Open Geospatial Consortium Inc.

Date: 2005-05-04

Reference number of this OGC™ project document: **OGC 03-064r10**

Version: 1.0

Category: OpenGIS® Implementation

Editor: Greg Reynolds, SYS Inc. / OGC

GO-1 Application Objects

Contents ....

i. Preface

This document is the Open Geospatial 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) SYS Technologies

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	EMAIL
Eric Bertel	SYS Technologies	eric@polexis.com
Greg Reynolds	SYS Technologies	greynolds@polexis.com

CONTACT	COMPANY	EMAIL
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
Christopher Dillard	SYS Technologies	cdillard@polexis.com
Jody Garnett	Refractions Research	jgarnett@refractions.net

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 B. Modified 6.1.1.5,	Third public draft

			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.
15 Dec 04	0.6.0	C. Dillard, J. Garnett	Add Figure 4, Figure 5, Figure 6, Add Figures 27 - 32. Figure 35. Add 6.1.1.7, 6.2.1.1, 6.2.1.2, 6.2.2.2, 6.2.2.4, Add 6.4-6.6, 7.2.1, 7.3, 7.5.1.	Added section 6.1.1.7 on Map2D. Added sections 6.4-6.6 on Features, FeatureStores, and FeatureCanvas. Diagrams and text modified to reflect renaming of GraphicCurveSegment to GraphicLineString and addition of GraphicPolygon. Also rewrote SLD section to be more clear.
16 Dec 04	0.7.0	C. Dillard	Added section 6.3.1.7. Edit section 2. Add annexes E, F, and G. Added content to Annex C.	Added section on Geometry mutability. Changed the conformance types to include Feature and Data Provider conformance.
17 Dec 04	0.7.1	G. Reynolds	Added section 2.6, Update 6.3.2.1	Add Conformance Summary and diagram. General document cleanup. Change TemporalCRS to TimeCRS.
30 Dec 04	0.7.2	C. Dillard	Figure 18, 19, 21	Updated obsolete figures.
5 Jan 05	0.7.3	G. Reynolds	Added 6.7 GraphicStore. Revised Annex F.	Changed references from DataStore to FeatureStore to reflect GeoAPI update. Added GraphicStore API. Updates to Annex F.
17 Jan 05	0.7.4	D. Arctur, G. Reynolds	Modified 1.0. Reordered 2.0.	Updated Scope; qualified conformance levels; sourced Simple Features for SQL.

25 Jan 05	0.8	C. Dillard	6.1.2.3 (removed), 6.2.1.1, 6.2.1.6 (added), 6.2.2.3, 6.2.2.6 (added).	Separated event and editability sections. Corrected obsolete references to ManagerSupport. Corrected style property table.

v. Changes to the OpenGIS→ Abstract Specification

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

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 well-defined mechanism for allowing individual implementations to add new graphical primitives and corresponding graphic style interfaces.

- 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. `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*).

In the future, a sub-profile of existing 19107 interfaces should be defined that allows implementations to support only Simple Features for SQL (99-049). As a related compliance issue, the conversion process between Simple Features and 19107 Geometry should be explored.

Web services providing for delivery of asynchronous messages between peers is envisioned as a future need. Such a Web Notification Service should be readily interoperable with the event notification mechanisms provided by GO-1.

Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial 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 Geospatial 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* (Open GIS Consortium Inc). 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 OGC document describes the specification for Application Objects. These are the Java and other 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). The focus for interoperability in recent years has been toward coarse-grained, loosely coupled web services. Application Objects provides a complementary API-based approach that can yield improved functionality over web services, at the expense of increased coupling. Both approaches can be valid within various enterprise IT environments.

While this specification outlines certain service interfaces, it does not require the use of any particular service implementation. A GO-1 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.

2 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.

2.1 Conformance Overview

In order for a GO-1 implementation to be conformant, FeatureStore and FeatureCanvas both require a conformant implementation of **Features**. Topic 1 - Feature Geometry conformance is required for GO-1 FeatureCollection and Graphics API's. Topic 2 - Spatial Referencing by Coordinates conformance is required for GO-1 Canvas. For an implementation of **Features** to be conformant, it requires conformance to Filter Encoding and Styled Layer Descriptor.

The figure below illustrates the aforementioned conformance relationships. Spatial conformance is shaded green, and feature conformance is shaded blue.

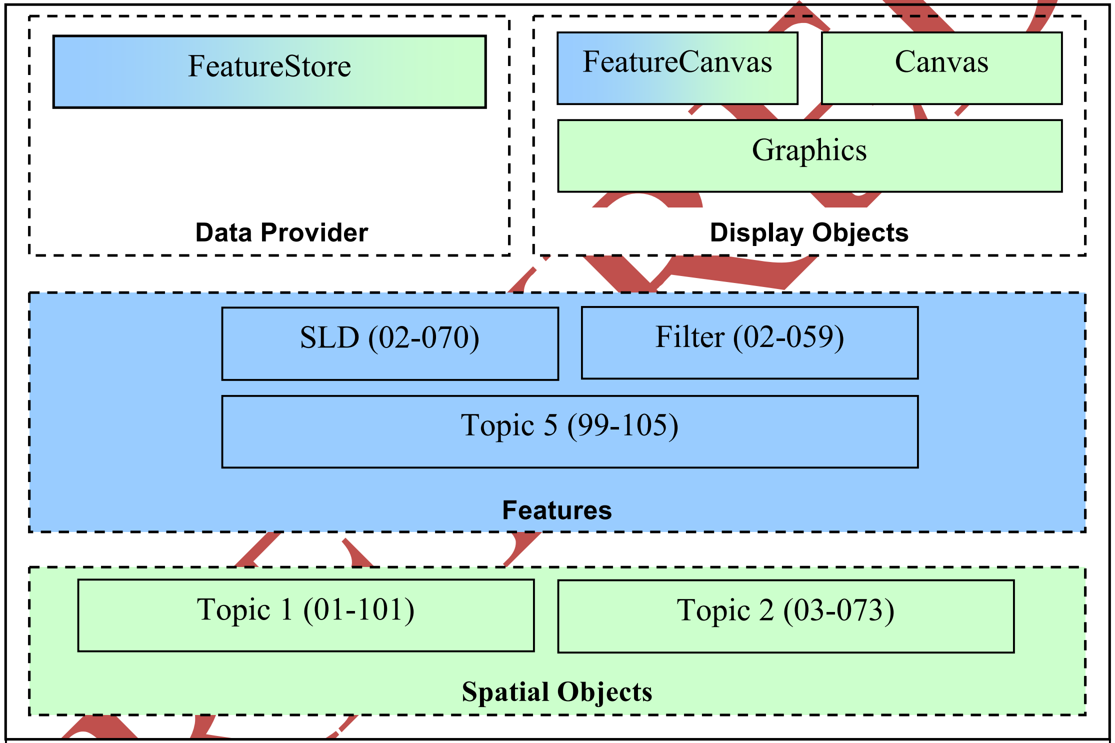


Figure 1 - Conformance Types

2.2 Spatial and Feature Conformance

Feature, Geometry, Coordinate Reference System, and related entities constitute the spatial and feature 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, OGC Abstract Specification Topic 5, OGC Simple Features for SQL (OGC 99-049), and other 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. The interoperability includes both the geometric properties of objects (Geometry) and the non-geometric properties (Feature attributes).

Feature conformance in GO-1 requires Spatial and Feature Conformance as well as implementation of the FeatureCanvas API.

In javadocs, implementations shall use the annotation "`SPATIAL`" for identifying Spatial conformance and "FEATURE" for Feature conformance.

2.3 Data Provider Conformance

FeatureStore and related entities constitute the data provider interfaces defined by GO-1. This interface builds upon the efforts of the OpenSource community (in particular GeoTools) to provide a standardized mechanism for data providers to provide queriable access to data. The interfaces prevent the client from having to know anything about the implementation of the data connection.

FeatureStore conformance in GO-1 also requires Feature conformance.

In javadocs, implementations shall use the annotation "`DATA_PROVIDER`" for identifying this conformance level.

2.4 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 surface that the user sees and interacts with. 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.

This specification recognizes two different levels of Display Object Conformance, basic Display Objects and Display Objects with Events and Editing. In the first level, conforming implementations correctly display Graphic primitives according to their associated GraphicStyle. In the second level of conformance, an implementation additionally correctly fires events when the user interacts with the Canvas and the editability properties of Graphics are fully supported.

Both of these levels of conformance require the implementation of the Canvas, Graphic, and GraphicStyle interfaces. Only in the second level are implementations required to handle the event model described here. The following sections are required for Display Objects with Events and Editing:Events, and all sub sections.

- 7.2.1 Graphic Editability

Display Object 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.

In API documentation the annotation "`DISPLAY_OBJECT`" for identifying Display Object conformance level, and "`EDITABLE_DISPLAY_OBJECT`" for Editable Display Objects with Events.

2.5 OGC Service Conformance

The GO-1 specification does not mandate or exclude any particular web services. 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.

2.6 Coordinate Transformations

2.6.1 Coordinate Transformations

GO-1 implementations should support coordinate transformations. Below is a suggested list of coordinate transformations that could be supported.

- 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

2.6.2 Operation Methods

GO-1 implementations should support operation methods. The following operation methods are suggested.

- 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).Grid-Based Transform (NADCON and NTv2).

2.6.3 Required Datum

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

- WGS-84 World Geographic Survey, 1984.

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.

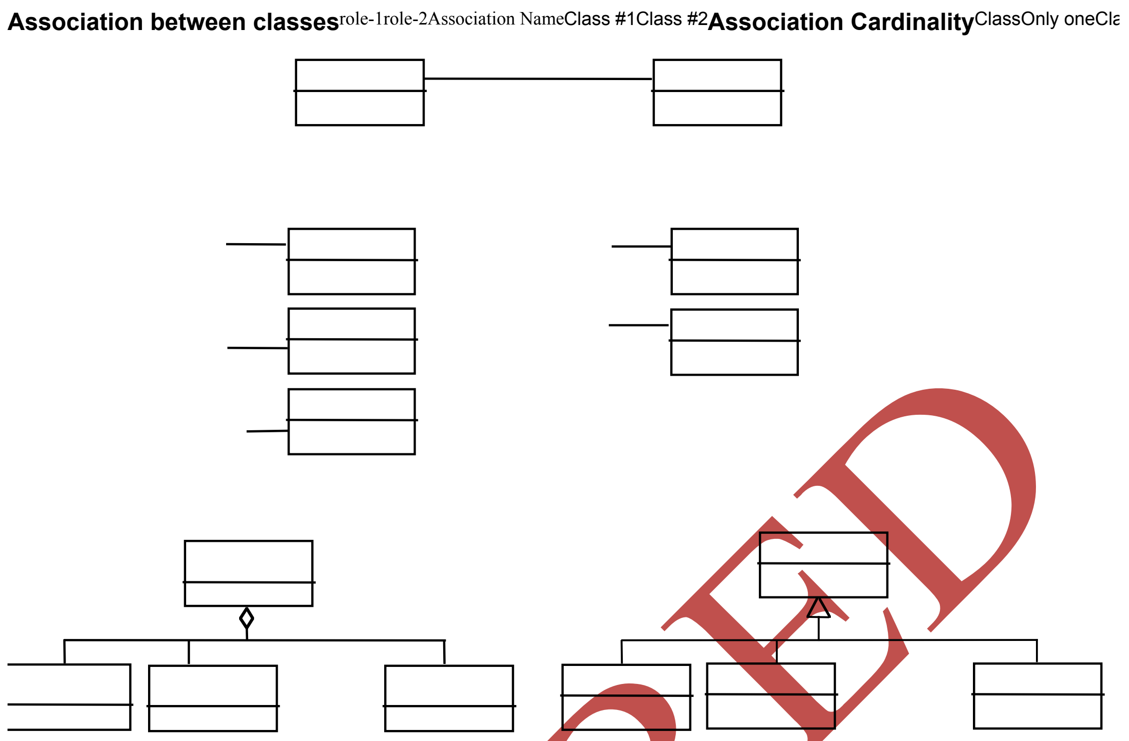


Figure 2 - UML notation

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

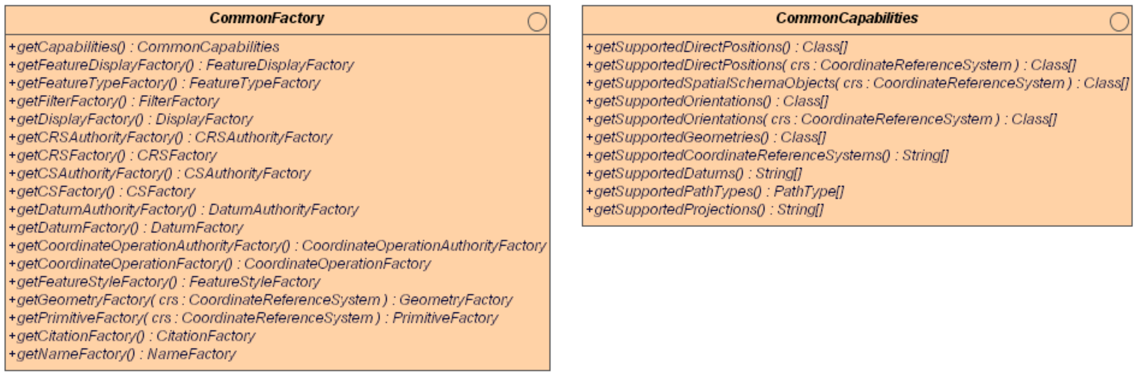

CommonFactory
+getCapabilities() : CommonCapabilities
+getFeatureDisplayFactory() : FeatureDisplayFactory
+getFeatureTypeFactory() : FeatureTypeFactory
+getFilterFactory() : FilterFactory
+getDisplayFactory() : DisplayFactory
+getCRSAuthorityFactory() : CRSAuthorityFactory
+getCRSFactory() : CRSFactory
+getCSAuthorityFactory() : CSAuthorityFactory
+getCSFactory() : CSFactory
+getDatumAuthorityFactory() : DatumAuthorityFactory
+getDatumFactory() : DatumFactory
+getCoordinateOperationAuthorityFactory() : CoordinateOperationAuthorityFactory
+getCoordinateOperationFactory() : CoordinateOperationFactory
+getFeatureStyleFactory() : FeatureStyleFactory
+getGeometryFactory(crs : CoordinateReferenceSystem) : GeometryFactory
+getPrimitiveFactory(crs : CoordinateReferenceSystem) : PrimitiveFactory
+getCitationFactory() : CitationFactory
+getNameFactory() : NameFactory
CommonCapabilities
+getSupportedDirectPositions() : Class[]
+getSupportedDirectPositions(crs : CoordinateReferenceSystem) : Class[]
+getSupportedSpatialSchemaObjects(crs : CoordinateReferenceSystem) : Class[]
+getSupportedOrientations() : Class[]
+getSupportedOrientations(crs : CoordinateReferenceSystem) : Class[]
+getSupportedGeometries() : Class[]
+getSupportedCoordinateReferenceSystems() : String[]
+getSupportedDatums() : String[]
+getSupportedPathTypes() : PathType[]
+getSupportedProjections() : String[]

Figure 3 - CommonFactory

The GO-1 `CommonFactory` class supports a `getCapabilities()` operation that allows it to describe the supported features. 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.

The `CommonFactory` is the gateway into the GO-1 implementation. To access other factories used in GO-1, they are first obtained from the `CommonFactory`. For example, `CommonFactory.getCRSAuthorityFactory()` returns the CRSAuthorityFactory object from the `CommonFactory` implementation.

6.1.1 Graphic Object Creation

Graphic objects are created by invocation of the `DisplayFactory .createGraphic()` method. 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`.

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.

6.2 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.

6.2.1 Canvas

6.2.1.1 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.

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

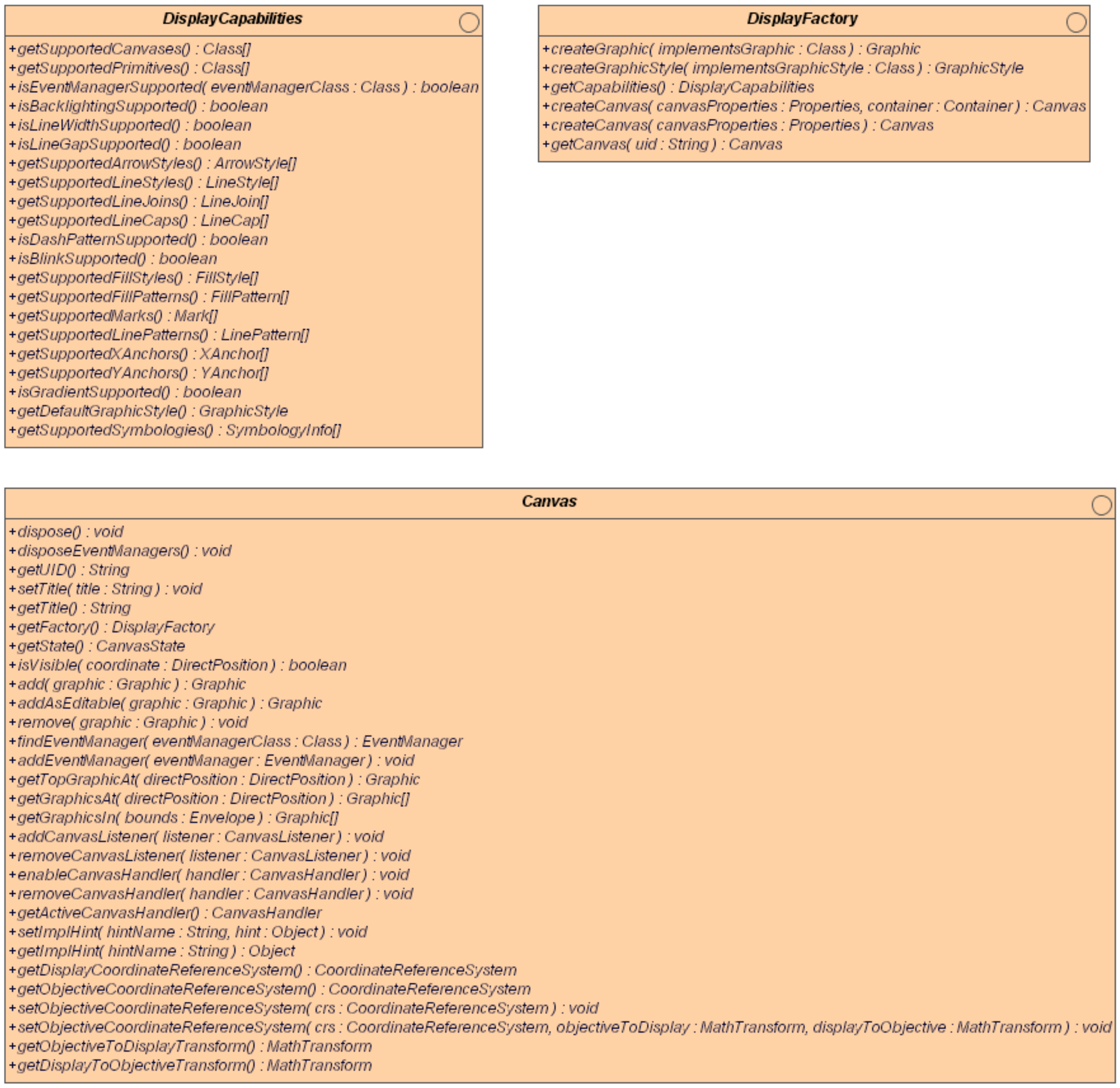

DisplayCapabilities
+getSupportedCanvases() : Class[]
+getSupportedPrimitives() : Class[]
+isEventManagerSupported(eventManagerClass : Class) : boolean
+isBacklightingSupported() : boolean
+isLineWidthSupported() : boolean
+isLineGapSupported() : boolean
+getSupportedArrowStyles() : ArrowStyle[]
+getSupportedLineStyles() : LineStyle[]
+getSupportedLineJoins() : LineJoin[]
+getSupportedLineCaps() : LineCap[]
+isDashPatternSupported() : boolean
+isBlinkSupported() : boolean
+getSupportedFillStyles() : FillStyle[]
+getSupportedFillPatterns() : FillPattern[]
+getSupportedMarks() : Mark[]
+getSupportedLinePatterns() : LinePattern[]
+getSupportedXAnchors() : XAnchor[]
+getSupportedYAnchors() : YAnchor[]
+isGradientSupported() : boolean
+getDefaultGraphicStyle() : GraphicStyle
+getSupportedSymbologies() : SymbologyInfo[]
DisplayFactory
+createGraphic(implementsGraphic : Class) : Graphic
+createGraphicStyle(implementsGraphicStyle : Class) : GraphicStyle
+getCapabilities() : DisplayCapabilities
+createCanvas(canvasProperties : Properties, container : Container) : Canvas
+createCanvas(canvasProperties : Properties) : Canvas
+getCanvas(uid : String) : Canvas
Canvas
+dispose() : void
+disposeEventManagers() : void
+getUID() : String
+setTitle(title : String) : void
+getTitle() : String
+getFactory() : DisplayFactory
+getState() : CanvasState
+isVisible(coordinate : DirectPosition) : boolean
+add(graphic : Graphic) : Graphic
+addAsEditable(graphic : Graphic) : Graphic
+remove(graphic : Graphic) : void
+findEventManager(eventManagerClass : Class) : EventManager
+addEventManager(eventManager : EventManager) : void
+getTopGraphicAt(directPosition : DirectPosition) : Graphic
+getGraphicsAt(directPosition : DirectPosition) : Graphic[]
+getGraphicsIn(bounds : Envelope) : Graphic[]
+addCanvasListener(listener : CanvasListener) : void
+removeCanvasListener(listener : CanvasListener) : void
+enableCanvasHandler(handler : CanvasHandler) : void
+removeCanvasHandler(handler : CanvasHandler) : void
+getActiveCanvasHandler() : CanvasHandler
+setImplHint(hintName : String, hint : Object) : void
+getImplHint(hintName : String) : Object
+getDisplayCoordinateReferenceSystem() : CoordinateReferenceSystem
+getObjectiveCoordinateReferenceSystem() : CoordinateReferenceSystem
+setObjectiveCoordinateReferenceSystem(crs : CoordinateReferenceSystem) : void
+setObjectiveCoordinateReferenceSystem(crs : CoordinateReferenceSystem, objectiveToDisplay : MathTransform, displayToObjective : MathTransform) : void
+getObjectiveToDisplayTransform() : MathTransform
+getDisplayToObjectiveTransform() : MathTransform

Figure 4 - Canvas and related classes

6.2.1.2 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.

6.2.1.3 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` and to interact with the Canvas in other ways. The `Canvas` manages the input events from these devices. See section 6.1.2 for a description of how events are to be handled.

6.2.1.4 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.

6.2.1.5 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.

6.2.1.6 Canvas State

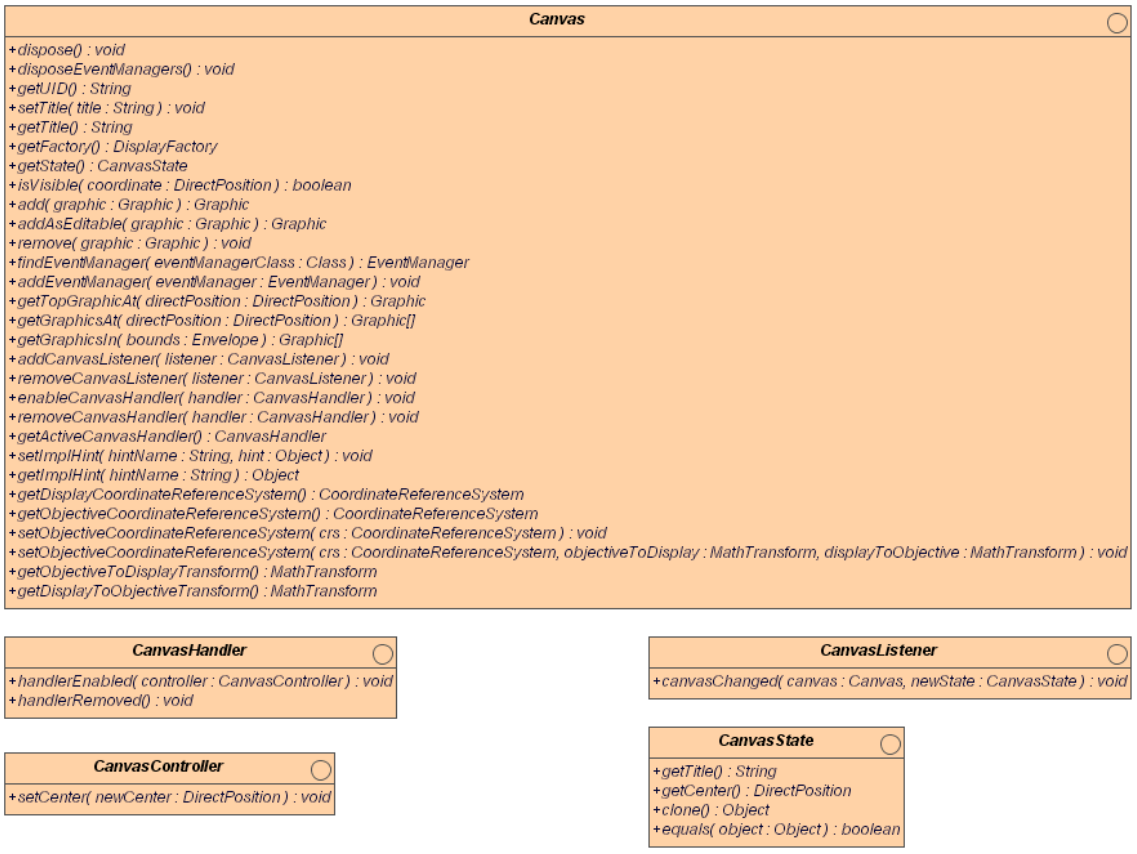

Canvas
+dispose() : void
+disposeEventManagers() : void
+getUID() : String
+setTitle(title : String) : void
+getTitle() : String
+getFactory() : DisplayFactory
+getState() : CanvasState
+isVisible(coordinate : DirectPosition) : boolean
+add(graphic : Graphic) : Graphic
+addAsEditable(graphic : Graphic)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

Figure 5 - 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` or one of its subclasses. 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.

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.

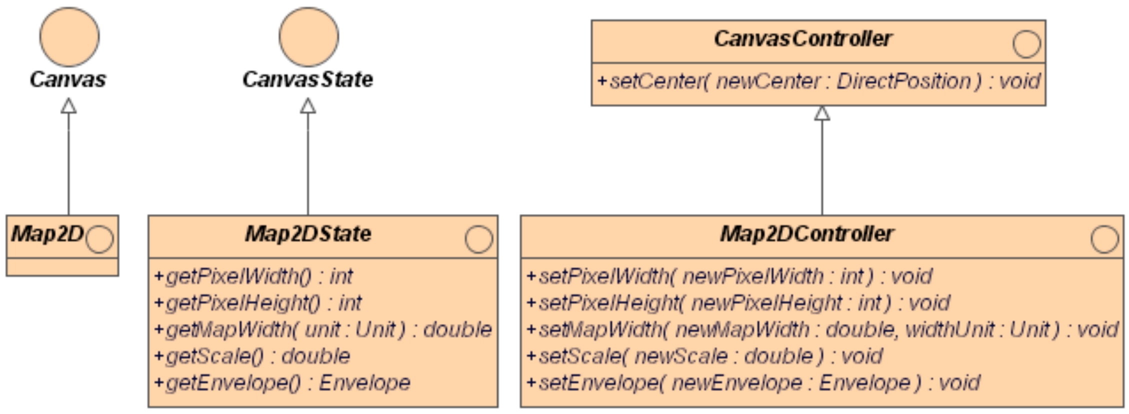
If an entity needs to change the state of a `Canvas`, it must implement the `CanvasHandler` interface. This interface provides a mechanism for multiple entities to change `Canvas` properties without contention or deadlock. The `Canvas` enables exactly one `CanvasHandler` at a time. When a `CanvasHandler` 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.

6.2.1.7 Specialized 2D Canvas Interfaces

In the majority of cases, a Canvas will be a representation of the earth, either in a projected coordinate system or in a geographic coordinate system. In these cases, the Canvas will additionally implement the Map2DCanvas interface. Such a canvas will return instances of Map2DState (which extends CanvasState) from its getState() method. And if a CanvasHandler is added to a Map2D, the CanvasController that is passed to it will be an instance of Map2DController.



Canvas
CanvasState
CanvasController
+setCenter(newCenter : DirectPosition) : void
Map2D
Map2DState
+getPixelWidth() : int
+getPixelHeight() : int
+getMapWidth(unit : Unit) : double
+getScale() : double
+getEnvelope() : Envelope
Map2DController
+setPixelWidth(newPixelWidth : int) : void
+setPixelHeight(newPixelHeight : int) : void
+setMapWidth(newMapWidth : double, widthUnit : Unit) : void
+setScale(newScale : double) : void
+setEnvelope(newEnvelope : Envelope) : void

Figure 6 - Map2D, Map2DState, and Map2DController

If the user knows that the Canvas will be an instance of Map2D, then these classes can be used to control the various 2D parameters of the Canvas.

