Open GIS Consortium Inc.

Date: 2003-06-03

Reference number of this OpenGIS© project document: **OGC 03-064r1**

Version: 0.2.0

Category: OpenGIS© Implementation

Editor: Phillip C. Dibner, Ecosystem Associates / OGC

GO-1 Application Objects Draft Interoperability Program Report

Document type:	OpenGIS® Interoperability Program Report
Document stage:	Draft
Document language:	English

Contents

i. Preface

This document is a draft of the OpenGIS“ Application Objects Implementation Specification, hereinafter “AOS”. The AOS is one of a family of specifications that make up the OGC Geographic Objects activity. The Geographic Objects Initiative 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. It provides both an abstract object specification (in UML) and a JAVA specific profile to that that specification. The language-specific binding specifications themselves serve as open Application Programmer Interface (API) specifications for these Application Objects.

ii. Submitting organizations

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

a) Polexis

b) Northrop Grumman Information Technology

c) Pennsylvania State University

iii. Submission contact points

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

CONTACT	COMPANY	ADDRESS	PHONE/FAX	EMAIL
Eric Bertel	Polexis			eric@polexis.com
John Davidson	Image Matters			johnd@imagematte

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

iv. Revision history

Date	Release	Author	Paragraph modified	Description
19 May 03	0.1.9	P. Dibner		First public draft
03 June 03	0.2.0	P. Dibner		Cleanup for June 2003 TC

v. Changes to the OpenGIS® Abstract Specification

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

vi. Future Work

The Application Objects specification currently defines a set of core packages that support the management of Features, 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 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 reuse Geometry and Graphics objects to render many similar items (e.g., thousands of LineStrings) 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 gridded 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:

- Relative Coordinates - It remains to be explored whether there are CRS, especially well-known or easily constructed ones, for which the GO-1 approach for relative coordinates would lead to inconsistencies or other unanticipated consequences.

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

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

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

Foreword

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

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

• Clause 1: Scope

• Clause 2: Conformance

• Clause 3: Normative references

• Clause 4: Terms and definitions

• Clause 5: Conventions

• Clause 6: Design and Specification for Application Objects

• Clause 7 Behaviours

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

• Bibliography

Introduction

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

OpenGIS© Interface — Application Objects

1 Scope

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

2 Conformance

Although a discussion of conformance is not a requirement for an OGC IP IPR, DIPR, or Discussion Paper, we provide one here in the anticipation that this document will be promoted as a draft specification.

2.1 Types of Conformance

This document recognises two broad categories of 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* says not only that the required operations can be invoked, but that the component performs the operations in a standard and universally understood manner.

Because the GeoAPI 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 Information Objects, can be implemented as compliant standalone object libraries. Others 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 them conformance implies conformance to the object suites upon which they depend as well.

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

2.2 Display Object Conformance

The Display Objects described in this document include the Canvas, Graphic, Control, and GraphicStyle. Controls provide user input to the Canvas. Graphics objects are the entities that a Canvas manipulates and renders according to the styling attributes of a GraphicStyle object. Together, they constitute the display subsystem of an application.

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

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

2. Among the patterns of interest will be a consistent means of ingesting data from a variety of sources, including OGC Features and related objects.

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

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

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

2.3 Information Object Conformance

Geometries, Features and Feature Collections, styling objects like SLDs, and related entities constitute the information objects defined by GO-1. These build upon the body of work that has resulted in the OGC Simple Features specifications, ISO-19107, and several OGC Discussion and Recommendation Papers.

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

2.4 OGC Service Conformance

An application may derive its data from one or more OGC data services as defined by any of the OGC web data service specifications. It may also act as a client to transformation or other processing services when they become available.

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

3 Normative references

The following normative documents contain provisions which, 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
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf

CRS	Coordinate Reference System
DCE	Distributed Computing Environment
DCP	Distributed Computing Platform
DCOM	Distributed Component Object Model
GO-1	Geographic Objects, Phase 1
IDL	Interface Definition Language
ISO	International Organisation for Standardisation
OGC	Open GIS Consortium
SLD	Styled Layer Descriptor
SRS	Spatial Reference System
UML	Unified Modelling Language
XML	eXtended Markup Language
1D	One Dimensional
2D	Two Dimensional
3D	Three Dimensional

5.2 UML Notation

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

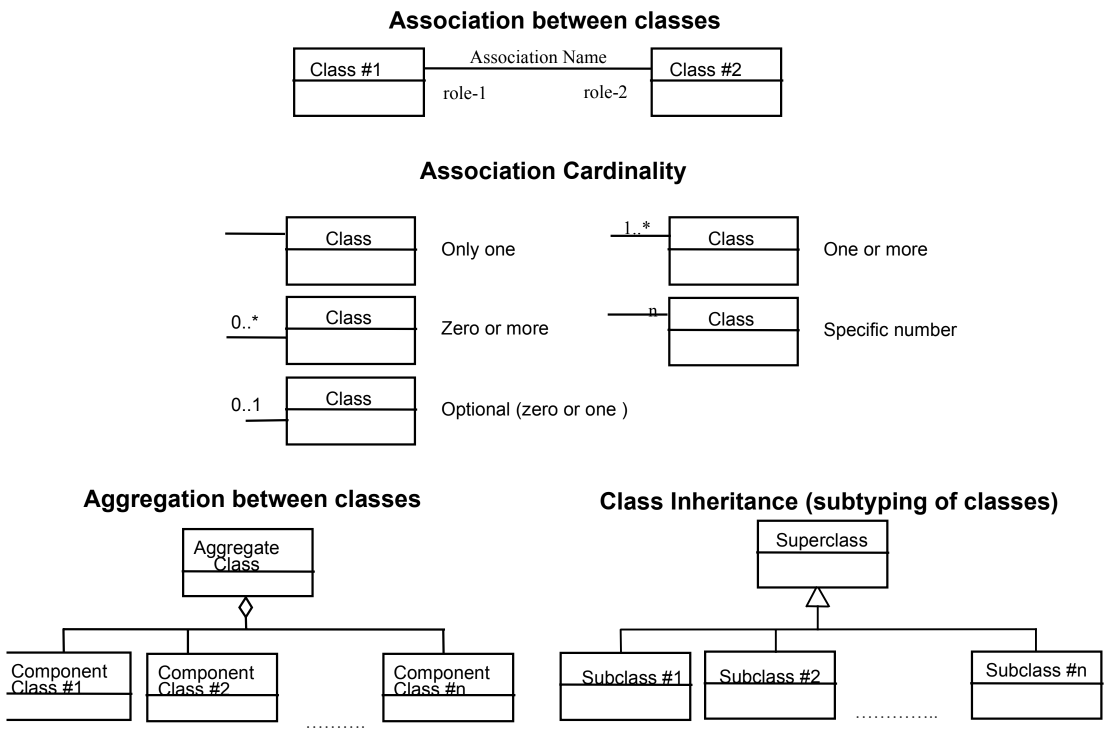


Figure 1 - UML notation

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

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

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

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

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

a) CharacterString – A sequence of characters

b) Integer – An integer number

c) Double – A double precision floating point number

d) Float – A single precision floating point number

6 Application Object Definitions

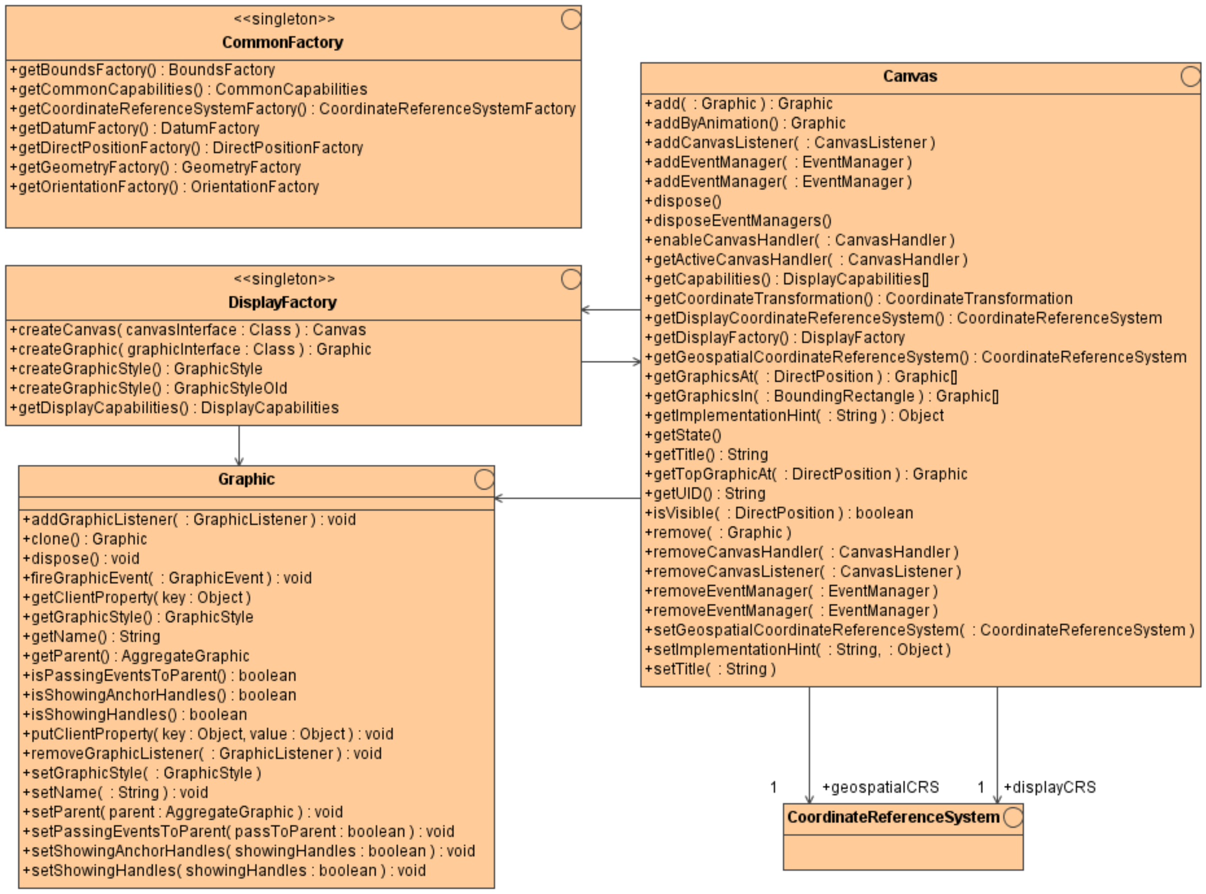
6.1 Display Objects

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

6.1.1 Canvas

6.1.1.1 General Description

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



<<singleton>>
CommonFactory
+getBoundsFactory() : BoundsFactory
+getCommonCapabilities() : CommonCapabilities
+getCoordinateReferenceSystemFactory() : CoordinateReferenceSystemFactory
+getDatumFactory() : DatumFactory
+getDirectPositionFactory() : DirectPositionFactory
+getGeometryFactory() : GeometryFactory
+getOrientationFactory() : OrientationFactory
<<singleton>>
DisplayFactory
+createCanvas(canvasInterface : Class) : Canvas
+createGraphic(graphicInterface : Class) : Graphic
+createGraphicStyle() : GraphicStyle
+createGraphicStyle() : GraphicStyleOld
+getDisplayCapabilities() : DisplayCapabilities
Graphic
+addGraphicListener(: GraphicListener) : void
+clone() : Graphic
+dispose() : void
+fireGraphicEvent(: GraphicEvent) : void
+getClientProperty(key : Object)
+getGraphicStyle() : GraphicStyle
+getName() : String
+getParent() : AggregateGraphic
+isPassingEventsToParent() : boolean
+isShowingAnchorHandles() : boolean
+isShowingHandles() : boolean
+putClientProperty(key : Object, value : Object) : void
+removeGraphicListener(: GraphicListener) : void
+setGraphicStyle(: GraphicStyle)
+setName(: String) : void
+setParent(parent : AggregateGraphic) : void
+setPassingEventsToParent(passToParent : boolean) : void
+setShowingAnchorHandles(showingHandles : boolean) : void
+setShowingHandles(showingHandles : boolean) : void
Canvas
+add(: Graphic) : Graphic
+addByAnimation() : Graphic
+addCanvasListener(: CanvasListener)
+addEventManager(: EventManager)
+addEventManager(: EventManager)
+dispose()
+disposeEventManagers()
+enableCanvasHandler(: CanvasHandler)
+getActiveCanvasHandler(: CanvasHandler)
+getCapabilities() : DisplayCapabilities[]
+getCoordinateTransformation() : CoordinateTransformation
+getDisplayCoordinateReferenceSystem() : CoordinateReferenceSystem
+getDisplayFactory() : DisplayFactory
+getGeospatialCoordinateReferenceSystem() : CoordinateReferenceSystem
+getGraphicsAt(: DirectPosition) : Graphic[]
+getGraphicsIn(: BoundingRectangle) : Graphic[]
+getImplementationHint(: String) : Object
+getState()
+getTitle() : String
+getTopGraphicAt(: DirectPosition) : Graphic
+getUID() : String
+isVisible(: DirectPosition) : boolean
+remove(: Graphic)
+removeCanvasHandler(: CanvasHandler)
+removeCanvasListener(: CanvasListener)
+removeEventManager(: EventManager)
+removeEventManager(: EventManager)
+setGeospatialCoordinateReferenceSystem(: CoordinateReferenceSystem)
+setImplementationHint(: String, : Object)
+setTitle(: String)
1
+geospatialCRS
1
+displayCRS
CoordinateReferenceSystem

Figure 2 - Canvas and related classes

6.1.1.2 Output Device

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

6.1.1.3 Input Device

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

6.1.1.4 Coordinate Reference System

The `Canvas` maintains two coordinate reference systems (CRS). The first `Canvas` CRS is associated with the geometry of the display device. Most computer screens are a rectangular array of pixels, and would use an `XY` or `PixelCoordinate` CRS. A planetarium or IMAX theatre is a spherical display, and might require a spherical CRS.

The second `Canvas` CRS is associated with the geospatial data rendered by the `Canvas`. This is typically the CRS for the rendered map. As `Graphics` are added to the `Canvas`, the `Canvas` determines the CRS for each `Graphic`. If that CRS is different from the `Canvas` geospatial CRS, then the `Canvas` uses a Coordinate Transformation Service (CTS) to align the `Graphic` geometry properly with the map. Any such transformations are private to the `Canvas`; the `Graphic` object as seen by external entities does not change. The `Canvas` may implement such a CTS internally, or use one or more external, possibly remote, services. This is a matter of implementation only, and is invisible to clients of the `Canvas`.

A `Canvas` may support Relative Coordinates. While not mandatory, this feature can confer significant advantages by allowing objects to be associated with one another while only having to maintain and update the absolute position of one of them.

6.1.1.5 Z-Order and Rendering of Graphics

The `Canvas` controls the visual layering, or z-order, of the `Graphic` objects it contains. The z-order allows `Graphics` to overlap and occlude each other in a controllable way. The Canvas may optimise its display by not rendering `Graphics` that are fully occluded.

Furthermore, when a location on the display is selected by an input device, 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, `Graphics` 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 may assume a very large range of values, perhaps as many as a long integer.

6.1.1.6 Canvas State

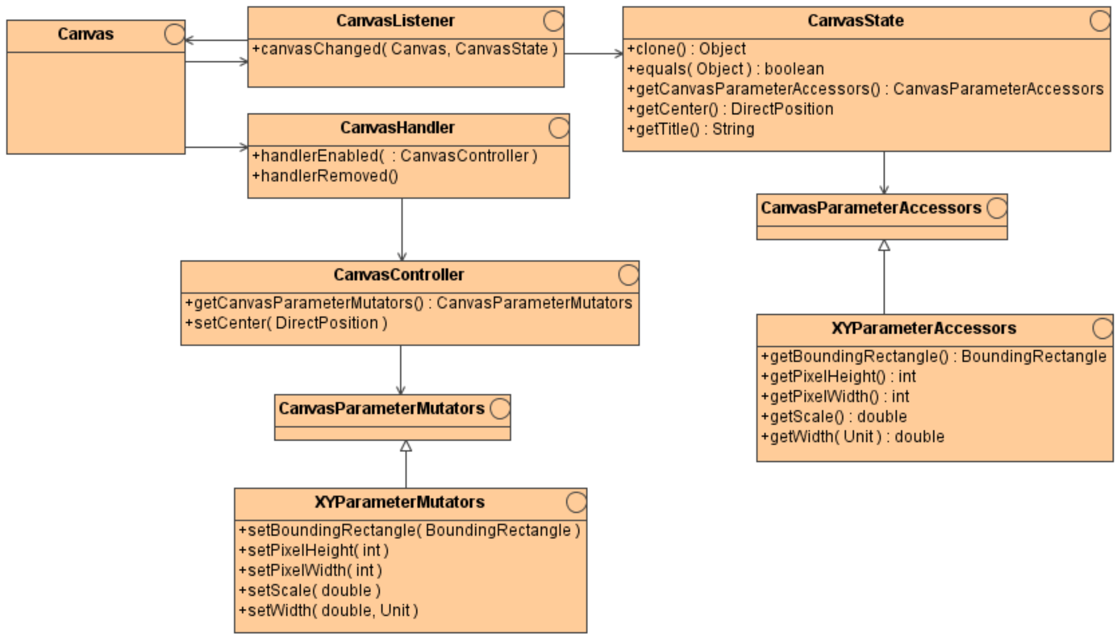

Canvas
CanvasListener
+canvasChanged(Canvas, CanvasState)
CanvasState
+clone() : Object
+equals(Object) : boolean
+getCanvasParameterAccessors() : CanvasParameterAccessors
+getCenter() : DirectPosition
+getTitle() : String
CanvasHandler
+handlerEnabled(: CanvasController)
+handlerRemoved()
CanvasParameterAccessors
CanvasController
+getCanvasParameterMutators() : CanvasParameterMutators
+setCenter(DirectPosition)
XYParameterAccessors
+getBoundingRectangle() : BoundingRectangle
+getPixelHeight() : int
+getPixelWidth() : int
+getScale() : double
+getWidth(Unit) : double
CanvasParameterMutators
XYParameterMutators
+setBoundingRectangle(BoundingRectangle)
+setPixelHeight(int)
+setPixelWidth(int)
+setScale(double)
+setWidth(double, Unit)

Figure 3 - Canvas state and controls

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

For an XY display CRS, `CanvasState` consists of the following properties:

* `pixelWidth`

* `pixelHeight`

* `center`

* `width`

* `scale`

* `boundingRectangle`

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

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

6.1.1.7 Canvas Capabilities

Like many OGC services, the `Canvas` supports a getCapabilities operation that allows it to describe the features it supports. An application attempting to use a given Canvas can invoke this method to determine whether that Canvas is suitable for rendering its graphic information, or whether it would have to do extra work in order to use the Canvas.

