
# deegree development

lat/lon

Fitzke / Fretter / Poth GbR
Mekenheimer Allee 176
53115 Bonn
Tel. ++49 228 732838

Content

Figures

deegree includes implemenations of the the following OGC and ISO specifications:

- Web Feature Service (WFS) 1.0.0
- Web Map Service (WMS) 1.1.1
- Web Coverage Service (WCS) 1.0.0
- Web Terrain Server (WTS) 0.3.2
- OGC Web Services Stateless Catalog Profile 0.0.6 (WCAS)
- Filterencoding 1.0.0
- Styled Layer Descriptor (SLD) 1.0.0
- GML 2.1.2
- Grid Coverage
- Coordinate System Transformation
- ISO 19107 geometry model

The realization of the specification in deegree determines its package structure:

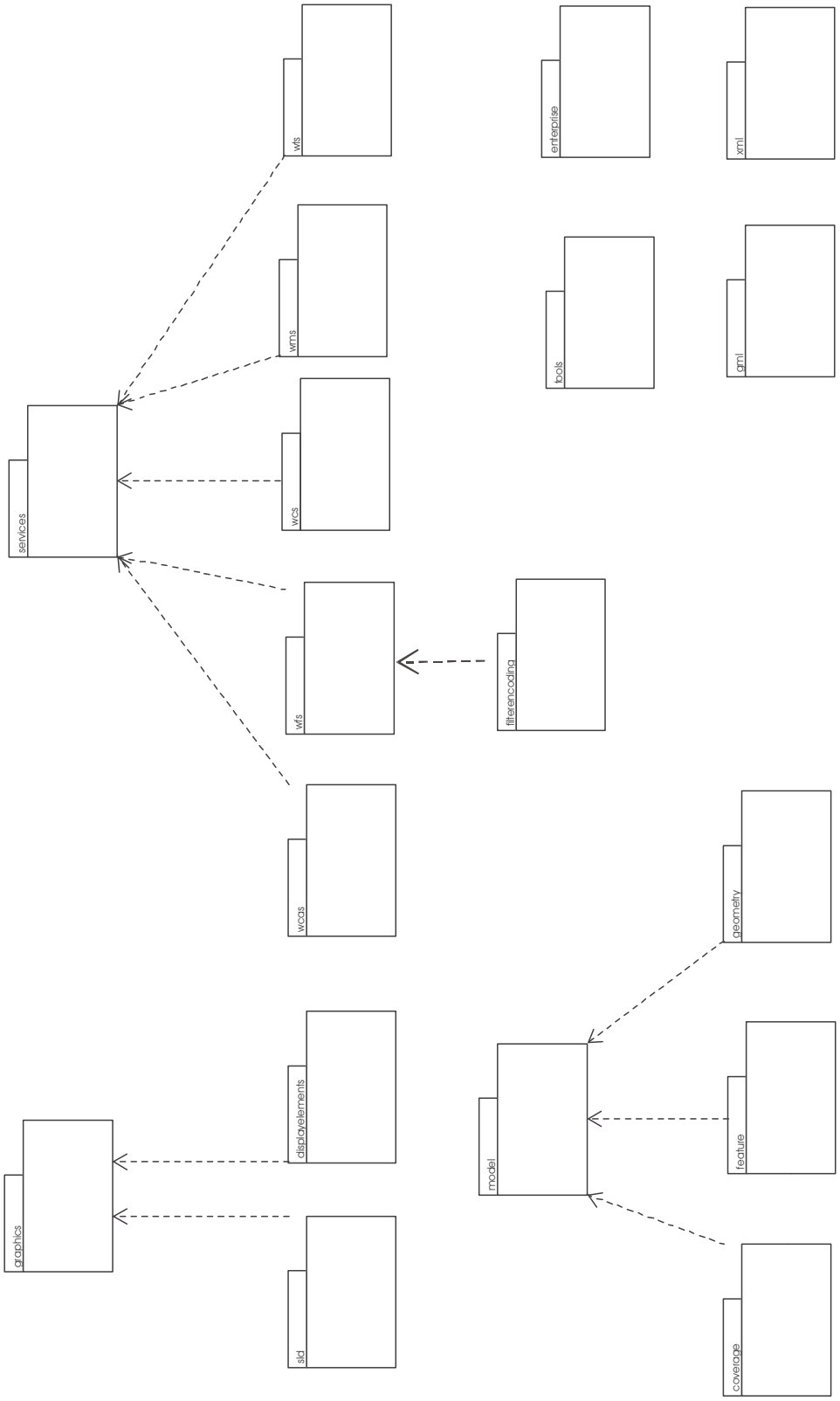



**Figure 1 extract of the deegree package structure**

# deegree geometry and feature model

**Figure 2 extract of the deegree geometry model (overview)**

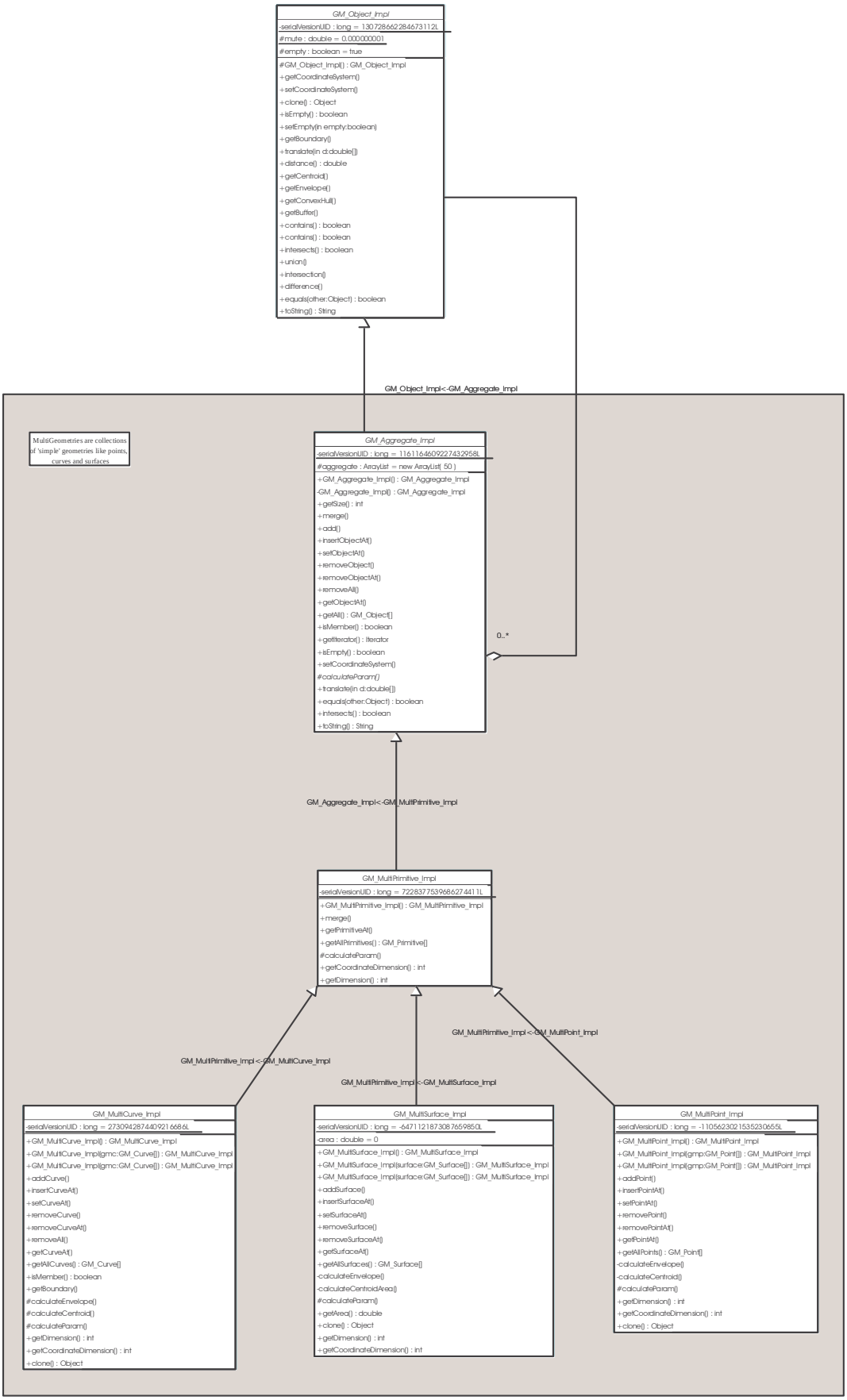



**Figure 3 extract of the deegree geometry model (detail I)**

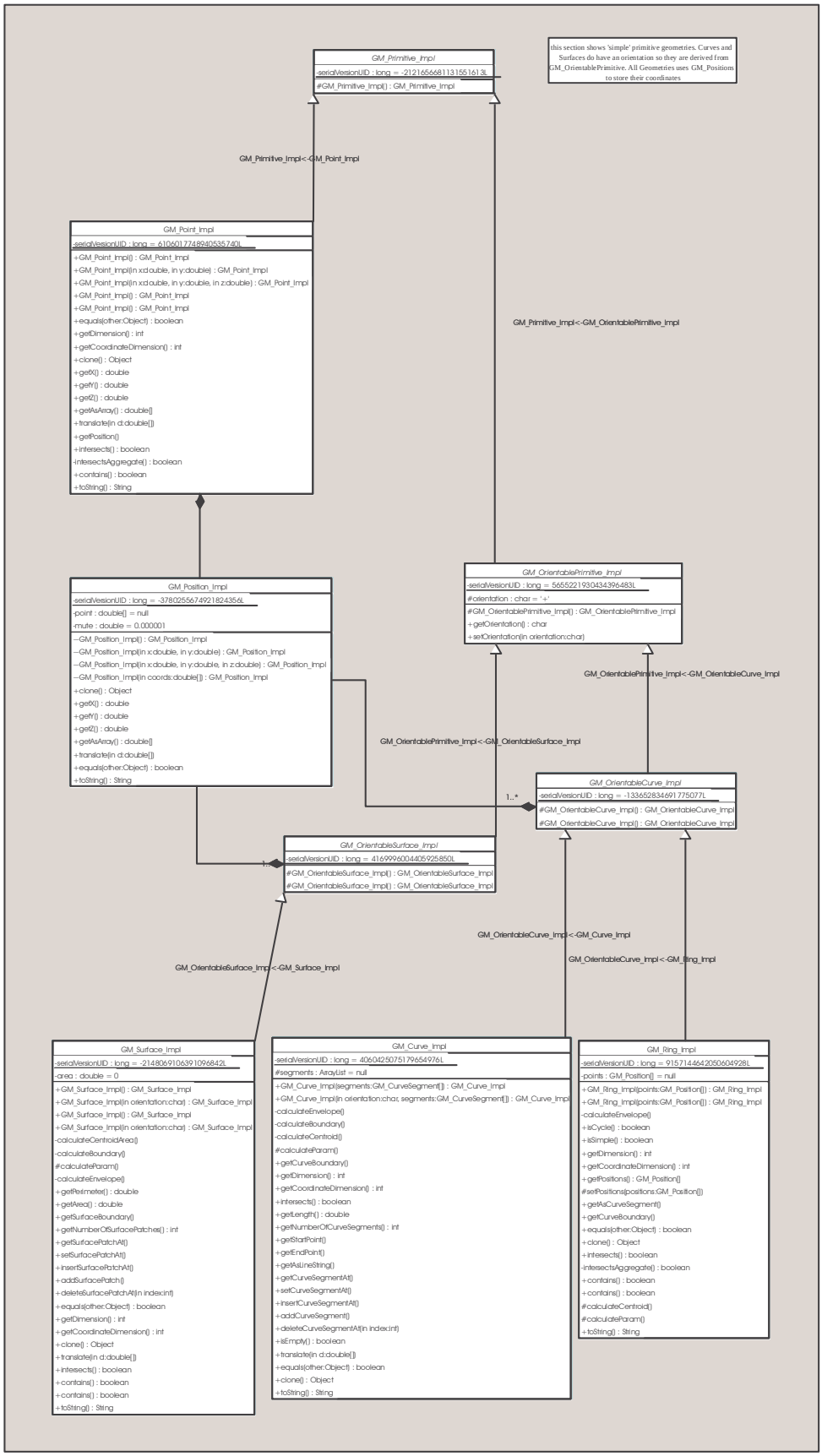



**Figure 4 extract of the deegree geometry model (detail II)**

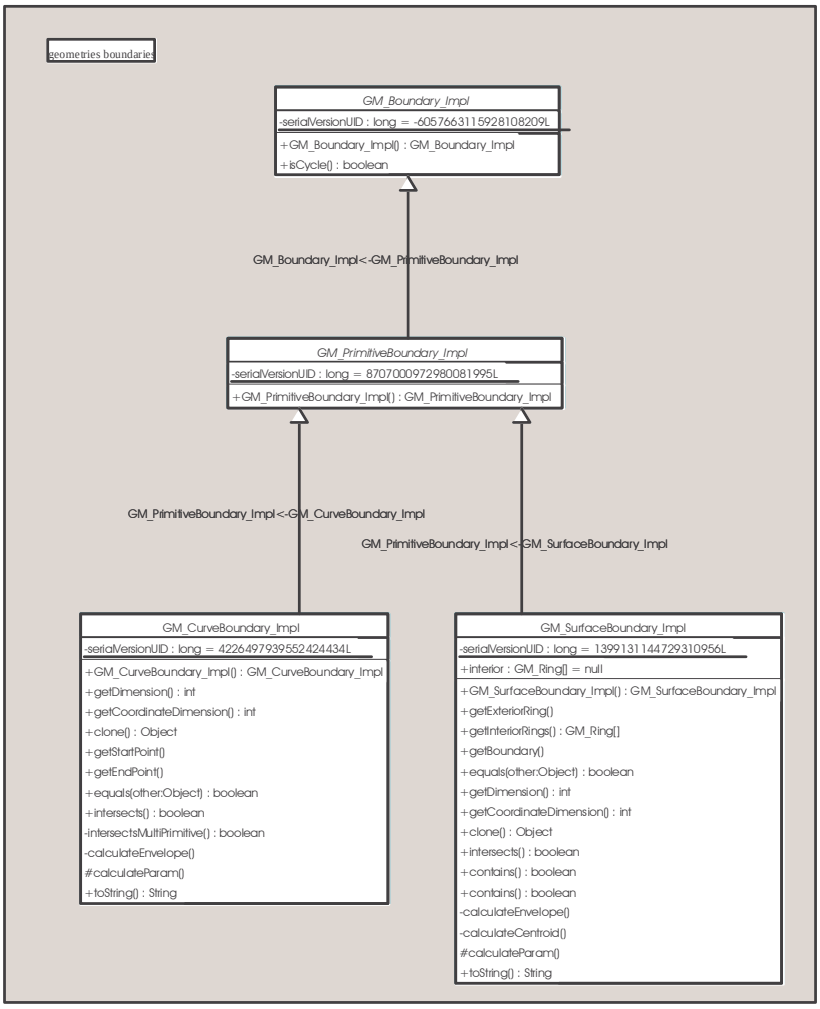



**Figure 5 extract of the deegree geometry model (detail III)**

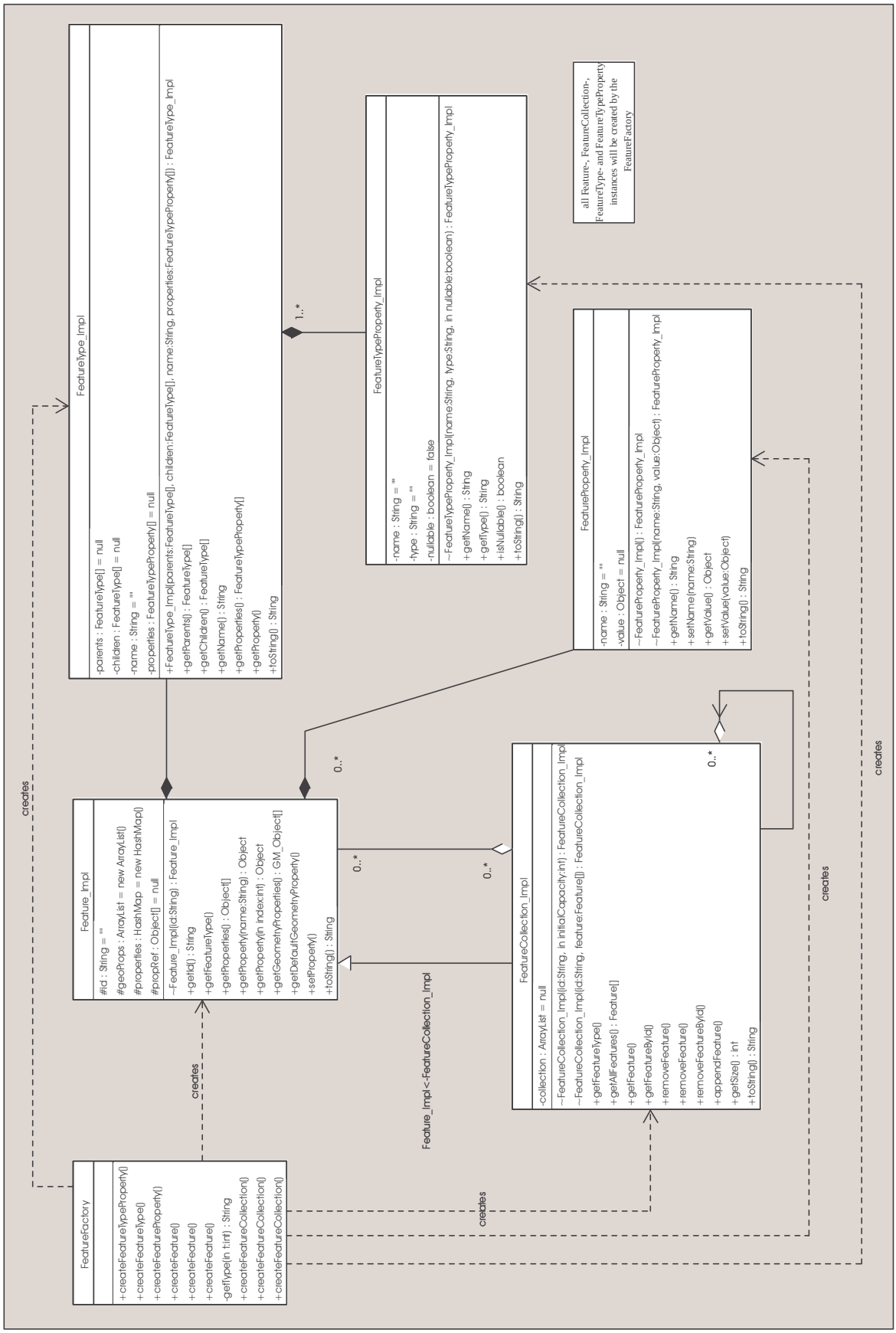
