deegree developer guide

(Bonn, September 27th 2006)

lat/lon GmbH
Aennchenstr. 19
53177 Bonn
Tel. ++49 228 18496-0

Dept. of Geography
Bonn University
Meckenheimer Allee 166
53115 Bonn
Tel. ++49 228 732098

Change log

Date	*Person*	*Description*
6.8.2006	A. Poth	Document renamed into deegree developer guide; Change log table added; Introduction slightly changed; Chapter 'Constants' added; Usage of JDK 5 based capabilities suggested; References to new deegree utility classes/methods added; Chapter about dealing with loggers enhanced; Chapter describing how to deal with constants added; Chapter about avoiding useless code added
25.8.2006	A. Poth	Properties and xml/xsl header added
28.8.2006	A. Poth	Chapter 'package depencies' added
30.8.2006	A. Poth	Chapter 'Strings' enhanced
11.9.2006	A. Poth	Chapter XML added
24.9.2006	A. Poth	Chapter Messaging/Internationalization added
2.10.2006	R. Bezema	logDebug -> logError and small lingual corrections

Table of Contents

1 Introduction

This document provides a style guide for deegree developers and contributors. The major intention is to have a general code style for all classes of deegree so everybody will be able to read and understand code written by somebody else.

As its name says: this is a guide and not a dogmatic canon of rules. Only a few rules are not open to discussion and possible change (e.g. Class headers and footers). So you are free to make some changes if you prefer other styling. But consider that if these changes are going too far we won't be able and willing to review and consider your code within the deegree framework. A lot of rules and hints given in this document can be defined as formatting rules within current IDEs (a formatting configuration for eclipse can be found in the deegree CVS). Doing this will help you writing and all others reading deegree code.

2 Naming

For all classes, variables, method names and comments English is used as sole language. Package, class, interface, method and variable names shouldn't be much longer than 20 characters (this rule may be broken if names are given by external specifications).

- *Package names:* A name of a package always will just use lower case characters and does not use '_' or language specific characters like 'ä' or 'é'. The name of a package should give a hint what it includes.
- *Class/Interface names:* The name of a class should give a hint about its function/usage. If a class is directly related to an OGC or ISO specification its name should reflect the part of the specification it is encapsulating; e.g. GetFeature (GetFeature request from OGC WFS specification); UserLayer (part of the OGC SLD specification) etc.. If specifications use prefixes for objects (like a number of ISO specs do) they will be skipped (e.g. deegree.model.spatialschema package). Do not use '_' or language specific characters like 'ä' or 'é' in class names. Only use latin letters and arabic digits. If a class name contains an abbreviation like OGC, WFS, ISO or XML these will always be written in upper case characters.

  Abstract classes should use 'Abstract' as prefix (e.g. `AbstractHandler`) as long as the name is not given by an external specification. If a class implements an interface or an abstract class and it is very probable that there will be exten-

sions/specializations of it in the future such a class may use the prefix 'Default' (e.g. `DefaultGetMapHandler`).

- *Method names*: Always start a method name with a lower case character and a verb. A method 'does' something, so its name shall describe the thing it does, e.g. `calculateArea()`, `addValue()`, `transformCoordinates()` etc.. Accessor methods always begins with 'get' (except for boolean variable, which use 'is' or 'has' instead); mutator methods start with 'set', 'add', 'put' or 'remove' (see java style guide for bean methods). In most cases accessor methods expect none or one argument; set, add and remove methods expect one argument and put-methods expect two arguments.
- *Variable names*: Instance variables and variables within method scope always begin with a lower case character. The same applies to arguments passed to a method. Class variables (constants) should be written completely in upper case. Variable names should always provide the reader with a hint on their purpose; loop variables ('i', 'j',...) may be exceptions to this rule.

3 Best practice

3.1 General rules

- A method should not exceed 40-60 lines length (20-40 lines are better). If a method exceeds this limit you should think about splitting it into two or more methods. Like a method, a class shouldn't exceed a limited size. This is around 200-300 lines of code (without header and footer). If a class exceeds this length one should think of splitting it.