The getCapabilities method returns an object that implements the Capabilities interface. This object may then be queried about support for specific features.

Among the kinds of information an application may discover are various types of graphical rendering that the Canvas 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.

It is also possible to ask whether a Canvas can accept Graphics in Coordinate Reference Systems (CRS) other than its own current geospatial CRS. If so, it returns a reference to a Coordinate Transformation Service (CTS) object, against which the client software may invoke a getCapabilities query to learn which CRS the Canvas is able to accommodate.

A Canvas may also be queried to find out if it supports Relative Coordinates. Not all displays are capable of dealing with them, and the query mechanism provides a way for applications to become aware of such systems and deal with them gracefully.

Several classes associated with a Canvas are portrayed schematically in Figure 4.

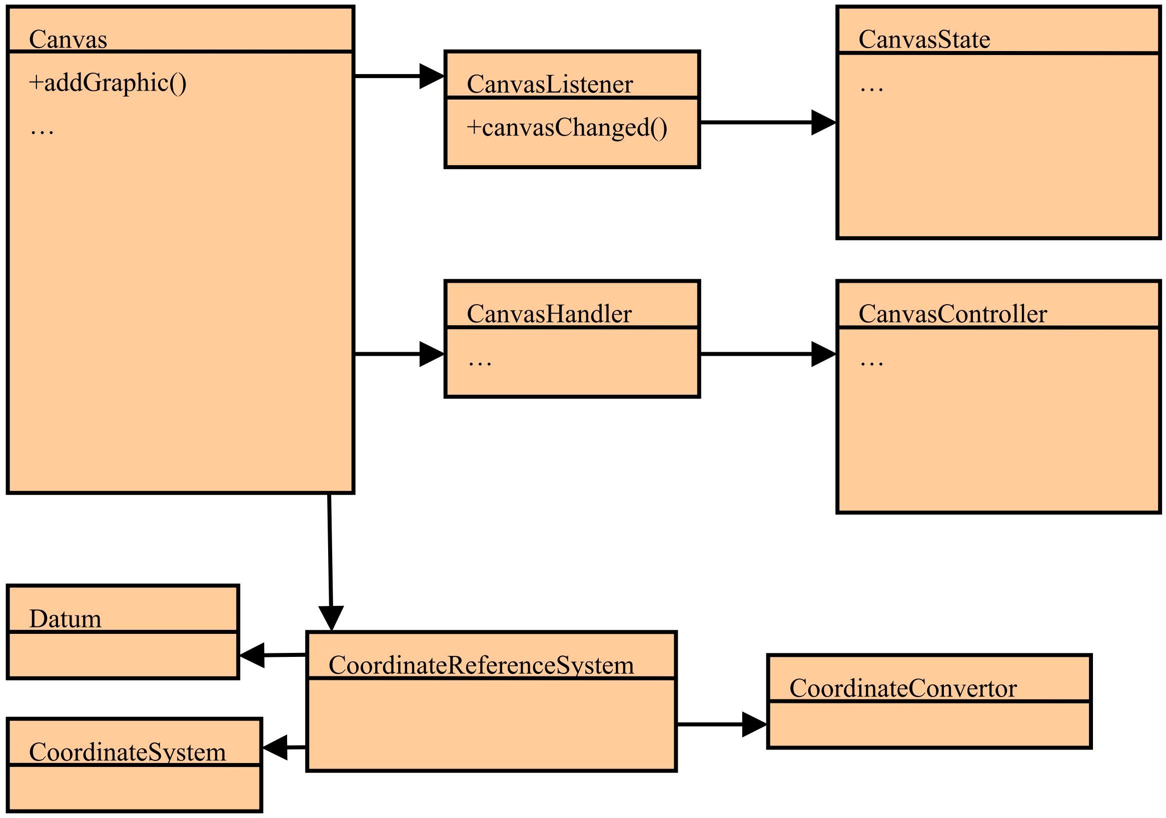


Figure 4 - Canvas and Associated Classes

6.1.2 Control

6.1.2.1 Input

6.1.2.1.1 Model and Rationale

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

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

The event-handling machinery for each device includes a stack of `EventHandlers`. When the `EventListener` receives an Event, it passes it to the first Handler on the stack. Each Handler implements specialized functionality – the system's response to the Event - that it executes when it receives an Event. Then it can either consume the `Event`, or pass it on to the next Handler in the stack. Thus the response of the system to a control input depends directly on the flow of `Events` through the Handler stack.

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

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

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

6.1.2.1.2 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 Handling system. 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 (also represented by an object, the `EventSource`) in order to receive events generated by that source. Some event sources, such as those that generate mouse or keyboard events, are present by default in the underlying system. Others may be implemented in the application or in library packages. Event sources *per se* are not a part of the response system documented here, but they motivate one important aspect of its organisation: for each source in a GO-1 implementation there is an event handling subsystem whose structure is described by the following paragraphs.

6.1.2.1.3 ManagerSupport Object

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

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

6.1.2.1.4 EventHandler Stack

For each event chain, there is at least one `EventHandler` object. The Handlers do the actual work of mediating the system's response to an event. For example, a `MouseHandler` implements a `mouseClicked()` operation that may cause an object to be selected or highlighted.

Like the `ManagerSupport` object, a GO-1 Handler implements the `EventListener` interface appropriate to its source, but it does not register as a listener. Instead, it implements the interface in order to inherit (and perhaps override) the relevant event handling methods.

As noted above, each `Support` object has a stack of Handlers. When it receives an event, the `Support` object invokes the appropriate action method against the top Handler on the stack. The `Handler` performs whatever specialized function this method implements, and then either consumes the event or returns it. In the latter case, the `Support` object invokes the method against the next handler on the stack, and so on until the event is consumed. Thus an event may trigger a series of responses that varies according to the arrangement of `EventHandlers` on the stack. This mechanism may be used to implement modal behaviours in response to input events, such as a change from selection behaviours to editing behaviours in the application's response to mouse gestures.

6.1.2.1.5 EventManager

The `ManagerSupport` object also implements a subclass of the `EventManager` interface. `EventManagers` are concerned primarily with maintaining the stack of Handlers. They have methods to enable, push, pop, find, remove, and replace Handlers on the stack. Thus `EventManager` is the interface through which modal changes in the response to an event are mediated.

Much of the user input will be processed by the `Canvas`, which has its own specialized variation of this structure. The `Graphic` class, described under Graphical Data Objects below, also has a structure that differs in some respects from the above. These differences are described in the following two sections.

6.1.2.2 Canvas Control

Control issues affecting and implemented by the `Canvas` class are discussed above in Section 6.1.1.6, Canvas State.

6.1.2.3 Graphics Control

Graphics control does not differ greatly from generic control issues except that several events can occur essentially in parallel. Graphical events are received by objects that implement `GraphicListener`. The default class that does this is `GraphicListenerSupport`, which is also able to fire `Events` that are sent to input event management objects.

6.2 Graphical Data Objects

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

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

The classes described here are therefore lighter weight and less general than the ISO Geometry classes described in Section 6.3, the OGC Abstract Specification, and ISO document 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 information objects as arguments.

6.2.1 Graphic

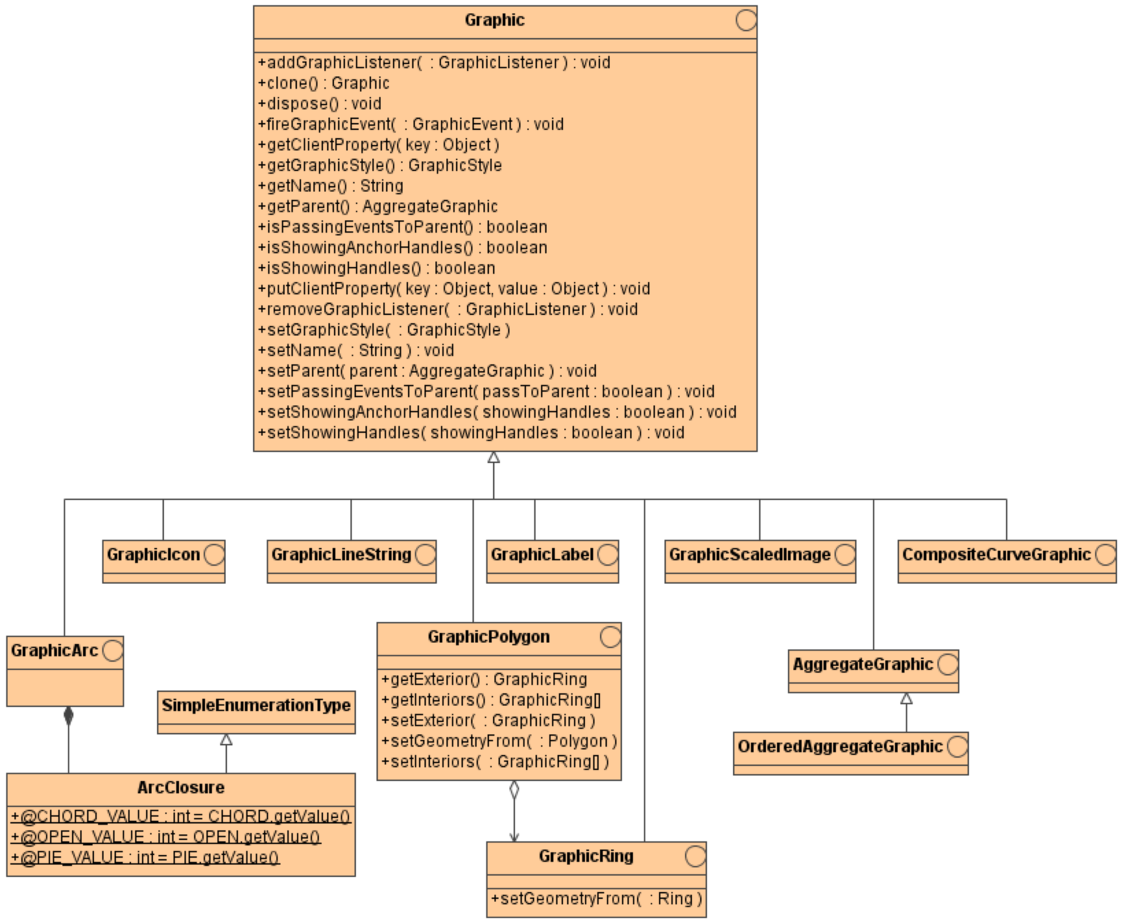

Graphic
+addGraphicListener(: GraphicListener) : void
+clone() : Graphic
+dispose() : void
+fireGraphicEvent(: GraphicEvent) : void
+getClientProperty(key : Object)
+getGraphicStyle() : GraphicStyle
+getName() : String
+getParent() : AggregateGraphic
+isPassingEventsToParent() : boolean
+isShowingAnchorHandles() : boolean
+isShowingHandles() : boolean
+putClientProperty(key : Object, value : Object) : void
+removeGraphicListener(: GraphicListener) : void
+setGraphicStyle(: GraphicStyle)
+setName(: String) : void
+setParent(parent : AggregateGraphic) : void
+setPassingEventsToParent(passToParent : boolean) : void
+setShowingAnchorHandles(showingHandles : boolean) : void
+setShowingHandles(showingHandles : boolean) : void
GraphicIcon
GraphicLineString
GraphicLabel
GraphicScaledImage
CompositeCurveGraphic
GraphicArc
SimpleEnumerationType
GraphicPolygon
+getExterior() : GraphicRing
+getInteriors() : GraphicRing[]
+setExterior(: GraphicRing)
+setGeometryFrom(: Polygon)
+setInteriors(: GraphicRing[])
AggregateGraphic
OrderedAggregateGraphic
ArcClosure
+@CHORD_VALUE : int = CHORD.getValue()
+@OPEN_VALUE : int = OPEN.getValue()
+@PIE_VALUE : int = PIE.getValue()
GraphicRing
+setGeometryFrom(: Ring)

Figure 5 - Graphic

6.2.1.1 General Description

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

There are two broad categories of `Graphics`: primitives and aggregates. Primitive types are subclassed directly from Graphic, and include `GraphicLineString`, `GraphicScaledImage`, `GraphicIcon`, `GraphicArc`, and `GraphicLabel`. `Aggregates are collections of primitives.`

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.

`Graphic` objects are instantiated with a Factory pattern.

6.2.1.2 Primitives

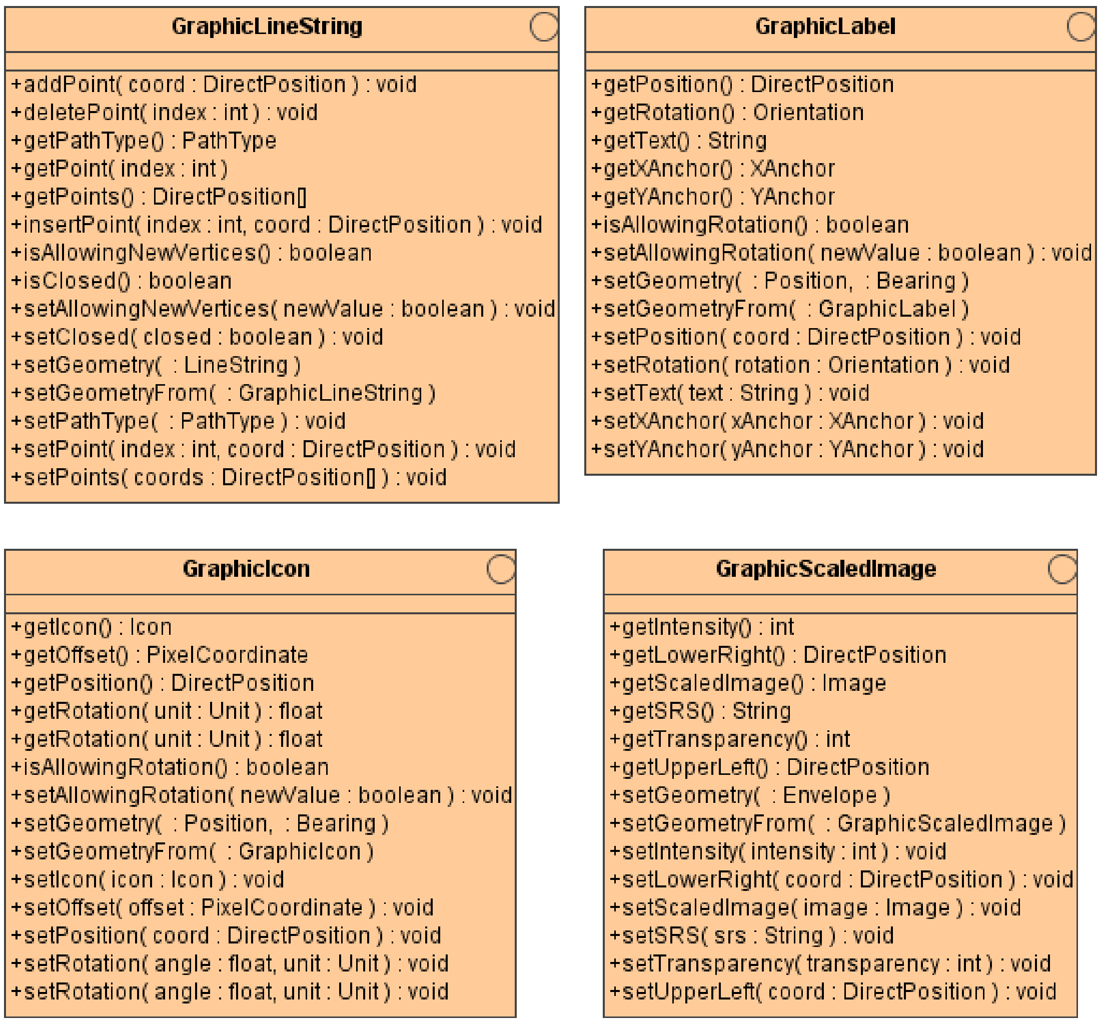

GraphicLineString
+addPoint(coord : DirectPosition) : void
+deletePoint(index : int) : void
+getPathType() : PathType
+getPoint(index : int)
+getPoints() : DirectPosition[]
+insertPoint(index : int, coord : DirectPosition) : void
+isAllowingNewVertices() : boolean
+isClosed() : boolean
+setAllowingNewVertices(newValue : boolean) : void
+setClosed(closed : boolean) : void
+setGeometry(: LineString)
+setGeometryFrom(: GraphicLineString)
+setPathType(: PathType) : void
+setPoint(index : int, coord : DirectPosition) : void
+setPoints(coords : DirectPosition[]) : void
GraphicLabel
+getPosition() : DirectPosition
+getRotation() : Orientation
+getText() : String
+getXAnchor() : XAnchor
+getYAnchor() : YAnchor
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean) : void
+setGeometry(: Position, : Bearing)
+setGeometryFrom(: GraphicLabel)
+setPosition(coord : DirectPosition) : void
+setRotation(rotation : Orientation) : void
+setText(text : String) : void
+setXAnchor(xAnchor : XAnchor) : void
+setYAnchor(yAnchor : YAnchor) : void
GraphicIcon
+getIcon() : Icon
+getOffset() : PixelCoordinate
+getPosition() : DirectPosition
+getRotation(unit : Unit) : float
+getRotation(unit : Unit) : float
+isAllowingRotation() : boolean
+setAllowingRotation(newValue : boolean) : void
+setGeometry(: Position, : Bearing)
+setGeometryFrom(: GraphicIcon)
+setIcon(icon : Icon) : void
+setOffset(offset : PixelCoordinate) : void
+setPosition(coord : DirectPosition) : void
+setRotation(angle : float, unit : Unit) : void
+setRotation(angle : float, unit : Unit) : void
GraphicScaledImage
+getIntensity() : int
+getLowerRight() : DirectPosition
+getScaledImage() : Image
+getSRS() : String
+getTransparency() : int
+getUpperLeft() : DirectPosition
+setGeometry(: Envelope)
+setGeometryFrom(: GraphicScaledImage)
+setIntensity(intensity : int) : void
+setLowerRight(coord : DirectPosition) : void
+setScaledImage(image : Image) : void
+setSRS(srs : String) : void
+setTransparency(transparency : int) : void
+setUpperLeft(coord : DirectPosition) : void

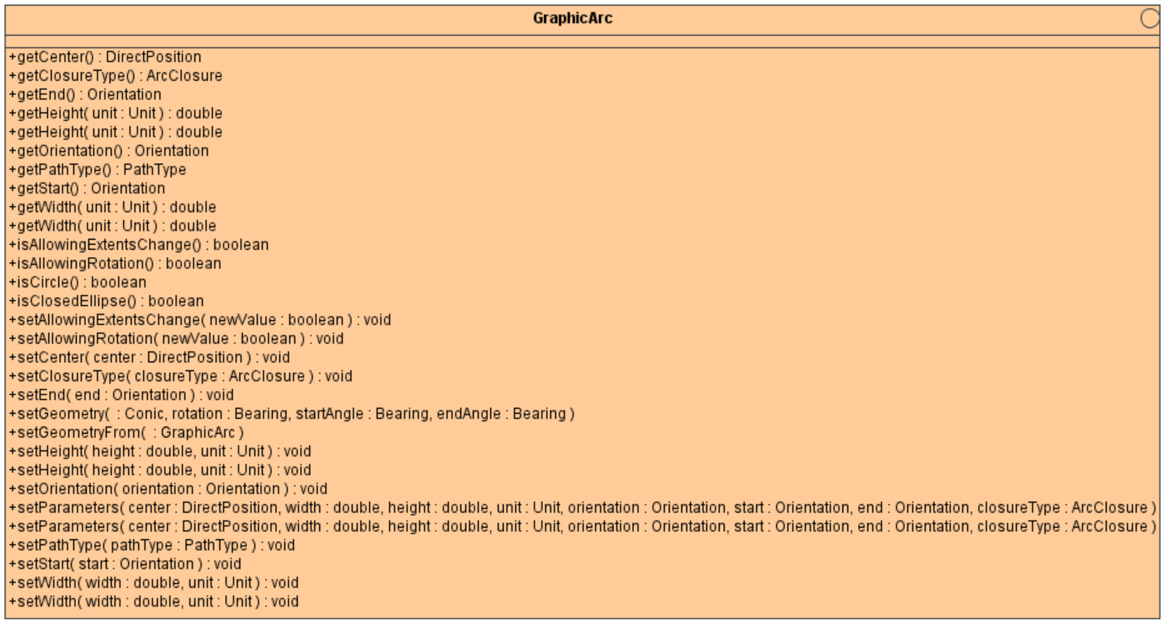
Figure 6 - Graphic Styles

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`. This class defines common abstractions for implementations of 1-dimensional lines made of one or more line segments, as well as closed polygons made of a closed set of three or more line segments. A settable attribute determines whether the linestring is closed.