For example, in a projected geographic CRS, a class implementing Map2DState shall provide access to the following properties:

- pixelWidth – width in pixels of the visible map window
- pixelHeight – height in pixels of the visible map window
- center – the DirectPosition center point of the map
- width – the map width in map width units
- scale – the ratio of map space to real world space
- envelope – the visible geographic map boundary

6.2.2 Events

6.2.2.1 Model and Rationale

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

For each control device (e.g. a mouse or keyboard), there is specialized `Event` object type. (In Java, for example, there are the KeyEvent and MouseEvent classes. Other language implementations of GO-1 will use different classes as appropriate.) When the device changes state (e.g., a mouse button is pressed), the system sends an instance of that `Event` to the Canvas.The actual mechanism through which the event is delivered to the Canvas is system and language dependent. However, a conforming implementation of the Canvas interface must handle the event as described below.

A Canvas maintains a stack of `EventHandlers` for each event type (keyboard, mouse, etc). When it receives an event, the Canvas passes it to the first Handler on the corresponding 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.

The stacks of event handlers on a Canvas are represented as instances of the `EventManager` interface. Methods on this interface allow the application to control what Handlers are on a given 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.

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

6.2.2.2 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.

6.2.2.3 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.

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 `EventHandler` object has a stack of Handlers. When it receives an event, the `EventHandler` 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 `EventHandler` 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.

6.2.2.4 EventManager

`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`.

6.2.2.5 Event Types

There are several event flags in GO-1. There are four required for Display Object conformance; GRAPHIC_CHANGED, GRAPHIC_SELECTED, GRAPHIC_DESELECTED, and GRAPHIC_DISPOSED.

6.3 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.

6.3.1 Graphic Overview

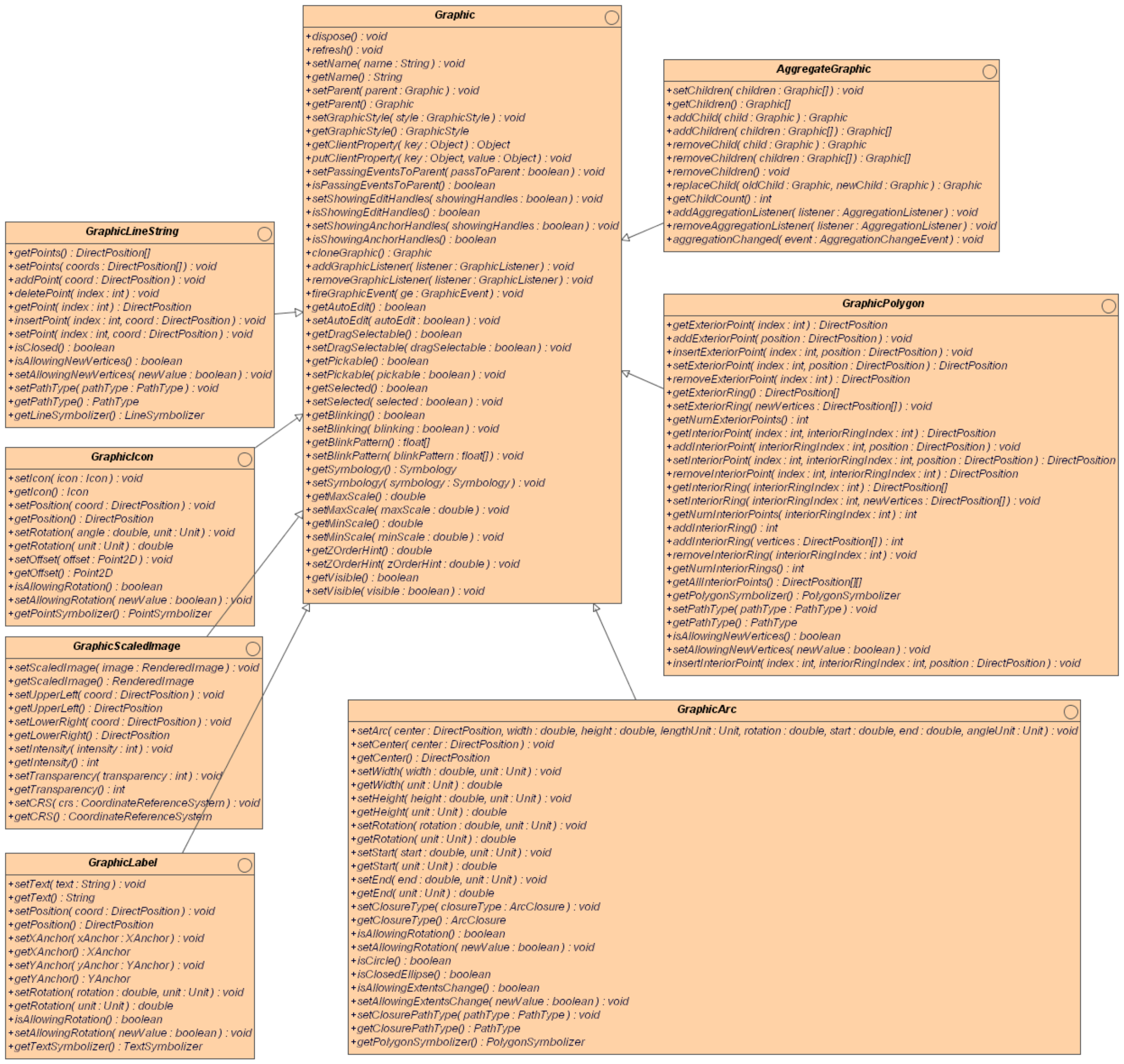

Graphic
+dispose() : void
+refresh() : void
+setName(name : String) : void
+getName() : String
+setParent(parent : Graphic) : void
+getParent() : Graphic
+setGraphicStyle(style : GraphicStyle) : void
+getGraphicStyle() : GraphicStyle
+getClientProperty(key : Object) : Object
+putClientProperty(key : Object, value : Object) : void
+setPassingEventsToParent(passToParent : boolean) : void
+isPassingEventsToParent() : boolean
+setShowingEditHandles(showingHandles : boolean) : void
+isShowingEditHandles() : boolean
+setShowingAnchorHandles(showingHandles : boolean) : void
+isShowingAnchorHandles() : boolean
+cloneGraphic() : Graphic
+addGraphicListener(listener : GraphicListener) : void
+removeGraphicListener(listener : GraphicListener) : void
+fireGraphicEvent(ge : GraphicEvent) : void
+getAutoEdit() : boolean
+setAutoEdit(autoEdit : boolean) : void
+getDragSelectable() : boolean
+setDragSelectable(dragSelectable : boolean) : void
+getPickable() : boolean
+setPickable(pickable : boolean) : void
+getSelected() : boolean
+setSelected(selected : boolean) : void
+getBlinking() : boolean
+setBlinking(blinking : boolean) : void
+getBlinkPattern() : float[]
+setBlinkPattern(blinkPattern : float[]) : void
+getSymbology() : Symbology
+setSymbology(symbology : Symbology) : void
+getMaxScale() : double
+setMaxScale(maxScale : double) : void
+getMinScale() : double
+setMinScale(minScale : double) : void
+getZOrderHint() : double
+setZOrderHint(zOrderHint : double) : void
+getVisible() : boolean
+setVisible(visible : boolean) : void
AggregateGraphic
+setChildren(children : Graphic[]) : void
+getChildren() : Graphic[]
+addChild(child : Graphic) : Graphic
+addChildren(children : Graphic[]) : Graphic[]
+removeChild(child : Graphic) : Graphic
+removeChildren(children : Graphic[]) : Graphic[]
+removeChildren() : void
+replaceChild(oldChild : Graphic, newChild : Graphic) : Graphic
+getChildCount() : int
+addAggregationListener(listener : AggregationListener) : void
+removeAggregationListener(listener : AggregationListener) : void
+aggregationChanged(event : AggregationChangeEvent) : void
GraphicLineString
+getPoints() : DirectPosition[]
+setPoints(coords : DirectPosition[]) : void
+addPoint(coord : DirectPosition) : void
+deletePoint(index : int) : void
+getPoint(index : int) : DirectPosition
+insertPoint(index : int, coord : DirectPosition) : void
+setPoint(index : int, coord : DirectPosition) : void
+isClosed() : boolean
+isAllowingNewVertices() : boolean
+setAllowingNewVertices(newValue : boolean) : void
+setPathType(pathType : PathType) : void
+getPathType() : PathType
+getLineSymbolizer() : LineSymbolizer
GraphicPolygon
+getExteriorPoint(index : int) : DirectPosition
+addExteriorPoint(position : DirectPosition) : void
+insertExteriorPoint(index : int, position : DirectPosition) : void
+setExteriorPoint(index : int, position : DirectPosition) : DirectPosition
+removeExteriorPoint(index : int) : DirectPosition
+getExteriorRing() : DirectPosition[]
+setExteriorRing(newVertices : DirectPosition[]) : void
+getNumExteriorPoints() : int
+getInteriorPoint(index : int, interiorRingIndex : int) : DirectPosition
+addInteriorPoint(interiorRingIndex : int, position : DirectPosition) : void
+setInteriorPoint(index : int, interiorRingIndex : int, position : DirectPosition) : DirectPosition
+removeInteriorPoint(index : int, interiorRingIndex : int) : DirectPosition
+getInteriorRing(interiorRingIndex : int) : DirectPosition[]
+setInteriorRing(interiorRingIndex : int, newVertices : DirectPosition[]) : void
+getNumInteriorPoints(interiorRingIndex : int) : int
+addInteriorRing() : int
+addInteriorRing(vertices : DirectPosition[]) : int
+removeInteriorRing(interiorRingIndex : int) : void
+getNumInteriorRings() : int
+getAllInteriorPoints() : DirectPosition[][]
+getPolygonSymbolizer() : PolygonSymbolizer
+setPathType(pathType : PathType) : void
+getPathType() : PathType
+isAllowingNewVertices() : boolean
+setAllowingNewVertices(newValue : boolean) : void
+insertInteriorPoint(index : int, interiorRingIndex : int, position : DirectPosition) : void
GraphicIcon
+setIcon(icon : Icon) : void
+getIcon() : Icon
+setPosition(coord : DirectPosition) : void
+getPosition() : DirectPosition
+setRotation(angle : double, unit : Unit) : void
+getRotation(unit : Unit) : double
+setOffset(offset : Point2D) : void
+getOffset() : Point2D
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean) : void
+getPointSymbolizer() : PointSymbolizer
GraphicScaledImage
+setScaledImage(image : RenderedImage) : void
+getScaledImage() : RenderedImage
+setUpperLeft(coord : DirectPosition) : void
+getUpperLeft() : DirectPosition
+setLowerRight(coord : DirectPosition) : void
+getLowerRight() : DirectPosition
+setIntensity(intensity : int) : void
+getIntensity() : int
+setTransparency(transparency : int) : void
+getTransparency() : int
+setCRS(crs : CoordinateReferenceSystem) : void
+getCRS() : CoordinateReferenceSystem
GraphicArc
+setArc(center : DirectPosition, width : double, height : double, lengthUnit : Unit, rotation : double, start : double, end : double, angleUnit : Unit) : void
+setCenter(center : DirectPosition) : void
+getCenter() : DirectPosition
+setWidth(width : double, unit : Unit) : void
+getWidth(unit : Unit) : double
+setHeight(height : double, unit : Unit) : void
+getHeight(unit : Unit) : double
+setRotation(rotation : double, unit : Unit) : void
+getRotation(unit : Unit) : double
+setStart(start : double, unit : Unit) : void
+getStart(unit : Unit) : double
+setEnd(end : double, unit : Unit) : void
+getEnd(unit : Unit) : double
+setClosureType(closureType : ArcClosure) : void
+getClosureType() : ArcClosure
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean) : void
+isCircle() : boolean
+isClosedEllipse() : boolean
+isAllowingExtentsChange() : boolean
+setAllowingExtentsChange(newValue : boolean) : void
+setClosurePathType(pathType : PathType) : void
+getClosurePathType() : PathType
+getPolygonSymbolizer() : PolygonSymbolizer
GraphicLabel
+setText(text : String) : void
+getText() : String
+setPosition(coord : DirectPosition) : void
+getPosition() : DirectPosition
+setXAnchor(xAnchor : XAnchor) : void
+getXAnchor() : XAnchor
+setYAnchor(yAnchor : YAnchor) : void
+getYAnchor() : YAnchor
+setRotation(rotation : double, unit : Unit) : void
+getRotation(unit : Unit) : double
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean) : void
+getTextSymbolizer() : TextSymbolizer

Figure 7 - Graphics

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 `GraphicLineString`, `GraphicPolygon`, `GraphicScaledImage`, `GraphicIcon`, `GraphicArc`, and `GraphicLabel`. `Aggregates` are collections of primitives as represented by `AggregateGraphic`.

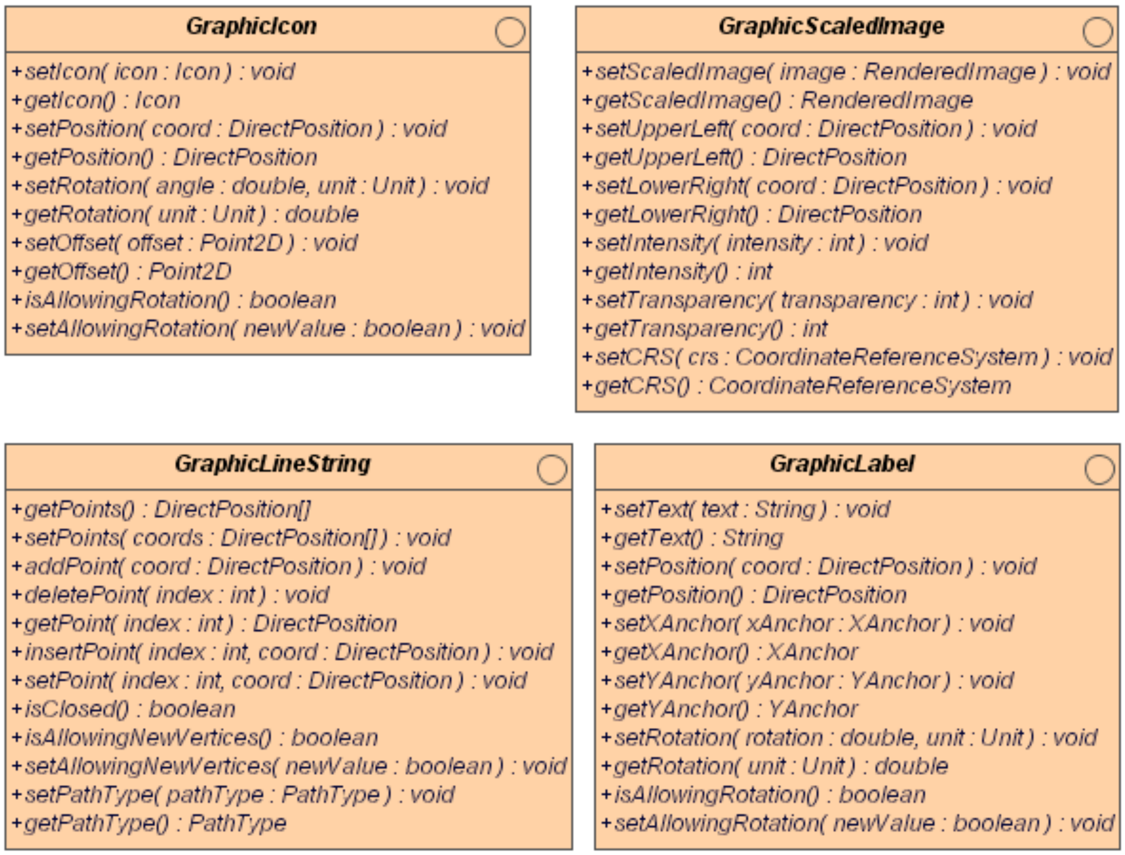
While the layout of each primitive graphic object corresponds to ISO-19107 `Geometry`, translation of `Geometry` objects to `Graphic` primitives is left to the implementation.

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.

6.3.2 Primitives



```
GraphicIcon
+setIcon( icon : Icon ) : void
+getIcon() : Icon
+setPosition( coord : DirectPosition ) : void
+getPosition() : DirectPosition
+setRotation( angle : double, unit : Unit ) : void
+getRotation( unit : Unit ) : double
+setOffset( offset : Point2D ) : void
+getOffset() : Point2D
+isAllowingRotation() : boolean
+setAllowingRotation( newValue : boolean ) : void

GraphicScaledImage
+setScaledImage( image : RenderedImage ) : void
+getScaledImage() : RenderedImage
+setUpperLeft( coord : DirectPosition ) : void
+getUpperLeft() : DirectPosition
+setLowerRight( coord : DirectPosition ) : void
+getLowerRight() : DirectPosition
+setIntensity( intensity : int ) : void
+getIntensity() : int
+setTransparency( transparency : int ) : void
+getTransparency() : int
+setCRS( crs : CoordinateReferenceSystem ) : void
+getCRS() : CoordinateReferenceSystem

GraphicLineString
+getPoints() : DirectPosition[]
+setPoints( coords : DirectPosition[] ) : void
+addPoint( coord : DirectPosition ) : void
+deletePoint( index : int ) : void
+getPoint( index : int ) : DirectPosition
+insertPoint( index : int, coord : DirectPosition ) : void
+setPoint( index : int, coord : DirectPosition ) : void
+isClosed() : boolean
+isAllowingNewVertices() : boolean
+setAllowingNewVertices( newValue : boolean ) : void
+setPathType( pathType : PathType ) : void
+getPathType() : PathType

GraphicLabel
+setText( text : String ) : void
+getText() : String
+setPosition( coord : DirectPosition ) : void
+getPosition() : DirectPosition
+setXAnchor( xAnchor : XAnchor ) : void
+getXAnchor() : XAnchor
+setYAnchor( yAnchor : YAnchor ) : void
+getYAnchor() : YAnchor
+setRotation( rotation : double, unit : Unit ) : void
+getRotation( unit : Unit ) : double
+isAllowingRotation() : boolean
+setAllowingRotation( newValue : boolean ) : void
```

Figure 8 – GraphicLabel, GraphicIcon, GraphicScaledImage, GraphicLineString

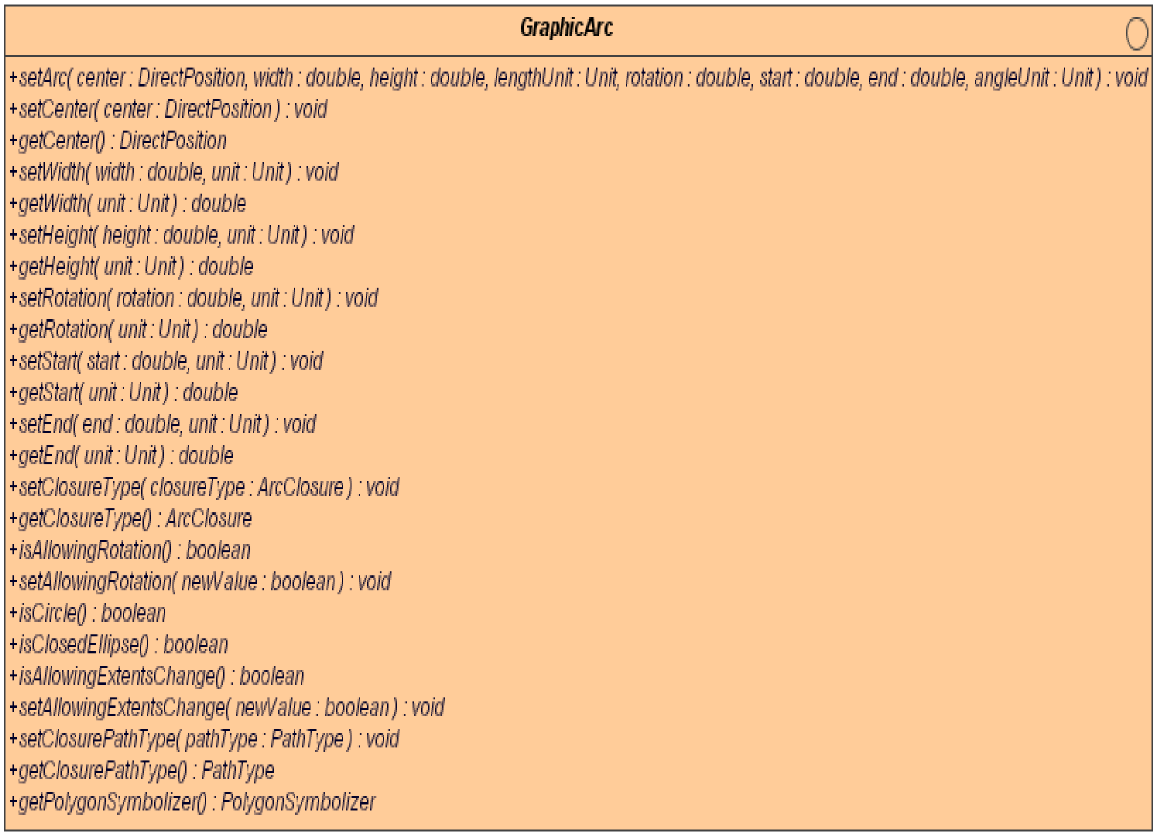
The palette of primitive shapes available to a Graphic is limited to a set that is sufficient for manipulation and rendering in graphical environments. Graphic objects themselves are subclassed according to the kind of geometry that they implement, and include the following:

`GraphicLineString` defines a common abstraction for implementations of 1-D lines made of one or more line segments. A settable `PathType` attribute determines the interpolation between segment endpoints.

`GraphicScaledImage` provides an abstraction for implementing projected images defined by an upper and lower corner 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`.

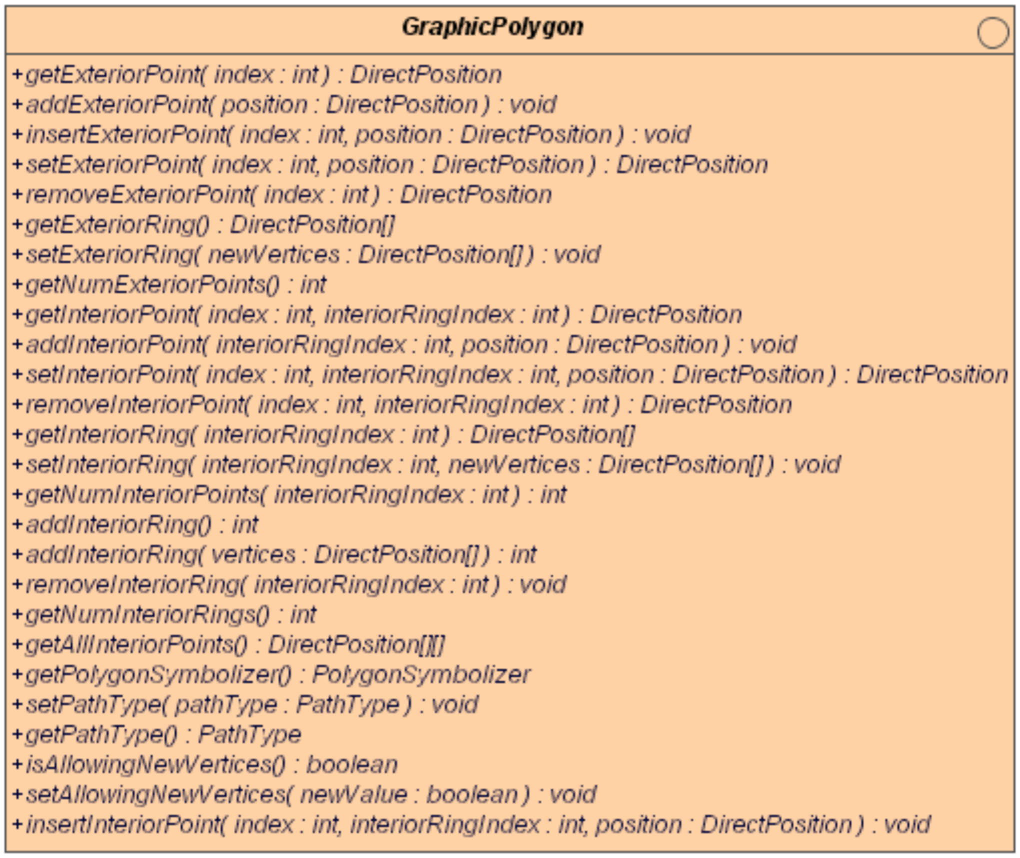
`GraphicLabel` defines a common abstraction for implementations to render text on a drawing surface. The position of the label in the `CoordinateReferenceSystem` is idealised as a single point attribute. The alignment the label to this point is specified by the x-anchor and y-anchor. The rotation of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


GraphicArc

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

Figure 9 - 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.



```
GraphicPolygon
+getExteriorPoint( index : int ) : DirectPosition
+addExteriorPoint( position : DirectPosition ) : void
+insertExteriorPoint( index : int, position : DirectPosition ) : void
+setExteriorPoint( index : int, position : DirectPosition ) : DirectPosition
+removeExteriorPoint( index : int ) : DirectPosition
+getExteriorRing() : DirectPosition[]
+setExteriorRing( newVertices : DirectPosition[] ) : void
+getNumExteriorPoints() : int
+getInteriorPoint( index : int, interiorRingIndex : int ) : DirectPosition
+addInteriorPoint( interiorRingIndex : int, position : DirectPosition ) : void
+setInteriorPoint( index : int, interiorRingIndex : int, position : DirectPosition ) : DirectPosition
+removeInteriorPoint( index : int, interiorRingIndex : int ) : DirectPosition
+getInteriorRing( interiorRingIndex : int ) : DirectPosition[]
+setInteriorRing( interiorRingIndex : int, newVertices : DirectPosition[] ) : void
+getNumInteriorPoints( interiorRingIndex : int ) : int
+addInteriorRing() : int
+addInteriorRing( vertices : DirectPosition[] ) : int
+removeInteriorRing( interiorRingIndex : int ) : void
+getNumInteriorRings() : int
+getAllInteriorPoints() : DirectPosition[][]
+getPolygonSymbolizer() : PolygonSymbolizer
+setPathType( pathType : PathType ) : void
+getPathType() : PathType
+isAllowingNewVertices() : boolean
+setAllowingNewVertices( newValue : boolean ) : void
+insertInteriorPoint( index : int, interiorRingIndex : int, position : DirectPosition ) : void
```

Figure 10 - GraphicPolygon

`GraphicPolygon` defines a common abstraction for a graphic representation of polygons with holes. A `GraphicPolygon` consists of an exterior ring of vertices, and a set of non-mutually overlapping interior rings of vertices. The exterior and interior rings of a polygon are defined by a list of `DirectPosition` objects. Technically speaking these rings are required to be closed such that first and last points coincide. However, this interface allows the user to create rings that are not closed. In such cases it is left up to the implementation to derive an additional segment between the first and last vertex. Implementers are responsible for validating that the arrays of points represent paths that do not cross themselves, that the interior rings are non-overlapping, and that they are indeed interior to the exterior ring.

6.3.3 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.

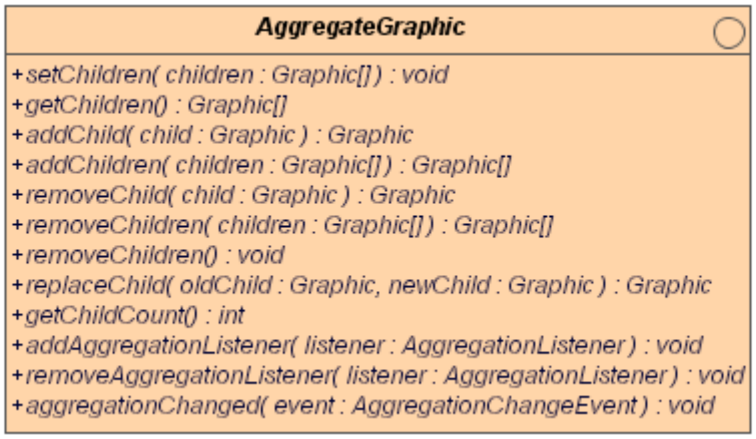

AggregateGraphic
+setChildren(children : Graphic[]) : void
+getChildren() : Graphic[]
+addChild(child : Graphic) : Graphic
+addChildren(children : Graphic[]) : Graphic[]
+removeChild(child : Graphic) : Graphic
+removeChildren(children : Graphic[]) : Graphic[]
+removeChildren() : void
+replaceChild(oldChild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

Figure 11 – AggregateGraphic

6.3.4 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.

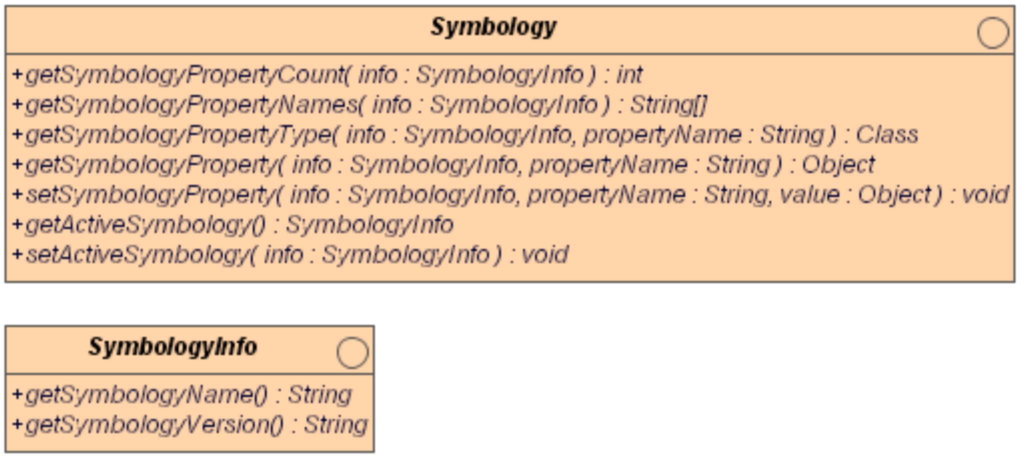

Symbology
+getSymbologyPropertyCount(info : SymbologyInfo) : int
+getSymbologyPropertyNames(info : SymbologyInfo) : String[]
+getSymbologyPropertyType(info : SymbologyInfo, propertyName : String) : Class
+getSymbologyProperty(info : SymbologyInfo, propertyName : String) : Object
+setSymbologyProperty(info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

Figure 12 – Symbology

6.3.4.1 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>
```

6.3.4.2 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);
```

6.3.5 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.

