Plate-forme

– Guide utilisateur –

Projet GeOxygene
Mars 2005
Version provisoire en français

Titre :
Plate-forme GeOxygene – Guide utilisateur.

Mots-clés :
GeOxygene, architecture, modèle, orienté-objet, plate-forme, SIG, Java, SGBD, Oracle, PostGIS, mapping objet-relationnel, services web, laboratoire COGIT.

Résumé :
Le projet open-source GeOxygene est issu des développements menés au laboratoire COGIT de l'Institut Géographique National (IGN, France) pour construire une plate-forme logicielle, ouverte et modulaire dédiée aux applications de recherche en information géographique, avec une architecture et un modèle de données communs, permettant le partage des codes, leur documentation et leur maintenance. En s'appuyant sur des technologies innovantes, GeOxygene est un prototype devant permettre le déploiement des applications de recherche sous la forme de services web. En utilisant les standards de l'ISO et de l'OGC, GeOxygene s'appuie sur un socle pérenne et interopérable.

Ce document présente la plate-forme GeOxygene et aide à sa prise en main.

Auteurs initiaux :
Thierry Badard
Arnaud Braun

Contributeurs et relecteurs :
Anne Ruas

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

TABLE DES MATIERES

1. PRESENTATION GENERALE 4
1.1. PROBLEMATIQUE ET HISTORIQUE 4
1.2. OBJECTIFS 4
1.3. ARCHITECTURE GENERALE 5
1.4. STRUCTURE DU CODE 7

2. INSTALLATION ET CONFIGURATION 9
2.1. PREREQUIS 9
2.2. COMPILATION DU PROJET 9
2.3. CONFIGURATION 10

3. PRISE EN MAIN 12
3.1. EXEMPLE SIMPLE DE MAPPING 12
3.2. CHARGEMENT DE DONNEES GEOGRAPHIQUES 13
3.3. PREMIER EXEMPLE DE MANIPULATION DE DONNEES GEOGRAPHIQUES 14
3.4. IMPLEMENTATION D'UN MODELE COMPLEXE 14
3.5. LA MANIPULATION DES DONNEES 15
3.6. LE VISUALISATEUR 16
3.7. LE BROWSER 16

ANNEXE A : ELEMENTS DE COMPREHENSION DU SCHEMA 17
A.1. LE SCHEMA GEOGRAPHIQUE (package « fr.ign.cogit.geoxygene.feature ») 17
A.2. LA GEOMETRIE (package « fr.ign.cogit.geoxygene.spatial ») 17
A.3. LA TOPOLOGIE (package « fr.ign.cogit.geoxygene.spatial ») 21

ANNEXE B : CONVENTIONS DE CODAGE 23

1. PRESENTATION GENERALE

1.1. PROBLEMATIQUE ET HISTORIQUE

Le développement d'applications liées à l'information géographique se heurte à de nombreux problèmes :

- L'absence d'interopérabilité entre les modèles de données des différents SIG, et ceci malgré les efforts de standardisation de l'ISO et de l'OGC. Une application développée avec un modèle non standard a ainsi peu de chance d'être réutilisable sur différents systèmes sans modifications parfois profondes.
- Les langages de programmation liés aux logiciels SIG sont bien souvent des langages propriétaires ; ainsi le partage de code entre les différents SIG est impossible, et les utilisateurs sont très dépendants des évolutions technologiques de l'éditeur.
- Sans des compléments onéreux, les logiciels SIG ne sont pas ouverts sur le web. De plus, si de tels compléments permettent l'accès en ligne aux données, l'appel de processus à distance n'est quant à lui pas toujours possible (notion de service web de traitement).
- Les logiciels SIG ne sont pas de véritables SGBD (systèmes de Gestion de Bases de Données), et des problèmes résolus par ces derniers ne sont pas toujours adressés par les SIG (accès concurrent, sécurité, etc.).

Pour surmonter ces problèmes, des technologies ont récemment émergé, notamment en génie logiciel : des langages orienté-objets ouverts sur le web (comme Java), des techniques d'analyse et de conception orientée-objet, basées sur la notion de réutilisabilité des composants (comme UML), des SGBD relationnels intégrant des fonctionnalités objets, et permettant le stockage de l'information géographique grâce à des moteurs géographiques (comme Oracle, PostGIS, etc.), des langages structurés pour l'échange d'information sur les réseaux (comme XML), et des technologies pour les services web permettant la description et l'appel de procédures à distance dans des environnements informatiques hétérogènes et distribués (comme WSDL et SOAP).

Basée sur ces technologies, une nouvelle plate-forme baptisée GeOxygene (initialement nommée OXYGENE) a été conçue et développée dans le cadre des activités de recherche du laboratoire COGIT de l'Institut Géographique National (IGN) en France (Badard et Braun, 2004)[1]. Elle a succédé à une ancienne plate-forme développée dans les années 90 : $GéO_2$, une sur-couche géographique construite au dessus du SGBD orienté-objet O_2. Elle fournit un socle pérenne et robuste au laboratoire COGIT pour rendre possible notamment les recherches concernant l'aide à la consultation et à la diffusion des données, des mises à jour et des traitements.

Le laboratoire COGIT a décidé début 2005 de reverser une partie des codes sources de GeOxygene sous la forme d'un projet open-source.

1.2. OBJECTIFS

Le but initial de GeOxygene était de fournir un cadre ouvert, modulaire et interopérable pour le développement des applications de recherche au laboratoire COGIT, pour les actions de recherche sur la représentation multiple, la qualité des données pour les études de risques, la consultation et la diffusion des données, en s'appuyant sur un modèle de données commun et une architecture commune, permettant le partage des codes, leur documentation et leur maintenance. L'objectif était (et est toujours) la pérennisation et la valorisation des travaux de recherche. En s'appuyant sur des technologies innovantes, GeOxygene est un prototype devant permettre le déploiement des applications de recherche sous la forme de services web. En s'appuyant sur les standards de l'ISO et de l'OGC, GeOxygene se place dans un contexte d'interopérabilité totale.

