GeOxygene

Platform

– User Guide –

GeOxygene Project
March 2005
Version 1.0

Title
GeOxygene Platform – User Guide

Keywords
GeOxygene, architecture, model, object oriented, platform, GIS, Java, DBMS, Oracle, PostGIS, object/relational mapping, Web services, COGIT Laboratory.

Abstract
The GeOxygene open source project is the result of work by the COGIT laboratory of the Institut Géographique National (IGN, France) to develop an open and modular software platform dedicated to geographical information research applications, with a common architecture and data model so that the software, documentation and maintenance can be shared. The GeOxygene prototype is based on innovative technology to enable research applications to be deployed in the form of web services. GeOxygene uses a long term, interoperable base and implements ISO standards and OGC specifications.

This document describes the GeOxygene platform and how to use it.

Authors:
Thierry Badard
Arnaud Braun

Contributions and review:
Anne Ruas

GeOxygene Project
http://sourceforge.net/projects/oxygene-project

TABLE OF CONTENTS

1. INTRODUCTION 4
1.1. BACKGROUND 4
1.2. AIM 4
1.3. GENERAL ARCHITECTURE 5
1.4. DIRECTORY STRUCTURE 7

2. INSTALLATION AND CONFIGURATION 9
2.1. REQUIREMENTS 9
2.2. COMPILING GEOXYGENE 9
2.3. CONFIGURATION 10

3. USING GEOXYGENE 12
3.1. SIMPLE MAPPING EXAMPLE 12
3.2. LOADING GEOGRAPHIC DATA 13
3.3. FIRST STEP IN GEOGRAPHICAL DATA HANDLING 14
3.4. IMPLEMENTING A COMPLEX MODEL 14
3.5. DATA HANDLING 15
3.6. THE VIEWER 16
3.7. THE BROWSER 16

APPENDIX A UNDERSTANDING THE SCHEMA 17
A.1. FEATURES (package *fr.ign.cogit.geoxygene.feature*) 17
A.2. GEOMETRY (package *fr.ign.cogit.geoxygene.spatial*) 17
A.3. TOPOLOGY (package *fr.ign.cogit.geoxygene.spatial*) 20

APPENDIX B CODING CONVENTIONS 22

1. INTRODUCTION

1.1. BACKGROUND

Many problems have to be overcome to develop geographical information related applications.

- The data models of the various geographical information systems are not interoperable, despite attempts made by ISO and OGC (Open Geospatial Consortium) to introduce standardised interfaces. It is, therefore, unlikely that an application developed using a particular model can be used with a different system without significant modification.
- The programming languages used by GIS applications are often proprietary languages. Software cannot, therefore, be shared with other GIS applications and users are dependent on the software suppliers for upgrades.
- Current GIS applications are not accessible via the Web without costly extensions. Moreover, although these extensions enable data to be accessed on line, they do not always include web services for remote procedure calls.
- Current GIS applications are not genuine database management systems and do not always provide essential facilities for data management (concurrent access, security, etc.).

A number of software technologies can be used to address these problems: web accessible object oriented languages (such as Java), object oriented reusable component design and analysis techniques (such as UML), relational database management systems able to handle objects for storing geographical data for geographical information systems (such as Oracle, PostGIS, etc.), structured languages for transferring data over the Web (such as XML) and web service descriptions and remote procedure calls in heterogeneous, distributed computing environments such as WSDL (Web Services Description Language) and SOAP (Simple Object Access Protocol).

The COGIT laboratory of the Institut Géographique National (IGN), France, has designed and developed a new platform called GeOxygene (originally OXYGENE) based on these technologies (Badard and Braun, 2004)[1]. It replaces GéO$_2$, a geographical system built on top of the O$_2$ object oriented database management system, that was developed in the 1990s. It provides a long term, robust base for the COGIT Laboratory for their research work, in particular for research into search aids, data distribution, data maintenance and processing.

At the beginning of 2005, the COGIT Laboratory decided to publish part of the GeOxygene source in the form of an open source project.

1.2. AIM

GeOxygene was intended initially to provide an open, modular, interoperable framework for the development of research applications at the COGIT Laboratory, for research on multiple representation, data quality for risk studies, looking up and distributing data, using a common data model and a common architecture, so that the software, documentation and maintenance could be shared. The aim was (and still is) to establish and exploit research work. The GeOxygene prototype based on innovative technology is designed to enable research applications to be deployed in the form of web services. GeOxygene implements ISO standards and OGC specifications. It is designed for total interoperability.

The aim of publishing part of the GeOxygene source code as open source is not to compete with or replace any of the many current geographical open source projects, but rather to provide another building brick for the construction of open source solutions. The GeOxygene project is certainly less mature and complete than projects such as Geotools (http://www.geotools.org) or Deegree (http://deegree.sourceforge.net), as it is more recent and has been developed with fewer resources. However, even in its current state, GeOxygene can be considered to be a realistic starting point for a complete, rigorous implementation of ISO 19107 and the specifications of features (features, feature

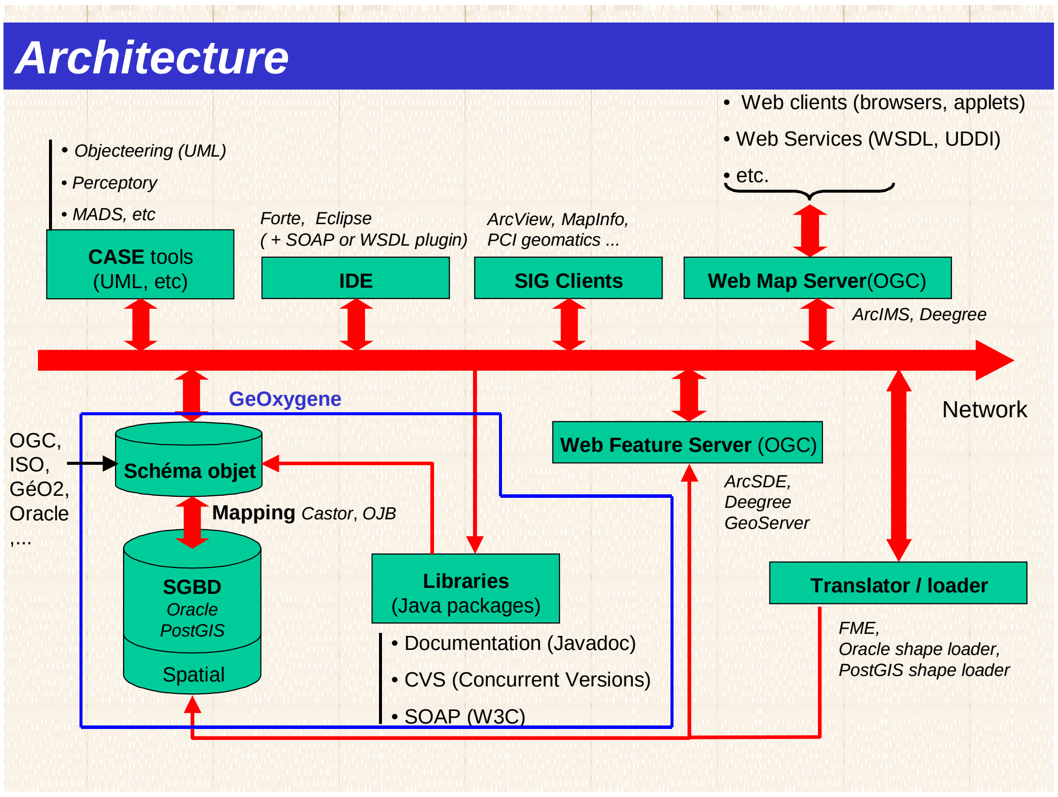
[1] (Badard et Braun, 2004) Badard, T., Braun, A. *OXYGENE: A Platform for the Development of interoperable Geographic Applications and Web Services*. Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA'04), IEEE Press, August 30 - September 03, 2004, Zaragoza, Spain, pp. 888-892.