`GraphicScaledImage` provides an abstraction for implementing projected images defined by an upper left and a lower right point. This class includes methods for setting the image transparency and intensity as well as the image data. There are also methods for setting and getting a string that specifies the Spatial Reference System (SRS) that is to create the image that this object represents. The format of this string is as specified in the OGC Web Map Server Interfaces Implementation Specification, Revision 1.1.0, section 6.5.5.

`GraphicIcon` defines a common abstraction for implementations that render icons on a drawing surface. The position of the icon is idealised as a single point attribute, and the location at which it is actually rendered is specified by another attribute as a pixel offset from the icon's upper left corner. Other attributes control the icon's rotation.



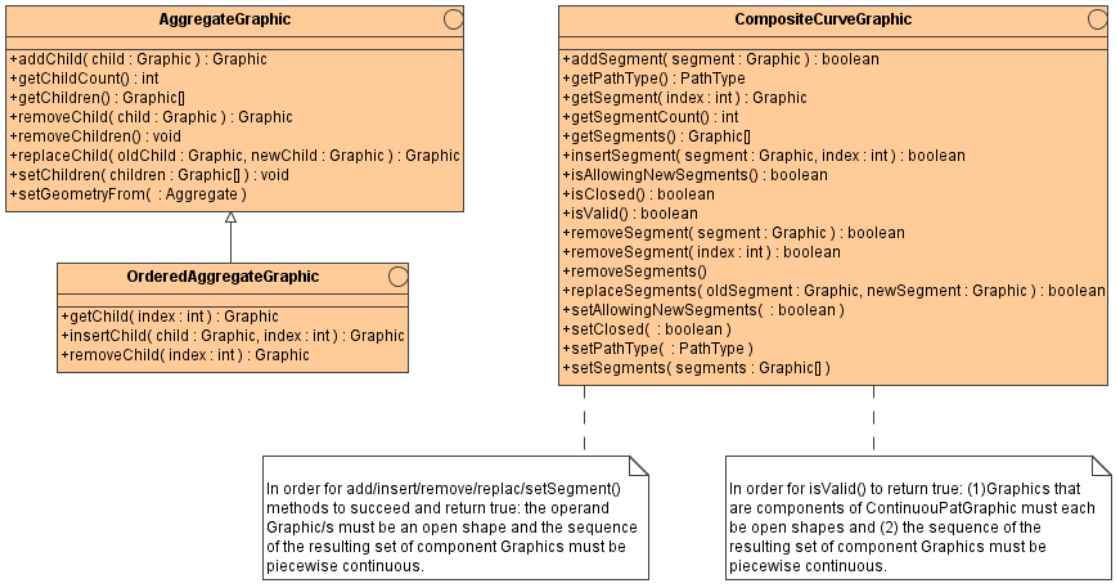
```
GraphicArc

+getCenter() : DirectPosition
+getClosureType() : ArcClosure
+getEnd() : Orientation
+getHeight( unit : Unit ) : double
+getHeight( unit : Unit ) : double
+getOrientation() : Orientation
+getPathType() : PathType
+getStart() : Orientation
+getWidth( unit : Unit ) : double
+getWidth( unit : Unit ) : double
+isAllowingExtentsChange() : boolean
+isAllowingRotation() : boolean
+isCircle() : boolean
+isClosedEllipse() : boolean
+setAllowingExtentsChange( newValue : boolean ) : void
+setAllowingRotation( newValue : boolean ) : void
+setCenter( center : DirectPosition ) : void
+setClosureType( closureType : ArcClosure ) : void
+setEnd( end : Orientation ) : void
+setGeometry( : Conic, rotation : Bearing, startAngle : Bearing, endAngle : Bearing )
+setGeometryFrom( : GraphicArc )
+setHeight( height : double, unit : Unit ) : void
+setHeight( height : double, unit : Unit ) : void
+setOrientation( orientation : Orientation ) : void
+setParameters( center : DirectPosition, width : double, height : double, unit : Unit, orientation : Orientation, start : Orientation, end : Orientation, closureType : ArcClosure )
+setParameters( center : DirectPosition, width : double, height : double, unit : Unit, orientation : Orientation, start : Orientation, end : Orientation, closureType : ArcClosure )
+setPathType( pathType : PathType ) : void
+setStart( start : Orientation ) : void
+setWidth( width : double, unit : Unit ) : void
+setWidth( width : double, unit : Unit ) : void
```

Figure 7 - Graphic Arc

`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.

6.2.1.3 Aggregates

This specification describes three kinds of aggregate graphical entities.


AggregateGraphic
+addChild(child : Graphic) : Graphic
+getChildCount() : int
+getChildren() : Graphic[]
+removeChild(child : Graphic) : Graphic
+removeChildren() : void
+replaceChild(oldChild : Graphic, newChild : Graphic) : Graphic
+setChildren(children : Graphic[]) : void
+setGeometryFrom(: Aggregate)
OrderedAggregateGraphic
+getChild(index : int) : Graphic
+insertChild(child : Graphic, index : int) : Graphic
+removeChild(index : int) : Graphic
CompositeCurveGraphic
+addSegment(segment : Graphic) : boolean
+getPathType() : PathType
+getSegment(index : int) : Graphic
+getSegmentCount() : int
+getSegments() : Graphic[]
+insertSegment(segment : Graphic, index : int) : boolean
+isAllowingNewSegments() : boolean
+isClosed() : boolean
+isValid() : boolean
+removeSegment(segment : Graphic) : boolean
+removeSegment(index : int) : boolean
+removeSegments()
+replaceSegments(oldSegment : Graphic, newSegment : Graphic) : boolean
+setAllowingNewSegments(: boolean)
+setClosed(: boolean)
+setPathType(: PathType)
+setSegments(segments : Graphic[])

In order for add/insert/remove/replac/setSegment() methods to succeed and return true: the operand Graphic/s must be an open shape and the sequence of the resulting set of component Graphics must be piecewise continuous.

In order for isValid() to return true: (1)Graphics that are components of ContinuouPatGraphic must each be open shapes and (2) the sequence of the resulting set of component Graphics must be piecewise continuous.

Figure 8 - Aggregate Graphics

`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 zeroeth element of the array is considered the frontmost (highest z-order) object and the `Graphic` in the largest element of the array is considered the bottommost (lowest z-order) object. Alternatively, the `Graphics` may be stored in a more efficient data structure.

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

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

`CompositeCurveGraphic` extends the `Graphic` class to accommodate the creation of open or closed continuous curves of arbitrary type. The aggregate is ordered, and the end point of each element in the aggregate is the beginning point of the next one. This interface includes methods to add, insert, remove, and replace component Graphics within this sequence.

`GraphicRing` is a graphical representation of a `Ring` geometry. `GraphicRing` can be explicitly composed of other `Graphic` objects (via the methods inherited from

`CompositeCurve`), or can get its geometry information directly from a `Ring` (via the `setGeometryFrom()` method).

`GraphicPolygon` is a graphical representation of a `Polygon` geometry. `GraphicPolygon` can be explicitly composed from `GraphicRing` objects, or can get its geometry information directly from a `Polygon` geometry. If a `GraphicPolygon` is generated by the aggregation of `GraphicRings`, it is left up to the implementation to ensure that the equivalent `Polygon` geometry is topologically correct (i.e. the constituent `Rings` do not touch or overlap).

6.2.1.4 Graphic Object Creation

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

6.2.1.5 Graphic and ISO Geometries

Each primitive `Graphic` class has a related primitive Geometry class. The virtual coordinate set of the `Graphic` class is generated by a direct transformation of the concrete Coordinate set of the corresponding `Geometry` class. Hence a GraphicArc object can be generated from a `GeometryArc` object, but not from a `GeometryPolyline` object.

The relationships between the `Graphic` objects and ISO `Geometry` objects are indicated in Table 1:

Graphic		**Geometry Equivalent (ISO 19107)**
member	**type**	**-**
GraphicIcon		Position plus Bearing
position	DirectPosition	Position
rotation	Orientation	Bearing
isAllowingRotation	boolean	n/a
icon	java.swing.Icon	n/a
offset	PixelCoordinate	n/a
GraphicLabel		Position plus Bearing
position	DirectPosition	Position
rotation	Orientation	Bearing

Graphic		**Geometry Equivalent (ISO 19107)**
member	**type**	**-**
isAllowingRotation	boolean	n/a
text	java.lang.String	n/a
x/y anchors	enumeration	n/a
GraphicLineString		LineString
points[]	DirectPosition[]	DirectPosition[] -> Position[] -> PointArray -> LineString
pathType	java.lang.String	interpolation
isAllowingNewVerteces	boolean	n/a
isClosed	boolean	n/a
GraphicScaledImage		Envelope
upperLeft	DirectPosition	Envelope.Position
lowerRight	DirectPosition	Envelope.Position
image	java.awt.Image	n/a
SRS	java.lang.String	OGC work item - See sections 6.3 and 5.2
transparency	int	n/a
intensity	int	n/a
GraphicArc		Conic as ellipse (eccen < 1) plus Three Angles
center	DirectPosition	Position - ((semiLR * eccen)/((1 - eccen^2)^0.5)), shift=false
width	double, Units	(2 * semiLR)/(1 - eccen^2)
height	double, Units	(2 * semiLR)/((1 - eccen^2)^0.5)
orientation	Orientation	Bearing
start	Orientation	Bearing
end	Orientation	Bearing
closureType	java.lang.String	LineString
pathType	java.lang.String	interpolation
isAllowingExtentsChange	boolean	n/a
isAllowingRotation	boolean	n/a

Table 1 - Relationship of Graphics to ISO Geometry

6.2.1.6 Path Type

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

- Global

- Unprojected
- Vector

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 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 surface of the earth.

6.2.1.7 Relative Coordinates

A "relative coordinate" is an absolute coordinate in a relative Coordinate Reference System (CRS). A "relative CRS" is a Derived CRS of type Engineering or Image. The derived CRS does not have a datum, but instead has an a conversion from another CRS. If this conversion is defined to include a non-null spatial translation and/or rotation, then the first CRS is relative to the second CRS.

Examples of relative coordinates from a Derived CRS of type Engineering are XY, XYZ, and RangeBearing. An example of a relative coordinate from a Derived CRS of type Image is a Pixel coordinate.

6.2.1.8 Relationship to GraphicStyle

`Graphics` may have a `GraphicStyle` object that allows them to be visually decorated. This relationship is discussed more fully under GraphicStyle, in Section 6.2.2.1, "Relationship to Graphic."

6.2.2 GraphicStyle

6.2.2.1 Relationship to Graphic.

The `GraphicStyle` class allows a `Graphic` to be visually decorated.

Each `Graphic` instance can have zero or one instances of `GraphicStyle`, but a `GraphicStyle` instance may be referenced by at most one `Graphic` instance.

If a `Graphic` does not have a reference to a `GraphicStyle`, then the `Graphic` uses the `GraphicStyle` information of either its parent `Graphic` (the nearest containing `Graphic` instance) or if it is not contained by another `Graphic`, the default `Canvas GraphicStyle` instance.

6.2.2.2 Relationship to OGC SLD

The taxonomy of the `GraphicStyle` classes has been developed to be as symmetric as possible with SLD. Ideally, the SLD `Symbolizers` would correlate to `Graphics`, however, since three `Graphic` classes (`GraphicLineSegment`, `GraphicArc`, and `CompositeCurveGraphic`) can be either topologically open or closed, there is no direct mapping. Therefore `GraphicStyle` extends all components referenced by the `Symbolizer` interfaces [ref `GraphicStyle` class diagram.] The `Symbolizer` interfaces are provided for illustration and as a reference point for future development of the GO-1 specification.

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

6.2.2.3 GraphicStyle superinterfaces

GraphicStyle extends all of the following interfaces: Stroke, Fill, Point, LabelPlacement, Halo, Font, Viewability, Editability, and Highlight.

* Stroke is closely related to SLD Stroke in that it decorates lines.

* Fill is closely related to SLD Fill in that it decorates polygonal areas.

* Point is closely related to the SLD Point Symbolizer in that it decorates icons.

* LabelPlacement is closely related to the SLD Text Symbolizer Line Placement component.

* Halo is closely related to the SLD Text Symbolizer Halo component.

* Font is closely related to the SLD Text Symbolizer Font component.

* Viewability allows a Graphic to be made unconditionally invisible, or conditionally invisible based on a range specified by maxScale and/or minScale. If maxScale is set, and the Canvas exceeds that scale, the Graphic is made invisible. Similarly if minScale is set and the Canvas drops below that scale, the Graphic is made invisible. This in/visibility does not change the transparency values of GraphicStyle components, but instead overrides their effect. The z-order hint is used by the Canvas to place the Graphic in the z-order [see the Canvas section].

* Editability allows the Graphic to be edited.

* Highlight controls whether a Graphic can blink, and if so, at what rate.

6.2.2.4 Graphic-to-GraphicStyle superinterface usage

The following Graphic classes are decorated by the listed superinterfaces of GraphicStyle:

* Graphic: Viewability, Editability (optional), Highlight (optional).

* GraphicLineString (opened), GraphicArc (opened): Stroke (and Fill, by reference).

* GraphicLineString (closed), GraphicArc (closed): Stroke, Fill.

* GraphicLabel: LabelPlacement, Halo, Font, Fill.

* GraphicIcon: Point (and both Stroke and Fill, by reference).

6.2.2.5 GraphicStyle inheritance

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

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

A `GraphicStyle` object inherits styling by calling `GraphicStyle.setInheritStyle(true)`. It will inherit styling from its aggregating object.

A `Graphic` can force its aggregated `Graphic` objects to inherit its style by setting `GraphicStyle.setOverrideAggregatedGraphics(true)`. This will force the aggregated Graphic objects to be rendered with the `GraphicStyle` of the aggregating `Graphic`, but will not change the `GraphicStyle` objects corresponding to each of the aggregated `Graphic` objects. This case will occur even if an aggregated `Graphic` has the setting `GraphicStyle.setInheritStyle(false)`.

6.3 Information Objects

Information objects are those that contain geometric, geolocation, attribute, and general-purpose styling information. They may appear both in the service space, i.e., on servers, encoded on the wire, within the data space of running applications, and also within and under the control of display systems where they may be used to provide geometric or styling information to `Graphic` objects.

This specification recognises that these two environments may have substantially different requirements. In a general geoprocessing environment, there is a need for great generality and robustness in the face of widely varying uses. Classes that conform to existing published OGC standards substantially meet these requirements. These are the topic of the present section.

Object definitions that meet the less extensive, but in some respects more demanding needs of a graphical display environment are presented in Section 6.2 above, Graphical Data Objects.

The material described in the present section is the focus of an ongoing, separate work item of the GO-1 Initiative. Most of the Java packages and interface suites discussed here were developed directly from formal UML models, using automated tools, as part of an assessment and demonstration of the Model Driven Architecture (MDA) approach for specification and interface development. This work will be reported more extensively in another document. The following sections provide a brief overview of the results of these efforts.