Le but d'un reversement open-source d'une partie des codes de GeOxygene n'est pas de concurrencer ou de remplacer l'un des nombreux projets open-source déjà existants et relatifs à

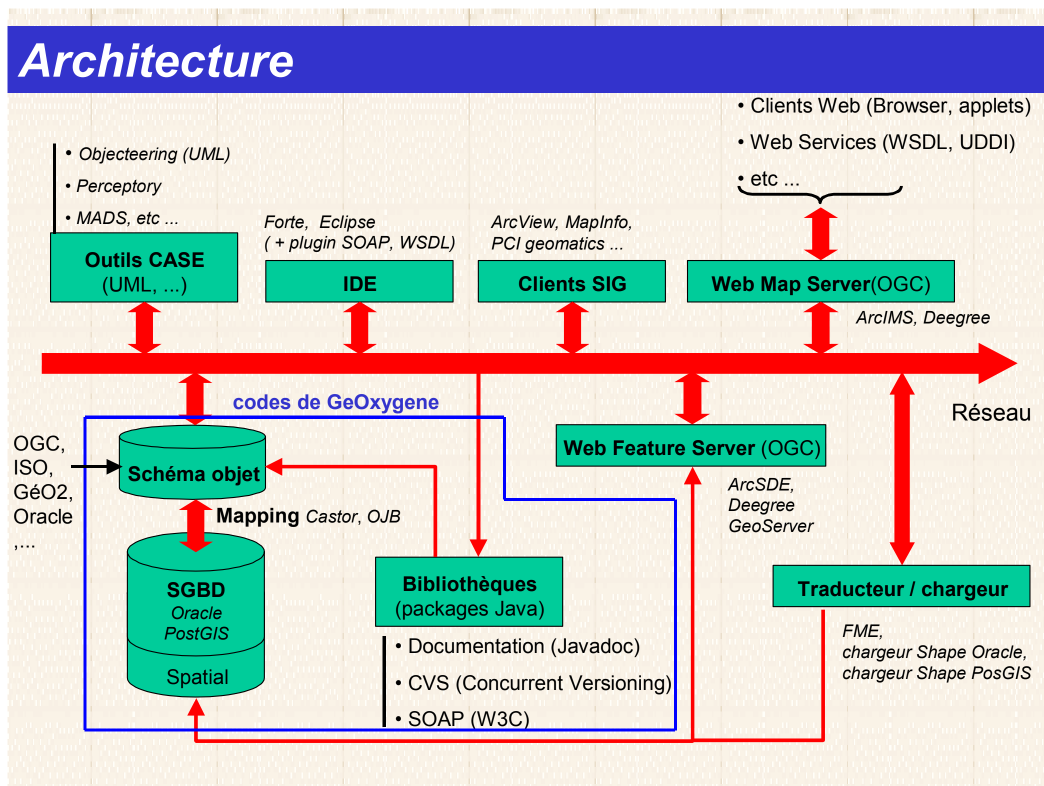
[1] (Badard et Braun, 2004) Badard, T., Braun, A. *OXYGENE: A Platform for the Development of interoperable Geographic Applications and Web Services*. Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA'04), IEEE Press, August 30 - September 03, 2004, Zaragoza,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

l'information géographique, mais bien d'apporter une nouvelle brique complémentaire à la construction des solutions open-source. Le projet GeOxygene est certes moins mature et étoffé que des projets comme Geotools (http://www.geotools.org) ou Deegree (http://deegree.sourceforge.net), car il est moins ancien et a mobilisé beaucoup moins de ressources de développement, mais même dans son état actuel, GeOxygene peut avoir son utilité comme point de départ crédible d'une implémentation vraiment complète et rigoureuse de la norme ISO 19107 et des spécifications relatives à la notion de feature (Features, Feature Collection et Relationships between features), ainsi que comme point de départ pour l'utilisation de JDO pour les bases de données géographiques. GeOxygene constitue ainsi une brique pour la construction de solutions logicielles, s'appuyant sur des données géographiques, réellement interopérables car implémentant l'ensemble des spécifications OGC et normes ISO. Afin de s'intégrer dans la partition des projets open-source existants, réalisant actuellement un effort de rationalisation de leurs efforts au travers du projet GeoAPI, GeOxygene devra lui aussi, dans un futur proche, implémenter les interfaces (Java) définies par ce projet.

GeOxygene ne s'adresse pas directement à des utilisateurs finaux, car il ne s'agit pas d'un produit clé en main où tout est géré de façon transparente, via des interfaces graphiques. Il concerne plutôt des utilisateurs avertis, développeurs Java ou étudiants-chercheurs, ayant à écrire des applications géomatiques. Il peut être à ce titre considéré comme un « noyau » pour le développement de telles applications.

1.3. ARCHITECTURE GENERALE

L'architecture de GeOxygene est entièrement modulaire (cf. figure ci après). Elle est construite autour du réseau pour permettre la communication entre les composants et le déploiement des développements. Ses composants logiciels sont indépendants et interopérables, et sont exploités selon les usages auxquels ils sont les mieux adaptés : on utilise ainsi un SGBD pour le stockage des données et uniquement pour cela, les opérateurs géométriques sont implémentés dans un autre composant. Cela est fait pour éviter la construction d'une architecture de type « couteau suisse » où un composant non dédié, en arrive à réaliser des choses pour lesquelles il n'a pas été nativement conçu. De telles utilisations conduisent très vite à des pertes importantes de performance et à l'impossibilité de maintenir les développements au gré des départs des développeurs.