collection and relationships between features), as well as a starting point for using JDO (Java Data Objects) for geographical databases. GeOxygene is, therefore, a building brick for constructing geographical information systems that are truly interoperable as they implement OGC specifications and ISO standards. The GeoAPI project has been set up to rationalise the current open source projects and, if GeOxygene is to be included, it must, in the near future, implement the Java interfaces defined by GeoAPI.

GeOxygene is not designed for end users as it is not a turnkey product where everything is handled invisibly behind graphical interfaces. It is destined for power users, Java developers and scientists who need to write geographical information systems. It can, therefore, be considered as a kernel for the development of such applications.

1.3. GENERAL ARCHITECTURE

GeOxygene is entirely modular (cf. below) using a network architecture to communicate between the components and for the deployment of developments. The software components are independent and interoperable and are used for the purposes for which they are best suited. For example, the DBMS is used exclusively for storing data with the geometrical operators in a separate module. This approach has been adopted to avoid the "all things to all men" type of architecture where a non-specific component ends up carrying out operations for which it was not originally designed. This type of approach leads to very rapid decline in performance and to serious problems in maintenance when the original developers are no longer working on the system.


Architecture
• Web clients (browsers, applets)
• Web Services (WSDL, UDDI)
• etc.
• Objecteering (UML)
• Perceptory
• MADS, etc
CASE tools (UML, etc)
Forte, Eclipse (+ SOAP or WSDL plugin)
IDE
ArcView, MapInfo, PCI geomatics ...
SIG Clients
Web Map Server(OGC)
ArcIMS, Deegree
Network
GeOxygene
OGC, ISO, GéO2, Oracle ,...
Schéma objet
Mapping Castor, OJB
SGBD Oracle PostGIS Spatial
Libraries (Java packages)
• Documentation (Javadoc)
• CVS (Concurrent Versions)
• SOAP (W3C)
Web Feature Server (OGC)
ArcSDE, Deegree GeoServer
Translator / loader
FME, Oracle shape loader, PostGIS shape loader

GeOxygene platform architecture showing typical software components

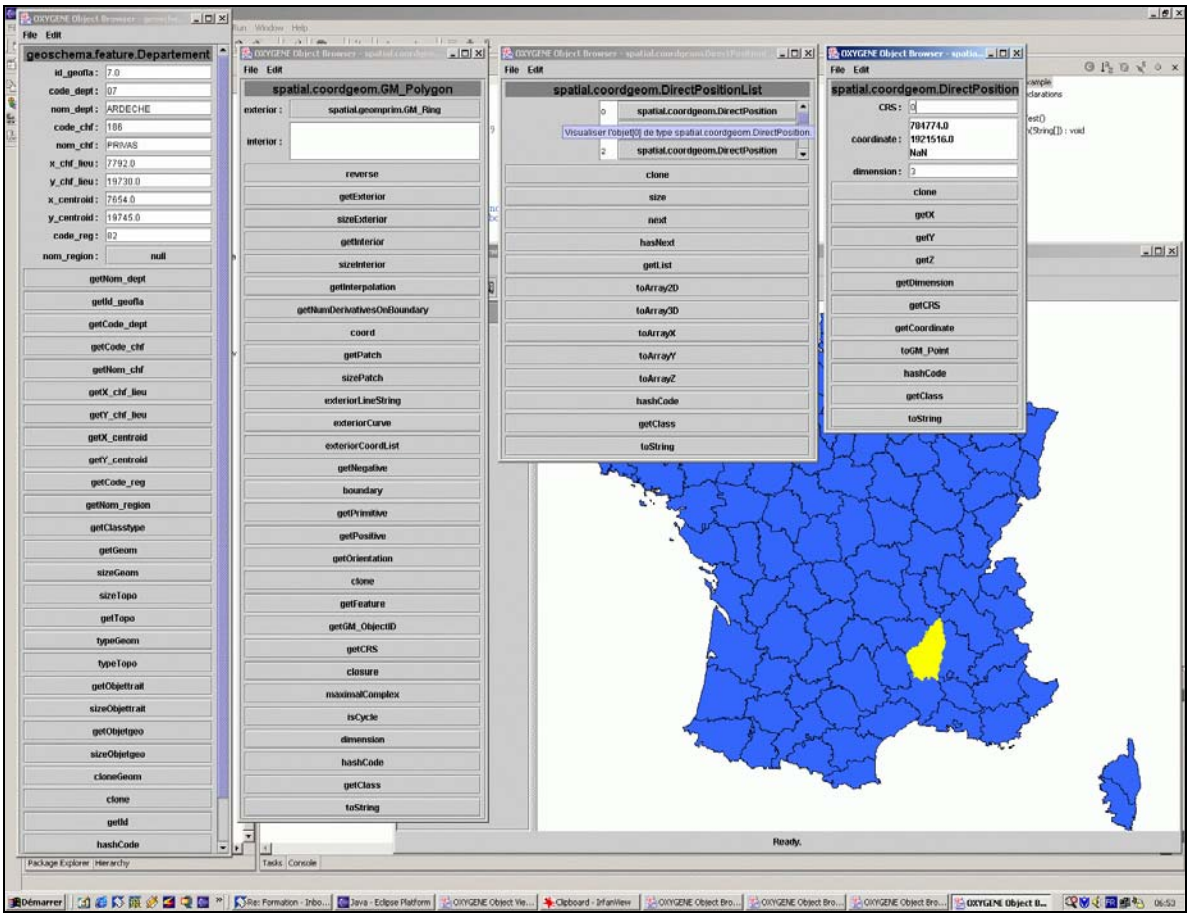
The software components

- The GeOxygene platform was developed mainly in Java, an object oriented language that was chosen principally for its portability, incorporating some components from open source projects. It can model and operate on all the aspects of geographical information (semantics, geometry, topology and metadata) using an object oriented schema which implements OGC specifications and ISO standards. The Java code for this object oriented schema is part of the open source distribution.
- Data is stored in a relational DBMS (two systems are currently supported: Oracle with the spatial cartridge and PostGIS, an open source geographical layer on PostgreSQL) to provide fast response. The SQL scripts for handling the DBMS are included in the open source distribution.
- Flexible mapping (i.e. the correspondence between Java classes and tables) between the object schema and the relational tables is provided by OJB (ObJect relational Bridge, an open source component from the Apache foundation). Users, therefore, have a consistent object view of the information they are handling. The application is modelled in UML and written in Java, using the platform's ISO/OGC object model. The XML mapping files are included in the open source distribution, as well as all the software for using such files.
- The geographic functions are programmed in separate libraries to ensure that developments are independent. These functions are to be found on the web (eg: the open source JTS Topology Suite for geometric algorithms) or from old projects that have been carried out at COGIT. They may be written in C, C++, Fortran or Ada, but are interfaced with Java using the Java Native Interface (JNI). The interface functions for the JTS Topology Suite are included in the open source distribution.

