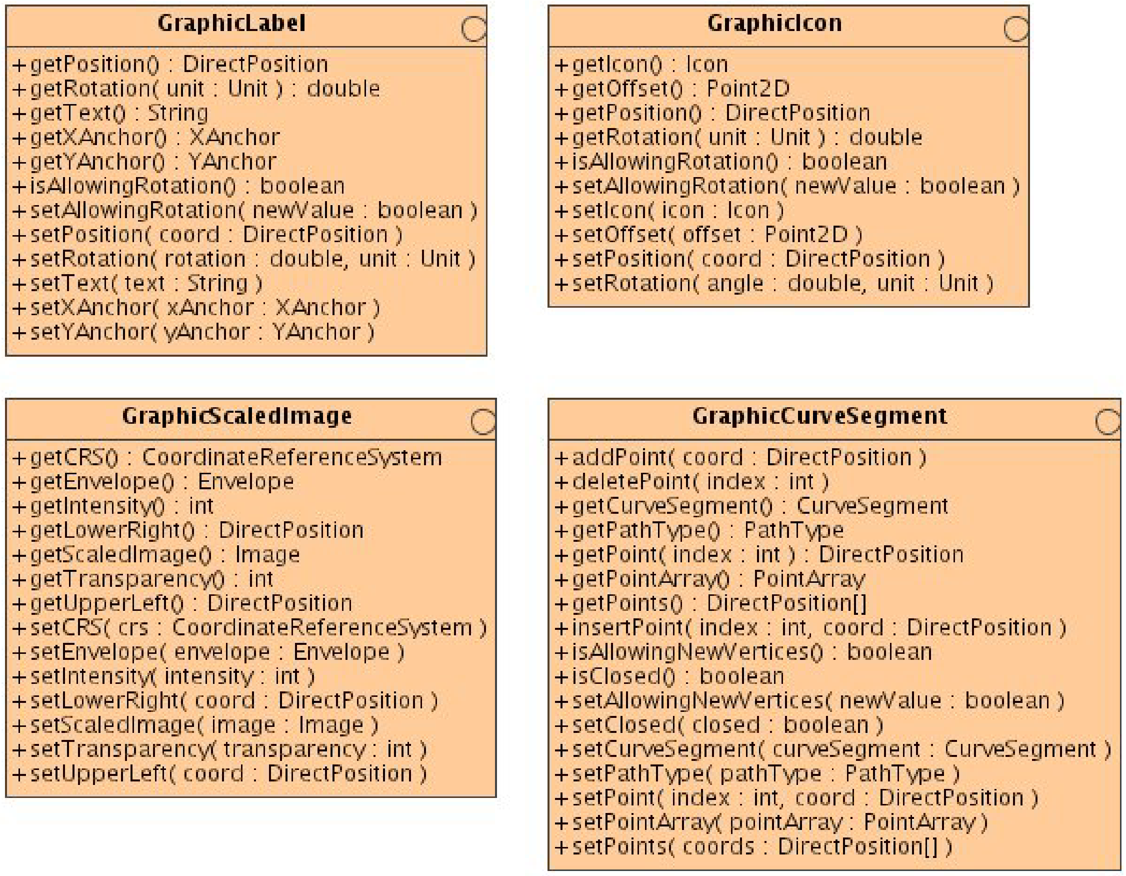

GraphicLabel
+getPosition() : DirectPosition
+getRotation(unit : Unit) : double
+getText() : String
+getXAnchor() : XAnchor
+getYAnchor() : YAnchor
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean)
+setPosition(coord : DirectPosition)
+setRotation(rotation : double, unit : Unit)
+setText(text : String)
+setXAnchor(xAnchor : XAnchor)
+setYAnchor(yAnchor : YAnchor)
GraphicIcon
+getIcon() : Icon
+getOffset() : Point2D
+getPosition() : DirectPosition
+getRotation(unit : Unit) : double
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean)
+setIcon(icon : Icon)
+setOffset(offset : Point2D)
+setPosition(coord : DirectPosition)
+setRotation(angle : double, unit : Unit)
GraphicScaledImage
+getCRS() : CoordinateReferenceSystem
+getEnvelope() : Envelope
+getIntensity() : int
+getLowerRight() : DirectPosition
+getScaledImage() : Image
+getTransparency() : int
+getUpperLeft() : DirectPosition
+setCRS(crs : CoordinateReferenceSystem)
+setEnvelope(envelope : Envelope)
+setIntensity(intensity : int)
+setLowerRight(coord : DirectPosition)
+setScaledImage(image : Image)
+setTransparency(transparency : int)
+setUpperLeft(coord : DirectPosition)
GraphicCurveSegment
+addPoint(coord : DirectPosition)
+deletePoint(index : int)
+getCurveSegment() : CurveSegment
+getPathType() : PathType
+getPoint(index : int) : DirectPosition
+getPointArray() : PointArray
+getPoints() : DirectPosition[]
+insertPoint(index : int, coord : DirectPosition)
+isAllowingNewVertices() : boolean
+isClosed() : boolean
+setAllowingNewVertices(newValue : boolean)
+setClosed(closed : boolean)
+setCurveSegment(curveSegment : CurveSegment)
+setPathType(pathType : PathType)
+setPoint(index : int, coord : DirectPosition)
+setPointArray(pointArray : PointArray)
+setPoints(coords : DirectPosition[])

Figure 5 – GraphicLabel, GraphicIcon, GraphicScaledImage, GraphicCurveSegment

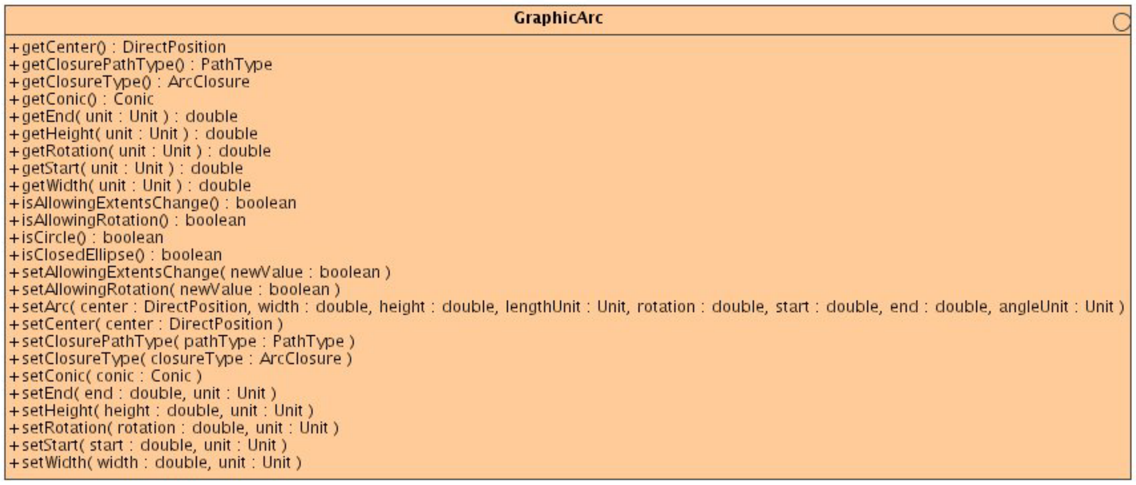
The palette of primitive shapes available to a Graphic is limited to a set that is sufficient for manipulation and rendering in graphical environments. Graphic objects themselves are subclassed according to the kind of geometry that they implement, and include the following:

`GraphicCurveSegment` defines common abstractions for implementations of 1-D lines made of one or more curve segments, as well as closed polygons made of a closed set of three or more curve segments. A settable attribute determines whether the `GraphicCurveSegment`is closed. A settable `PathType` attribute determines the interpolation between segment endpoints.

`GraphicScaledImage` provides an abstraction for implementing projected images defined by an upper left and a lower right point. This class includes methods for setting the image transparency and intensity as well as the image data. There are also methods for setting and getting the `CoordinateReferenceSystem` of the underlying `Envelope`, which specifies the projection of the image.

`GraphicIcon` defines a common abstraction for implementations to render icons on a drawing surface. The position of the icon in the `CoordinateReferenceSystem` is idealised as a single point attribute. The alignment of the icon to this point is specified as a pixel offset from the icon's upper left corner. The rotation of the label is measured positively as a clockwise angle, starting from a reference line within the `CoordinateReferenceSystem`.

`GraphicLabel` defines a common abstraction for implementations to render text on a drawing surface. The position of the label in the `CoordinateReferenceSystem` is idealised as a single point attribute. The alignment the label to this point is specified by the x-anchor and y-anchor. The rotation of the label is measured positively as a clockwise angle, starting from a reference line within the `CoordinateReferenceSystem`.



```
GraphicArc
+getCenter() : DirectPosition
+getClosurePathType() : PathType
+getClosureType() : ArcClosure
+getConic() : Conic
+getEnd( unit : Unit ) : double
+getHeight( unit : Unit ) : double
+getRotation( unit : Unit ) : double
+getStart( unit : Unit ) : double
+getWidth( unit : Unit ) : double
+isAllowingExtentsChange() : boolean
+isAllowingRotation() : boolean
+isCircle() : boolean
+isClosedEllipse() : boolean
+setAllowingExtentsChange( newValue : boolean )
+setAllowingRotation( newValue : boolean )
+setArc( center : DirectPosition, width : double, height : double, lengthUnit : Unit, rotation : double, start : double, end : double, angleUnit : Unit )
+setCenter( center : DirectPosition )
+setClosurePathType( pathType : PathType )
+setClosureType( closureType : ArcClosure )
+setConic( conic : Conic )
+setEnd( end : double, unit : Unit )
+setHeight( height : double, unit : Unit )
+setRotation( rotation : double, unit : Unit )
+setStart( start : double, unit : Unit )
+setWidth( width : double, unit : Unit )
```

Figure 6 - GraphicArc

`GraphicArc` provides definitions for closed circles and ellipses, as well as circular or elliptical arcs. Various settable attributes control its size, width, height, and orientation, and whether the object can be rotated or resized by the user. In this context *width* always refers to the major axis and *height* to the minor axis. Orientation start and end angles are defined counter-clockwise from the x-axis.

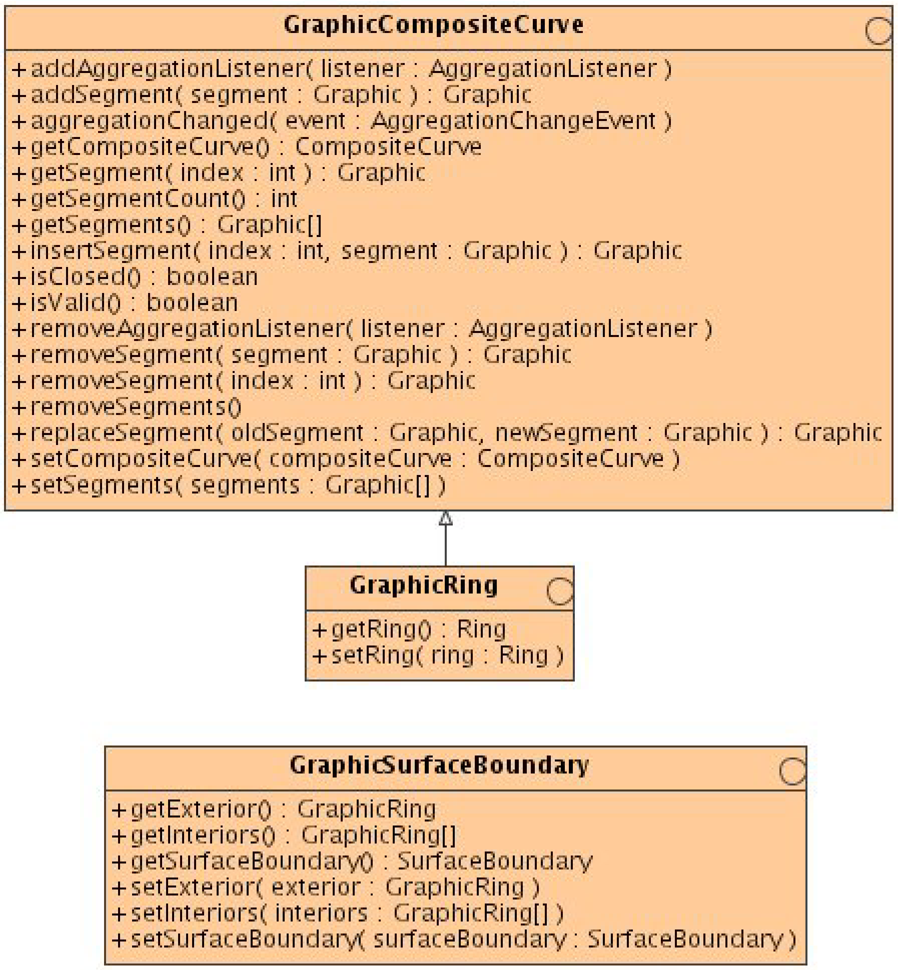

GraphicCompositeCurve
+addAggregationListener(listener : AggregationListener)
+addSegment(segment : Graphic) : Graphic
+aggregationChanged(event : AggregationChangeEvent)
+getCompositeCurve() : CompositeCurve
+getSegment(index : int) : Graphic
+getSegmentCount() : int
+getSegments() : Graphic[]
+insertSegment(index : int, segment : Graphic) : Graphic
+isClosed() : boolean
+isValid() : boolean
+removeAggregationListener(listener : AggregationListener)
+removeSegment(segment : Graphic) : Graphic
+removeSegment(index : int) : Graphic
+removeSegments()
+replaceSegment(oldSegment : Graphic, newSegment : Graphic) : Graphic
+setCompositeCurve(compositeCurve : CompositeCurve)
+setSegments(segments : Graphic[])
GraphicRing
+getRing() : Ring
+setRing(ring : Ring)
GraphicSurfaceBoundary
+getExterior() : GraphicRing
+getInteriors() : GraphicRing[]
+getSurfaceBoundary() : SurfaceBoundary
+setExterior(exterior : GraphicRing)
+setInteriors(interiors : GraphicRing[])
+setSurfaceBoundary(surfaceBoundary : SurfaceBoundary)

Figure 7 – GraphicCompositeCurve, GraphicRing, GraphicSurfaceBoundary

`GraphicCompositeCurve` extends the `Graphic` class to accommodate the creation of open or closed continuous curves of arbitrary type. For example, a figure "8" is a closed ISO-19107 `CompositeCurve`, while a capital "W" shape is an open `CompositeCurve`. The aggregate is ordered, and the end point of each element in the aggregate is the beginning point of the next one. This interface includes methods to add, insert, remove, and replace component Graphics within this sequence.

`GraphicRing` is a graphical representation of a `Ring` geometry. A `Ring` is a CompositeCurve that is closed and whose line segments do not cross. This behaviour must be enforced by the implementation on the methods inherited from the parent class. `GraphicRing` can be explicitly composed of other `Graphic` objects (via the methods inherited from `GraphicCompositeCurve`), or can get its geometry information directly from a `Ring` (via the `setRing()` method).

`GraphicSurfaceBoundary` is a graphical representation of a `SurfaceBoundary` geometry. `GraphicSurfaceBoundary` can be explicitly composed from `GraphicRing` objects, or can get its geometry information directly from a

SurfaceBoundary geometry. If a GraphicSurfaceBoundary is generated by the aggregation of GraphicRings, it is left up to the implementation to ensure that the equivalent SurfaceBoundary geometry is topologically correct (i.e. the constituent Rings do not touch or overlap).

6.2.1.3 Aggregates

AggregateGraphic defines a common abstraction for implementations of aggregated Graphic objects. This abstraction makes no assumptions about how the Graphics are stored within the aggregate. For example, the Graphics may be stored in an array such that the Graphic in the zero element of the array is considered the front most (highest z-order) object and the Graphic in the largest element of the array is considered the bottommost (lowest z-order) object. Alternatively, the Graphics may be stored in a more efficient data structure.

This abstraction makes no assumptions about thread safety. Implementations of Graphic that are to be used in a multi-threaded environment must address thread safety by using synchronised methods or by invoking all methods from a single thread.

OrderedAggregateGraphic extends the AggregateGraphic interface to add the ability for the user to specify a stacking order or Z-order. When the objects contained in this aggregate are drawn, they should be drawn in the order they appear in the list of children, starting with index 0.

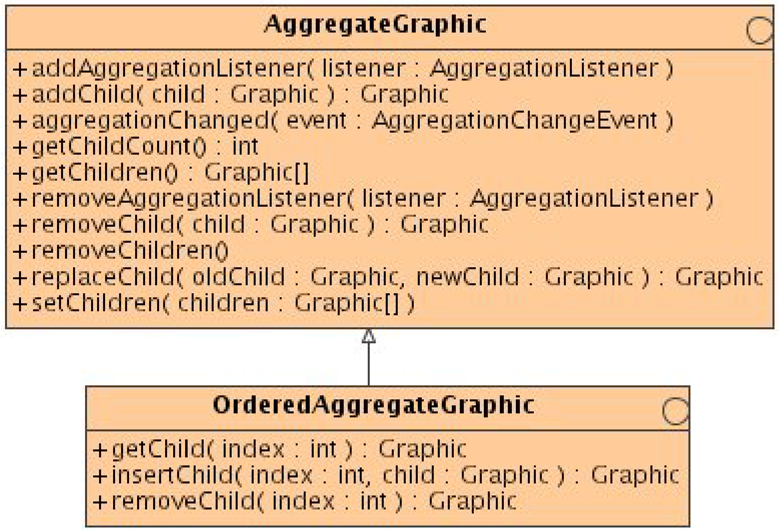

AggregateGraphic
+addAggregationListener(listener : AggregationListener)
+addChild(child : Graphic) : Graphic
+aggregationChanged(event : AggregationChangeEvent)
+getChildCount() : int
+getChildren() : Graphic[]
+removeAggregationListener(listener : AggregationListener)
+removeChild(child : Graphic) : Graphic
+removeChildren()
+replaceChild(oldChild : Graphic, newChild : Graphic) : Graphic
+setChildren(children : Graphic[])
OrderedAggregateGraphic
+getChild(index : int) : Graphic
+insertChild(index : int, child : Graphic) : Graphic
+removeChild(index : int) : Graphic

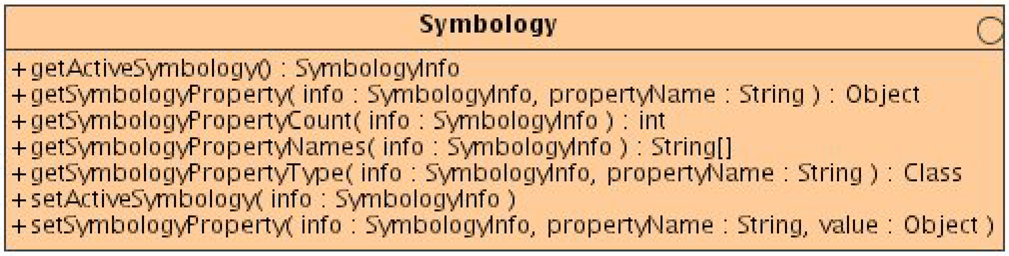
Figure 8 - AggregateGraphic, OrderedAggregateGraphic

6.2.1.4 Graphic Symbols

A symbol can be depicted on a map using one of two techniques, pictorial or abstract. Pictorial symbols are those that are designed to replicate or look like the feature they represent, such as a cross to identify a hospital or a ship to symbolize a port. They do not necessarily have a direct connection to what they identify. Abstract symbols are usually represented by a geometric shape, and bear no relationship to the form of the object they symbolize.