Architecture de la plate-forme GeOxygene avec des exemples de composants logiciels

Les composants de l'architecture sont les suivants :

- Développée principalement en Java, langage orienté-objet choisi principalement pour sa portabilité, et s'appuyant en partie sur des composants provenant de projets open-source, la plate-forme GeOxygene permet de modéliser et manipuler toutes les facettes de l'information géographique (sémantique, géométrie, topologie, métadonnées) au travers d'un schéma orienté-objet compatible avec les spécifications et normes de l'OGC et de l'ISO. Les codes Java de ce schéma objet sont inclus dans la distribution open-source.
- Les données sont stockées dans un SGBD relationnel (actuellement, Oracle avec sa cartouche spatiale et le SGBD open source PostGIS – sur-couche géographique de PostgreSQL – sont supportés) afin d'assurer des temps de réponse et d'accès rapides. Des scripts SQL permettant de manipuler le SGBD sont inclus dans la distribution open-source.
- Un « mapping » (i.e. « telle classe Java correspond à telle table ») flexible entre le schéma objet et les tables relationnelles est assuré par un composant open-source, OJB (ObJect relational Bridge, de la fondation Apache). Les utilisateurs ont ainsi une perception entièrement objet de l'information qu'ils manipulent : ils modélisent leur application en UML et les codent en Java, en s'appuyant sur le modèle objet conforme ISO/OGC offert par la plate-forme. Des fichiers de mapping (écrits en XML) sont inclus dans la distribution open-source, ainsi que tous les codes permettant l'utilisation de tels fichiers.
- Afin d'assurer l'indépendance des développements, les opérateurs géographiques sont codés dans des bibliothèques séparées. Ce sont des algorithmes provenant du Web (ex. : la bibliothèque open-source JTS pour les calculs de géométrie algorithmique.) ou d'anciens développements déjà réalisés au COGIT. Ils sont éventuellement codés en C, C++, Fortran ou Ada, mais sont interfacés avec Java via la technologie JNI (Java Native Interface). Les fonctions d'interfaçage avec la bibliothèque JTS sont incluses dans la distribution open-source.

L'architecture de GeOxygene telle que mise en œuvre au laboratoire COGIT, inclut également de nombreux autres outils. Ils ne sont pas inclus dans la distribution open-source, chaque utilisateur de GeOxygene pouvant se bâtir sa propre architecture. Ces outils permettent :

- la génération automatique de documentation (Javadoc),
- la partage coopératif et cohérent des codes sources (CVS : Concurent Versioning System),
- le chargement des données géographiques, leur visualisation et leur analyse avec la connexion de clients SIG,
- la modélisation des applications avec un AGL (Atelier de Génie Logiciel, ou outil CASE),
- le développement des applications avec un IDE (Environnement de Développement Intégré). L'IDE utilisé est Eclipse (fruit du projet open-source du même nom, basé sur une contribution de IBM), avec ses nombreux plugins, notamment UML, qui en font presque un AGL.

De plus, 2 outils additionnels ont été développés et inclus dans la distribution pour tirer pleinement parti du point de vue orienté-objet sur les données géographiques fourni par le schéma de GeOxygene : un visualisateur d'objets géographiques (basé sur le projet open-source Geotools, version 1) et un navigateur graphique générique d'objets (pour afficher les états dans le temps des objets et déclencher dynamiquement des méthodes). La figure ci-après fournit une illustration de ces deux composants :

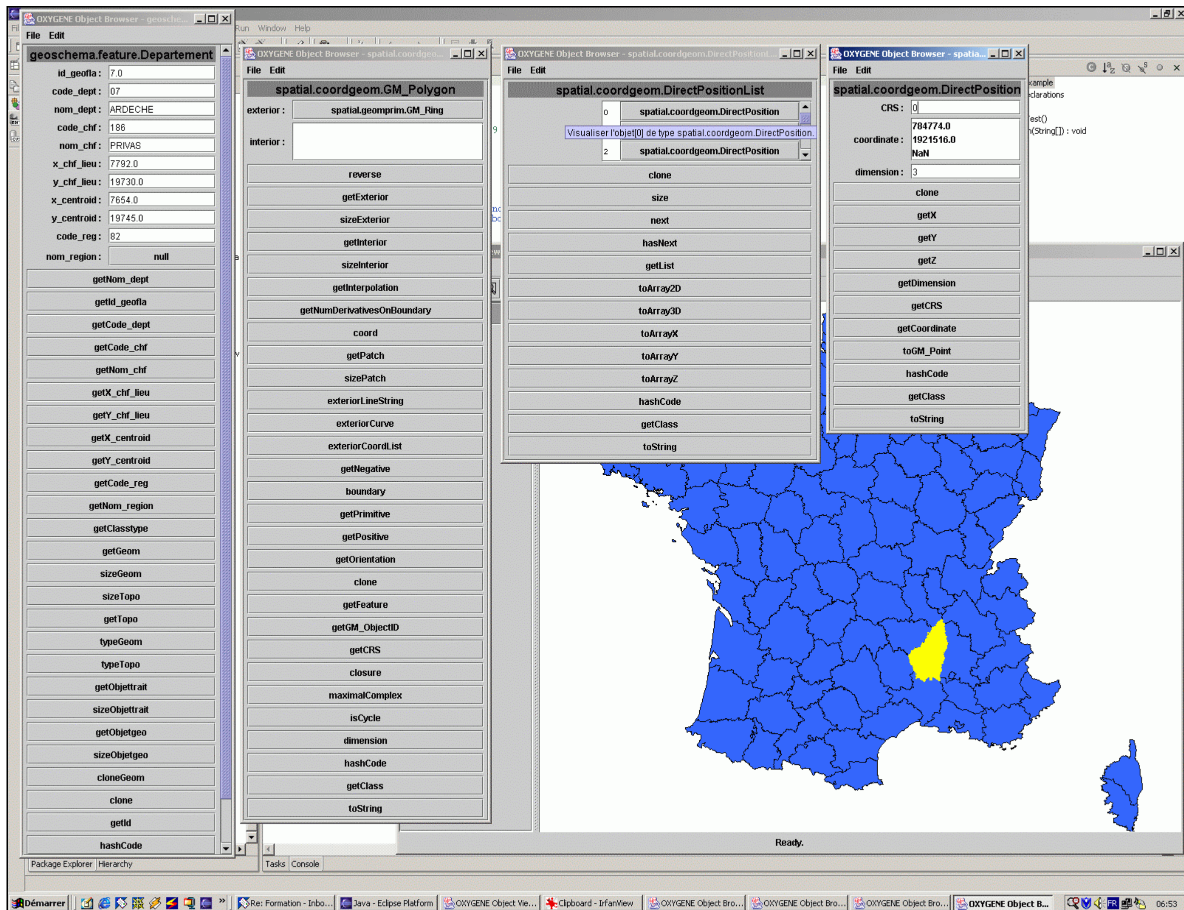


Illustration du visualisateur et du navigateur graphique d'objets
de la plate-forme GeOxygene

Enfin, conçue dès le départ pour permettre l'intégration et l'exploitation de technologies comme SOAP (Simple Object Access Protocol) et WSDL (Web Services Description Language) édictées par le World Wide Web Consortium, ou de services en ligne spécifiés par l'OGC, comme WFS (Web Feature Service) et WMS (Web Map Service), cette plate-forme permet de déployer les développements réalisés en son sein, sous la forme de véritables services web (Badard et Braun, 2003)[2].