GeOxygene as used at COGIT has a number of other tools that are not included in the open source distribution as each GeOxygene user can build his own system. These tools are used for:

- generating documentation (Javadoc)
- source code version maintenance (CVS: Concurrent Versions System)
- loading geographic data, displaying it and analysing it using GIS clients
- building applications with Computer Aided Software Engineering (CASE) tools
- developing applications with an Integrated Development Environment (IDE). The IDE used at COGIT is Eclipse (from the Eclipse open source project, based on a contribution from IBM), with a large number of plugins, in particular UML, which make it almost a CASE tool.

Two further tools have been developed and are included in the open source distribution to benefit from the GeOxygene object oriented view of the geographical data: a geographical object viewer (based on version 0.8 of the open source Geotools project) and a generic graphic object browser (to display the status of an object at a given time and invoke the methods dynamically). These two components are shown below.


geoschema.feature.Departement

id_geofla : 7.0
code_dept : 07
nom_dept : ARDECHE
code_chf : 186
nom_chf : PRIVAS
x_chf_lieu : 7792.0
y_chf_lieu : 19730.0
x_centroid : 7654.0
y_centroid : 19745.0
code_reg : 82
nom_region : null

spatial.coordgeom.GM_Polygon

exterior : spatial.geomprim.GM_Ring
interior :

spatial.coordgeom.DirectPositionList

0 spatial.coordgeom.DirectPosition
2 spatial.coordgeom.DirectPosition

Visualiser l'objet[0] de type spatial.coordgeom.DirectPosition.

spatial.coordgeom.DirectPosition

CRS : 0
coordinate : 704774.0
1921516.0
NaN
dimension : 3

GeOxygene platform viewer and object graphic browser

Finally, GeOxygene was designed from the start to use SOAP and WSDL protocols and language defined by the World Wide Web Consortium (W3C) and provide the WFS (Web Feature Service) and WMS (Web Map Service) on line services specified by OGC. It can, therefore, be used to deliver the functions developed for it as web services (Badard and Braun, 2003)[2].

1.4. DIRECTORY STRUCTURE

GeOxygene is distributed with the following structure.

- The *src* directory has the implementations of standards, specifications and tools:
    - o ISO 19107 (geometry and topology)
    - o ISO 19109 (metamodels for defining application schemas)
    - o ISO 19115 (metadata), partial implementation
    - o OGC Features abstract specification (geographical objects)
    - o tools for accessing geographical databases using object/relational mapping
    - o utilities:
        - → in memory spatial indexing
        - → loading data into Oracle and PostGIS
        - → geometric algorithms
        - → graphic object navigator
        - → viewer
    - o examples and tests

- The "data" directory has the geographical classes specific to each user.

[2] (Badard et Braun, 2003) Badard, T., Braun, A. *OXYGENE: An Open Framework For the Deployment of Geographic Web Services,* International Cartographic Conference, Durban, South Africa, 2003.

- The source code for each user is placed in a directory specific to each user. For example, a sample Douglas-Peucker filter is to be found in the *src* directory, in the *fr.ign.cogit.geoxygene.generalisation* package*.*

The SQL scripts for accessing the DBMS are to be found in the *sql* directory which has two sub-directories *oracle* and *postgis* for the scripts for the two DBMSs supported.

The XML object/relational mapping files are to be found in the *mapping* directory which has a sub-directory *ojb1* for the OJB mapping tool currently supported.

The files in the *castor* sub-directories of *sql* and *mapping* are the implementation of the Castor mapping tool (http://castor.codehaus.org/index.html) which is less efficient than OJB, which is no longer used and is not currently working. Given the modularity of GeOxygene, Castor could, however, be put back in service with a significant modification of the code in these two sub-directories.

2. INSTALLATION AND CONFIGURATION

2.1. REQUIREMENTS

■ *OPERATING SYSTEM*

GeOxygene has been tested under Solaris, Linux and Windows.

■ *JAVA*

Must be installed:

- Java Development Kit (JDK) Standard Edition (J2SE) 1.4 or later
- Java Development Kit (JDK) Enterprise Edition (J2EE) 1.3.1

NB: support for J2EE 1.4 has not been tested.

■ *ORACLE*

Both

- the SDOAPI library (*sdoapi.zip* or *sdoapi.jar* depending on the Oracle version)
- the Oracle JDBC driver (eg: *classes12.zip*)

must be installed.

NB: currently, these libraries are required for compiling the project. They are, of course, not required if you use a compiled version of the project with PostGIS.

2.2. COMPILING GEOXYGENE

GeOxygene can be compiled:

- Either by using Ant (http://ant.apache.org) developed by the Apache foundation
- Or by using the open source Eclipse IDE (http://www.eclipse.org)

■ *COMPILING USING APACHE ANT*

Edit the file *build.properties* in the GeOxygene root directory. Set the path (`j2ee.dir`) for the J2EE installation directory on your system. Save the changes and close the file.

Change the CLASSPATH variable so that the Oracle libraries are available to the Java compiler or copy the two libraries required to the GeOxygene *lib* sub-directory.

The default build target for Ant will compile GeOxygene. Therefore, to compile GeOxygene, you just need to type the command:

```
% ant
```

in the GeOxygene root directory (the directory which holds the file *build.xml*).

There are other build targets, type:

```
% ant -p
```

to display the list of build targets and their descriptions.

Note that you can also use the *build.xml* and *build.properties* files from Eclipse to compile GeOxgene, generate the Java documentation, etc.

■ *CONFIGURING ECLIPSE*

Creating a project

File Menu -> New -> Project, a window appears.
On the left: "Java", on the right "Java Project", Next
Project Name: "GeOxygene"
Project Contents: uncheck "Use defaults"
Select the "GeOxygene" root directory
(for example: D:\users\foo\GeOxygene).
Next. There is a certain delay.

Configuring the compilation

"Default output folder" (at the bottom): GeOxygene/classes
Select the directory using "Browse", if necessary create the directory using "Create new folder".

"Source" tab: add the following two directories using "Add Folder".