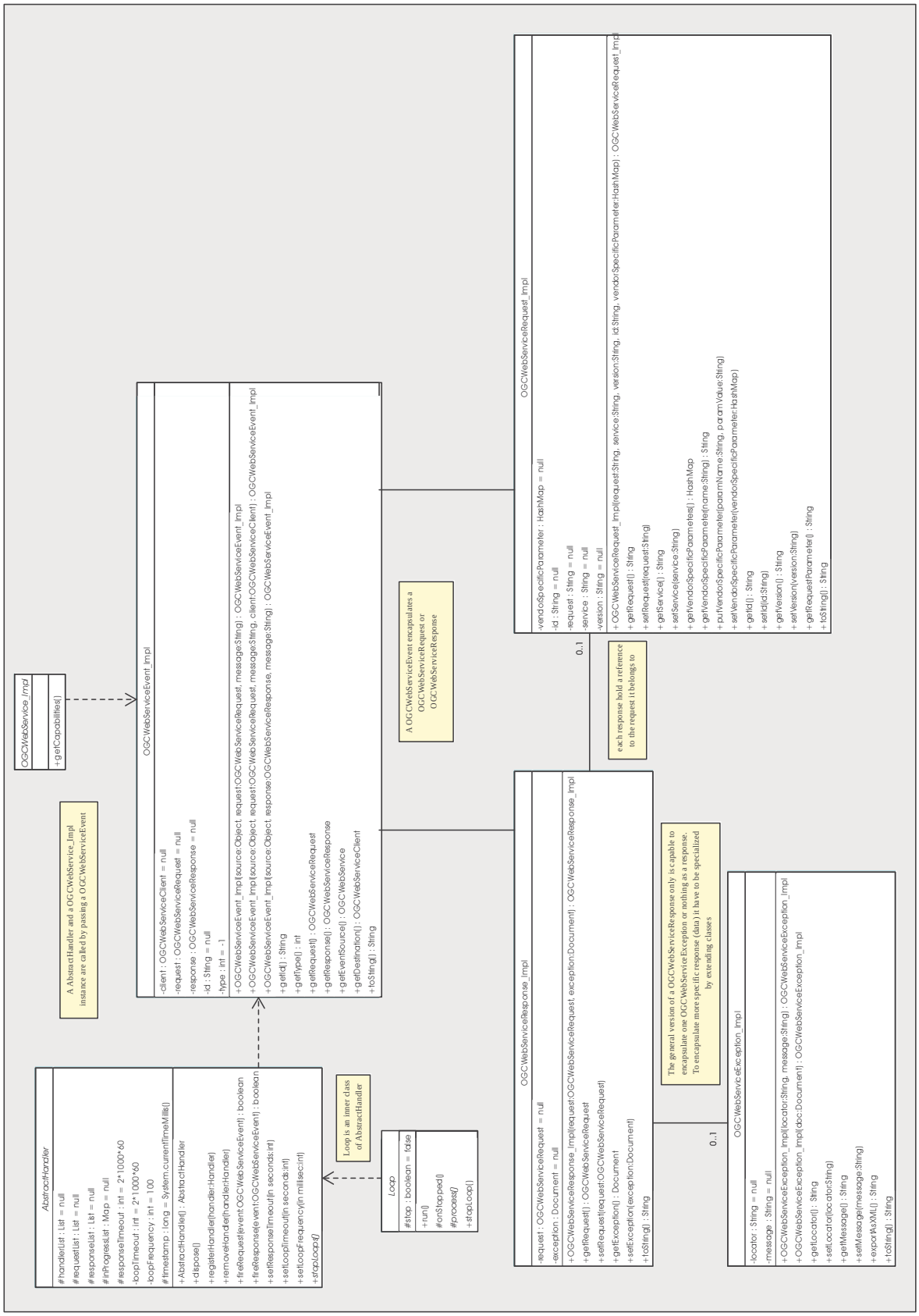



**Figure 6 deegree feature model**

# Introduction to the deegree communication- and event-model



# WMS / WFS architecture and processing




**Figure 7 WMS request processing**

deegree is splitted up into several more or less independ moduls. One of these moduls is the WMS. It contains 'just' the ability to collect data and render them according to defined style informations. The WMS modul don't contains any classes for direct accessing data from shapefile, databases etc.. It's just a kind of 'render engine'. Beside the WMS there exists WFS and WCS as data-accessing moduls. What's common the WXX-moduls is that they are just simple classes implementing the org.deegree.services.OGCWebService interface. So each of them provides a method doService(OGCWebServiceEvent). So they can be called by this method within any program without establishing network-connections. This it what happens if data sources of the WMS are declared as LOCALWFS or LOCALWCS. The WMS modul invokes the doService-method of the WFS resp. WCS modul (WFService_Impl and WCService_Impl). --> see sequence diagram attached to the mail.