6.3.1 Geometry

GO-1 supports a “simple geometry” profile of the robust model for geospatial geometry developed and published by the International Standards Organisation as ISO document 19107. [6] The ISO model provides an implementable, international standard for geometry. This model has been implemented, with some minor changes, in the Open GIS Consortium Geographic Markup Language (GML) specification version 3.0. [11]

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 full 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 structure 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: *GO-1 Geometry object definitions described here were developed directly from the ISO 19107 UML models in a GO-1 work item intended to provide a demonstration and assessment of the Model Driven Architecture (MDA) approach and tools for specification and interface development. The results of this activity are reported in more detail in a separate document.*

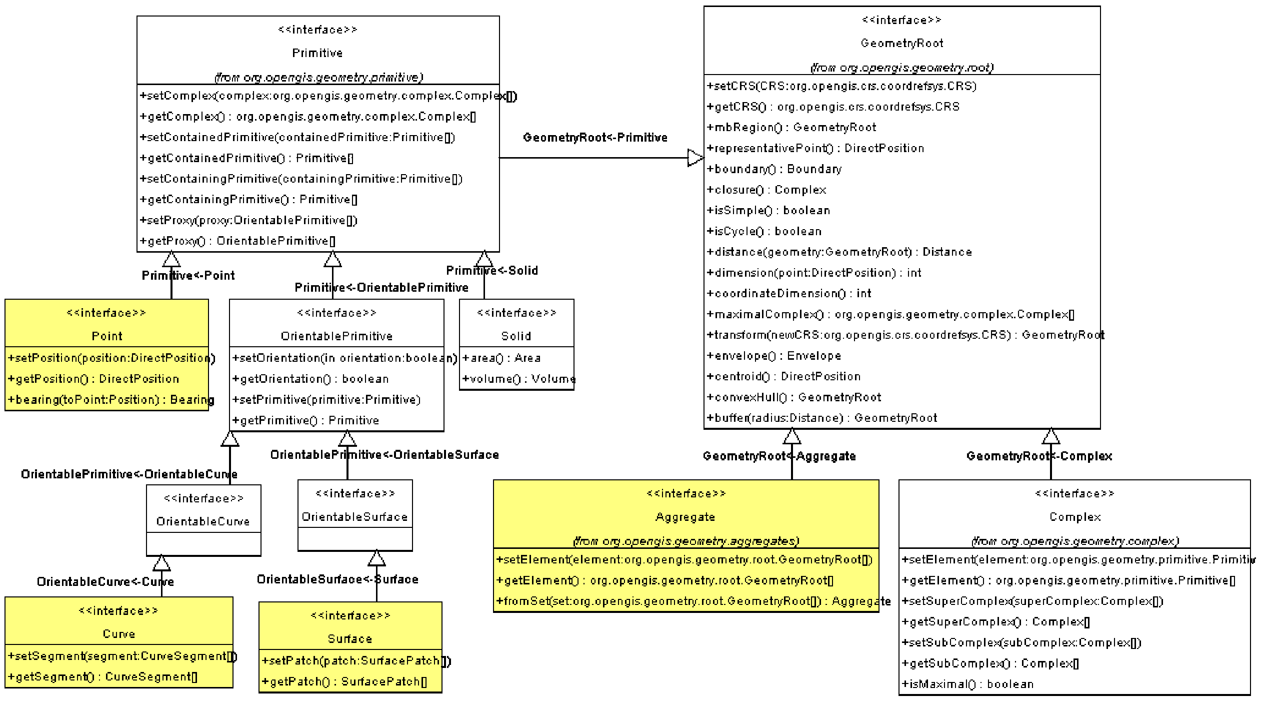

<<interface>>
Primitive
(from org.opengis.geometry.primitive)
+setComplex(complex:org.opengis.geometry.complex.Complex[])
+getComplex() : org.opengis.geometry.complex.Complex[]
+setContainedPrimitive(containedPrimitive:Primitive[])
+getContainedPrimitive() : Primitive[]
+setContainingPrimitive(containingPrimitive:Primitive[])
+getContainingPrimitive() : Primitive[]
+setProxy(proxy:OrientablePrimitive[])
+getProxy() : OrientablePrimitive[]
GeometryRoot<-Primitive
<<interface>>
GeometryRoot
(from org.opengis.geometry.root)
+setCRS(CRS:org.opengis.crs.coordrefsys.CRS)
+getCRS() : org.opengis.crs.coordrefsys.CRS
+mbRegion() : GeometryRoot
+representativePoint() : DirectPosition
+boundary() : Boundary
+closure() : Complex
+isSimple() : boolean
+isCycle() : boolean
+distance(geometry:GeometryRoot) : Distance
+dimension(point:DirectPosition) : int
+coordinateDimension() : int
+maximalComplex() : org.opengis.geometry.complex.Complex[]
+transform(newCRS:org.opengis.crs.coordrefsys.CRS) : GeometryRoot
+envelope() : Envelope
+centroid() : DirectPosition
+convexHull() : GeometryRoot
+buffer(radius:Distance) : GeometryRoot
Primitive<-Point
Primitive<-OrientablePrimitive
Primitive<-Solid
<<interface>>
Point
+setPosition(position:DirectPosition)
+getPosition() : DirectPosition
+bearing(toPoint:Position) : Bearing
<<interface>>
OrientablePrimitive
+setOrientation(in orientation:boolean)
+getOrientation() : boolean
+setPrimitive(primitive:Primitive)
+getPrimitive() : Primitive
<<interface>>
Solid
+area() : Area
+volume() : Volume
OrientablePrimitive<-OrientableCurve
OrientablePrimitive<-OrientableSurface
<<interface>>
OrientableCurve
<<interface>>
OrientableSurface
OrientableCurve<-Curve
OrientableSurface<-Surface
<<interface>>
Curve
+setSegment(segment:CurveSegment[])
+getSegment() : CurveSegment[]
<<interface>>
Surface
+setPatch(patch:SurfacePatch[])
+getPatch() : SurfacePatch[]
GeometryRoot<-Aggregate
GeometryRoot<-Complex
<<interface>>
Aggregate
(from org.opengis.geometry.aggregates)
+setElement(element:org.opengis.geometry.root.GeometryRoot[])
+getElement() : org.opengis.geometry.root.GeometryRoot[]
+fromSet(set:org.opengis.geometry.root.GeometryRoot[]) : Aggregate
<<interface>>
Complex
(from org.opengis.geometry.complex)
+setElement(element:org.opengis.geometry.primitive.Primitive
+getElement() : org.opengis.geometry.primitive.Primitive[]
+setSuperComplex(superComplex:Complex[])
+getSuperComplex() : Complex[]
+setSubComplex(subComplex:Complex[])
+getSubComplex() : Complex[]
+isMaximal() : boolean

Figure 9 - Top-Level classes of GO-1 Geometry

Figure 9 depicts the top-level Java interfaces of the GO-1 Geometry model. These Java interfaces are auto-generated directly from the ISO 19107 Geometry models. The highlighted interfaces (Point, Curve, Surface and Aggregate) are the top-level interfaces for the key geometries that are the focus of GO-1: Point, LineString, Polygon and aggregates of these. These interfaces are briefly described below.

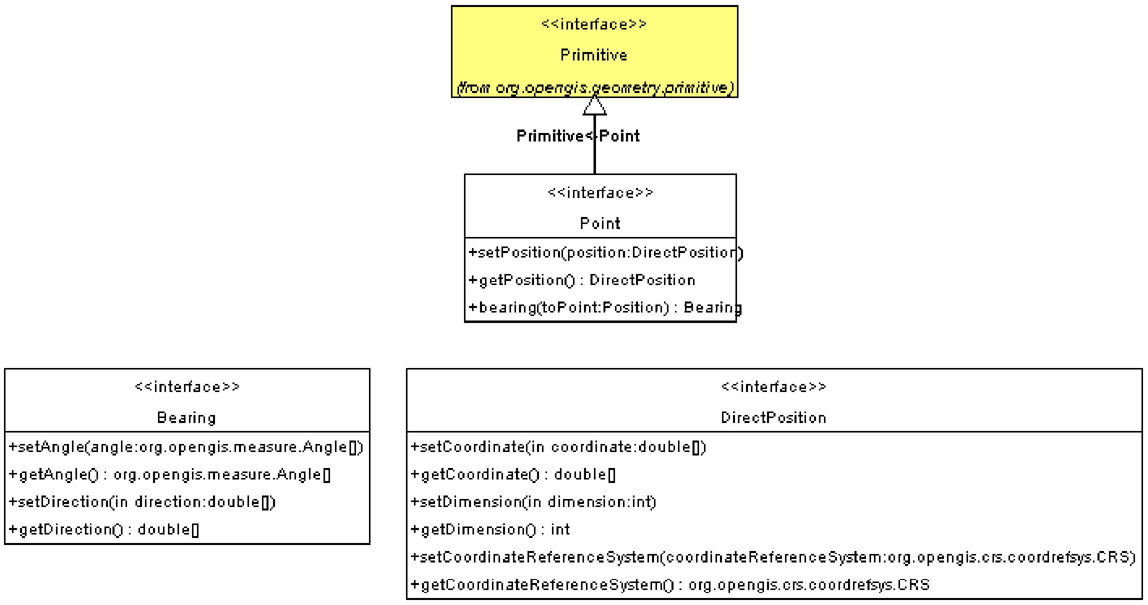
`GeometryRoot` is the root class of the geometric object taxonomy and supports interfaces common to all geographically referenced geometric objects. `GeometryRoot` instances are sets of direct positions in a particular coordinate reference system. A `GeometryRoot` can be regarded as an infinite set of 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 geometric primitive (`Primitive`) is a geometric object that is not decomposed further into other primitives in the system. This includes curves and surfaces, even though they are composed of curve segments and surface patches, respectively. This composition is a strong aggregation: curve segments and surface patches cannot exist outside the context of a primitive.

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

6.3.1.1 Point

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



<<interface>>
Primitive
(from org.opengis.geometry.primitive)
Primitive<-Point
<<interface>>
Point
+setPosition(position:DirectPosition)
+getPosition() : DirectPosition
+bearing(toPoint:Position) : Bearing
<<interface>>
Bearing
+setAngle(angle:org.opengis.measure.Angle[])
+getAngle() : org.opengis.measure.Angle[]
+setDirection(in direction:double[])
+getDirection() : double[]
<<interface>>
DirectPosition
+setCoordinate(in coordinate:double[])
+getCoordinate() : double[]
+setDimension(in dimension:int)
+getDimension() : int
+setCoordinateReferenceSystem(coordinateReferenceSystem:org.opengis.crs.coordrefsys.CRS)
+getCoordinateReferenceSystem() : org.opengis.crs.coordrefsys.CRS

Figure 10 - Point Primitive

`DirectPosition` object data types hold the coordinates for a position within some coordinate reference system. The coordinate reference system is described in `org.opengis.crs.crsrefsystem.CRS` (from ISO19111::SC_CRS). Since `DirectPositions`, as data types, will often be included in larger objects (such as `GeometryRoot`) that have references to ISO19111::SC_CRS, the `DirectPosition::cordinateReferenceSystem` may be left NULL if this particular `DirectPosition` is included in a larger object with such a reference to a SC_CRS. In this case, the DirectPosition::cordinateReferenceSystem is implicitly assumed to take on the value of the containing object's SC_CRS.

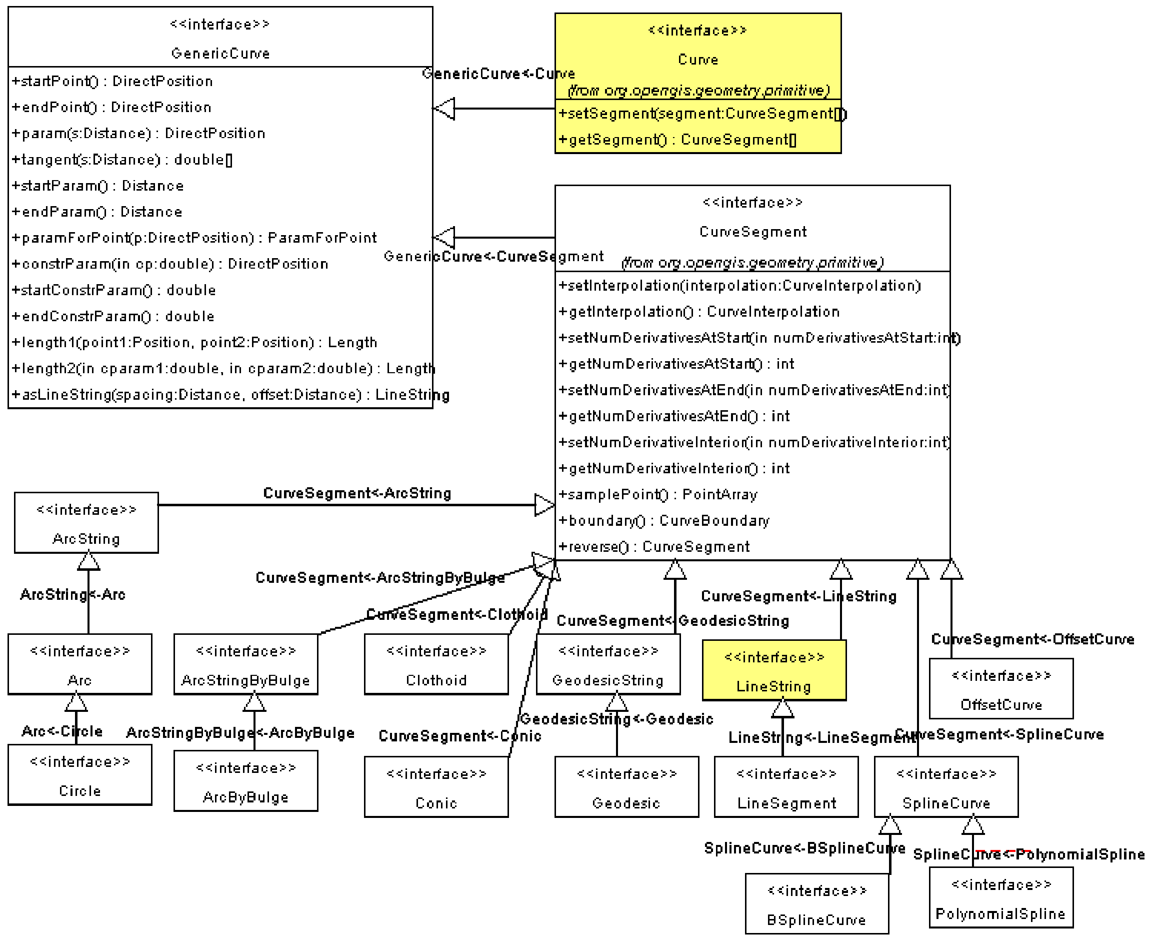
`Bearing` is a data type used to represent direction in the coordinate reference system. In a 2D coordinate reference system, this can be accomplished using a "angle measured from true north" or a 2D vector point in that direction. In a 3D coordinate reference system, two angles or any 3D vector is possible.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

6.3.1.2 Curve

As shown in Figure 9, `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 curve segment within a curve may be defined using a different interpolation method. The curve segments 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.



<<interface>>
GenericCurve
+startPoint() : DirectPosition
+endPoint() : DirectPosition
+param(s:Distance) : DirectPosition
+tangent(s:Distance) : double[]
+startParam() : Distance
+endParam() : Distance
+paramForPoint(p:DirectPosition) : ParamForPoint
+constrParam(in cp:double) : DirectPosition
+startConstrParam() : double
+endConstrParam() : double
+length1(point1:Position, point2:Position) : Length
+length2(in cparam1:double, in cparam2:double) : Length
+asLineString(spacing:Distance, offset:Distance) : LineString
GenericCurve<-Curve
<<interface>>
Curve
(from org.opengis.geometry.primitive)
+setSegment(segment:CurveSegment[])
+getSegment() : CurveSegment[]
GenericCurve<-CurveSegment
<<interface>>
CurveSegment
(from org.opengis.geometry.primitive)
+setInterpolation(interpolation:CurveInterpolation)
+getInterpolation() : CurveInterpolation
+setNumDerivativesAtStart(in numDerivativesAtStart:int)
+getNumDerivativesAtStart() : int
+setNumDerivativesAtEnd(in numDerivativesAtEnd:int)
+getNumDerivativesAtEnd() : int
+setNumDerivativeInterior(in numDerivativeInterior:int)
+getNumDerivativeInterior() : int
+samplePoint() : PointArray
+boundary() : CurveBoundary
+reverse() : CurveSegment
CurveSegment<-ArcString
<<interface>>
ArcString
ArcString<-Arc
CurveSegment<-ArcStringByBulge
CurveSegment<-Clothoid
CurveSegment<-GeodesicString
CurveSegment<-LineString
CurveSegment<-OffsetCurve
<<interface>>
Arc
<<interface>>
ArcStringByBulge
<<interface>>
Clothoid
<<interface>>
GeodesicString
<<interface>>
LineString
<<interface>>
OffsetCurve
Arc<-Circle
ArcStringByBulge<-ArcByBulge
CurveSegment<-Conic
GeodesicString<-Geodesic
LineString<-LineSegment
CurveSegment<-SplineCurve
<<interface>>
Circle
<<interface>>
ArcByBulge
<<interface>>
Conic
<<interface>>
Geodesic
<<interface>>
LineSegment
<<interface>>
SplineCurve
SplineCurve<-BSplineCurve
SplineCurve<-PolynomialSpline
<<interface>>
BSplineCurve
<<interface>>
PolynomialSpline

Figure 11 - Curve and CurveSegment

6.3.1.2.1 LineString

A `LineString` consists of sequence of line segments, each having a parameterization like the one for `LineSegment`. The class essentially combines a `Sequence<LineSegments>` into a single object, with the obvious savings of storage space.

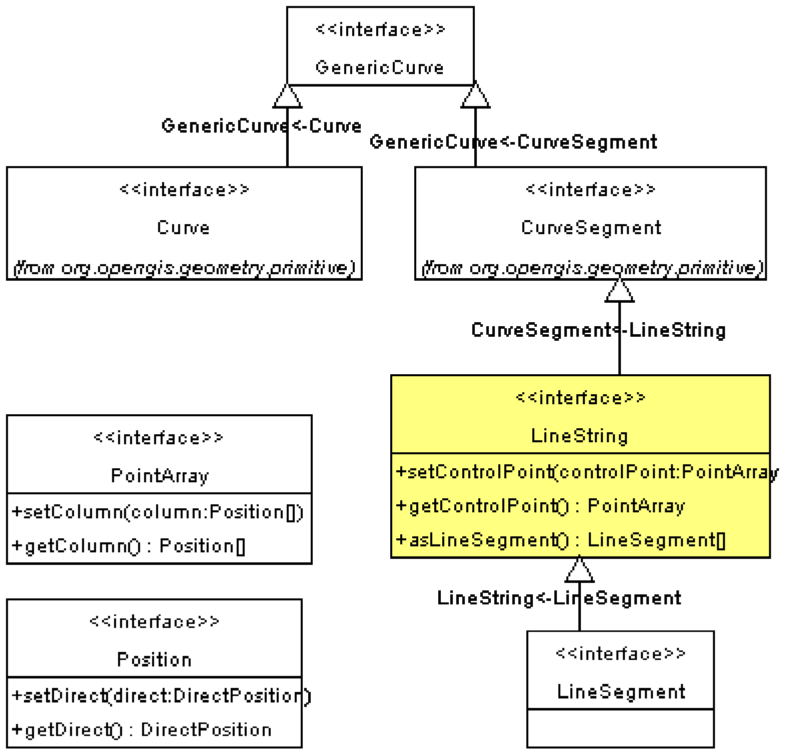

<<interface>>
GenericCurve
GenericCurve<-Curve
GenericCurve<-CurveSegment
<<interface>>
Curve
(from org.opengis.geometry.primitive)
<<interface>>
CurveSegment
(from org.opengis.geometry.primitive)
CurveSegment<-LineString
<<interface>>
LineString
+setControlPoint(controlPoint:PointArray)
+getControlPoint() : PointArray
+asLineSegment() : LineSegment[]
<<interface>>
PointArray
+setColumn(column:Position[])
+getColumn() : Position[]
<<interface>>
Position
+setDirect(direct:DirectPosition)
+getDirect() : DirectPosition
LineString<-LineSegment
<<interface>>
LineSegment

Figure 12 - LineString

A LineSegment consists of two distinct DirectPositions (the startPoint and endPoint) joined by a straight line. Thus its interpolation attribute shall be "linear".