- GeOxygene/src
- GeOxygene/data

If the dialogue box "Do you want to remove the project as source folder" is displayed, select "Yes".

"Libraries" tab:
Select the following libraries from the *GeOxygene/lib* directory using "Add JARs":

- ojb-1.0.jar (or another version)
- batik-all.jar
- jdbc3-postgresql.jar
- postgis.jar
- jts.jar
- geotools1-cogit.jar (or another version)
- batik-all.jar

Using "Add External JARs":

- select *j2ee.jar* from the J2EE *lib* installation directory (depends on the installation, but it is often in *C:\j2sdk1.3.1\lib* under Windows or */usr/j2sdk1.3.1/lib* under Unix/Linux)
- select the *sdoapi.zip* and *classes12.zip* files (or others, depending on the Oracle version used) from the directory holding these libraries (cf. section 2.1).

Click "Finish" and the compilation will start.

2.3. CONFIGURATION

■ *CREATING THE BASIC DBMS TABLES*

Before using the system for the first time, the script *superscript_ojb.sql* in the *oracle* or *postgis* sub-directory of the *sql* directory must be executed for the DBMS. It creates the tables required for the operation of the platform (the tables required for OJB and, for Oracle, the table required for certain queries).
The script *resultat.sql* may be executed to create the table for storing results.

■ *CONFIGURING THE OJB.PROPERTIES FILE*

Edit the *OJB.properties* file in the GeOxygene *src* directory.
Set the full pathname (not relative path) to the *repository.xml* mapping file.

For example:

RepositoryFile=D:/users/foo/GeOxygene/mapping/ojb1/duschmok/repository.xml

■ *CONFIGURING THE REPOSITORY_DATABASE.XML FILE*

This file is in the sub-directory *mapping/ojb1.* It holds the parameters for connecting to the DBMS (name, password, etc). Edit it to specify your DBMS and connection parameters (server name, user name, password). There are two sample configurations, one for Oracle and one for PostGIS.

3. USING GEOXYGENE

3.1. SIMPLE MAPPING EXAMPLE

This section describes a simple example of creating and using a persistent class. The aim is to understand how the Java code works: there is no question of geometry or geography.

Class fr.ign.cogit.geoxygene.example.tutorial.MyClass
Examine *MyClass* in the *fr.ign.cogit.geoxygene.example.tutorial* package.
This is a class like any other. There is nothing to show that it will be persistent (i.e. storable in a database): the persistence is non-intrusive, that is it is not specified in the code. It is however necessary to have a field for the identifier ("id").
This class is defined as a "bean", that is that each property has "get" and "set" methods called accessors, to access it. This is the cleanest method of coding.

Create the corresponding table in the DBMS
Examine the *maclasse.sql* table creation SQL script in *sql/oracle* or *sql/postgis*. Execute this script for the DBMS.

Mapping MaClasse
Examine the XML file *repository_maclasse.xml* in the directory *mapping/ojb1.* It shows the correspondences between the Java world and the DBMS world. The syntax is fairly intuitive. This file is used by OJB.

Configuring the OJB configuration file
Edit the file *OJB.properties* in the GeOxygene *src* directory.
Specify the full pathname of the mapping file *repository.xml.*
NB: the full pathname must be given, not the relative path.
For example: *RepositoryFile=D:/users/duschmok/GeOxygene/mapping/ojb1/duschmok/repository.xml*

Configuring mapping files
The *repository.xml* file is read by OJB to initialise the connection to the DBMS and load the mapping data.
This file is stored in the *mapping/ojb1* directory*.* Its pathname must be defined in the file *OJB.properties.* Examine it.
This file points towards other XML files (repository_*.xml) which themselves hold the mapping data.
Two declarations MUST be uncommented.

- *&database* which specifies the *repository_database.xml* file. It holds the parameters for connecting to the DBMS (name, password, etc). Edit it to specify your DBMS and connection parameters (server name, user name, password). There are two sample configurations, one for Oracle and one for PostGIS.
- This is followed by the list of mapping files to be used. Some files which are required for GeOxygene are already specified. *&tutorial* should point to *repository_maclasse.xml* and should be <u>uncommented</u>.

Executing the application
MyClass is now exercised by the class *fr.ign.cogit.geoxygene.example.tutorialTestMyClass.*

Look at the comments in the code.

This code shows the main functions of OJB: making objects persistent, loading objects from the database into Java and making an OQL (Object Query Language) query.

The main points to remember from this example are:

- `Geodatabase db = new GeodatabaseOjbOracle();`

This line initialises the DBMS connection and loads the mapping data. When this statement is executed the file *repository.xml* is read and analysed by the OJB object/relational mapping engine.

- `db.begin();`

  This line begins a transaction.

- `db.commit();`

  This line commits a transaction.

- `db.makePersistent(obj);`

  This line makes an object (whose class is defined in the mapping file) persistent in the database.

- `db.loadAll(class);`

  This line loads all the objects of a class, i.e. all the DBMS records are translated into Java objects. "load" has several different forms:

  - `db.load(class, id);` → to load a single object of a class using its identifier.
  - `db.loadAllFeatures(class);` → to load all <u>geographic</u> objects of a class.
  - `db.loadOQL(query, obj);` → to load using an OQL query (an example of this can be found in the "interroge" procedure). OQL is a database query language, similar to SQL, that works directly on Java classes rather than working on DBMS tables.

3.2. LOADING GEOGRAPHIC DATA

Loading geographical data into the DBMS from GIS files (eg: ESRI Shapefile)
The data is loaded using your own preferred tools (eg: FME) or the GIS file loaders supplied by Oracle and PostGIS (eg: shp2pgsql).

Creating Java classes and mapping files
Execute *fr.ign.cogit.geoxygene.appli.Console*
Select "SQL → Java".
Select the table that has been loaded into the DBMS.
Click OK to accept the default settings.

A message is displayed reminding you to compile the classes that have been generated. A Java class will have been generated in the directory selected (*geoxygene.geodata* by default, in the *data* directory*).* Edit this class and compile it. You will see that it reproduces the structure of the table. Note the property *geom* of type *GM_Object* that represents the geometry.

Edit the file *repository_geo.xml (*or the name that you have given in the console if it is different) which will be in the directory *mapping/ojb1/.* Remember to set the pointer to this file in the *repository.xml* file (i.e.: the declaration *geo* should point to *repository_geo.xml* and *&geo* should be <u>uncommented</u>). Note the mapping of the geometry property.

Formatting the data in the DBMS
Select "Manage Data".
For Oracle, the console will suggest that ids are generated. Accept. The console will create a column COGITID in the table defined as the primary key.
The console will also suggest that the geometry should be made uniform. Accept. There are sometimes composite geometry elements that are created owing to edge effects and shortcomings in certain translators. These will be corrected here. It may take some time.

If there is a large amount of data in the table, it is probable that the console will run out of memory. The console must be restarted with the "magic" option -Xmx512M (*java -Xmx512M appli.Console)*, for example ;-) which increases the memory allocated to the virtual machine. You must restart directly at this step.