6.3.6 Graphic Attributes

6.3.6.1 Viewability

A Graphic may unconditionally be set invisible using its visible attribute. It can be made 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. The invisible state 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 Graphics Section].

6.3.6.2 Symbology

Symbology when defined supersedes any of the other symbolizer objects [see Symbology Section].

6.3.6.3 Highlight

Highlight attributes control whether a Graphic can blink, and if so, at what rate through blinking and blinkPattern.

6.3.6.4 Editability

Editability attributes allow the Graphic to be edited or selected through user interaction with the Canvas, usually using mouse gestures. Editability includes autoEdit, dragSelectable, selected, and pickable. See Graphic Editability section for a more thorough description.

6.3.7 GraphicStyle

6.3.7.1 Relationship to Graphic

The GraphicStyle class allows a Graphic to be visually decorated. Each Graphic contains a GraphicStyle object. If a particular property on a GraphicStyle is not set then a default value is used.

Most rendering style attributes are defined in one of the four subclasses of GraphicStyle, each of which is tailored for the needs of specific graphic types. LineSymbolizer maintains attributes that apply to line-stroked Graphics, such as GraphicLineString and open GraphicArcs. PolygonSymbolizer extends LineSymbolizer to maintain fill attributes for filled stroked objects, such as GraphicPolygon and closed GraphicArcs. TextSymbolizer governs rendering of labels. PointSymbolizer is used for rendering other objects that are anchored to a single point and painted in a pixel-wise manner without regard to map location; it is the GraphicStyle used by GraphicIcon.

A number of basic rendering attributes are maintained within the `Graphic` itself. These include visibility and Z order, selection state, pickability and drag selectability, editability, and highlighting.

Outside of `GraphicStyle`, the `Symbology` interface governs rendering that will conform to a defined symbology set. A `Graphic` may maintain a `Symbology` object, and when it does so, this overrides any conflicting style attributes set within the `GraphicStyle` object.

6.3.7.2 Relationship to OGC SLD

The `GraphicStyle` class and its subclasses have been developed to be as symmetric as possible with SLD [14]. As such, SLD styles have many similar properties to a `GraphicStyle`. However, SLD style properties, such as color or line thickness, can be functions of feature properties, and thus vary from feature to feature. `GraphicStyles` are fixed until explicit API calls change their values. `GraphicStyle` contains concepts not found in SLD (e.g. Text BackgroundColor, `ArrowStyle`, `DashArray`, `FillStyle`, `FillPattern`). It is recommended that the SLD specification be expanded to express these concepts. See Future Work section.

The SLD-analogous interfaces are:

- `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.

6.3.7.3 GraphicStyle Elements

The following table details the properties of the various symbolizer interfaces, their types, and their default values.

Interface	Element	Type	Default
`LineSymbolizer`	`StrokeBeginArrowStyle`	`ArrowStyle`	`ArrowStyle.NONE`
	`StrokeColor`	`Color`	`Color.BLACK`
	`StrokeDashArray`	`DashArray`	`DashArray.NONE`
	`StrokeDashOffset`	`float`	`0.0`
	`StrokeEndArrowStyle`	`ArrowStyle`	`ArrowStyle.NONE`

	StrokeFillBackgroundColor	Color	Color.GRAY
	StrokeFillColor	Color	Color.BLACK
	StrokeFillGradientPoints	float[2]	*N/A*
	StrokeFillOpacity	float	1.0
	StrokeFillPattern	FillPattern	FillPattern.NONE
	StrokeFillStyle	FillStyle	FillStyle.SOLID
	StrokeLineCap	LineCap	LineCap.BUTT
	StrokeLineGap	float	10.0
	StrokeLineJoin	LineJoin	LineJoin.BEVEL
	StrokeLineStyle	LineStyle	LineStyle.SINGLE
	StrokeOpacity	float	1.0
	StrokePattern	FillPattern	FillPattern.NONE
	StrokeWidth	float	1.0
PolygonSymbolizer	FillBackgroundColor	Color	Color.GRAY
	FillColor	Color	Color.BLACK
	FillGradientPoints	float[2]	*N/A*
	FillOpacity	float	1.0
	FillPattern	FillPattern	FillPattern.NONE
	FillStyle	FillStyle	FillStyle.SOLID
	StrokeBeginArrowStyle	ArrowStyle	ArrowStyle.NONE
	StrokeColor	Color	Color.BLACK
	StrokeDashArray	DashArray	DashArray.NONE
	StrokeDashOffset	Float	0.0
	StrokeEndArrowStyle	ArrowStyle	ArrowStyle.NONE
	StrokeFillBackgroundColor	Color	Color.GRAY
	StrokeFillColor	Color	Color.BLACK
	StrokeFillGradientPoints	float[2]	*N/A*
	StrokeFillOpacity	float	1.0
	StrokeFillPattern	FillPattern	FillPattern.NONE
	StrokeFillStyle	FillStyle	FillStyle.SOLID
	StrokeLineCap	LineCap	LineCap.BUTT
	StrokeLineGap	float	10.0
	StrokeLineJoin	LineJoin	LineJoin.BEVEL
	StrokeLineStyle	LineStyle	LineStyle.SINGLE
	StrokeOpacity	float	1.0
	StrokePattern	FillPattern	FillPattern.NONE
	StrokeWidth	float	1.0
PointSymbolizer	FillBackgroundColor	Color	Color.GRAY
	FillColor	Color	Color.BLACK
	FillGradientPoints	float[2]	*N/A*
	FillOpacity	float	1.0
	FillPattern	FillPattern	FillPattern.NONE
	FillStyle	FillStyle	FillStyle.SOLID
	Mark	Mark	Mark.CIRCLE
	Opacity	float	1.0
	Rotation	float	0.0
	Size	float	16.0
	StrokeBeginArrowStyle	ArrowStyle	ArrowStyle.NONE
	StrokeColor	Color	Color.BLACK
	StrokeDashArray	DashArray	DashArray.NONE
	StrokeDashOffset	Float	0.0
	StrokeEndArrowStyle	ArrowStyle	ArrowStyle.NONE
	StrokeFillBackgroundColor	Color	Color.GRAY
	StrokeFillColor	Color	Color.BLACK
	StrokeFillGradientPoints	float[2]	*N/A*
	StrokeFillOpacity	float	1.0

	StrokeFillPattern	FillPattern	FillPattern.NONE
	StrokeFillStyle	FillStyle	FillStyle.SOLID
	StrokeLineCap	LineCap	LineCap.BUTT
	StrokeLineGap	float	10.0
	StrokeLineJoin	LineJoin	LineJoin.BEVEL
	StrokeLineStyle	LineStyle	LineStyle.SINGLE
	StrokeOpacity	float	1.0
	StrokePattern	FillPattern	FillPattern.NONE
	StrokeWidth	float	1.0
TextSymbolizer	BackgroundColor	Color	*N/A*
	FillBackgroundColor	Color	Color.WHITE
	FillColor	Color	Color.BLACK
	FillGradientPoints	float[2]	*N/A*
	FillOpacity	float	1.0
	FillPattern	FillPattern	FillPattern.NONE
	FillStyle	FillStyle	FillStyle.SOLID
	Font	Font	*N/A*
	HaloRadius	Float	1.0
	Label	String	*N/A*
	LabelRotation	float	0.0
	LabelShowLabel	boolean	false
	LabelXAnchor	Xanchor	Xanchor.LEFT
	LabelXDisplacement	float	0.0
	LabelYAnchor	YAnchor	YAnchor.MIDDLE
	LabelYDisplacement	float	0.0

Table 2 – GraphicStyle Elements

6.3.7.4 GraphicStyle Events

`GraphicStyleListeners` may be registered on any `GraphicStyle` object to be notified of attribute changes. Implementations may do so for several purposes, such as knowing to repaint a `Graphic` for modification of a styling attribute. This may also be used for propagating a change in an attribute among `Graphics` that do not share the same `GraphicStyle` object. For example, note that `AggregateGraphic` objects are required to maintain a `GraphicStyle` object even though its attributes will not apply to rendering any of its constituent `Graphics`. However, it would be allowable to assign it an object of one of the four symbolizer types. Children that have diverse `GraphicStyles` could listen for changes to a certain attribute (say, stroke color), and update their own styles if such a `GraphicStyleEvent` is detected.

6.3.7.5 Graphic Selectability

Many systems for displaying geometric shapes allow the user to edit those shapes by using the mouse to change, add, or remove vertices. For such systems, the editability attributes of a Graphic allow the user to toggle on and off this behaviour. Here is a description of the relevant accessor methods:

- get/setDragSelectable - Many systems allow the user to select objects on the screen by dragging a bounding area, such as a box, with a mouse gesture. This property, if set to true, allows the graphic to be included in selection sets created in this way.

- get/setPickable - Many systems allow the user to select individual objects by gesturing (mouse clicking) over the object. This property, if true, allows the graphic to be selected in this way.

- get/setSelected - For systems that allow individual graphics to be selected in the display (usually with a mouse gesture or by choosing an item from a list), this property sets or queries whether the given graphic is currently selected. Systems that have no notion of selection may always have a value of false for this property and the set method will have no effect.

6.3.8 Graphic Events

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`) can register `AggregationListeners` to listen for `AggregationChangeEvent's`. These events are notifications when elements are added, removed, or (if applicable) reordered within the aggregation.

Whenever a `Graphic` is selected, a GraphicChangeEvent must be fired by the implementation.

6.4 Spatial Objects

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

6.4.1 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].

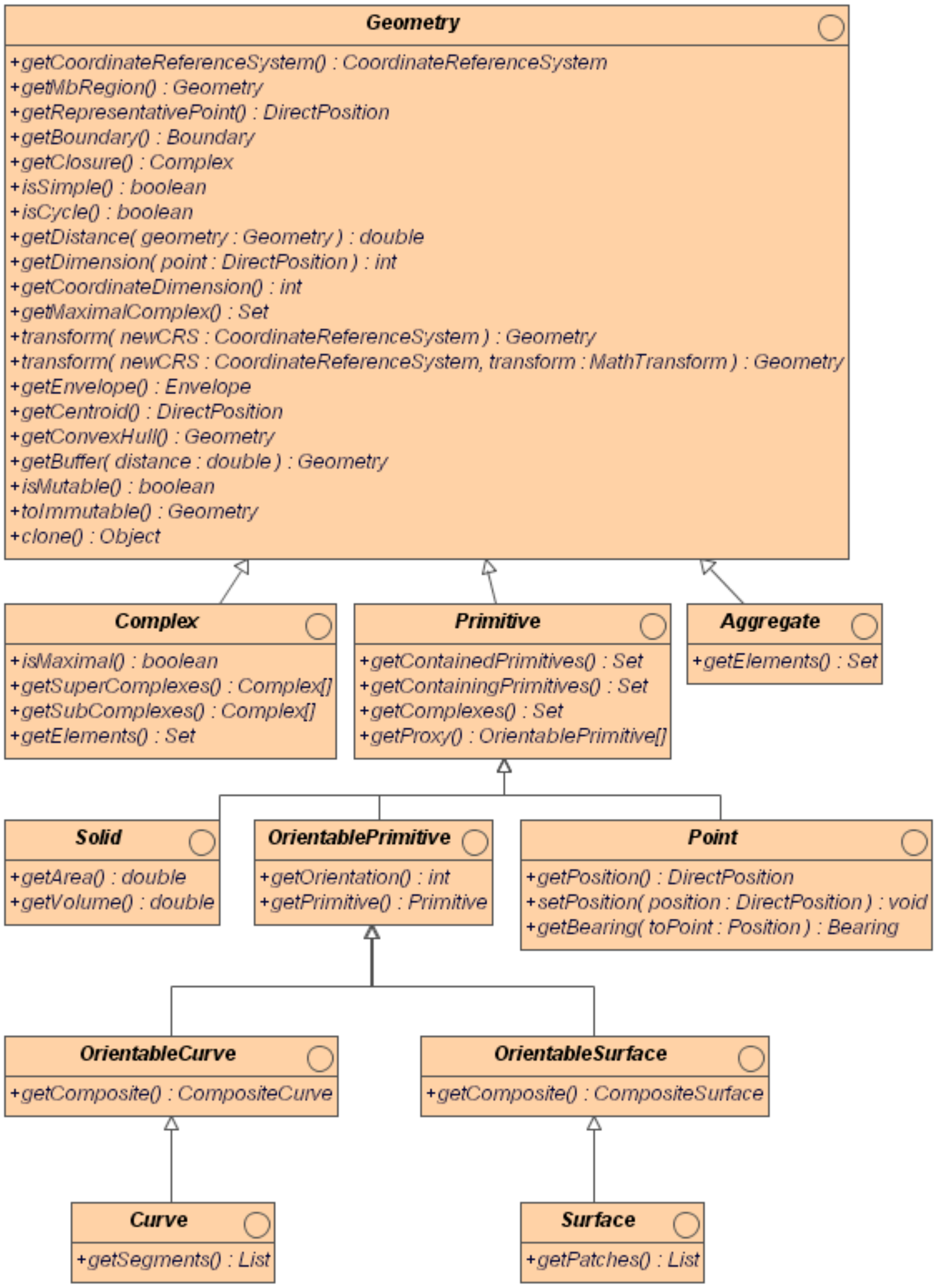

Geometry
+getCoordinateReferenceSystem() : CoordinateReferenceSystem
+getMbRegion() : Geometry
+getRepresentativePoint() : DirectPosition
+getBoundary() : Boundary
+getClosure() : Complex
+isSimple() : boolean
+isCycle() : boolean
+getDistance(geometry : Geometry) : double
+getDimension(point : DirectPosition)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

Figure 13 – 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.

6.4.1.1 DirectPosition

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

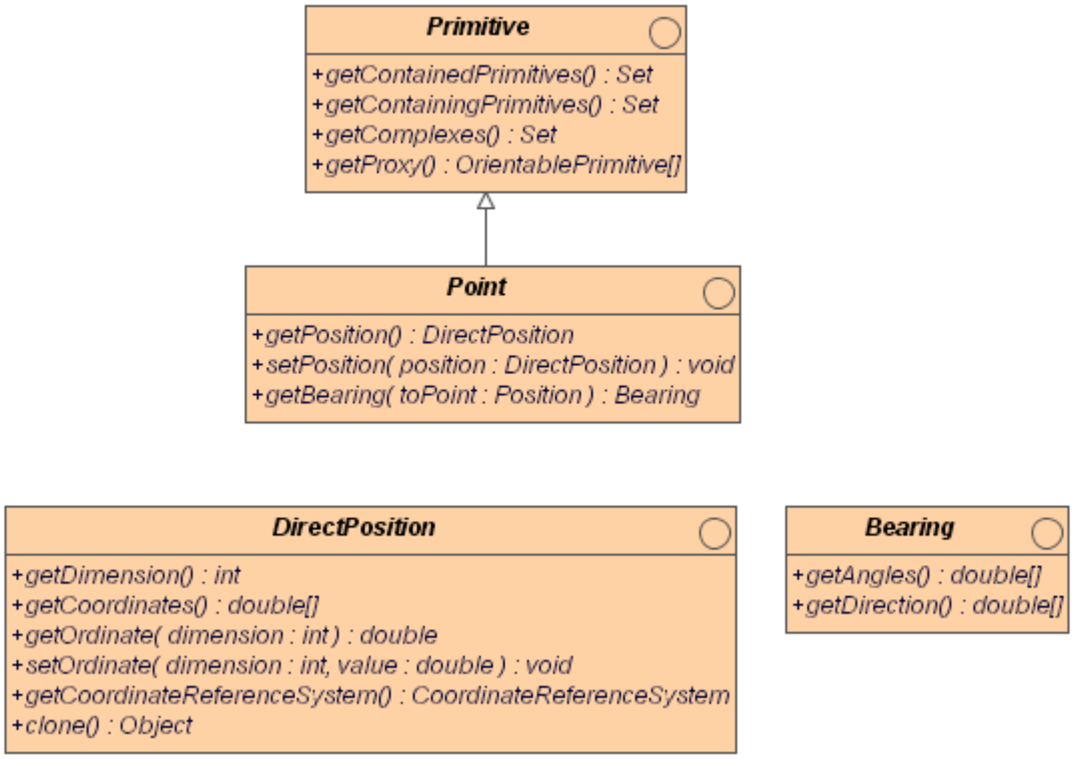

Primitive
+getContainedPrimitives() : Set
+getContainingPrimitives() : Set
+getComplexes() : Set
+getProxy() : OrientablePrimitive[]
Point
+getPosition() : DirectPosition
+setPosition(position : DirectPosition) : void
+getBearing(toPoint : Position) : Bearing
DirectPosition
+getDimension() : int
+getCoordinates()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

Figure 14 – 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.

6.4.1.2 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.

`Curves` 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 or equal to unity. In two dimensions, a non-negative eccentricity of less than one will generate a closed ellipse. An eccentricity of exactly one results in a parabola, and a negative eccentricity yields a hyperbola.

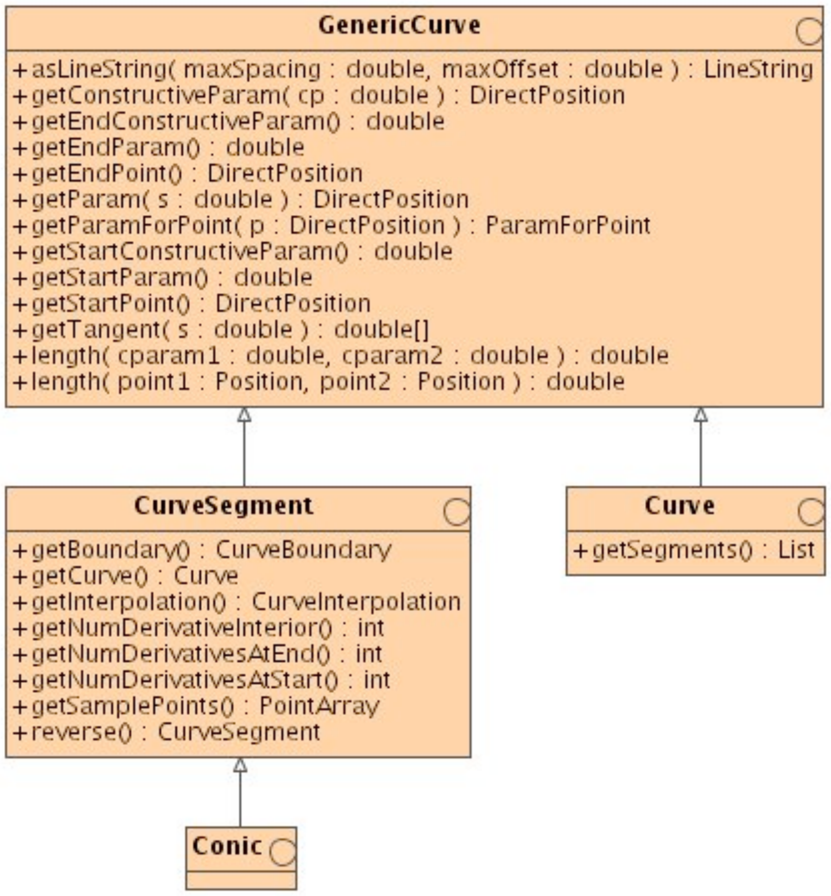

GenericCurve
+asLineString(maxSpacing : double, maxOffset : double) : LineString
+getConstructiveParam(cp : double) : DirectPosition
+getEndConstructiveParam() : double
+getEndParam() : double
+getEndPoint() : DirectPosition
+getParam(s : double) : DirectPosition
+getParamForPoint(p : DirectPosition) : 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 15 – CurveSegment and Conic

6.4.1.3 CompositeCurve and Ring

A `CompositeCurve` is a `Composite` with all the geometric properties of a `Curve`. Essentially, a composite curve is a list of `OrientableCurves` 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*.

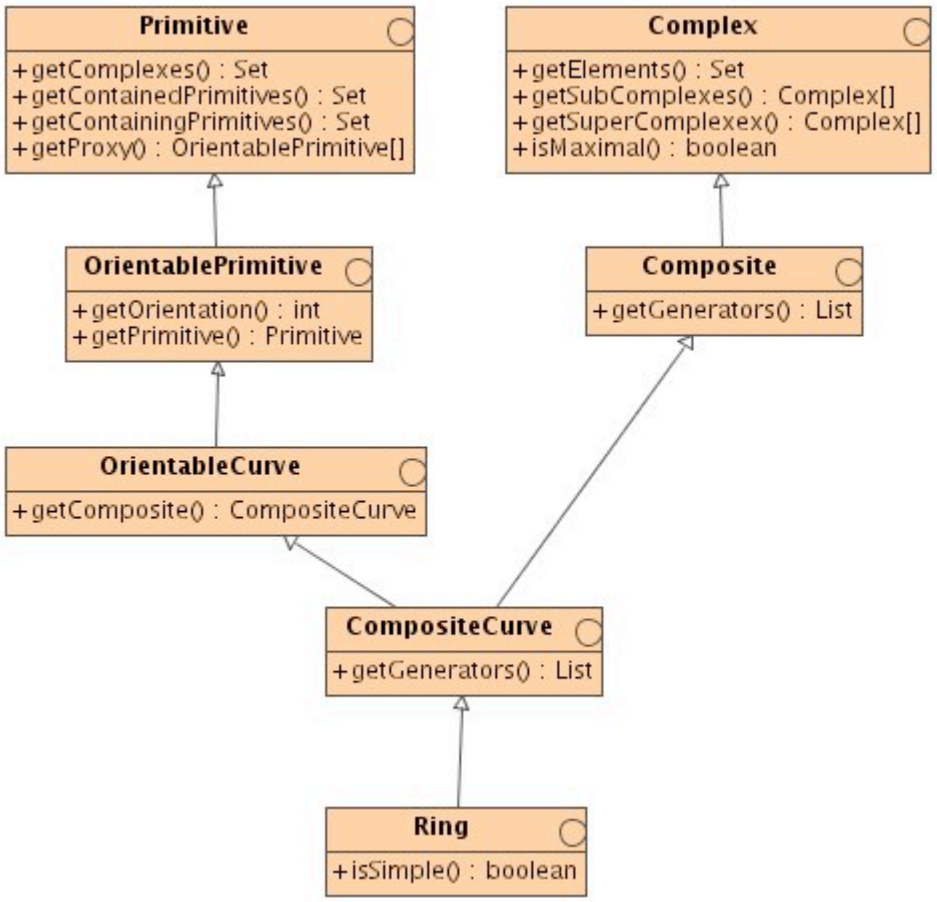


Figure 16 – CompositeCurve and Ring

6.4.1.4 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.

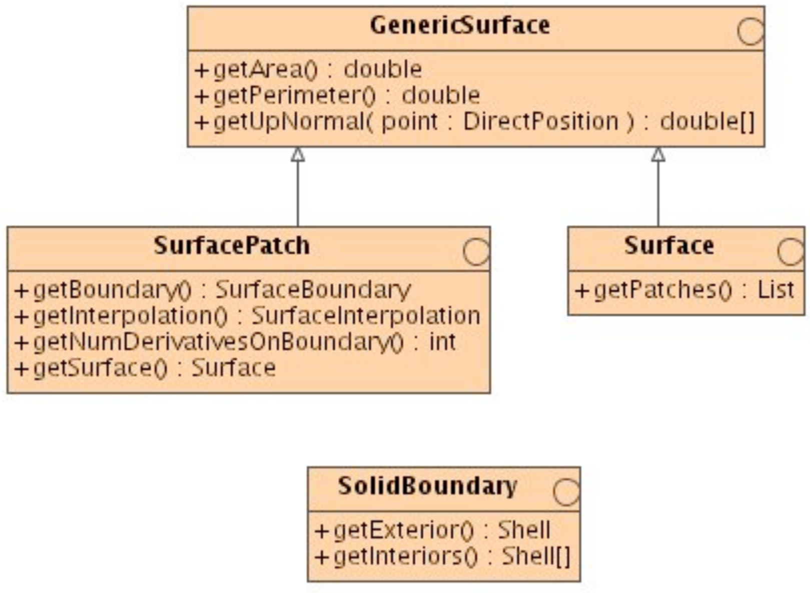

GenericSurface
+getArea() : double
+getPerimeter() : double
+getUpNormal(point : DirectPosition) : double[]
SurfacePatch
+getBoundary() : SurfaceBoundary
+getInterpolation() : SurfaceInterpolation
+getNumDerivativesOnBoundary() : int
+getSurface() : Surface
Surface
+getPatches() : List
SolidBoundary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

Figure 17 - SurfaceBoundary

6.4.1.5 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.

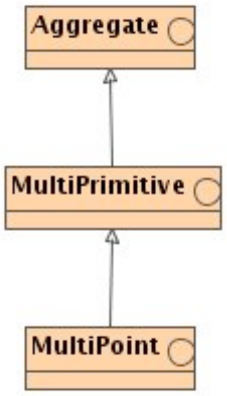


Figure 18 - Aggregate

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.

6.4.1.6 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.

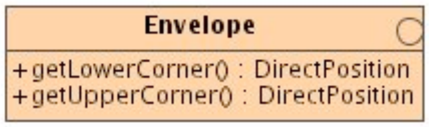

Envelope
+getLowerCorner() : DirectPosition
+getUpperCorner() : DirectPosition

Figure 19 - Envelope

6.4.1.7 Geometry Mutability

The geometric primitive interfaces defined in this specification include methods that allow the API user to change the underlying geometry of a primitive over time. For example, the Point interface includes a setPosition(...) method. However, the writers of this specification recognize that not all implementations of geometry will be changeable. For this reason, the Geometry base class also includes a method, isMutable(), that can be called at runtime to determine whether the underlying implementation allows changes of the geometry. A return value of true indicates that the object can be changed by calling its set methods. A return value of false indicates that the set methods will throw an UnsupportedOperationException and that the values in the object will never change. If a user of a geometry requires a copy that will never change, he can make a private copy using the clone() method, or the toImmutable() method can be used to make a copy that is guaranteed never to change.

6.4.2 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].

6.4.2.1 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.