  To both rules there exist exceptions where it doesn't make a lot of sense to split methods and classes, but *these are exceptions and not the usual case!*
- Avoid using 'break' and 'continue' in loops.
- Use Java Generics if possible and useful use JDK 1.5 specific syntax.
- No need to re-invent the wheel. For a lot of problems there already exists a solution in deegree or in other open source frameworks. For more general functions see
  - `org.deegree.framework.util` for several utility classes
  - `org.deegree.framework.util.xml` for XML handling methods
  - `org.deegree.tools` for tools (applications)
  - `org.deegree.framework.concurent` for threading

An additional framework that offers implementations for a variety of problems is Apache Jakarta Commons, which is used throughout deegree [http://jakarta.apache.org/commons/].

- Avoid using methods and classes that are marked as deprecated.
- If you need additional functionality not provided by deegree think of extending an existing class instead of writing a completely new one.
- Define interfaces and abstract classes if it's probable that more than one implementation of a function will be needed.
- Use established design patterns if possible and useful.
- Do not include configuration details, path and web address settings in the code; preferably use (XML) configuration files.
- Exclude any kind of messages into external properties-files
- If you need to define a name for something use `org.deegree.datatypes.QualifiedName` instead of simple strings.
- Be careful using nested statements. `new URL( getName() );` is OK but avoid expressions like this:

```
sensorMetadata.add(new SensorMetadata(((Identifier[]) identifier-
s.toArray(new Identifier[identifiers.size()])), ((Classifier[])
classifiers.toArray(new
Classifier[classifiers.size()])),hasCRS,((LocationModel[]) loca-
tionModels.toArray(new LocationModel[locationModels.size()])), de-
scribedBy, attachedTo,((Product[]) measures.toArray(new
Product[measures.size()])));
```

as nobody will understand that.

3.2 XML-Handling

- If you need more than one XML representation of a class or a document think of using XSLT instead of writing a java class for each representation.
- If you write a parser for a configuration, a capabilities document or something similar extend one of the following classes:
  - `org.deegree.framework.xml.XMLFragment`,
  - `org.deegree.ogcbase.OGCDocument`,
  - `org.deegree.ogcwebservices.getcapabilities.OGCCapabilitiesDocument`
  - `org.deegree.owscommon.OWSCommonCapabilitiesDocument`