Finally, the console suggests calculating a spatial index (for Oracle and PostGIS) and the space required (for Oracle): accept these as well.

Click OK to start execution.

Difference between these two examples
In example 3.1, we started with Java code to generate the SQL code. This goes from design (in this case a very simple design as there was only one class) to implementation. In example 3.2, we started with an SQL table and generated the Java class from this table. This is based on the data loaded. The last two steps above are used if and only if the database structure is generated from data loaded.

3.3. FIRST STEP IN GEOGRAPHICAL DATA HANDLING

Load a table of road segments, called *Troncon_route,* from any set of data. You should now know enough to understand and execute the example *FirstExample* in the package *fr.ign.cogit.geoxygene.example*. All the information necessary is in the Java code. If necessary, change the test class name in the Java code (variable *nomClasse*).

NB: the declaration of the mapping file *repository_result.xml* must be <u>uncommented</u>. This file has the mapping for the class *fr.ign.cogit.geoxygene.example.Resultat* which will be used in this example. The SQL script *resultat.sql* must be executed to create the corresponding table.

This class has a constructor to set up the mapping. If you use PostGIS, change `GeodatabaseOjbOracle()` to `GeodatabaseOjbPostgis()`.

The first example (`void exemple1()`) loads an object using the id, retrieves its geometry, creates a buffer for this geometry and then creates a new object of class *Resultat,* assigns the buffered geometry and makes the object persistent.

The second example (`void exemple2()`) loads all the objects of the class and, for each of them, retrieves the geometry, applies a Douglas-Peucker filter, creates a new object of class *Resultat,* assigns the filtered geometry and makes the object persistent.

3.4. IMPLEMENTING A COMPLEX MODEL

The model in the package *fr.ign.cogit.geoxygene.example.relations* shows all the complex features that may be encountered:

- Inheritance
- 1:1 relationships, uni-directional or bi-directional
- 1:n relationships, uni-directional or bi-directional
- n:m relationships, uni-directional or bi-directional

The relationships may be persistent or not.

NB. Implementing relationships in Java is tricky, especially for bi-directional relationships.

The model
No UML schema has been drawn as the model is very simple.

- Classes *AAA* and *BBB* both inherit from the abstract class *ClasseMere*.
- Classes *AAA* and *BBB* have 1:1, 1:n and n:m bi-directional relationships.
- Classes *AAA* have 1:1, 1:n and n:m uni-directional relationships towards *BBB* (i.e. you cannot find *AAA* from *BBB*).

Implementing the model in Java
This model is implemented in the package *fr.ign.cogit.geoxygene.example.relations.* Examine the Java codes of classes *ClasseMere, AAA* and *BBB*.

Rules for converting UML to Java

- There is no problem with inheritance in Java (there is no multiple inheritance here).
- Implementing relationships in Java is tricky. Relationships are defined using properties.

  - A 1:1 relationship is implemented as a property of type "la classe en relation".
  - A 1:n or n:m relationship is implemented with a list of objects of type "la classe en relation".

- Accessors (get, set and add methods) are used to manage bi-directional relationships. Coding is not simple. Examine the code carefully and cut and paste the code if you need relationships.
- The parent class of a hierarchy must have an “id” property to handle the persistence. This property is the identifier for the objects. It is inherited by the child classes.

Implementing the model in SQL
Examine and execute the script *init_relations_AAA_BBB.sql* for the DBMS.

Implementing mapping
Examine the directory *repository_AAA_BBB.xml.* Try to understand its structure and how the relationships are implemented, referring to the OJB documentation. Mapping bi-directional relationships in one direction only is not obvious but it improves the performance and reduces the number of joins on loading.

Application programs
There four test programs in the package *fr.ign.cogit.geoxygene.example.relations* that use this model:

- One for testing non-persistent uni-directional relationships
- One for testing persistent uni-directional relationships
- One for testing non-persistent bi-directional relationships
- One for testing persistent bi-directional relationships

3.5. DATA HANDLING

The package *fr.ign.cogit.geoxygene.datatools* has the classes required for data handling. It is important for users to understand the operation of two classes.

***Geodatabase* interface**

- *Geodatabase* is an interface for connecting to a DBMS via an object/relational mapper. This class is used to begin transactions, to load persistent data, to make the data persistent, to query and to commit or abort transactions. This interface is used for all operations on the database. It is instanced by a class which depends on the type of object/relational mapper and the type of DBMS.

For example:

```
Geodatabase db = new GeodatabaseOjbOracle ()
```

where *GeodatabaseOjbOracle* is the class for OJB and Oracle.

***Metadata* interface**
Metadata represents the object/relational mapping metadata: which table corresponds with which class and also the space required for the geographical data.

Example
Use the *TestGeodatabase* example in the package *fr.ign.cogit.geoxygene.example* to get an idea of the scope of the *Geodatabase* and *Metadata* interfaces*.*

3.6. THE VIEWER

A geographical object viewer has been developed for GeOxygene. It is run by using the command:

```
ObjectViewer viewer = new ObjectViewer (db);
```

where “db” represents the active *Geodatabase*.

The viewer is then asked to display collections of FT_Feature using the command:

```
Viewer.addFeatureCollection (collection, nom);
```

where "collection" is a FT_FeatureCollection and "nom" a string that will be the name (or title) of the collection displayed by the viewer.

Use is fairly intuitive. The command to display a class can be invoked from the Java code or directly from the interface.

Objects can be selected in the viewer and the browser run to view their properties, run methods and navigate the object oriented model.

Use the *TestViewer* example in the package *fr.ign.cogit.geoxygene.example* to get an idea of the scope of the viewer.

3.7. THE BROWSER

An graphic object browser has been developed for GeOxygene. It makes it possible to make full use of the object oriented aspect of the data stored in the platform.

It can be run from the viewer when an object has been selected. It can also be run directly using Java:

```
ObjectBrowser.browse (obj);
```

where "obj" is any Java object.

Use the *TestBrowser, TestBrowserObjTest* and *TestBrowserApplicationLauncher* examples in the package *fr.ign.cogit.geoxygene.example* to get an idea of the scope of the graphic object browser.