1.4. STRUCTURE DU CODE

Les codes de GeOxygene sont structurés de la manière suivante :

- Le dossier « src » comprend des implémentations de normes et spécifications, et différents outils :
  - o norme ISO 19107 (géométrie et topologie),
  - o norme ISO 19109 (métamodèle pour construire les schémas applicatifs),
  - o norme ISO 19115 (métadonnées), implémentation partielle,
  - o spécification abstraite OGC « Features » (objets géographiques),
  - o outils d'accès aux bases de données géographiques par mapping objet-relationel,
  - o utilitaires :
    - → indexation spatiale en mémoire
    - → chargement de données sous Oracle et PostGIS
    - → géométrie algorithmique
    - → navigateur graphique d'objets
    - → visualisateur

[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.

- exemples et tests d'utilisation.

- Le dossier « data », accueille les classes géographiques propres au schéma de chaque utilisateur.

- Les codes des applications de chaque utilisateur seront hébergées dans un dossier propre à chaque utilisateur. A titre d'exemple, un exemple de code de filtrage de Douglas-Peucker a été mis dans le dossier « src », dans le package *fr.ign.cogit.geoxygene.generalisation.*

Les scripts SQL pour l'accès au SGBD : ils se trouvent dans un dossier « sql » avec 2 sous-dossiers relatifs aux 2 SGBD supportés : « oracle » et « postgis ».

Les fichiers XML pour le mapping objet-relationnel dits « fichiers de mapping » se trouvent dans le dossier « mapping » avec un sous-dossier relatif à l'outil actuellement supporté : « ojb1 ».

Les fichiers contenus dans les sous répertoires « castor » des répertoires « sql » et « mapping » correspondent à une mise en œuvre de l'outil Castor (http://castor.codehaus.org/index.html, moins performant qu'OJB) qui n'est plus utilisée et ne fonctionne pas en l'état. L'utilisation à nouveau de cet outil pourrait néanmoins être possible grâce à l'architecture modulaire de GeOxygene, via une retouche conséquente des codes contenus dans les deux sous-répertoires.

2. INSTALLATION ET CONFIGURATION

2.1. PREREQUIS

■ *ENVIRONNEMENT*

GeOxygene fonctionne et a été testé sous Solaris, Linux et Windows.

■ *JAVA*

Doivent être installés:

- Java Developpment Kit (JDK) Standard Edition (J2SE) 1.4 au minimum
- Java Developpment Kit (JDK) Entreprise Edition (J2EE) 1.3.1

Note : le support J2EE 1.4 n'a pas été testé.

■ *ORACLE*

Doivent être disponibles sur le disque:

- la bibliothèque SDOAPI (sdoapi.zip ou sdoapi.jar suivant la version d'Oracle)
- le driver JDBC Oracle (classes12.zip par exemple)

Attention, ces bibliothèques sont, pour l'instant, nécessaires à la compilation du projet ! Elles ne sont bien entendu pas nécessaires à l'exécution si vous utilisez une version compilée du projet et que vous utilisez le SGBD PostGIS …

2.2. COMPILATION DU PROJET

2 méthodes sont détaillées ci-après afin de réaliser la compilation du projet :

- Soit en utilisant le gestionnaire de projet Ant (http://ant.apache.org) développé par la fondation Apache.
- Soit en utilisant l'IDE open-source Eclipse (http://www.eclipse.org).

■ *COMPILATION AVEC APACHE ANT*

Editez le fichier « build.properties » présent à la racine du projet. Changez le chemin (`j2ee.dir`) où se trouve le répertoire d'installation de J2EE en fonction de votre système. Enregistrez vos modifications et fermez le fichier.

Modifiez votre variable CLASSPATH afin que les bibliothèques Oracle soient disponibles au compilateur Java ou copiez les deux bibliothèques nécessaires dans le sous répertoire « lib » du projet.

La cible Ant par défaut réalise la compilation du projet. Aussi pour compiler GeOxygene, vous n'avez qu'à taper :

```
% ant
```

à la racine du projet (le répertoire qui contient le fichier build.xml) depuis la ligne de commande.

D'autres cibles sont disponibles, tapez :

```
% ant -p
```

pour en afficher la liste et obtenir le descriptif de ce qu'elles réalisent.

A noter, vous pouvez également utiliser les fichiers build.xml et build.properties depuis Eclipse afin de compiler le projet, générer la javadoc, etc.

■ *CONFIGURATION D'ECLIPSE*

Création du projet

Menu File -> New -> Project, un assistant apparaît.
A gauche : "Java", à droite "Java Project", Next
Project Name : "GeOxygene"
Project Contents : décocher "Use defaults"
Choisir le chemin du répertoire "GeOxygene" existant
(par exemple : D:\users\foo\GeOxygene).
Next. Un certain temps d'analyse s'écoule.

Configuration de la compilation

"Default output folder" (en bas) : GeOxygene/classes
Le sélectionner à l'aide "Browse", éventuellement créer ce répertoire à l'aide "Create new folder".

Onglet "Source" : on doit trouver les 2 répertoires suivants, à ajouter à l'aide de "Add Folder".

- GeOxygene/src
- GeOxygene/data

Répondre "Yes" à l'éventuelle question " Do you want to remove the project as source folder".