To provide networking capabilities to the WXX-modules Servlets located in the org.deegree_impl.enterprise package are used as facades for the WFService_Impl , WCService_Impl ... classes. So there is no need to register WFS and WCS at the JBOSS/Tomcat if just like to use your local data with your WMS.

But the WMS also has the capability to access data from WFS, WCS and cascaded WMS neither running in the same VM nor being a deegree based service. The access to these services (datasources) is encapsulated within the RemoteWXX-classes. These classes also implementes the org.deegree.services.OGCWebService interface and hides the networking code. So a remote web feature service runing somewhere in this world can be invoked by the WMS just by calling the doService-method of the RemoteWFS class.

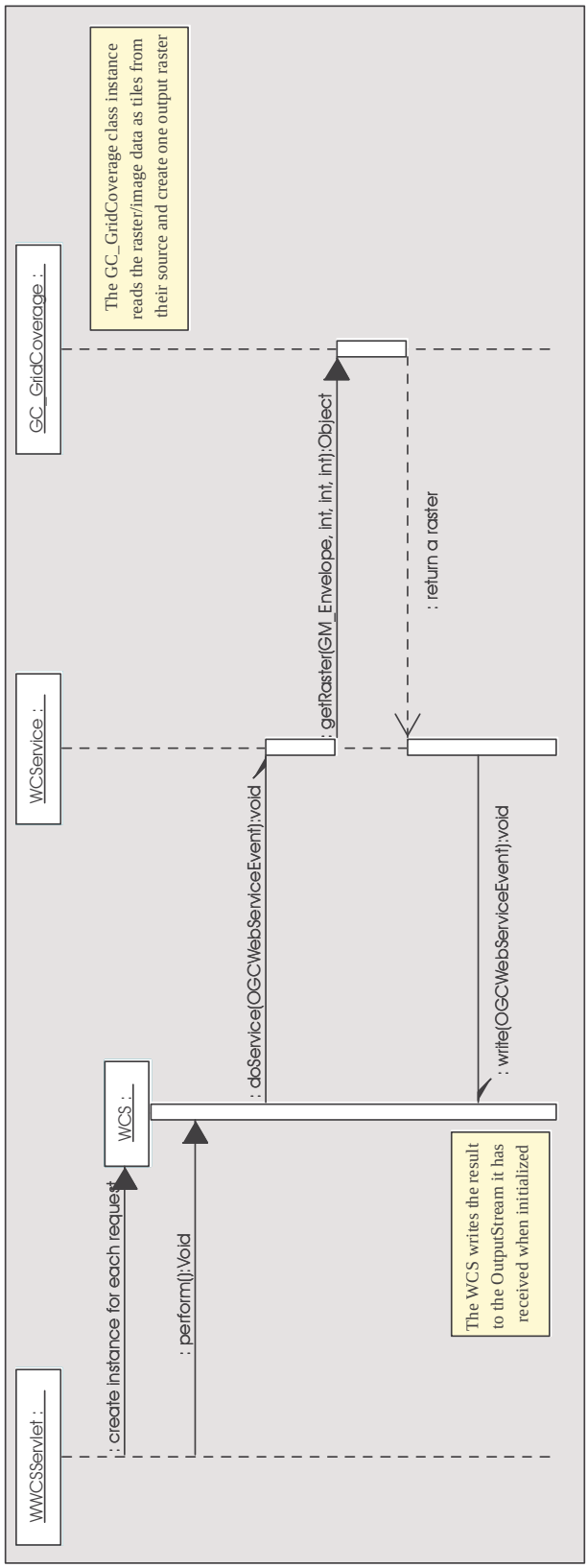
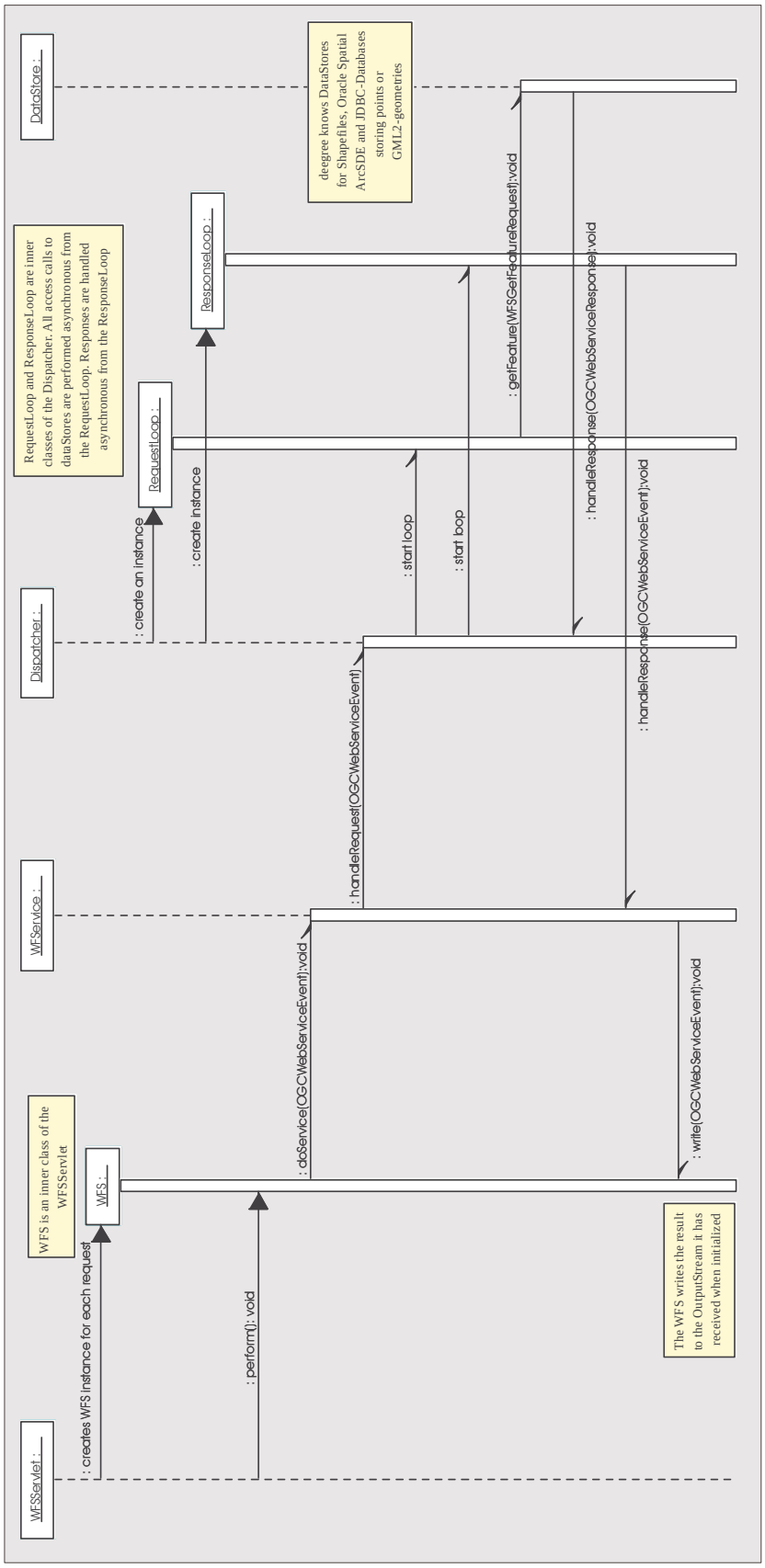



**Figure 8 WCS request processing**




As has been seen the deegree WMS has no own data access modul. All data access for vector data is handled by local or remote Web Feature Services. So the deegree WFS can be used as 'internal' modul without a web interface to provide vector data access to the deegree WMS or as an independent web service linked into a servlet (WFSServlet & WFS) to be used within a servlet engine.
Because one GetFeature request (as well as other WFS requests) can target more then one feature type it is important for optimizing the performance to run request processing in independ threads. So the WFService class splits up a incoming request into independ requests for each contained feature type deligates their request processing to the Dispatcher. The Dispatcher starts to independ threads; one for fireing request events to the responsible datastores (RequestLoop) and one for collecting responses from the datastores (ResponseLoop). If all involved datastores has answered the Dispatcher creates a response object, encapsulates it into a event and calls the handleResponse(..)-method of the responsible classes. In most cases this will be the WFService which had forced the GetFeature request but this is not required it may be any other class that implements the Handler interface and returns 'true' if the 'isInterested'-method is called by the Dispatcher. This enables that more then one object can be receiver of a response (or a request) which can be used for logging for example --> one object performes the request/response and one logs the performed actions. The WFService itself will send the response back to the calling client that may is a sevlet (see figure) or the deegree WMS or any other application/class that implements the OGCWebService interface