Symbols can be further defined in terms of their dimension; point (no-dimension), line (1-dimension), area (2-dimensional), and volume (3-dimensional). Other visual attributes used to describe a symbol include a combination of size, shape, orientation, color, and pattern. Most or all of these symbol attributes are pre-determined when a symbology standard is applied.

Currently there exist a large number of symbology standards covering a broad range of public and private sector applications. Some of these standards are currently under development, while new standards are being proposed. Our approach seeks to provide generic support for both existing and future standards, without mandating the use of any specific standard.



```
+getActiveSymbology() : SymbologyInfo
+getSymbologyProperty( info : SymbologyInfo, propertyName : String ) : Object
+getSymbologyPropertyCount( info : SymbologyInfo ) : int
+getSymbologyPropertyNames( info : SymbologyInfo ) : String[]
+getSymbologyPropertyType( info : SymbologyInfo, propertyName : String ) : Class
+setActiveSymbology( info : SymbologyInfo )
+setSymbologyProperty( info : SymbologyInfo, propertyName : String, value : Object )
```

Figure 9 – Symbology

6.2.1.4.1 Symbology

GO-1 uses a canonical approach to represent standard symbology sets. Tag names and corresponding data type values are explicitly typed in advance, and described in Appendix B. Some tags are well known and defined by existing accepted standards. Others must be derived from best industry practices, existing conventions, or consensus. Additions to the symbology tags presented in this document are expected and encouraged, and stakeholders are urged to collaborate on additions and revisions. Prior to acceptance of this proposed standard, the content of Appendix B is subject to revision. Tags defined post-acceptance of this specification may be subject to deprecation, but should not be modified or deleted. An XML schema to describe tags for a supported Symbology would look like the following.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.polexis.com/site"
   xmlns:site="http://www.polexis.com/site"
   xmlns:sld="http://www.opengis.net/sld"
elementFormDefault="qualified">
<xs:element name="tag" type=symbologyTag>
    <xs:complexType name=symbologyTag>
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="type" type="xs:string"/>
```

```
        <xs:element name="description" type="xs:string" use=”optional”/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

6.2.1.4.2 Visibility Tag

A “.visibility” tag modifier can be optionally appended to any tag name. The data type for visibility is Boolean. Presence of a visibility value of Boolean.TRUE would mean that component should be rendered, and a value of Boolean.FALSE would mean don’t display the component. The following snippet would turn off the AdditionalInformation indicator for the MIL-STD 2525 symbol use case in Section 7.5.2.

```
symbology.setSymbologyProperty(symInfo, "AdditionalInformation", "RAF418");
symbology. setSymbologyProperty(symInfo, "AdditionalInformation.visibility",
Boolean.FALSE);
```

6.2.1.5 Graphic Object Creation

Graphic objects are created by invocation of the relevant creation method against a `DisplayFactory`. `Graphic` creation methods may instantiate `Graphic` objects based on ISO-19107 geometries presented to them, but they may also be created using Shapefiles, or other formats for setting the geometry and geospatial location of a `Graphic`.

The GO-1 factory classes support a `getCapabilities()` operation that allows it to describe the features it supports. An application attempting to use a given implementation can invoke this method to determine whether an implementation is suitable for rendering its graphic information, or whether it would have to do extra work in order to use the implementation.

For example, the `CommonFactory.getCapabilities()` method returns an object that implements the `CommonCapabilities` interface. This object may then be queried about support for specific features. In order to ascertain display capabilities, the DisplayFactory.getCapabilities() method is used to obtain capabilities related to graphic primitives and styles. Among other things, one may query for the types of graphical rendering that the implementation is capable of doing, e.g., kinds of stroke and fill patterns available, support for blinking or backlighting, colour palette, line join styles and end caps, etc.

6.2.1.6 Path Type

Path types describe how lines are rendered with respect to the modelled surface of the earth. The categories of path type are:

- Global, consisting of *rhumbline* and *great circle* types

- Unprojected, consisting of *pixel straight* and *spline* types
- Vector

`PathType` serves as the base class for objects that represent the various methods for computing a path between two locations. Singleton instances of `PathType` will exist to represent, for example, a path of constant bearing (rhumbline), or a great circle path.

Path type is an algorithmic sequence of interpolation and projection.

- For **rhumbline**, **great circle**, and **vector**, first *interpolation* is done on the vertices, which gives in-between points. These in-between points are then *projected* into the `Canvas` display CRS, which converts them to display points.
- For **pixel-straight** and **spline**, the vertices are first *projected* into the `Canvas` display CRS as display points. These display points are *interpolated*, which generates in-between display points.

For each path type, an implementation will iteratively apply the respective algorithms until the appropriate display resolution is reached.

Path Type	**Interpolation Method**
rhumbline	constant bearing
great circle	geodesic
vector	linear in world space (interpolation before projection)
pixel-straight	linear in display space (interpolation after projection)
spline	cubic in display space (interpolation after projection)

Table 1 – Path Types

The ***Global path type*** methods calculate a path between two locations, considering the shape of the earth. The in-between points of the path satisfy two conditions:

1. The in-between points are the same regardless of the way the current path is displayed (i.e., the path is independent of map projection, `Canvas`, or other considerations affecting rendering or portrayal).

2. The in-between points are calculated along a surface that the points are projected onto, such as the surface of the 3D earth.

The second condition implies that altitude is not taken into account when calculating Global paths. Hence, paths of this type are well suited for navigation of surface ships or vehicles.

This specification defines four path types:

- Great Circle Ellipsoidal
- Great Circle Spherical
- Rhumbline Ellipsoidal
- Rhumbline Spherical

Great circle uses the shortest line on the surface of the earth, assuming either a spherical or an ellipsoidal earth model. Rhumbline uses a line of constant bearing along the surface of the earth, also using either a spherical or an ellipsoidal model.

The ***Unprojected path type*** methods calculate a path between two locations, not considering the shape of the earth, but considering the surface of the `Canvas`.

The methods are:

- Pixel Straight
- Continuous Spline

Pixel straight connects each sequential point with the shortest line on the `Canvas`. Continuous Spline uses an interpolation method to connect more than two points.

The ***Vector path type*** considers the surface of the earth, but connects sequential point locations with the shortest direct line, even if it travels below the curved surface of the 3D earth.

6.2.2 GraphicStyle

6.2.2.1 Relationship to Graphic

The `GraphicStyle` class allows a `Graphic` to be visually decorated. Each `Graphic` contains a `GraphicStyle` object, and no other graphic can own that style object. If a particular property on a GraphicStyle is not set, the Graphic will use the property of its nearest ancestor Graphic that defines the property. If no ancestor defines the property then a default value is used.

6.2.2.2 Relationship to OGC SLD

The `GraphicStyle` class has been developed to be as symmetric as possible with SLD. However, SLD requires that objects be either topologically open or closed, and immutable after being styled. `GraphicArc`, `GraphicCurveSegment`, and `GraphicCompositeCurve` can be topologically open or closed, and are modifiable after being styled, so no direct mapping to SLD is possible.

GraphicStyle can express certain concepts not found in SLD (e.g. Viewability, Editability, Highlight, ArrowStyle, FillStyle, FillPattern, Symbology). It is recommended that the SLD specification be expanded to express these concepts. See Future Work Section.

6.2.2.3 GraphicStyle elements

GraphicStyle is structured as a single interface, extending sub-interfaces containing groups of paired accessor and mutator methods corresponding to an individual style element (see table below).

Additionally, GraphicStyle has an accessor and mutator for sub-interfaces consisting of groups of like elements. An example of such an interface is LineSymbolizer.

The SLD-analogous interfaces are: LineSymbolizer, PolygonSymbolizer, PointSymbolizer, TextSymbolizer. Additional non-SLD interfaces are: Viewability, Editability, Highlight, and Symbology.

- LineSymbolizer is closely related to SLD LineSymbolizer in that it decorates lines.
- PolygonSymbolizer is closely related to SLD PolygonSymbolizer in that it decorates polygonal shapes.
- PointSymbolizer is closely related to the SLD PointSymbolizer in that it decorates icons.
- TextSymbolizer is closely related to the SLD TextSymbolizer. The TextSymbolizer.FONT component is defined as the Java implementation of Font, as it more accurately defines and distinguishes the concepts of character and glyph than does the SLD model.
- Viewability allows a Graphic to be made unconditionally invisible, or conditionally invisible based on a range specified by maxScale and/or minScale. If maxScale is set, and the Canvas exceeds that scale, the Graphic is made invisible. Similarly if minScale is set and the Canvas drops below that scale, the Graphic is made invisible. This in/visibility does not change the transparency values of GraphicStyle components, but instead overrides their effect. The z-order hint is used by the Canvas to place the Graphic in the z-order [see Section 6.1.1.5].
- Editability allows the Graphic to be edited.
- Highlight controls whether a Graphic can blink, and if so, at what rate
- Symbology when defined supersedes any of the other symbolizer objects.

The following table details the group interfaces, the elements, their types, and their default values.

Interface	Element	Type	Default
LineSymbolizer	LINE_STROKE_BEGIN_ARROW_STYLE	ArrowStyle	ArrowStyle.NONE
	LINE_STROKE_COLOR	Color	Color.BLACK
	LINE_STROKE_DASH_ARRAY	DashArray	DashArray.NONE
	LINE_STROKE_DASH_OFFSET	float	0.0
	LINE_STROKE_END_ARROW_STYLE	ArrowStyle	ArrowStyle.NONE
	LINE_STROKE_FILL_BACKGROUND_COLOR	Color	Color.GRAY
	LINE_STROKE_FILL_COLOR	Color	Color.BLACK
	LINE_STROKE_FILL_GRADIENT_POINTS	float[2]	*N/A*
	LINE_STROKE_FILL_OPACITY	float	1.0
	LINE_STROKE_FILL_PATTERN	FillPattern	FillPattern.NONE
	LINE_STROKE_FILL_STYLE	FillStyle	FillStyle.SOLID
	LINE_STROKE_LINECAP	LineCap	LineCap.BUTT
	LINE_STROKE_LINEGAP	float	10.0
	LINE_STROKE_LINEJOIN	LineJoin	LineJoin.BEVEL
	LINE_STROKE_LINESTYLE	LineStyle	LineStyle.SINGLE
	LINE_STROKE_OPACITY	float	1.0
	LINE_STROKE_PATTERN	FillPattern	FillPattern.NONE
	LINE_STROKE_WIDTH	float	1.0
PolygonSymbolizer	POLYGON_FILL_BACKGROUND_COLOR	Color	Color.GRAY
	POLYGON_FILL_COLOR	Color	Color.BLACK
	POLYGON_FILL_GRADIENT_POINTS	float[2]	*N/A*
	POLYGON_FILL_OPACITY	float	1.0
	POLYGON_FILL_PATTERN	FillPattern	FillPattern.NONE
	POLYGON_FILL_STYLE	FillStyle	FillStyle.SOLID
	POLYGON_STROKE_BEGIN_ARROW_STYLE	ArrowStyle	ArrowStyle.NONE
	POLYGON_STROKE_COLOR	Color	Color.BLACK
	POLYGON_STROKE_DASH_ARRAY	DashArray	DashArray.NONE
	POLYGON_STROKE_DASH_OFFSET	Float	0.0
	POLYGON_STROKE_END_ARROW_STYLE	ArrowStyle	ArrowStyle.NONE
	POLYGON_STROKE_FILL_BACKGROUND_COLOR	Color	Color.GRAY
	POLYGON_STROKE_FILL_COLOR	Color	Color.BLACK
	POLYGON_STROKE_FILL_GRADIENT_POINTS	float[2]	*N/A*
	POLYGON_STROKE_FILL_OPACITY	float	1.0
	POLYGON_STROKE_FILL_PATTERN	FillPattern	FillPattern.NONE
	POLYGON_STROKE_FILL_STYLE	FillStyle	FillStyle.SOLID
	POLYGON_STROKE_LINECAP	LineCap	LineCap.BUTT
	POLYGON_STROKE_LINEGAP	float	10.0
	POLYGON_STROKE_LINEJOIN	LineJoin	LineJoin.BEVEL
	POLYGON_STROKE_LINESTYLE	LineStyle	LineStyle.SINGLE
	POLYGON_STROKE_OPACITY	float	1.0
	POLYGON_STROKE_PATTERN	FillPattern	FillPattern.NONE
	POLYGON_STROKE_WIDTH	float	1.0
PointSymbolizer	POINT_FILL_BACKGROUND_COLOR	Color	Color.GRAY
	POINT_FILL_COLOR	Color	Color.BLACK
	POINT_FILL_GRADIENT_POINTS	float[2]	*N/A*
	POINT_FILL_OPACITY	float	1.0
	POINT_FILL_PATTERN	FillPattern	FillPattern.NONE
	POINT_FILL_STYLE	FillStyle	FillStyle.SOLID
	POINT_MARK	Mark	Mark.CIRCLE
	POINT_OPACITY	float	1.0
	POINT_ROTATION	float	0.0

	POINT_SIZE	float	16.0
	POINT_STROKE_BEGIN_ARROW_STYLE	ArrowStyle	ArrowStyle.NONE
	POINT_STROKE_COLOR	Color	Color.BLACK
	POINT_STROKE_DASH_ARRAY	DashArray	DashArray.NONE
	POINT_STROKE_DASH_OFFSET	Float	0.0
	POINT_STROKE_END_ARROW_STYLE	ArrowStyle	ArrowStyle.NONE
	POINT_STROKE_FILL_BACKGROUND_COLOR	Color	Color.GRAY
	POINT_STROKE_FILL_COLOR	Color	Color.BLACK
	POINT_STROKE_FILL_GRADIENT_POINTS	float[2]	*N/A*
	POINT_STROKE_FILL_OPACITY	float	1.0
	POINT_STROKE_FILL_PATTERN	FillPattern	FillPattern.NONE
	POINT_STROKE_FILL_STYLE	FillStyle	FillStyle.SOLID
	POINT_STROKE_LINECAP	LineCap	LineCap.BUTT
	POINT_STROKE_LINEGAP	float	10.0
	POINT_STROKE_LINEJOIN	LineJoin	LineJoin.BEVEL
	POINT_STROKE_LINESTYLE	LineStyle	LineStyle.SINGLE
	POINT_STROKE_OPACITY	float	1.0
	POINT_STROKE_PATTERN	FillPattern	FillPattern.NONE
	POINT_STROKE_WIDTH	float	1.0
TextSymbolizer	TEXT_FILL_BACKGROUND_COLOR	Color	Color.WHITE
	TEXT_FILL_COLOR	Color	Color.BLACK
	TEXT_FILL_GRADIENT_POINTS	float[2]	*N/A*
	TEXT_FILL_OPACITY	float	1.0
	TEXT_FILL_PATTERN	FillPattern	FillPattern.NONE
	TEXT_FILL_STYLE	FillStyle	FillStyle.SOLID
	TEXT_FONT	Font	*N/A*
	TEXT_HALO_RADIUS	Float	1.0
	TEXT_LABEL	String	*N/A*
	TEXT_LABEL_ROTATION	float	0.0
	TEXT_LABEL_SHOW_LABEL	boolean	false
	TEXT_LABEL_XANCHOR	Xanchor	Xanchor.LEFT
	TEXT_LABEL_Y_DISPLACEMENT	float	0.0
	TEXT_LABEL_YANCHOR	YAnchor	YAnchor.MIDDLE
	TEXT_LABEL_Y_DISPLACEMENT	float	0.0
Viewability	VIEWABILITY_MAX_SCALE	int	Integer.MAX_VALUE
	VIEWABILITY_MIN_SCALE	int	1.0
	VIEWABILITY_VISIBLE	boolean	true
	VIEWABILITY_Z_ORDER_HINT	double	0.0
Editability	EDITABILITY_AUTO_EDIT	boolean	true
	EDITABILITY_DRAG_SELECTABLE	boolean	true
	EDIABILITY_PICKABLE	boolean	true
	EDITABILITY_SELECTED	boolean	false
Highlight	HIGHLIGHT_BLINKING	boolean	false
	HIGHLIGHT_BLINK_PATTERN	float[2]	{0.5,0.5}

Table 2 – GraphicStyle Elements

6.2.2.4 Graphic-to-GraphicStyle element applicability

The following `Graphic` classes are decorated by the given `GraphicStyle` name categories:

- `Graphic`: `Viewability`, `Editability` (optional), `Highlight` (optional), `Symbology` (optional).

- `GraphicCurveSegment` (opened), `GraphicArc` (opened), `GraphicCompositeCurve` (opened): `LineSymbolizer`.

- `GraphicCurveSegment` (closed), `GraphicArc` (closed), `GraphicCompositeCurve` (closed), `GraphicRing`, `GraphicSurfaceBoundary`: `PolygonSymbolizer`.

- `GraphicLabel`: `TextSymbolizer`.

- `GraphicIcon`: `PointSymbolizer`.

6.2.2.5 GraphicStyle inheritance

When a `Graphic` object is instantiated by a `DisplayFactory`, the `Graphic` object has a unique `GraphicStyle` object instantiated and associated with it.

A `GraphicStyle` object has all possible stylings, because if its associated `Graphic` aggregates other `Graphic` objects, those objects may inherit styling from the `GraphicStyle` object. Since the actual geometry of the aggregated `Graphics` are not known, `GraphicStyle` must carry all known types of styling information.

If the `GraphicStyle` element is not already set, and `GraphicStyle.INHERITANCE_INHERIT_STYLE_FROM_PARENT` is `true` (note the default value is `true`), then a `GraphicStyle` object will inherit a styling property value from its aggregating object.

A `Graphic` can force its aggregated `Graphic` objects to inherit its own styling by setting `GraphicStyle.INHERITANCE_OVERRIDE_AGGREGATED_GRAPHICS` to `true` (note that the default value is `false`). When `true`, this will force the aggregated Graphic objects to be rendered with the `GraphicStyle` of the aggregating `Graphic`, but will not change the values of the individual `GraphicStyle` objects corresponding to each of the aggregated `Graphic` objects. This case will override the behaviour of an aggregated `Graphic` with the setting `GraphicStyle.INHERITANCE_INHERIT_STYLE_FROM_PARENT` of `false`.

6.3 Spatial Objects

Spatial objects are those that contain geometric or location information. GO-1 utilizes spatial objects to provide this information to `Graphic` objects.

The material described in the present section is the focus of an ongoing, separate work item of the GO-1 Initiative, collectively referred to as the GeoAPI Project. Most of the Java packages and interface suites discussed here were developed directly from formal UML models, using automated tools, as part of an assessment and demonstration of the Model Driven Architecture (MDA) approach for specification and interface development.

6.3.1 Geometry

GO-1 supports a “simple geometry” profile of the robust model for geospatial geometry developed and published by the International Standards Organisation as ISO-19107 which was adopted with modifications in OGC Topic 1: Feature Geometry [5]. The ISO model provides an international standard for realizable geometry. This model has been implemented, with some minor changes, in the Open GIS Consortium Geographic Markup Language (GML) specification version 3.0 [12].

ISO-19107 is an all-inclusive model, intended to address the most demanding needs of a geospatial application. Many applications, in particular graphics subsystems, do not need the full capabilities of this model. The sections below identify the components of the ISO-19107 Geometry model that are the focus of GO-1.

GO-1 has adopted a subset of ISO-19107 Geometry for handling simple 0, 1 and 2 dimensional geometric primitives. The full semantic and detailed structures of these geometries are documented in the ISO-19107 specification. Context diagrams and brief descriptions of the geometries most relevant to GO-1 requirements are provided below.

Note: Except where noted, all descriptive text accompanying the context diagrams in this section is taken directly from [5].