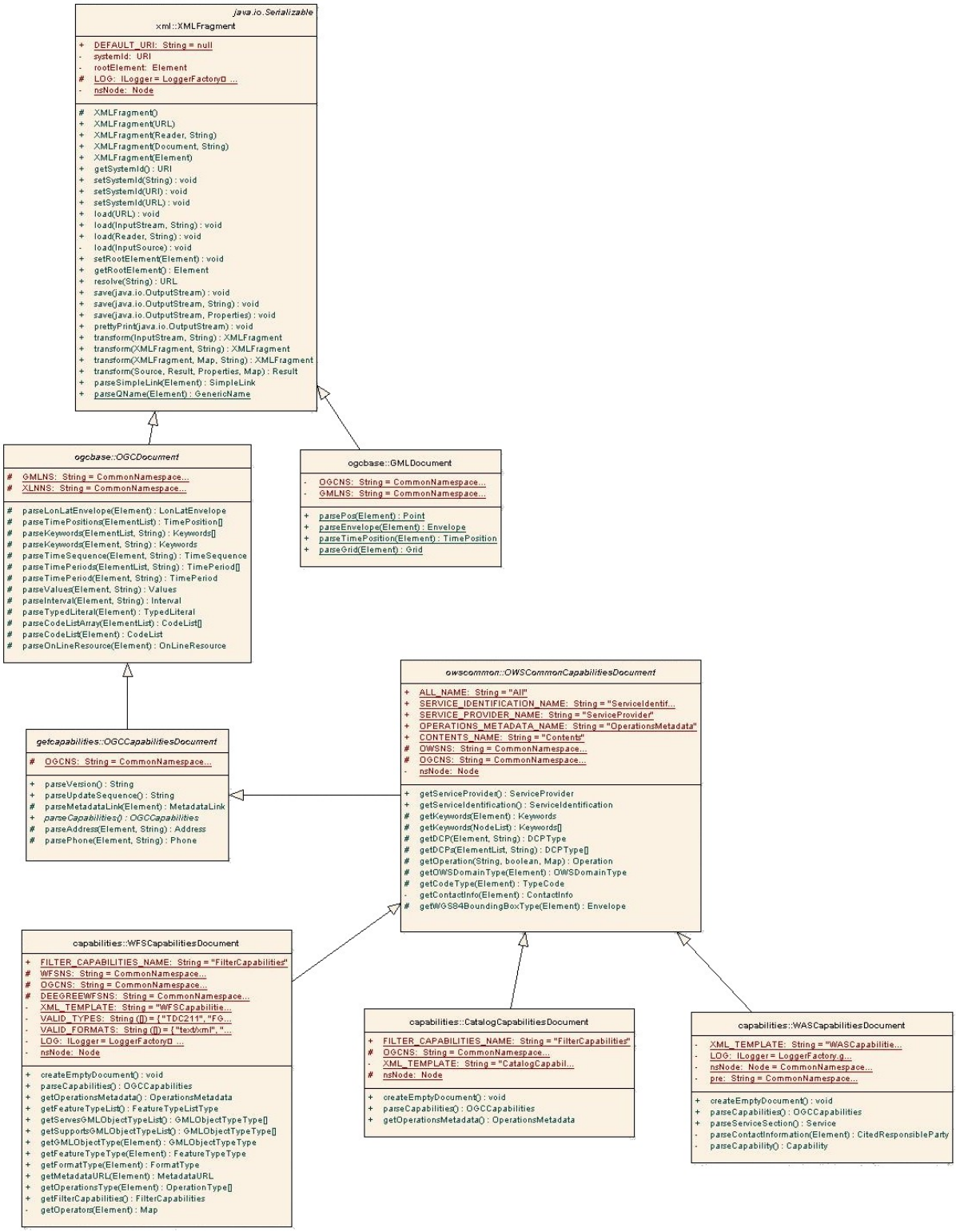

Illustration 1: Example class hierarchy using deegree XML document classes

3.3 Strings

- Avoid using string concatenation. E.g:

  ```
  String s = "aString” + "bString”+ "cString”;
  ```

  Use a StringBuffer instead (exceptions to this rule can be made when constructing an exception message or when just two string will be concatenated).

  ```
  StringBuffer sb = new StringBuffer( 100);
  sb.append( "aString” );
  sb.append( "bString” );
  sb.append( "cString” );
  ```

- If using a StringBuffer initialize it with the expected string length:

  ```
  StringBuffer sb = new StringBuffer( 20000 );
  ```

- As an alternative to StringBuffers you may use StringTools.concat(...) from package org.deegree.framework.util that is a bit easier to handle
- Use char: `char c = 'a';` instead of a String having just one character:

  ```
  String s = "a";
  ```

- If an explicit character encoding is required or reasonable (e.g. `URLDecoding/Encoding`) use

  ```
  org.deegree.framwork.util.CharsetUtility.getSystemCharset().
  ```

- If evaluation if a variable contains an empty String use

  ```
  if ( s != null && s.length() > 0 )
  ```

  instead of

  ```
  if ( !"".equals( s ) )
  ```

  because the second causes creation of a new instance of String that allocates memory and must be removed by the garbage collector.

3.4 Lists / Arrays

- Use `ArrayList` and `HashMap` instead of `Vector` and `Hashtable`.
- If you need a synchronized List or Map use

  ```
  Collections.synchronizedList( new ArrayList() );
  Collections.synchronizedMap( new HashMap() );
  ```

- If a List or a Map will take a lot of objects initialize it with the expected size

  ```
  List list = new ArrayList( 1000 );
  Map map = new HashMap( 1000 );
  ```

- Avoid using long strings as keys for Maps (this is not just a bottleneck it may cause some unexpected effects).
- Return typed lists and maps (e.g. List<String> or Map<String, Image>) instead of arrays whenever useful.

  This means that whenever the length of the returned array is fixed independently from the context the method it is called by and its length will not be changed a array shall be returned. Otherwise a typed list shall be used. E.g. org.deegree.model.spatialschema.Position defines a method named getAsArray(). This method returns an array because the returned array always has a length of three (not just because it is named like this). On the other hand org.deegree.security.owsproxy.OperationParameter method getValues() returns a List<String> because the amount of returned Strings is not known before.