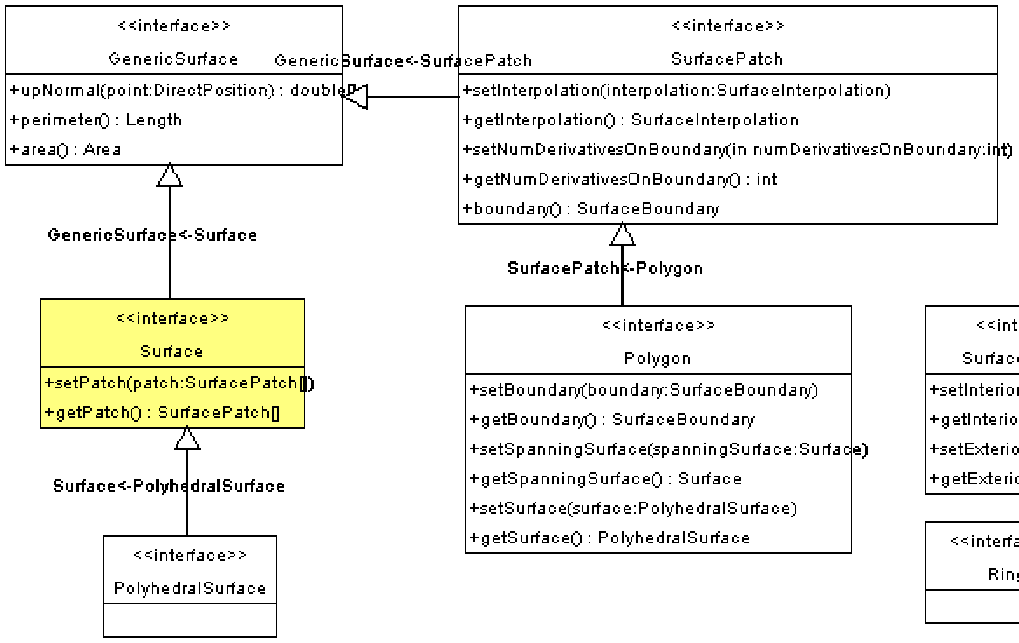
Any other point in the controlPoint array must fall on this line. The control points of a LineSegment shall all lie on the straight line between its start point and end point. Between these two points, other positions may be interpolated linearly.

6.3.1.3 Polygon

A Polygon is a surface patch that is defined by a set of boundary curves and an underlying surface to which these curves adhere. The default is that the curves are coplanar and the polygon uses planar interpolation in its interior.

A SurfacePatch defines a homogeneous portion of a Surface. The multiplicity of the association "Segmentation" 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.



<<interface>>
GenericSurface
+upNormal(point:DirectPosition) : double[]
+perimeter() : Length
+area() : Area
GenericSurface<-SurfacePatch
<<interface>>
SurfacePatch
+setInterpolation(interpolation:SurfaceInterpolation)
+getInterpolation() : SurfaceInterpolation
+setNumDerivativesOnBoundary(in numDerivativesOnBoundary:int)
+getNumDerivativesOnBoundary() : int
+boundary() : SurfaceBoundary
GenericSurface<-Surface
SurfacePatch<-Polygon
<<interface>>
Surface
+setPatch(patch:SurfacePatch[])
+getPatch() : SurfacePatch[]
<<interface>>
Polygon
+setBoundary(boundary:SurfaceBoundary)
+getBoundary() : SurfaceBoundary
+setSpanningSurface(spanningSurface:Surface)
+getSpanningSurface() : Surface
+setSurface(surface:PolyhedralSurface)
+getSurface() : PolyhedralSurface
<<interface>>
SurfaceBoundary
+setInterior(interior:Ring[])
+getInterior() : Ring[]
+setExterior(exterior:Ring)
+getExterior() : Ring
Surface<-PolyhedralSurface
<<interface>>
PolyhedralSurface
<<interface>>
Ring

Figure 13 - Polygon

`SurfaceBoundary` represents the boundary of `Surfaces` (i.e., an exterior and zero or more interior rings).

6.3.1.4 Ring

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 each ring is simple.

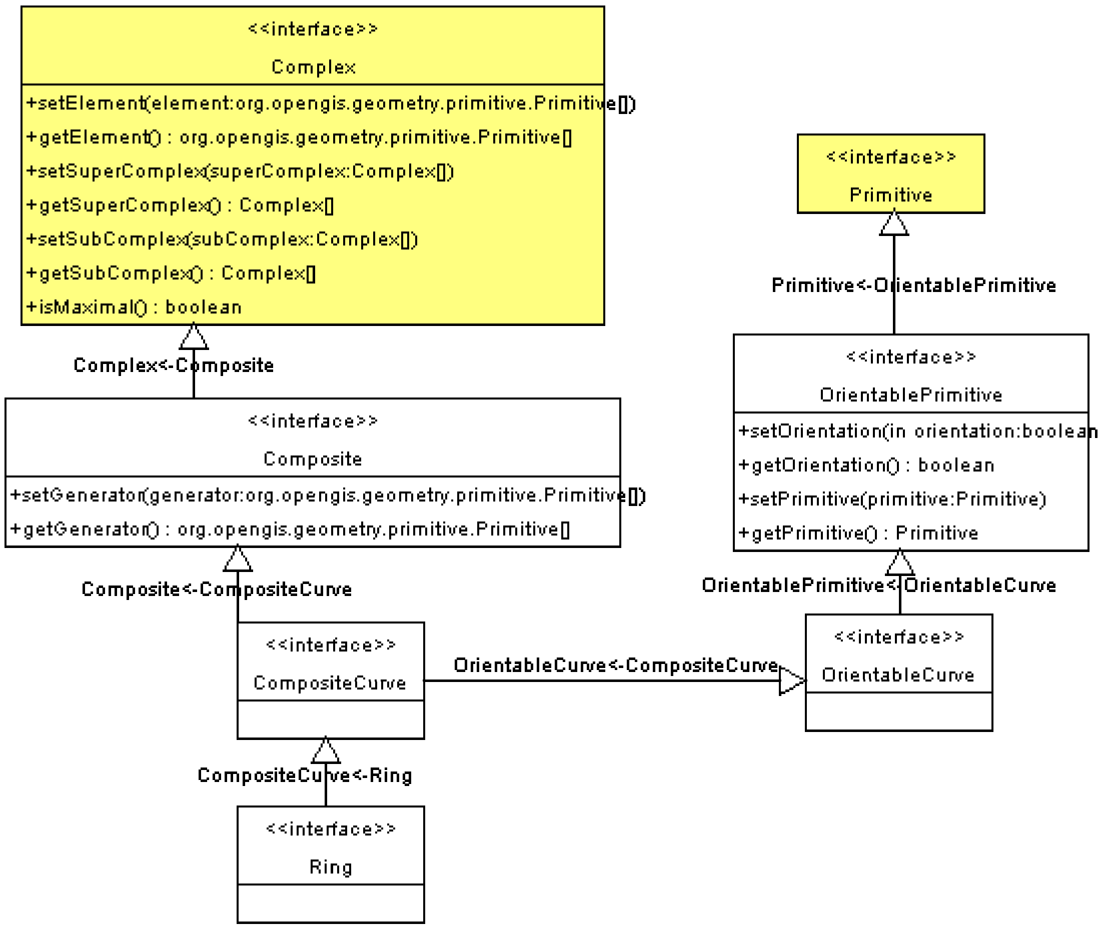

<<interface>>
Complex
+setElement(element:org.opengis.geometry.primitive.Primitive[])
+getElement() : org.opengis.geometry.primitive.Primitive[]
+setSuperComplex(superComplex:Complex[])
+getSuperComplex() : Complex[]
+setSubComplex(subComplex:Complex[])
+getSubComplex() : Complex[]
+isMaximal() : boolean
Complex<-Composite
<<interface>>
Composite
+setGenerator(generator:org.opengis.geometry.primitive.Primitive[])
+getGenerator() : org.opengis.geometry.primitive.Primitive[]
Composite<-CompositeCurve
<<interface>>
CompositeCurve
CompositeCurve<-Ring
<<interface>>
Ring
OrientableCurve<-CompositeCurve
<<interface>>
Primitive
Primitive<-OrientablePrimitive
<<interface>>
OrientablePrimitive
+setOrientation(in orientation:boolean
+getOrientation() : boolean
+setPrimitive(primitive:Primitive)
+getPrimitive() : Primitive
OrientablePrimitive<-OrientableCurve
<<interface>>
OrientableCurve

Figure 14 - Ring.

NOTE: Even though each `Ring` is 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.

6.3.1.5 Aggregates

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 features that use multiple geometric objects in their representations, such as a collection of points to represent a tank farm or orchard.

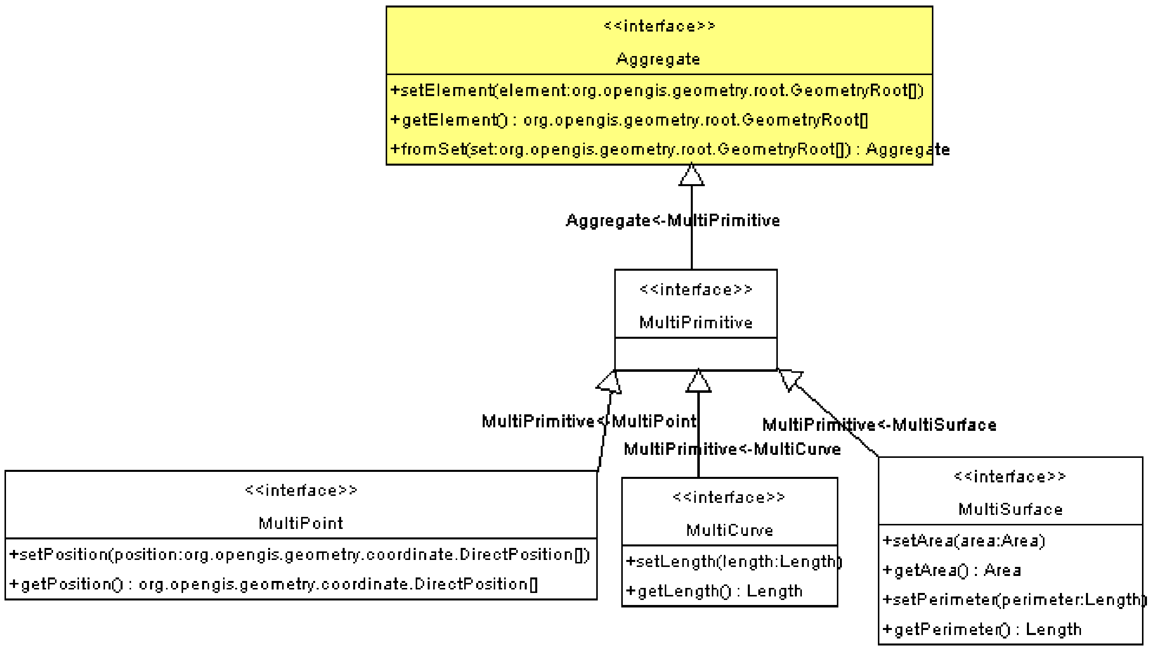

<<interface>>
Aggregate
+setElement(element:org.opengis.geometry.root.GeometryRoot[])
+getElement() : org.opengis.geometry.root.GeometryRoot[]
+fromSet(set:org.opengis.geometry.root.GeometryRoot[]) : Aggregate
Aggregate<-MultiPrimitive
<<interface>>
MultiPrimitive
MultiPrimitive<-MultiPoint
MultiPrimitive<-MultiCurve
MultiPrimitive<-MultiSurface
<<interface>>
MultiPoint
+setPosition(position:org.opengis.geometry.coordinate.DirectPosition[])
+getPosition() : org.opengis.geometry.coordinate.DirectPosition[]
<<interface>>
MultiCurve
+setLength(length:Length)
+getLength() : Length
<<interface>>
MultiSurface
+setArea(area:Area)
+getArea() : Area
+setPerimeter(perimeter:Length)
+getPerimeter() : Length

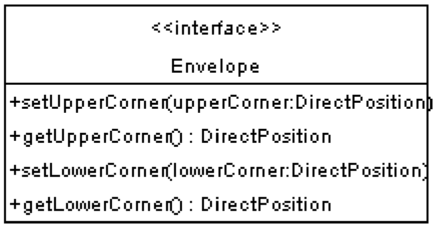
Figure 15 - Aggregates

The aggregates, Aggregate, gather geometric objects. Since they 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 are contained in features, and cannot be held in collections that are strong aggregations. For this reason, the collections described in this clause are all weak aggregations, and shall use references to include geometric objects.

6.3.1.6 Envelope

`Envelope` is often referred to as a minimum bounding box or rectangle. Regardless of dimension, a `Envelope` can be represented without ambiguity as two direct positions (coordinate points). To encode a `Envelope`, it is sufficient to encode these two points. This is consistent with all of the data types in this standard, their state is represented by their publicly accessible attributes.


<<interface>>
Envelope
+setUpperCorner(upperCorner:DirectPosition)
+getUpperCorner() : DirectPosition
+setLowerCorner(lowerCorner:DirectPosition)
+getLowerCorner() : DirectPosition

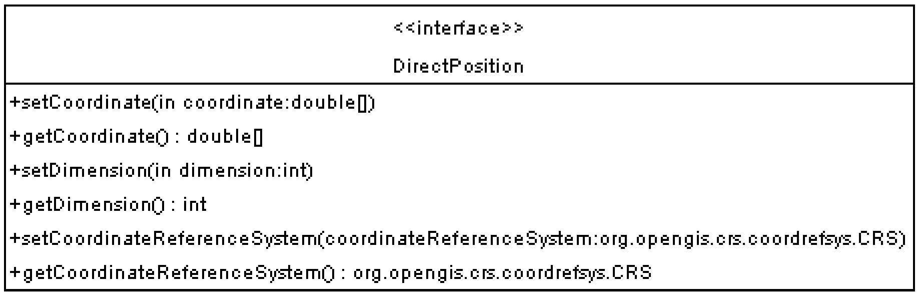

<<interface>>
DirectPosition
+setCoordinate(in coordinate:double[])
+getCoordinate() : double[]
+setDimension(in dimension:int)
+getDimension() : int
+setCoordinateReferenceSystem(coordinateReferenceSystem:org.opengis.crs.coordrefsys.CRS)
+getCoordinateReferenceSystem() : org.opengis.crs.coordrefsys.CRS

Figure 16 - Envelope

6.3.1.7 Geometry Factory

GO-1 defines a `GeometryFactory` interface that extends the ISO Geometry model. `GeometryFactory` defines constructors for the simple geometry types most commonly required: `Point`, `LineString` and `Polygon` (and their constituents).

In addition, there is a generic constructor for materialising geometry instances from GML documents.

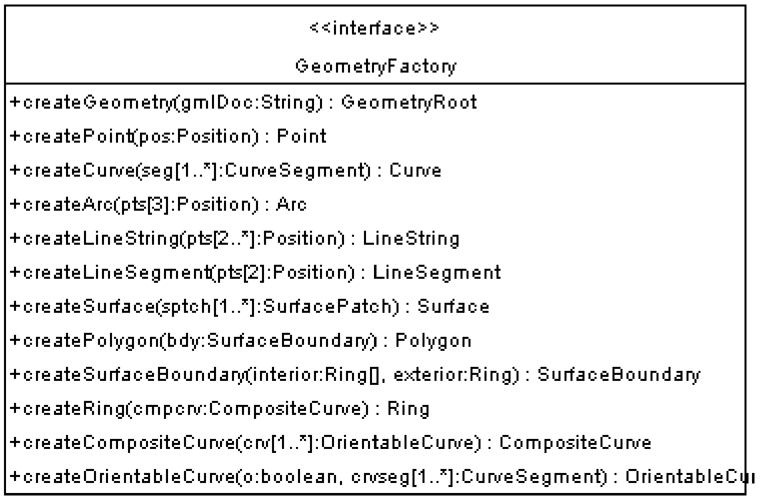

<<interface>>
GeometryFactory
+createGeometry(gmlDoc:String) : GeometryRoot
+createPoint(pos:Position) : Point
+createCurve(seg[1..*]:CurveSegment) : Curve
+createArc(pts[3]:Position) : Arc
+createLineString(pts[2..*]:Position) : LineString
+createLineSegment(pts[2]:Position) : LineSegment
+createSurface(sptch[1..*]:SurfacePatch) : Surface
+createPolygon(bdy:SurfaceBoundary) : Polygon
+createSurfaceBoundary(interior:Ring[], exterior:Ring) : SurfaceBoundary
+createRing(cmpcrv:CompositeCurve) : Ring
+createCompositeCurve(crv[1..*]:OrientableCurve) : CompositeCurve
+createOrientableCurve(o:boolean, crvseg[1..*]:CurveSegment) : OrientableCu

Figure 17 - GeometryFactory

6.3.2 Feature

A geographic feature is a meaningful object in the selected domain of discourse such as a Road, River, Person, Vehicle or Administrative Boundary. This follows the general definition of a feature given in ISO 19109 and the OGC Abstract Specification Topic 5. [7]

The `Feature` and `FeatureType` interfaces work together to represent a geographic feature of arbitrary complexity. These interfaces serve two important purposes:

1. They give client applications a unified, consistent framework for accessing and manipulating feature data.

2. They give implementers a framework for constraining and enforcing constraints (respectively) on allowed feature types. As such, this interface is as general as possible in terms of the types of objects to which it provides access. For the vast majority of feature schemas, this generic feature model will work fine.

The `Feature` interface combines a "default" GO-1 Geometry object with additional attributes in the form of potentially nested key-value pairs (`FeatureAttributes`) that characterise an instance of a geographic feature. For example, a road class may consist of a LineString geometry with attributes for name, route number, number of lanes, and surface type.

`FeatureType` is a metadata template for a feature of arbitrary complexity. Objects implementing the `FeatureType` metaclass interface get instantiated as classes that represent individual feature types. A certain feature type is the class for all instances of that feature type. The instances of a class that represents an individual feature type are feature instances. Feature types are equivalent to *classes* and feature instances are equivalent to *objects* in object oriented modeling.

`FeatureAttribute` is a characteristic property of a feature and has a name, a data type, and a value domain associated with it. A feature attribute for a feature instance also has an attribute value taken from the value domain.

`FeatureAttributeType` is a metadata template for a single attribute object. Objects implementing the `FeatureAttributeType` metaclass interface get instantiated as classes that represent individual feature attribute types. `FeatureAttributeType` objects store metadata about `FeatureAttributes` such as:

- description – a human-readable description of the attribute
- multiplicity – the number of instances of this attribute per Feature.
- type – the expected class of this attribute.