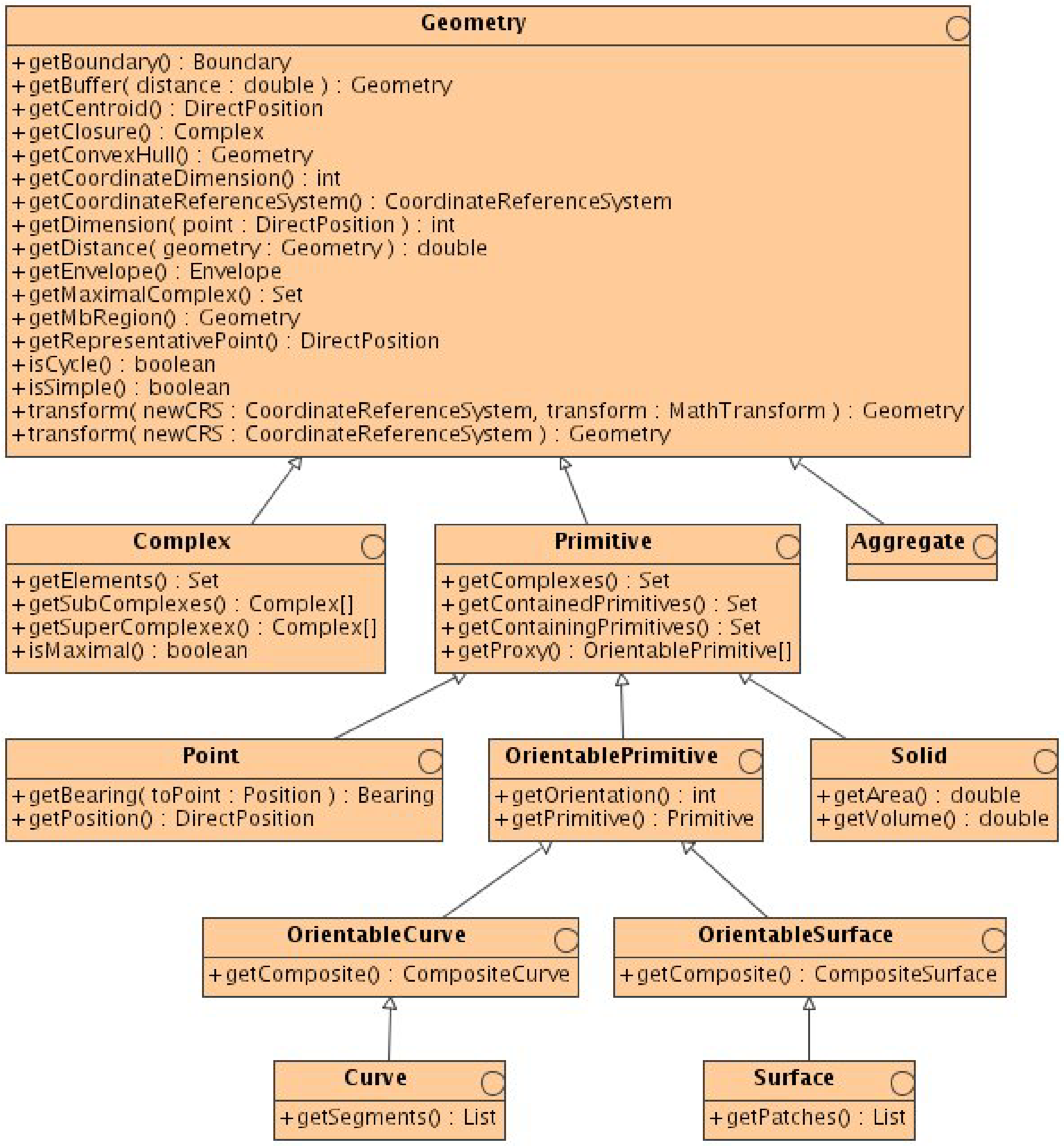

Geometry
+getBoundary() : Boundary
+getBuffer(distance : double) : Geometry
+getCentroid() : DirectPosition
+getClosure() : Complex
+getConvexHull() : Geometry
+getCoordinateDimension() : int
+getCoordinateReferenceSystem() : CoordinateReferenceSystem
+getDimension(point : DirectPosition) : int
+getDistance(geometry : Geometry) : double
+getEnvelope() : Envelope
+getMaximalComplex() : Set
+getMbRegion() : Geometry
+getRepresentativePoint() : DirectPosition
+isCycle() : boolean
+isSimple() : boolean
+transform(newCRS : CoordinateReferenceSystem, transform : MathTransform) : Geometry
+transform(newCRS : CoordinateReferenceSystem) : Geometry
Complex
+getElements() : Set
+getSubComplexes() : Complex[]
+getSuperComplexex() : Complex[]
+isMaximal() : boolean
Primitive
+getComplexes() : Set
+getContainedPrimitives() : Set
+getContainingPrimitives() : Set
+getProxy() : OrientablePrimitive[]
Aggregate
Point
+getBearing(toPoint : Position) : Bearing
+getPosition() : DirectPosition
OrientablePrimitive
+getOrientation() : int
+getPrimitive() : Primitive
Solid
+getArea() : double
+getVolume() : double
OrientableCurve
+getComposite() : CompositeCurve
OrientableSurface
+getComposite() : CompositeSurface
Curve
+getSegments() : List
Surface
+getPatches() : List

Figure 10 – Geometry top-level classes

The adjacent figure depicts the top-level Java interfaces of the GO-1 geometry model. These Java interfaces are generated directly from the ISO-19107 geometry models. These are the top-level interfaces for the key geometries that are the focus of GO-1. These interfaces are briefly described below.

`Geometry` is the root class of the geometric object taxonomy and supports interfaces common to all geographically referenced geometric objects. `Geometry` instances are sets of direct positions in a particular coordinate reference system. A `Geometry` can be regarded as an infinite set of points that satisfies the set operation interfaces for a set of direct positions, TransfiniteSet<DirectPosition>.

`Primitive` is the abstract root class of the geometric primitives. Its main purpose is to define the basic "boundary" operation that ties the primitives in each dimension together. A `Primitive` is a geometric object that is not decomposed further into other primitives in the system. This includes curves and surfaces, even though they are composed of curve segments and surface patches, respectively. This composition is a strong aggregation: curve segments and surface patches cannot exist outside the context of a primitive.

`Complex` is set of disjoint geometric primitives such that the boundary of each primitive can be represented as the union of other geometric primitives within the complex.

6.3.1.1 DirectPosition

`Point` is the basic data type for a geometric object consisting of one and only one point.

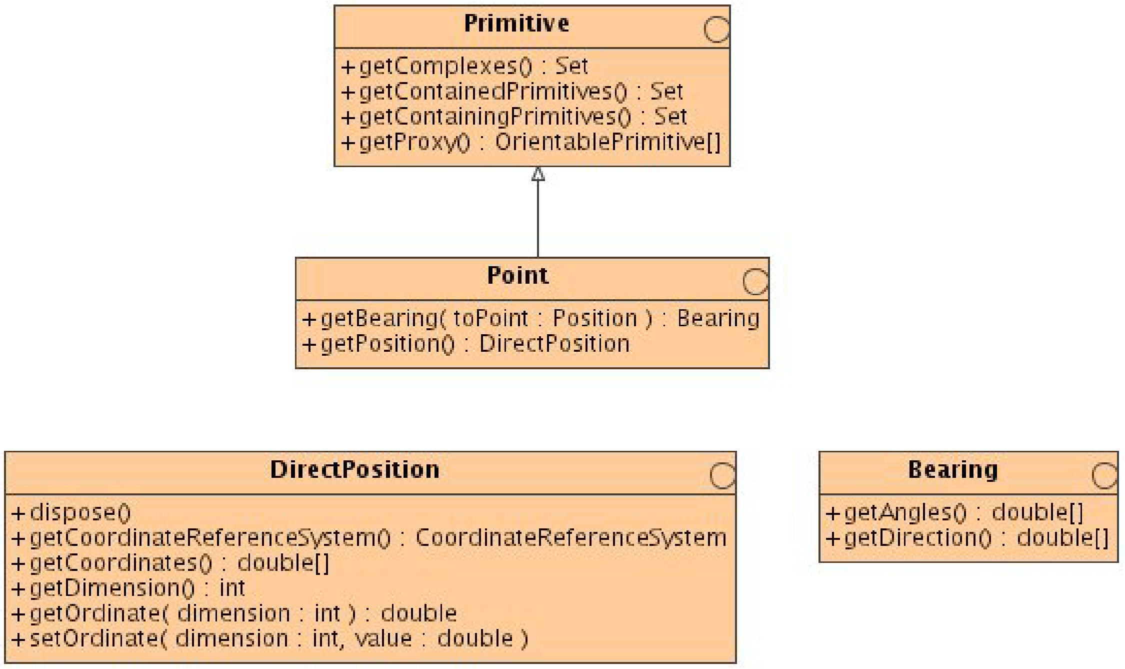

Primitive
+getComplexes() : Set
+getContainedPrimitives() : Set
+getContainingPrimitives() : Set
+getProxy() : OrientablePrimitive[]
Point
+getBearing(toPoint : Position) : Bearing
+getPosition() : DirectPosition
DirectPosition
+dispose()
+getCoordinateReferenceSystem() : CoordinateReferenceSystem
+getCoordinates() : double[]
+getDimension() : int
+getOrdinate(dimension : int) : double
+setOrdinate(dimension : int, value : double)
Bearing
+getAngles() : double[]
+getDirection() : double[]

Figure 11 – DirectPosition and Bearing

`DirectPosition` object data types hold the coordinates for a position within some `CoordinateReferenceSystem` (`CoordinateReferenceSystem` is described in Section 6.3.2). `DirectPositions`, as a data type, are utilized by in other objects, such as `Geometry`. When part of a `Geometry`, a `DirectPosition` will have the same `CoordinateReferenceSystem` as that `Geometry`.

`Bearing` is a data type used to represent direction in the coordinate reference system. In a 2D coordinate reference system, this can be accomplished using a "angle measured from true north" or a 2D vector point in that direction. In a 3D coordinate reference system, two angles or any 3D vector is possible. If both a set of angles and a vector are given, then they shall be consistent with one another.

6.3.1.2 CurveSegment and Conic

`Curve` is a descendent subtype of `Primitive` through `OrientablePrimitive`. It is the basis for 1-dimensional geometry. A curve is a continuous image of an open interval.

`Curve`s are continuous, connected, and have a measurable length in terms of the coordinate system. The orientation of the `Curve` is determined by this parameterization, and is consistent with the tangent function, which approximates the derivative function of the parameterization and shall always point in the "forward" direction.

A `Curve` is composed of one or more `CurveSegments`. Each `CurveSegment` within a `Curve` may be defined using a different interpolation method. The `CurveSegments` are connected to one another, with the end point of each segment except the last being the start point of the next segment in the segment list.

The `Conic` object represents any general conic curve, with the constraint that the eccentricity is less than unity. In two dimensions, this will generate a closed ellipse.

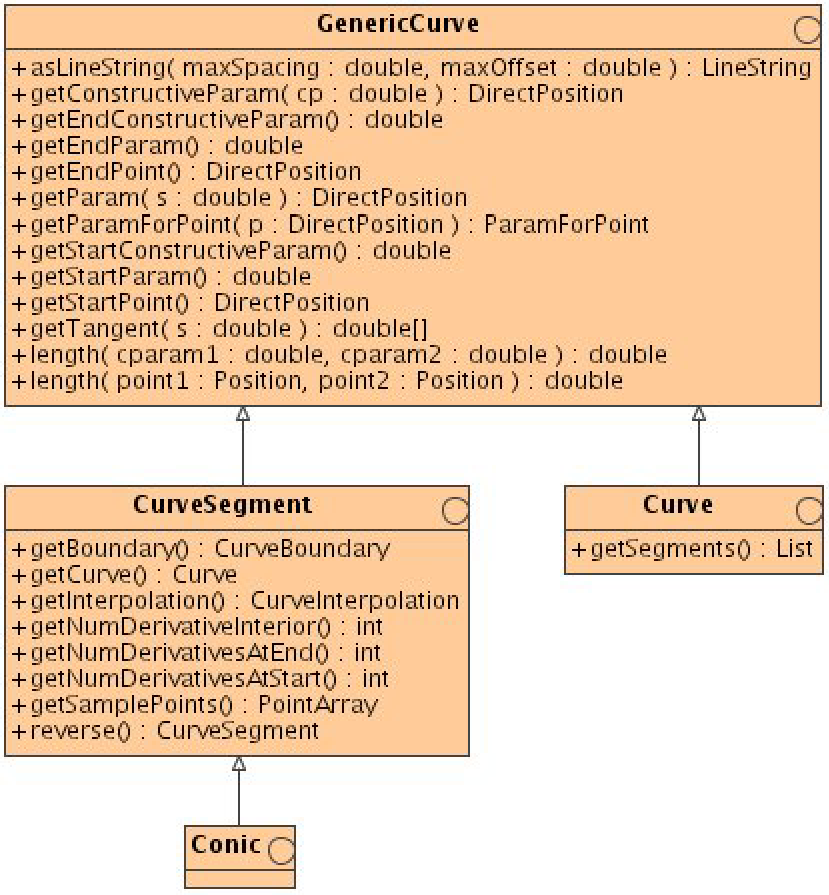

GenericCurve
+asLineString(maxSpacing : double, maxOffset : double) : LineString
+getConstructiveParam(cp : double) : DirectPosition
+getEndConstructiveParam() : double
+getEndParam() : double
+getEndPoint() : DirectPosition
+getParam(s : double) : DirectPosition
+getParamForPoint(p : DirectPosition) : ParamForPoint
+getStartConstructiveParam() : double
+getStartParam() : double
+getStartPoint() : DirectPosition
+getTangent(s : double) : double[]
+length(cparam1 : double, cparam2 : double) : double
+length(point1 : Position, point2 : Position) : double
CurveSegment
+getBoundary() : CurveBoundary
+getCurve() : Curve
+getInterpolation() : CurveInterpolation
+getNumDerivativeInterior() : int
+getNumDerivativesAtEnd() : int
+getNumDerivativesAtStart() : int
+getSamplePoints() : PointArray
+reverse() : CurveSegment
Curve
+getSegments() : List
Conic

Figure 12 – CurveSegment and Conic

6.3.1.3 CompositeCurve and Ring

A `CompositeCurve` is a `Composite` with all the geometric properties of a `Curve`. Essentially, a composite curve is a list of `OrientableCurve`s agreeing in orientation in a manner such that each curve (except the first) begins where the previous one ends.

A `Ring` is used to represent a single connected component of a `SurfaceBoundary`. It consists of a number of references to `OrientableCurves` connected in a cycle (an object whose boundary is empty).

A `Ring` is structurally similar to a `CompositeCurve` in that the endPoint of each `OrientedCurve` in the sequence is the startPoint of the next `OrientableCurve` in the sequence. Since the sequence is circular, there is no exception to this rule. Each ring, like all boundaries, is a cycle and does not intersect itself.

Even though each `Ring` is topologically simple, the boundary need not be simple. The easiest case of this is where one of the interior rings of a surface is tangent to its exterior ring. Implementations may enforce stronger restrictions on the interaction of boundary elements.

The basic difference between a `CompositeCurve` and a `Ring` is that a `CompositeCurve` may be *open* (the end of the last `OrientableCurve` does not touch the beginning of the first `OrientableCurve`) or *closed* (the end of the last `OrientableCurve` touches the beginning of the first `OrientableCurve`), however a `Ring` is always *closed*.

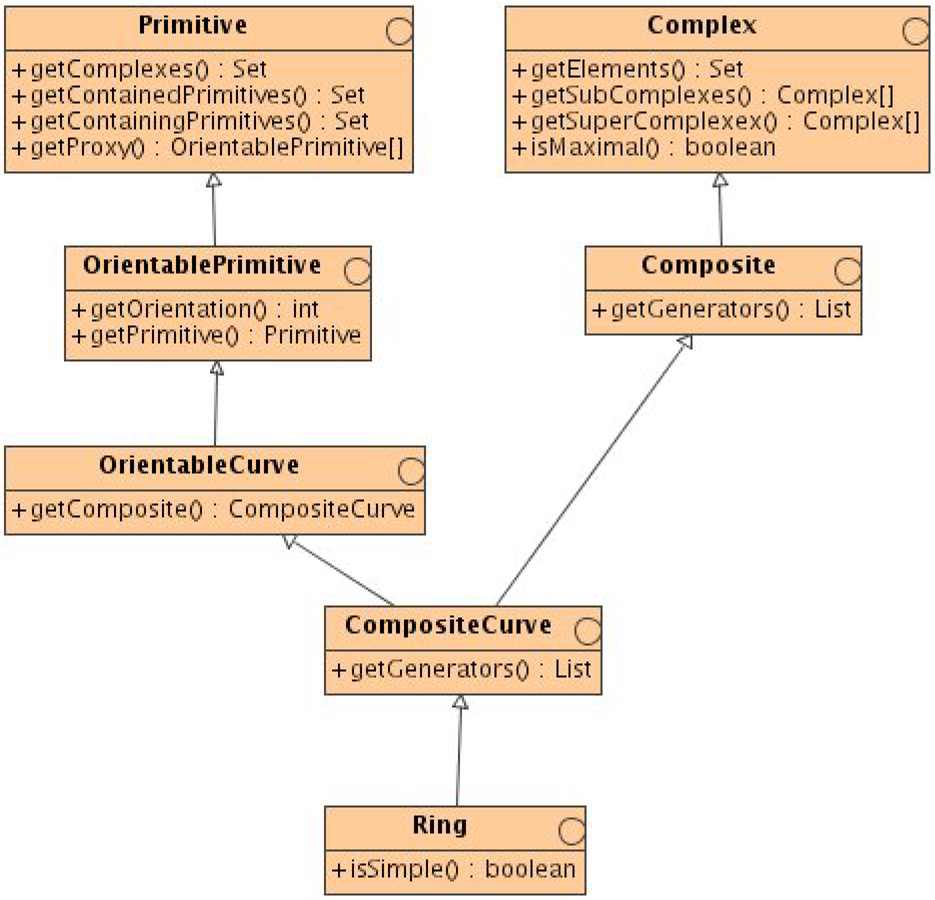


Figure 13 – CompositeCurve and Ring

6.3.1.4 SurfaceBoundary

A `SurfacePatch` defines a homogeneous portion of a `Surface`. The multiplicity of the segmentation association specifies that each `SurfacePatch` shall be in at most one `Surface`.

`Surface` is a subclass of `Primitive` and is the basis for 2-dimensional geometry. Unorientable surfaces such as the Möbius band are not allowed. The orientation of a surface chooses an "up" direction through the choice of the upward normal, which, if the surface is not a cycle, is the side of the surface from which the exterior boundary appears counterclockwise. Reversal of the surface orientation reverses the curve orientation of each boundary component, and interchanges the conceptual "up" and "down" direction of the surface. If the surface is the boundary of a solid, the "up" direction is usually outward. For closed surfaces, which have no boundary, the up direction is that of the surface patches, which must be consistent with one another. Its included `SurfacePatches` describe the interior structure of a `Surface`.

A `SurfaceBoundary` consists of a number of `Rings`, corresponding to the various components of its boundary. In the normal 2D case, one of the `Rings` is distinguished as being the exterior boundary. There is exactly one exterior `Ring` and zero or more interior `Rings`. None of the `Rings` may touch or intersect each other.