3.5 Exceptions

Don't ignore exceptions! If an exception can not be handled by the class/method where it's thrown or from which it is received it shall be forwarded. In a few exceptional cases it is valid to do nothing if an exception occurs (e.g. the most simple way to determine if a string represents a number or not is trying to convert it, if an exception occurs you know it isn't a number (which may be valid)). In this case indicate with a comment that you're aware of this. Avoid throwing `java.lang.Exception` (a few older deegree classes may still be doing this).

Especially when writing classes/methods in the context of OGC-Specifications a few rules apply.

- In most cases `org.deegree.ogcbase.OGCException` or if an exception is assigned to a OGC web service `org.deegree.ogcwebservice.OGCWebServiceException` should be used or extended.
- Some OGC specifications define a specific message/code for some exceptions. Make sure you're using these messages/codes when throwing such an exception.
- On top level of OGC web services just OGCWebServiceExceptions are accepted to be thrown (this is required to create the XML representation of an exception as defined by the OGCCommons specification). So you must encapsulate low-level exceptions with this. (This should be done as late as possible/useful.)

3.5.1 Exceptions and design

It's usually preferable to use standard exceptions instead of your own. Defining your own exception must offer you an advantage. If you use your own exceptions don't use them in an application to trigger the control flow.

Declaring an exception

When declaring Exceptions in a throws clause keep them stable. Adding or removing an exception causes significant changes to the calling code. So try to use a small number of high-level exceptions types instead of many individual low-level exception types: it prevents the throws clause from changing every time the method changes.

Throwing an exception

When you throw an exception be aware of the differences between application and run-time exceptions. An application exception is an instance of what is called a checked exception, while for example an IllegalArgumentException is a run-time exception.

The basic rule is that the caller of a method that throws a checked exception must handle the exception in a catch clause or further propagate it. Checked exceptions are a mechanism for requiring the programmer to deal with exceptional conditions that are raised. By contrast, this treatment is not required for run-time exceptions. Calling a method which throws an run-time exception, which is not caught, causes that the current thread (and the program) terminates. This can happen because a method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

In general, checked exceptions are used for recoverable errors, such as a non-existent file or a wrong input parameter. Run-time exceptions, by comparison, are used for programming errors. If you're writing a file browser, for example, it's quite plausible that a user might specify a non-existent file as part of some operation. But an empty string passed as a file name possibly indicates a non-recoverable programming error, something that is not supposed to happen.

A third kind of exception is a subclass of Error and is, by convention, reserved for use by the Java virtual machine. OutOfMemoryError is an example of this type of exception.

3.6 Logging

Use the formal logging policy for developing within the deegree framework.

Normally you have a hodge podge of System.out statements, System.err statements and the occasional printStackTrace() within your application.

Try to avoid this and use the logging functionality provided with the framework in package:

`org.deegree.framework.log`

To initialise the logger, each class should have a static variable "logger" created like so :

```
private static Logger LOG = LoggerFactory.getLogger(AClass.class);
```

To actually log statements, you then just need to do any of the following (in order of priority, lowest to highest):

```
logger.logDebug("A log statement");
logger.logInfo("A log statement");
logger.logWarning("A log statement");
logger.logError("A log statement");
```

All methods are overloaded with the parameter set: (String message, Throwable exception).

The log level Error sends automatically an email to the system administrator with all system entries.

Just log information that is worth of being logged. Do not log code comments or similar things. Logging a few hundreds of pieces of information (INFO-mode) is just confusing and not helpful.

Choose the right log-level! A coarse rule is:

- logDebug: technical information for developers and people who have a deeper understanding of the deegree code. This information can be used to understand why something does not work like it should.
- logInfo: information for users/administrators of an application/service. e.g. Printing the used deegree version; which feature type has been loaded etc..
- logWarning: use this level if something isn't like it should but not critical for the application. e.g. a point is outside the defined range of a CRS but can be handled.
- logError: use whenever an exception occurs

3.7 Constants

Use constants (class variables marked as final) whenever a value is known before starting a program and does not change during program execution. Define constants in classes where they will be used. If a constant will be used by more than one class think of using a common superclass. If a constant is used by several classes in different contexts extract the constants into a separate class (e.g. org.deegree.datatypes.Types). Using i special class for constants should be the exception; such a class shall not be initialisable and so shall have a private constructor.