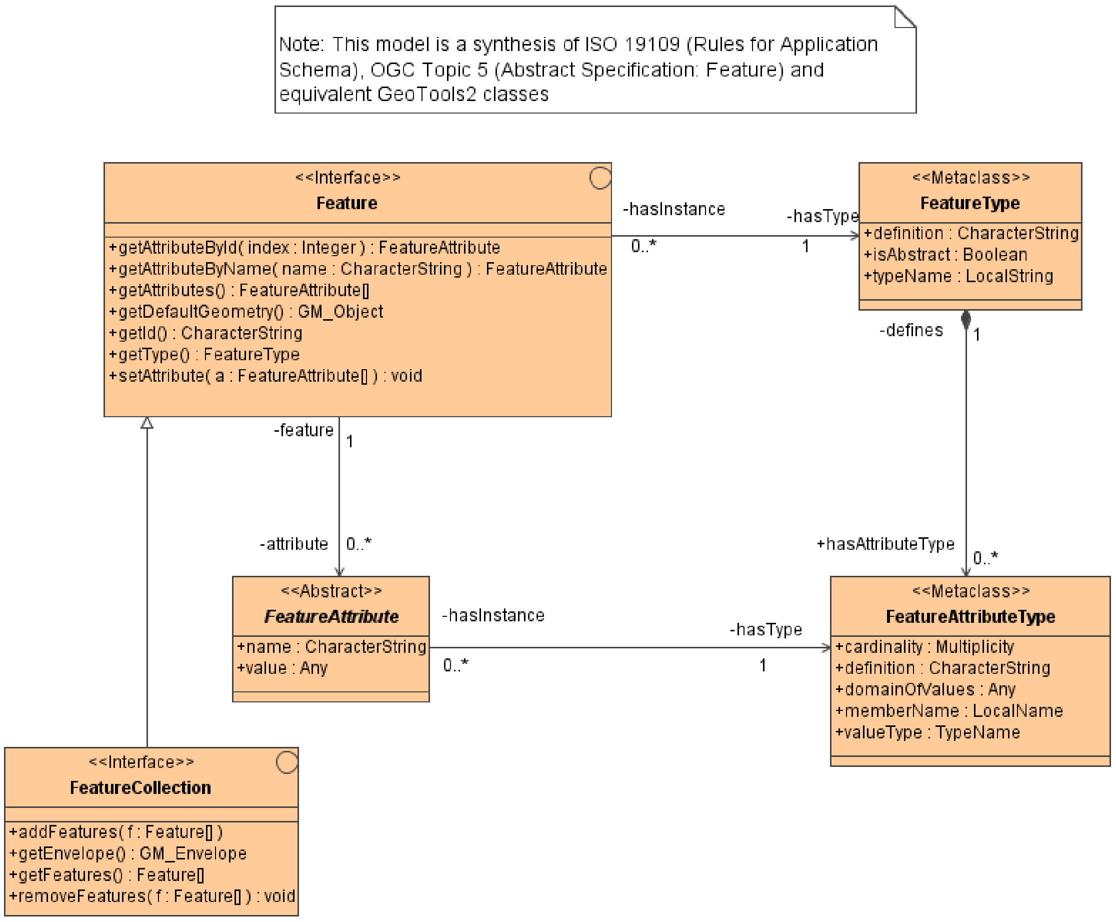

Note: This model is a synthesis of ISO 19109 (Rules for Application Schema), OGC Topic 5 (Abstract Specification: Feature) and equivalent GeoTools2 classes
<<Interface>>
Feature
+getAttributeById(index : Integer) : FeatureAttribute
+getAttributeByName(name : CharacterString) : FeatureAttribute
+getAttributes() : FeatureAttribute[]
+getDefaultGeometry() : GM_Object
+getId() : CharacterString
+getType() : FeatureType
+setAttribute(a : FeatureAttribute[]) : void
-hasInstance
0..*
-hasType
1
<<Metaclass>>
FeatureType
+definition : CharacterString
+isAbstract : Boolean
+typeName : LocalString
-defines
1
-feature
1
-attribute
0..*
+hasAttributeType
0..*
<<Abstract>>
FeatureAttribute
+name : CharacterString
+value : Any
-hasInstance
0..*
-hasType
1
<<Metaclass>>
FeatureAttributeType
+cardinality : Multiplicity
+definition : CharacterString
+domainOfValues : Any
+memberName : LocalName
+valueType : TypeName
<<Interface>>
FeatureCollection
+addFeatures(f : Feature[])
+getEnvelope() : GM_Envelope
+getFeatures() : Feature[]
+removeFeatures(f : Feature[]) : void

Figure 18 - Feature Model.

A `FeatureCollection` is a collection of feature instances that can behave as a feature. The `FeatureCollection` interface is used to aggregate one or more `Feature` objects. A town, for example, could be a `FeatureCollection` containing road, river, railroad, and building Features.

6.3.2.1 Feature Factory

`FeatureFactory` provides support for creation of a `Feature` instance of a given `FeatureType`. `Feature` instances can be constructed from an array of `FeatureAttributes` (which may or may not include a Geometry-type attribute), a GML (XML) document, or with a Geometry object explicitly specified. In all cases, the type of feature must be specified.

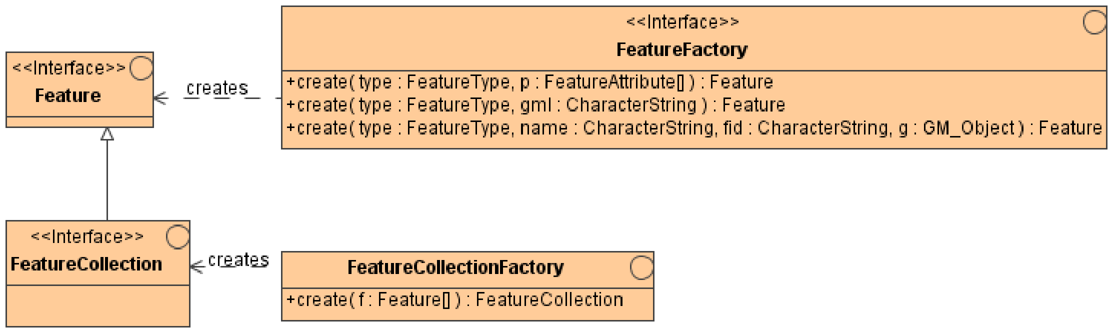

<<Interface>>
Feature
creates
<<Interface>>
FeatureFactory
+create(type : FeatureType, p : FeatureAttribute[]) : Feature
+create(type : FeatureType, gml : CharacterString) : Feature
+create(type : FeatureType, name : CharacterString, fid : CharacterString, g : GM_Object) : Feature
<<Interface>>
FeatureCollection
creates
FeatureCollectionFactory
+create(f : Feature[]) : FeatureCollection

Figure 19 - FeatureFactory Model

`FeatureCollectionFactory` provides support for creation of `FeatureCollection` instances from an array of `Feature` instances.

6.3.3 OGC Styled Layer Descriptor

GO-1 supports and incorporates the OGC Styled Layer Descriptor (SLD) as the conveyance for styling information for Geometry and OGC Feature objects.

In a GO-1 application environment, the SLD may be used as a source for styling data to be used in the GraphicStyle of a Graphics object for ultimate management or rendering by a Canvas.

6.3.4 Coordinate Reference System

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 Information Object 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 Information Object classes, CRS objects are instantiated by a factory that hides the details of object creation from client applications or libraries.

Note: except where noted, all descriptive text accompanying the context diagrams below is taken directly from [26].

6.3.4.1 Reference System

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

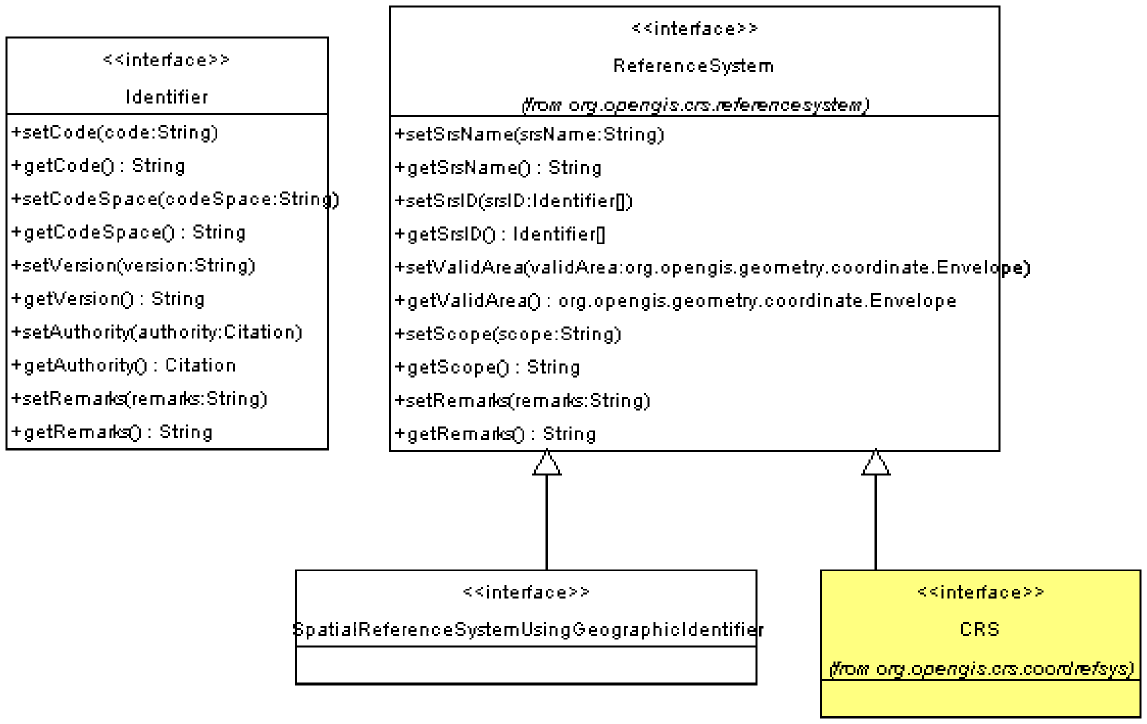

<<interface>>
Identifier
+setCode(code:String)
+getCode() : String
+setCodeSpace(codeSpace:String)
+getCodeSpace() : String
+setVersion(version:String)
+getVersion() : String
+setAuthority(authority:Citation)
+getAuthority() : Citation
+setRemarks(remarks:String)
+getRemarks() : String
<<interface>>
ReferenceSystem
(from org.opengis.crs.referencesystem)
+setSrsName(srsName:String)
+getSrsName() : String
+setSrsID(srsID:Identifier[])
+getSrsID() : Identifier[]
+setValidArea(validArea:org.opengis.geometry.coordinate.Envelope)
+getValidArea() : org.opengis.geometry.coordinate.Envelope
+setScope(scope:String)
+getScope() : String
+setRemarks(remarks:String)
+getRemarks() : String
<<interface>>
SpatialReferenceSystemUsingGeographicIdentifier
<<interface>>
CRS
(from org.opengis.crs.coordrefsys)

Figure 20 - Reference System

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

6.3.4.2 Coordinate Reference System

`CoordinateReferenceSystem` consists of an ordered sequence of coordinate system axes that are related to the earth through a datum. A coordinate reference system is defined by one datum and by one coordinate system. Most coordinate reference system do not move relative to the earth, except for engineering coordinate reference systems (`EngineeringCRS`) defined on moving platforms such as cars, ships, aircraft, and spacecraft. For further information, see section 6.3.4.6.

Coordinate reference systems are commonly divided into sub-types. The common classification criterion for sub-typing of coordinate reference systems is the way in which they deal with earth curvature. This has a direct effect on the portion of the earth's surface that can be covered by that type of CRS with an acceptable degree of error.

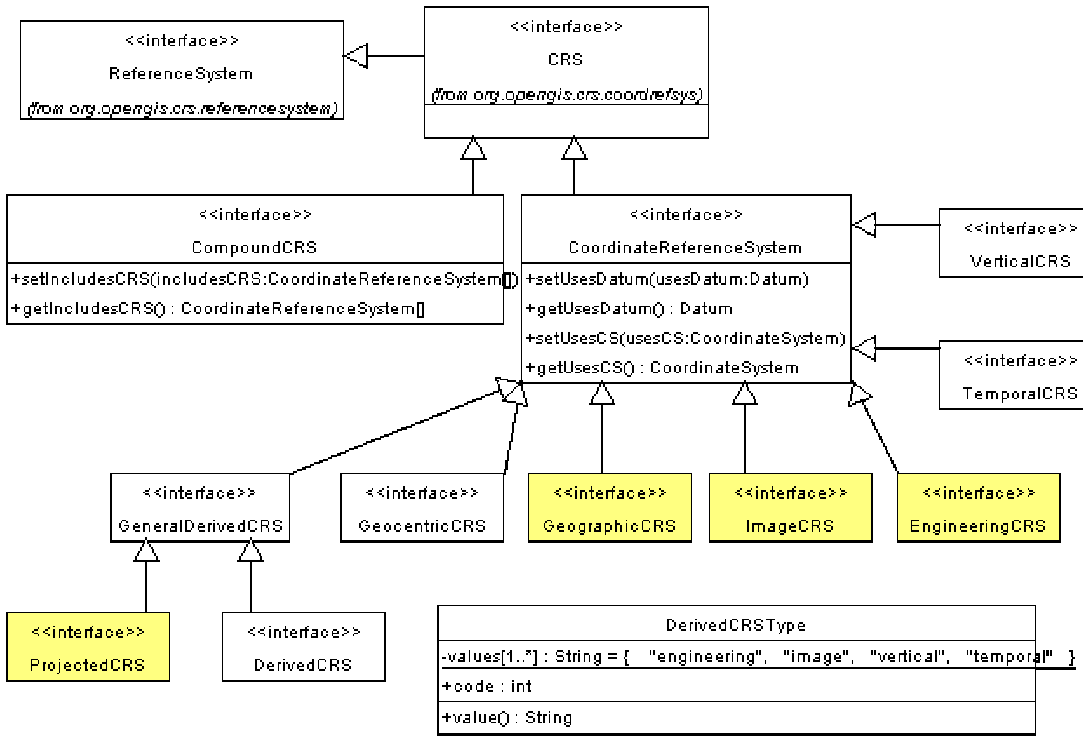


Figure 21 - Coordinate Reference System

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.

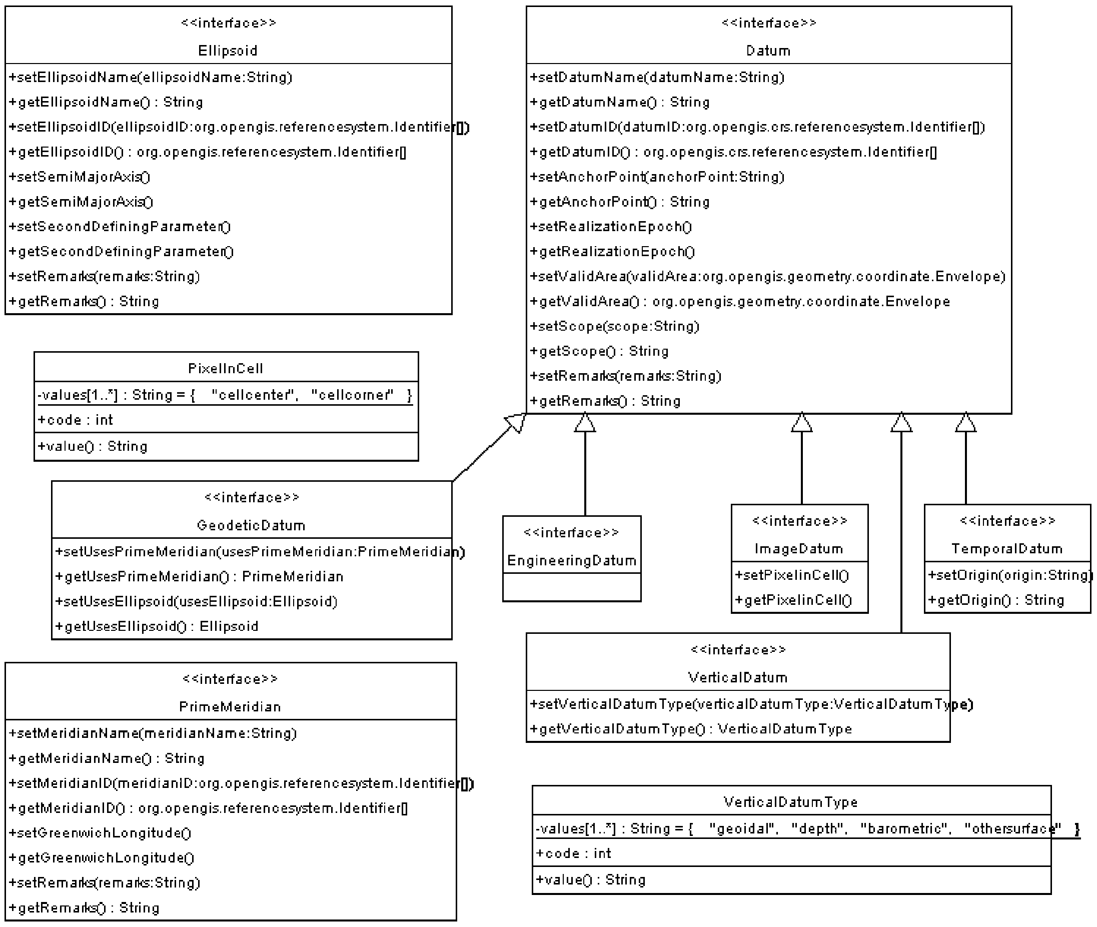
- `EngineeringCRS` — A contextually local coordinate reference system; which can be divided into two broad categories:

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.

6.3.4.3 Datum

`Datum` specifies the 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.