APPENDIX A UNDERSTANDING THE SCHEMA

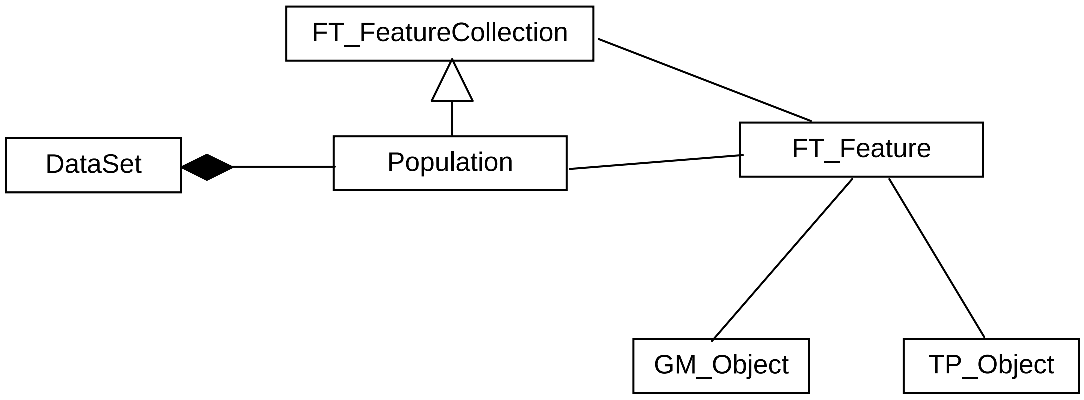
This section gives the background concepts for the schema used by GeOxygene.

A.1. FEATURES (package *fr.ign.cogit.geoxygene.feature*)

- *FT_Feature* is the parent class of the geographical classes. *FT_FeatureCollection* is the collection of *FT_Feature* objects. *FT_FeatureCollection* holds the spatial indexing methods. See example *fr.ign.cogit.geoxygene.example.TestIndex* for an idea of the operation of the spatial indexes.

- The class *DataSet* represents a set of data: for example an extract from the database for a limited geographical area, dated 2003, or the hydrographic "theme" from a topographical database. A "theme" is a sub-set of a *DataSet* and is itself a *DataSet.* A *DataSet* holds metadata (area, year, etc.).

- A *DataSet* is made up of several *Populations.* The class *Population* represents a particular *FT_FeatureCollection*: it includes ALL the *FT_Feature* objects of a *DataSet,* of the same type.



Important: Although the classes *FT_Feature* and *FT_FeatureCollection* conform to the OGC model, this is not the case for the *DataSet* and *Population* classes. **These classes are extensions to the OGC specifications.** If these two classes are used, then the application will no longer be interoperable with other OGC applications.

A.2. GEOMETRY (package *fr.ign.cogit.geoxygene.spatial*)

- *GM_Object* is a parent class of geographical objects. A *GM_Object* may be a primitive (*GM_Primitive*), an aggregate (*GM_Aggregate*) or a complex (*GM_Complex*). Aggregates and complexes are rather special collections of primitives (cf. below).

- The geometry is implemented in the package *fr.ign.cogit.geoxyegene.spatial* which is itself divided into sub-packages.

  - ❑ The package *fr.ign.cogit.geoxyegene.spatial.geomroot* has only the parent class GM_Object.
  - ❑ The package *fr.ign.cogit.geoxyegene.spatial.geomaggr* has the aggregate geometrical classes.
  - ❑ The package *fr.ign.cogit.geoxyegene.spatial.geomcomp* has the geometrical complex classes.
  - ❑ The package *fr.ign.cogit.geoxyegene.spatial.geomprim* has the classes of primitives that do not store co-ordinates directly (including GM_Point, GM_Curve and GM_Surface).
  - ❑ The package *fr.ign.cogit.geoxyegene.spatial.coordgeom* has the classes of primitives that store co-ordinates (including *DirectPosition, GM_LineString, GM_Polygon*).

- An aggregate is a collection of *GM_Object* without any internal structure. There are heterogeneous aggregates (*GM_Aggregate*), i.e. composed of different types of primitive, and homogenous aggregates (*GM_MultiPrimitive* and its sub-classes), i.e. composed of the same type of primitive (point, line, surface).

- A complex is a structured collection of *GM_Primitive*. The primitives must be connected. The class *GM_Complex* is composed of heterogeneous primitives and is not currently used. The sub-class *GM_Composite* and its sub-classes are composed of a single type of primitive and are similar to the primitive.

  - ❑ A *GM_CompositePoint* is similar to a *GM_Point*, and is, therefore, composed of just one point (not very useful! ☺)
  - ❑ A *GM_CompositeCurve* is similar to a curve: it is composed of line primitives connected so that the end point of one line is the start point for the following.
  - ❑ A *GM_Ring* is a special case of *GM_CompositeCurve* which is closed (the start point of the first primitive = end point of the last primitive).
  - ❑ A *GM_CompositeSurface* is similar to a surface: it is composed of adjacent surface primitives that do not overlap.
  - ❑ A *GM_Shell* is a special case of *GM_CompositeSurface* which is closed.

- The line primitive is *GM_Curve*. A *GM_Curve* is composed of one or more *GM_CurveSegment* objects. A *GM_CurveSegment* can be a polyline (*GM_LineString*), a series of arcs *(GM_ArcString*), a spline curve (*GM_SplineCurve*), etc. The model allows for a large number of *GM_CurveSegment* objects.

- The polyline *GM_LineString* is the only *GM_CurveSegment* implemented in GeOxygene. Current DBMSs and GISs do not have very many alternatives for storing line primitives. *GM_LineString* is, moreover, a special *GM_Curve*, composed of one and only one segment which is itself (extension of the ISO standard). This extension makes it possible to work directly and easily using instances of the class *GM_LineString*, which is the most common. However, *GM_Curve* exists and can be used.

- The boundary of a *GM_Polygon* is a *GM_Ring*.

- The modelling of surfaces is similar to the modelling of lines. The surface primitive defined in the standard is *GM_Surface*. It is composed of *GM_SurfacePatch* objects. A *GM_SurfacePatch* can be a polygon (*GM_Polygon*) or more complicated forms for three dimensional surfaces (cylinder, sphere, etc.). *GM_Polygon* is a special *GM_Surface* object composed of one and only one *GM_SurfacePatch* which is itself. This makes it possible to work directly with *GM_Polygon*. In the current version of GeOxygene, only class *GM_Polygon* can be used (*GM_Surface* cannot be used).