Constants – as classes – shall have the lowest visibility as possible; a constant used just by the class where it is defined does not have to be 'public'.

To increase readability constants shall be ordered thematically and within a 'thematic section' alphabetically.

If constants are used as keys or marker use integer values instead of strings.

3.8 Threading / parallel processing

Parallel processing and time limit processing of tasks are centralized in org.deegree.framework.concurent package. None of the deegree classes shall implement Runnable/Callable interface nor extend Thread. Tasks for parallel and time limited processing shall be delegated to org.deegree.framework.concurrent.Executor.

This is for two reasons:

a) avoiding problems with independent Threads started by different deegree modules

b) saving resources because Executor class internally uses a Thread pool that can be share by all tasks

4 Messaging / Internationalization

Messages generated by deegree - especially exception messages - shall be externalized into a properties file. Debug/Logging messages must not be externalized! As default org.deegree.i18n.messages_en.properties shall be used for this. A user may override messages by creating an additional properties file in the root directory. In this case deegree first reads messages_en.properties. After this

deegree tries to find a message properties files matching locale of the VM in the root directory (messages_$LOCALE$.properties). If such a file is available, properties defined there will be read and override the default definitions read from org.deegree.i18n.messages_en.properties.

Messages can be referenced by using the static method org.deegree.i18n.Messages.getMessage(String, Object[]). The first parameter defines the name of the message, the object array can be used to pass N parameters to the message. Example:

```
# message definition
...
FRAMEWORK_ERROR_LOADING_MESSAGES=error loading message name {0}; timestamp: {1}
...

// code fragment
...
if ( message == null ) {
       throw
       new Exception( Messages.getMessage( ''FRAMEWORK_ERROR_LOADING_MESSAGES'',
                     ''MyMessage'', System.currentMillis() );
}
...
```

All message keys shall be in upper case characters and shall start with a prefix that gives a hint about the source of the message. Valid prefixes are:

- DATASTORE
- ENTERPRISE
- FRAMEWORK
- GRAPHICS
- IGEO_STD
- IGEO_PORT
- IO
- MODEL
- OGC
- PORTAL
- SECURITY
- SOS
- TYPES
- TOOLS

- WASS
- WCS
- WFS
- WMPS
- WMS
- WPVS

Each word forming a key shall be separated by a '_' where the complete key shall give a hint what kind of message is assigned to it (e.g. ERROR, INVALID_PARAMETER ...)

5 Code style / Code Format

As a rule of thumb, one line of code shall not exceed 120 characters. This rule may be broken depending on your screen size and if a few (not 20, 50 or 100) characters more per line increase readability of the code.

5.1 Indentation

Indents are always four character size; don't use tabs. Class definition starts at the first column; variables, static code blocks, methods, constructors and inner classes are indented by four characters. Each variable declaration and code block within a method as well as nested code blocks are indented in the same way.

```
class CurveImpl extends OrientableCurveImpl implements Curve, GenericCurve,
                                                       Serializable {


    protected ArrayList segments = null;

    /**
     * initialize the curve by submitting a spatial reference system and
     * an array of curve segments. the orientation of the curve is '+'
     *
     * @param segments array of CurveSegment
     */
    public CurveImpl( CurveSegment segments ) throws GeometryException {
        this( '+', new CurveSegment[] {segments} );
    }

    /**
     * initialize the curve by submitting a spatial reference system,
     * an array of curve segments and the orientation of the curve
     *
     * @param segments array of CurveSegment
     * @param orientation of the curve
     */
```

```
    public CurveImpl( char orientation, CurveSegment[] segments )
                                                throws GeometryException {
        super( segments[0].getCoordinateSystem(), orientation );

        this.segments = new ArrayList( segments.length );

        if ( segments != null ) {
            for ( int i = 0; i < segments.length; i++ ) {
                this.segments.add( segments[i] );

                if ( i > 0 ) {
                    if ( !segments[i - 1].getEndPoint().
                           equals( segments[i].getStartPoint() ) ) {
                        throw new GeometryException( "end-point of segment[i-1] " +
                                  "doesn't match start-point of segment[i]!" );
                    }
                }
            }
        }

        setValid( false );
    }
...
}
```

5.2 Visibility

Use 'lowest' visibility for classes and methods as possible. A method that is just used by the class where it is defined doesn't have to be public or protected. Use package-visibility instead of public declaration if possible.