Onglet "Libraries" :
A l'aide de "Add JARs", choisir dans le répertoire GeOxygene/lib :

- ojb-1.0.jar (ou autre version)
- batik-all.jar
- pg74.1jdbc3.jar
- postgis.jar
- jts.jar
- geotools1-cogit.jar (ou autre version)

A l'aide de "Add External JARs" :

- choisir "j2ee.jar" dans le répertoire LIB du répertoire d'installation de J2EE (dépend de l'installation, mais il s'agit souvent C:\j2sdk1.3.1\lib sous Windows ou /usr/j2sdk1.3.1/lib sous Unix/Linux)
- choisir les fichiers sdoapi.zip et classes12.zip (ou autres, en fonction de la version d'Oracle utilisée) dans le répertoire où ces bibliothèques sont disponibles (cf. section 2.1).

Cliquez sur "Finish" … et la compilation commence.

2.3. CONFIGURATION

■ *CREATION DES TABLES UTILES DANS LE SGBD*

Avant toute utilisation, il faut exécuter dans le SGBD le script *superscript_ojb.sql* qu'on trouvera dans le répertoire *sql* du SGBD (Oracle ou PostGIS). Il crée les tables nécessaires au fonctionnement de la plate-forme (tables nécessaires à OJB et pour Oracle table nécessaire à l'exécution de certaines requêtes).
Le script *resultat.sql* peut être exécuté : il crée une table permettant de stocker des résultats.

■ *CONFIGURATION DU FICHIER OJB.PROPERTIES*

Editer le fichier *OJB.properties* qu'on trouvera à la racine des codes sources de GeOxygene, à savoir dans le répertoire « src ».
Indiquer l'emplacement du fichier de mapping *repository.xml :* (attention chemin ABSOLU et non relatif).

Par exemple :

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

■ *CONFIGURATION DU FICHIER REPOSITORY_DATABASE.XML*

Ce fichier se trouve dans le sous répertoire *mapping/ojb1.* Il contient les paramètres de connexion au SGBD (nom, mot de passe, etc.…). L'éditer et l'adapter à votre SGBD et à votre connexion (nom du serveur, nom d'utilisateur, mot de passe). On trouve 2 exemples de configuration, l'un pour Oracle et l'autre pour PostGIS.

3. PRISE EN MAIN

3.1. EXEMPLE SIMPLE DE MAPPING

Cette section décrit un exemple simple de création et d'utilisation d'une classe persistante. Le but étant de comprendre le fonctionnement d'un code, aucune notion de géométrie ou de géographie n'est abordée ici.

La classe "fr.ign.cogit.geoxygene.example.tutorial.MyClass"
Examiner la classe *MyClass* du package *fr.ign.cogit.geoxygene.example.tutorial.*
C'est une classe comme une autre. Rien n'indique ici qu'elle sera persistante (i.e. stockable dans le SGBD) : la persistance est dite « non-intrusive », c'est-à-dire qu'elle n'intervient pas dans le code. Il est néanmoins nécessaire d'avoir un champ correspondant à l'identifiant (id).
Cette classe est définie comme un « bean », c'est-à-dire que chaque attribut est accompagné de 2 méthodes « get » et « set » appelées accesseurs, pour le manipuler. C'est la manière la plus propre de coder.

Création de la table correspondante dans le SGBD
Examiner dans le dossier */sql/oracle ou /sql/postgis*, le script SQL de création de table *maclasse.sql*. Exécuter ce script dans votre SGBD.

Mapping de la classe MaClasse
Examiner le fichier XML *repository_maclasse.xml* dans le répertoire */mapping/ojb1.* Il indique les correspondances entre le monde Java et le monde du SGBD. La syntaxe est assez intuitive. Ce fichier sera utilisé par OJB.

Configuration du fichier de configuration d'OJB
Editer le fichier *OJB.properties* qu'on trouvera à la racine des codes de GeOxygene (i.e. dans le répertoire src de l'arborescence).
Indiquer l'emplacement du fichier de mapping *repository.xml :*
Attention, il faut indiquer un chemin ABSOLU et non relatif ver ce fichier !
Par exemple : *RepositoryFile=D:/users/duschmok/GeOxygene/mapping/ojb1/duschmok/repository.xml*

Configuration des fichiers de mapping
Pour initialiser la connexion au SGBD et charger les informations relatives au mapping, un fichier nommé *repositroy.xml* est lu par OJB.
Ce fichier se trouve dans le répertoire */mapping/ojb1.* Son chemin est rigoureusement défini dans le fichier *OJB.properties .* Examinez le.
Ce fichier pointe vers d'autres fichiers xml (repository_*.xml) qui contiennent eux-mêmes les informations de mapping.
Deux appels doivent ABSOLUMENT être décommentés :

- *&database* qui pointe vers *repository_database.xml*. Ce dernier fichier contient les paramètres de connexion au SGBD (nom, mot de passe, etc.…). L'éditer et l'adapter à votre SGBD et à votre connexion (nom du serveur, nom d'utilisateur, mot de passe). On trouve 2 exemples de configuration, un pour Oracle, l'autre pour PostGIS.
- Vient ensuite la liste des fichiers de mapping à utiliser. Un certain nombre de fichiers existent déjà, nécessaires au fonctionnement de GeOxygene. C'est alors que *&tutorial* doit pointer vers *repository_maclasse.xml* et doit être décommenté.

Exécution du code applicatif
Il s'agit maintenant de "faire tourner" cette classe MyClass. C'est le but de la classe *fr.ign.cogit.geoxygene.example.tutorialTestMyClass.*

Regardez les commentaires dans le code.

Ce code montre les principales fonctionnalités de OJB : rendre des objets persistants, charger en Java des objets venant de la base, faire une requête OQL.

Les point-clés à retenir de cet exemple sont :

- `Geodatabase db = new GeodatabaseOjbOracle();`

  Cette ligne initialise la connexion au SGBD et charge les informations de mapping. Concrètement, à cet endroit le fichier *repository.xml* est lu et analysé par le moteur objet-relationnel OJB.

- `db.begin();`

  Cette ligne initie une transaction.

- `db.commit();`

  Cette ligne ferme une transaction en la validant.

- `db.makePersistent(obj);`

  Cette ligne rend un objet (dont la classe a été définie dans le fichier de mapping) persistant dans la base.