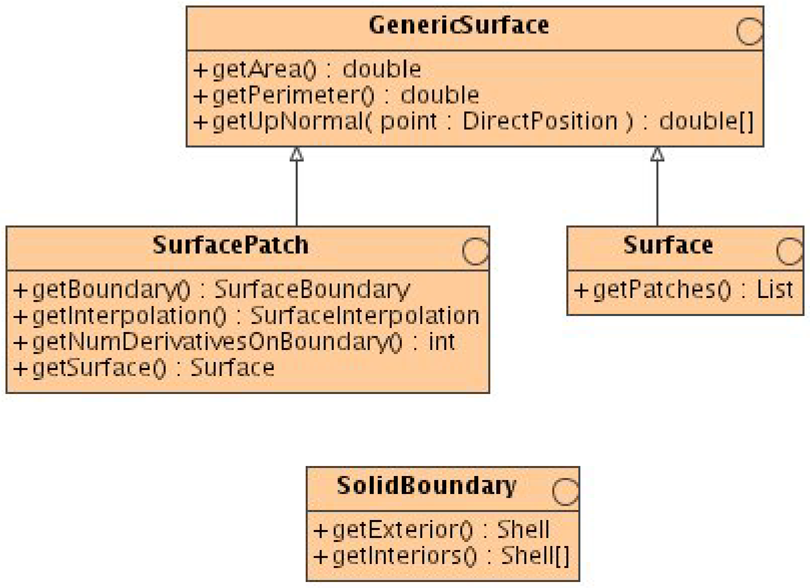

GenericSurface
+getArea() : double
+getPerimeter() : double
+getUpNormal(point : DirectPosition) : double[]
SurfacePatch
+getBoundary() : SurfaceBoundary
+getInterpolation() : SurfaceInterpolation
+getNumDerivativesOnBoundary() : int
+getSurface() : Surface
Surface
+getPatches() : List
SolidBoundary
+getExterior() : Shell
+getInteriors() : Shell[]

Figure 14 - SurfaceBoundary

6.3.1.5 Aggregate

Arbitrary aggregations of geometric objects are possible. These are not assumed to have any additional internal structure and are used to "collect" pieces of geometry of a specified type. Operations on these aggregations shall be the accumulators that are derived from the class operations of their elements. Applications may use aggregates for objects that use multiple geometric objects in their representations.

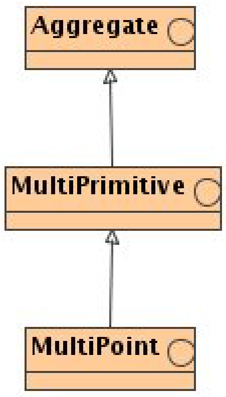


Figure 15 - Aggregate

The Aggregate gathers geometric objects. Since it will often use orientation modification, the curve reference and surface references do not go directly to the Curve and Surface, but are directed to OrientableCurve and OrientableSurface.

Most geometric objects cannot be held in collections that are strong aggregations. For this reason, the collections described in this clause are all weak aggregations, and shall use references to include geometric objects.

6.3.1.6 Envelope

Envelope is often referred to as a minimum bounding box or rectangle. Regardless of dimension, a Envelope can be represented without ambiguity as two DirectPositions. To encode a Envelope, it is sufficient to encode these two points. The lower corner refers to the DirectPosition whose coordinates are the minimum numeric values, and the upper corner refers to the DirectPosition whose coordinates are the maximum numeric values. The terms lower and upper should not be interpreted as spatially above and/or below.

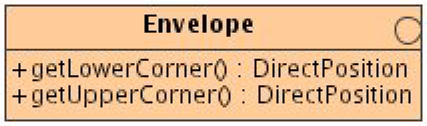

Envelope
+getLowerCorner() : DirectPosition
+getUpperCorner() : DirectPosition

Figure 16 - Envelope

6.3.2 Coordinate Reference System Model

The GO-1 Coordinate Reference System (CRS) definition is derived from and is fundamentally consistent with the content of OGC documents 03-009 and 03-010. The CRS interface, like those for other geometry interfaces, has been derived from UML models using automated tools. This process and the resulting interfaces are more completely described in the document that reports upon that effort.

Also, like the other Spatial Object classes, CRS objects are instantiated by a family of factories that hides the details of object creation from client applications or libraries.

Note: Except where noted, all descriptive text accompanying the context diagrams in this section is taken directly from [5] and [28].

6.3.2.1 Coordinate System

A CoordinateSystem is the set of coordinate system axes that spans a given coordinate space. A CoordinateSystem is derived from a set of (mathematical) rules for specifying how coordinates in a given space are to be assigned to points. The coordinate values in a coordinate tuple shall be recorded in the order in which the coordinate system axes associations are recorded, whenever those coordinates use a coordinate reference system that uses this coordinate system, and no other specification of axis order is provided.

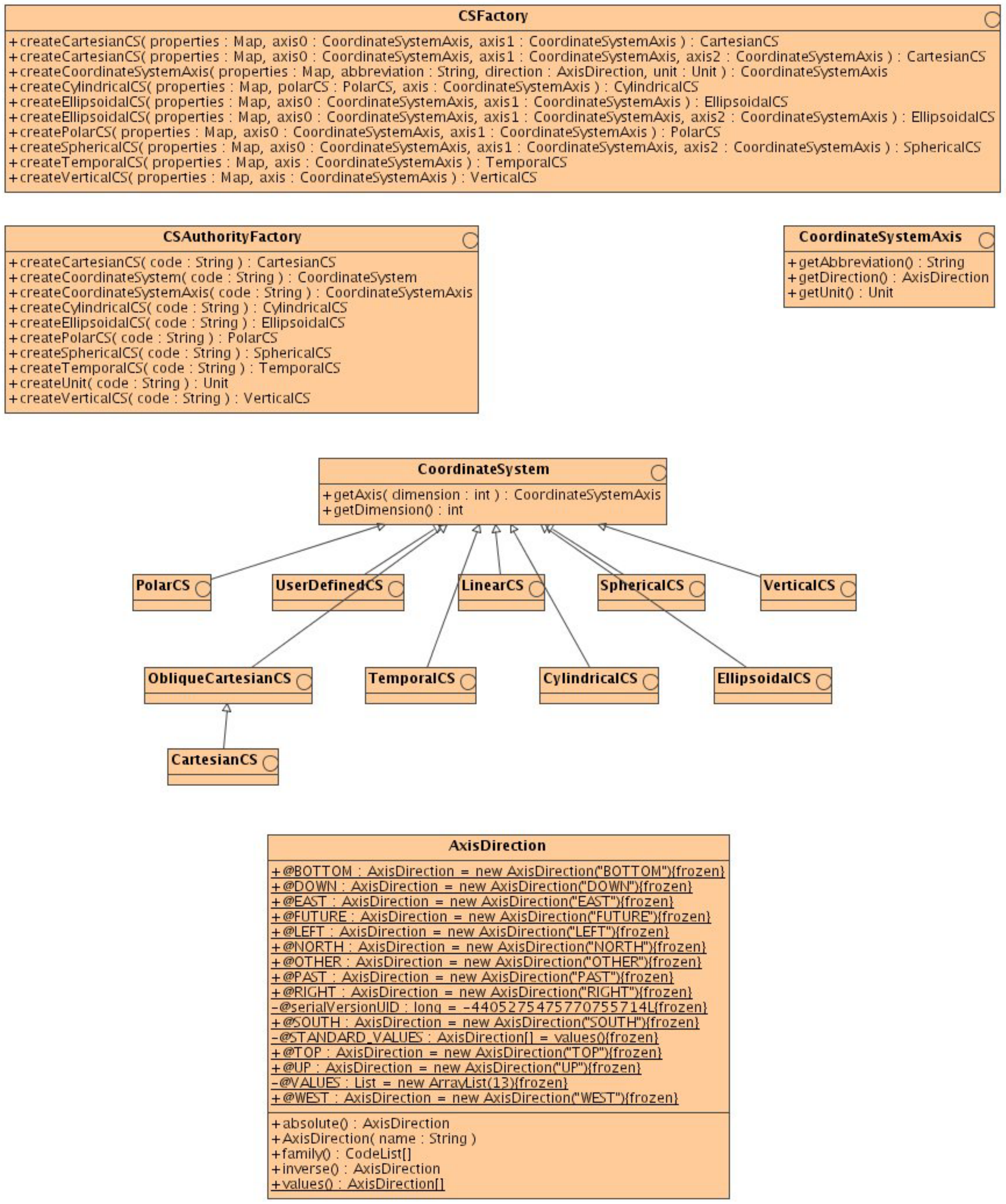

CSFactory
+createCartesianCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis) : CartesianCS
+createCartesianCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis, axis2 : CoordinateSystemAxis) : CartesianCS
+createCoordinateSystemAxis(properties : Map, abbreviation : String, direction : AxisDirection, unit : Unit) : CoordinateSystemAxis
+createCylindricalCS(properties : Map, polarCS : PolarCS, axis : CoordinateSystemAxis) : CylindricalCS
+createEllipsoidalCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis) : EllipsoidalCS
+createEllipsoidalCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis, axis2 : CoordinateSystemAxis) : EllipsoidalCS
+createPolarCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis) : PolarCS
+createSphericalCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis, axis2 : CoordinateSystemAxis) : SphericalCS
+createTemporalCS(properties : Map, axis : CoordinateSystemAxis) : TemporalCS
+createVerticalCS(properties : Map, axis : CoordinateSystemAxis) : VerticalCS
CSAuthorityFactory
+createCartesianCS(code : String) : CartesianCS
+createCoordinateSystem(code : String) : CoordinateSystem
+createCoordinateSystemAxis(code : String) : CoordinateSystemAxis
+createCylindricalCS(code : String) : CylindricalCS
+createEllipsoidalCS(code : String) : EllipsoidalCS
+createPolarCS(code : String) : PolarCS
+createSphericalCS(code : String) : SphericalCS
+createTemporalCS(code : String) : TemporalCS
+createUnit(code : String) : Unit
+createVerticalCS(code : String) : VerticalCS
CoordinateSystemAxis
+getAbbreviation() : String
+getDirection() : AxisDirection
+getUnit() : Unit
CoordinateSystem
+getAxis(dimension : int) : CoordinateSystemAxis
+getDimension() : int
PolarCS
UserDefinedCS
LinearCS
SphericalCS
VerticalCS
ObliqueCartesianCS
TemporalCS
CylindricalCS
EllipsoidalCS
CartesianCS
AxisDirection
+@BOTTOM : AxisDirection = new AxisDirection("BOTTOM"){frozen}
+@DOWN : AxisDirection = new AxisDirection("DOWN"){frozen}
+@EAST : AxisDirection = new AxisDirection("EAST"){frozen}
+@FUTURE : AxisDirection = new AxisDirection("FUTURE"){frozen}
+@LEFT : AxisDirection = new AxisDirection("LEFT"){frozen}
+@NORTH : AxisDirection = new AxisDirection("NORTH"){frozen}
+@OTHER : AxisDirection = new AxisDirection("OTHER"){frozen}
+@PAST : AxisDirection = new AxisDirection("PAST"){frozen}
+@RIGHT : AxisDirection = new AxisDirection("RIGHT"){frozen}
-@serialVersionUID : long = -4405275475770755714L{frozen}
+@SOUTH : AxisDirection = new AxisDirection("SOUTH"){frozen}
-@STANDARD_VALUES : AxisDirection[] = values(){frozen}
+@TOP : AxisDirection = new AxisDirection("TOP"){frozen}
+@UP : AxisDirection = new AxisDirection("UP"){frozen}
-@VALUES : List = new ArrayList(13){frozen}
+@WEST : AxisDirection = new AxisDirection("WEST"){frozen}
+absolute() : AxisDirection
+AxisDirection(name : String)
+family() : CodeList[]
+inverse() : AxisDirection
+values() : AxisDirection[]

Figure 17 - Coordinate System

`CartesianCS` defines a 1-, 2-, or 3-dimensional coordinate system. It gives the position of points relative to orthogonal straight axes in the 2- and 3-dimensional cases. In the 1-dimensional case, it contains a single straight coordinate axis. In the multidimensional case, all axes shall have the same length unit of measure. A CartesianCS shall have one, two, or three `usesAxis` associations.

`ObliqueCartesianCS` defines a 2- or 3-dimensional coordinate system with straight axes that are not necessarily orthogonal.

`EllipsoidalCS` defines a 2- or 3-dimensional coordinate system in which position is specified by geodetic latitude, geodetic longitude and (in the three-dimensional case) ellipsoidal height, associated with one or more geographic coordinate reference systems.

`SphericalCS` defines a 3-dimensional coordinate system with one distance, measured from the origin, and two angular coordinates. Not to be confused with an ellipsoidal coordinate system based on an ellipsoid 'degenerated' into a sphere

`CylindricalCS` defines a 3-dimensional coordinate system consisting of a polar coordinate system extended by a straight coordinate axis perpendicular to the plane spanned by the polar coordinate system.

`PolarCS` defines a 2-dimensional coordinate system in which position is specified by distance to the origin and the angle between the line from origin to point and a reference direction.

`VerticalCS` defines a 1-dimensional coordinate system used to record the heights (or depths) of points dependent on the Earth's gravity field.

`LinearCS` defines a 1-dimensional coordinate system that consists of the points that lie on the single axis described. The associated ordinate is the distance from the specified origin to the point along the axis. Example: usage of the line feature representing a road to describe points on or along that road.

`TemporalCS` defines a 1-dimensional coordinate system containing a single time axis and used to describe the temporal position of a point in the specified time units from a specified time origin.

`UserDefinedCS` defines a two- or three-dimensional coordinate system that consists of any combination of coordinate axes not covered by any other Coordinate System type. An example is a multi-linear coordinate system which contains one coordinate axis that may have any 1-D shape which has no intersections with itself. This non-straight axis is supplemented by one or two straight axes to complete a 2 or 3 dimensional coordinate system. The non-straight axis is typically incrementally straight or curved.

6.3.2.2 Reference System

`ReferenceSystem` provides a description of a spatial and temporal reference system used by a dataset.

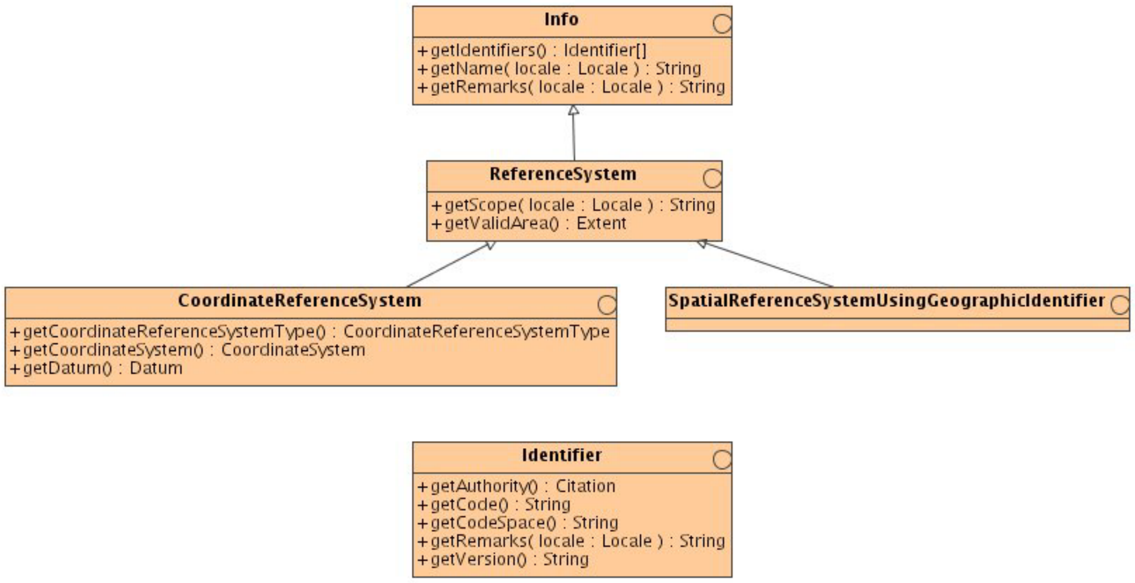

Info
+getIdentifiers() : Identifier[]
+getName(locale : Locale) : String
+getRemarks(locale : Locale) : String
ReferenceSystem
+getScope(locale : Locale) : String
+getValidArea() : Extent
CoordinateReferenceSystem
+getCoordinateReferenceSystemType() : CoordinateReferenceSystemType
+getCoordinateSystem() : CoordinateSystem
+getDatum() : Datum
SpatialReferenceSystemUsingGeographicIdentifier
Identifier
+getAuthority() : Citation
+getCode() : String
+getCodeSpace() : String
+getRemarks(locale : Locale) : String
+getVersion() : String

Figure 18 - Reference System

`Identifier` provides an identification of a reference system object. The first use of an `Identifier` for an object, if any, is normally the primary identification code, and any others are aliases.

6.3.2.3 Datum

`Datum` is commonly used to specify a relationship of a coordinate system to the earth, thus creating a coordinate reference system. A datum uses a parameter or set of parameters that determine the location of the origin, the orientation, and the scale of a coordinate reference system. The `anchorPoint` property of `Datum` is a description, possibly including coordinates, of the point or points used to anchor the datum to the Earth.