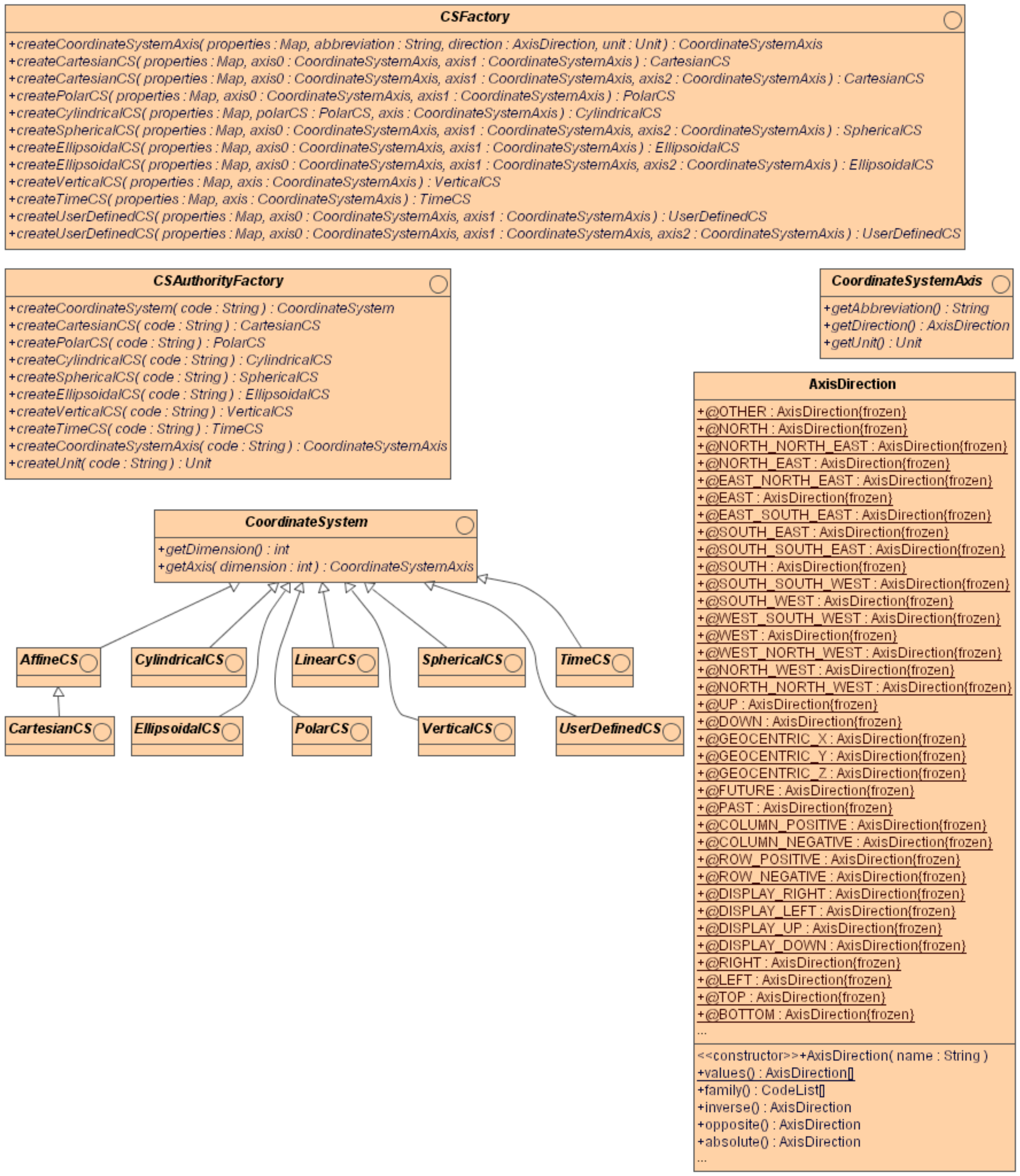

CSFactory
+createCoordinateSystemAxis(properties : Map, abbreviation : String, direction : AxisDirection, unit : Unit) : CoordinateSystemAxis
+createCartesianCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis) : CartesianCS
+createCartesianCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis, axis2 : CoordinateSystemAxis) : CartesianCS
+createPolarCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis) : PolarCS
+createCylindricalCS(properties : Map, polarCS : PolarCS, axis : CoordinateSystemAxis) : CylindricalCS
+createSphericalCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis, axis2 : CoordinateSystemAxis) : SphericalCS
+createEllipsoidalCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis) : EllipsoidalCS
+createEllipsoidalCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis, axis2 : CoordinateSystemAxis) : EllipsoidalCS
+createVerticalCS(properties : Map, axis : CoordinateSystemAxis) : VerticalCS
+createTimeCS(properties : Map, axis : CoordinateSystemAxis) : TimeCS
+createUserDefinedCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis) : UserDefinedCS
+createUserDefinedCS(properties : Map, axis0 : CoordinateSystemAxis, axis1 : CoordinateSystemAxis, axis2 : CoordinateSystemAxis) : UserDefinedCS
CSAuthorityFactory
+createCoordinateSystem(code : String) : CoordinateSystem
+createCartesianCS(code : String) : CartesianCS
+createPolarCS(code : String) : PolarCS
+createCylindricalCS(code : String) : CylindricalCS
+createSphericalCS(code : String) : SphericalCS
+createEllipsoidalCS(code : String) : EllipsoidalCS
+createVerticalCS(code : String) : VerticalCS
+createTimeCS(code : String) : TimeCS
+createCoordinateSystemAxis(code : String) : CoordinateSystemAxis
+createUnit(code : String) : Unit
CoordinateSystemAxis
+getAbbreviation() : String
+getDirection() : AxisDirection
+getUnit() : Unit
CoordinateSystem
+getDimension() : int
+getAxis(dimension : int) : CoordinateSystemAxis
AffineCS
CylindricalCS
LinearCS
SphericalCS
TimeCS
CartesianCS
EllipsoidalCS
PolarCS
VerticalCS
UserDefinedCS
AxisDirection
+@OTHER : AxisDirection{frozen}
+@NORTH : AxisDirection{frozen}
+@NORTH_NORTH_EAST : AxisDirection{frozen}
+@NORTH_EAST : AxisDirection{frozen}
+@EAST_NORTH_EAST : AxisDirection{frozen}
+@EAST : AxisDirection{frozen}
+@EAST_SOUTH_EAST : AxisDirection{frozen}
+@SOUTH_EAST : AxisDirection{frozen}
+@SOUTH_SOUTH_EAST : AxisDirection{frozen}
+@SOUTH : AxisDirection{frozen}
+@SOUTH_SOUTH_WEST : AxisDirection{frozen}
+@SOUTH_WEST : AxisDirection{frozen}
+@WEST_SOUTH_WEST : AxisDirection{frozen}
+@WEST : AxisDirection{frozen}
+@WEST_NORTH_WEST : AxisDirection{frozen}
+@NORTH_WEST : AxisDirection{frozen}
+@NORTH_NORTH_WEST : AxisDirection{frozen}
+@UP : AxisDirection{frozen}
+@DOWN : AxisDirection{frozen}
+@GEOCENTRIC_X : AxisDirection{frozen}
+@GEOCENTRIC_Y : AxisDirection{frozen}
+@GEOCENTRIC_Z : AxisDirection{frozen}
+@FUTURE : AxisDirection{frozen}
+@PAST : AxisDirection{frozen}
+@COLUMN_POSITIVE : AxisDirection{frozen}
+@COLUMN_NEGATIVE : AxisDirection{frozen}
+@ROW_POSITIVE : AxisDirection{frozen}
+@ROW_NEGATIVE : AxisDirection{frozen}
+@DISPLAY_RIGHT : AxisDirection{frozen}
+@DISPLAY_LEFT : AxisDirection{frozen}
+@DISPLAY_UP : AxisDirection{frozen}
+@DISPLAY_DOWN : AxisDirection{frozen}
+@RIGHT : AxisDirection{frozen}
+@LEFT : AxisDirection{frozen}
+@TOP : AxisDirection{frozen}
+@BOTTOM : AxisDirection{frozen}
...
<<constructor>>+AxisDirection(name : String)
+values() : AxisDirection[]
+family() : CodeList[]
+inverse() : AxisDirection
+opposite() : AxisDirection
+absolute() : AxisDirection
...

Figure 20 - 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 multi-dimensional 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.

`TimeCS` 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.

6.4.2.2 Reference System

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

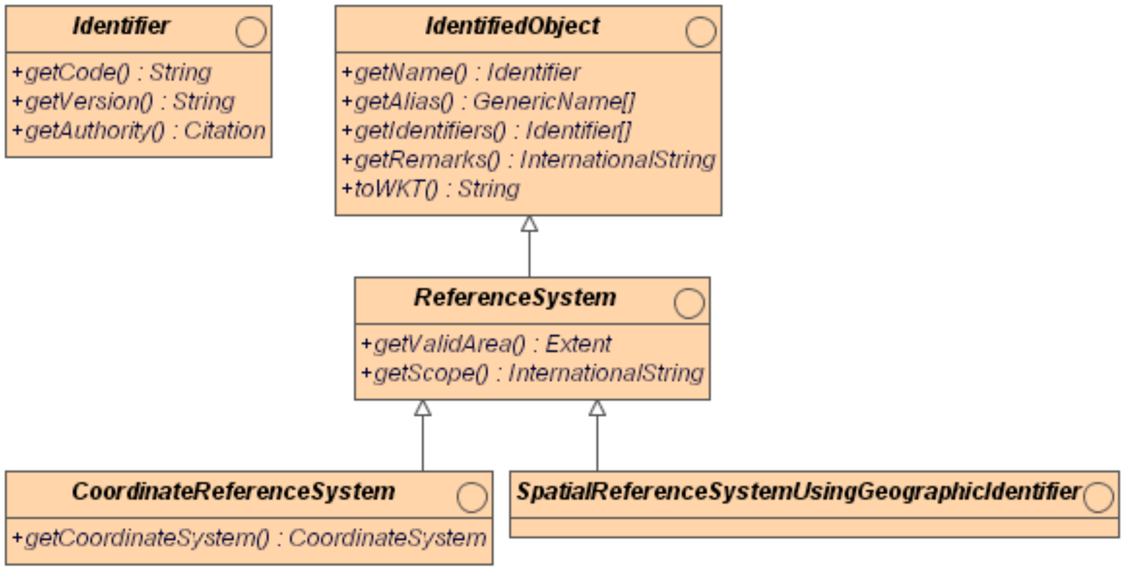

Identifier
+getCode() : String
+getVersion() : String
+getAuthority() : Citation
IdentifiedObject
+getName() : Identifier
+getAlias() : GenericName[]
+getIdentifiers() : Identifier[]
+getRemarks() : InternationalString
+toWKT() : String
ReferenceSystem
+getValidArea() : Extent
+getScope() : InternationalString
CoordinateReferenceSystem
+getCoordinateSystem() : CoordinateSystem
SpatialReferenceSystemUsingGeographicIdentifier

Figure 2 - 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.

6.4.2.3 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.

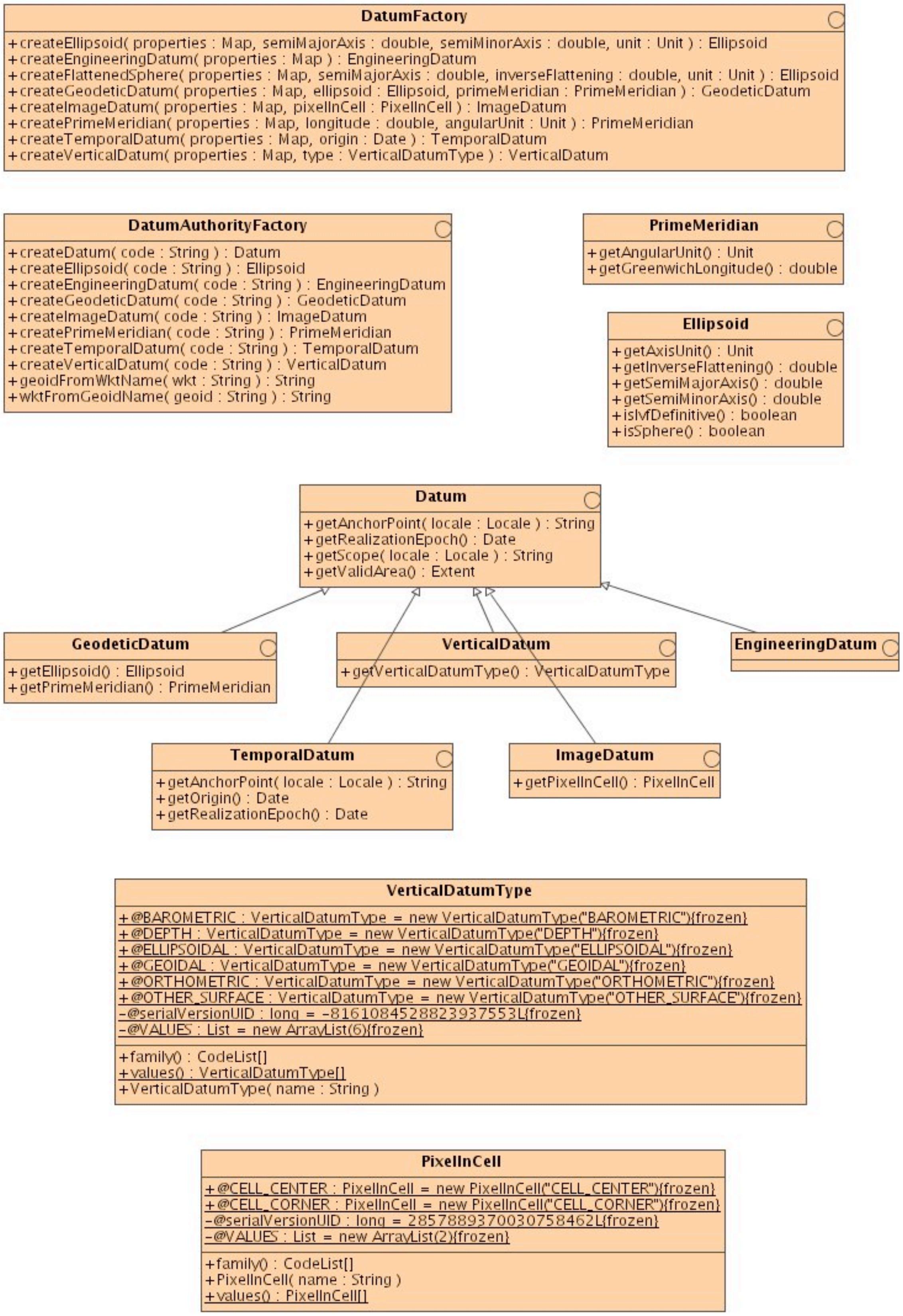

DatumFactory
+createEllipsoid(properties : Map, semiMajorAxis : double, semiMinorAxis : double, unit : Unit) : Ellipsoid
+createEngineeringDatum(properties : Map) : EngineeringDatum
+createFlattenedSphere(properties : Map, semiMajorAxis : double, inverseFlattening : double, unit : Unit) : Ellipsoid
+createGeodeticDatum(properties : 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 21 - 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.

6.4.2.4 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).

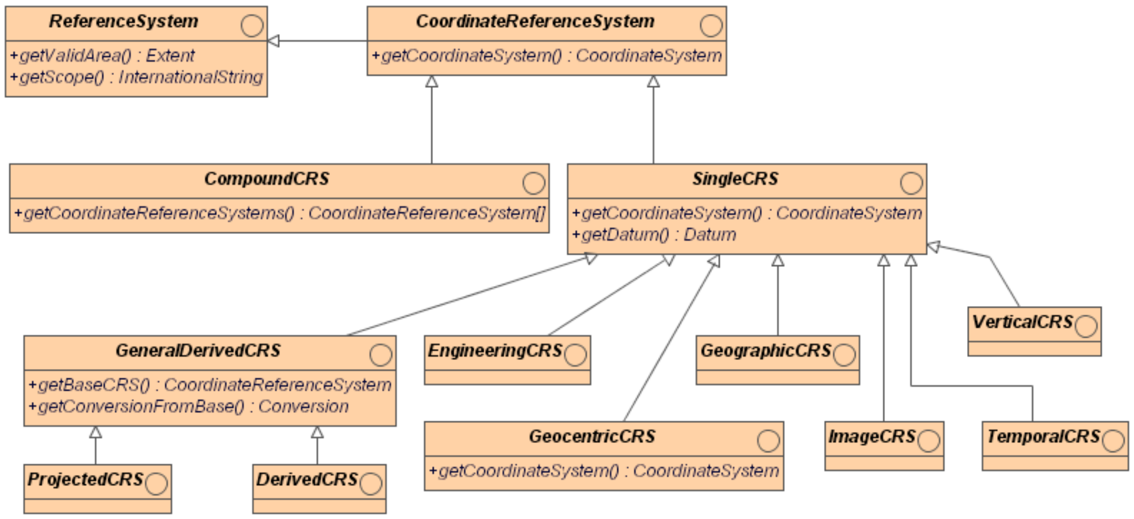

ReferenceSystem
+getValidArea() : Extent
+getScope() : InternationalString
CoordinateReferenceSystem
+getCoordinateSystem() : CoordinateSystem
CompoundCRS
+getCoordinateReferenceSystems() : CoordinateReferenceSystem[]
SingleCRS
+getCoordinateSystem() : CoordinateSystem
+getDatum() : Datum
GeneralDerivedCRS
+getBaseCRS() : CoordinateReferenceSystem
+getConversionFromBase() : Conversion
EngineeringCRS
GeographicCRS
VerticalCRS
GeocentricCRS
+getCoordinateSystem() : CoordinateSystem
ImageCRS
TemporalCRS
ProjectedCRS
DerivedCRS

Figure 22 - Coordinate Reference System model from Topic 2

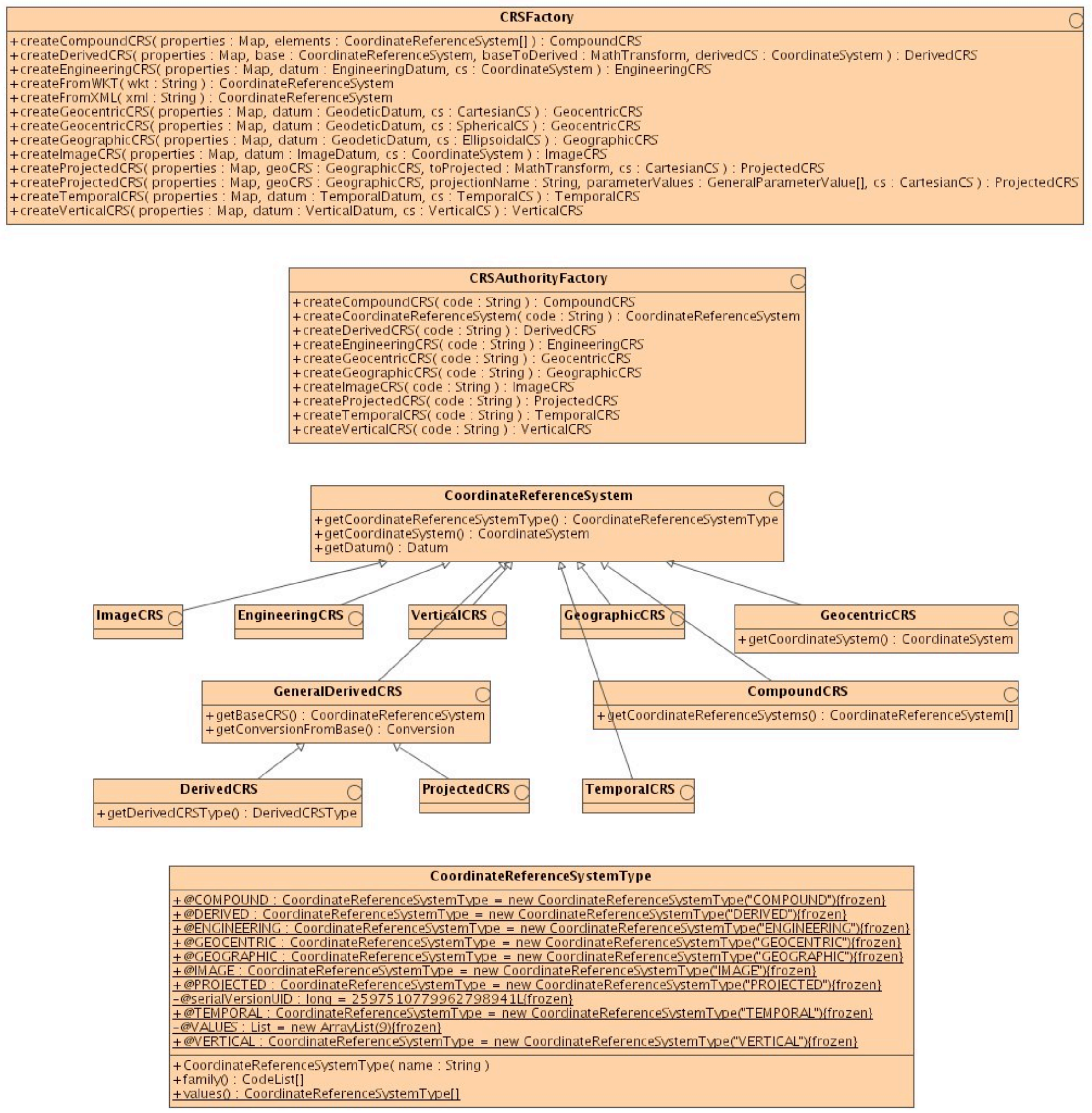

CRSFactory
+createCompoundCRS(properties : Map, elements : CoordinateReferenceSystem[]) : CompoundCRS
+createDerivedCRS(properties : Map, base : CoordinateReferenceSystem, baseToDerived : MathTransform, derivedCS : CoordinateSystem) : DerivedCRS
+createEngineeringCRS(properties : Map, datum : EngineeringDatum, 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 3 - 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.

6.4.2.5 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.

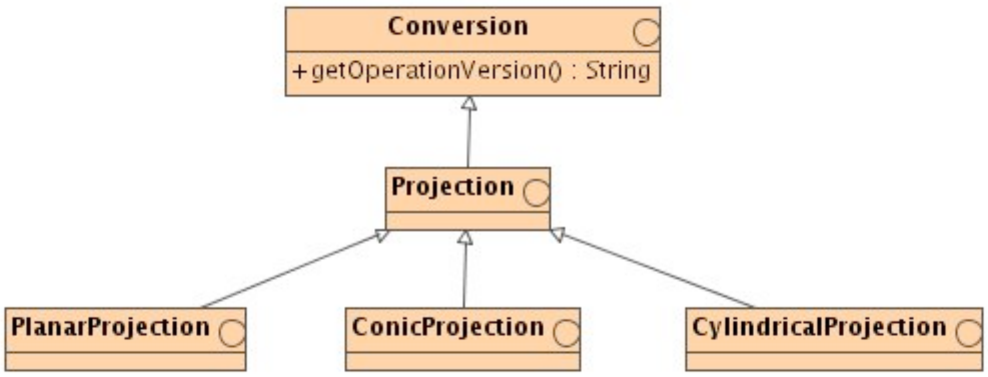
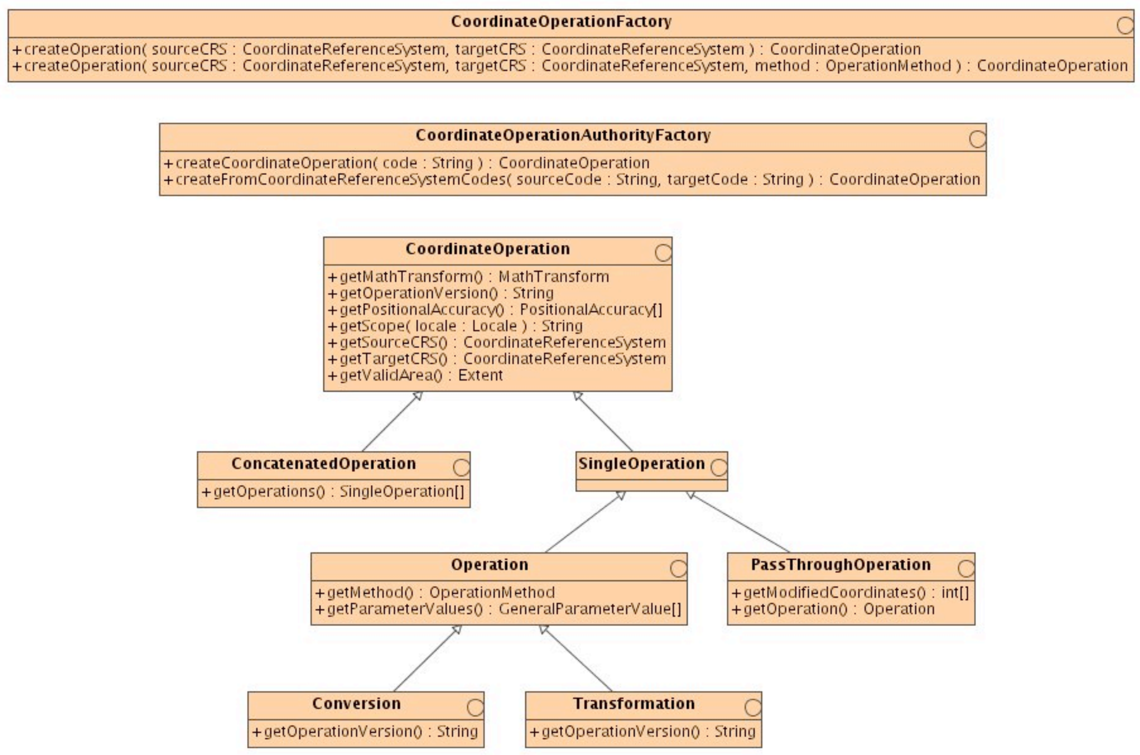


Figure 4 - Projection

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.

6.4.2.6 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 24 - 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.

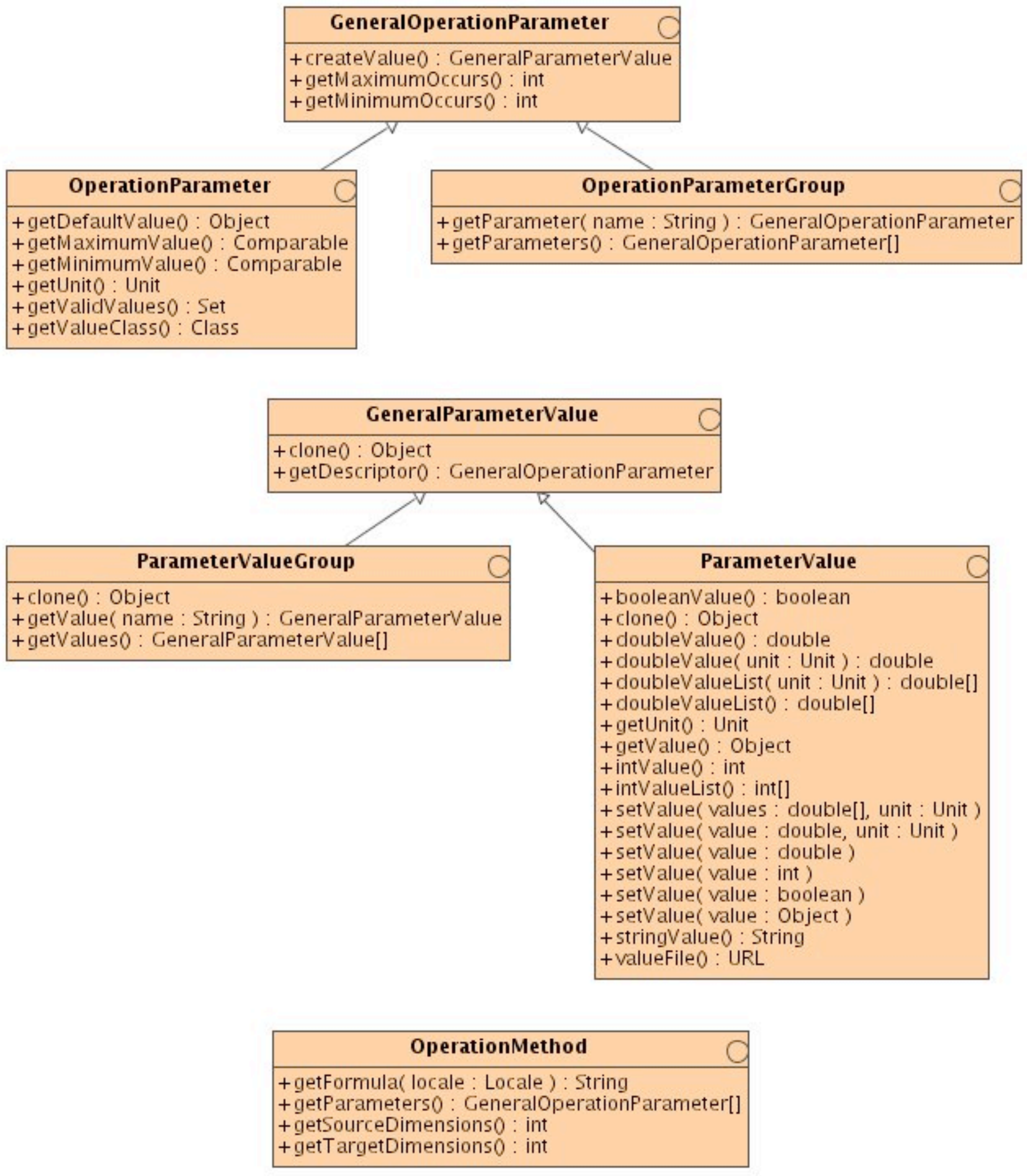

GeneralOperationParameter
+createValue() : GeneralParameterValue
+getMaximumOccurs() : int
+getMinimumOccurs() : int
OperationParameter
+getDefaultValue() : Object
+getMaximumValue() : Comparable
+getMinimumValue() : Comparable
+getUnit() : Unit
+getValidValues() : Set
+getValueClass() : Class
OperationParameterGroup
+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 25 - 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.

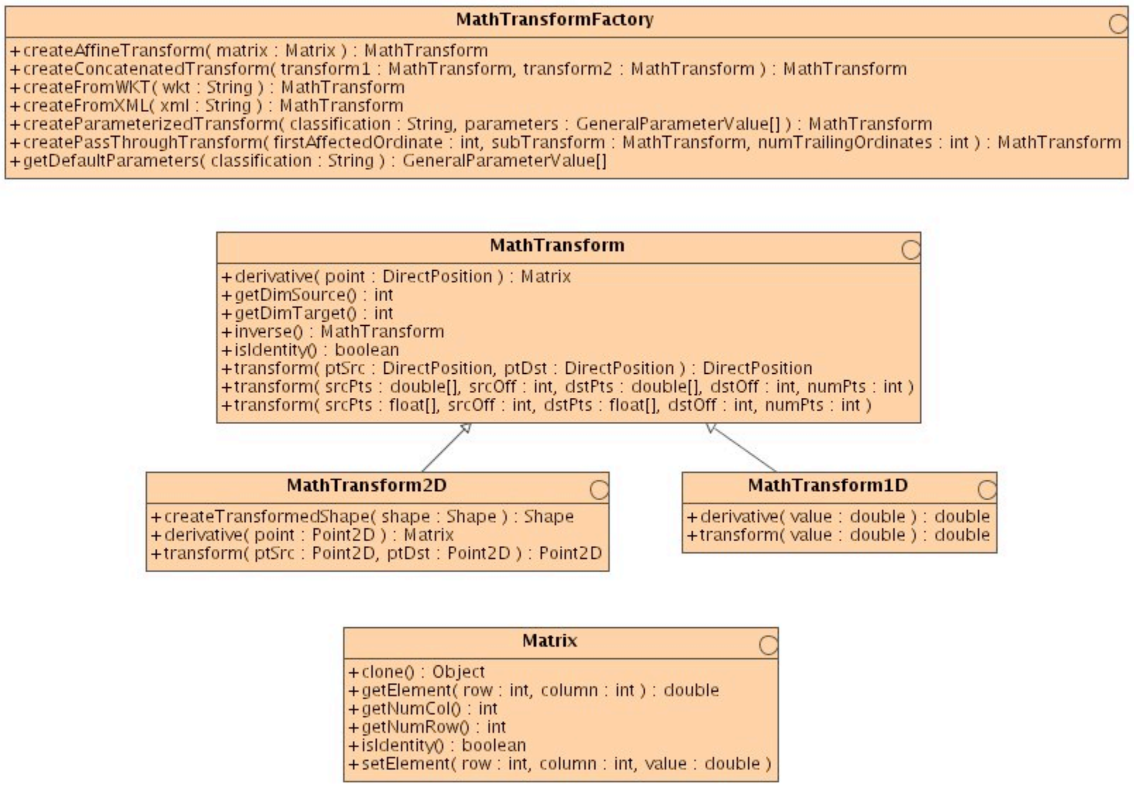

MathTransformFactory
+createAffineTransform(matrix : Matrix) : MathTransform
+createConcatenatedTransform(transform1 : MathTransform, transform2 : MathTransform) : MathTransform
+createFromWKT(wkt : String) : MathTransform
+createFromXML(xml : String) : MathTransform
+createParameterizedTransform(classification : 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 26 - MathTransform

6.4.2.7 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.