- `db.loadAll(classe);`

  Cette ligne charge tous les objets d'une classe, autrement dit les enregistrements du SGBD sont traduits en objets Java. Le « load » se décline sous différents formes :

  - `db.load(classe, id);` → pour charger un seul objet d'une classe par son identifiant.
  - `db.loadAllFeatures(classe);` → pour charger tous les objets <u>géographiques</u> d'une classe.
  - `db.loadOQL(query, obj);` → pour charger avec une requête OQL (un exemple se trouve dans le code, dans la procédure « interroge »). OQL est un langage d'interrogation des bases de données, similaire à SQL, mais on travaille directement sur les classes Java (i.e. sur des objets) au lieu de travailler sur les tables du SGBD.

3.2. CHARGEMENT DE DONNEES GEOGRAPHIQUES

Chargement de données géographiques dans le SGBD, à partir de fichiers dans un format SIG (ESRI Shapefile par exemple)
Un tel chargement s'effectue avec l'outil de votre choix : FME par exemple, ou des chargeurs directs de fichiers SIG dans le SGBD tels que ceux fournis par Oracle ou PostGIS (shp2pgsql).

Le programme de création des classes java et des fichiers de mapping
Exécutez le programme *fr.ign.cogit.geoxygene.appli.Console*
Choisir « SQL → Java ».
Choisir dans l'interface la table chargée dans le SGBD.
Cliquer OK en acceptant les différents choix proposés.

Un message apparaît, qui demande de compiler les classes générées. En effet, une classe Java a été générée dans le répertoire proposé (*geoxygene.geodata* par défaut, dans le dossier *data).* Editer cette classe et la compiler. On voit qu'elle reprend la structure de la table. Remarquer l'attribut *geom* de type *GM_Object* pour représenter la géométrie.

Editer le fichier *repository_geo.xml (*ou un autre nom si vous avez indiqué un autre nom dans l'interface de la console) qui se trouve dans le répertoire */mapping/ojb1/.* Ne pas oublier de faire pointer le fichier racine *repository.xml* vers ce fichier *repository_geo.xml* (i.e. : l'appel *geo* doit pointer vers *repository_geo.xml* et *&geo* doit être <u>décommenté</u>). Remarquer le mapping de l'attribut portant la géométrie.

Mise en forme des données dans le SGBD
Choisir « Manage Data ».
Pour Oracle : le programme propose de générer des identifiants, acceptez ! Le programme va en fait créer une colonne COGITID sur la table, et la définir comme clé primaire.
Le programme propose aussi d'homogénéiser les géométries, acceptez également ! En effet, parfois on trouve dans les données des géométries composites dues au effets de bords et aux défauts de certains traducteurs. Ceci va être corrigé ici. Ca peut être un peu long.

Si la table contient beaucoup de données, il est probable que le programme manque de mémoire. Il faudra alors relancer *Console* avec l'option « magique » -Xmx512M (*java -Xmx512M appli.Console)*, par exemple ;-), qui augmente la taille de la mémoire allouée à la machine virtuelle. Il faudra revenir directement à cette étape.

Enfin, on propose de calculer un index spatial (pour Oracle et PostGIS) et une emprise (pour Oracle) : là aussi acceptez !

Cliquer OK pour lancer l'exécution.

Différence avec l'exemple précédent
Dans l'exemple 3.1, nous sommes partis du code Java pour générer du code SQL. C'est une logique qui va de la conception (conception bien légère ici, puisque le modèle ne comporte qu'une classe) vers l'implémentation. Dans l'exemple 3.2, nous sommes partis d'une table SQL pour générer une classe Java. C'est une logique de chargement de données. C'est dans ce cas et uniquement dans ce cas que l'on procède aux étapes précédentes.

3.3. PREMIER EXEMPLE DE MANIPULATION DE DONNEES GEOGRAPHIQUES

Charger une table de tronçons, nommée *Troncon_route,* de n'importe quel jeu de données . Vous en savez maintenant assez pour comprendre et exécuter l'exemple *FirstExample* du package *fr.ign.cogit.geoxygene.example*. Toutes les explications se trouvent dans le code. Eventuellement, changer le nom de la classe de test dans le code (variable *nomClasse*).

Attention : l'appel au fichier de mapping *repository_result.xml* doit être <u>décommenté</u>. Ce fichier comporte le mapping de la classe *fr.ign.cogit.geoxygene.example.Resultat* qui va servir dans cet exemple. Il faudra aussi exécuter le script SQL *resultat.sql* qui crée la table correspondante.

Cette classe comporte un constructeur, qui va initialiser le mapping. Eventuellement, changer `GeodatabaseOjbOracle()` en `GeodatabaseOjbPostgis()` selon le SGBD que vous utilisez.

Le premier exemple (`void exemple1()`) charge un objet par son identifiant, récupère sa géométrie, crée un buffer autour de cette géométrie, puis crée un nouvel objet de la classe *Resultat,* lui affecte la géométrie bufférisée, et rend cet objet persistant.

Le second exemple (`void exemple2()`) charge tous les objets de la classe, pour chacun d'eux récupère la géométrie, calcule un filtrage de Douglas-Peucker, crée un nouvel objet de la classe *Resultat,* lui affecte la géométrie filtrée, et rend cet objet persistant.

3.4. IMPLEMENTATION D'UN MODELE COMPLEXE

Considérons le modèle implémenté dans le package *fr.ign.cogit.geoxygene.example.relations.* Ce modèle présente tous les cas complexes que l'on peut rencontrer :

- Héritage
- Relation 1-1, mono- ou bi-directionnelle.
- Relation 1-n, mono- ou bi-directionnelle.
- Relation n-m, mono- ou bi-directionnelle.

Les relations peuvent être persistantes ou non.

On insiste sur le fait que l'implémentation des relations avec le langage Java est délicate, surtout dans le cas de relations bi-directionnelles.

Le modèle
Aucun schéma UML n'a été dessiné car le modèle est très simple.

- Les classes *AAA* et *BBB* héritent toutes deux de la classe abstraite *ClasseMere*.
- Les classes *AAA* et *BBB* possèdent des relations bidirectionnelles 1-1, 1-N et N-M.
- La classes *AAA* possède des relations mono-directionnelles vers *BBB* (i.e. depuis *BBB* on ne peut pas remonter ces relations) : 1-1, 1-N et N-M.