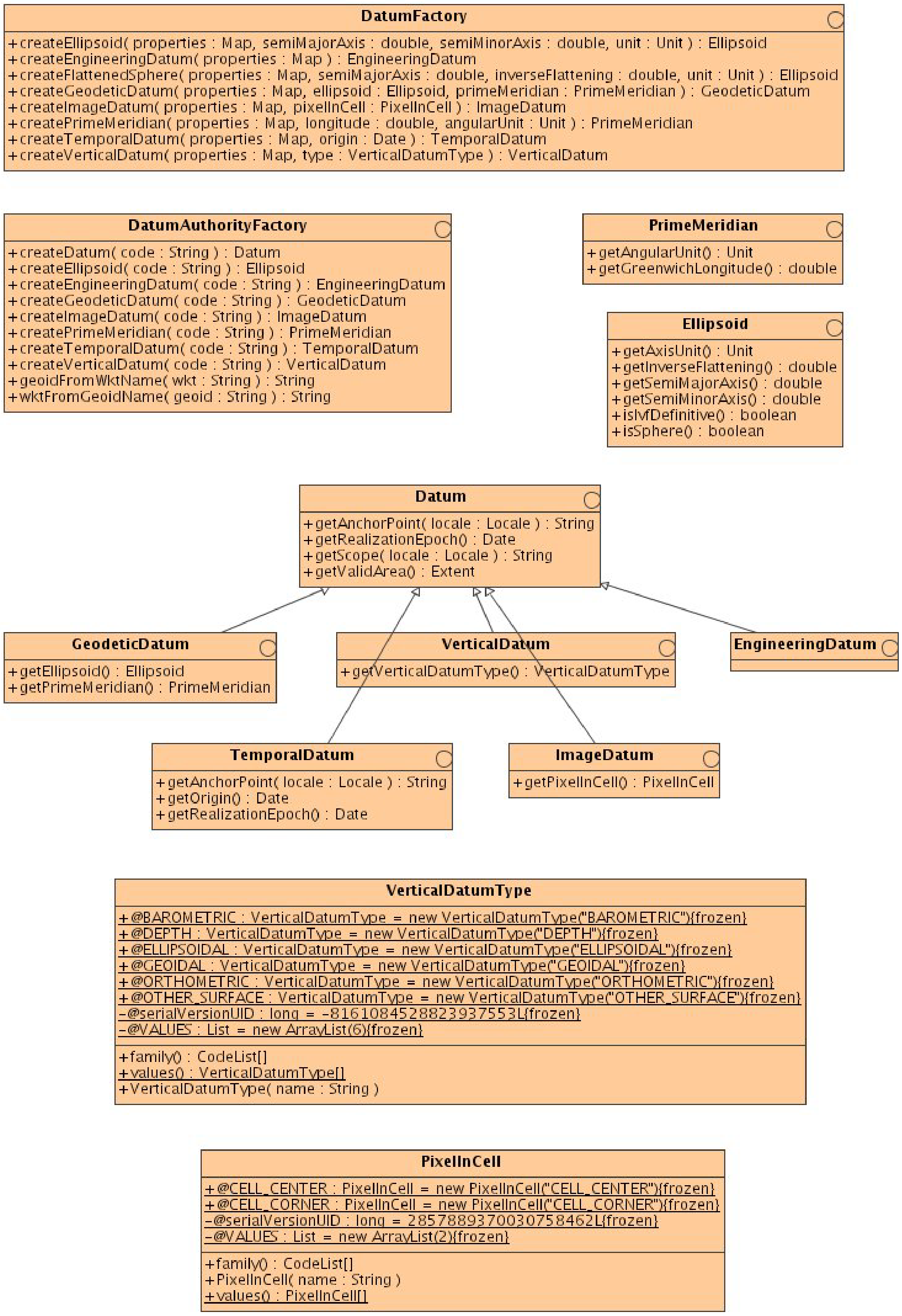

DatumFactory
+createEllipsoid(properties : Map, semiMajorAxis : double, semiMinorAxis : double, unit : Unit) : Ellipsoid
+createEngineeringDatum(properties : Map) : EngineeringDatum
+createFlattenedSphere(properties : Map, semiMajorAxis : double, inverseFlattening : double, unit : Unit) : Ellipsoid
+createGeodeticDatum(properties : Map, ellipsoid : Ellipsoid, primeMeridian : PrimeMeridian) : GeodeticDatum
+createImageDatum(properties : Map, pixelInCell : PixelInCell) : ImageDatum
+createPrimeMeridian(properties : Map, longitude : double, angularUnit : Unit) : PrimeMeridian
+createTemporalDatum(properties : Map, origin : Date) : TemporalDatum
+createVerticalDatum(properties : Map, type : VerticalDatumType) : VerticalDatum
DatumAuthorityFactory
+createDatum(code : String) : Datum
+createEllipsoid(code : String) : Ellipsoid
+createEngineeringDatum(code : String) : EngineeringDatum
+createGeodeticDatum(code : String) : GeodeticDatum
+createImageDatum(code : String) : ImageDatum
+createPrimeMeridian(code : String) : PrimeMeridian
+createTemporalDatum(code : String) : TemporalDatum
+createVerticalDatum(code : String) : VerticalDatum
+geoidFromWktName(wkt : String) : String
+wktFromGeoidName(geoid : String) : String
PrimeMeridian
+getAngularUnit() : Unit
+getGreenwichLongitude() : double
Ellipsoid
+getAxisUnit() : Unit
+getInverseFlattening() : double
+getSemiMajorAxis() : double
+getSemiMinorAxis() : double
+isIvfDefinitive() : boolean
+isSphere() : boolean
Datum
+getAnchorPoint(locale : Locale) : String
+getRealizationEpoch() : Date
+getScope(locale : Locale) : String
+getValidArea() : Extent
GeodeticDatum
+getEllipsoid() : Ellipsoid
+getPrimeMeridian() : PrimeMeridian
VerticalDatum
+getVerticalDatumType() : VerticalDatumType
EngineeringDatum
TemporalDatum
+getAnchorPoint(locale : Locale) : String
+getOrigin() : Date
+getRealizationEpoch() : Date
ImageDatum
+getPixelInCell() : PixelInCell
VerticalDatumType
+@BAROMETRIC : VerticalDatumType = new VerticalDatumType("BAROMETRIC"){frozen}
+@DEPTH : VerticalDatumType = new VerticalDatumType("DEPTH"){frozen}
+@ELLIPSOIDAL : VerticalDatumType = new VerticalDatumType("ELLIPSOIDAL"){frozen}
+@GEOIDAL : VerticalDatumType = new VerticalDatumType("GEOIDAL"){frozen}
+@ORTHOMETRIC : VerticalDatumType = new VerticalDatumType("ORTHOMETRIC"){frozen}
+@OTHER_SURFACE : VerticalDatumType = new VerticalDatumType("OTHER_SURFACE"){frozen}
-@serialVersionUID : long = -8161084528823937553L{frozen}
-@VALUES : List = new ArrayList(6){frozen}
+family() : CodeList[]
+values() : VerticalDatumType[]
+VerticalDatumType(name : String)
PixelInCell
+@CELL_CENTER : PixelInCell = new PixelInCell("CELL_CENTER"){frozen}
+@CELL_CORNER : PixelInCell = new PixelInCell("CELL_CORNER"){frozen}
-@serialVersionUID : long = 2857889370030758462L{frozen}
-@VALUES : List = new ArrayList(2){frozen}
+family() : CodeList[]
+PixelInCell(name : String)
+values() : PixelInCell[]

Figure 19 - Datum

`Ellipsoid` is a geometric figure that can be used to describe the approximate shape of the earth. In mathematical terms, it is a surface formed by the rotation of an ellipse about its minor axis.

`EngineeringDatum` defines the origin and axes directions of an engineering coordinate reference system. Normally used in a local context only.

`GeodeticDatum` references an `Ellipsoid` which models the shape of the earth. Due to irregularities in the surface of the Earth, some ellipsoids limit the portion of the earth's surface that can be accurately modelled. A `GeodeticDatum` references a `PrimeMeridian` that defines the origin from which longitude values are determined.

`ImageDatum` defines the origin of an image coordinate reference system. This is used in a local context only. For an image datum, the anchor point is usually either the centre of the image or the corner of the image.

`VerticalDatum` defines the surface of zero altitude. An example would be Mean Sea Level.

`TemporalDatum` defines the zero time for some epoch, accessed by getOrigin(). In Java, this would be Jan 1, 1970.

6.3.2.4 Coordinate Reference System

`CoordinateReferenceSystem` consists of an ordered sequence of coordinate system axes that are related to the earth (or another physical object) through a datum. A coordinate reference system is defined by one datum and by one coordinate system. Most coordinate reference systems do not move, except for `EngineeringCRS` objects defined with respect to moving platforms such as cars, ships, aircraft, and spacecraft. There are several sub-classes of `CoordinateReferenceSystem` (see figure below).

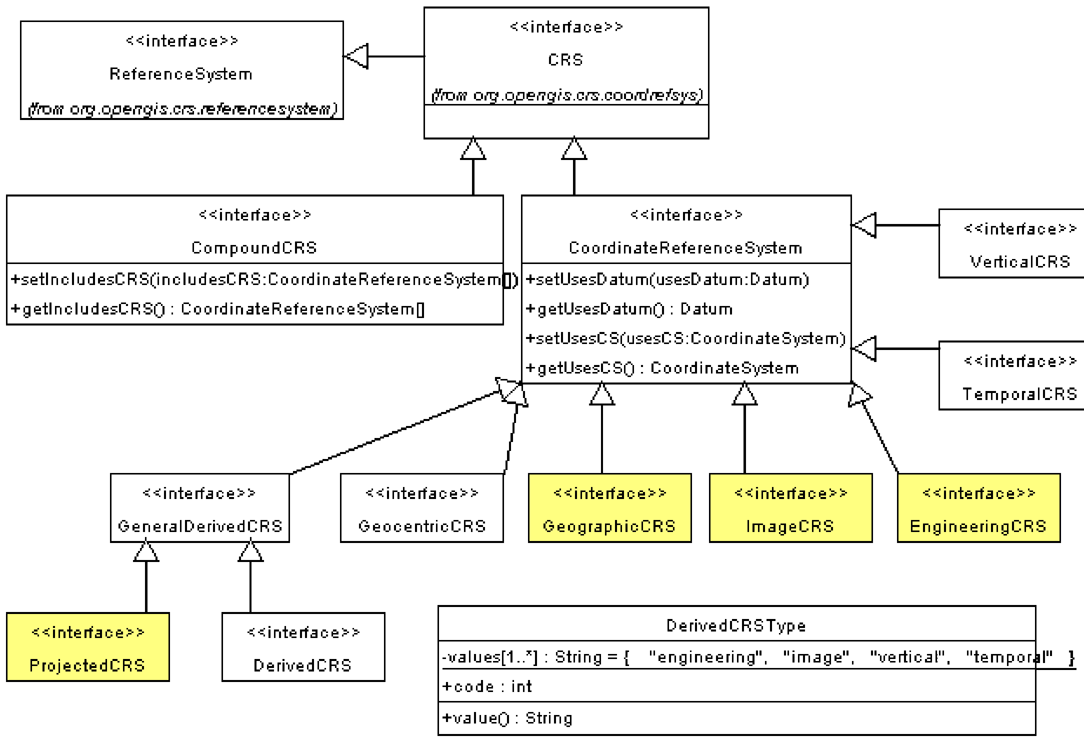


Figure 20 - Coordinate Reference System model from Topic 2

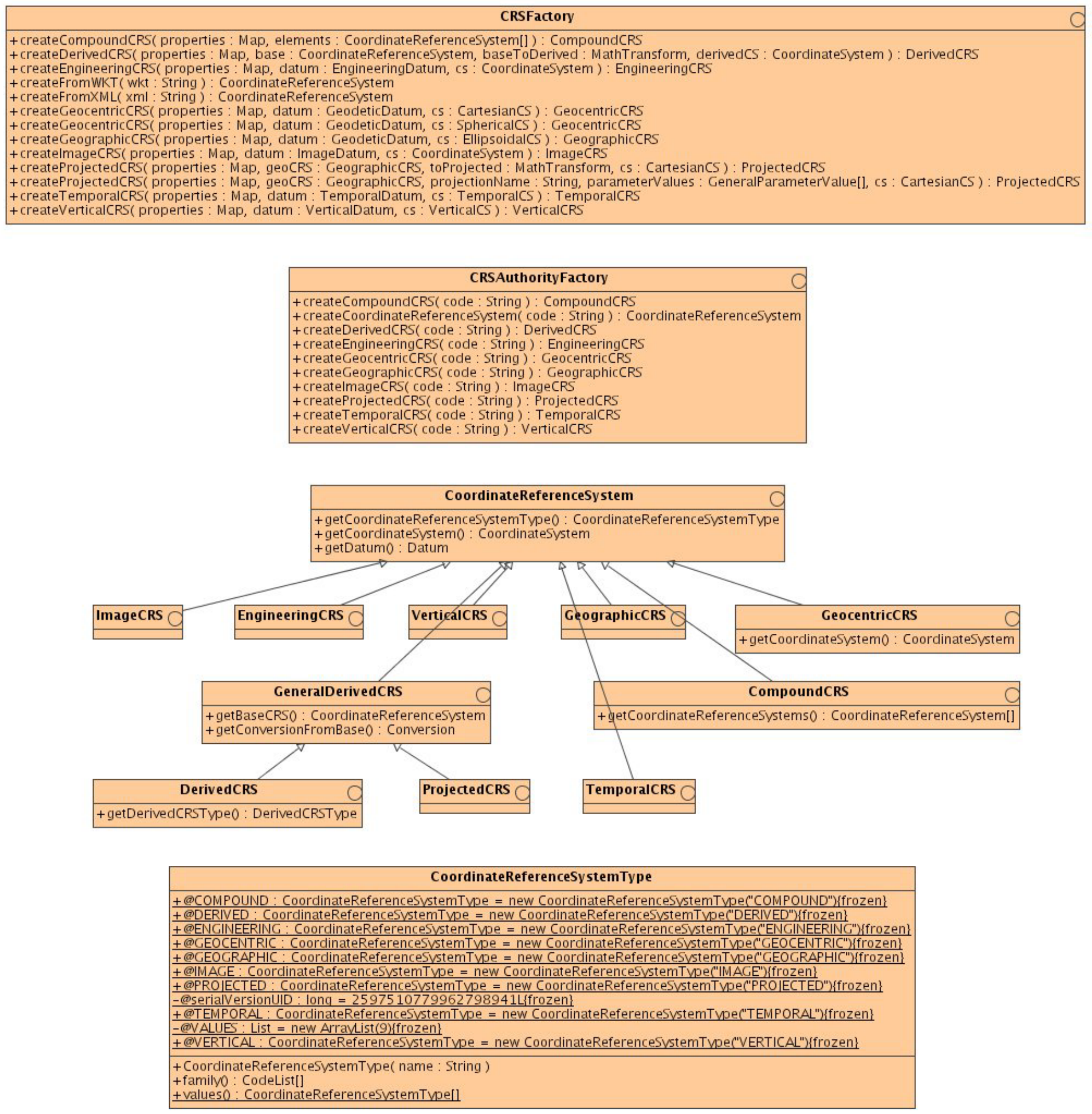

CRSFactory
+createCompoundCRS(properties : Map, elements : CoordinateReferenceSystem[]) : CompoundCRS
+createDerivedCRS(properties : Map, base : CoordinateReferenceSystem, baseToDerived : MathTransform, derivedCS : CoordinateSystem) : DerivedCRS
+createEngineeringCRS(properties : Map, datum : EngineeringDatum, cs : CoordinateSystem) : EngineeringCRS
+createFromWKT(wkt : String) : CoordinateReferenceSystem
+createFromXML(xml : String) : CoordinateReferenceSystem
+createGeocentricCRS(properties : Map, datum : GeodeticDatum, cs : CartesianCS) : GeocentricCRS
+createGeocentricCRS(properties : Map, datum : GeodeticDatum, cs : SphericalCS) : GeocentricCRS
+createGeographicCRS(properties : Map, datum : GeodeticDatum, cs : EllipsoidalCS) : GeographicCRS
+createImageCRS(properties : Map, datum : ImageDatum, cs : CoordinateSystem) : ImageCRS
+createProjectedCRS(properties : Map, geoCRS : GeographicCRS, toProjected : MathTransform, cs : CartesianCS) : ProjectedCRS
+createProjectedCRS(properties : Map, geoCRS : GeographicCRS, projectionName : String, parameterValues : GeneralParameterValue[], cs : CartesianCS) : ProjectedCRS
+createTemporalCRS(properties : Map, datum : TemporalDatum, cs : TemporalCS) : TemporalCRS
+createVerticalCRS(properties : Map, datum : VerticalDatum, cs : VerticalCS) : VerticalCRS
CRSAuthorityFactory
+createCompoundCRS(code : String) : CompoundCRS
+createCoordinateReferenceSystem(code : String) : CoordinateReferenceSystem
+createDerivedCRS(code : String) : DerivedCRS
+createEngineeringCRS(code : String) : EngineeringCRS
+createGeocentricCRS(code : String) : GeocentricCRS
+createGeographicCRS(code : String) : GeographicCRS
+createImageCRS(code : String) : ImageCRS
+createProjectedCRS(code : String) : ProjectedCRS
+createTemporalCRS(code : String) : TemporalCRS
+createVerticalCRS(code : String) : VerticalCRS
CoordinateReferenceSystem
+getCoordinateReferenceSystemType() : CoordinateReferenceSystemType
+getCoordinateSystem() : CoordinateSystem
+getDatum() : Datum
ImageCRS
EngineeringCRS
VerticalCRS
GeographicCRS
GeocentricCRS
+getCoordinateSystem() : CoordinateSystem
GeneralDerivedCRS
+getBaseCRS() : CoordinateReferenceSystem
+getConversionFromBase() : Conversion
CompoundCRS
+getCoordinateReferenceSystems() : CoordinateReferenceSystem[]
DerivedCRS
+getDerivedCRSType() : DerivedCRSType
ProjectedCRS
TemporalCRS
CoordinateReferenceSystemType
+@COMPOUND : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("COMPOUND"){frozen}
+@DERIVED : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("DERIVED"){frozen}
+@ENGINEERING : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("ENGINEERING"){frozen}
+@GEOCENTRIC : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("GEOCENTRIC"){frozen}
+@GEOGRAPHIC : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("GEOGRAPHIC"){frozen}
+@IMAGE : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("IMAGE"){frozen}
+@PROJECTED : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("PROJECTED"){frozen}
-@serialVersionUID : long = 2597510779962798941L{frozen}
+@TEMPORAL : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("TEMPORAL"){frozen}
-@VALUES : List = new ArrayList(9){frozen}
+@VERTICAL : CoordinateReferenceSystemType = new CoordinateReferenceSystemType("VERTICAL"){frozen}
+CoordinateReferenceSystemType(name : String)
+family() : CodeList[]
+values() : CoordinateReferenceSystemType[]

Figure 21 - Coordinate Reference System implementation in GO-1

For GO-1, the common `CoordinateReferenceSystem` subtypes of primary interest are:

- `ProjectedCRS` — A 2D coordinate reference system used to approximate the shape of the earth on a planar surface, but in such a way that the distortion that is inherent to the approximation is carefully controlled and known. Distortion correction is commonly applied to calculated bearings and distances to produce values that are a close match to actual field values.

- `GeographicCRS` — A coordinate reference system based on an ellipsoidal approximation of the geoid; this provides an accurate representation of the geometry of geographic features for a large portion of the earth's surface.

- `ImageCRS` — An engineering coordinate reference system applied to locations in images. Image coordinate reference systems are treated as a separate sub-type because a separate user community exists for images with its own terms of reference.

- `EngineeringCRS` — A contextually local coordinate reference system; which can be divided into two broad categories:
    1) Earth-fixed systems applied to engineering activities on or near the surface of the earth;
    2) CRSs on moving platforms such as road vehicles, vessels, aircraft, or spacecraft.

The GO-1 specification implements Topic 2 by combining the two abstract objects `SC_CRS` and `SC_CoordinateReferenceSystem` into a single interface `CoordinateReferenceSystem`. This allows the child interface `CompoundCRS` to hold instances of itself. Furthermore, an implementation can iterate over instances of `CoordinateReferenceSystem` without type-checking. The inherited methods `CompoundCRS.getDatum()` and `CompoundCRS.getCoordinateSystem()` `may` return null values for ComopoundCRS.

6.3.2.5 Map Projection

A map projection mediates the transformation of coordinates between the spatial `CoordinateReferenceSystem` and a corresponding flat representation. A `ProjectedCRS` maintains a `Projection` object that defines such a transformation process.

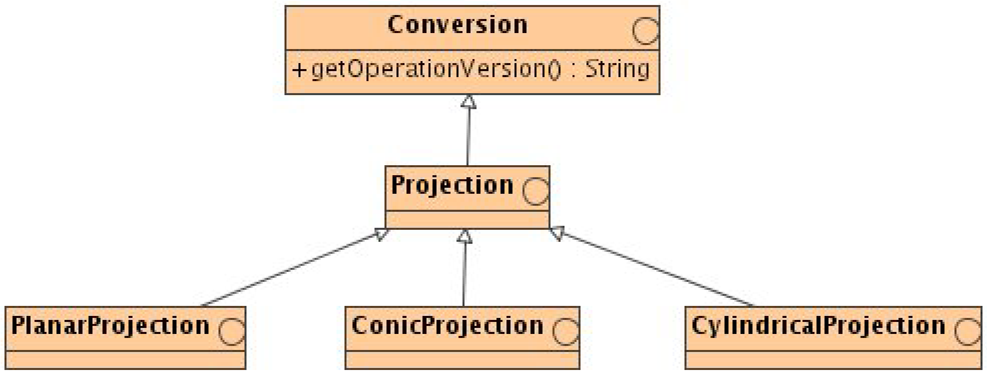


Figure 22 - Projection

Implementers may need to define concrete realizations of `Projection` for each supported projection, but details of how each will translate spatial coordinates to either display coordinates or intermediate grids are optional. Implementation of some projections will be mandatory, but most will be optional.