6.4.3 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.

6.5 Features

6.5.1 Model and Rationale

The feature model in GeoAPI is based on the OGC abstract specification for features (Topic 5), but draws much influence from the practical lessons learned by various open-source efforts, most notably GeoTools and Deegree.

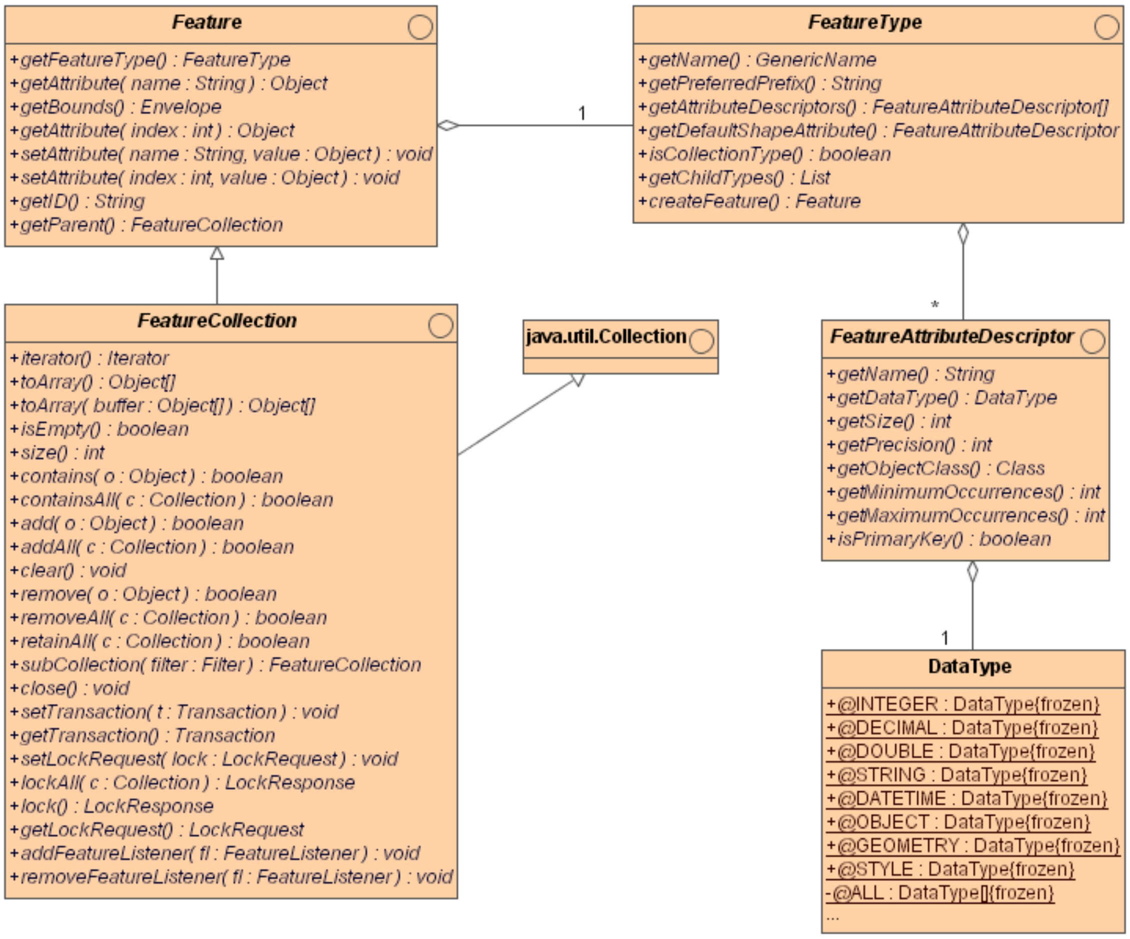

Feature
+getFeatureType() : FeatureType
+getAttribute(name : String) : Object
+getBounds() : Envelope
+getAttribute(index : int) : Object
+setAttribute(name : String, value : Object) : void
+setAttribute(index : int, value : Object) : void
+getID() : String
+getParent() : FeatureCollection
1
FeatureType
+getName() : GenericName
+getPreferredPrefix() : String
+getAttributeDescriptors() : FeatureAttributeDescriptor[]
+getDefaultShapeAttribute() : FeatureAttributeDescriptor
+isCollectionType() : boolean
+getChildTypes() : List
+createFeature() : Feature
*
FeatureCollection
+iterator() : Iterator
+toArray() : Object[]
+toArray(buffer : Object[]) : Object[]
+isEmpty() : boolean
+size() : int
+contains(o : Object) : boolean
+containsAll(c : Collection) : boolean
+add(o : Object) : boolean
+addAll(c : Collection) : boolean
+clear() : void
+remove(o : Object) : boolean
+removeAll(c : Collection) : boolean
+retainAll(c : Collection) : boolean
+subCollection(filter : Filter) : FeatureCollection
+close() : void
+setTransaction(t : Transaction) : void
+getTransaction() : Transaction
+setLockRequest(lock : LockRequest) : void
+lockAll(c : Collection) : LockResponse
+lock() : LockResponse
+getLockRequest() : LockRequest
+addFeatureListener(fl : FeatureListener) : void
+removeFeatureListener(fl : FeatureListener) : void
java.util.Collection
FeatureAttributeDescriptor
+getName() : String
+getDataType() : DataType
+getSize() : int
+getPrecision() : int
+getObjectClass() : Class
+getMinimumOccurrences() : int
+getMaximumOccurrences() : int
+isPrimaryKey() : boolean
1
DataType
+@INTEGER : DataType{frozen}
+@DECIMAL : DataType{frozen}
+@DOUBLE : DataType{frozen}
+@STRING : DataType{frozen}
+@DATETIME : DataType{frozen}
+@OBJECT : DataType{frozen}
+@GEOMETRY : DataType{frozen}
+@STYLE : DataType{frozen}
-@ALL : DataType[]{frozen}
...

Figure 27 - Feature, FeatureCollection, and FeatureType

Note that a `FeatureCollection` is itself a `Feature`, as in the OGC abstract specification, which allows a `FeatureCollection` to have a `FeatureType` and attributes of its own. But beyond the `Feature` methods, there are a large number of the methods in the `FeatureCollection` interface that come from extending the Java `Collection` interface. Extending the Java interface allows for easy interoperability with other Java collections.

In order to unambiguously define feature events and transaction semantics, instances of `Feature` must belong to at most one `FeatureCollection`. (The containing feature collection is returned from the `getFeatureColleciton()` method on `Feature`.) Different instances of the `Feature` interface may represent the same conceptual feature, but must belong to different `FeatureCollection`s. Nothing in this specification prevents the implementor from sharing attribute state between two such features. Indeed two feature objects may compare as equal using the `equals()` method and not have the same parent `FeatureCollection`.

6.5.2 Feature Attributes and Geometry

As shown in the diagrams above, a Feature has any number of attributes whose values can be retrieved by invoking one of the two `getAttribute(...)` methods. The attributes can be retrieved by integer index or by the name of the attribute. The return value of these methods is an `Object`, which means that primitive values (`int`, `double`, etc) are wrapped in their corresponding wrapper class (`Integer`, `Double`, etc).

This model for attributes differs slightly from a literal interpretation of OGC Topic 5. In Topic 5, features have attribute objects, and the attribute objects have values. Here, we allow the user to get the attribute value directly from the feature.

Features have methods for retrieving two special attributes, bounds and ID. The `getID` method must return a non-null `String` whose value uniquely identifies the feature within the scope of the current Java virtual machine. If possible, the ID should be universally unique, but this is not a requirement. The bounds of the feature give the extent of the geometry of this feature (or the extent of the union of the geometries of child features, if the feature is itself a feature collection).

Feature geometry is treated just like any other attribute. It has a name and an index and is retrieved using the `getAttribute` method of `Feature`. A feature may have several attributes which are geometries, but one of those geometries is used to render the feature. In such cases, the "default" geometry attribute should be indicated by the return value of `getDefaultShapeAttribute` of the feature's `FeatureType`. The value of geometry attributes should be an instance of one of the subclasses of `org.opengis.spatialschema.geometry.primitive.Primitive`. (However, some producers and consumers of features may wish to exchange geometry in the form of other Java objects, such as those from the Java Topology Suite, so this specification does not absolutely require that geometric attributes be subclasses of `Primitive`.)

6.5.3 FeatureTypes

Every feature has a corresponding `FeatureType` that describes it. In particular, the `FeatureType` lists the following:

- The attributes of the `Feature`, including their name and type
- The name of the `FeatureType`, encoded as an instance of `GenericName`
- A preferred namespace prefix to use when encoding the feature as GML
- An indicator as to whether features of this type are also `FeatureCollection`s
- If the features are `FeatureCollection`s, a list of types of features that could potentially be children of the feature.

The information about the attributes of features are stored in instances of the `FeatureAttributeDescriptor` interface. This interface gives the name, the data type (as an instnace of the `DataType` enumeration class), and other metadata necessary to fully specify the type, such as attribute size.

Every `FeatureType` has a name. These names are provided as instances of the `GenericName` interface. `GenericName` is a base class for two "concrete" classes:

- `LocalName` - This is a simple name that has one part that is just a string. This string has no inherent scope. A typical use for a `LocalName` is for name of an XML element.
- `ScopedName` - This is a name that has two parts, a scope, and a `LocalName`. The scope is itself a `GenericName` and thus may either be a `LocalName` or a `ScopedName`. In typical cases, the scope of a `ScopedName` will a `LocalName` representing the namespace URI of an XML element and the `LocalName` of the `ScopedName` will be the name of the XML element.

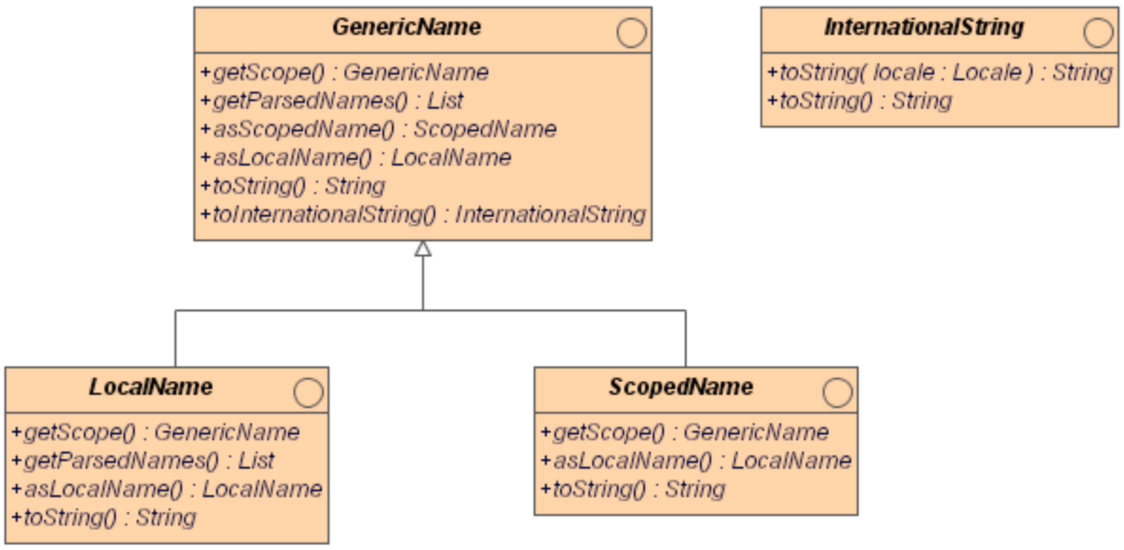

GenericName
+getScope() : GenericName
+getParsedNames() : List
+asScopedName() : ScopedName
+asLocalName() : LocalName
+toString() : String
+toInternationalString() : InternationalString
InternationalString
+toString(locale : Locale) : String
+toString() : String
LocalName
+getScope() : GenericName
+getParsedNames() : List
+asLocalName() : LocalName
+toString() : String
ScopedName
+getScope() : GenericName
+asLocalName() : LocalName
+toString() : String

Figure 28 - GenericName

6.5.4 Modifying Features

The features in this specification can be modified using the `setAttribute(...)` method. Feature instances may be backed by data in some sort of persistent backing store, so calling `setAttribute(...)` should cause changes to be written to the persistent store. Feature collections can also be modified using the methods inherited from Java's `Collection` interface (such as `add(...)`, `remove(...)`, etc). Such changes should also be written to the persistent store.

6.5.5 FeatureCollections

A `FeatureCollection` represents any grouping of `Features`. This may be the return value of a query on a server, or it may be a collection created by the API user. In either case, the members of a `FeatureCollection` can be enumerated by invoking the `iterator()` method. This returns a Java iterator over the members of the `Collection`.

`FeatureCollection`s may often be backed by a persistent store of some kind. In such cases, the methods that modify the collection, such as `add(...)` and `remove(...)`, are required to make modifications to the persistent store. Such modifications take place within the context of the current transaction the `FeatureCollection`. See the section below for more information about transactions.

Also, because `FeatureCollection`s are often backed by a persistent store, network connections may have to be established to provide access to the members. Because of this, there is a `close()` method on `FeatureCollection` that can be invoked to indicate to the implementation that the caller no longer has need of the data contained within the `FeatureCollection`, and any resources, such as network connections, can be cleaned up.

Note that the members of a `FeatureCollection` may themselves be instances of the `FeatureCollection` interface. This gives rise to the possibility that there would be hierarchies of `Feature`s contained within a `FeatureStore`.

The `FeatureCollection` interface extends the Java `Collection` interface, which is the base class for both unordered `Set`s and ordered `List`s. Implementors of the `FeatureCollection` interface are encouraged to implement a more specific subclass (such as `Set` or `List`) if their collection does have those semantics. This might allow, for example, random access of the collection elements if the user knows that the collection is a `List`.

6.5.6 Feature Events

The `FeatureCollection`s in this specification provide a mechanism that allows the API user to register for notification of certain events, namely the addition, removal, or changing of features within that collection.

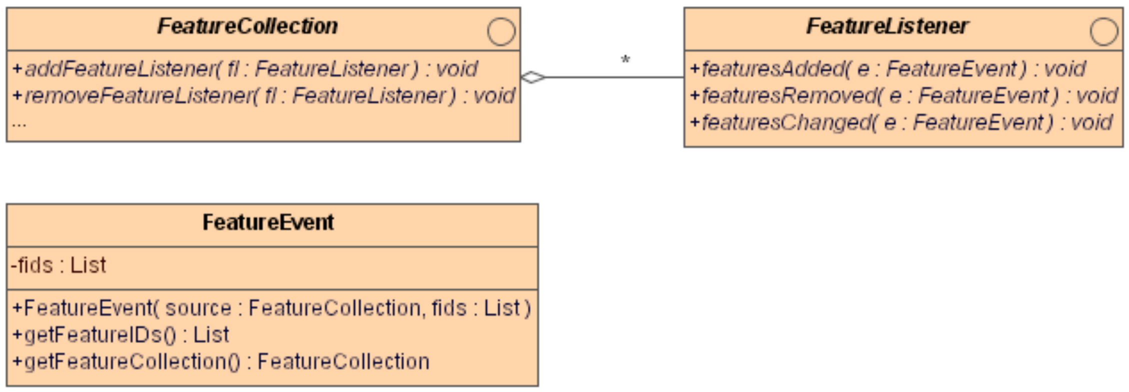

FeatureCollection
+addFeatureListener(fl : FeatureListener) : void
+removeFeatureListener(fl : FeatureListener) : void
...
*
FeatureListener
+featuresAdded(e : FeatureEvent) : void
+featuresRemoved(e : FeatureEvent) : void
+featuresChanged(e : FeatureEvent) : void
FeatureEvent
-fids : List
+FeatureEvent(source : FeatureCollection, fids : List)
+getFeatureIDs() : List
+getFeatureCollection() : FeatureCollection

Figure 29 - Feature Listeners

`FeatureCollections` that support event listeners should invoke the appropriate listener methods when the features they contain are modified, added, or removed. Note that it is possible that two distinct `FeatureCollections` could contain a `Feature` that represents the same conceptual feature. If this feature were modified in one collection, an intelligent implementation should notify listeners on *both* `FeatureCollections` that the feature has changed.

6.5.7 Transactions

An instance of `Transaction` represents a set of operations performed on various `FeatureStores`, where the entire set of operations can be rolled back or committed at the same time. The constant `Transaction.AUTO_COMMIT` is used to request immediate change, or during event notification to indicate the persistent state has changed. When a `FeatureCollection` is using a `Transaction` its contents may still be modified, but the changes are simply not persisted until the `Transaction`'s `commit()` method is called.

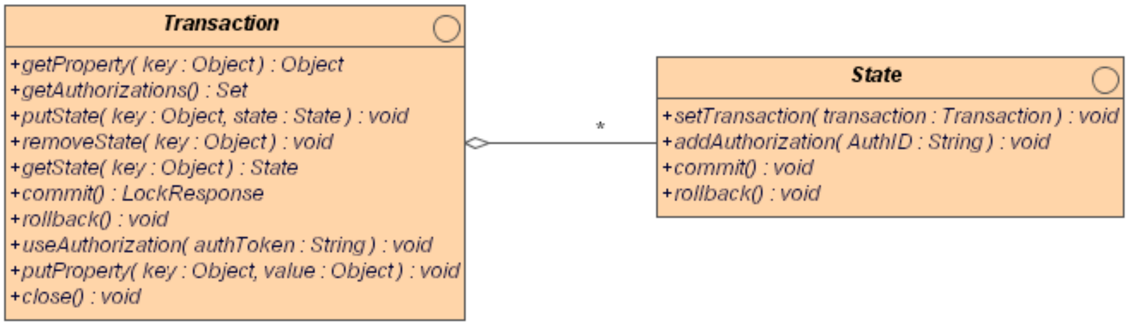

Transaction
+getProperty(key : Object) : Object
+getAuthorizations() : Set
+putState(key : Object, state : State) : void
+removeState(key : Object) : void
+getState(key : Object) : State
+commit() : LockResponse
+rollback() : void
+useAuthorization(authToken : String) : void
+putProperty(key : Object, value : Object) : void
+close() : void
*
State
+setTransaction(transaction : Transaction) : void
+addAuthorization(AuthID : String) : void
+commit() : void
+rollback() : void

Figure 30 - Transaction and Transaction.State

The following pseudo-code demonstrates how transactions might be used by a client.

```
FeatureStore featureStore = ...;
FeatureCollection features = featureStore.getFeatures(...);
Feature feature1 = new DefaultFeature(...);

// Set to auto-commit
features.setTransaction( Transaction.AUTO_COMMIT );
// Add feature1 to the collection.  It is immediately persisted.
features.add( feature1 );

// Set to a new transaction
Transaction t = new DefaultTransaction();
features.setTransacstion( t );

features.remove( feature1 );
// feature1 is removed, but the change has not be persisted

features.rollback();
// the remove of feature1 has been canceled, features is
// restored to its previous state
```

Event notification for a `FeatureCollection` is largely unaffected by transactions. However, two additional notifications are needed:

- `Transaction.commit()` causes the firing of an event to indicate persistent state change for other `FeatureCollections` that may have common members.

- `Transaction.rollback()` fires an event for listeners on the current `FeatureCollection` indicating the state of the feature collection has changed. Other `FeatureCollections` that may have common members do not receive events if a transaction is rolled back.

Another pseudo-code example demonstrates how events are fired for collections containing common members:

```
FeatureStore featureStore = ...;
FeatureCollection featuresA = featureStore.getFeatures(...);
FeatureCollection featuresB = featuresA.clone();
// we now have two FeatureCollections w/ the same contents

// Create listeners and add them to the collections
FeatureListener listenerA = new MyFeatureListener(...);
FeatureListener listenerB = new MyFeatureListener(...);
featuresA.addListener( listenerA );
featuresB.addListener( listenerB );

featuresA.setTransaction( Transaction.AUTO_COMMIT );
Transaction transaction = new DefaultTransaction();
featuresB.setTransaction( transaction );

featuresA.remove( feature1 );
// feature1 is removed from both featuresA  and featuresB
// listenerA and listenerB recieve event

featuresB.add( feature1 );
// listenerB recieves notification
featuresB.add( feature2 );
// listenerB recieves notification
featuresB.commit();
// listenerA recieves notification
```

6.5.8 Transaction from the implementors perspective

The writers of this specification expect that the `Transaction` interface will be implemented one time in a utility library since all of the behavior of the `Transaction` class can be performed independent of backing store implementation. The "interesting" code for implementors is in the implementation of the `Transaction.State` interface and the implementation of `FeatureCollection.setTransaction(...)`. The following Figure illustrates the sequence of events that occurs when a transaction is set on a feature collection. Steps 3, 4, 7, and 11 are the steps that have logic specific to a given data source. (See the class Javadocs for more detailed information about the contract of these interfaces.)

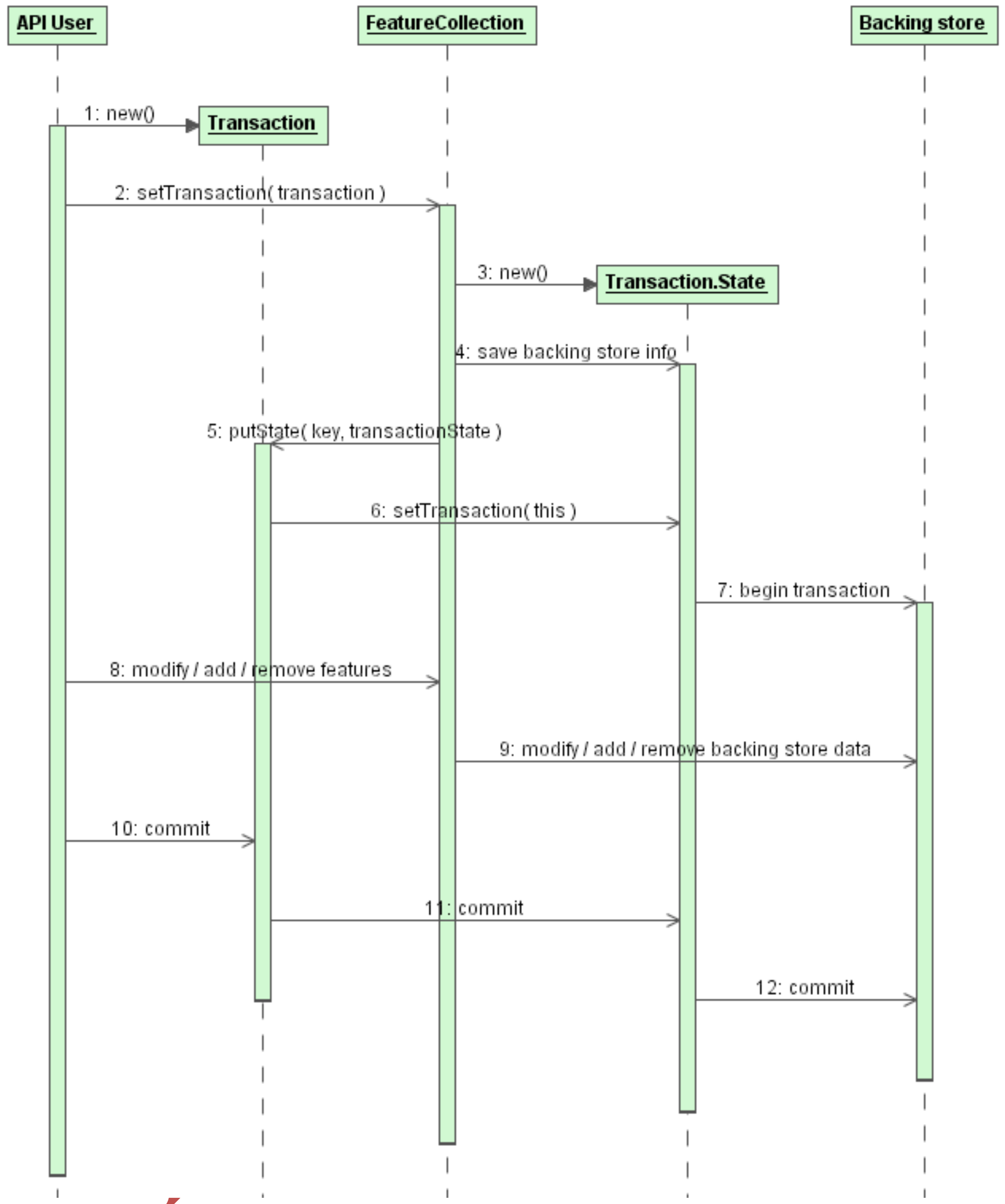

API User
FeatureCollection
Backing store
1: new()
Transaction
2: setTransaction(transaction)
3: new()
Transaction.State
4: save backing store info
5: putState(key, transactionState)
6: setTransaction(this)
7: begin transaction
8: modify / add / remove features
9: modify / add / remove backing store data
10: commit
11: commit
12: commit

Figure 31 - Transaction sequence diagram

6.5.9 Transaction field and method detail

- `AUTO_COMMIT` - Constant used to request that any modifications to a feature or a feature collection should occur immediately. This member follows the "Special Case" design pattern.
- `close()` - Disposes of any resources held by the transaction. If changes are pending on the current transaction, they are rolled back.

- `commit()` - Makes all changes since the previous `commit/rollback` permanent. This method returns a `LockResult` indicating the success of any lock operations made of the course of the transaction.
- `rollback()` - Undoes all changes performed since the last commit or rollback.
- `getAuthorizations(),useAuthorization( token )` - Manage and access a `Set` of authorization tokens allowing access to locked content. See the section on locking below.
- `getProperty(key),setProperty(key,value)` - Allows for user specified hints. This might allow access to features of a `FeatureStore` not provided in the GeoAPI interfaces.
- `getState(key),removeState(key),putState(key,value)` - Allows `FeatureStores` to externalize state under transaction control. Follows the momento design pattern. When the user invokes the setTransaction method on a `FeatureCollection`, the implementation is expected to call these methods to attach or retrieve state information about the transaction. Then when the transaction is committed, rolled back, or closed, the state objects are enumerated and the corresponding `commit` or `rollback` method is called.

6.5.10 Locking

The writers of this specification view the FeatureStore interface as a programming language interface parallel of the WFS web interface. Therefore our design for locking is motivated by the locking functionality given by a WFS. Both transaction-duration locks and WFS-style long-term locks are supported by this API.

This specification provides an attractive middle ground between full versioned Features, and light-weight in-process or file lock based approaches. Locks are maintained for a requested duration. A successful lock operation results in an authorization token that may be used at a later time. A transaction may be assigned such a token allowing it to work on previously locked features. Locks are allowed to be taken out across multiple `FeatureCollections` (each returning a unique authorization token).

In general, the sequence of events for long-term locking is the following:

- The user retrieves a `FeatureCollection`.
- The user sets a `LockRequest` on the `FeatureCollection`.
- The user invokes `lock()` on the `FeatureCollection` to lock all of its features.
- If using a `Transaction`, the user commits the transaction to get the result of locking. (If not using a `Transaction`, the `lock()` method returns the result.) Now the features are locked in the database. The result contains an authorization token that is stored for later use.
- At a later time, the user creates a new `Transaction` and adds the authorization token to it.

- The user again retrieves a `FeatureCollection` with some or all of the locked features in it and sets its transaction to the one with the authorization token.
- The user makes changes to the collection and commits the `Transaction`.
- The lock is released.

Locking workflow is described by capabilities of the FeatureCollection.lock() methods, LockRequest, LockResponse, and Transaction.