5.3 Braces

Opening braces are always located in the same line as the definition (class, constructor, method, array ...) the brace is assigned to. For short definitions like arrays a closing brace may be located in the same line as the opening brace:

```
int a = new int[] { 3, 2, 5, 8 };
```

Closing braces for classes, methods, and code blocks must always be located in the same column as the first character of the class, method, and code block (see code fragment above). Exceptions to this rule can be made for empty classes, methods, and code blocks.

Don't use if-statements and loops without braces.

5.4 White spaces

Never use more than one white space between tokens other than for indentation.

- use a white space before and after operators :

  ```
  int a = 1; i < 3; a == b etc.
  ```

- use a white space before and after opening and closing braces except if a ';' follows the closing brace or nothing included between an opening and closing brace:

  ```
  if ( a == b ) { ...; getArea();
  ```

- use a white space behind a ',' (`map.put( "key", value );`)
- do not use a white space after an opening and before a closing '[' ']' (`a[4];`)

5.5 Blank lines

Blank lines should be used:

- after package declaration
- after import declarations
- after class declaration (if a class is not empty)
- after instance and class variable declaration
- between two methods, two constructors and between constructors and methods

Other usage of blank lines is not handled so strictly but a blank can be used if the readability of the code is increased (e.g. before and after loops or if-blocks)

5.6 Line wrapping / new lines

- as written above a line shall not exceed 120 character length without a good reason
- Keep 'else if' in one line

If line wrapping is required

```
Point p4 =
    getCurveSegmentAt( index + 1 ).getStartPoint();
```

is prefered instead of

```
Point p4 = getCurveSegmentAt( index + 1 )
                            .getStartPoint();

throw new GeometryException( "end-point of segment[i-1] " +
                             "doesn't match start-point of segment[i]!" );
```

is prefered to

```
throw new GeometryException(
            "end-point of segment[i-1] doesn't match start-point of segment[i]!" );

protected GetFeature(String version, String id, String handle,
                     String resultType, String outputFormat, int maxFeatures,
                     int startPosition, int traverseXLinkDepth,
                     int traverseXLinkExpiry, Query[] query)
```

is prefered to

```
protected GetFeature(String version, String id, String handle,
            String resultType, String outputFormat, int maxFeatures,
            int startPosition, int traverseXLinkDepth, int traverseXLinkExpiry,
            Query[] query)

public static GetFeature create(String id, Element element)
                                          throws OGCWebServiceException,
                                          MissingParameterValueException {
```

is prefered instead of

```
public static GetFeature create(String id, Element element)
            throws OGCWebServiceException,MissingParameterValueException {
```

5.7 Header and Footer

5.7.1 classes

All deegree class files share the same header and footer entries.

Header:

```
//$Header: Exp $
/*----------------    FILE HEADER  ------------------------------------------

 This file is part of deegree.
 Copyright (C) 2001-2006 by:
 EXSE, Department of Geography, University of Bonn
 http://www.giub.uni-bonn.de/exse/
 lat/lon GmbH
 http://www.lat-lon.de

 This library is free software; you can redistribute it and/or
 modify it under the terms of the GNU Lesser General Public
 License as published by the Free Software Foundation; either
 version 2.1 of the License, or (at your option) any later version.

 This library is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 Lesser General Public License for more details.

 You should have received a copy of the GNU Lesser General Public
 License along with this library; if not, write to the Free Software
 Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

 Contact:

 Dr. Andreas Poth
 lat/lon GmbH
 Aennchenstr. 19
 53177 Bonn
 Germany
 E-Mail: poth@lat-lon.de

 Prof. Dr. Klaus Greve
 Department of Geography
 University of Bonn
 Meckenheimer Allee 166
 53115 Bonn
 Germany
 E-Mail: greve@giub.uni-bonn.de

 ---------------------------------------------------------------------------*/
```

External contributors to deegree may replace the copyright note

Footer:

```
/* ********************************************************************
Changes to this class. What the people have been up to:
$Log: MYNEWCLASS.java,v $

******************************************************************** */
```

5.7.2 properties files

All deegree properties files share the same header and footer entries.