**Figure 9 WFS request processing part I**

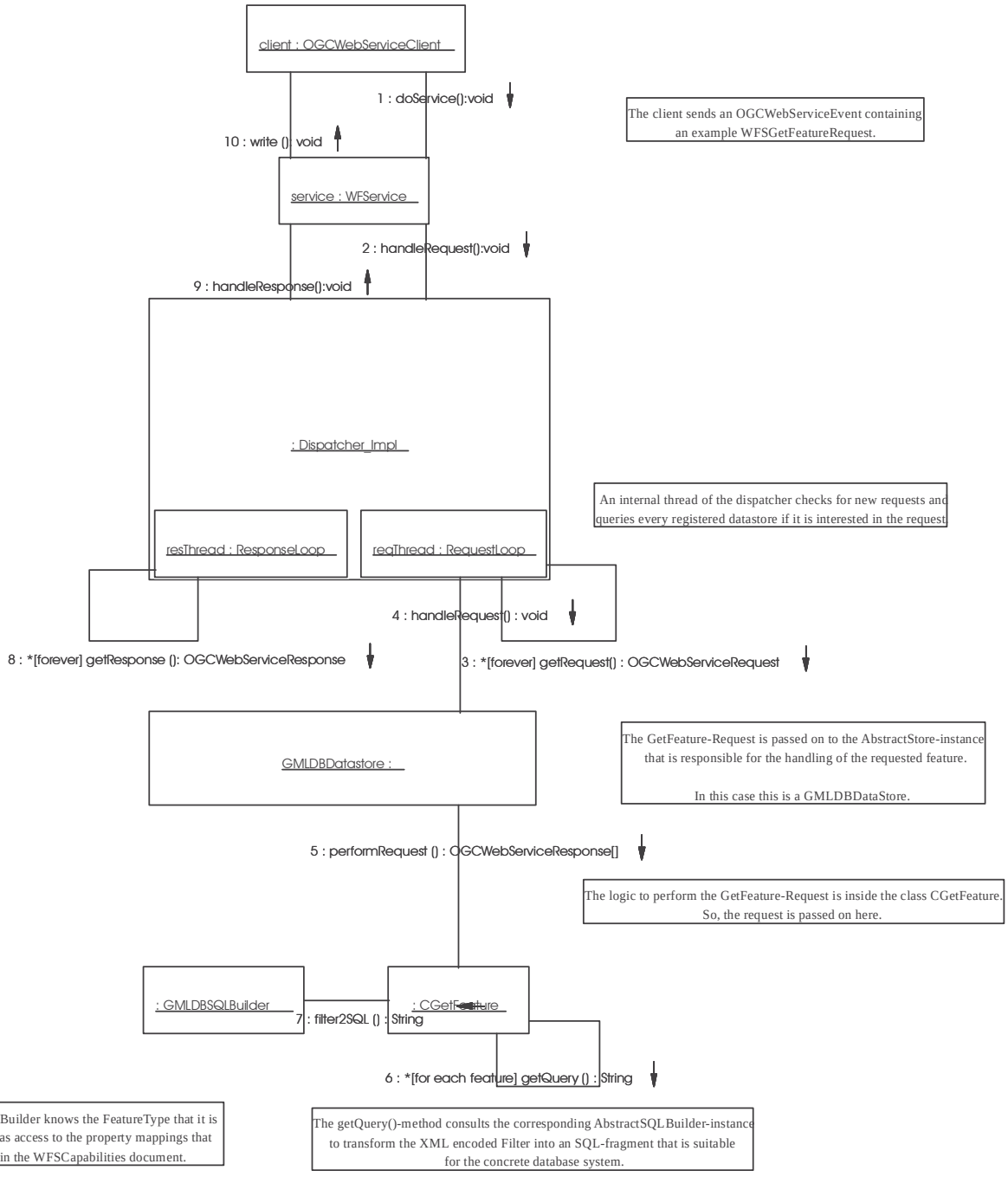



**Figure 10 WFS request processing part II**

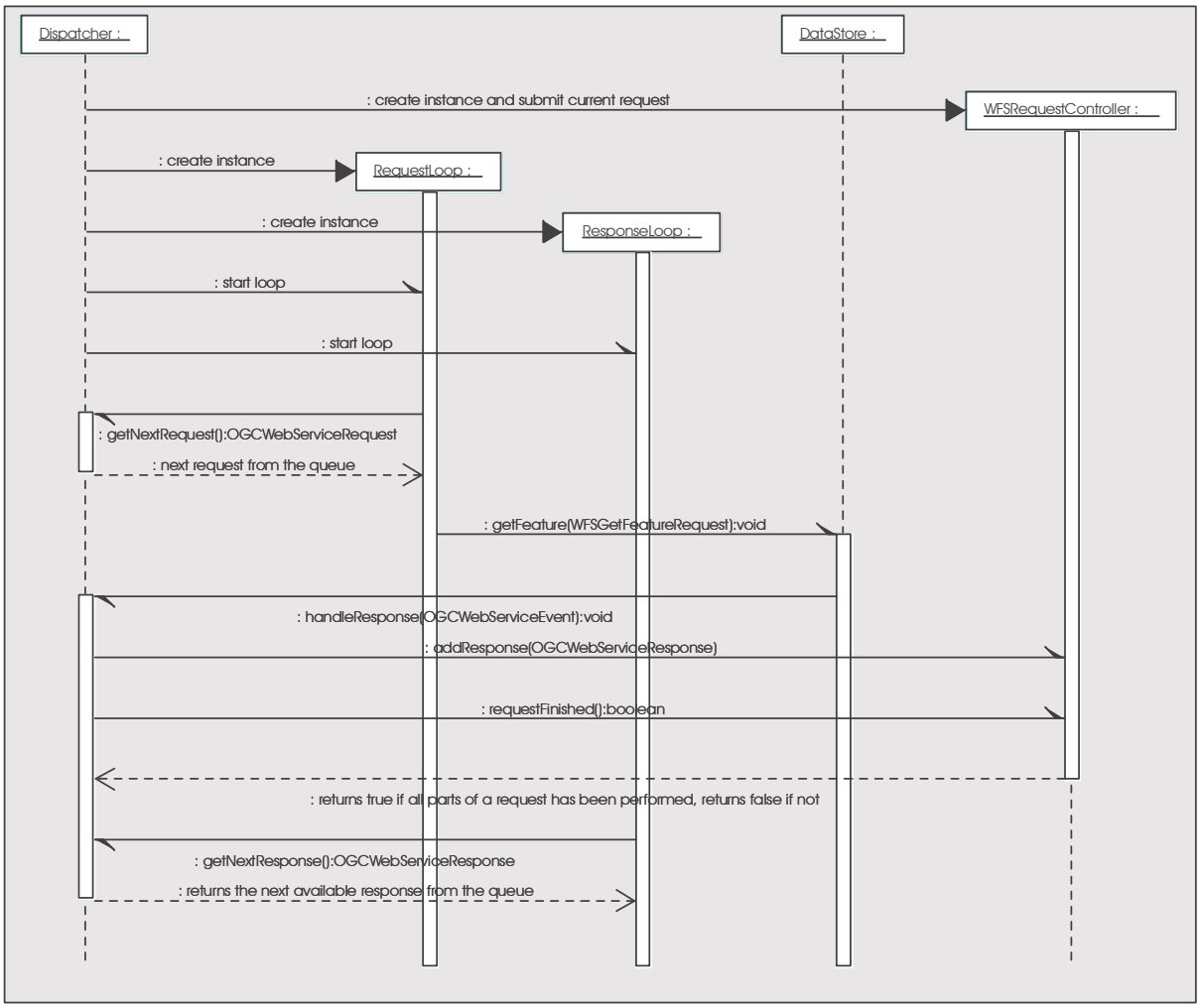



**Figure 11 workflow of the Dispatcher processing WFS requests**

## Basic concept of the data access using deegree datastore classes

Datastrores are acting as a facaded to concrete data sources to provide a unique 'view' to different formats like ESRI shapefiles, Oracle Spatial or Postgis database. Each datastore includes three componets:

1. data accessing class(es): These classes are located in the org.deegree_impl.io package and its sub-packages and provide methods for reading and writing data from/to vendor specific data formats. This includes file formats as well as database accessing classes. Each database accessing class is derieved from the DBAccess class (Figure 12).
2. filter encoding classes (see below): These classes realizes the filter/query functionalty specified in the OGC Filter Encoding 1.0.0 specification. For database based datastores classes are offered that transforms a filter encoding expression to a SQL statement.
3. the datastore façade: For each supported datasource type a deegree offeres a datastore. A datastore implementes the DataStore interface (or extends a derieved class) to offer the same interface/access to different data formats.

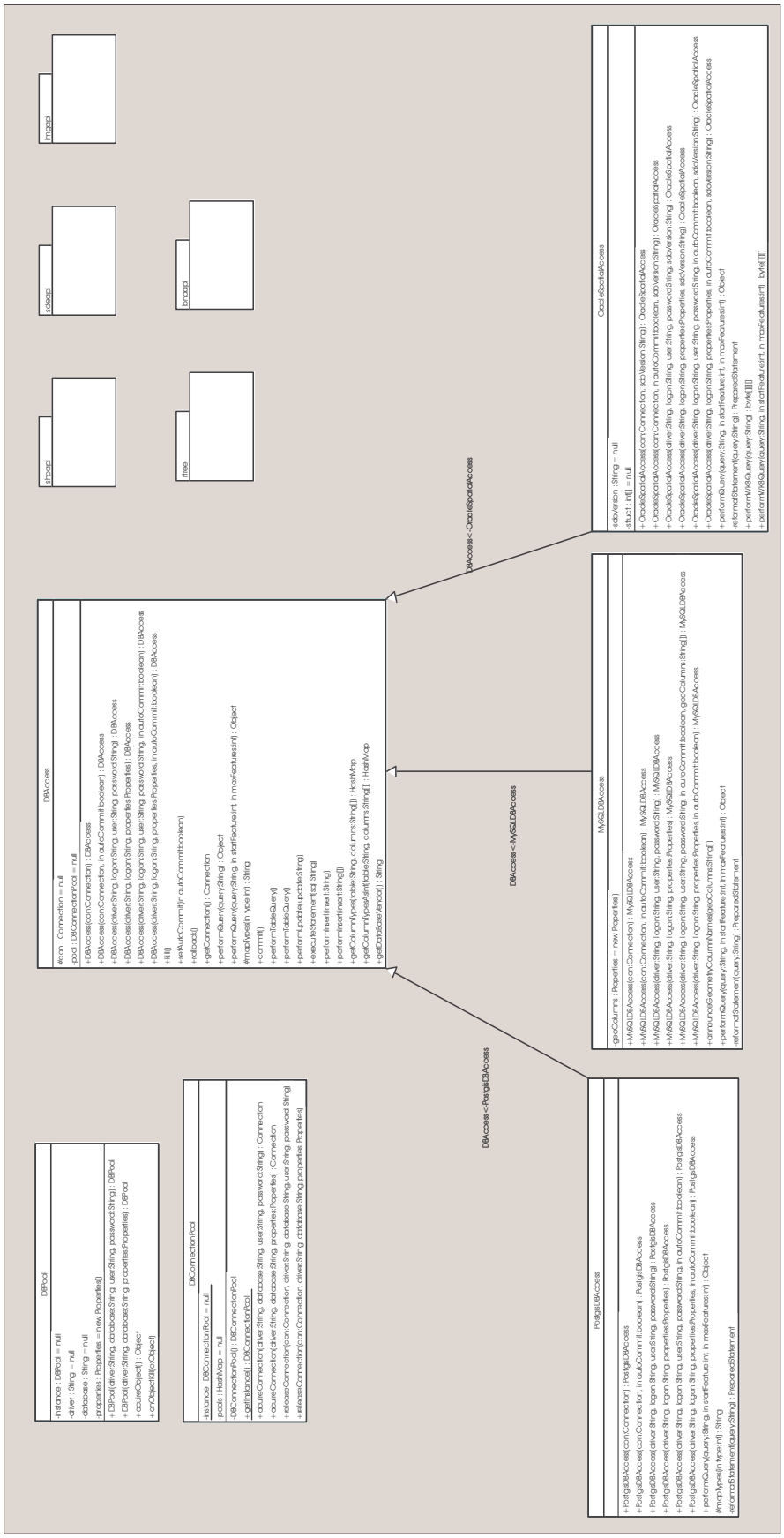



**Figure 12 org.deegree_impl.io package**

The construction of a deegree datastore shall be explained on the expample of the OracleDataStore.

As has been seen deegree web services are more or less indepent from data access. The data access for raster data is encapsulated into GridCoverage implementation the vector data access is encapsulated into datastores. The basic idea of the datastore concept is to have a unique facade for each data source format. So an application can easyly switch between different datasources just by accessing another datastore. Each concrete datastore have to implement the Datastore interface or must extend (and that is the usual way) the AbstractDatastore class. Beside some techincal method a datastore has to implement five data accessing methods corresponding to the data accessing and manipulating requestes defined at the OGC WFS specification. In the already implemented datastores of deegree each of these method (if implemented) starts a new thread and delegates the request processing to it for enableing one datastore instance to handle more then one request at the same time. So most datastores - one for each data format - defines their own classes for GetFeature, Transaction, DescribeFeatureType, GetFeatureWithLock and LockFeature. For some of these classes exists abstract parent classes located in the org.deegree_impl.services.wfs package.

If one like to implement a new datastore that shall be integrated into deegree it shall be located in its own package embedded in the org.deegree_impl.services.wfs package. For convenience a datastore package should be named like the datasource that shall be encapsulated (e.g. shape, sde, oracle ... ). The way how a datastore will be implemented is not specified by the deegree framework. But it is highly recommented to delegate request performing to independ threads.

At least a datastore shall support GetFeature and DescribeFeatureType because they are mandatory requests of the OGC WFS specification. For the same reason GML2 output format must be supported for GetFeature requests. In addition it is highly recommeted that a datastore is able to support the deegree FEATUTETYPE format which is just a serialized org.deegree.model.feature.FeatureCollection. The developer is free to implement output formatting class for any other desired format.

Output formats will be realized by offering formatting classes. A formatting class have to implement the DataStoreOutputFormat interface. The output formatting classes will be defined in the configuration document for a datastore and intitialized through java reflecting mechanism. So don't use named class initializations of output formatting classes within a datastore (see createResponse(HashMap map, String[] affectedFeatureTypes) method of AbstractGetFeature class).