<<interface>>
Ellipsoid
+setEllipsoidName(ellipsoidName:String)
+getEllipsoidName() : String
+setEllipsoidID(ellipsoidID:org.opengis.referencesystem.Identifier[])
+getEllipsoidID() : org.opengis.referencesystem.Identifier[]
+setSemiMajorAxis()
+getSemiMajorAxis()
+setSecondDefiningParameter()
+getSecondDefiningParameter()
+setRemarks(remarks:String)
+getRemarks() : String
<<interface>>
Datum
+setDatumName(datumName:String)
+getDatumName() : String
+setDatumID(datumID:org.opengis.crs.referencesystem.Identifier[])
+getDatumID() : org.opengis.crs.referencesystem.Identifier[]
+setAnchorPoint(anchorPoint:String)
+getAnchorPoint() : String
+setRealizationEpoch()
+getRealizationEpoch()
+setValidArea(validArea:org.opengis.geometry.coordinate.Envelope)
+getValidArea() : org.opengis.geometry.coordinate.Envelope
+setScope(scope:String)
+getScope() : String
+setRemarks(remarks:String)
+getRemarks() : String
PixelInCell
-values[1..*] : String = { "cellcenter", "cellcorner" }
+code : int
+value() : String
<<interface>>
GeodeticDatum
+setUsesPrimeMeridian(usesPrimeMeridian:PrimeMeridian)
+getUsesPrimeMeridian() : PrimeMeridian
+setUsesEllipsoid(usesEllipsoid:Ellipsoid)
+getUsesEllipsoid() : Ellipsoid
<<interface>>
EngineeringDatum
<<interface>>
ImageDatum
+setPixelinCell()
+getPixelinCell()
<<interface>>
TemporalDatum
+setOrigin(origin:String)
+getOrigin() : String
<<interface>>
PrimeMeridian
+setMeridianName(meridianName:String)
+getMeridianName() : String
+setMeridianID(meridianID:org.opengis.referencesystem.Identifier[])
+getMeridianID() : org.opengis.referencesystem.Identifier[]
+setGreenwichLongitude()
+getGreenwichLongitude()
+setRemarks(remarks:String)
+getRemarks() : String
<<interface>>
VerticalDatum
+setVerticalDatumType(verticalDatumType:VerticalDatumType)
+getVerticalDatumType() : VerticalDatumType
VerticalDatumType
-values[1..*] : String = { "geoidal", "depth", "barometric", "othersurface" }
+code : int
+value() : String

Figure 22 - Datum

The `anchorPoint` property of `Datum` is a “description, possibly including coordinates, of the point or points used to anchor the datum to the Earth.” Also known as the "origin", especially for Engineering and Image Datums.

Of particular note for GO-1, is the use of `anchorPoint` as follows:

- For an engineering datum, the anchor point may be a physical point, or it may be a point with defined coordinates in another CRS.
- For an image datum, the anchor point is usually either the centre of the image or the corner of the image.

`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.

`ImageDatum` defines the origin of an image coordinate reference system. 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.

`PrimeMeridian` defines the origin from which longitude values are determined.

6.3.4.4 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.

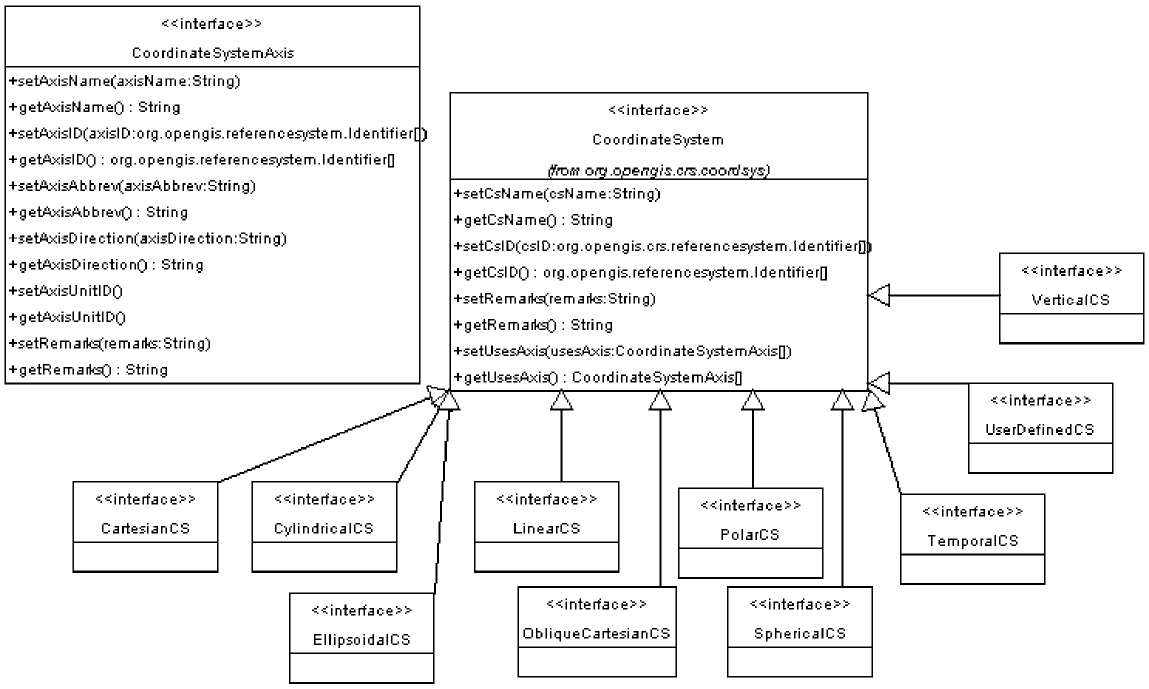

<<interface>>
CoordinateSystemAxis
+setAxisName(axisName:String)
+getAxisName() : String
+setAxisID(axisID:org.opengis.referencesystem.Identifier[])
+getAxisID() : org.opengis.referencesystem.Identifier[]
+setAxisAbbrev(axisAbbrev:String)
+getAxisAbbrev() : String
+setAxisDirection(axisDirection:String)
+getAxisDirection() : String
+setAxisUnitID()
+getAxisUnitID()
+setRemarks(remarks:String)
+getRemarks() : String
<<interface>>
CoordinateSystem
(from org.opengis.crs.coordsys)
+setCsName(csName:String)
+getCsName() : String
+setCsID(csID:org.opengis.crs.referencesystem.Identifier[])
+getCsID() : org.opengis.referencesystem.Identifier[]
+setRemarks(remarks:String)
+getRemarks() : String
+setUsesAxis(usesAxis:CoordinateSystemAxis[])
+getUsesAxis() : CoordinateSystemAxis[]
<<interface>>
VerticalCS
<<interface>>
UserDefinedCS
<<interface>>
CartesianCS
<<interface>>
CylindricalCS
<<interface>>
LinearCS
<<interface>>
PolarCS
<<interface>>
TemporalCS
<<interface>>
EllipsoidalCS
<<interface>>
ObliqueCartesianCS
<<interface>>
SphericalCS

Figure 23 - 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.

6.3.4.5 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) also 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.

Note attached to sourceCRS and targetCRS associations: The "sourceCRS" and "targetCRS" associations are mandatory for `Transformations` only. `Conversions` have a source CRS and a target CRS that are NOT specified through these associations, but through associations from GeneralDerivedCRS to `CoordinateReferenceSystem`.

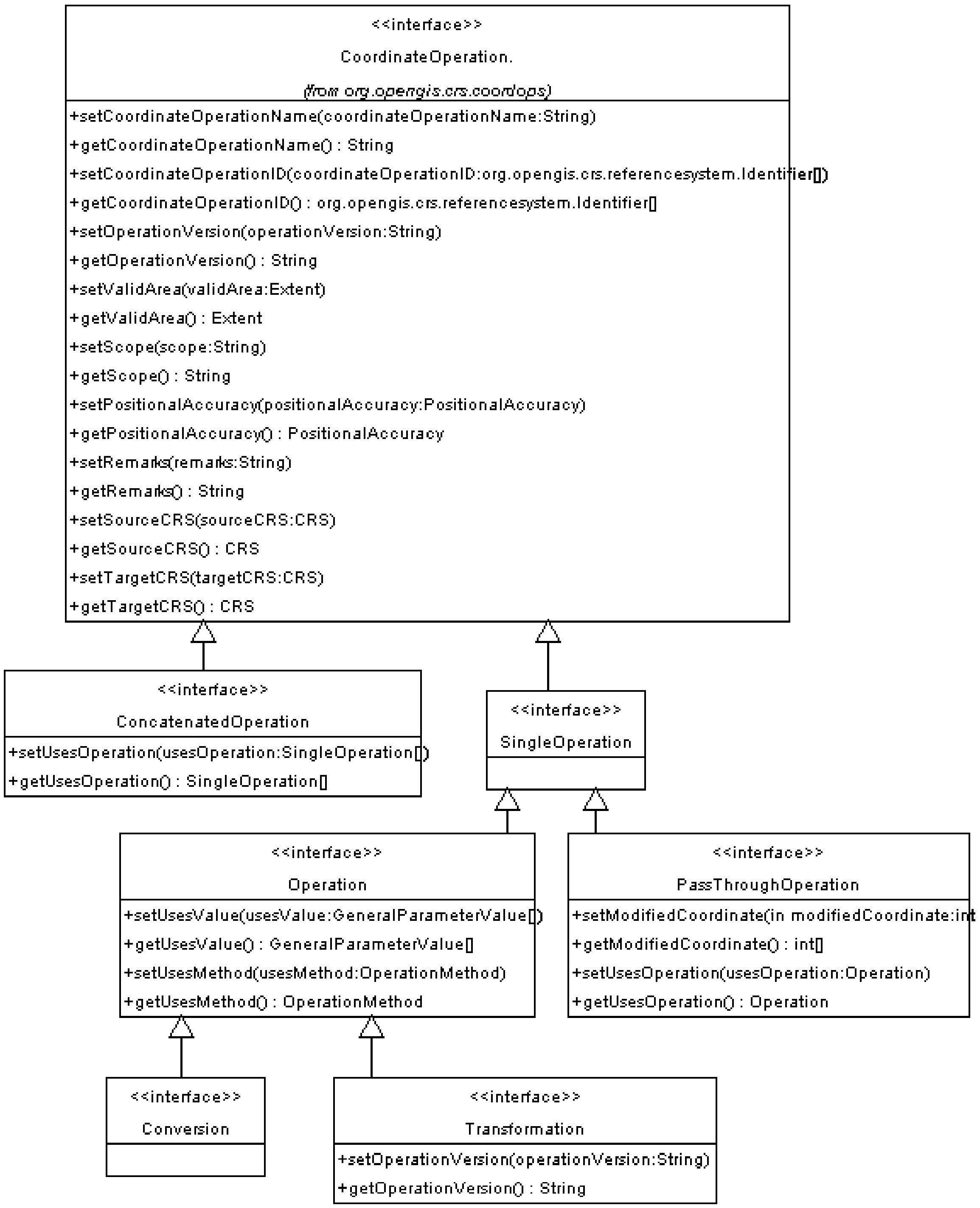

<<interface>>
CoordinateOperation.
(from org.opengis.crs.coordops)
+setCoordinateOperationName(coordinateOperationName:String)
+getCoordinateOperationName() : String
+setCoordinateOperationID(coordinateOperationID:org.opengis.crs.referencesystem.Identifier[])
+getCoordinateOperationID() : org.opengis.crs.referencesystem.Identifier[]
+setOperationVersion(operationVersion:String)
+getOperationVersion() : String
+setValidArea(validArea:Extent)
+getValidArea() : Extent
+setScope(scope:String)
+getScope() : String
+setPositionalAccuracy(positionalAccuracy:PositionalAccuracy)
+getPositionalAccuracy() : PositionalAccuracy
+setRemarks(remarks:String)
+getRemarks() : String
+setSourceCRS(sourceCRS:CRS)
+getSourceCRS() : CRS
+setTargetCRS(targetCRS:CRS)
+getTargetCRS() : CRS
<<interface>>
ConcatenatedOperation
+setUsesOperation(usesOperation:SingleOperation[])
+getUsesOperation() : SingleOperation[]
<<interface>>
SingleOperation
<<interface>>
Operation
+setUsesValue(usesValue:GeneralParameterValue[])
+getUsesValue() : GeneralParameterValue[]
+setUsesMethod(usesMethod:OperationMethod)
+getUsesMethod() : OperationMethod
<<interface>>
PassThroughOperation
+setModifiedCoordinate(in modifiedCoordinate:int
+getModifiedCoordinate() : int[]
+setUsesOperation(usesOperation:Operation)
+getUsesOperation() : Operation
<<interface>>
Conversion
<<interface>>
Transformation
+setOperationVersion(operationVersion:String)
+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`. 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.

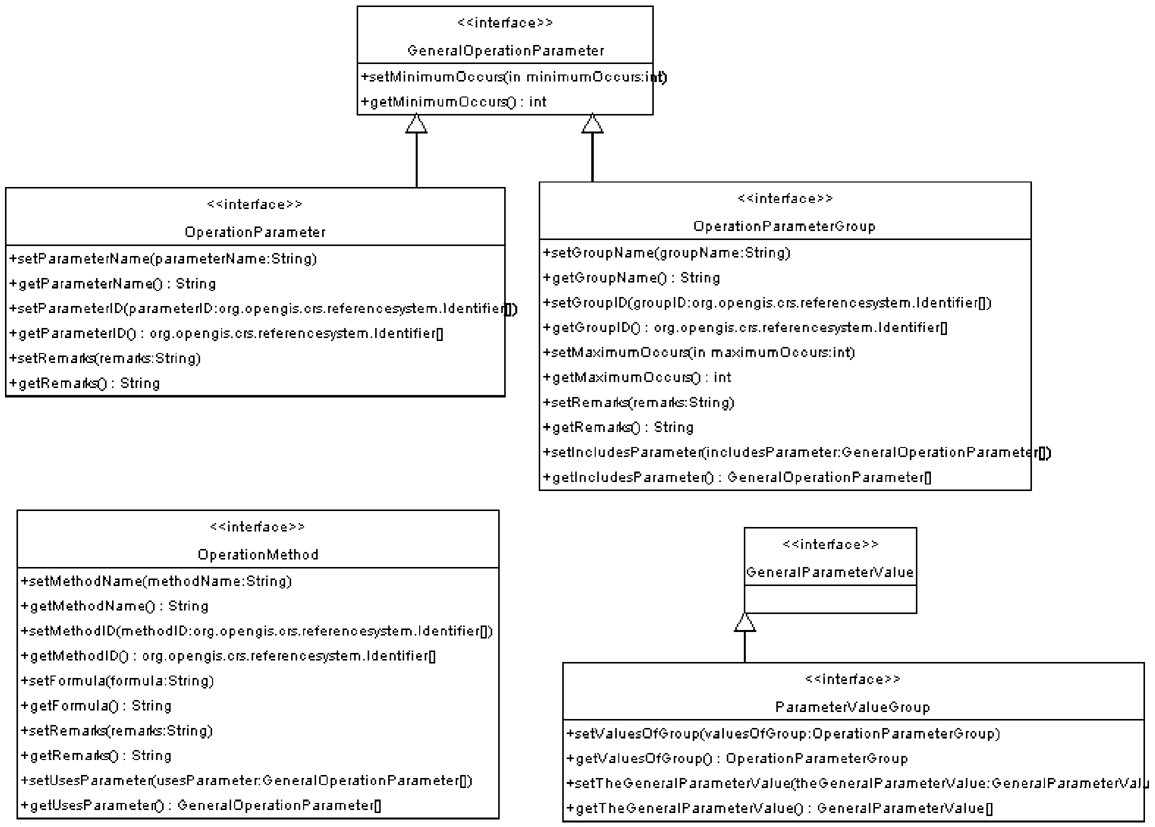

<<interface>>
GeneralOperationParameter
+setMinimumOccurs(in minimumOccurs:int)
+getMinimumOccurs() : int
<<interface>>
OperationParameter
+setParameterName(parameterName:String)
+getParameterName() : String
+setParameterID(parameterID:org.opengis.crs.referencesystem.Identifier[])
+getParameterID() : org.opengis.crs.referencesystem.Identifier[]
+setRemarks(remarks:String)
+getRemarks() : String
<<interface>>
OperationParameterGroup
+setGroupName(groupName:String)
+getGroupName() : String
+setGroupID(groupID:org.opengis.crs.referencesystem.Identifier[])
+getGroupID() : org.opengis.crs.referencesystem.Identifier[]
+setMaximumOccurs(in maximumOccurs:int)
+getMaximumOccurs() : int
+setRemarks(remarks:String)
+getRemarks() : String
+setIncludesParameter(includesParameter:GeneralOperationParameter[])
+getIncludesParameter() : GeneralOperationParameter[]
<<interface>>
OperationMethod
+setMethodName(methodName:String)
+getMethodName() : String
+setMethodID(methodID:org.opengis.crs.referencesystem.Identifier[])
+getMethodID() : org.opengis.crs.referencesystem.Identifier[]
+setFormula(formula:String)
+getFormula() : String
+setRemarks(remarks:String)
+getRemarks() : String
+setUsesParameter(usesParameter:GeneralOperationParameter[])
+getUsesParameter() : GeneralOperationParameter[]
<<interface>>
GeneralParameterValue
<<interface>>
ParameterValueGroup
+setValuesOfGroup(valuesOfGroup:OperationParameterGroup)
+getValuesOfGroup() : OperationParameterGroup
+setTheGeneralParameterValue(theGeneralParameterValue:GeneralParameterValu
+getTheGeneralParameterValue() : GeneralParameterValue[]

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.

6.3.4.6 Support for Relative Coordinates

One capability that has been identified as a requirement for GO-1 Application Objects, but has not yet been captured in any prior OGC Specifications, is the ability to support relative coordinates. This need is fulfilled by creating a coordinate system that effectively has its origin moved to the reference point of interest. The method proposed to accomplish this is setReferencePosition(), which takes a DirectPosition as its sole argument and sets the origin of the object against which it is invoked, to the position specified by the argument in its own CRS.

Given that the OGC definition for a CRS is essentially that of a data structure, there is nothing in any extant specification that prohibits this operation provided a reasonable meaning for the origin of a CRS can be found. The most immediately obvious means of implementing setReferencePosition() would follow the precedent set by the Engineering CRS, and set the AnchorPoint of the datum of the CRS to an appropriate representation of the given DirectPosition. Image CRS are also amenable to this type of transformation. These ideas are expanded below.

6.3.5.6.1 Relative Coordinate Reference Systems

Coordinates in GO-1 implementations are implemented as `DirectPositions`. Each `DirectPosition` has a `CoordinateReferenceSystem` (CRS). This CRS may be Derived from another CRS. The exact relationship of this derivation is defined by a conversion and a type. The conversion determines (a) whether and how the Derived CRS scales with the referenced CRS and (b) which `DirectPosition` in the referenced CRS corresponds to the origin of the Derived CRS. The type is either Engineering or Image. A `DirectPosition` within a Derived CRS is a relative coordinate with respect to a coordinate in the referenced CRS.

A Coordinate Transformation Service that operates on such relative coordinates must incorporate the conversion information when dealing with a Derived CRS.

6.3.5.6.2 Relative CRS and the Canvas

A `Canvas` has a spatial (type Geographic) CRS and a display CRS. The display CRS is categorised as a Derived CRS of type Image and is derived from the spatial CRS. Most implementations will further define the display CRS as a Pixel CRS, which is a rasterization of the Derived Image CRS. The Pixel CRS does not scale with the spatial CRS.