6.3.2.6 Coordinate Operations

`CoordinateOperation` represents a mathematical operation on coordinates that transforms or converts coordinates to another coordinate reference system. Many but not all coordinate operations (from CRS A to CRS B) uniquely define the inverse operation (from CRS B to CRS A). In some cases, the operation method algorithm for the inverse operation is the same as for the forward algorithm, but the signs of some operation parameter values must be reversed. In other cases, different algorithms are required for the forward and inverse operations, but the same operation parameter values are used. If (some) entirely different parameter values are needed, a different coordinate operation shall be defined.

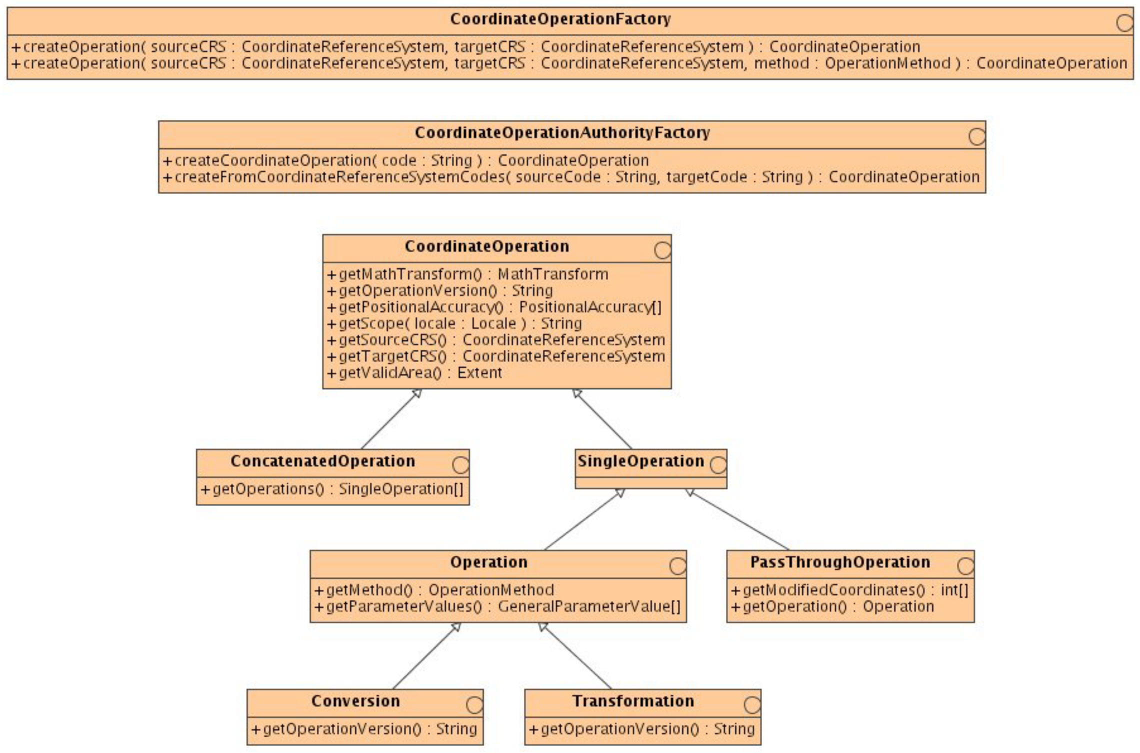

CoordinateOperationFactory
+createOperation(sourceCRS : CoordinateReferenceSystem, targetCRS : CoordinateReferenceSystem) : CoordinateOperation
+createOperation(sourceCRS : CoordinateReferenceSystem, targetCRS : CoordinateReferenceSystem, method : OperationMethod) : CoordinateOperation
CoordinateOperationAuthorityFactory
+createCoordinateOperation(code : String) : CoordinateOperation
+createFromCoordinateReferenceSystemCodes(sourceCode : String, targetCode : String) : CoordinateOperation
CoordinateOperation
+getMathTransform() : MathTransform
+getOperationVersion() : String
+getPositionalAccuracy() : PositionalAccuracy[]
+getScope(locale : Locale) : String
+getSourceCRS() : CoordinateReferenceSystem
+getTargetCRS() : CoordinateReferenceSystem
+getValidArea() : Extent
ConcatenatedOperation
+getOperations() : SingleOperation[]
SingleOperation
Operation
+getMethod() : OperationMethod
+getParameterValues() : GeneralParameterValue[]
PassThroughOperation
+getModifiedCoordinates() : int[]
+getOperation() : Operation
Conversion
+getOperationVersion() : String
Transformation
+getOperationVersion() : String

Figure 23 - Coordinate Operation

`Operation` is a parameterised mathematical operation on coordinates that transforms or converts coordinates to another coordinate reference system. This coordinate operation thus uses an operation method, usually with associated parameter values.

`Transformation` objects define an operation on coordinates that usually includes a change of `Datum`. They may also mediate conversion from a `ProjectedCRS` (which has a datum) to a flat screen. The parameters of a coordinate transformation are empirically derived from data containing the coordinates of a series of points in both coordinate reference systems. This computational process is usually "over-determined", allowing derivation of error (or accuracy) estimates for the transformation. Also, the stochastic nature of the parameters may result in multiple (different) versions of the same coordinate transformation.

`Conversion` objects define an operation on coordinates that does not include any change of Datum. The best-known example of a coordinate conversion is a map projection. The parameters describing coordinate conversions are defined rather than empirically derived. Note that some conversions have no parameters.

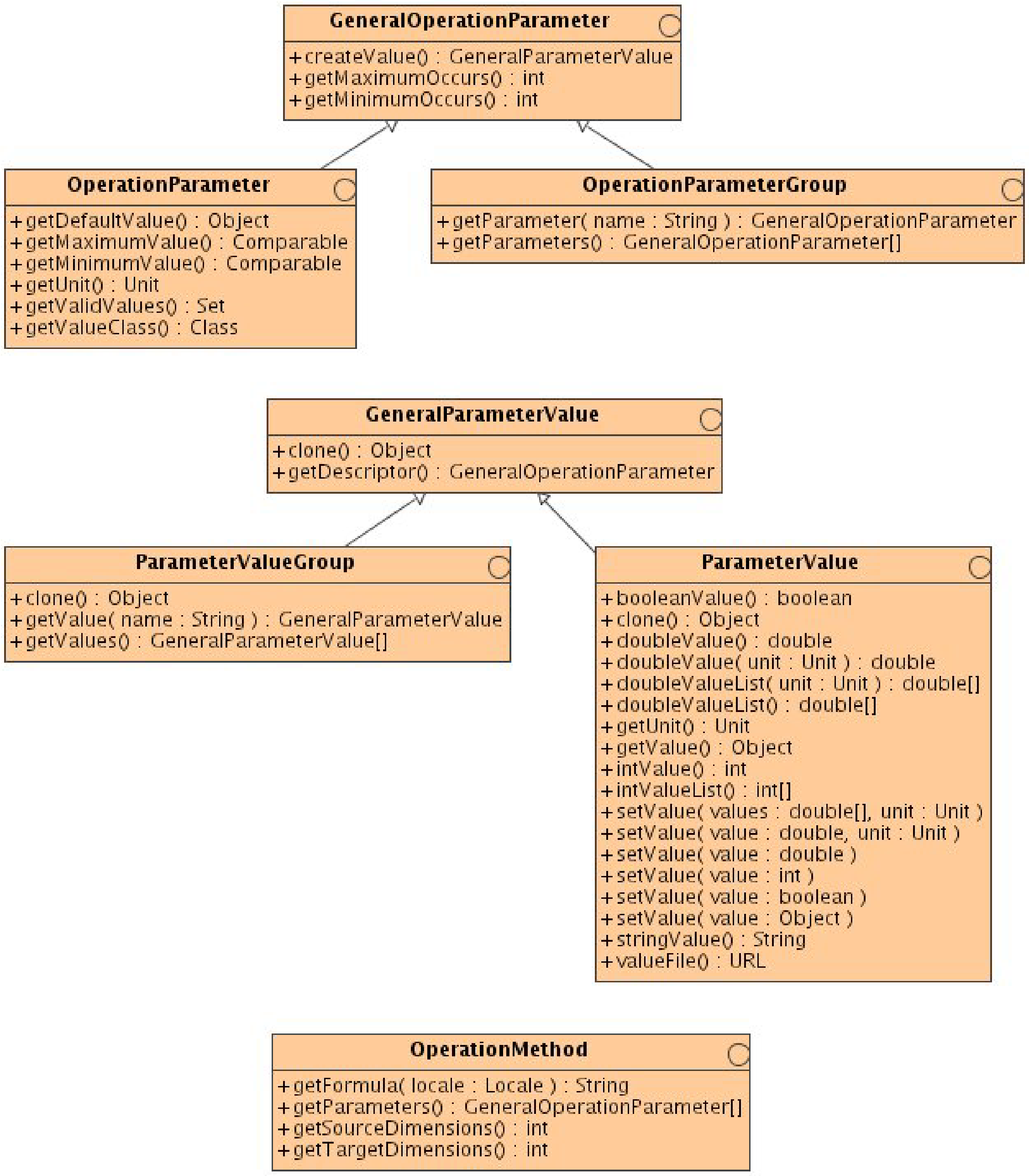

GeneralOperationParameter
+createValue() : GeneralParameterValue
+getMaximumOccurs() : int
+getMinimumOccurs() : int
OperationParameter
+getDefaultValue() : Object
+getMaximumValue() : Comparable
+getMinimumValue() : Comparable
+getUnit() : Unit
+getValidValues() : Set
+getValueClass() : Class
OperationParameterGroup
+getParameter(name : String) : GeneralOperationParameter
+getParameters() : GeneralOperationParameter[]
GeneralParameterValue
+clone() : Object
+getDescriptor() : GeneralOperationParameter
ParameterValueGroup
+clone() : Object
+getValue(name : String) : GeneralParameterValue
+getValues() : GeneralParameterValue[]
ParameterValue
+booleanValue() : boolean
+clone() : Object
+doubleValue() : double
+doubleValue(unit : Unit) : double
+doubleValueList(unit : Unit) : double[]
+doubleValueList() : double[]
+getUnit() : Unit
+getValue() : Object
+intValue() : int
+intValueList() : int[]
+setValue(values : double[], unit : Unit)
+setValue(value : double, unit : Unit)
+setValue(value : double)
+setValue(value : int)
+setValue(value : boolean)
+setValue(value : Object)
+stringValue() : String
+valueFile() : URL
OperationMethod
+getFormula(locale : Locale) : String
+getParameters() : GeneralOperationParameter[]
+getSourceDimensions() : int
+getTargetDimensions() : int

Figure 24 - Operation Parameter

`OperationParameter` is the definition of a parameter used by an operation method. Most parameter values are numeric, but other types of parameter values are possible.

`OperationMethod` is the definition of an algorithm used to perform a coordinate operation. Most operation methods use a number of operation parameters, although some coordinate conversions use none. Each coordinate operation using the method assigns values to these parameters.

The `MathTransform` object does the work of applying formulae to coordinate values. A `MathTransform` does not know or care how the coordinates relate to positions in the real world. `MathTransform` objects are intended to be generic in nature; they may be agnostic to the spatial-coordinate domain, and may be equally applicable to non-spatial-coordinate domains.

A `CoordinateOperation` contains a source `CoordinateReferenceSystem`, a target `CoordinateReferenceSystem`, and a `MathTransform`. The `MathTransform` transforms from the source coordinate values to the target coordinate values.

`CoordinateOperation` exposes to a user the operation, allowing a user to analyse the operation from a spatial coordinate context. `MathTransform` is the backend implementation of an operation, but has no provision for user analysis.

`MathTransform` objects consisting of algorithms (or chains of algorithms) that have identical inputs and identical outputs are themselves interchangeable. Substituting a `MathTransform` object with an interchangeable `MathTransform` object will not affect the behaviour of the containing `CoordinateOperation`. An implementation is allowed to do so if deemed desirable.

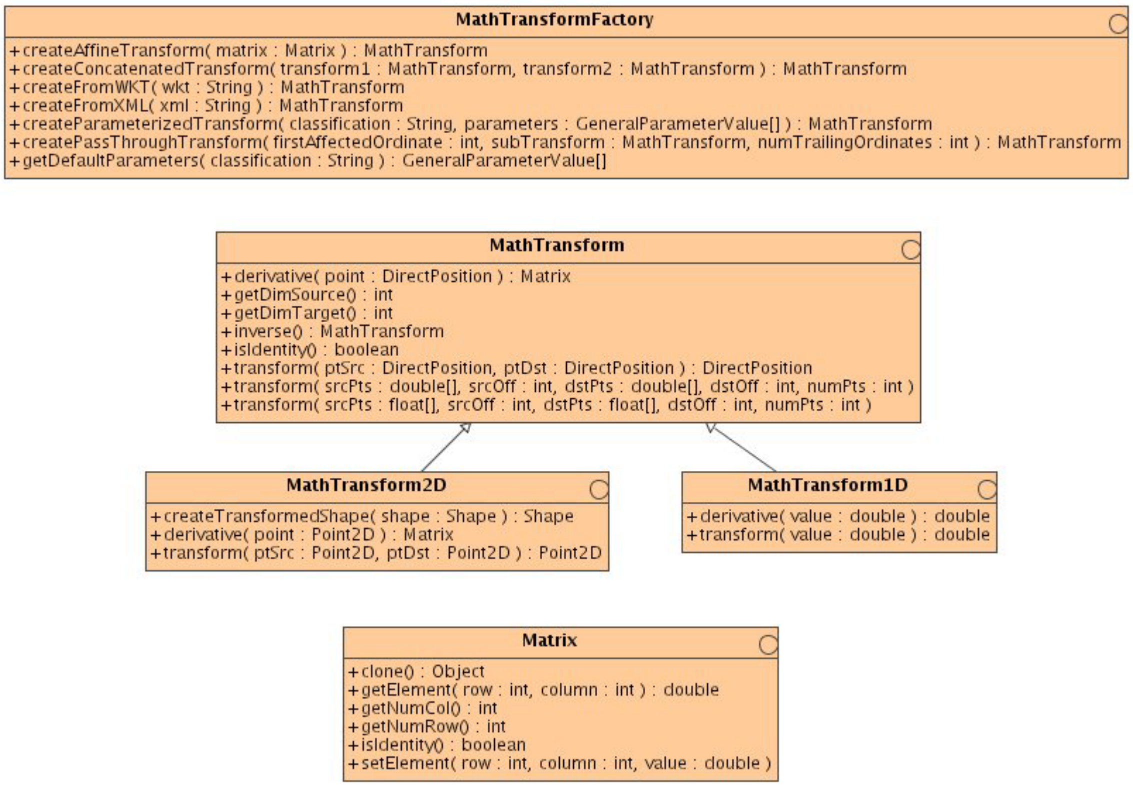

MathTransformFactory
+createAffineTransform(matrix : Matrix) : MathTransform
+createConcatenatedTransform(transform1 : MathTransform, transform2 : MathTransform) : MathTransform
+createFromWKT(wkt : String) : MathTransform
+createFromXML(xml : String) : MathTransform
+createParameterizedTransform(classification : String, parameters : GeneralParameterValue[]) : MathTransform
+createPassThroughTransform(firstAffectedOrdinate : int, subTransform : MathTransform, numTrailingOrdinates : int) : MathTransform
+getDefaultParameters(classification : String) : GeneralParameterValue[]
MathTransform
+derivative(point : DirectPosition) : Matrix
+getDimSource() : int
+getDimTarget() : int
+inverse() : MathTransform
+isIdentity() : boolean
+transform(ptSrc : DirectPosition, ptDst : DirectPosition) : DirectPosition
+transform(srcPts : double[], srcOff : int, dstPts : double[], dstOff : int, numPts : int)
+transform(srcPts : float[], srcOff : int, dstPts : float[], dstOff : int, numPts : int)
MathTransform2D
+createTransformedShape(shape : Shape) : Shape
+derivative(point : Point2D) : Matrix
+transform(ptSrc : Point2D, ptDst : Point2D) : Point2D
MathTransform1D
+derivative(value : double) : double
+transform(value : double) : double
Matrix
+clone() : Object
+getElement(row : int, column : int) : double
+getNumCol() : int
+getNumRow() : int
+isIdentity() : boolean
+setElement(row : int, column : int, value : double)

Figure 25 - MathTransform

6.3.2.6.1 Required Coordinate Transformations

GO-1 implementations are required to support the following coordinate transformations:

- Geocentric to Geocentric
- Geocentric to Geographic
- Geographic to Geocentric
- Geographic to Geographic
- Geocentric or Geographic to Projected
- Projected to Geocentric or Geographic
- Engineering to Geocentric or Geographic
- Geocentric or Geographic to Engineering

6.3.2.6.2 Required Operation Methods

GO-1 implementations are required to support the following operation methods:

- Molodenski Transform (7 parameter).

- Abridged Molodenski Transform (7 parameter).
- Geocentric Translation (3 parameter).
- Helmert Transform (7 parameter, with identifiers for Position Vector and Coordinate Frame Rotation variants).
- Affine Transform 2D.
- Polynomial Transform (described by NIMA TR 8350.2).

GO-1 implementations may optionally support the following operation methods (for conversions and transformations):

- Grid-Based Transform (NADCON and NTv2).

6.3.2.6.3 Required Datum

GO-1 implementations are required to support the following `Datum`:

- WGS-84 World Geographic Survey 1984.

6.3.2.7 Relative Coordinates

A technique exists to support relative coordinates. This technique can only be used in GO-1 implementations that support the restriction that all `DirectPositions` within a `Geometry` have the same `CoordinateReferenceSystem`.

The technique proposed to accomplish this uses `Geometry`, which always has a current `CoordinateReferenceSystem`. The method `Geometry.transform( CoordinateReferenceSystem, MathTransform)`, returns another `Geometry` instance in the given `CoordinateReferenceSystem` transformed from the first `Geometry` instance using the given `MathTransform`.

The original `Geometry` has a reference to the new `Geometry`, which has a reference to the new `CoordinateReferenceSystem`. Thus a Geometric object can effectively "move" to any given `CoordinateReferenceSystem`.

If `Geometry` G_A in `CoordinateReferenceSystem` A desires to transform to a particular target `CoordinateReferenceSystem` C, but only has an intermediate `CoordinateReferenceSystem` B and `MathTransforms` A-to-B and B-to-C, the methods `MathTransformFactory.createConcatenatedTransform( MathTransform, MathTransform)` and `MathTransformFactory.createPassThroughTransform(int, MathTransform, int)` can be utilised to create `MathTransform` A-to-C, and thereby eliminate the need to instantiate the intermediate `Geometry` G_B object.

6.3.3 Reference System Factories and Authority Factories

The GO-1 specification for CoordinateReferenceSystem, CoordinateSystem, Datum, and Operation has a layered factory pattern consisting of a Factory and one or more implementations of an AuthorityFactory. CRSFactory, CSFactory, DatumFactory create objects using a properties java.util.Map object for many of the required parameters. The CRSAuthorityFactory, CSAuthorityFactory, DatumAuthorityFactory, and CoordinateOperationAuthorityFactory objects are intended to connect to real-world authority databases, such as the European Petroleum Survey Group (EPSG) or the International Hydrographic Organization (IHO). Each CRSAuthorityFactory (for example) wraps the CRSFactory and delegates implementation-specific creation tasks.

For information on the CoordinateReferenceSystems, CoordinateSystems, Projections, and Datums supported by an implementation, one may query its CommonFactory. This object also provides supported Geometry types.