Header:

```
##########################################################################
#
# This file is part of deegree.
# Copyright (C) 2001-2006 by:
# EXSE, Department of Geography, University of Bonn
# http://www.giub.uni-bonn.de/deegree/
# lat/lon GmbH
# http://www.lat-lon.de
#
# This library is free software; you can redistribute it and/or
# modify it under the terms of the GNU Lesser General Public
# License as published by the Free Software Foundation; either
# version 2.1 of the License, or (at your option) any later version.
#
# This library is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public
# License along with this library; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
#
# Contact:
#
# Andreas Poth
# lat/lon GmbH
# Aennchenstr. 19
# 53177 Bonn
# Germany
# E-Mail: poth@lat-lon.de
#
# Prof. Dr. Klaus Greve
# Department of Geography
# University of Bonn
# Meckenheimer Allee 166
# 53115 Bonn
# Germany
# E-Mail: greve@giub.uni-bonn.de
#
# @version $Revision: 1.2 $
# @author <a href="mailto:poth@lat-lon.de">Andreas Poth</a>
# @author last edited by: $Author: poth $
#
# @version 1.0. $Revision: 1.2 $, $Date: 2006/08/17 12:20:37 $
#
##########################################################################
```

External contributors to deegree may replace the copyright note

Footer:

```
################################################################################
# Changes to this class. What the people have been up to:
# $Log:  $
#
################################################################################
```

5.7.3 xml/xsl files

All deegree properties files share the same header and footer entries.

Header:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--  ======================================================================

This file is part of deegree.
Copyright (C) 2001-2006 by:
EXSE, Department of Geography, University of Bonn
http://www.giub.uni-bonn.de/deegree/
lat/lon GmbH
http://www.lat-lon.de

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

Contact:

Andreas Poth
lat/lon GmbH
Aennchenstr. 19
53177 Bonn
Germany
E-Mail: poth@lat-lon.de

Prof. Dr. Klaus Greve
Department of Geography
University of Bonn
Meckenheimer Allee 166
53115 Bonn
Germany
E-Mail: greve@giub.uni-bonn.de

@version $Revision: 1.4 $
@author <a href="mailto:poth@lat-lon.de">Andreas Poth</a>
@author last edited by: $Author: poth $

@version 1.0. $Revision: 1.4 $, $Date: 2006/08/04 07:17:43 $
```

```
========================================================================== -->
```

External contributors to deegree may replace the copyright note

Footer:

```
<!-- ========================================================================
Changes to this class. What the people have been up to:
$Log:  $
 ========================================================================== -->
```

5.8 Comments

All comments shall be valid against java comment definition; e.g. use validators offered by most IDEs to verify the validity of your comments.

5.8.1 Classes/Interfaces

All classes (except inner classes) share the same type of comment:

```
/**
 *
 * describe function and usage of the class here. If the class is directly
 * assigned to an OGC or ISO specification declare which and what version.
 *
 * @version $Revision:  $
 * @author <a href="mailto:poth@lat-lon.de">Andreas Poth</a>
 * @author last edited by: $Author: $
 *
 * @version 1.0. $Revision: $, $Date: 2005/03/07 08:23:40 $
 *
 */
```

Notice that several elements of the comment will be set when checked into the CVS.

Inner classes may just have a subset of these elements:

```
    /**
     * private inner class for enabling multi threaded request processing
     *
     * @author <a href="mailto:poth@lat-lon.de">Andreas Poth</a>
     */
    protected class Performer extends AbstractPerformer {
    }
```

5.8.2 Methods

Each public and protected method shall have a comment that explains what the method does. Just for simple getter- and setter methods which names are self ex-

plaining a comment can be left away (e.g. getArea() of a Surface or setName(String)). But this should be an exception and not the usual case. Private methods need not be exhaustively commented. But if a private method is doing a more or less complex job it should have one. If a method implements a behavior described in an OGC or ISO specification it may be useful to add the related part of the specification to the method's comment.

References to other classes and/or methods within a comment should be declared using '@see' tag. Class, method and variable names should be surrounded by <code>...</code> . Consider to use HTML-tags to ensure a complex comment will have a proper layout when transformed into the java api doc.

If a method or constructor receives parameters when it is called a short explanation of each parameter should be given (only if method and parameter are self explaining it can be left out, e.g. setName(String name).

A method may throw one or more exceptions. It is helpful for other programmers to declare the contract when an exception will be thrown, especially if an exception will just be passed through from another method called within a method.

A method comment could look like this:

```
/**
 * Returns the source data for a coverage.
 * This is intended to allow applications to establish what <code>Coverage</code>s
 * will be affected when others are updated, as well as to trace back to the
 * "raw data".
 *
 * @param sourceDataIndex Source coverage index. Indexes start at 0.
 * @return The source data for a coverage.
 * @throws IndexOutOfBoundsException if <code>sourceDataIndex</code> is out
 * of bounds.
 *
 * @see #getNumSources
 * @see org.opengis.coverage.grid.GridCoverage#getSource
 */
public Coverage getSource(int sourceDataIndex) throws IndexOutOfBoundsException {
       //do something
}
```

Even though a nice method shouldn't have more than ~40-60 lines of code (less would be better) comments should be added within the code whenever a complex job is done that may not be self explaining. Comments such as

// call getter to retrieve area of a surface'

are not useful. A comment should explain the idea behind something or its relation to other parts of a context, accessing the area of a surface by calling 'getArea()' doesn't need to be commented.

Please consult the Sun documentation

5.8.3 Variables

Private instance variables don't have to be commented, protected instance variables should be commented and public instance variables (although they should be avoided anyway) must be commented. The same is true for class variables.

5.9 Useless code

Avoid useless code because it reduces readability! Common places for useless code are:

- local variables are declared but never used (read). *In several cases this indicates an error.*
- variables passed to methods are never read. *In several cases this indicates an error.*
- private methods are never called
- constants are never used
- type casts are not required (especially since JDK 5)
- classes/packages will be declared in import section of another class but not be used
- declaring of exceptions that never will be thrown

5.10 Package dependencies

Reduce package dependencies to a required limit. Functionality encapsulated by the classes of a package shall be provided be a clear set of interfaces. Classes of a package do not shall depend on classes of nested packages. Just define classes as public (visible for other packages) if necessary (see 4.2).

6 XML

XML, XSD and XSLT is wildly used in the context of deegree. It is used as well for configuration of web services implemented in deegree as for request templates or right definitions. Because XML allows several different ways to encode the same content/schema some restrictions shall be made in the context of deegree to simplify usage even for those persons who are not so familiar with it.

The rules are:

- Use namespaces whenever it is possible (in context of older OGC web services this is not possible because they use DTDs instead of XML schema's).
- If you use namespaces use prefixes for each element in an XML document that is within this namespace.
- Do not use 'unqualified' option within XML schema's.
- Define prefix-namespace binding document-wide (within the root element of a document).
- Avoid namespaces for element attributes when possible.
- deegree specific elements shall be in an namespace defined in deegree (starting with http://www.deegree.org/). The assigned prefix shall start with 'deegree', e.g. deegreeWCS.

Examples:

not like this

```
<?xml version="1.0" encoding="UTF-8"?>
<a xmlns="mynamespace1">
   <b>
       <c xmlns="mynamespace2">
           <d>text</d>
       </c>
   </b>
   <b>
       <c xmlns="mynamespace2">
           <d>text</d>
       </c>
       <c xmlns="mynamespace2">
           <d>text</d>
       </c>
   </b>
</a>
```

and not like this

```
<?xml version="1.0" encoding="UTF-8"?>
<a xmlns="mynamespace1">
   <b>
       <cs:c xmlns:cs="mynamespace2">
```

```
            <d>text</d>
        </cs:c>
    </b>
    <b>
        <cs:c xmlns="mynamespace2">
            <d>text</d>
        </cs:c>
        <cs:c xmlns="mynamespace2">
            <d>text</d>
        </cs:c>
    </b>
</a>
```

this is the desired formatting

```
<?xml version="1.0" encoding="UTF-8"?>
<as:a xmlns:as="mynamespace1" xmlns:cs="mynamespace2">
    <as:b>
        <cs:c>
            <d>text</d>
        </cs:c>
    </as:b>
    <as:b>
        <cs:c>
            <d>text</d>
        </cs:c>
        <cs:c>
            <d>text</d>
        </cs:c>
    </as:b>
</as:a>
```