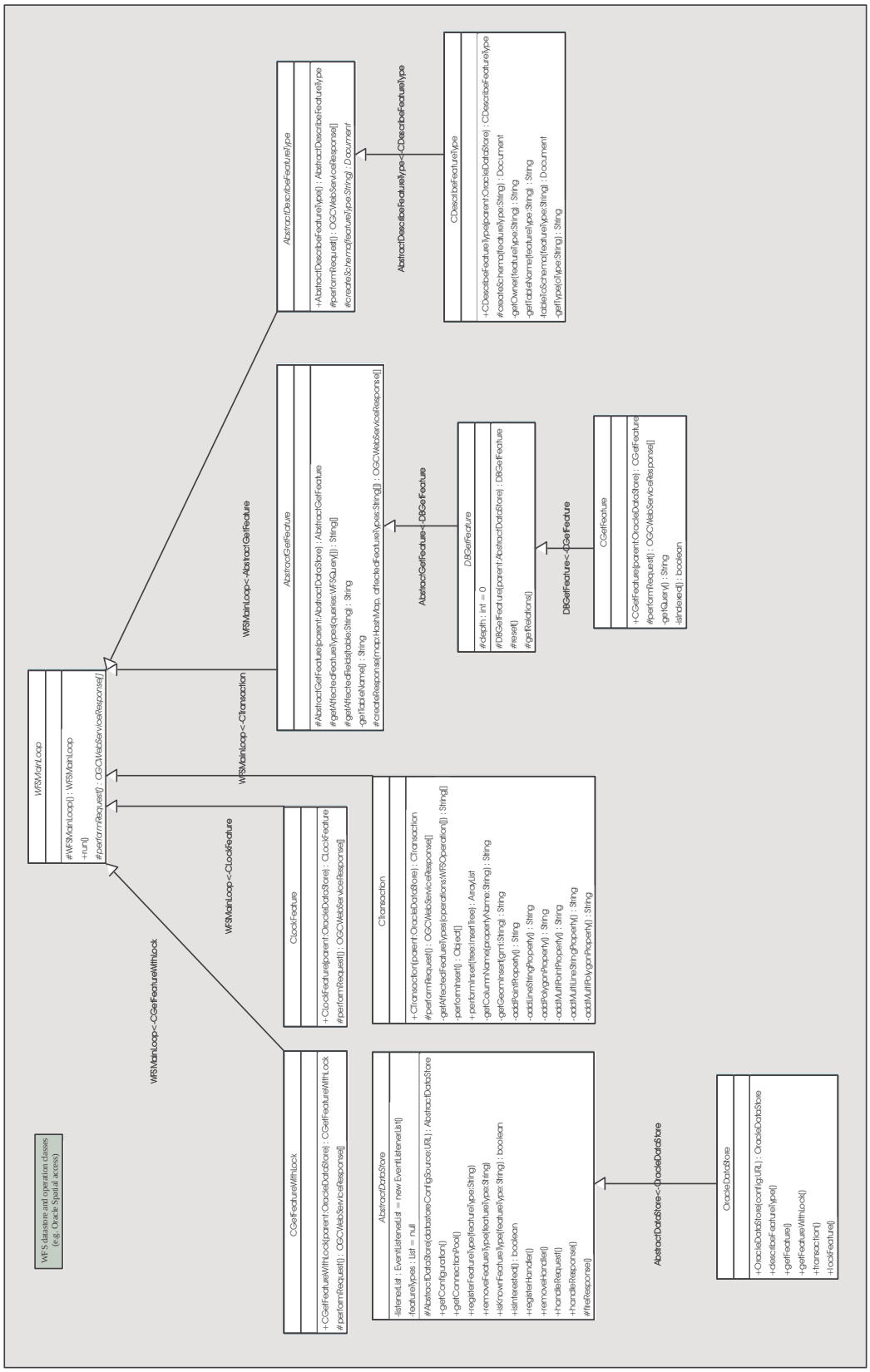
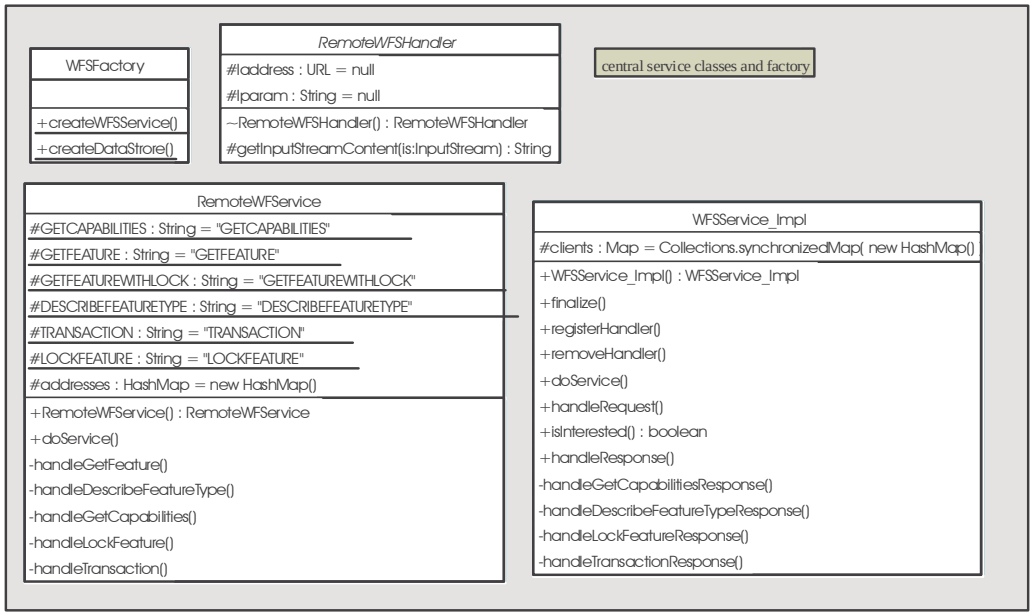



**Figure 13 Oracle datastore part I**



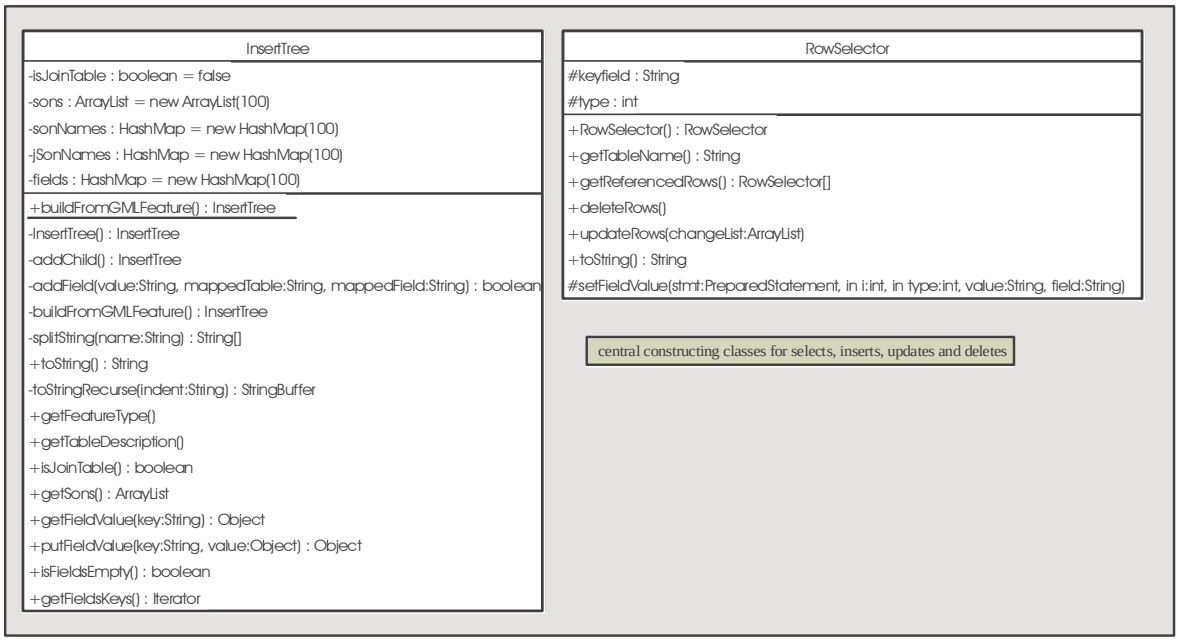




**Figure 14 Oracle datastore part II**

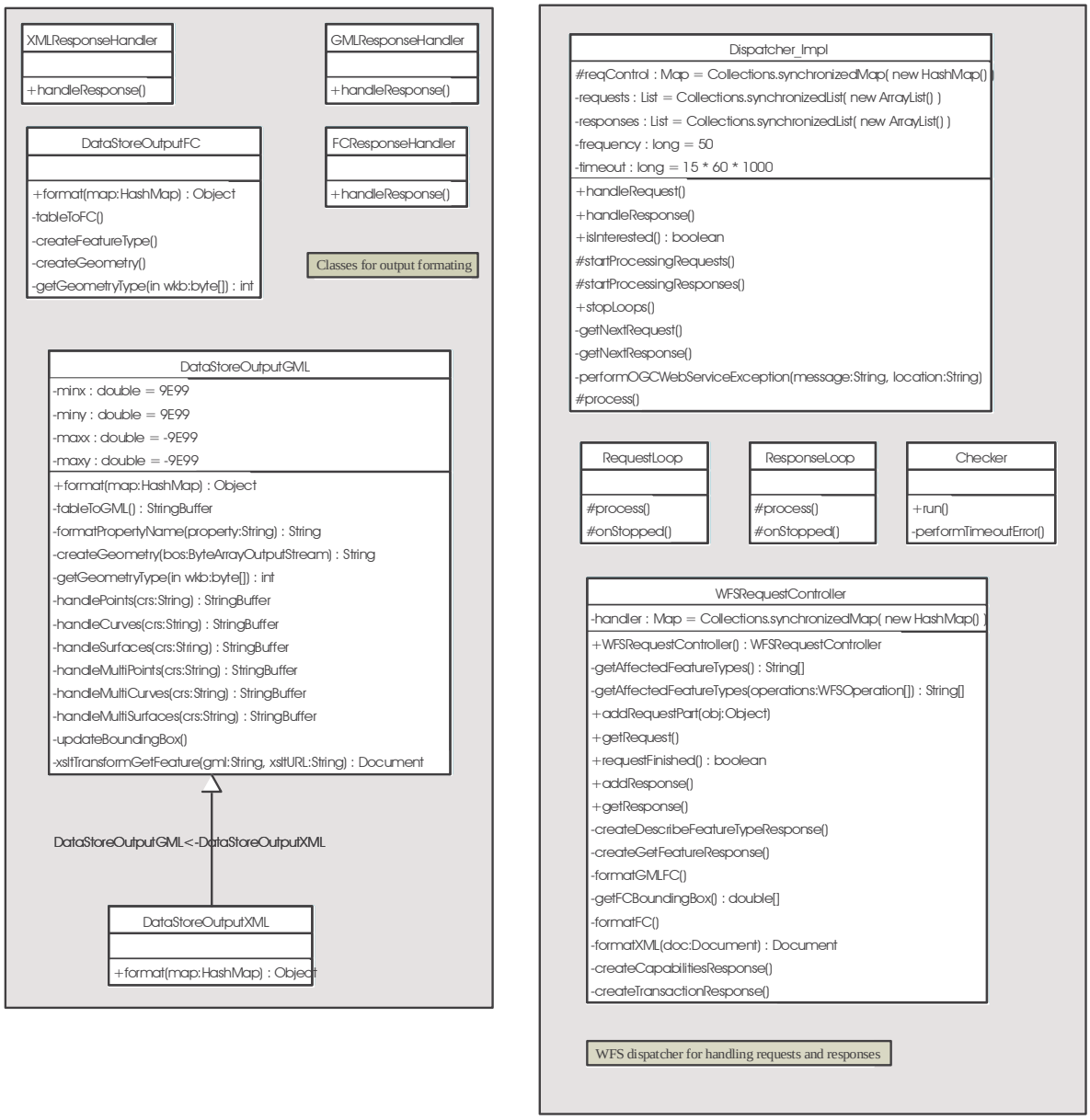



**Figure 15 Oracle datastore part III**

## Filterencoding

Filters have to conform to the OGC Filter Encoding Specification. They have two important uses in deegree:

1. They are used in the context of the WFS-datastores when selecting certain features; this is similar to specifying a WHERE-clause in SQL and it will most likely be converted into such a WHERE-clause (if the queried datastore is SQL-based).
2. The SLD (Styled Layer Specification) that the WMS uses allows to define styles that shall only apply to certain features. Furthermore it is possible to use Expression-elements (part of the Filter-DTD) to define dynamic drawing styles; for example one could specify that the radius of the circle that represents a city should be dependant on the number of inhabitants.