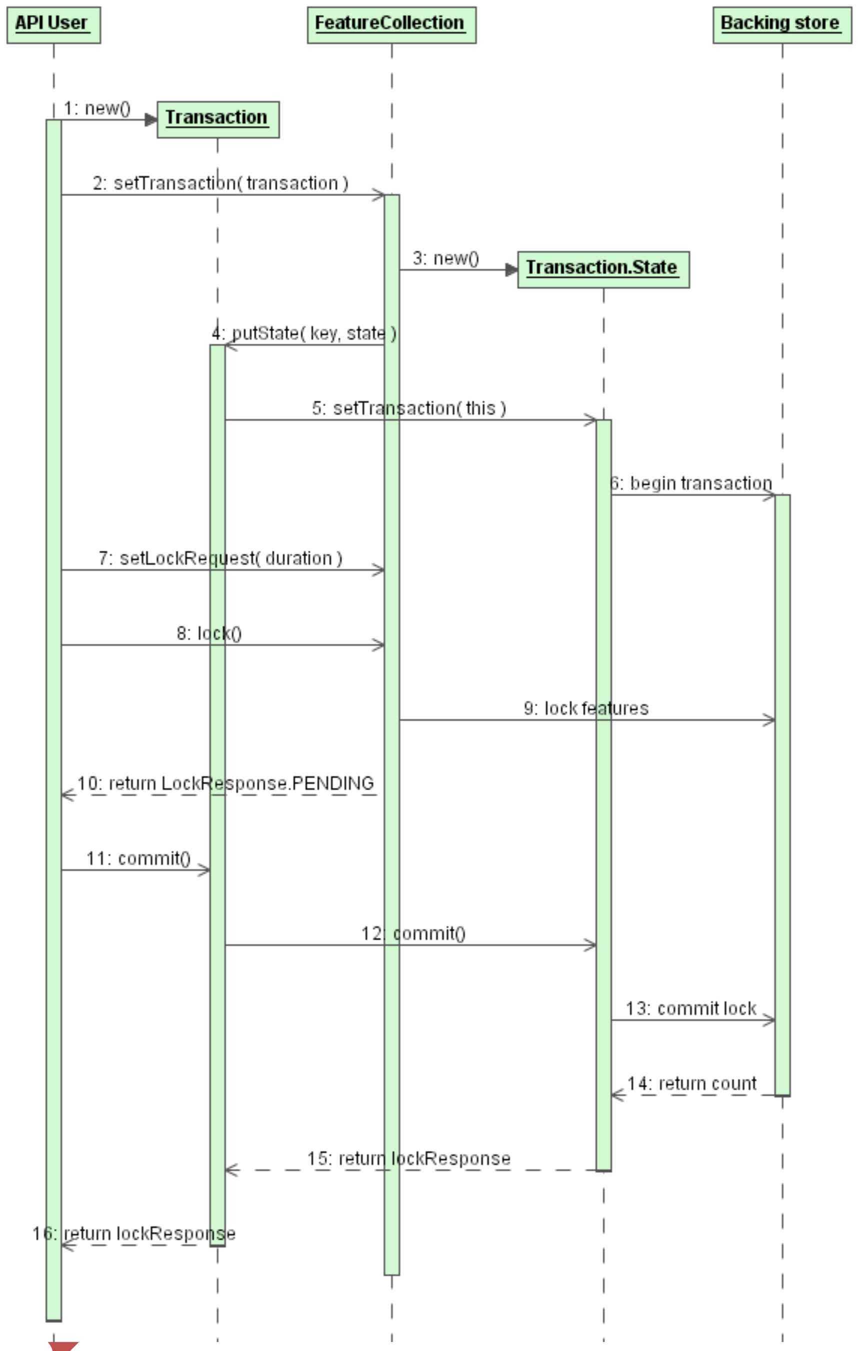


Figure32 - Initiating a long term lock while using a Transaction

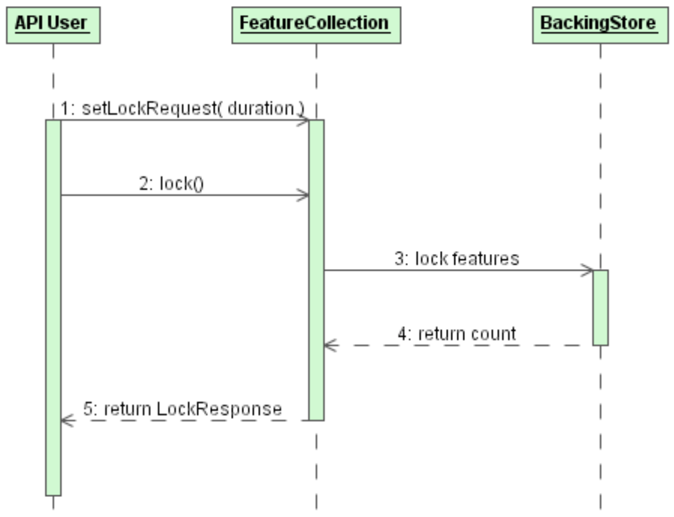


Figure 33 - Initiating a long term lock without a Transaction

The `LockRequest` class is a wrapper for a lock duration expressed in milliseconds. A static constant on the class, `LockRequest.TRANSACTION_LOCK`, can be used to request a lock lasting until the next commit or rollback. (This follows the Special Case design pattern.) Such locks are often used to ensure access before starting modification.

The `LockResponse` class indicates number of locked features and authorization tokens for each involved `FeatureStore`. The static constant `LockResponse.PENDING` indicates that the lock request is waiting until the next commit to provide authorization tokens and feature counts. The static constant `FeatureResponse.NONE` is used as a return value by `Transaction.commit()` to indicates no lock requests were made. The static constant `LockResponse.TRANSACTION_LOCK_RESPONSE` indicates that aquired locks will expire at the end of the transaction.

The following pseudo-code example demonstrates how locks could be used:

```
FeatureStore featureStore = ...;
GenericName roads = featureStore.getTypeNames().iterator().next();

// The "bnf" method is unfortunately fictitious
FeatureCollection kaslo = featureStore.getFeatures( roads,
    FilterFactory.bnf( "CITY='kaslo'" );
LockRequest lockRequest = new LockRequest( 45678 );
Transaction t = new DefaultTransaction();
kaslo.setTransaction( t );

// Lock the features
kaslo.setLockRequest( lockRequest );
kaslo.lock(); // returns LockResponse.PENDING
LockResponse lockResults = t.commit();
// kaslo roads are now locked

String token = lockResults.getToken();

// Now try some operations
kaslo.clear();
try {
    // commit will fail since the transaction does not have
    // the authorization token
    t.commit();
} catch (IOException locked){
     // datais safe - we did not use authorization
     System.out.println("expected locking message:"+locked);
}

t.useAuthorization( token );
kaslo.clear();
lockResults = t.commit();
// The commit succeeds and lockResults is LockResults.NONE
// since no new locks were performed
```

At the end of the above example, the roads in Kaslo are removed and the lock is released since the token has been used.

A couple of important points that prevent deadlock:

- Lock request does not produce an exception, success or failure can be determined by the LockResponse.
- Trying to perform a modification, or trying to lock a previously locked Feature will produce an exception.

6.6 FeatureStore

6.6.1 Model and Rationale

While the `Feature` and `FeatureCollection` APIs provide mechanisms for accessing feature data, the model as described thus far does not provide a mechanism for retrieving `FeatureCollection`s from a persistent store of some kind. Thus we introduce `FeatureStore`.

The FeatureStore interface provides an abstraction for an object that connects to a store of features and provides them to the user. For example, an application may implement the FeatureStore interface to connect to a relational database and provide the contents of the tables as features. (The user of the FeatureStore should not know this implementation detail, and it should only use methods in the FeatureStore interface.)

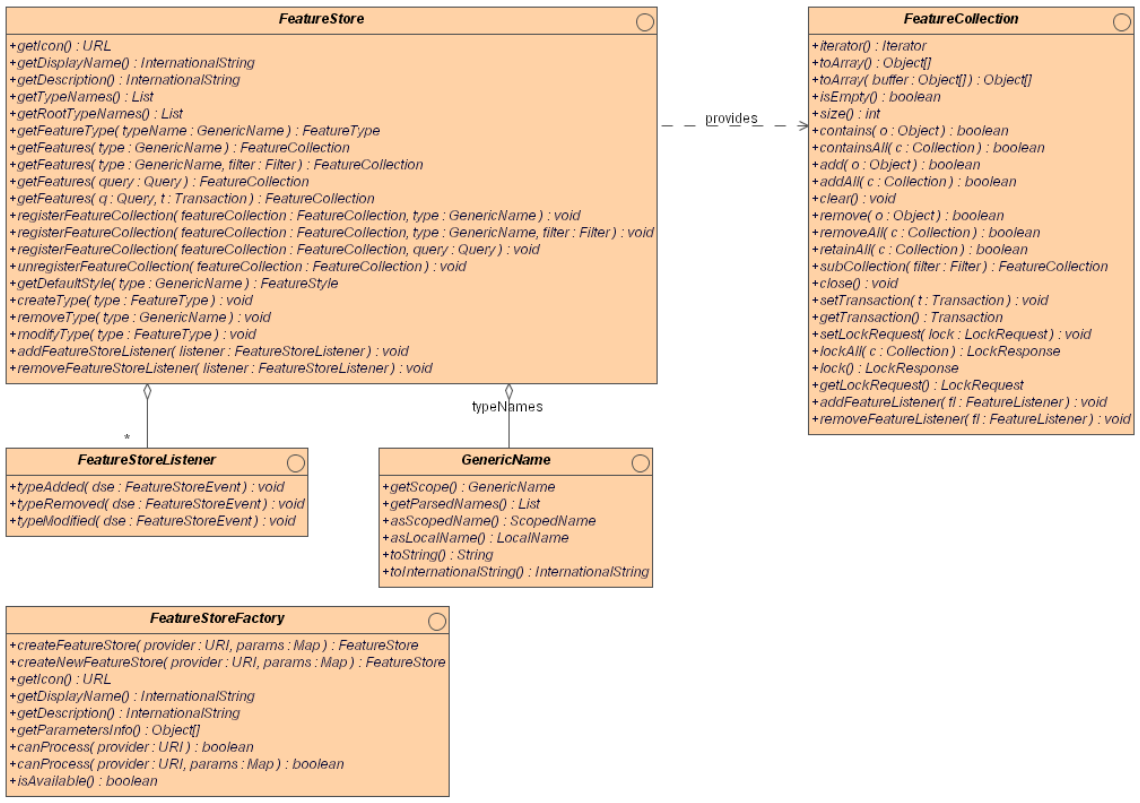

FeatureStore
+getIcon() : URL
+getDisplayName() : InternationalString
+getDescription() : InternationalString
+getTypeNames() : List
+getRootTypeNames() : List
+getFeatureType(typeName : GenericName) : FeatureType
+getFeatures(type : GenericName) : FeatureCollection
+getFeatures(type : GenericName, filter : Filter) : FeatureCollection
+getFeatures(query : Query) : FeatureCollection
+getFeatures(q : Query, t : Transaction) : FeatureCollection
+registerFeatureCollection(featureCollection : FeatureCollection, type : GenericName) : void
+registerFeatureCollection(featureCollection : FeatureCollection, type : GenericName, filter : Filter) : void
+registerFeatureCollection(featureCollection : FeatureCollection, query : Query) : void
+unregisterFeatureCollection(featureCollection : FeatureCollection) : void
+getDefaultStyle(type : GenericName) : FeatureStyle
+createType(type : FeatureType) : void
+removeType(type : GenericName) : void
+modifyType(type : FeatureType) : void
+addFeatureStoreListener(listener : FeatureStoreListener) : void
+removeFeatureStoreListener(listener : FeatureStoreListener) : void
provides
FeatureCollection
+iterator() : Iterator
+toArray() : Object[]
+toArray(buffer : Object[]) : Object[]
+isEmpty() : boolean
+size() : int
+contains(o : Object) : boolean
+containsAll(c : Collection) : boolean
+add(o : Object) : boolean
+addAll(c : Collection) : boolean
+clear() : void
+remove(o : Object) : boolean
+removeAll(c : Collection) : boolean
+retainAll(c : Collection) : boolean
+subCollection(filter : Filter) : FeatureCollection
+close() : void
+setTransaction(t : Transaction) : void
+getTransaction() : Transaction
+setLockRequest(lock : LockRequest) : void
+lockAll(c : Collection) : LockResponse
+lock() : LockResponse
+getLockRequest() : LockRequest
+addFeatureListener(fl : FeatureListener) : void
+removeFeatureListener(fl : FeatureListener) : void
typeNames
*
FeatureStoreListener
+typeAdded(dse : FeatureStoreEvent) : void
+typeRemoved(dse : FeatureStoreEvent) : void
+typeModified(dse : FeatureStoreEvent) : void
GenericName
+getScope() : GenericName
+getParsedNames() : List
+asScopedName() : ScopedName
+asLocalName() : LocalName
+toString() : String
+toInternationalString() : InternationalString
FeatureStoreFactory
+createFeatureStore(provider : URI, params : Map) : FeatureStore
+createNewFeatureStore(provider : URI, params : Map) : FeatureStore
+getIcon() : URL
+getDisplayName() : InternationalString
+getDescription() : InternationalString
+getParametersInfo() : Object[]
+canProcess(provider : URI) : boolean
+canProcess(provider : URI, params : Map) : boolean
+isAvailable() : boolean

Figure 34 - FeatureStore

Note that a `FeatureStore` implementation need not make any network or database connections to supply features. It may just as easily read a local file and parse all of its contents into memory. In such a case, some methods (such as those that are documented to close any open connections) may do nothing.

6.6.2 Nested Features

As discussed in other sections, `Features` can be `FeatureCollections` and thus contain child `Feature` objects. In such cases, there could be a large containment hierarchy of `Features`. A `FeatureStore` that provides such hierarchies of `Features` can give hints to the user about which types of `Features` are the "root" `Features` in such hierarchies by implementing the `getRootTypeNames()` method, described below.

6.6.3 FeatureStore method detail

- `getTypeNames()` - Returns a `List` of `GenericName` objects that name the types of features provided by this `FeatureStore`. Although the `List` interface returns `Objects` from its accessors, the `List` returned from this method is guaranteed only to contain instances of `GenericName`.
- `getRootTypeNames()` - This method returns a list of names of the types that are the "root" features in nested feature hierarchies. For many `FeatureStores`, this

method may very well return the same list as returned by `getTypeNames()`. But for all `FeatureStore`s, this list will contain a subset of the names returned by `getTypeNames()`.

- `getFeatureType(GenericName)` - This method returns a `FeatureType` instance that describes the details of the type whose name is given. The `FeatureType` includes information about the attributes of the features, as well as an indication of whether the features of this type are collections.
- `getFeatures(...)` - Each of these methods returns a collection of features from the `FeatureStore` that passes an optional `Filter`. Implementors of these methods are free to retrieve all of the features immediately or to defer the retrieval until the `FeatureCollection`'s `iterator()` method is invoked.
- `getDefaultStyle(GenericName)` - Some data providers may also wish to provide the user of the features with a default set of rules for styling their features. This method allows them to do so. It is valid to return `null` from this method if the data provider doesn't know a default style or doesn't care about styling.
- `createType`, `removeType`, `modifyType` - The implementor of a `FeatureStore` may wish to allow the user to create, remove, and modify the types of features that are available. For example, `createType` for a particular `FeatureStore` may create a new database table. All three of these methods may throw an exception if the `FeatureStore` does not support this type of operation.
- `addFeatureStoreListener`, `removeFeatureStoreListener` - For `FeatureStore`s that do support any of the three methods for changing the feature types in the store, they must also implement these methods that allow the user to receive notification when such a change has been made.

6.6.4 Filter

The various Filter classes were translated to Java from the XML schemas for Filter [16]:

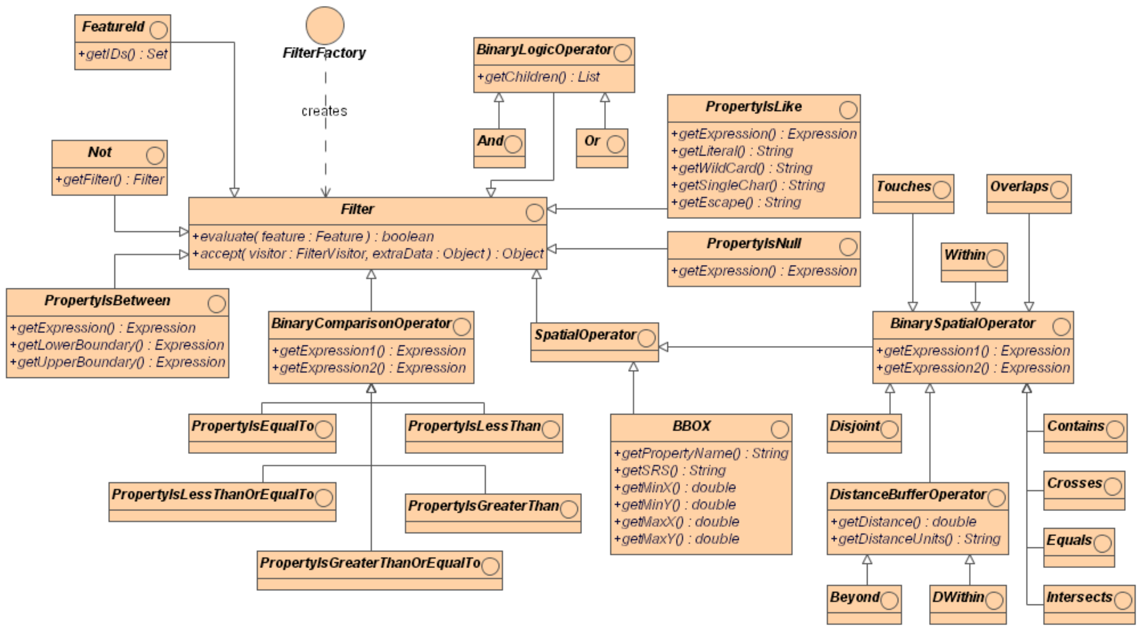


Figure 35 - Filter and subclasses

The meanings of the various members of these classes are identical to the meanings of their corresponding elements in the Filter encoding schema.

The only changes from the OGC Filter specification are the spatial operators. In the Filter specification, the spatial operators are only allowed to compare a Feature's geometry property with a constant geometry. Here both parameters are allowed to be any expression that evaluates to a geometry.

Filter objects cannot be changed after they are created, so they must be created with all of their parameters set. The FilterFactory interface provides methods for creating each type of Filter and these methods accept parameters for all of the pertinent properties.

Included in the filter package is the expression subpackage that provides classes that can evaluate a limited set of mathematical expressions based on the properties of features. These classes are also based on the corresponding classes from the OGC Filter specification and have the same semantics:

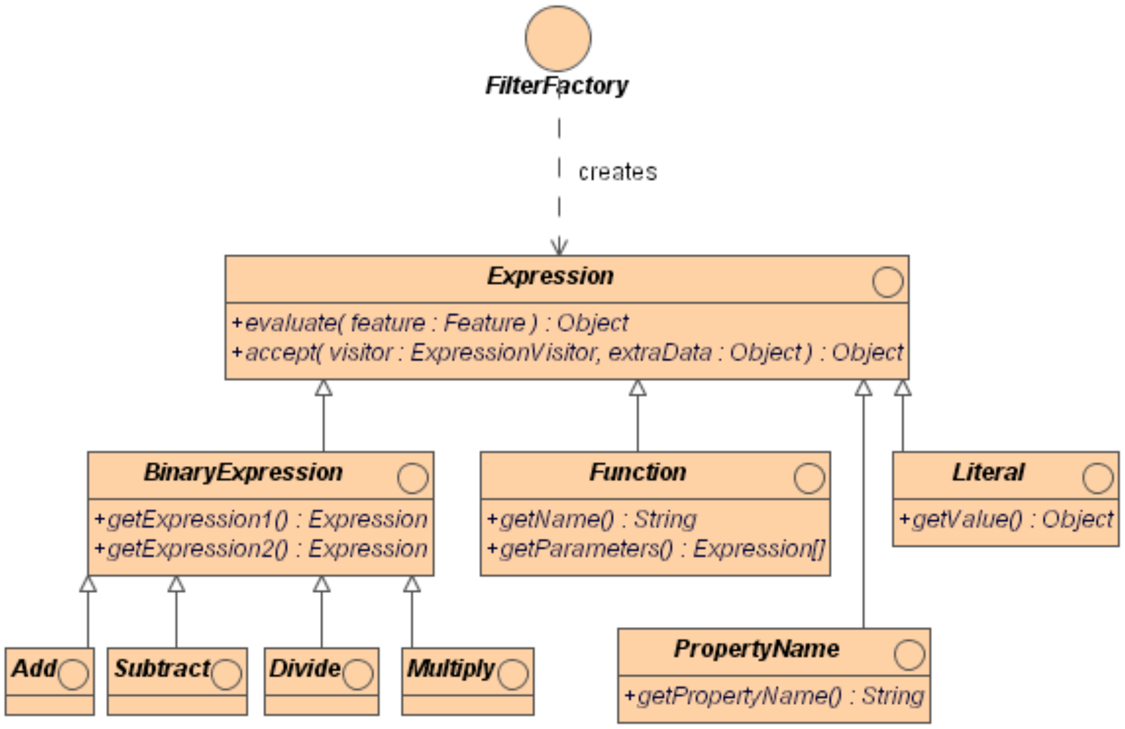

FilterFactory
creates
Expression
+evaluate(feature : Feature) : Object
+accept(visitor : ExpressionVisitor, extraData : Object) : Object
BinaryExpression
+getExpression1() : Expression
+getExpression2() : Expression
Function
+getName() : String
+getParameters() : Expression[]
Literal
+getValue() : Object
PropertyName
+getPropertyName() : String
Add
Subtract
Divide
Multiply

Figure 36 - Expression

6.6.5 FeatureStore Example Usage

The following code demonstrates how one can retrieve some features from a FeatureStore. This simply prints out the feature ID of each feature of the first type offered by the FeatureStore.

```
FeatureStore featureStore = ... ;
List featureTypes = featureStore.getTypeNames();
GenericName typeName = (GenericName) featureTypes.get(0);
System.err.println("Retrieving features of type " + typeName);
FeatureCollection features = featureStore.getFeatures(typeName);
Iterator iterator = features.iterator();
while (iterator.hasNext()) {
    Feature f = (Feature) iterator.next();
    System.err.println(f.getID());
}
System.err.println("Done.");
```

Another usage of FeatureStores is for creation of layers to be displayed in a FeatureCanvas. The following code demonstrates how one might create a Layer and add it to a canvas.

```
FeatureStore featureStore = ... ;
FeatureCanvas canvas = ... ;
List featureTypes = featureStore.getTypeNames();
GenericName typeName = (GenericName) featureTypes.get(0);
System.err.println("Displaying features of type " + typeName);
FeatureCollection features = featureStore.getFeatures(typeName);
FeatureLayer layer = new MyFeatureLayerImplementation(features,
featureStore.getDefaultStyle(typeName));
canvas.addLayer(layer);
```

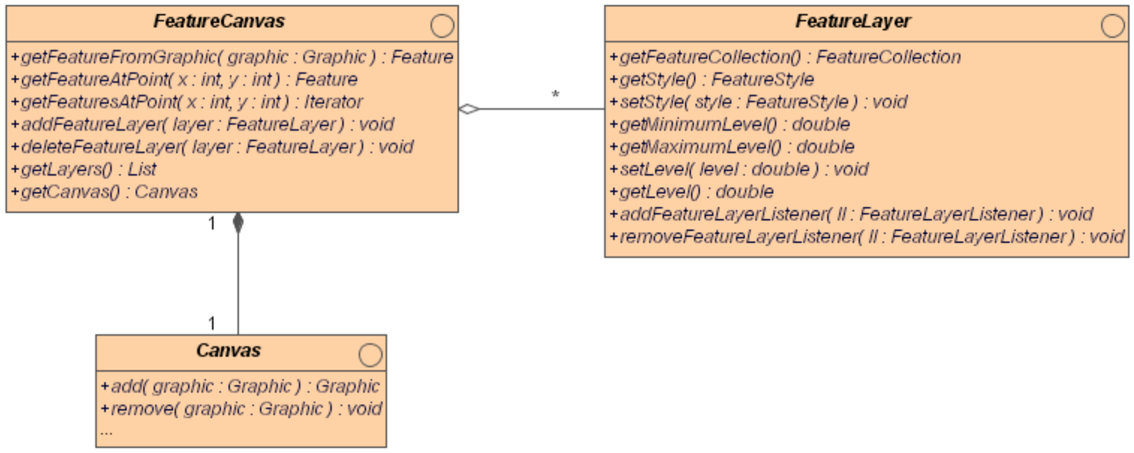
6.6.6 Relationship of FeatureStore to existing OGC standards

The FeatureStore interface does not have a direct counterpart in any of the existing abstract or implementation specifications. However, the `FeatureStore` Java interface (along with `FeatureCollection`) does very closely resemble the HTTP interface of a web feature server. Both have operations for describing a given type of feature, for retrieving features, for adding features, and for removing features.

6.7 FeatureCanvas

6.7.1 Model and Rationale

A `FeatureCanvas` represents a visual component that renders `Feature`s on a display. The `FeatureCanvas` can be viewed as a mechanism for converting `Features` plus `FeatureStyle`s into `Graphic`s on a GO-1 `Canvas`. Conceptually, the FeatureCanvas is very similar to a Web Map Server. It maintains a list of Layers, and each of these layers has a Z-order hint and a currently active style.



FeatureCanvas
+getFeatureFromGraphic(graphic : Graphic) : Feature
+getFeatureAtPoint(x : int, y : int) : Feature
+getFeaturesAtPoint(x : int, y : int) : Iterator
+addFeatureLayer(layer : FeatureLayer) : void
+deleteFeatureLayer(layer : FeatureLayer) : void
+getLayers() : List
+getCanvas() : Canvas
FeatureLayer
+getFeatureCollection() : FeatureCollection
+getStyle() : FeatureStyle
+setStyle(style : FeatureStyle) : void
+getMinimumLevel() : double
+getMaximumLevel() : double
+setLevel(level : double) : void
+getLevel() : double
+addFeatureLayerListener(ll : FeatureLayerListener) : void
+removeFeatureLayerListener(ll : FeatureLayerListener) : void
*
1
1
Canvas
+add(graphic : Graphic) : Graphic
+remove(graphic : Graphic) : void
...

Figure 37 - FeatureCanvas

6.7.2 FeatureStyle

The style portions of this specification are direct Java bindings of the Styled Layer Descriptor XML encoding specified by OGC. (The names for the various style interfaces were taken from version 1.0.20 of the SLD schemas since they are somewhat more compact than the version 1.0 names. However, the functionality of the interfaces is the same as in version 1.0.)

Style objects can be created from a `FeatureStyleFactory` instance in two ways. The first way is to parse an OGC SLD XML file from an input stream using one of the `parse(...)` methods. The second is to call the createStyleObject method, passing in a

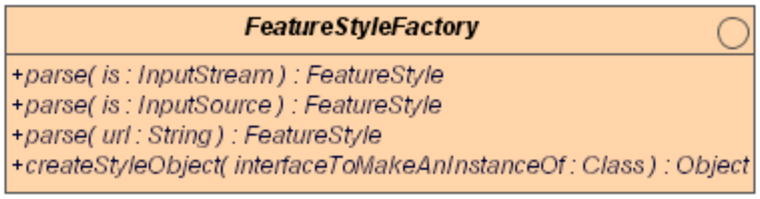

FeatureStyleFactory
+parse(is : InputStream) : FeatureStyle
+parse(is : InputSource) : FeatureStyle
+parse(url : String) : FeatureStyle
+createStyleObject(interfaceToMakeAnInstanceOf : Class) : Object

Figure38 - FeatureStyleFactory Interface

The following diagrams illustrate the top level classes used by an instance of FeatureStyle. As indicated in the picture, an instance of FeatureStyle contains any number of Rules, which in turn contain any number of Symbols. Symbol is an abstract interface; various subclasses of Symbol exist that correspond to differing types of geometry.