Implémentation du modèle en Java
Ce modèle est implémenté dans le package *fr.ign.cogit.geoxygene.example.relations.* Examiner les codes des classes *ClasseMere, AAA* et *BBB*.

Règles pour le passage d'UML à Java:
- Pas de problème pour l'héritage en java (il n'y a pas d'héritage multiple ici !).
- L'implémentation des relations en Java est délicate. Elle se fait sous forme d'attributs.

  - Une relation 1-1 s'implémente comme un attribut de type « la classe en relation ».
  - Une relation 1-n ou n-m s'implémente avec une liste d'objets de type « la classe en relation ».

- Les « accesseurs » (méthodes get, set et add) permettent de gérer la bi-direction des relations. Le codage n'est pas trivial, bien examiner les codes et faire du copier/coller si on a de telles relations à coder.
- Les classes au sommet d'une hiérarchie doivent avoir un attribut "id" pour gérer leur persistance. Cet attribut sera leur identifiant. Il se transmet par héritage sur les classes filles.

Implémentation du modèle en SQL
Examiner et exécuter le script *init_relations_AAA_BBB.sql* dans le SGBD.

Implémentation du mapping
Examiner le fichier *repository_AAA_BBB.xml.* Essayer de comprendre sa structure et comment sont implémentées les relations. Se référer à la documentation d'OJB. La question de ne mapper les relations bi-directionnelles que dans un sens est assez délicate à comprendre : cela influe sur les performances et diminue le nombre de jointures au chargement.

Programme applicatif
Quatre programmes de test du package *fr.ign.cogit.geoxygene.example.relations* exploitent ce modèle :

- Un pour tester les relations mono-directionnelles non persistantes
- Un pour tester les relations mono-directionnelles persistantes
- Un pour tester les relations bi-directionnelles non persistantes
- Un pour tester les relations bi-directionnelles persistantes

3.5. LA MANIPULATION DES DONNEES

Le package *fr.ign.cogit.geoxygene.datatools* contient les classes nécessaires à la manipulation des données. Deux classes sont particulièrement importantes à connaître pour l'utilisateur :

L'interface *Geodatabase*
- *Geodatabase* est une interface qui représente une connexion à un SGBD via un mappeur objet-relationnel. Cette classe permet d'initialiser des transactions, de charger des données persistantes, de rendre des données persistantes, de faire des requêtes, de valider des modifications dans la base (commit) ou de les annuler (abort). Bref, cette interface permet de réaliser tout ce qui est en interaction avec le SGBD. Cette interface est instanciée par une classe concrète qui dépend du SGBD et du mappeur objet-relationnel.

Par exemple :

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

où *GeodatabaseOjbOracle* est la classe pour OJB et Oracle.

L'interface *Metadata*
Metadata représente les métadonnées sur le mapping objet-relationnel : quelle table correspond à quelle classe, et aussi l'emprise pour les données géographiques.

Exemple d'utilisation
Utiliser l'exemple *TestGeodatabase* du package *fr.ign.cogit.geoxygene.example* pour avoir un éventail des possibilités des interfaces *Geodatabase* et *Metadata.*

3.6. LE VISUALISATEUR

Un visualisateur d'objets géographiques a été développé pour GeOxygene. On le lance comme ceci :

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

où "db" représente la *Geodatabase* active.

Ensuite, on demande au visualisateur d'afficher des collections de FT_Feature par la commande :

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

où « collection » est une FT_FeatureCollection, et « nom » un String qui sera le nom (ou titre) de la collection affichée dans le visualisateur.

L'utilisation est assez intuitive. On peut commander l'affichage d'une classe depuis le code ou directement depuis l'interface.

On peut dans le visualisateur sélectionner des objets et lancer le browser pour voir leurs attributs, lancer des méthodes et naviguer dans le modèle orienté-objet.

Utiliser l'exemple *TestViewer* du package *fr.ign.cogit.geoxygene.example* pour avoir un éventail des possibilités du visualisateur.

3.7. LE BROWSER

Un navigateur (browser en anglais) graphique d'objet a été développé pour GeOxygene. Il permet de profiter pleinement de l'aspect orienté-objet de l'information stocké dans la plate-forme.

Il peut se lancer depuis le visualisateur après sélection d'objet. On peut aussi le lancer directement de façon programmatique :

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

où « obj » est un objet Java quelconque.

Utiliser les exemples *TestBrowser, TestBrowserObjTest, TestBrowserApplicationLauncher* du package *fr.ign.cogit.geoxygene.example* pour avoir un éventail des possibilités du navigateur graphique d'objets.

ANNEXE A : ELEMENTS DE COMPREHENSION DU SCHEMA

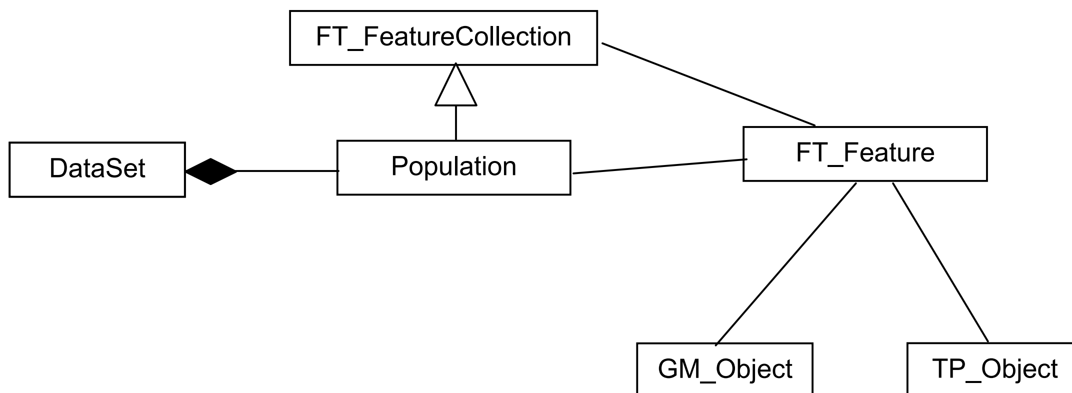
Cette section vise à fournir des éléments de compréhension du schéma mis en œuvre par GeOxygene. Il s'agit plus de notes d'intentions

A.1. LE SCHEMA GEOGRAPHIQUE (package « fr.ign.cogit.geoxygene.feature »)

- *FT_Feature* est la classe mère des classes géographiques. Des *FT_Feature* peuvent s'agréger en *FT_FeatureCollection,* classe qui représente donc un groupe de *FT_Feature* et qui porte des méthodes d'indexation spatiale. Voir l'exemple *fr.ign.cogit.geoxygene.example.TestIndex* pour comprendre le fonctionnement de l'index spatial.