6.3.5.6.3 Examples

The examples below assume the existence of the following subtypes of the spatial (Geographic) CRS: LatLonAlt and UTM; the following subtype of the Image CRS: Pixel; and the following subtypes of the Engineering CRS: RangeBearing and XYZ.

Example 1: Image (Pixel) Relative Coordinate Not Scaling with LatLonAlt CRS. A dynamically-constructed symbol that does not scale with the Canvas spatial CRS is to be displayed by the Canvas. The spatial CRS of a Canvas is a LatLonAlt CRS. The symbol is constructed as a square geometry that is defined by four Pixel relative coordinates (Pixel DirectPositions). The Pixel DirectPositions are associated with a single Pixel CRS (not the Canvas display CRS), which is a Derived CRS of type Image, whose origin is denoted by a LatLonAlt DirectPosition associated with the LatLonAlt CRS (the Canvas spatial CRS). Since the Pixel CRS references only one DirectPosition on the LatLonAlt CRS, it does not scale with the LatLonAlt CRS.

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 geographic scale (zoom factor). NOTE: This implies that coordinate transformation of this type must be done on an as needed basis, and repeated at least when the scale, location, or projection of the geographic CRS changes in the display. The transformation cannot be done once to convert the coordinates into their relative locations in lat/long because that transformation will become invalid at the time the scale or projection changes.

Example 2: Image Relative Coordinate Chained and Scaling with UTM CRS. A registered image is to be displayed in a fixed range and bearing from a fixed location in a spatial CRS. The spatial CRS is a UTM CRS. Derived from the UTM CRS is a CRS of type Engineering (RangeBearing). Derived from the Derived RangeBearing CRS is a CRS of type Image. The registered image is set with two DirectPositions in the Derived Image CRS. The Derived Image CRS scales with the Derived RangeBearing CRS. The Derived RangeBearing CRS scales with the UTM CRS.

Example 3: Engineering (RangeBearing) Relative Coordinate Scaling with UTM CRS. A sector is to be displayed that does not scale with a spatial CRS. The spatial CRS is a UTM CRS. A Derived CRS of type Engineering (specifically RangeBearing) is derived from the spatial UTM CRS. A sector geometry made up of two arcs and two straight lines are denoted with four RangeBearings. The Derived Engineering CRS does not scale with the UTM CRS.

Example 4: XYZ Relative Coordinate Scaling with LatLonAlt CRS. A ship is to be depicted coming into port. The port is described by a set of DirectPostions in a LatLonAlt CRS. The ship is described by a set of XYZ DirectPositions in a Derived CRS of type Engineering (measured in meters). The conversion for the Derived CRS includes a direct scaling with the LatLonAlt CRS, as well as the origin of the Derived CRS (such as at the center of buoyancy of the ship or at the forward-most point of the bow at main deck level) correlating a LatLonAlt DirectPosition that denotes the ship position in the LatLonAlt CRS.

For example, the graphic depicting the ship includes coordinates for the stern in meters from the bow, such as (100, 0, 0). Since the X coordinate is 100 meters, changing map scale will change the scale of the graphic, but changing the LatLonAlt reference for the ship will move the ship graphic to its new location.

6.4 Service Objects

Service Objects provide processing capabilities that can be accessed from but are not a part of the application. Service Objects should be available to an application regardless of the application environment that was used to build the service. For example, a service built using C on UNIX could expose interfaces to Java, Corba, and DCOM applications. The actual implementation of the service is opaque to the applications. It is sufficient that there is an interface available in their environment that is readily accessible.

6.4.1 Data Service Objects

Data services provide access to collections of data in repositories and databases. Resources accessible by Data Services can generally be referenced by a name (identity, address, etc). Given a name, Data Services can then find the resource. Data Services usually maintain indexes to help speed up the process of finding items by name or by other attributes of the item. The sections below describe the current OGC Reference Model (ORM) set of Data Services. Examples of ORM data services include:

- Feature Access Services (FAS)
- Coverage Access Services (CAS)
- Sensor Collection Service (SCS)
- Image Archive Services (IAS)

6.4.2 Portrayal Service Objects

Portrayal services provide visualisation of geospatial information. Portrayal Services are components that, given one or more inputs, produce rendered outputs (e.g., cartographically portrayed maps, perspective views of terrain, annotated images, views of dynamically changing features in space and time, etc.). Portrayal Services can be tightly or loosely coupled with other services such as Data and Processing Services and transform, combine, or create portrayed outputs. Portrayal Services may use styling rules specified during configuration or dynamically at runtime by Application Services. Portrayal Services can be sequenced into a “value-chain” of services to perform specialized processing in support of information production workflows and decision support. Examples of ORM portrayal services include:

- Map Portrayal Services (MPS)
- Coverage Portrayal Services (CPS)
- Mobile Presentation Services

6.4.3 Processing Service Objects

Processing services operate on geospatial data and provide "value-add" services for applications. They can transform, combine, or create data. Processing Services can be tightly or loosely coupled with other services such as Data and Portrayal Services. Processing Services can be sequenced into a "value-chain" of services to perform specialized processing in support of information production workflows and decision support. Examples of processing services include:

- Chaining Services
- Coordinate Transformation Services (CTS)
- Geocoder Services
- Gazetteer Services
- Geoparser Services
- Reverse Geocoder Services
- Route Determination Services

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, accompanied by sequence or state diagrams.

7.1 Adding information to a display

Description: A Feature is rendered on a display

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

Flow of events:

1. Application extracts Geometry object from the Feature.
2. Application requests a Graphic object from the DisplayFactory.
3. Application uses the Geometry to set the geometric attributes of the Graphic
4. Application adds the Graphic object to the Canvas object.

Postcondition: The Feature has been rendered with default styling on the display device.

This sequence of operations is depicted below.

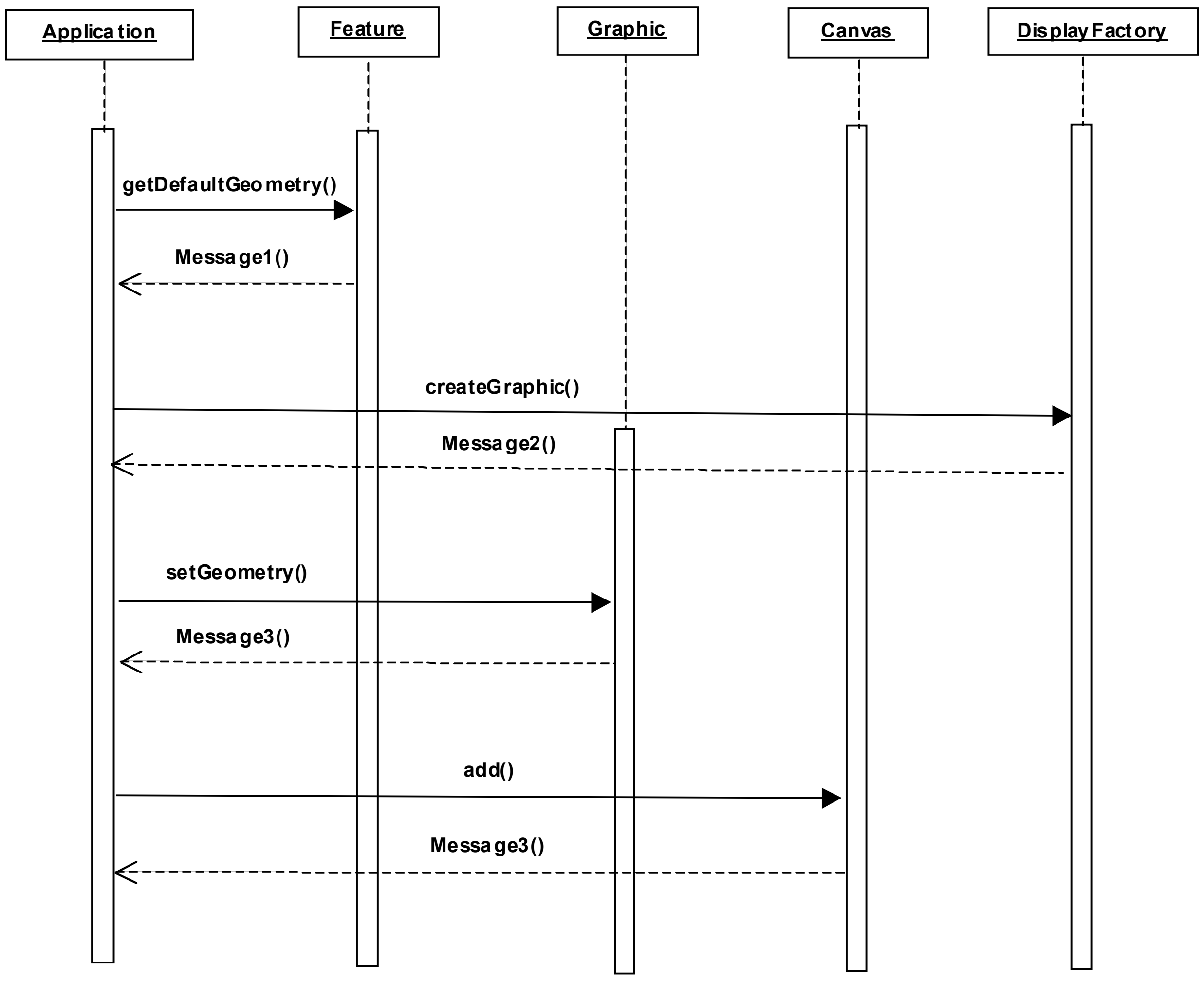


Figure 26 - Adding Information to a Display

7.2 Mouse click selects graphical object.

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

Preconditions: The user is using 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 an editable object on the display device, 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 caught and consumed by SelectItemsHandler

4. SelectItemsHandler acquires the GraphicStyle from the Graphic being edited.

5. SelectItemsHandler changes the GraphicStyle to indicate that the Graphic is being edited.

6. SelectItemsHandler tells the Graphic to go into editing mode and display its editing handles.

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

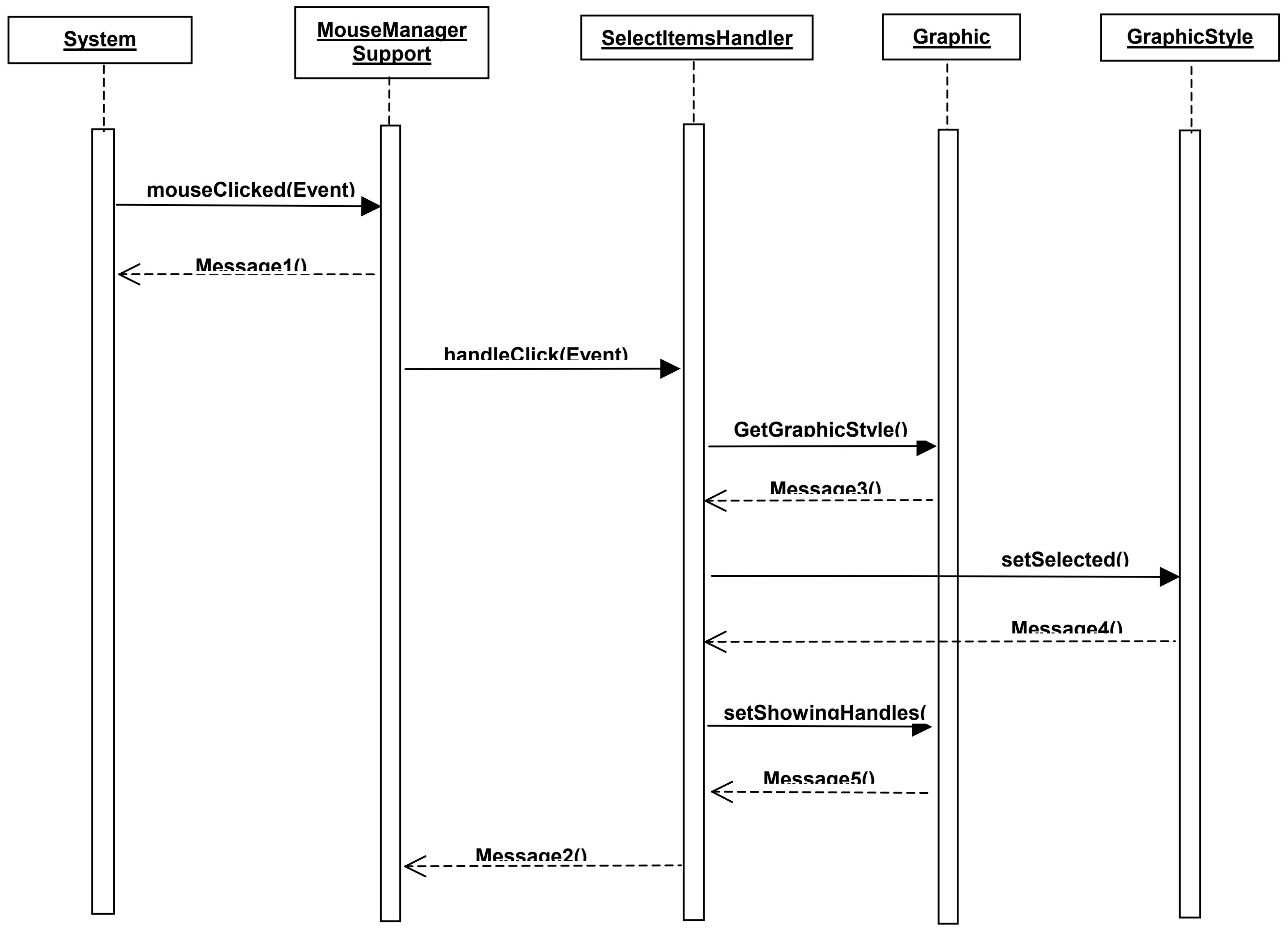


Figure 27 - Selecting a Graphic Object

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

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

Flow of events:

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

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

3. Application gets various styling attributes from the SLD.

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

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

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

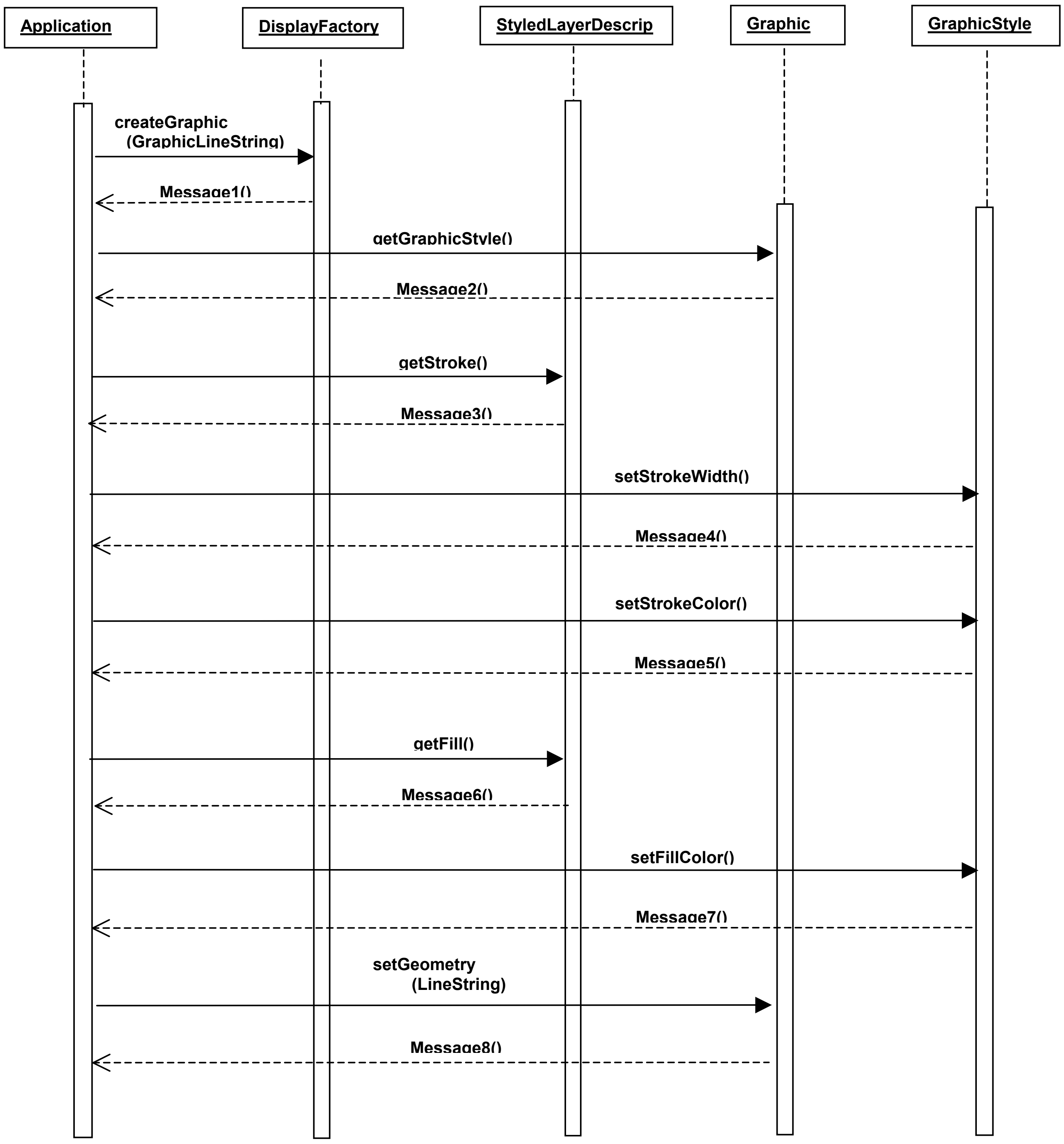


Figure 28 - Graphic Object Creation

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 Information Object and Service 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.

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

Simple Feature SQL:

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

Feature Geometry:

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

Feature:

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

Grid Coverages:

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

Catalog Service:

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

Geography Markup Language (GML):

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

[11] OpenGIS ® Geography Markup Language (GML) Implementation Specification (version 3.0), OGC document 02-023r4.

Web Map Server (WMS):

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

Styled Layer Descriptor (SLD):

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

Web Feature Server (WFS):

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

Filter Encoding:

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

Coordinate Transformation Service (CTS):

[16] OpenGIS ® Coordinate Transformation Services Implementation Specification, version 1.0, available at: http://www.opengis.org/techno/implementation.htm

Web Coverage Server (WCS):

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

Coverage Portrayal Server (CPS):

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

Style Management Service:

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

Registry Service:

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

Integrated Client:

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

Service Information Model:

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

OWS Architecture:

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

OGC Reference Model:

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

Spatial Reference System:

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

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

Open Source Implementation Baselines

Geobject

1. http://geobject.org/umldoc/2.0alpha

2. http://sourceforge.net/projects/geobject

Geotools2

1. http://modules.geotools.org/core

2. http://www.geotools.org