All reference system objects are immutable once created. For ProjectedCRS and other GeneralDerivedCRS objects this presents a complication, in that the Conversion returned by *getConversionFromBase* must always return the same object, and the CoordinateOperationFactory *createOperation* methods require that the ProjectedCRS be passed in as a parameter. Thus, one may not pass the required Projection object into the ProjectedCRS through its constructor, and the ProjectedCRS may not hold a pointer to the Projection. Instead, the CoordinateOperationFactory must create the Projection the first time a call is made to *createOperation* (passing in at least the ProjectedCRS and its base GeographicCRS). It will then store this Projection (typically within a Map) for retrieval on subsequent calls to *createOperation* in which the same two CoordinateReferenceObjects plus the relevant OperationMethod are passed in.

It follows from the creation mechanism that implementations of ProjectedCRS will need to define get methods for internal use in order to pass parameters to the CoordinateOperationFactory. However, in general CoordinateReferenceSystem objects do not expose their properties directly, but rather through Operation objects that involve them. To obtain access to parameters for a ProjectedCRS via the GO-1 API, one obtains the Projection via *getConversionFromBase*, queries it for its ParameterValues, examines the array for the ParameterValue corresponding to the attribute of interest, then queries that ParameterValue for its actual value.

7 Behaviours

Here we illustrate a few signature behaviours of an application that uses a GO-1 implementation. We present these behaviours as use cases, some accompanied by sequence or state diagrams.

7.1 Adding a Graphic to a display

Description: Create a graphic and add it to the display.

Precondition: Begin with an application that includes a full implementation of GO-1 Application Objects. All required Factory objects and a Canvas object have been instantiated, and a Graphic is ready to be added to the display.

Flow of events:

1. Application requests a Graphic object from the DisplayFactory.
2. Application sets the geometric attributes of the Graphic
3. Application sets the style attributes of the Graphic via getGraphicStyle()
4. Application adds the Graphic object to the Canvas object.

Postcondition: The Graphic is rendered with requested styling on the display device.

This sequence of operations is depicted below.

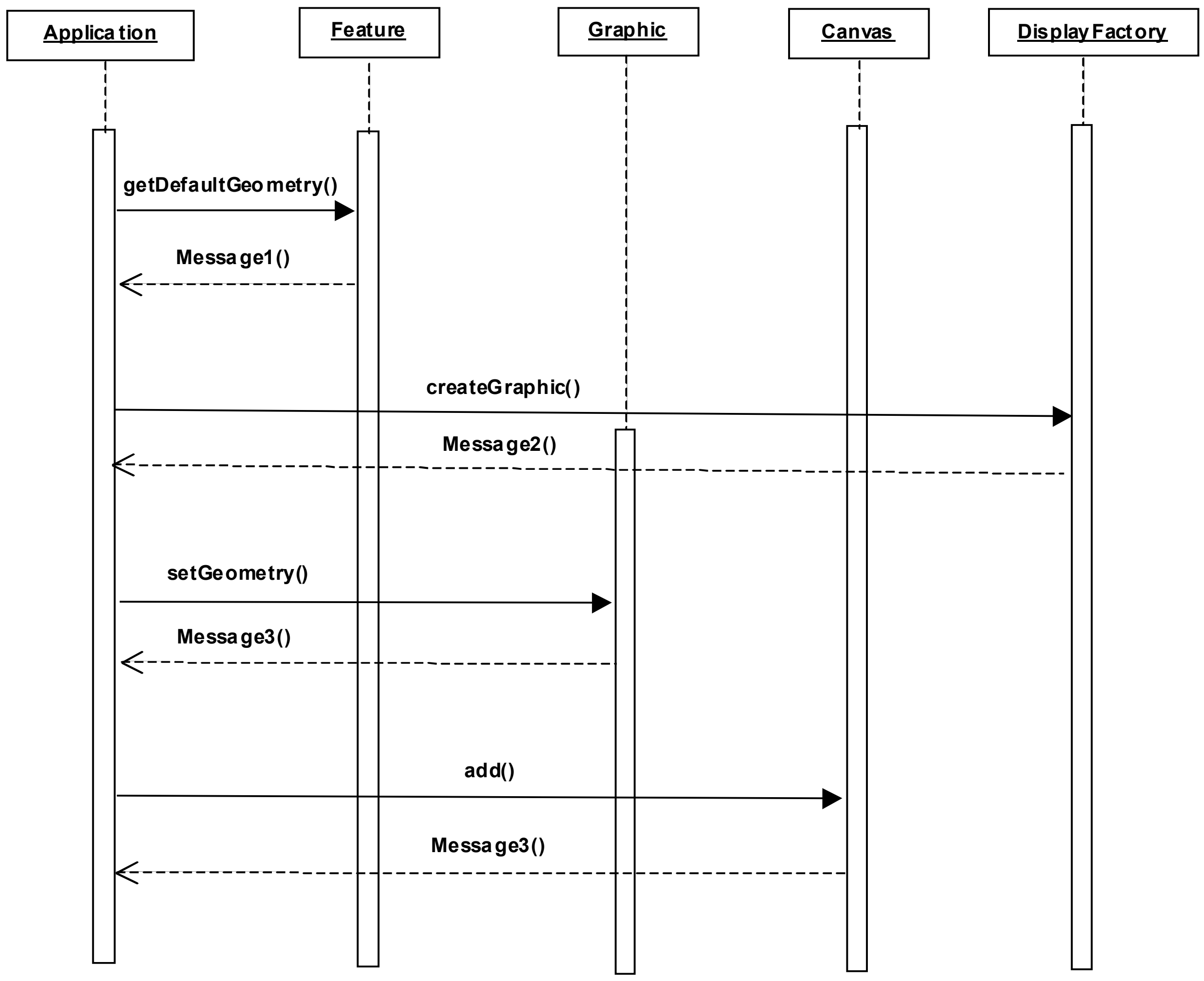


Figure 26 – Generalized Adding Information to a Display

7.2 Mouse click selects graphical object.

Description: A user selects a feature for editing in the graphical display.

Preconditions: Begin with an application that includes a full implementation of GO-1 Application Objects. The Canvas has a MouseManagerSupport object to which it delegates mouse event operations. A SelectItemsHandler class exists that implements MouseHandler. The MouseManagerSupport object has been registered as a MouseListener and a MouseMotionListener, and the SelectItemsHandler has been pushed onto the MouseManagerSupport's (empty) MouseHandler stack. (Even though SelectItemsHandler is a Java Listener, it is not registered with any EventSource. It is used as an event dispatcher.)

Flow of events:

1. User clicks on the `Canvas`, causing a `MouseEvent` to be fired.

2. The `MouseManagerSupport` receives the `MouseEvent`, and passes the `MouseEvent` to the first and only item on its `MouseHandler` stack.

3. The `MouseEvent` is received and consumed by `SelectItemsHandler`

4. `SelectItemsHandler` acquires `GraphicStyle` from the selected Graphic and calls `GraphicStyle.setEditabilitySelected(true)` to set it selected.

Postcondition: The user sees the object displayed with styling indicating it has been selected,.

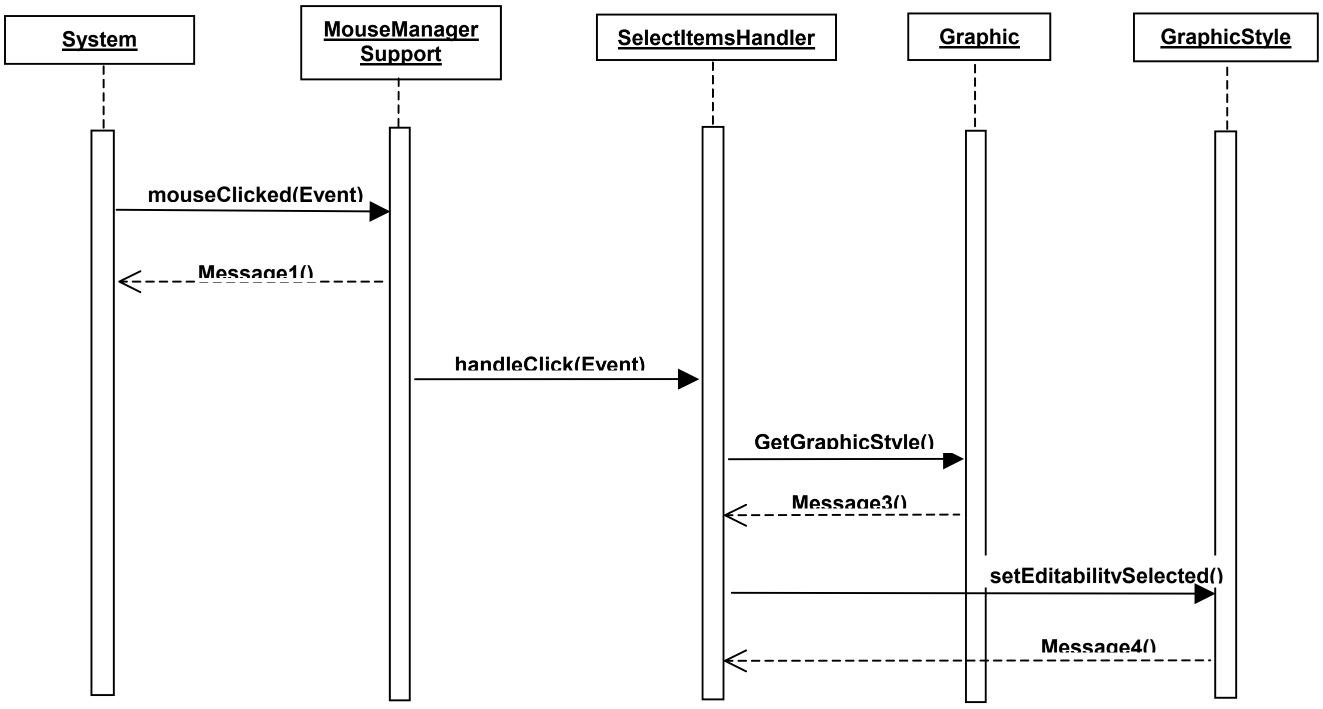


Figure 27 - Selecting a Graphic Object

7.2.1 Editing Graphics

Graphic objects purposefully leave editing up to the implementation. The two existing properties on Graphic, ShowingAnchorHandles and ShowingEditHandles, are the only hooks provided in the specification pertaining to editing. Their purpose is to offer a programmatic way of moving a Graphic to and from an editable mode determined by the implementation. An implementation may choose to edit a Graphic as it sees fit, but it is widely assumed that users will have the opportunity to modify a Graphic through gestures, particularly mouse gestures.

In order to move a Graphic into editing mode, one or both of the setShowingHandles methods must be called. If setShowingAnchorHandles is called, then the Graphic should display handles suitable for relocating the entire Graphic. Each Graphic should display editing handles for its particular Geometry, so a GraphicCurveSegment would display handles at its vertices while a GraphicIcon would only display a handle for rotating (the handle for relocating the GraphicIcon would be categorized as an anchor handle rather than an editing handle).

7.3 Graphic object is instantiated from a Geometry and an SLD.

Preconditions: Running application has instantiated a geometry CurveSegment and a compatible StyledLayerDescriptor (SLD) object.

Flow of events:

1. Application creates a new Graphic object with the DisplayFactory. Graphic has default styling.

2. Application gets the reference to the Graphic's GraphicStyle.

3. Application gets various styling attributes from the SLD.

4. Application sets the GraphicStyle's styling attributes with those obtained from the SLD.

5. Application sets the geometric attributes of the Graphic using the Geometry.

Postcondition: a styled Graphic has been created, and may be added to a Canvas for display.

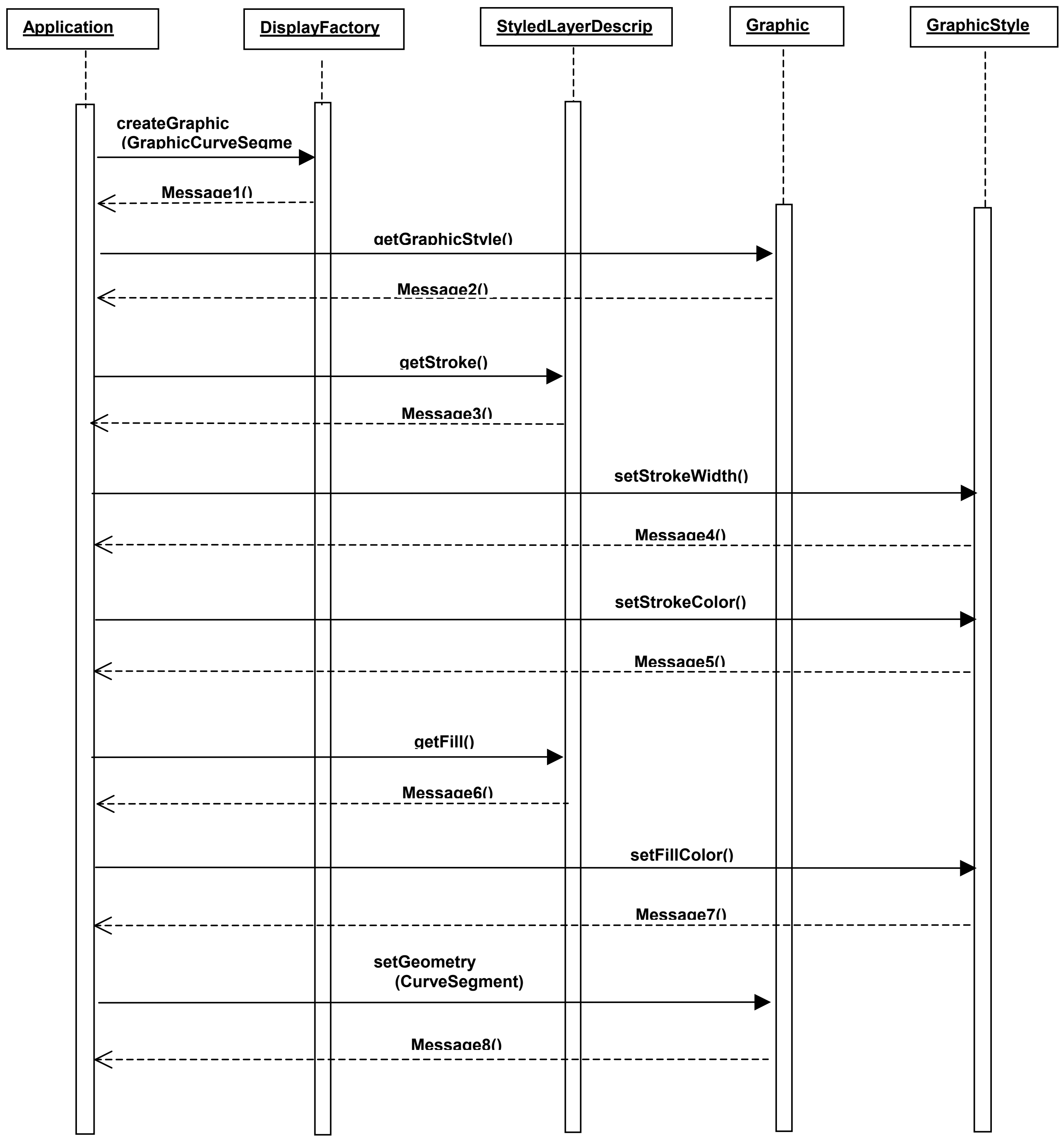


Figure 28 - Graphic Object Creation

7.4 Relative Coordinate Use Cases

7.4.1 An image that does not scale with a CRS

A dynamically-constructed Graphic that does not scale with the `Canvas` objective CRS is to be displayed by the `Canvas`. The objective CRS of a `Canvas` happens to be a `GeographicCRS`. In this example, the particular type of the display CRS of the `Canvas` does not matter. Also irrelevant to this example is the `MathTransform` (MT_{OD}) that the `Canvas` also holds to convert `Geometry` objects in the `Canvas` objective CRS (the `GeographicCRS`) into `Geometry` objects in the `Canvas` display CRS.

The Graphic is constructed using a square `Geometry` that is defined by four `DirectPosition` objects, which are associated with a single `ImageCRS` (which in this example is not the `Canvas` display CRS). A `MathTransform` (MT_I) is selected that associates the `ImageCRS` with the `GeographicCRS` in such a way that the `Geometry` does not translate, rotate, or scale.

For example, the square has pixel coordinates (-4, -4), (4, -4), (4, 4), (-4, 4). This square is always drawn with the top left corner 4 pixels above and to the right of the reference coordinate (e.g., a lat/long point) no matter where that point is on the display, and the square is always 8 pixels wide and 8 pixels high, no matter what the scale of `GeographicCRS` with respect to the `ImageCRS` (zoom factor). Note that this implies that the `Geometry.transform()` for the square `Geometry` may be called on an as-needed basis: whenever the scale, location, or projection of the `GeographicCRS` changes with respect to either the ImageCRS, or changes with respect to the `Canvas` display CRS.

7.4.2 An image that is in a CRS chain and scales with a `ProjectedCRS`

A registered image is to be displayed in a fixed range and bearing from a fixed location in a `Canvas` objective CRS, which happens to be a `ProjectedCRS`.

An `EngineeringCRS` exists, as does a `MathTransform` (MT_{EP}) that converts from the `EngineeringCRS` to the `ProjectedCRS`. An `ImageCRS` exists, as does a `MathTransform` (MT_{IE}) that converts from the `ImageCRS` to the `EngineeringCRS`.

The registered image is set with two `DirectPosition`s in the `ImageCRS`. The `ImageCRS` scales with the `EngineeringCRS`. The `EngineeringCRS` scales with the ProjectedCRS.

The `Canvas` holds another `MathTransform` (MT_{OD}) to convert the `Geometry` object in the `Canvas` objective CRS (the `ProjectedCRS`) into `Geometry` objects in the `Canvas` display CRS. The sequence of CRS objects (starting with the initial `ImageCRS` and ending with the `Canvas` display CRS) bound by the intervening MathTransform objects, together form a "chain".

An implementation can either create a new Geometry object at each transform() invocation in the chain, or can call `MathTransformFactory.createConcatenatedTransform()` in sequence on each MathTransform, and ultimately generate a MathTransform that will convert `Geometry` objects from the starting CRS (the initial `ImageCRS`) in the chain to those in the ending CRS (the `Canvas` display CRS) in the chain.

7.4.3 An `EngineeringCRS` scaling directly with another `EngineeringCRS`.

A ship is to be depicted coming into port. The ship is represented by a `Geometry` having an `EngineeringCRS` (CRS_S). The origin of CRS_S is a position within the ship's `Geometry`, such as at the centre of buoyancy of the ship or at the forward-most point of the bow at main deck level. The port is depicted by a set of `DirectPostions` having a different `EngineeringCRS` (CRS_P).

A `MathTransform` (MT_{SP}) exists that converts from CRS_S to CRS_P. MT_{SP} has the following qualities: (A) a direct identity scaling from CRS_S to CRS_P, (B) the behaviour that the origin `DirectPosition` of CRS_S corresponds to a particular `DirectPosition` in CRS_P, which denotes the ship position in CRS_P, and (C) an orientation of the CRS_S to CRS_P, denoting the rotation of the ship with respect to the port.

Note that a mathematically identical case would be if CRS_P is a georeferenced CRS, such as a `GeograhicCRS`. Similarly mathematically identical is the case where both CRS_S and CRS_P are georeferenced CRS types; however, Topic 2 would prohibit time-based changes to CRS_S in a canonically correct implementation.

7.5. Symbology Use Cases

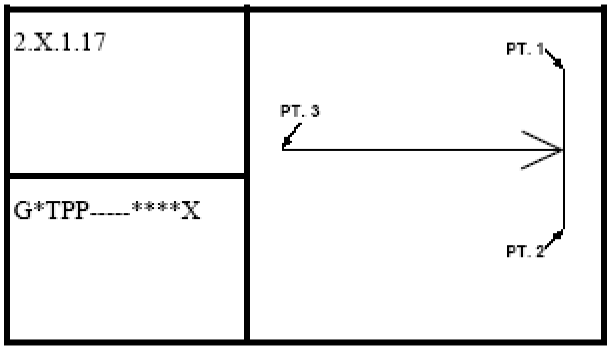
For all standard symbologies detailed in this document, tag sets are fully defined in Appendix B, and one or more use cases are presented here. Each selected symbology use case provides sample client side source code that could be used to construct the symbol, and an illustration of how a corresponding symbol might appear on a map.