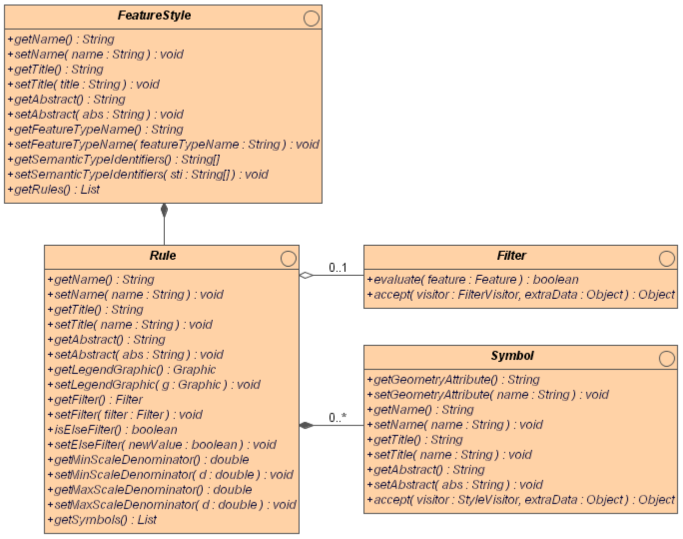

FeatureStyle
+getName() : String
+setName(name : String) : void
+getTitle() : String
+setTitle(title : String) : void
+getAbstract() : String
+setAbstract(abs : String) : void
+getFeatureTypeName() : String
+setFeatureTypeName(featureTypeName : String) : void
+getSemanticTypeIdentifiers() : String[]
+setSemanticTypeIdentifiers(sti : String[]) : void
+getRules() : List
Rule
+getName() : String
+setName(name : String) : void
+getTitle() : String
+setTitle(name : String) : void
+getAbstract() : String
+setAbstract(abs : String) : void
+getLegendGraphic() : Graphic
+setLegendGraphic(g : Graphic) : void
+getFilter() : Filter
+setFilter(filter : Filter) : void
+isElseFilter() : boolean
+setElseFilter(newValue : boolean) : void
+getMinScaleDenominator() : double
+setMinScaleDenominator(d : double) : void
+getMaxScaleDenominator() : double
+setMaxScaleDenominator(d : double) : void
+getSymbols() : List
0..1
Filter
+evaluate(feature : Feature) : boolean
+accept(visitor : FilterVisitor, extraData : Object) : Object
0..*
Symbol
+getGeometryAttribute() : String
+setGeometryAttribute(name : String) : void
+getName() : String
+setName(name : String) : void
+getTitle() : String
+setTitle(name : String) : void
+getAbstract() : String
+setAbstract(abs : String) : void
+accept(visitor : StyleVisitor, extraData : Object) : Object

Figure 39 - Style Interfaces

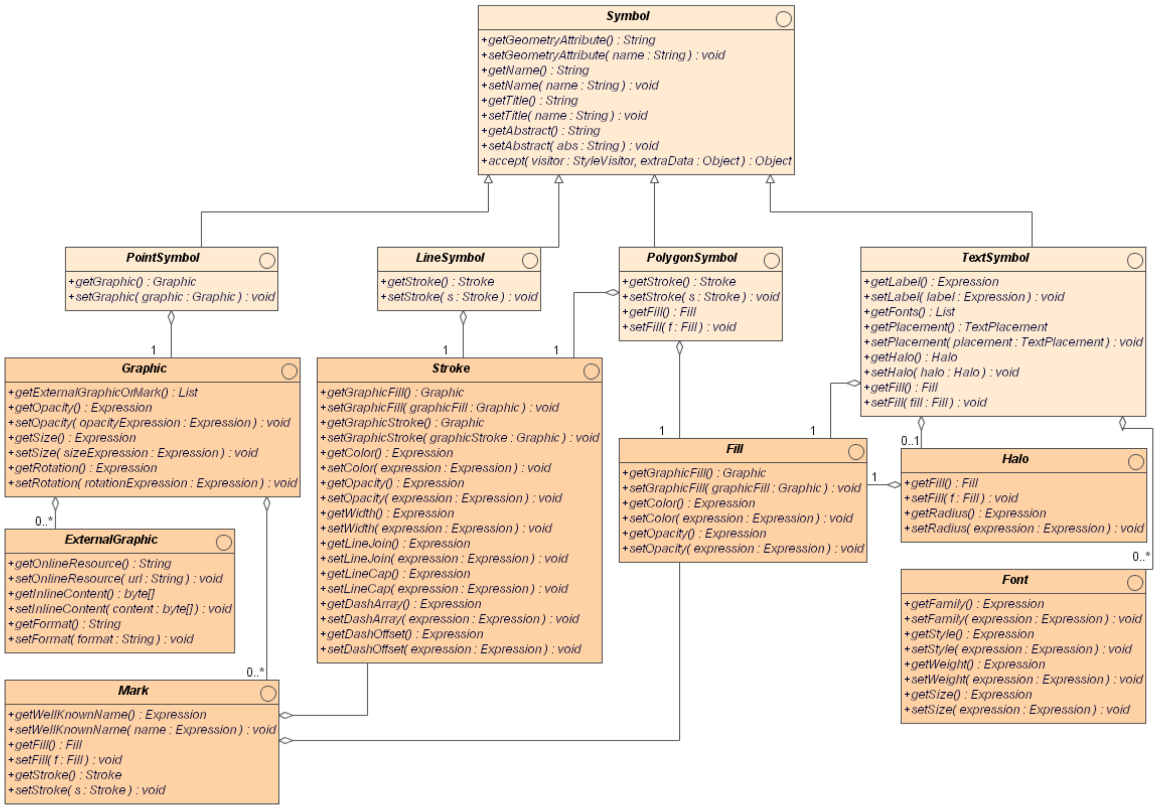

Symbol
+getGeometryAttribute() : String
+setGeometryAttribute(name : String) : void
+getName() : String
+setName(name : String) : void
+getTitle() : String
+setTitle(name : String) : void
+getAbstract() : String
+setAbstract(abs : String) : void
+accept(visitor : StyleVisitor, extraData : Object) : Object
PointSymbol
+getGraphic() : Graphic
+setGraphic(graphic : Graphic) : void
LineSymbol
+getStroke() : Stroke
+setStroke(s : Stroke) : void
PolygonSymbol
+getStroke() : Stroke
+setStroke(s : Stroke) : void
+getFill() : Fill
+setFill(f : Fill) : void
TextSymbol
+getLabel() : Expression
+setLabel(label : Expression) : void
+getFonts() : List
+getPlacement() : TextPlacement
+setPlacement(placement : TextPlacement) : void
+getHalo() : Halo
+setHalo(halo : Halo) : void
+getFill() : Fill
+setFill(fill : Fill) : void
1
1
1
Graphic
+getExternalGraphicOrMark() : List
+getOpacity() : Expression
+setOpacity(opacityExpression : Expression) : void
+getSize() : Expression
+setSize(sizeExpression : Expression) : void
+getRotation() : Expression
+setRotation(rotationExpression : Expression) : void
Stroke
+getGraphicFill() : Graphic
+setGraphicFill(graphicFill : Graphic) : void
+getGraphicStroke() : Graphic
+setGraphicStroke(graphicStroke : Graphic) : void
+getColor() : Expression
+setColor(expression : Expression) : void
+getOpacity() : Expression
+setOpacity(expression : Expression) : void
+getWidth() : Expression
+setWidth(expression : Expression) : void
+getLineJoin() : Expression
+setLineJoin(expression : Expression) : void
+getLineCap() : Expression
+setLineCap(expression : Expression) : void
+getDashArray() : Expression
+setDashArray(expression : Expression) : void
+getDashOffset() : Expression
+setDashOffset(expression : Expression) : void
1
1
0..1
Fill
+getGraphicFill() : Graphic
+setGraphicFill(graphicFill : Graphic) : void
+getColor() : Expression
+setColor(expression : Expression) : void
+getOpacity() : Expression
+setOpacity(expression : Expression) : void
1
Halo
+getFill() : Fill
+setFill(f : Fill) : void
+getRadius() : Expression
+setRadius(expression : Expression) : void
0..*
0..*
ExternalGraphic
+getOnlineResource() : String
+setOnlineResource(url : String) : void
+getInlineContent() : byte[]
+setInlineContent(content : byte[]) : void
+getFormat() : String
+setFormat(format : String) : void
Font
+getFamily() : Expression
+setFamily(expression : Expression) : void
+getStyle() : Expression
+setStyle(expression : Expression) : void
+getWeight() : Expression
+setWeight(expression : Expression) : void
+getSize() : Expression
+setSize(expression : Expression) : void
0..*
Mark
+getWellKnownName() : Expression
+setWellKnownName(name : Expression) : void
+getFill() : Fill
+setFill(f : Fill) : void
+getStroke() : Stroke
+setStroke(s : Stroke) : void

Figure 40 - Symbol and related classes

6.7.3 FeatureLayers

A `FeatureLayer` is a grouping of features that will be drawn on a display. Every `FeatureLayer` instance has a `FeatureCollection` that holds the features to be drawn, a z-ordering value, called its "level", and a `FeatureStyle` object that indicates how the features will be portrayed.

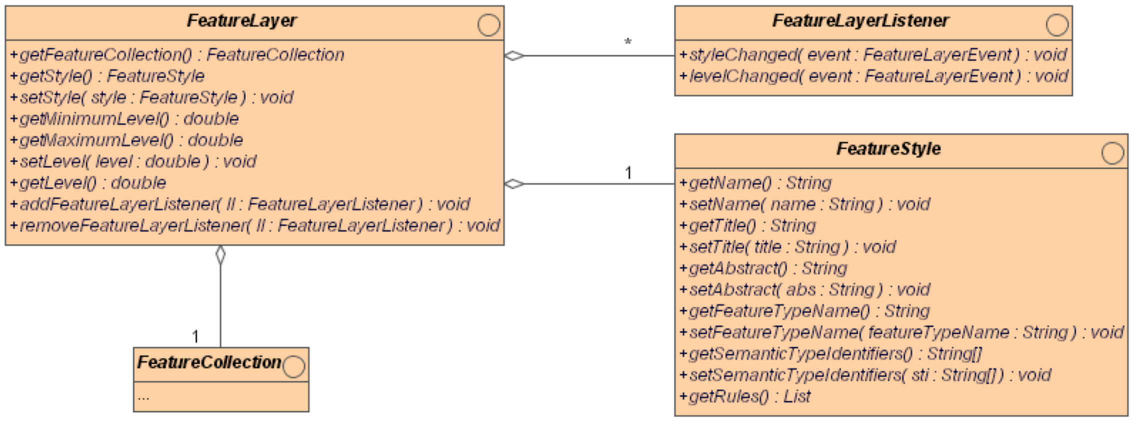

FeatureLayer
+getFeatureCollection() : FeatureCollection
+getStyle() : FeatureStyle
+setStyle(style : FeatureStyle) : void
+getMinimumLevel() : double
+getMaximumLevel() : double
+setLevel(level : double) : void
+getLevel() : double
+addFeatureLayerListener(ll : FeatureLayerListener) : void
+removeFeatureLayerListener(ll : FeatureLayerListener) : void
*
FeatureLayerListener
+styleChanged(event : FeatureLayerEvent) : void
+levelChanged(event : FeatureLayerEvent) : void
1
FeatureStyle
+getName() : String
+setName(name : String) : void
+getTitle() : String
+setTitle(title : String) : void
+getAbstract() : String
+setAbstract(abs : String) : void
+getFeatureTypeName() : String
+setFeatureTypeName(featureTypeName : String) : void
+getSemanticTypeIdentifiers() : String[]
+setSemanticTypeIdentifiers(sti : String[]) : void
+getRules() : List
1
FeatureCollection
...

Figure 41 - FeatureLayer

Additionally, `FeatureLayers` support an event mechanism that allows the user to receive notification when the style or z-order of the layer has been modified.

6.8 GraphicStore, GraphicStoreFactory

In some end user applications, there will be a need for "pluggable" components that provide `Graphic` objects to be drawn on a `Canvas`. The `GraphicStore` interface (similar in concept to the `FeatureStore` interface) gives a standard interface for such providers to implement.

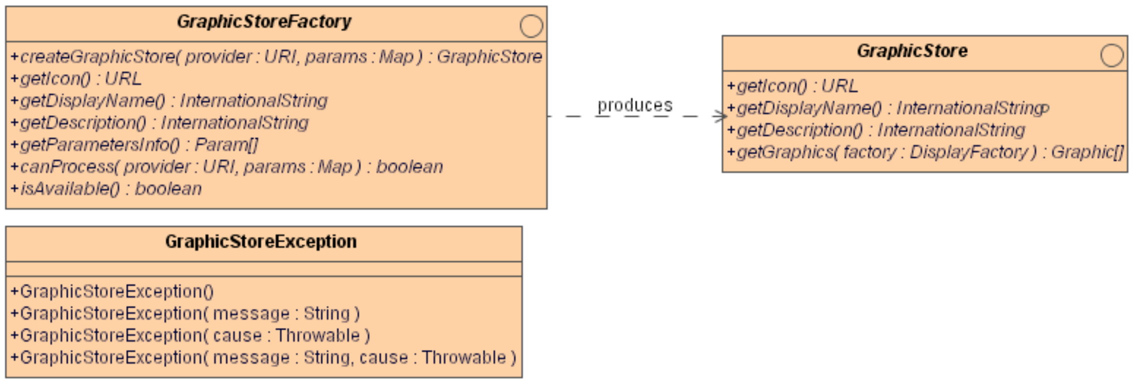

GraphicStoreFactory
+createGraphicStore(provider : URI, params : Map) : GraphicStore
+getIcon() : URL
+getDisplayName() : InternationalString
+getDescription() : InternationalString
+getParametersInfo() : Param[]
+canProcess(provider : URI, params : Map) : boolean
+isAvailable() : boolean
produces
GraphicStore
+getIcon() : URL
+getDisplayName() : InternationalString
+getDescription() : InternationalString
+getGraphics(factory : DisplayFactory) : Graphic[]
GraphicStoreException
+GraphicStoreException()
+GraphicStoreException(message : String)
+GraphicStoreException(cause : Throwable)
+GraphicStoreException(message : String, cause : Throwable)

Figure 42 - GraphicStore and GraphicStoreFactory

6.8.1 GraphicStore

There are only a few methods in the GraphicStore interface:

- `getIcon()` - Returns a URL to a small graphic that can be used to represent this `GraphicStore` in a user interface. Implementations are encouraged to keep this icon smaller than 32 pixels square.
- `getDisplayName()` - Returns a short text name that can be used to represent this `GraphicStore` in a user interface. This name should be unique enough to identify this instance, but need not be universally unique.
- `getDescription()` - Returns a longer, plain text description of the contents of this `GraphicStore`. This can be one or two sentences (or more).
- `getGraphics(...)` - This method is the heart of `GraphicStore`. The implementation uses the factory that is passed as a parameter to create new graphic instances for display. Implementations must create new `Graphic`s each time this method is called. The caller is responsible for caching `Graphic`s and may need to call this method multiple times with different factories for different displays.

6.8.2 GraphicStoreFactory

To facilitate "pluggable" architectures, the specification also defines a factory interface capable of creating `GraphicStore`s from a set of configuration parameters. These configuration parameters take the form of a URI, which indicates the raw data source, and a `Map` of parameters that provide other hints to the store for connecting to this raw data source (such as passwords, cache size, etc).

- `createGraphicStore(...)` - The heart of `GraphicStoreFactory`, this method instantiates a new `GraphicStore` based on the parameters given.
- `getIcon()`, `getDisplayName()`, `getDescription()` - Returns basic metadata that can be displayed to a user about this `GraphicStoreFactory`.
- `getParametersInfo()` - Returns a list of objects that detail which parameters can be passed in the `Map` given to the `createGraphicStore` method.
- `canProcess(...)` - This can be used by application code to determine if the given URL and map of parameters can be parsed and handled by this factory.
- `isAvailable()` - In certain cases, this factory may not be able to create the necessary connections to a back end data source. This could happen, for example, if a network connection is down, or if needed classes are not in the classpath. This method should return true if the necessary resources and libraries appear to be available and false otherwise.

6.8.3 Graphic Store Use Cases

There are four general categories this author can envision where application developers may find it useful to implement GraphicStores.

6.8.3.1 Support for Custom Graphics

A GraphicStore may be written in order to take advantage of support for custom graphics provided by a particular GO-1 implementation. An example might be an extension to an existing Graphic or implementation of a new graphic supporting a 3D geometry. Something like the following code snippet would be used by such a GraphicStore to provide the extended capability, and also default behaviour for standard GO-1 implementations. Note that a well-behaved GraphicStore taking advantage of extended capabilities should wherever possible provide a default behaviour based on standard GO-1 Graphics.

```
DisplayCapabilities displayCapabilities = displayFactory.getCapabilities();

Class specialPrimitiveClass = CustomGraphic.class;
boolean specialPrimitiveClassSupported = false;

// get the supported primitives
Class[] supportedPrimitives = capabilities.getSupportedPrimitives();

// iterate supported primitives, looking for the class we inquire about
for (int i = 0; i < supportedPrimitives.length &&
!specialPrimitiveClassSupported; i++) {
  if (supportedPrimitives[i] == specialPrimitiveClass) {
     specialPrimitiveClassSupported = true;
  }
}

// now either use the custom graphic or do some fallback routine
if (specialPrimitiveClassSupported) {
  CustomGraphic customGraphic = (CustomGraphic)
    displayFactory.createGraphic(CustomGraphic.class);
        // ...
} else {
  GraphicLineString fallbackGraphic = (GraphicLineString)
    displayFactory.createGraphic(GraphicLineString.class);
        // ...
}
```

6.8.3.2 Support for GraphicScaledImage

A GraphicStore can be written in order to support imagery that is not supported by the native implementation of the GO-1 `GraphicScaledImage`. For example, support for JPEG 2000 or MrSID could be achieved by business logic encapsulated in a `GraphicStore` that produced a `GraphicScaledImage` for use with the `Canvas`. The following code snippet illustrates how this could be done.

```
GraphicScaledImage scaledImage =
(GraphicScaledImage)displayFactory.createGraphic(GraphicScaledImage.class);

CRSAuthorityFactory crsAuthFactory = commonFactory.getCRSAuthorityFactory();
CoordinateReferenceSystem crs =
crsAuthFactory.createCoordinateReferenceSystem("EPSG:4326");
GeometryFactory geometryFactory = commonFactory.getGeometryFactory(crs);


// GraphicScaledImage corners behave as Envelope with lower/upper corners
DirectPosition lowerCorner =
   geometryFactory.createDirectPosition(new double[] { -90, -180 });
scaledImage.setLowerCorner(lowerCorner);

DirectPosition upperCorner =
   geometryFactory.createDirectPosition(new double[] { 90, 180 });
scaledImage.setUpperCorner(upperCorner);

scaledImage.setTransparency(50);

// Custom image generation code would go here
Image image = new ImageIcon(res.getString("wholeWorldImageFile")).getImage();

BufferedImage bufferedImage = new BufferedImage(
         image.getWidth(null),
         image.getHeight(null),
         BufferedImage.TYPE_INT_ARGB
```

```
    );
bufferedImage.getGraphics().drawImage(image, 0, 0, null);

scaledImage.setScaledImage(bufferedImage);
```

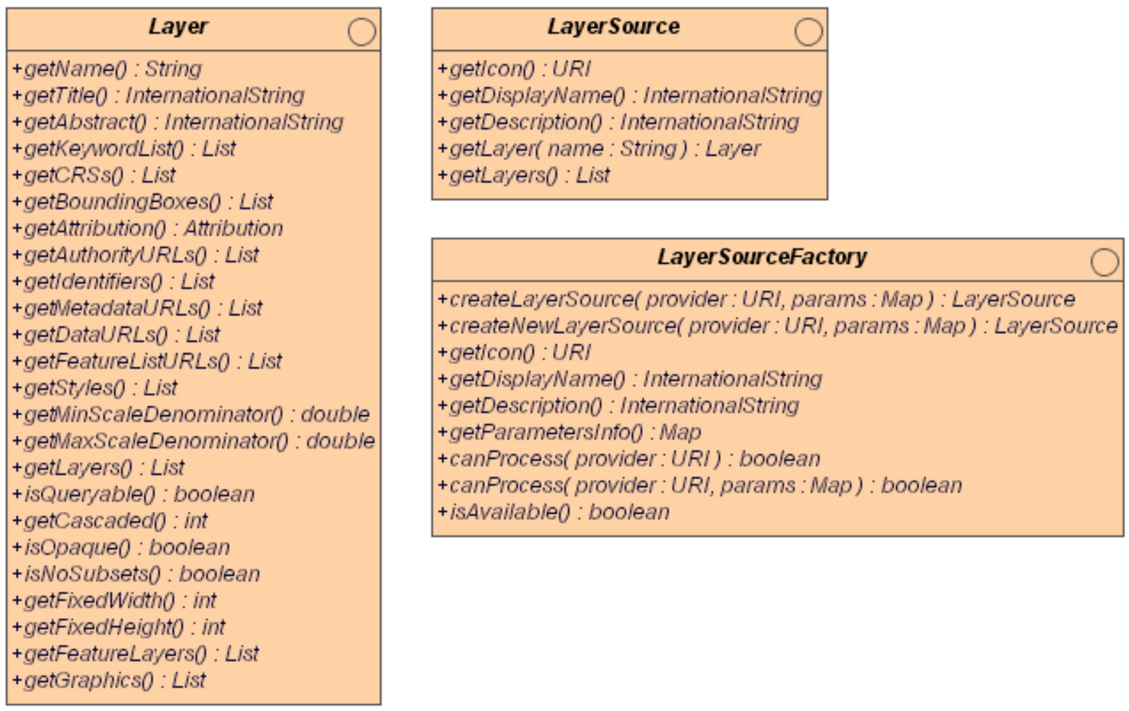
6.8.3.3 Data to Feature Mapping Issues

Implementing a `GraphicStore` may be useful in cases where an existing data source does not map well or at all to Features. For example, attempting to support gridded METOC displays with dynamically updating wind barbs would be a good candidate.

6.8.3.4 SLD Issues

In cases where SLD is insufficient, inefficient, or would be overly complex to implement it may prove a better choice to implement with a `GraphicStore`. For example, using SLD to support MS2525B with the full set of Tactical Graphics would require a multi-megabyte SLD file and likely require some type of service to serve up images. Business logic for the styling rules would also be complex.

6.9 Layer and LayerSource


Layer
+getName() : String
+getTitle() : InternationalString
+getAbstract() : InternationalString
+getKeywordList() : List
+getCRSs() : List
+getBoundingBoxes() : List
+getAttribution() : Attribution
+getAuthorityURLs() : List
+getIdentifiers() : List
+getMetadataURLs() : List
+getDataURLs() : List
+getFeatureListURLs() : List
+getStyles() : List
+getMinScaleDenominator() : double
+getMaxScaleDenominator() : double
+getLayers() : List
+isQueryable() : boolean
+getCascaded() : int
+isOpaque() : boolean
+isNoSubsets() : boolean
+getFixedWidth() : int
+getFixedHeight() : int
+getFeatureLayers() : List
+getGraphics() : List
LayerSource
+getIcon() : URI
+getDisplayName() : InternationalString
+getDescription() : InternationalString
+getLayer(name : String) : Layer
+getLayers() : List
LayerSourceFactory
+createLayerSource(provider : URI, params : Map) : LayerSource
+createNewLayerSource(provider : URI, params : Map) : LayerSource
+getIcon() : URI
+getDisplayName() : InternationalString
+getDescription() : InternationalString
+getParametersInfo() : Map
+canProcess(provider : URI) : boolean
+canProcess(provider : URI, params : Map) : boolean
+isAvailable() : boolean

6.9.1 Model and Rationale

In the development of applications using FeatureStores and GraphicStores as data sources, the Layer from ISO_19128 7.2.4.5 came across as an obvious mechanism for organizing data. Using a Layer, data could easily be presented in a Table of Contents or other data view. The Layer and its configuration helpers LayerSource and LayerSourceFactory were thus created.

The Layer provides an implementation of the ISO_19128 Layer and adds the additional GO-1 objects necessary to fulfull the connection between ISO_19128 and this specification. The Layer itself contains FeatureLayers and Graphics, while its creating LayerSource references the FeatureStores and GraphicStores used to create the GO-1 objects in the Layer.

6.9.2 LayerSource

With the exception of the FeatureLayer and Graphic methods, all methods in the Layer interface directly provide access to ISO_19128 Layer properties. Only accessors are provided as Layers are intended to be easily created or copied.

As a Layer implementing ISO_19128, a Layer may have many Styles. When getting the FeatureLayers and Graphics from a Layer, a GO-1 application may want to get FeatureStyles and GraphicStyles from the same Layer's Styles, or it may want to use some other style altogether.

LayerSource methods:

- `getLayers()` - Returns a List of Layers provided by this LayerSource. This List should not be a live List: modifying the returned List should not modify this LayerSource's Layers.
- `getLayer(String name)` - Returns the named Layer.

6.9.3 LayerSourceFactory

LayerSourceFactory methods:

- `createLayerSource(URI provider,Map<String,Object> params)`
  Ask for a LayerSource connecting to the indicated provider or service. The returned LayerSource may have been previously cached. Additional hints or configuration information may be provided according to the metadata indicated by `getParametersInfo()`. This information often includes security information such as username and password.
- `createNewLayerSource(URI provider,Map<String,Object> params)`
  Ask for a new LayerSource connecting to the indicated provider or service. Additional hints or configuration information may be provided according to the metadata indicated by `getParametersInfo()`. This information often includes security information such as username and password.
- `canProcess(URI provider)`
  Indicates this FeatureStoreFactory communicate with the indicated provider or service. This method should not fail. If a connection needs to be made to parse a GetCapabilities file or negotiate WMS versions, any IO problems indicate the inability to process the URI. This method may be considered the same as:

`canProcess(provider, hints)` where hints was generated by using all the default values specified by the `getParametersInfo()` method

6.9.4 Layer Example

The following code demonstrates how one can retrieve GO-1 objects from a Layer for display in a Canvas or FeatureCanvas.

```
Layer[] layers = getLayers();
Style[] styles = getStyles();
for (int i = 0; i < layers.length; i++) {
  // graphics
  final List<Graphic> graphics = layers[i].getGraphics();
  final List<GraphicStyle> graphicStyles =
styles[i].getGraphicStyles();
  for (int j = 0; j < graphics.size(); j++) {
    final Graphic graphic = (Graphic) graphics.get(j);
    for (int k = 0; k < graphicStyles.size(); k++) {
      final GraphicStyle graphicStyle = (GraphicStyle)
graphicStyles.get(i);
      graphic.setGraphicStyle(graphicStyle);
      canvas.add(graphic);
    }
    if (graphicStyles.size() == 0) {
      // use whatever style information graphic contains
      canvas.add(graphic);
    }
  }
  // featureLayers
  final List<FeatureLayer> featureLayers =
layers[i].getFeatureLayers();
  final List<FeatureStyle> featureStyles =
styles[i].getFeatureStyles();
  for (int j = 0; j < featureLayers.size(); j++) {
    final FeatureLayer featureLayer = (FeatureLayer)
featureLayers.get(j);
    for (int k = 0; k < featureStyles.size(); k++) {
      final FeatureStyle featureStyle = (FeatureStyle)
featureStyles.get(k);
      featureLayer.setStyle(featureStyle);
      featureCanvas.addFeatureLayer(featureLayer);
    }
    if (featureStyles.size() == 0) {
      // use whatever style information featureLayer contains
      featureCanvas.addFeatureLayer(featureLayer);
    }
  }
}
```

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.

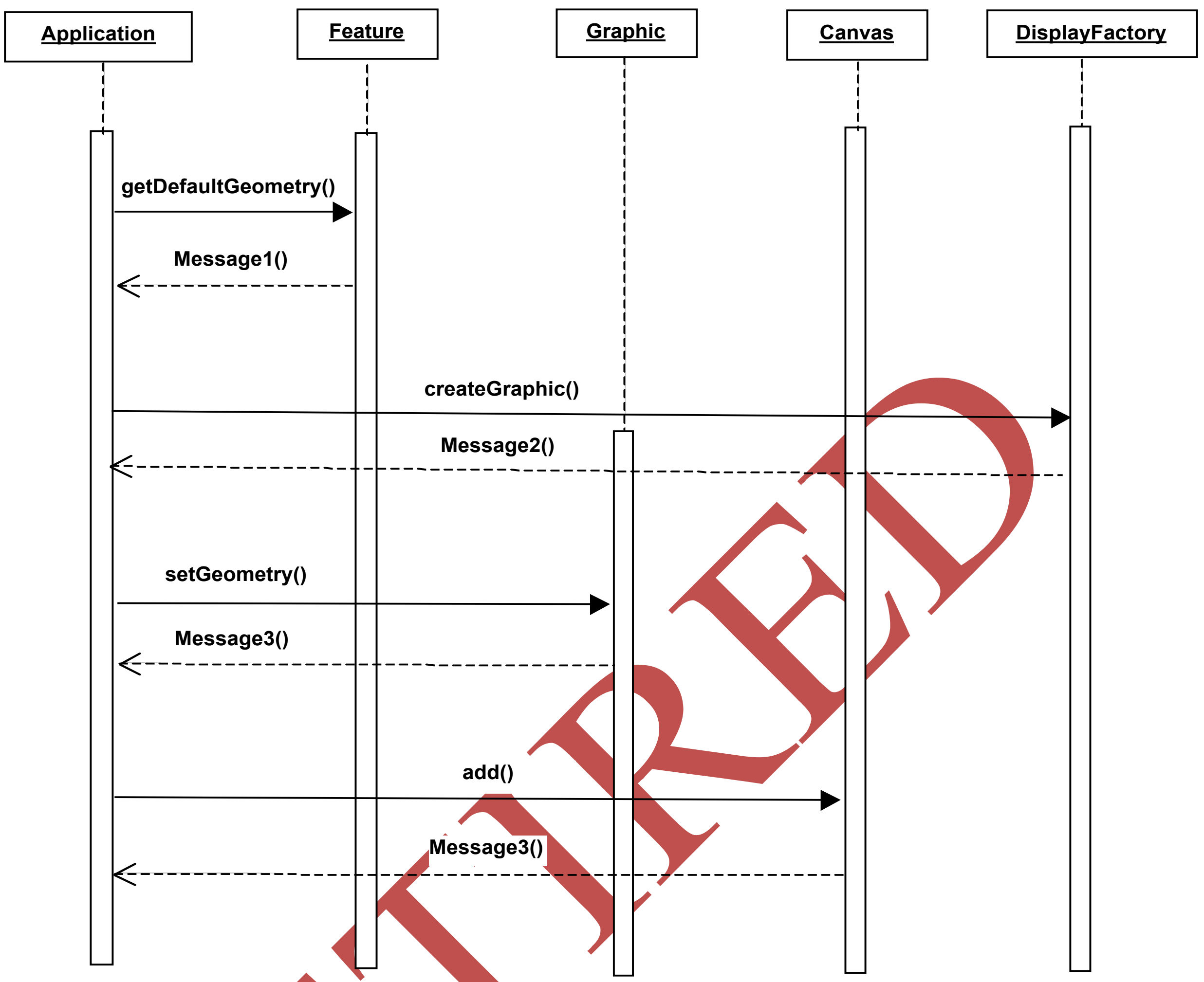


Figure 43 – Add Graphic Sequence Diagram

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,.