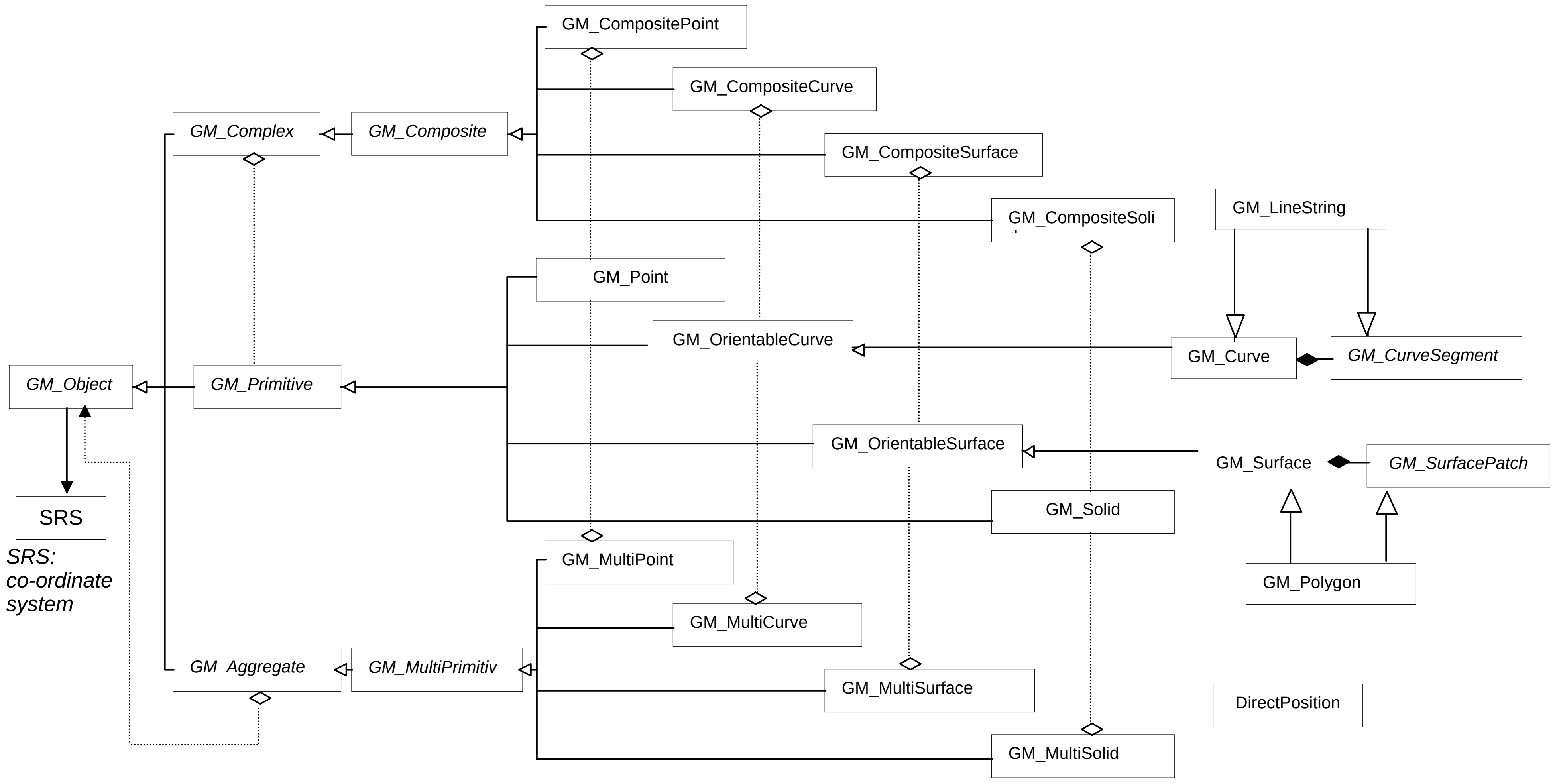
- There are two orientable primitive classes (*GM_OrientableCurve* and *GM_OrientableSurface)*. The "primitive" *(GM_Curve, GM_Polygon)* has positive orientation. Each orientable primitive is linked to its positive oriented primitive by the link *primitive*. In fact, the positive oriented primitive and the primitive are same object and may be linked to a topological primitive. The primitive is linked to its two oriented primitives by the link *proxy* (bear in mind that the positive oriented primitive is itself …).

- The class *DirectPosition* represents an X,Y,Z, table with associated methods (not described here). For two dimensional work, Z is not set. 3D primitives are not covered in this document.

- There are classes which are structures for representing the boundaries of geometrical objects (*GM_CurveBoundary* to represent the boundary of a *GM_Curve*, *GM_SurfaceBoundary* to represent the boundary of a *GM_Polygon*). They are not essential.

- Notes on the geometrical model

  - o Refer to the description of the model and ISO 19107.
  - o Look at the diagram on the following page.
  - o Use the example *TestGeomCurve* in the packag*e fr.ign.cogit.geoxygene.example*.

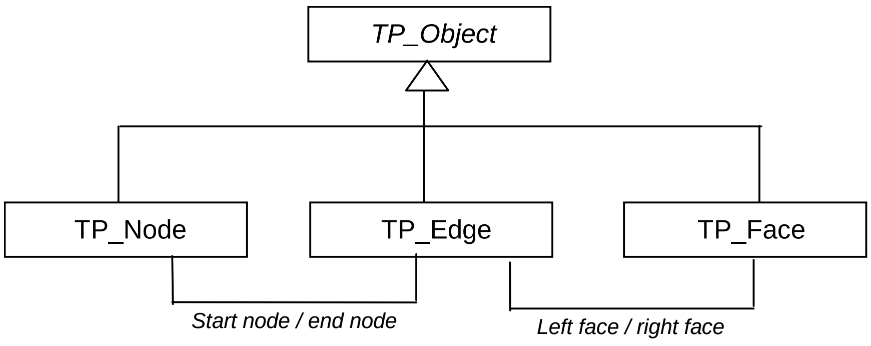


Open GIS (feature geometry) / ISO 19107: diagram of geometrical classes
(abstract classes in italics)

A.3. TOPOLOGY (package *fr.ign.cogit.geoxygene.spatial*)

- *TP_Object* is the abstract parent class for the topology in the same way as *GM_Object* was the abstract parent class for the geometry.

- The GeOxygene topology is a standard topology map: there are edges *(TP_Edge)* defined by start and end nodes (*TP_Node)* and faces *(TP_Face)*. The edges have a left face and a right face.



- The topology may be a net: there are only nodes and edges (without faces).

- The model has many classes and methods for representing and operating on the topology: directed nodes, directed edges, directed faces. The topological primitives can also be handled using polynomials. These are not essential for understanding the model and are not dealt with here.

- The topology is used to store spatial relationships, which are calculated when the geometry is used. Therefore, the long geometric algorithms become much simpler.

- There are three directed primitive classes (*TP_DirectedNode*, *TP_DirectedEdge* and *TP_DirectedFace*). The "primitive" (*TP_Node, TP_Edge, TP_Face)* has positive direction. Each directed primitive is linked to its positive directed primitive by the link *topo*. In fact, the positive directed primitive and the primitive are the same object and may be linked to a geometrical primitive. The primitive is linked to its two directed primitives by the link *proxy* (bear in mind that the positive directed primitive is itself …).