The second use differs from the first as deegree has to determine whether a feature matches the filter condition (or what the result of an expression is) rather than the datastore. This dynamic evaluation is performed using the evalute()-methods of the Filter classes.

Each datastore (for example OracleDatastore) must implement the methods that give access to the stored features, e.g. It must be able to process WFSGetFeature-Requests. The filters that are used in requests must be converted to a WHERE-clause that is suitable for this DBMS and the table schema used. This mapping information between feature properties and the database schema is given via an XML configuration file to the WFS. Usually, a datastore will use it’s own SQLBuilder implementation that extends the AbstractSQLBuilder.

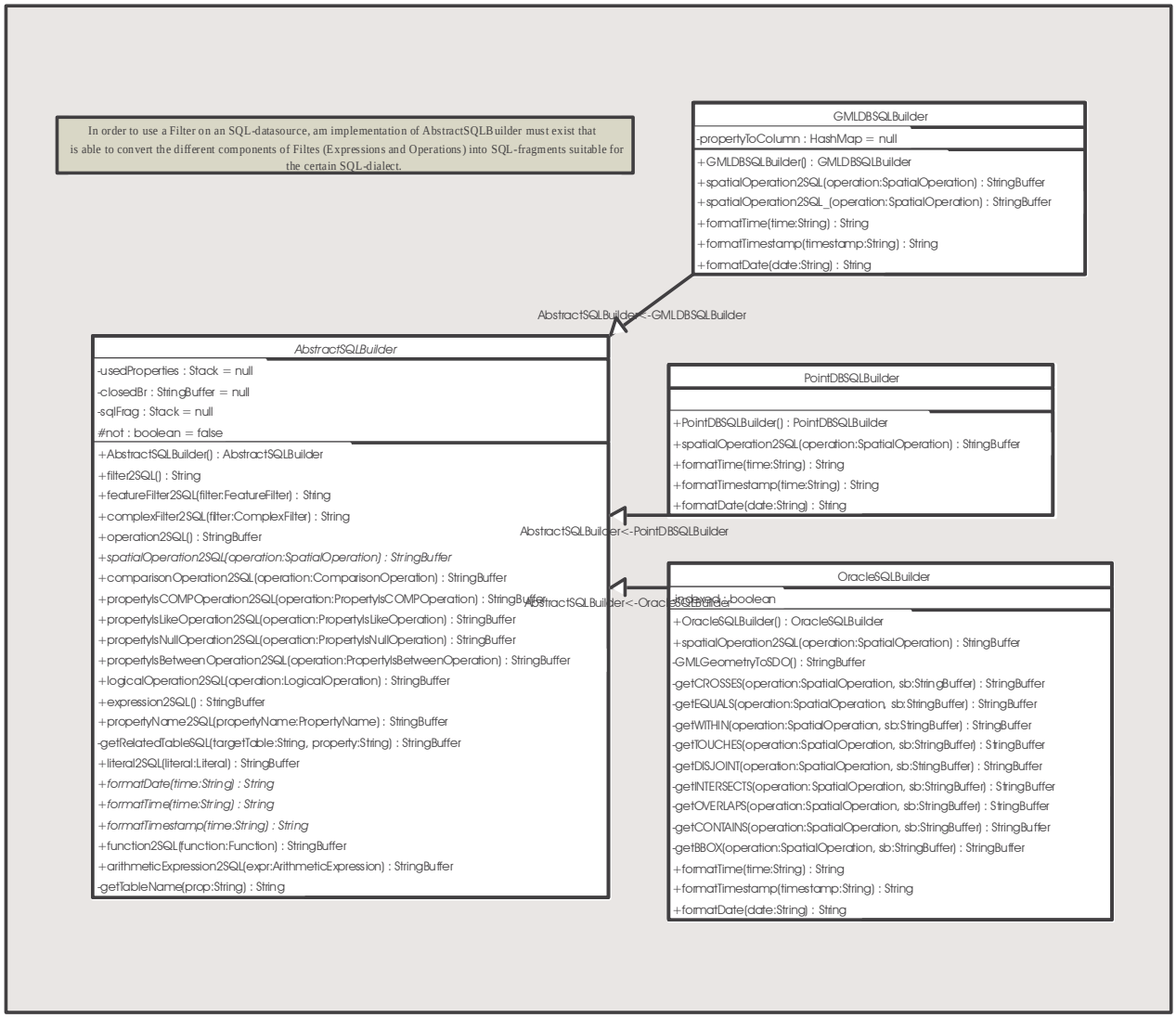



**Figure 16 Filterencoding - SQLBuilder**

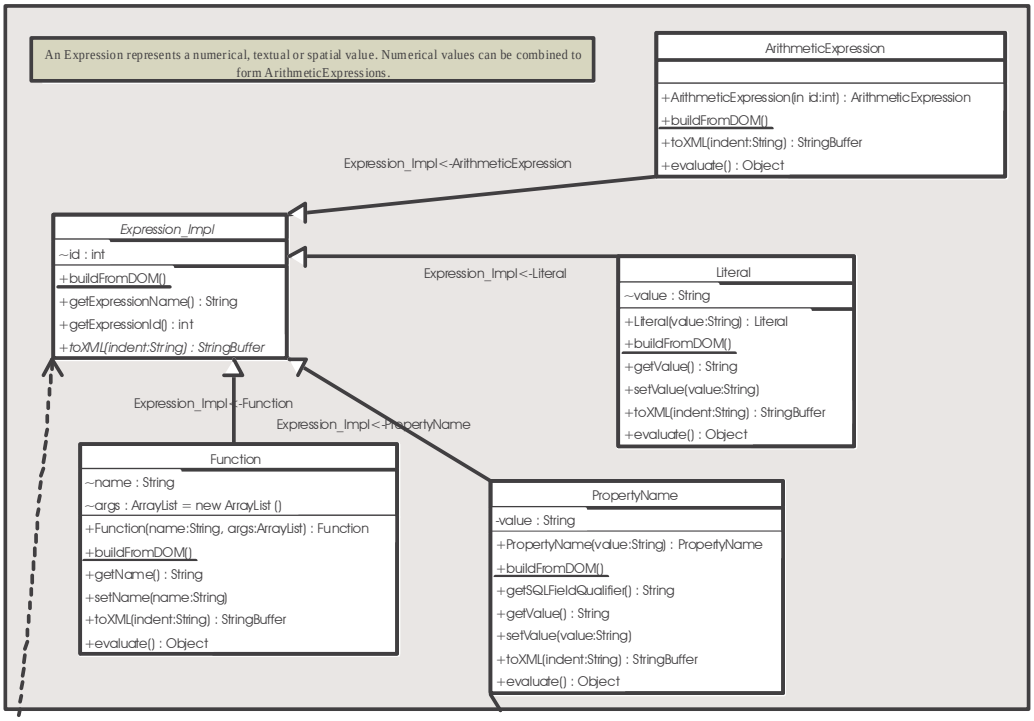



**Figure 17 Filterencding - Expressions**

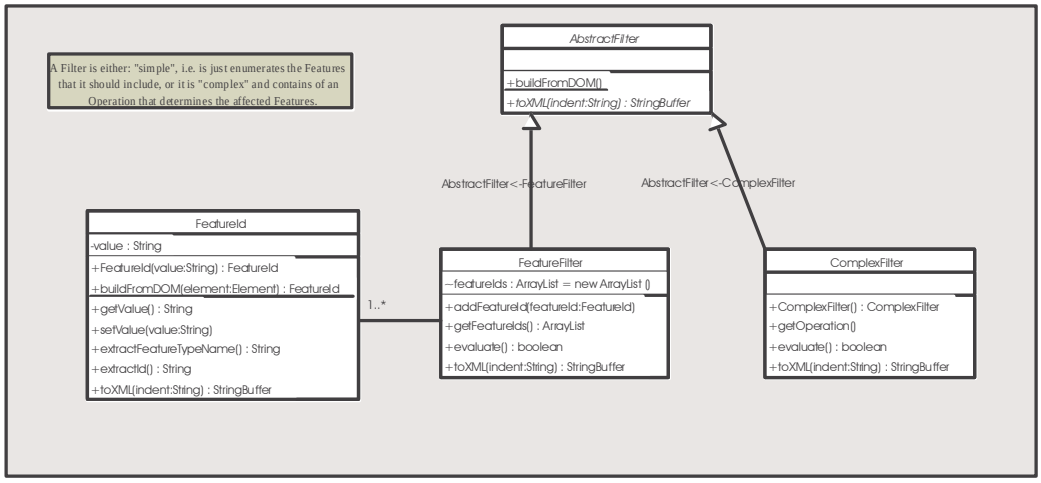
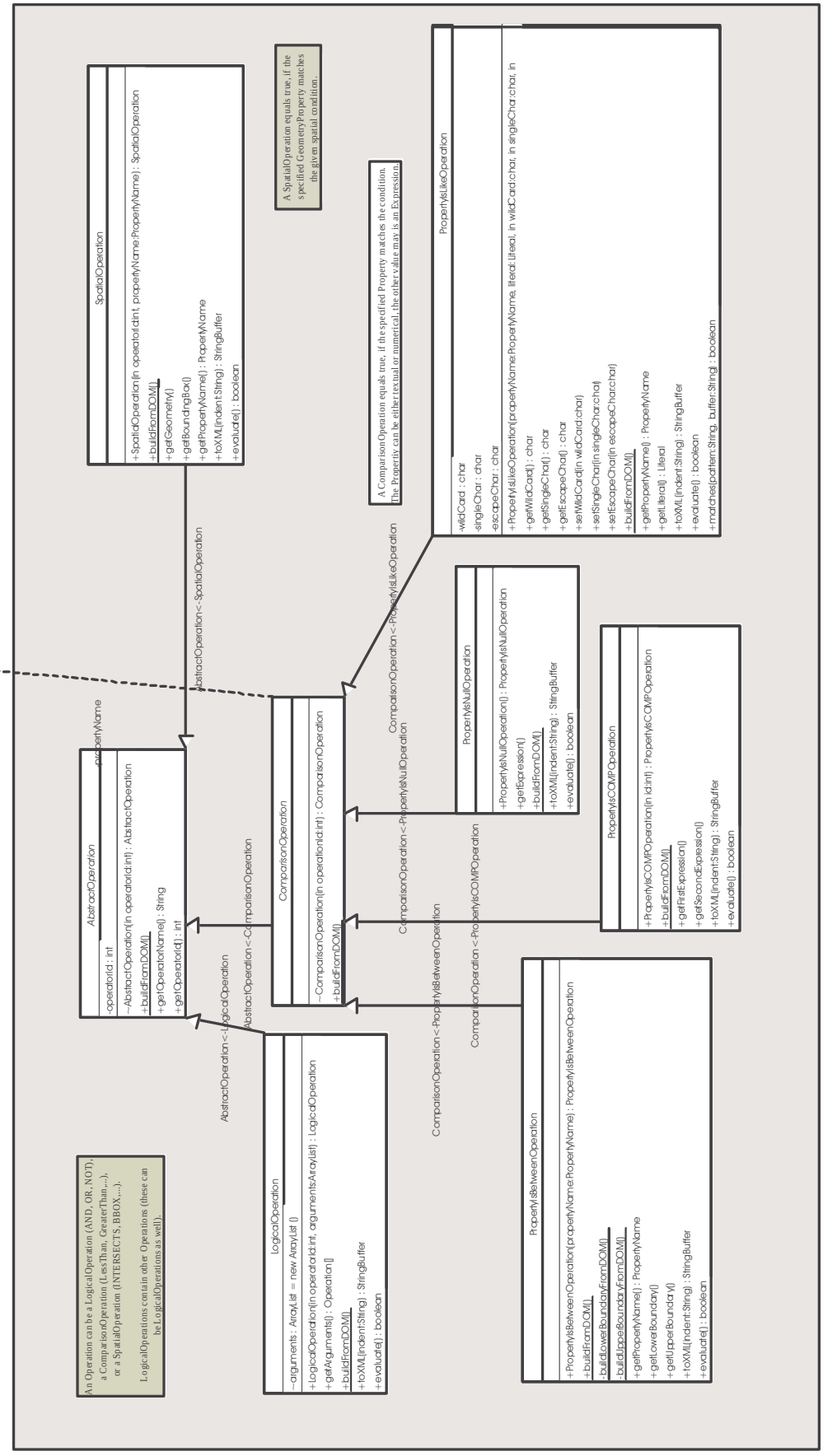