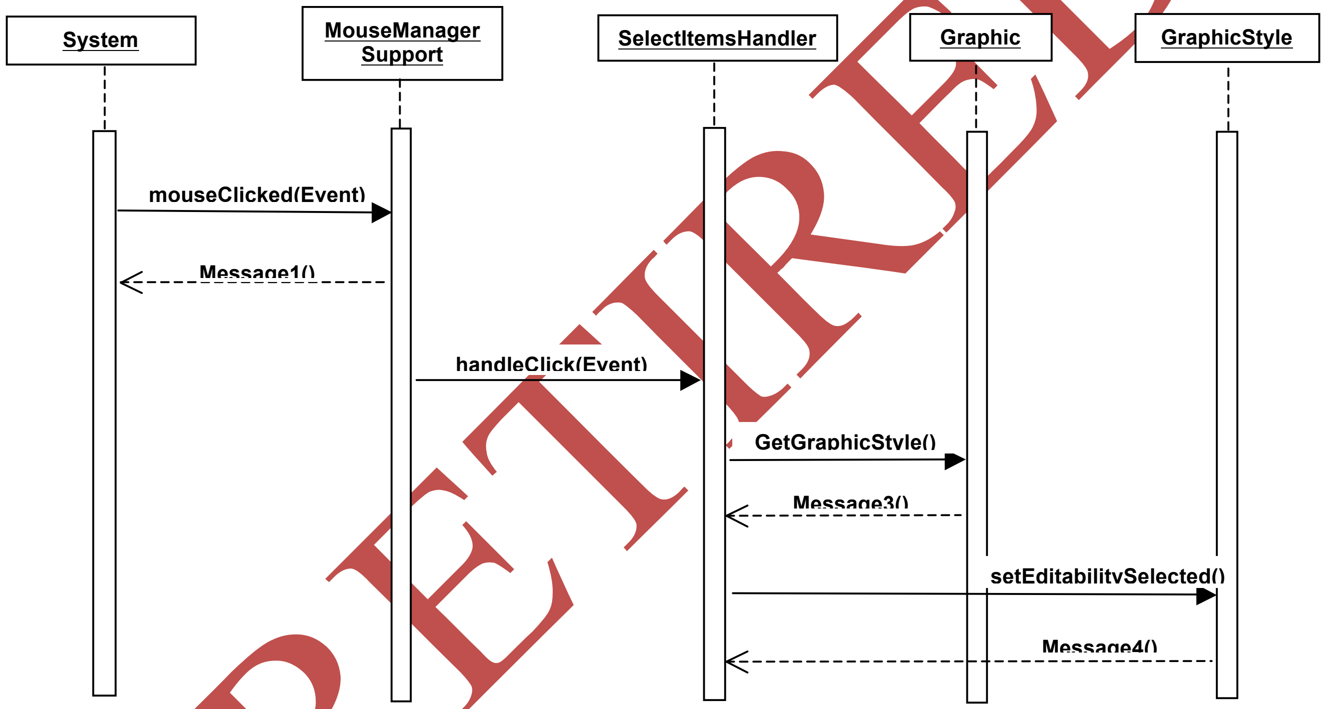


Figure 44 - Selecting a Graphic Object

7.2.1 Editing Graphics

Graphic objects purposefully leave editing up to the implementation. The two existing properties on Graphic base class, 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.

Graphic has an autoEdit attribute to support EditabilityDisplay conformance. When this property is true, the Canvas should provide a graphical interface that allows the user to modify the geometry of the graphic. Usually, this will take the form of "handles" (small circles or boxes) at the vertices or control points that the user can modify with 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 GraphicLineString 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).

Whenever a graphic is modified or deleted, a `GraphicChangeEvent` must be fired by the implementation. When editing starts, a `GraphicChangeEvent` with a flag of GraphicChangeEvent .EDITABLE_START is fired. When editing ends, a GraphicChangeEvent with a flag of GraphicChangeEvent .EDITABLE_END is fired.

There are seven event flags in total that must be supported by a GO-1 application. Three of these are specific to compliance with Editable Display Objects; EDITABLE_START, EDITABLE_CHANGED, and EDITABLE_END. The other four event flags are GRAPHIC_CHANGED, GRAPHIC_SELECTED, GRAPHIC_DESELECTED, and GRAPHIC_DISPOSED.

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

Preconditions: Running application has instantiated a geometry LineString 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.

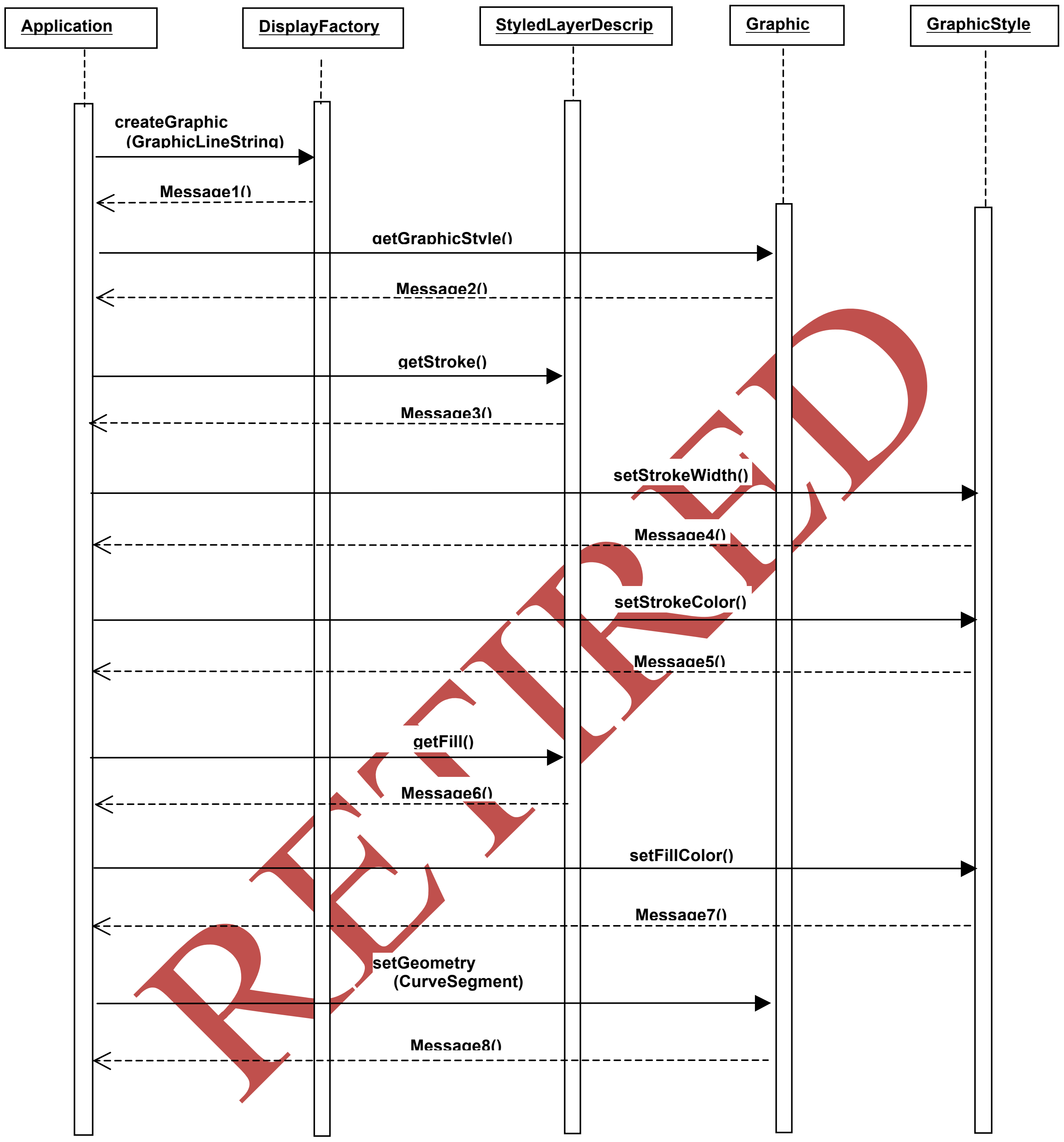


Figure 45 - Graphic Object Creation

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 `DirectPositions` 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.

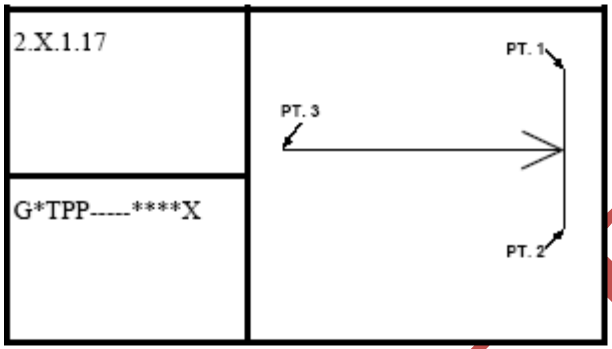


Figure 46 - MIL-STD 2525 Tactical Graphic

```
DisplayFactory displayFactory = commonFactory.getDisplayFactory();
GraphicLineString tacgraph =
(GraphicLineString)displayFactory.createGraphic(GraphicLineString.class);

CRSAuthorityFactory crsAuthFactory = commonFactory.getCRSAuthorityFactory();
CoordinateReferenceSystem crs =
crsAuthFactory.createCoordinateReferenceSystem("EPSG:4326");
GeometryFactory geometryFactory = commonFactory.getGeometryFactory(crs);

//anchor 1
DirectPosition anchor = geometryFactory.createDirectPosition(new double[] { 30,
20 });
tacgraph.addAnchorPoint(anchor);

//anchor 2
anchor = geometryFactory.createDirectPosition(new double[] { 30, -20 });
tacgraph.addPoint(anchor);

//anchor 3
anchor = geometryFactory.createDirectPosition(new double[] { 0, 0 });
tacgraph.addPoint (anchor);

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

tacgraph.getSymbology().setSymbologyProperty(symInfo, "SymbolID", "G*TPP-----
****X");
tacgraph.getSymbology().setSymbologyProperty(symInfo, "DirectionOfMovement",
new Double(6.281));
tacgraph.getSymbology().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.

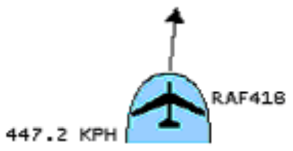


Figure 45 - MIL-STD 2525 Air Track

```
DisplayFactory displayFactory = commonFactory.getDisplayFactory();
GraphicIcon icon =
(GraphicIcon)displayFactory.createGraphic(GraphicIcon.class);

CRSAuthorityFactory crsAuthFactory = commonFactory.getCRSAuthorityFactory();
CoordinateReferenceSystem crs =
crsAuthFactory.createCoordinateReferenceSystem("EPSG:4326");
GeometryFactory geometryFactory = commonFactory.getGeometryFactory(crs);

//anchor point
DirectPosition anchor = geometryFactory.createDirectPosition(new double[] { 30,
20 });
icon.setPosition(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.getSymbology().setSymbologyProperty(symInfo, "SymbolID", "SFAPMF------
USA");
icon.getSymbology().setSymbologyProperty(symInfo, "AdditionalInformation",
"RAF418");
icon.getSymbology().setSymbologyProperty(symInfo, "Frame", Boolean.TRUE);
icon.getSymbology().setSymbologyProperty(symInfo, "Fill", Boolean.TRUE);
icon.getSymbology().setSymbologyProperty(symInfo, "Icon", Boolean.FALSE);
icon.getSymbology().setSymbologyProperty(symInfo, "Speed", new Double(447.2));
icon.getSymbology().setSymbologyProperty(symInfo, "DirectionOfMovement", new
Double(6.281));
icon.getSymbology().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".

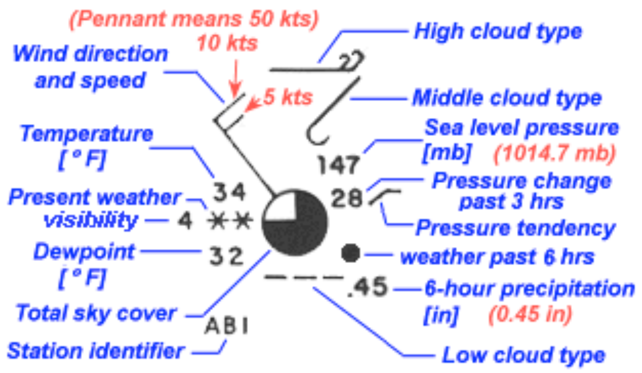


Figure 46 - Surface Weather (annotated)

```
DisplayFactory displayFactory = commonFactory.getDisplayFactory();
GraphicIcon icon =
(GraphicIcon)displayFactory.createGraphic(GraphicIcon.class);

CRSAuthorityFactory crsAuthFactory = commonFactory.getCRSAuthorityFactory();
CoordinateReferenceSystem crs =
crsAuthFactory.createCoordinateReferenceSystem("EPSG:4326");
GeometryFactory geometryFactory = commonFactory.getGeometryFactory(crs);

//anchor point
DirectPosition anchor = geometryFactory.createDirectPosition(new double[] {
24.2, -112.5 });

icon.setPosition(anchor);

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

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

icon.getSymbology().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 47 - Homeland Security Symbol

```
DisplayFactory displayFactory = commonFactory.getDisplayFactory();
GraphicIcon icon = (GraphicIcon)displayFactory.
createGraphic(GraphicIcon.class);

CRSAuthorityFactory crsAuthFactory = commonFactory.getCRSAuthorityFactory();
CoordinateReferenceSystem crs =
crsAuthFactory.createCoordinateReferenceSystem(“EPSG:4326”);
GeometryFactory geometryFactory = commonFactory.getGeometryFactory(crs);

//anchor point
DirectPosition anchor = geometryFactory.createDirectPosition(new double[] {
24.2, -112.5 });

icon.addAnchorPoint(anchor);
symbol.addAnchorPoint(anchor);

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

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

icon.getSymbology().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 Feature and 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=16

Annex C
(normative)
Open Source Information

The interfaces described in this document are available in source code form from the GeoAPI project's SourceForge web site:

http://geoapi.sourceforge.net

From this page, download version 1.0.0 to retrieve the version of the source code corresponding to this document.

The source can also be retrieved using anonymous CVS:

```
cvs -z9 -d :pserver:anonymous@cvs.sourceforge.net:/cvsroot/geoapi
   checkout -P -r Release-1_00_00 src
```

The GeoAPI working group maintains a mailing list where changes and issues with the API are discussed. To access this mailing list, including archives and subscription information, visit this page:

http://lists.sourceforge.net/lists/listinfo/geoapi-devel

Corresponding to this mailing list is an issue tracking tool allowing the developers to log issues and track progress on those issues. Visit this page for more information:

http://jira.codehaus.org/browse/GEO

Annex D
(non-normative)
FeatureCanvas Sequence Diagram

The following sequence diagram illustrates how a (naïve) implementation of FeatureCanvas might use a Layer to construct Graphics and add them to a Canvas.

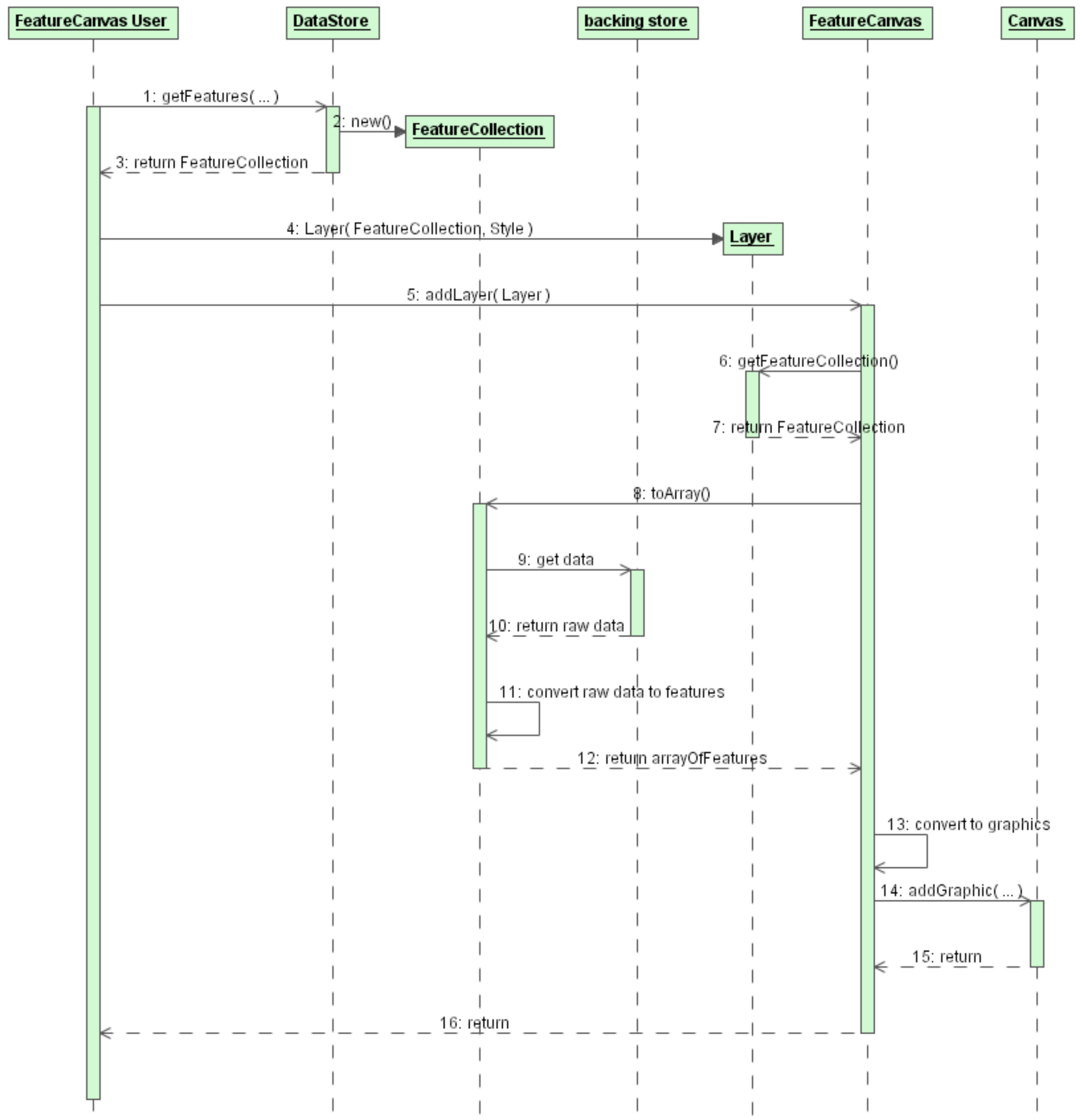


Figure 50 - One possible implementation of adding a Layer to a FeatureCanvas

Annex E
(non-normative)
Package Dependencies

Package Dependencies for selected GO1 Packages

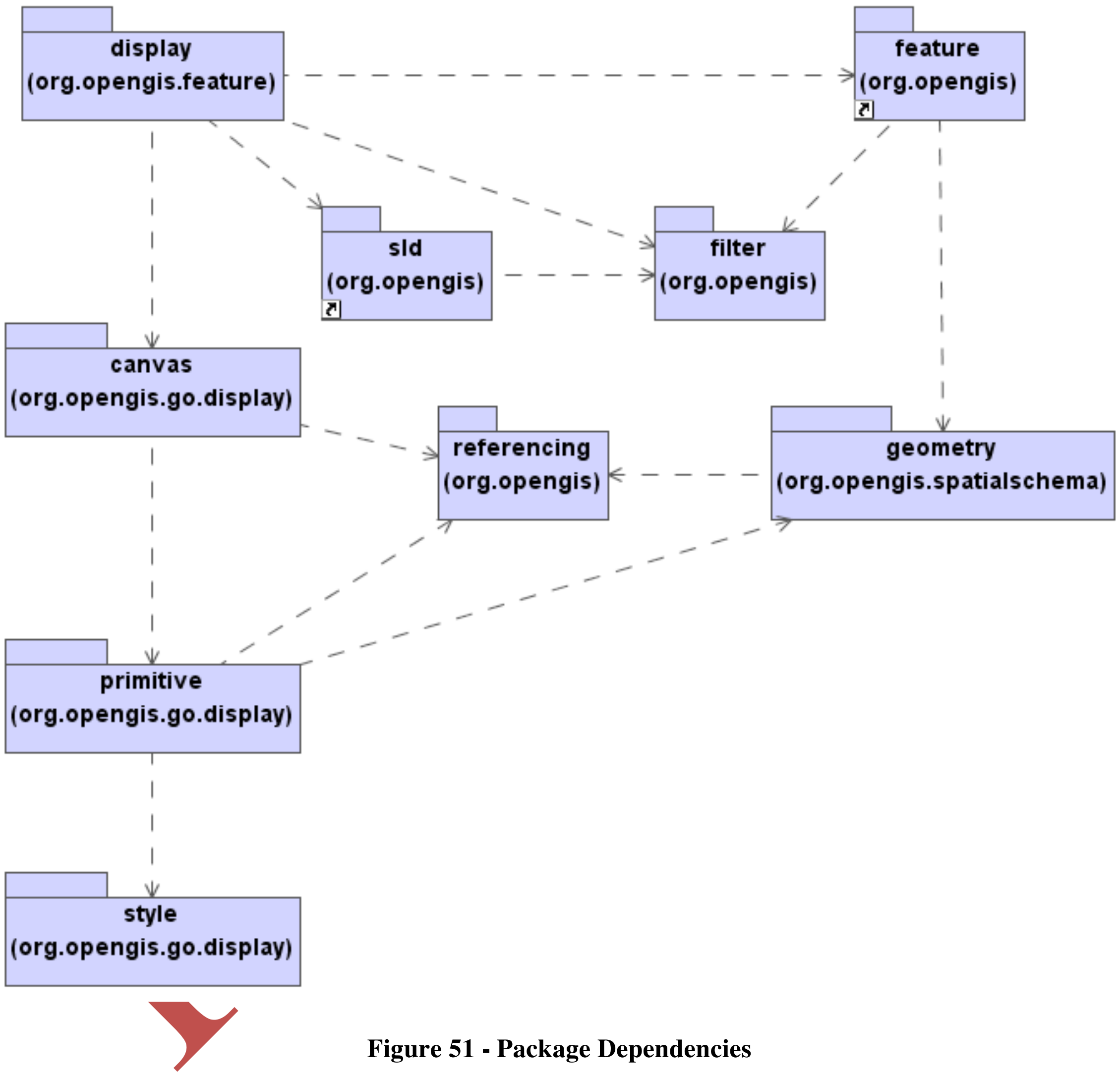


Figure 51 - Package Dependencies

Annex F
(non-normative)
Architecture Diagrams

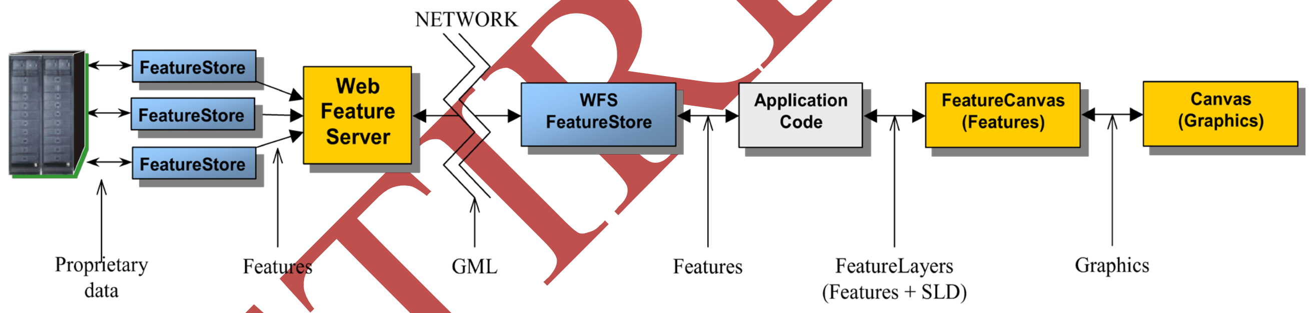


Figure 52 -A possible architecture using GO-1 components

Figure illustrates how an application using the GO-1 Architecture obtains data from a WFS and makes it available to the display. On the back end, the WFS use various `FeatureStores` to obtain feature data (ex. the existing ArcSDE `FeatureStore` or proposed TBMCS `FeatureStore`). The WFS `FeatureStore` allows features to be obtained from the WFS, again through the standardized GO-1 `FeatureStore` APIs. Application code then styles features and graphics and places them on the canvas.

This architecture is already proven to be viable by the GeoTools and GeoServer projects, although they use interfaces that were predecessors to those present in GO-1.

On the server side, a GO-1 Web Feature Server can easily be designed to only use the published `FeatureStore` interfaces for retrieving data from back-end data stores. This would allow for plugging in arbitrary `FeatureStores` to provide new data. Thus when new data is introduced to the system, the server does not change. A new `FeatureStore` is simply added to the classpath and the server is configured to read its feeatures.

On the client side, the `WFSFeatureStore` is capable of ingesting arbitrary GML from a WFS, so new data will not affect the application's data retrieval or display. Additionally, the application code can easily be written to only use `FeatureStore` APIs so that any other `FeatureStores` could be used locally as well.

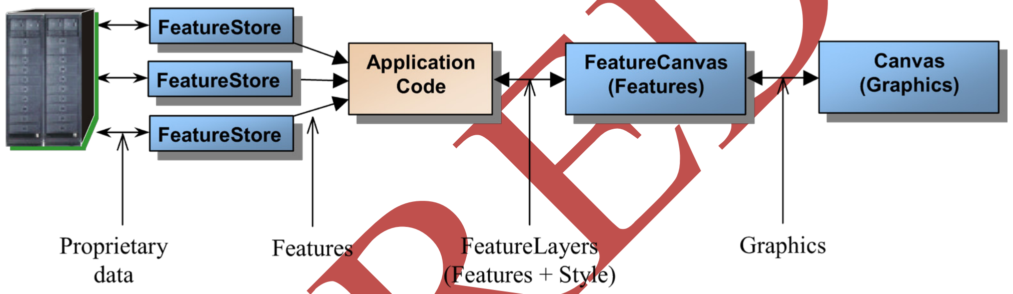


Figure53 - A non-network architecture using GO-1 components

Figure depicts a simpler, non-networked architecture that does not involve a Web Feature Server. Using this configuration, application code can be written in such a way that it only depends on GO-1 FeatureStore interfaces and does not have any dependencies on the characteristics of any particular Data Source.

Annex G
(non-normative)
Implementation Notes

Note that it is possible to implement the `FeatureCanvas` interface entirely on top of the GO-1 `Canvas` interfaces. In other words, you could write a `FeatureCanvas` implementation one time and reuse it for various `Canvas`es.

So suppose a new GIS rendering engine hits the market. To get `FeatureCanvas` support on this new rendering engine, all you have to do is implement the `Canvas` interfaces and you would get for very little effort a working `FeatureCanvas`.

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 01-101, v5.0. Available at: http://www.opengeospatial.org/docs/01-101.pdf

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

[7] OpenGIS ® Simple Feature Specification for SQLVersion, version 1.1. Available at: http://www.opengeospatial.org/docs/99-049.pdf

[8] OpenGIS ® Topic 5: The OpenGIS Feature, OGC document 99-105. Available at: http://www.opengeospatial.org/docs/01-105r2.pdf

[9] OpenGIS ® Grid Coverages Implementation Specification, version 1.0. Available at: http://portal.opengeospatial.org/files/?artifact_id=6628

[10] OpenGIS ® Catalog Service Implementation Specification, version 1.1.1. Available at: http://www.opengeospatial.org/docs/02-087r3.pdf

[11] OpenGIS ® Geography Markup Language (GML) Implementation Specification, version 2.1.2. Available at: http://www.opengeospatial.org/docs/02-069.pdf

[12] OpenGIS ® Geography Markup Language (GML) Implementation Specification (version 3.0), OGC document 02-023r4. Available at: https://portal.opengeospatial.org/files/?artifact_id=7174

[13] OpenGIS ® Web Mapping Server (WMS) Implementation Specification, version 1.1.1. Available at: http://www.opengeospatial.org/docs/01-068r3.pdf

[14] OpenGIS ® Styled Layer Descriptor (SLD) Implementation Specification, version 1.0. Available at: https://portal.opengeospatial.org/files/?artifact_id=1188

[15] OpenGIS ® Web Feature Server (WFS) Implementation Specification, version 1.0. Available at: https://portal.opengeospatial.org/files/?artifact_id=7176

Filter Encoding:

[16] OpenGIS® Filter Encoding Implementation Specification, version 1.0. Available at: http://www.opengeospatial.org/docs/02-059.pdf

[17] OpenGIS ® Coordinate Transformation Services Implementation Specification, version 1.0, OGC document 01-009. Available at: http://www.opengeospatial.org/docs/01-009.pdf

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

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

[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.opengeospatial.org/specs/?page=discussion

[22] Registry Service, Discussion Paper, OGC document 03-024. Available at: http://www.opengeospatial.org/specs/?page=discussion

[23] Integrated Client for OGC Services, Discussion Paper. OGC document 03-021. Available at: http://www.opengeospatial.org/specs/?page=discussion

[24] OWS Service Information Model, Discussion Paper, OGC document 03-026. Available at: http://www.opengeospatial.org/specs/?page=discussion

[25] OpenGIS ® Web Service Architecture, Discussion Paper, OGC document 03-025. Available at: http://www.opengeospatial.org/specs/?page=discussion

[26] OGC Reference Model (ORM), OGC document 02-077. Available at: http://www.opengeospatial.org/specs/?page=discussion

[27] Filter Encoding, OGC document 02-059, v1.0. Available at: http://www.opengeospatial.org/docs/02-059.pdf

[28] UML for Spatial Referencing by Coordinates, OGC document 03-009R5.

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