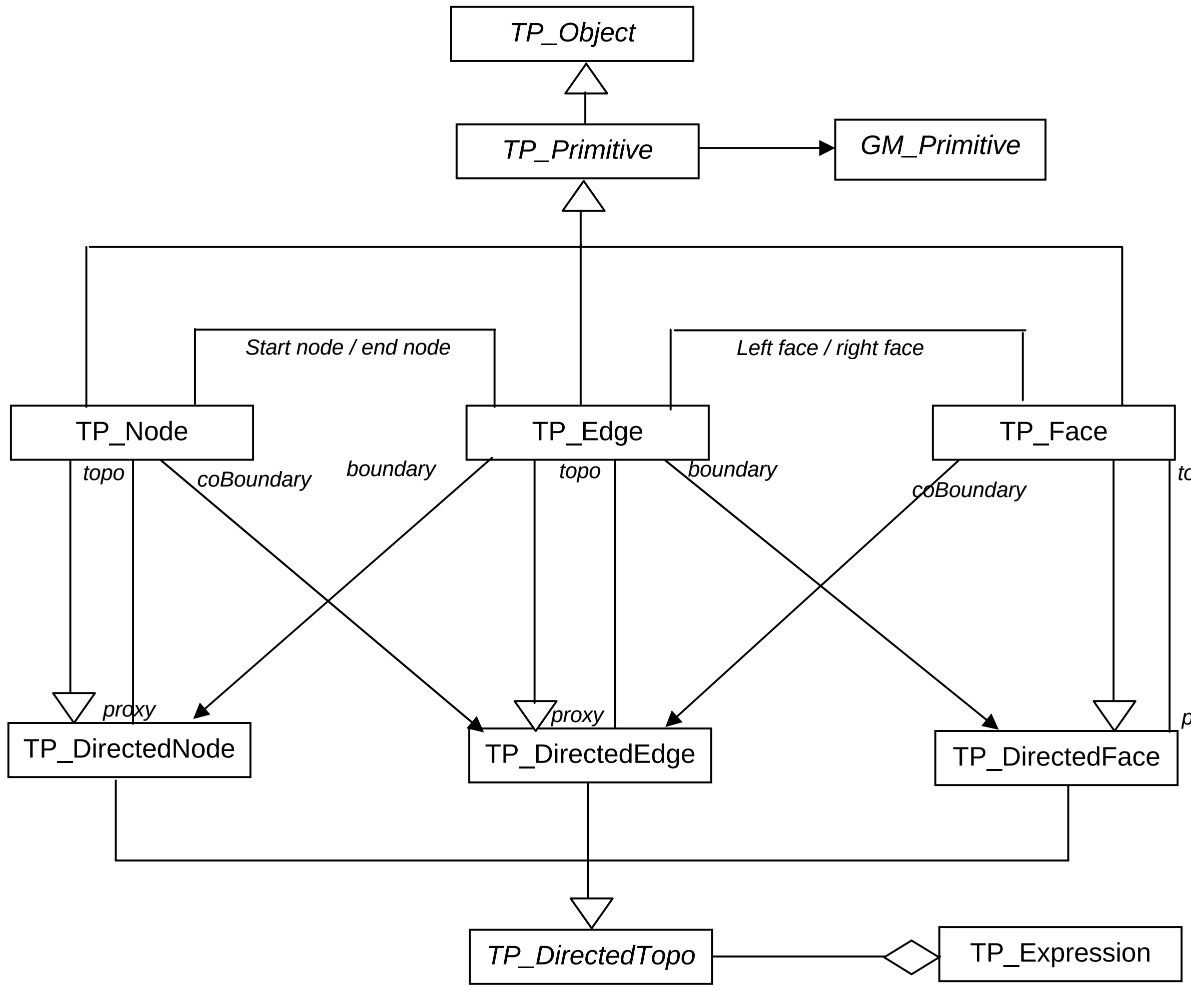
- The start node / end node links and left face / right face links define a topology map. They are not covered by the standard but they are useful for storing the relationships efficiently.

- The primitives have boundary "boundary( )" and "coBoundary( )" methods which return sets of *TP_DirectedTopo* objects. There are also methods for the next arc, previous arc, etc.

- *TP_Expression* objects are used for algebraic operations of directed primitives. This is useful, for example, for route planning. Refer to the example *TestTopo* in the packa*ge fr.ign.cogit.geoxygene.example.*

- Notes on the topological model:

    - Refer to the description of the model and ISO 19107.
    - Look at the diagram on the following page.
    - Use the example *TestTopo* in the packa*ge fr.ign.cogit.geoxygene.example*.



Open GIS (feature geometry) / ISO 19107: diagram of topology classes
(abstract classes in italics)

APPENDIX B
CODING CONVENTIONS

If you contribute to the development of GeOxygene, we should be grateful if you would follow the following coding conventions which we have adopted to make the source code coherent and easier to read and maintain.

■ *STRUCTURE AND DOCUMENTATION*

- Organise the source code in packages. A file *package.html* in the package source directory should provide a brief description of the contents and structure of the package. It will appear in the javadoc.
- One class per file. The source organised as follows:
  - Licence conditions, origin and disclaimers
  - Package name
  - Blank line
  - Imports
  - Javadoc comment /** … */. Do not forget the @author and @version tags.
  - Code
- Put a javadoc /** ... */.comment on each public method and property
  Parameters (tag @param) need not be described if they are intuitive.
- Use standard comments /* ... */ for comments that are not required for the documentation but are useful for understanding the code (details of an algorithm, notes, link to a document). Do not just describe what you are doing but why you are doing it.
- Use line comments // to comment sections of code that are a bit tricky. Ideally, avoid this. Rather than writing tricky code and writing comments, write clear code ! ;-)

  Example

  int index = -1; // -1 serves as flag meaning the index is not valid

  Rather:

  static final int INVALID = -1;
  int index = INVALID;

- One hundred characters maximum per line.
- Be spacious! Put in blank lines, use spaces to improve clarity.

■ *NAMING CONVENTIONS*

- *package*: all lower case — *likethis*
- *class*: first letter must be upper case — *LikeThis*
- *property*: first letter must be lower case — *likeThis*
- *constant (final* property): all in upper case, if necessary with underscores *LIKE_THIS*
- *method:* first letter must be lower case — *likeThis()*
- *method creating an object of type Fred:* — *void createFred()* or *void newFred( ).*
- *method converting an object to type Fred*: — *toFred( )*
- *for a property prop of type Type*
  method getting the property prop: — *getProp( )*
  method setting the property prop: — *void setProp(Type value)*
- *for a collection (set or list) thing containing objects of type Type*
  method getting the whole collection: — *Collection getThingList( )*
  method getting an indexed element (if list): — *Type getThing(int i)*
  method setting a value for index i: — *void setThing(int i, Type value)*
  method adding an element in the thing collection at the end: *void addThing(Type value)*
  method adding an element to the collection at index i: — *void addThing(int i, Type value)*

method removing an element from the collection thing:	*void removeThing(int i)*
	void removeThing (Type value)
method clearing the collection thing:	*void clearThing ( )*
method getting the size of the collection thing:	*int sizeThing ( )*
initialising an iterator in the collection:	*void initIteratorThing ( )*
next element:	*boolean hasNextThing ( )*
	Type nextThing ()

■ ***RECOMMENDATIONS***

- Imports: avoid *import package.**, say exactly what you are importing. This will help those who read your code (use the function "organize imports" under Eclipse for example).
- Use double rather than float.
- Manage exceptions cleanly, i.e. using the language elements or by creating your own exception handlers.
- The class with the method main must be used only for testing or for demonstrations.
- To compare objects, use the method *equals* rather than the operator ==.