7.5.1 MIL-STD 2525 Tactical Graphic

MIL-STD 2525B [34] has evolved from North Atlantic Treaty Organization (NATO) Standardization Agreement (STANAG) 2019 (APP 6), "Military Symbols for Land Based Systems," and U.S. Army Field Manual (FM) 101-5-1/Marine Corp Reference

Publication (MCRP) 5-2A, *Operational Terms and Graphics*. It provides common warfighting symbology along with details on its display and plotting to ensure the compatibility, and to the greatest extent possible, the interoperability of DOD Command, Control, Communications, Computer, and Intelligence (C4I) systems development, operations, and training. The standard addresses the efficient transmission of symbology information within the infosphere through the use of a standard methodology for symbol hierarchy, information taxonomy, and symbol identifiers. The standard applies to both automated and hand-drawn graphic displays. These symbols are designed to enhance DOD's joint warfighting interoperability by providing a standard set of common C4I symbols.

Figure 29 - MIL-STD 2525 Tactical Graphic



```
GraphicCurveSegment tacgraph =
(GraphicCurveSegment)displayFactory.createGraphic(GraphicCurveSegment.class);

//anchor 1
LatLonAlt anchor = new LatLonAlt();
anchor.setLatLon(30, 20, degreesUnits);
tacgraph.addAnchorPoint(anchor);

//anchor 2
anchor = new LatLonAlt();
anchor.setLatLon(30, -20, degreesUnits);
tacgraph.addPoint(anchor);

//anchor 3
anchor = new LatLonAlt();
anchor.setLatLon(0, 0, degreesUnits);
tacgraph.addPoint (anchor);

SymbologyInfo symInfo = new SymbologyInfo ("MIL-STD-2525", "B");

tacgraph.getGraphicStyle().setSymbologyProperty(symInfo, "SymbolID", "G*TPP----
-****X");
tacgraph.getGraphicStyle().setSymbologyProperty(symInfo, "DirectionOfMovement",
new Double(6.281));
tacgraph.getGraphicStyle().setActiveSymbology(symInfo);
canvas.add(tacgraph);
```

7.5.2 MIL-STD 2525 Air Track

This example illustrates combining point and line dimensions to form this MIL-STD 2525 Air Track symbol.

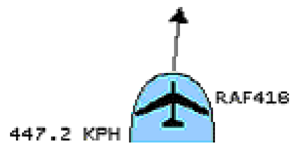


Figure 30 - MIL-STD 2525b Air Track

```
GraphicIcon icon =
(GraphicIcon)displayFactory.createGraphic(GraphicIcon.class);

//anchor point
LatLonAlt anchor = new LatLonAlt();
anchor.setLatLon(30, 20, degreesUnits);
icon.addAnchorPoint(anchor);

// note this GraphicIcon doesn't require a java Icon as the image
// will determined by the MIL-STD-2525 symbology.
SymbologyInfo symInfo = new SymbologyInfo ("MIL-STD-2525", "B");

icon.getGraphicStyle().setSymbologyProperty(symInfo, "SymbolID", "SFAPMF------
USA");
icon.getGraphicStyle().setSymbologyProperty(symInfo, "AdditionalInformation",
"RAF418");
icon.getGraphicStyle().setSymbologyProperty(symInfo, "Frame", Boolean.TRUE);
icon.getGraphicStyle().setSymbologyProperty(symInfo, "Fill", Boolean.TRUE);
icon.getGraphicStyle().setSymbologyProperty(symInfo, "Icon", Boolean.FALSE);
icon.getGraphicStyle().setSymbologyProperty(symInfo, "Speed", new
Double(447.2));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "DirectionOfMovement", new
Double(6.281));
icon.getGraphicStyle().setActiveSymbology(symInfo);
canvas.add(icon);
```

7.5.3 Surface Weather

Surface weather depictions are commonly found in weather reporting systems. This symbol is rather detailed, so additional annotations are provided in blue and red. This Surface weather symbology is based upon the U.S. National Weather Service (NWS) Meteorology standards [32]. Depiction of such symbology tends to vary slightly across regions, so implementations may adhere to display standards applicable to their own geographical regions. A common set of tags for this symbology appears in Appendix B, Table "SurfaceWeather".

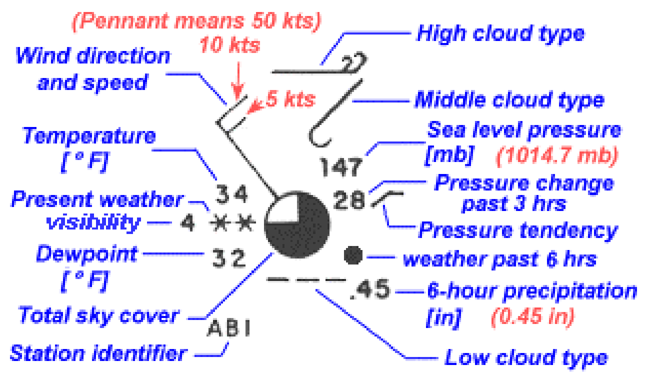


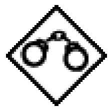
Figure 31 - Surface Weather (annotated)

```
GraphicIcon icon =
(GraphicIcon)displayFactory.createGraphic(GraphicIcon.class);

//anchor point
LatLonAlt anchor = new LatLonAlt();
anchor.setLatLon(24.2, -112.5, degreesUnits);
icon.addAnchorPoint(anchor);
symbol.addAnchorPoint(anchor);

Symbology symInfo = symbol.getSymbology("SurfaceWeather");

icon.getGraphicStyle().setSymbologyProperty(symInfo, "SymbolID", "SFAPMF------
USA");
icon.getGraphicStyle().setSymbologyProperty(symInfo, "WindDirection", new
Double(10.0));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "WindSpeed", new
Double(5.0));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "Temperature", new
Integer(34));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "PresentWeather", 71);
icon.getGraphicStyle().setSymbologyProperty(symInfo, "Visibility", new
Integer(4));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "Dewpoint", new
Integer(32));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "SkyCover", new
Integer(6);
icon.getGraphicStyle().setSymbologyProperty(symInfo, "StationIdentifier",
"AB1");
icon.getGraphicStyle().setSymbologyProperty(symInfo, "PressureChange", new
Integer(28));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "PressureTendency", new
Integer(1));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "PastWeather", new
Integer(6));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "PastPrecipitation", new
Double(0.45));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "HighCloudType", new
Integer(2));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "MiddleCloudType", new
Integer(4));
icon.getGraphicStyle().setSymbologyProperty(symInfo, "LowCloudType", new
Integer(7));

icon.getGraphicStyle().setActiveSymbology(symInfo);

canvas.add(icon);
```

7.5.4 Homeland Security

The Homeland Security symbology [33] is a work in progress by the Homeland Security Working Group. This standard has yet to be released as of this writing. Only point symbols are currently supported. Each SymbolCode is mapped to a single keystroke character. Those symbol codes are further subdivided into one of four categories; Incidents, Natural Events, Operations, and Infrastructures. Tags for this symbology appear in Appendix B, Table “FGDCHomelandSecurity”.

Figure 32 - Homeland Security Symbol



```
GraphicIcon icon = (GraphicIcon)displayFactory.
createGraphic(GraphicIcon.class);

//anchor point
LatLonAlt anchor = new LatLonAlt();
anchor.setLatLon(23.1, -114.75, degreesUnits);
icon.addAnchorPoint(anchor);
symbol.addAnchorPoint(anchor);

Symbology symInfo = symbol.getSymbology(“FGDCHomelandSecurity”);

// Criminal Activity Incident (Theme)
Symbology symbology = symbol.getSymbology(“FGDCHomelandSecurity”);
icon.getGraphicStyle().setSymbologyProperty(symInfo, “SymbolCode”, “E”);
icon.getGraphicStyle().setSymbologyProperty(symInfo, “SymbolType”, “Incident”)
icon.getGraphicStyle().setSymbologyProperty(symInfo, “Level”, new Integer(0));

icon.getGraphicStyle().setActiveSymbology(symInfo);

canvas.add(icon);
```

7.6 Z-order Use Case

For the GO-1 reference implementation, we chose to implement z-order in such as way as to assume that graphic elements that had not set the z-order hint were assumed to share the z-order of 0 (zero). Any items we desired to draw below the "standard" z-order of all graphic primitives were set at -1. Any items drawn above all objects at the default z-order were set at 1.

A use case for this would be the algorithmic determination of z-order by altitude. The GO-1 client application could pick any range of z for its objects. Let's assume that the client application picked -100.0 to +100.0 to represent algorithmically determined elevation (and depth) levels.

Assuming this to be the situation, the following would be true:

1) Items set with `graphic.getGraphicStyle().setViewabilityZOrderHint(101.0)` would appear "on top" of all items determined algorithmically.

2) Items set with `graphic.getGraphicStyle().setViewabilityZOrderHint(-101.0)` would appear "below" all items determined algorithmically.

3) Items with their z-order hint not set would appear at the level determined by inheritance.

4) Items in case #3 with no inherited z-order hint would be displayed at the GO-1 specification default z-order of 0.0.

There are a number of approaches by which z-order can be set:

1) Z-order can be set using the `getGraphicStyle().setViewabilityZOrderHint(double)` method, on each graphic.

2) A graphic can be added to an aggregate, which includes `OrderedAggregateGraphic`. If the graphic's z-order hint is unset, and inheritance is enabled, the graphic will be treated as having the z-order of the aggregate.

3) Using `OrderedAggregateGraphic`, which preserve rending order within themselves, the client application can guarantee the drawing order of all of its objects without using the z-order hint in the graphic styles at all. Since each `OrderedAggregateGraphic` is guaranteed to render its children in order, from index 0 to n, and `OrderedAggregateGraphic` can be added to others as children, this hierarchy of drawing order can be created.

4) Z-order hints set in the `GraphicStyle` of a `Graphic` object **always** override both the default z-ordering, and the ordering of an object within a `OrderedAggregateGraphic`. Setting the hint, in essence, specifies to the `Canvas` exactly at what z level the user wishes the object to be displayed.

Mixing and matching the aforementioned z-order approaches may lead to interesting and non-deterministic results, according to the implementation. It is our suggestion that, if z-order is used, one approach be applied consistently for all client objects in order to decrease the likelihood of unexpected visual results.

Annex A
(normative)

Application Objects Programming Interface for Java

A.1 General

The detailed specifications for the GO-1 Application Objects programming interface have been made available in Javadoc format. These materials are available under separate cover in 03-064_Annex_A.zip.

The Spatial Object interface specifications are being developed as a part of the Model Driven Architecture interface and specification development experiments described at various points throughout the text above.

Annex B
(normative)
Symbology Property Names

B.1 Surface Weather Symbology

References: How to read weather maps

Property Name	Type	Description
Dewpoint	Double	Degrees Fahrenheit
HighCloudType	Integer	Code from 1-9
LowCloudType	Integer	Code from 1-9
MiddleCloudType	Integer	Code from 1-9
*PastPrecipitation	Double	Inches in past six hours
PastWeather	Integer	Code from 0-9, past six hours
PresentWeather	Integer	Code from 0-99, present
PressureChange	Double	Millibars to nearest tenth
PressureTendency	Integer	Code from 0-8
SeaLevelPressure	Double	Millibars to nearest tenth
SkyCover	Integer	Code from 0-9, total cloud cover
*StationIdentifier	String	http://weather.noaa.gov/tg/site.shtml
Temperature	Double	Degrees Fahrenheit
Visibility	Integer	Miles
WindDirection	Double	Degrees from 0-360
WindSpeed	Double	Knots

Table 3 - SurfaceWeather

* Denotes a property name extension outside the specification used, but found to be in common use.

B.2 Homeland Security

Reference: Homeland Security Working Group

Property Name	Type	Description
`SymbolCode`	`String`	`keystroke`
`SymbolType`	`String`	`“Incident”, “NaturalEvent”, “Operation”, “Infrastructure”`
`Level`	`Integer`	`0-4, 0 is no level or n/a`

Table 4 - FGDCHomelandSecurity

B.3 U.S. Military Symbology

Reference: http://symbology.disa.mil/symbol/mil-std.html

Property Name	Type
AdditionalInformation	String
AltitudeDepth	String
AuxiliaryEquipment	Boolean
CombatEffectiveness	String
CommonIdentifier	String
DateTimeGroup	Date
DateTimeGroupAlt	Date
DirectionOfMovement	Double
EchelonIndicatorDescription	String
EquipmentTeardownTime	Integer
EvaluationRating	String
FeintDummy	Boolean
FrameShapeModifier	String
Frame	Boolean
Fill	Boolean
Headquarters	Boolean
HigherFormation	String
Hostile	String
Icon	Boolean
IFFSIF	String
Installation	Boolean
Location	String
Mobility	String
OffsetLocation	Boolean
PlatformType	String
Quantity	String
Reduced	Boolean
Reinforced	Boolean
SIGINTMobility	String
SignatureEquipment	String
SpecialC2Headquarters	String
Speed	String
StaffComments	String
SymbolID	String
TaskForce	Boolean
Type	String
UniqueDesignation	String

Table 5 - MIL-STD-2525B

* Refer to MIL-STD-2525B for property name description and behaviour.

B.4 Aeronautical Symbology (future)

The 6th Edition of the Aeronautical Chart User's Guide makes available in PDF format chart symbols for Visual Flight Rules (VFR), Instrument Flight Rules (IFR), and Instrument Approach (IAP). These are defined by the U.S. National Aeronautical Charting Office (NACO), and are found at http://www.naco.faa.gov/index.asp?xml=naco/online/aero_guide.

B.5 Nautical Symbology (future)

The current edition of NOAA Chart No. 1 Nautical Chart Symbols is available from the National Geospatial-Intelligence Agency (NGA) at http://pollux.nss.nima.mil/pubs/pubs_j_show_sections.html?vt=ON&dpath=Chart1&ptid=3&rid=164

Bibliography

[1] ISO 31 (all parts), Quantities and units.

[2] IEC 60027 (all parts), Letter symbols to be used in electrical technology.

[3] ISO 1000, SI units and recommendations for the use of their multiples and of certain other units.

[4] Guidelines for Successful OGC Interface Specifications, OGC document 00-014r1

[5] OpenGIS ® Topic 1: Feature Geometry (ISO 19107 Spatial Schema), version 5, OGC document 01-101r5. Available at: http://www.opengis.org/techno/abstract/01-101.pdf

[6] OpenGIS ® Topic 2: Spatial Referencing By Coordinates, OGC document 03-073r4. Available at http://www.opengis.org/docs/03-073r1.zip

[7] OpenGIS ® Simple Feature Specification for SQLVersion, version 1.1. Available at: http://www.opengis.org/techno/implementation.htm

[8] OpenGIS ® Topic 5: The OpenGIS Feature. Available at: http://www.opengis.org/techno/abstract/01-105r2.pdf

[9] OpenGIS ® Grid Coverages Implementation Specification, version 1.0. Available at: http://www.opengis.org/techno/implementation.htm

[10] OpenGIS ® Catalog Service Implementation Specification, version 1.1.1. Available at: http://www.opengis.org/techno/implementation.htm

[11] OpenGIS ® Geography Markup Language (GML) Implementation Specification, version 2.1.2. Available at: http://www.opengis.org/techno/implementation.htm

[12] OpenGIS ® Geography Markup Language (GML) Implementation Specification (version 3.0), OGC document 02-023r4. Available at: http://www.opengis.org/docs/02-023r4.pdf

[13] OpenGIS ® Web Mapping Server (WMS) Implementation Specification, version 1.1.1. Available at: http://www.opengis.org/techno/implementation.htm

[14] OpenGIS ® Styled Layer Descriptor (SLD) Implementation Specification, version 1.0. Available at: http://www.opengis.org/techno/implementation.htm

[15] OpenGIS ® Web Feature Server (WFS) Implementation Specification, version 1.0. Available at: http://www.opengis.org/techno/implementation.htm

Filter Encoding:

[16] OpenGIS® Filter Encoding Implementation Specification, version 1.0. Available at: http://www.opengis.org/techno/implementation.htm

[17] OpenGIS ® Coordinate Transformation Services Implementation Specification, version 1.0, OGC document 01-009. Available at: http://www.opengis.org/techno/implementation.htm

[18] Web Coordinate Transformation Service (WCTS), v0.0.4, OGC document 02-061r1. Available at: http://www.opengis.org/docs/02-061r1.pdf

[19] OpenGIS ® Web Coverage Server (WCS) Discussion Paper, OGC document 02-024r1. Available at: http://www.opengis.org/techno/requests.htm

[20] Coverage Portrayal Service Specification (CPS), OWS1.1 IPR. OGC document 02-019r1.

[21] Style Management Service IPR, Discussion Paper, OGC document 03-031. (including proposed changes to SLD). Available at: http://www.opengis.org/info/discussion.htm

[22] Registry Service, Discussion Paper, OGC document 03-024. Available at: http://www.opengis.org/info/discussion.htm

[23] Integrated Client for OGC Services, Discussion Paper. OGC document 03-021. Available at: http://www.opengis.org/info/discussion.htm

[24] OWS Service Information Model, Discussion Paper, OGC document 03-026. Available at: http://www.opengis.org/info/discussion.htm

[25] OpenGIS ® Web Service Architecture, Discussion Paper, OGC document 03-025. Available at: http://www.opengis.org/info/discussion.htm

[26] OGC Reference Model (ORM), OGC document 02-077. Available at: http://www.opengis.org/info/discussion.htm

[27] OGC Spatial Reference Systems, OGC document 02-102. Available at: http://www.opengis.org/techno/abstract/02-102.pdf

[28] UML for Spatial Referencing by Coordinates, OGC document 03-009R5. Available at: http://member.opengis.org/tc/archive/arch03/03-009r5.doc

[29] Recommended XML Encoding of CRS Definitions (XML for CRS), v2.1.0, OGC document 03-010r9. Available at: http://www.opengis.org/docs/03-010r9.zip

[30] CT Definition Data for Coordinate Reference (DD CRS), v1.1.0, OGC document 01-014r5. Available at: http://www.opengis.org/docs/01-014r5.pdf

[31] High-Level Ground Coordinate Transformation Interface (HLG-CT), v0.0.3, OGC document 01-013r1. Available at: http://www.opengis.org/docs/01-013r1.pdf

[32] How To Read Weather Maps, National Weather Service - JetStream. Available at http://www.srh.weather.gov/srh/jetstream/synoptic/wxmaps.htm - ww_type or download at http://www.srh.weather.gov/srh/jetstream/zippedfiles/synoptic_030904.exe

[33] Homeland Security Symbology Reference, FGDC Homeland Security Working Group. Available at http://www.fgdc.gov/HSWG/downloadSymbols.htm

[34] MIL-STD-2525B Common Warfighting Symbology, ver B, 30 Jan 1999. Available at http://symbology.disa.mil/symbol/mil-std.html

Open Source Implementation Baselines

GO-1 and GeoAPI:

- http://sourceforge.net/projects/geoapi

Geobject 1.3 and Geobject 2.0a:

- http://geobject.org/umldoc/2.0alpha
- http://sourceforge.net/projects/geobject

Geotools and Geotools2:

- http://modules.geotools.org/core
- http://www.geotools.org