**Figure 18 Filterencoding Filter**

Figure on next page:

**Figure 19 Filterencoding Operations**


SpatialOperation
+SpatialOperation(in operatorId:int, propertyName:PropertyName) : SpatialOperation
+buildFromDOM()
+getGeometry()
+getBoundingBox()
+getPropertyName() : PropertyName
+toXML(indent:String) : StringBuffer
+evaluate() : boolean
A SpatialOperation equals true, if the specified GeometryProperty matches the given spatial condition.
PropertyIsLikeOperation
-wildCard : char
-singleChar : char
-escapeChar : char
+PropertyIsLikeOperation(propertyName:PropertyName, literal:Literal, in wildCard:char, in singleChar:char, in
+getWildCard() : char
+getSingleChar() : char
+getEscapeChar() : char
+setWildCard(in wildCard:char)
+setSingleChar(in singleChar:char)
+setEscapeChar(in escapeChar:char)
+buildFromDOM()
+getPropertyName() : PropertyName
+getLiteral() : Literal
+toXML(indent:String) : StringBuffer
+evaluate() : boolean
+matches(pattern:String, buffer:String) : boolean
A ComparisonOperation equals true, if the specified Property matches the condition. The Property can be either textual or numerical, the other value may is an Expression.
AbstractOperation
-operatorId : int
~AbstractOperation(in operatorId:int) : AbstractOperation
+buildFromDOM()
+getOperatorName() : String
+getOperatorId() : int
PropertyName
AbstractOperation<-SpatialOperation
ComparisonOperation
~ComparisonOperation(in operatorId:int) : ComparisonOperation
+buildFromDOM()
ComparisonOperation<-PropertyIsLikeOperation
PropertyIsNullOperation
+PropertyIsNullOperation() : PropertyIsNullOperation
+getExpression()
+buildFromDOM()
+toXML(indent:String) : StringBuffer
+evaluate() : boolean
ComparisonOperation<-PropertyIsNullOperation
PropertyIsCOMPOperation
+PropertyIsCOMPOperation(in id:int) : PropertyIsCOMPOperation
+buildFromDOM()
+getFirstXpression()
+getSecondExpression()
+toXML(indent:String) : StringBuffer
+evaluate() : boolean
ComparisonOperation<-PropertyIsCOMPOperation
AbstractOperation<-ComparisonOperation
LogicalOperation
-arguments : ArrayList = new ArrayList ()
+LogicalOperation(in operatorId:int, argumentsArrayList) : LogicalOperation
+getArguments() : Operation[]
+buildFromDOM()
+toXML(indent:String) : StringBuffer
+evaluate() : boolean
AbstractOperation<-LogicalOperation
PropertyIsBetweenOperation
+PropertyIsBetweenOperation(propertyName:PropertyName) : PropertyIsBetweenOperation
+buildFromDOM()
-buildLowerBoundaryFromDOM()
-buildUpperBoundaryFromDOM()
+getPropertyName() : PropertyName
+getLowerBoundary()
+getUpperBoundary()
+toXML(indent:String) : StringBuffer
+evaluate() : boolean
ComparisonOperation<-PropertyIsBetweenOperation
An Operation can be a LogicalOperation (AND, OR, NOT), a ComparisonOperation (LessThan, GreaterThan,...), or a SpatialOperation (INTERSECTS, BBOX,...).
LogicalOperations contain other Operations (these can be LogicalOperations as well).

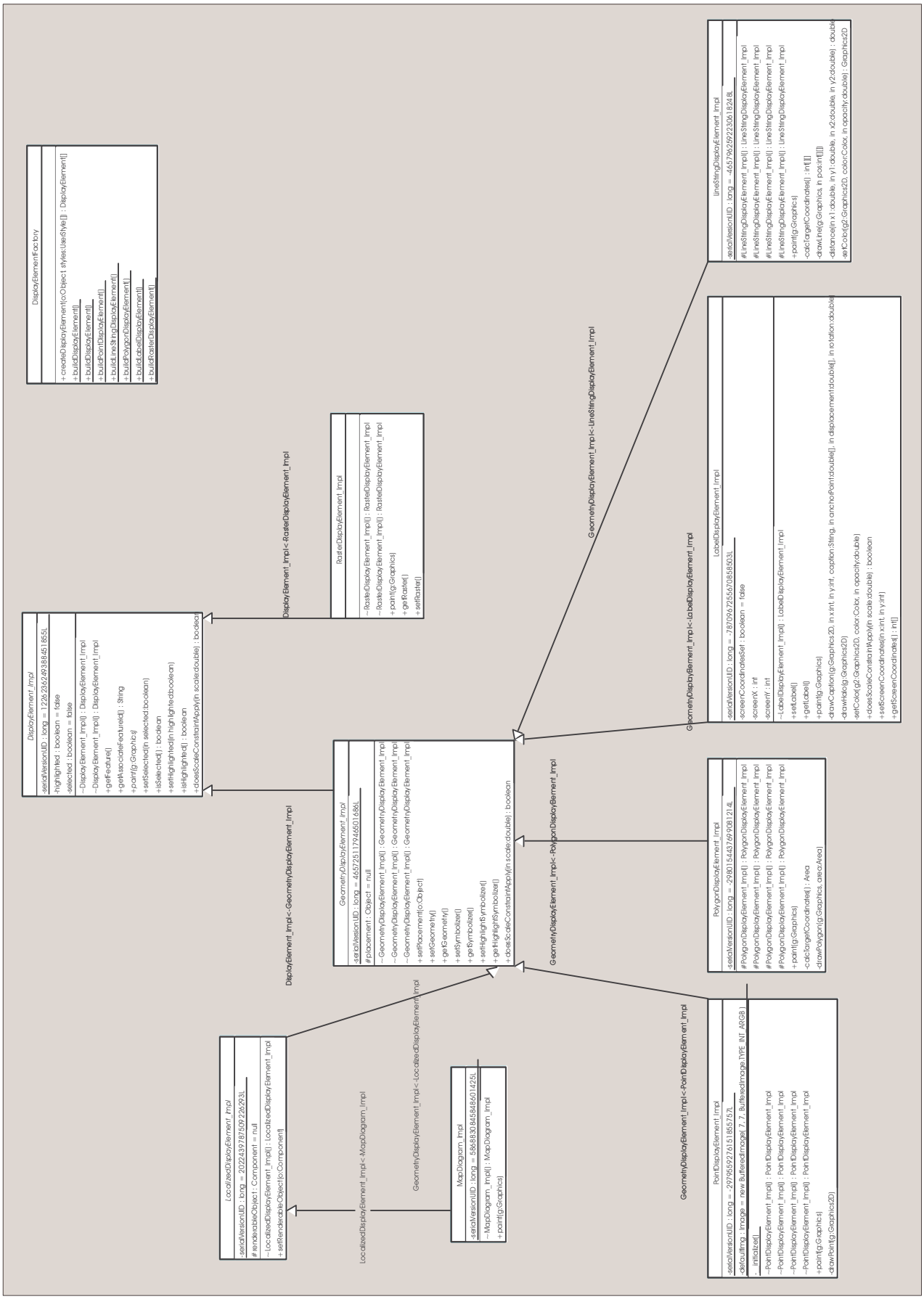
## DisplayElements

DisplayElements are a combination of a Geometry with a Style and for localized DisplayElements with an additional Object (not implemented yet).All DisplayElements are DisplayElementrived from a base class the DisplayElementclares the methods that are common to all DisplayElements. Especialy these are methods for painting the DisplayElements, accessing its underlying Feature and marking it as selected and/or highlighted.

Geometry DisplayElements in oposition to a Raster DisplayElement must contain a GM_Objectthat contains the information where to paint the DisplayElements. Raster DisplayElements are different because they just contain a GC_GridCoverage. This contains a GM_Envelope that enables the DisplayElements to calculate the position and size of the raster on the map.

Figure on next page:

**Figure 20 DisplayElements class diagram**


DisplayElementFactory
+ createDisplayElement(oObject styles:UserStyle[]) : DisplayElement[]
+ buildDisplayElement()
+ buildDisplayElement()
+ buildPointDisplayElement()
+ buildLineStringDisplayElement()
+ buildPolygonDisplayElement()
+ buildLabelDisplayElement()
+ buildRasterDisplayElement()
DisplayElement_Impl
-serialVersionUID : long = 1226.2362497388451855L
#highlighted : boolean = false
-selected : boolean = false
~DisplayElement_Impl() : DisplayElement_Impl
~DisplayElement_Impl() : DisplayElement_Impl
+getFeature()
+getAssociateFeature4j() : String
+paint(g:Graphics)
+setSelected(in selected:boolean)
+isSelected() : boolean
+setHighlighted(in highlighted:boolean)
+isHighlighted() : boolean
-doesScaleConstraintApply(in scale:double) : boolean
DisplayElement_Impl<-RasterDisplayElement_Impl
RasterDisplayElement_Impl
~RasterDisplayElement_Impl()
~RasterDisplayElement_Impl()
+paint(g:Graphics)
+getRaster()
+setRaster()
GeometryDisplayElement_Impl
-serialVersionUID : long = 4657251179465016861
#placement : Object = null
~GeometryDisplayElement_Impl()
~GeometryDisplayElement_Impl()
~GeometryDisplayElement_Impl()
+setPlacement(in oObject)
+setGeometry()
+getGeometry()
+setSymbolizer()
+getSymbolizer()
+setHighlightSymbolizer()
+getHighlightSymbolizer()
+doesScaleConstraintApply(in scale:double) : boolean
GeometryDisplayElement_Impl<-LineStringDisplayElement_Impl
LineStringDisplayElement_Impl
-serialVersionUID : long = -4657962992230618248L
#LineStringDisplayElement_Impl()
#LineStringDisplayElement_Impl()
#LineStringDisplayElement_Impl()
#LineStringDisplayElement_Impl()
#LineStringDisplayElement_Impl()
+paint(g:Graphics)
-calcTargetCoordinates() : int[][]
-drawLine(g:Graphics, in posint[][])
-distance(in x1:double, in y1:double, in x2:double, in y2:double) : double
-setColor(g2:Graphics2D, colorColor, in opacity:double) : Graphics2D
GeometryDisplayElement_Impl<-LabelDisplayElement_Impl
LabelDisplayElement_Impl
-serialVersionUID : long = -7870967255670858803L
-screenCoordinatesSet : boolean = false
-screenX : int
-screenY : int
~LabelDisplayElement_Impl() : LabelDisplayElement_Impl
+setLabel()
+getLabel()
+paint(g:Graphics)
-drawCaption(g:Graphics2D, in x:int, in y:int, caption:String, in anchorPoint:double[], in displacement:double[], in rotation:double)
-drawHalo(g:Graphics2D)
-setColor(g2:Graphics2D, color:Color, in opacity:double)
+doesScaleConstraintApply(in scale:double) : boolean
+setScreenCoordinates(in x:int, in y:int)
+getScreenCoordinates() : int[]
DisplayElement_Impl<-PolygonDisplayElement_Impl
PolygonDisplayElement_Impl
-serialVersionUID : long = -2980154437690811214L
#PolygonDisplayElement_Impl() : PolygonDisplayElement_Impl
#PolygonDisplayElement_Impl() : PolygonDisplayElement_Impl
#PolygonDisplayElement_Impl() : PolygonDisplayElement_Impl
#PolygonDisplayElement_Impl() : PolygonDisplayElement_Impl
+paint(g:Graphics)
-calcTargetCoordinates() : Area
-drawPolygon(g:Graphics, area:Area)
GeometryDisplayElement_Impl<-PointDisplayElement_Impl
PointDisplayElement_Impl
-serialVersionUID : long = -2979559276161855757L
-defaultImg : Image = new BufferedImage(7, 7, BufferedImage.TYPE_INT_ARGB)
-initialize()
~PointDisplayElement_Impl()
~PointDisplayElement_Impl()
~PointDisplayElement_Impl()
~PointDisplayElement_Impl()
+paint(g:Graphics)
-drawPoint(g:Graphics2D)
DisplayElement_Impl<-GeometryDisplayElement_Impl
LocalizedDisplayElement_Impl
-serialVersionUID : long = 2022439787509226293L
#renderableObject : Component = null
~LocalizedDisplayElement_Impl() : LocalizedDisplayElement_Impl
+setRenderableObject(oComponent)
LocalizedDisplayElement_Impl<-MapDiagram_Impl
GeometryDisplayElement_Impl<-LocalizedDisplayElement_Impl
MapDiagram_Impl
-serialVersionUID : long = 5868830845848601425L
~MapDiagram_Impl() : MapDiagram_Impl
+paint(g:Graphics)

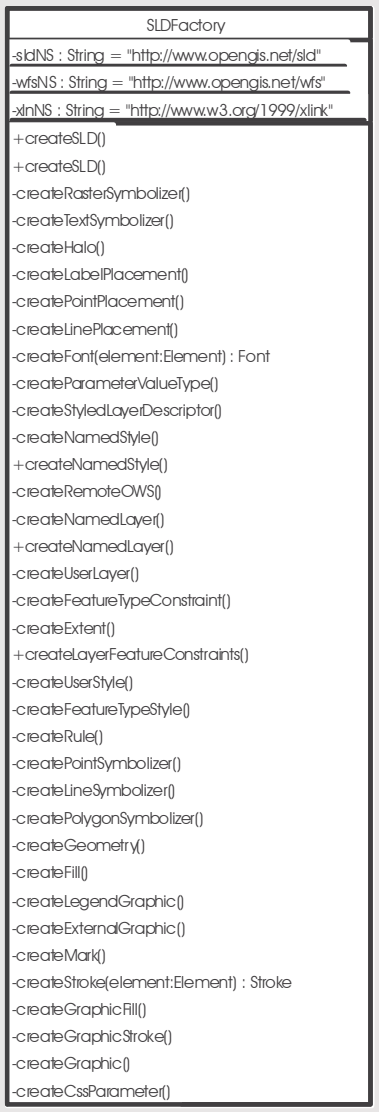
## Styled Layer Descriptor

StyledLayerDescriptor documents are used to specify how geometric information is to be painted (colors, stroke-width,...). The WMS uses a StyledLayerDescriptor-document with only one layer as a style definition file. Because the layers of the WMS are defined in a capabilites document (which is a separate file), the layer information of the styles-SLD is not needed. It merely defines styles that can be referenced in the WMS-requests by their names.

In the deegree-WMS, the features that are to be visualized by a certain rule are used to built Display-Elements that encapsulate the rule's Symbolizer and the feature. When the DisplayElement is signalled to paint itself, it uses the style information from this Symbolizer.

Figure on the next page
**Figure 21 Styled Layer Descriptor part I**

SLDFactory contains methods to create an object representation of an XML document that validates against the SLD.dtd. There are create()-methods for the elements that are defined in the DTD. These expect the Element-Node as argument and produce a corresponding Java-object with the same name. Nested elements are transformed into java objects automatically, so a call to createSLD() produces a tree structure that corresponds to the document instance. If the provided document has syntactic (or certain semantic) errors, an XMLParsingException is thrown.





The StyleFactory on the right offeres several convenience methods for creating simple and complex style on application run time without requirement for defining a SLD(XML) document.

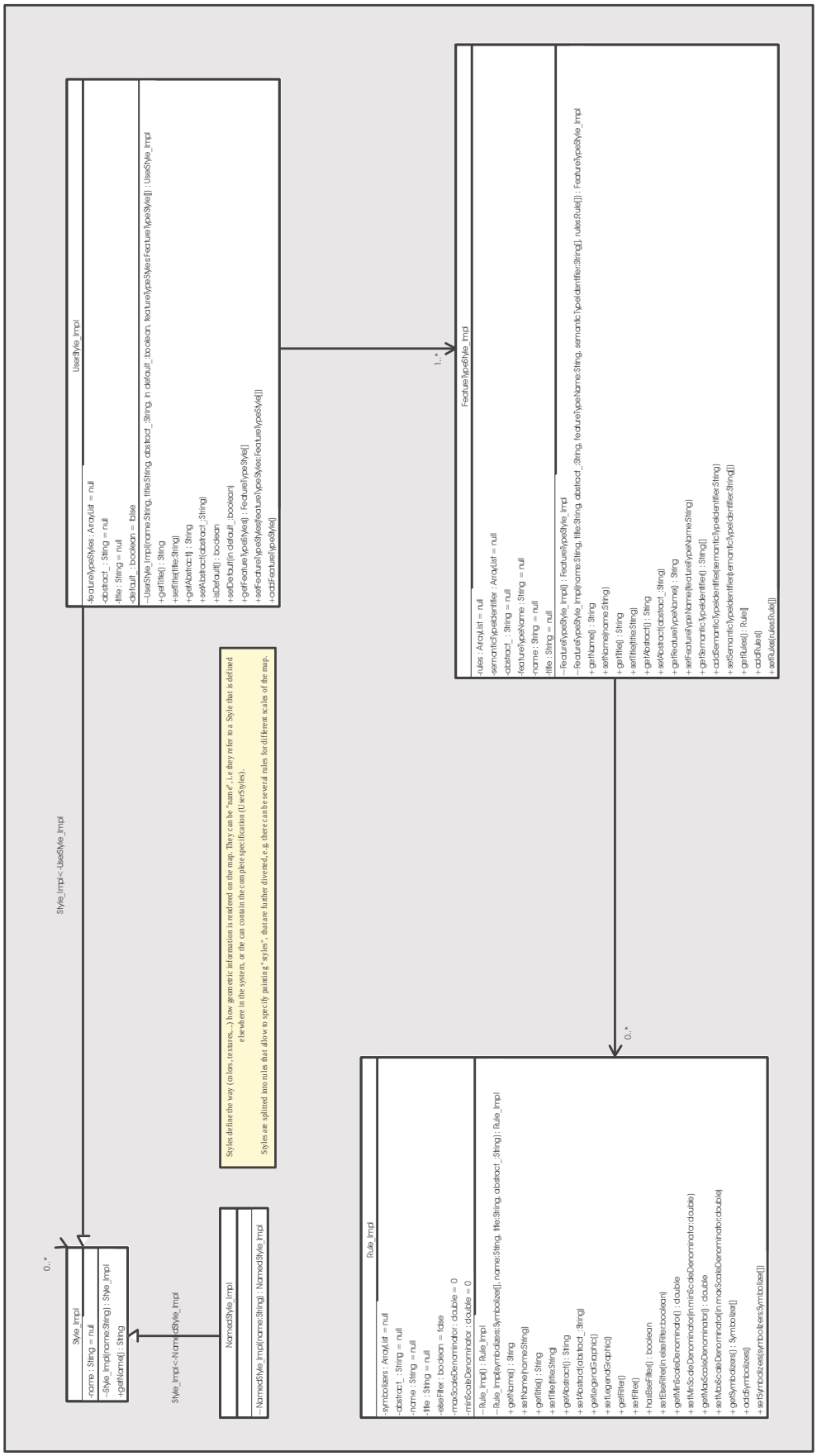



**Figure 22 Styled Layer Descriptor part II**

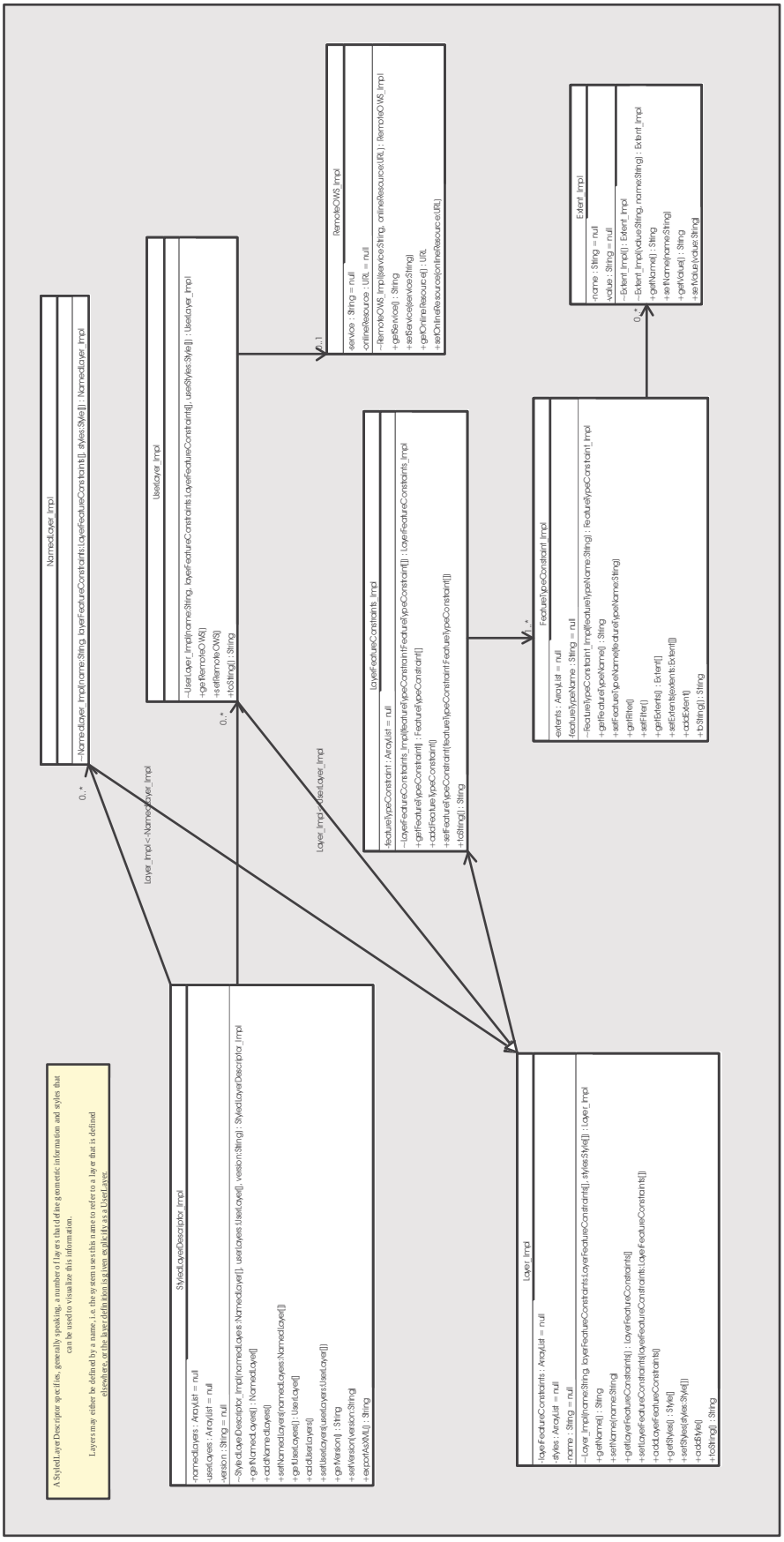



**Figure 23 Styled Layer Descriptor part III**