- La classe *DataSet* représente un jeu de données : par exemple un extrait de base de données sur une zone géographique limitée, datent de l'année 2003, ou encore le thème hydrographie d'une base de données topographiques. Un « thème », sous-ensemble d'un jeu de données, est lui-même un *DataSet.* Un *DataSet* porte quelques métadonnées (zone, année, etc.).

- Un *DataSet* se compose de plusieurs *Populations.* La classe *Population* représente une *FT_FeatureCollection* particulière : il s'agit de TOUS les *FT_Feature* d'un *DataSet,* de même type.



Remarque importante : Si les classes *FT_Feature* et *FT_FeatureCollection* sont conformes au modèle de l'OGC, ce n'est pas le cas des classes *DataSet* et *Population*. **Ces classes sont un enrichissement des spécifications OGC.** Un utilisateur de ces 2 classes devra être conscient qu'il ne développe plus de solutions dont le modèle est interopérable.

A.2. LA GEOMETRIE (package « fr.ign.cogit.geoxygene.spatial »)

- *GM_Object* est la classe mère des objets géographiques. Un *GM_Object* peut-être soit une primitive (*GM_Primitive*), un agrégat (*GM_Agregate*) ou un complexe (*GM_Complex*). Agrégats et complexes sont des collections de primitives plus ou moins particulières (cf. plus bas).

- La géométrie est implémentée dans le package *fr.ign.cogit.geoxyegene.spatial* lui-même découpé en sous-packages.

  - Le package *fr.ign.cogit.geoxyegene.spatial.geomroot* ne contient que la classe mère GM_Object.
  - Le package *fr.ign.cogit.geoxyegene.spatial.geomaggr* contient les classes géométriques d'agrégats.
  - Le package *fr.ign.cogit.geoxyegene.spatial.geomcomp* contient les classes de complexes géométriques.

- Le package *fr.ign.cogit.geoxyegene.spatial.geomprim* contient les classes de primitives qui ne stockent pas directement de coordonnées (dont GM_Point, GM_Curve et GM_Surface).
- Le package *fr.ign.cogit.geoxyegene.spatial.coordgeom* contient les classes de primitives qui stockent des coordonnées (dont *DirectPosition, GM_LineString, GM_Polygon*).

- Un agrégat est une collection de *GM_Object* sans aucune structure interne. On peut trouver des agrégats hétérogènes (*GM_Aggregate*) c'est-à-dire composés de primitives de différents types, ou des agrégats homogènes (*GM_MultiPrimitive* et ses sous-classes) c'est-à-dire composés de primitives du même type (point, ligne, surface).

- Un complexe est une collection structurée de *GM_Primitive*. On impose en particulier aux primitives d'être connectées. La classe *GM_Complex* rassemble des primitives hétérogènes ; on ne l'utilise pas pour le moment. On utilise la sous-classe *GM_Composite* et ses sous-classes : c'est une collection de primitives, homogène à une primitive. Ceci signifie :

  - Un *GM_CompositePoint* est homogène à un *GM_Point*, donc est composé d'un et d'un seul point (ça ne sert pas à grand chose ! ☺)
  - Une *GM_CompositeCurve* est homogène à une courbe : elle est composée de primitives linéaires connectées telles que le point final de l'une soit le point initial de la suivante.
  - Un *GM_Ring* est une *GM_CompositeCurve* particulière : elle se referme (point initial de la première primitive = point final de la dernière primitive).
  - Une *GM_CompositeSurface* est homogène à une surface : elle est composée de primitives surfaciques adjacentes qui ne se recouvrent pas.
  - Un *GM_Shell* est une *GM_CompositeSurface* particulière : elle se referme.

- La primitive linéaire de base s'appelle *GM_Curve*. Une *GM_Curve* est composée d'un ou plusieurs *GM_CurveSegment*. Un *GM_CurveSegment* peut être une polyligne (*GM_LineString*), une suite d'arcs de cercles *(GM_ArcString*), une spline (*GM_SplineCurve*), etc… Un nombre conséquent de *GM_CurveSegment* est prévu par le modèle.

- Le seul *GM_CurveSegment* implémenté dans GeOxygene est la polyligne *GM_LineString*. En effet, les SGBD et SIG actuels n'offrent pas beaucoup d'autres alternatives pour le stockage des primitives linéaires. La *GM_LineString* est de surcroît une *GM_Curve* particulière, composée d'un et d'un seul segment qui est elle-même (extension de la norme ISO). Cette extension à la norme permet de travailler directement et facilement sur les instances de la classe *GM_LineString*, qui est le cas le plus courant. Malgré tout, la classe *GM_Curve* existe et est utilisable.

- La frontière d'un *GM_Polygon* est un *GM_Ring*.

- On retrouve, pour les surfaces, une modélisation analogue à celle sur les linéaires. La primitive surfacique de base s'appelle dans la norme *GM_Surface*. Elle est composée de *GM_SurfacePatch*. Un *GM_SurfacePatch* peut être un polygone (*GM_Polygon*) ou des choses plus compliquées pour permettre de travailler en 3D (cylindre, sphère, etc.). *GM_Polygon* est ici une *GM_Surface* particulière composée d'un et d'un seul *GM_SurfacePatch* qui est lui-même. Ceci permet là aussi de travailler directement sur *GM_Polygon*. En pratique, dans la version actuelle de GeOxygene, seule la classe *GM_Polygon* est utilisable (*GM_Surface* ne l'est pas).

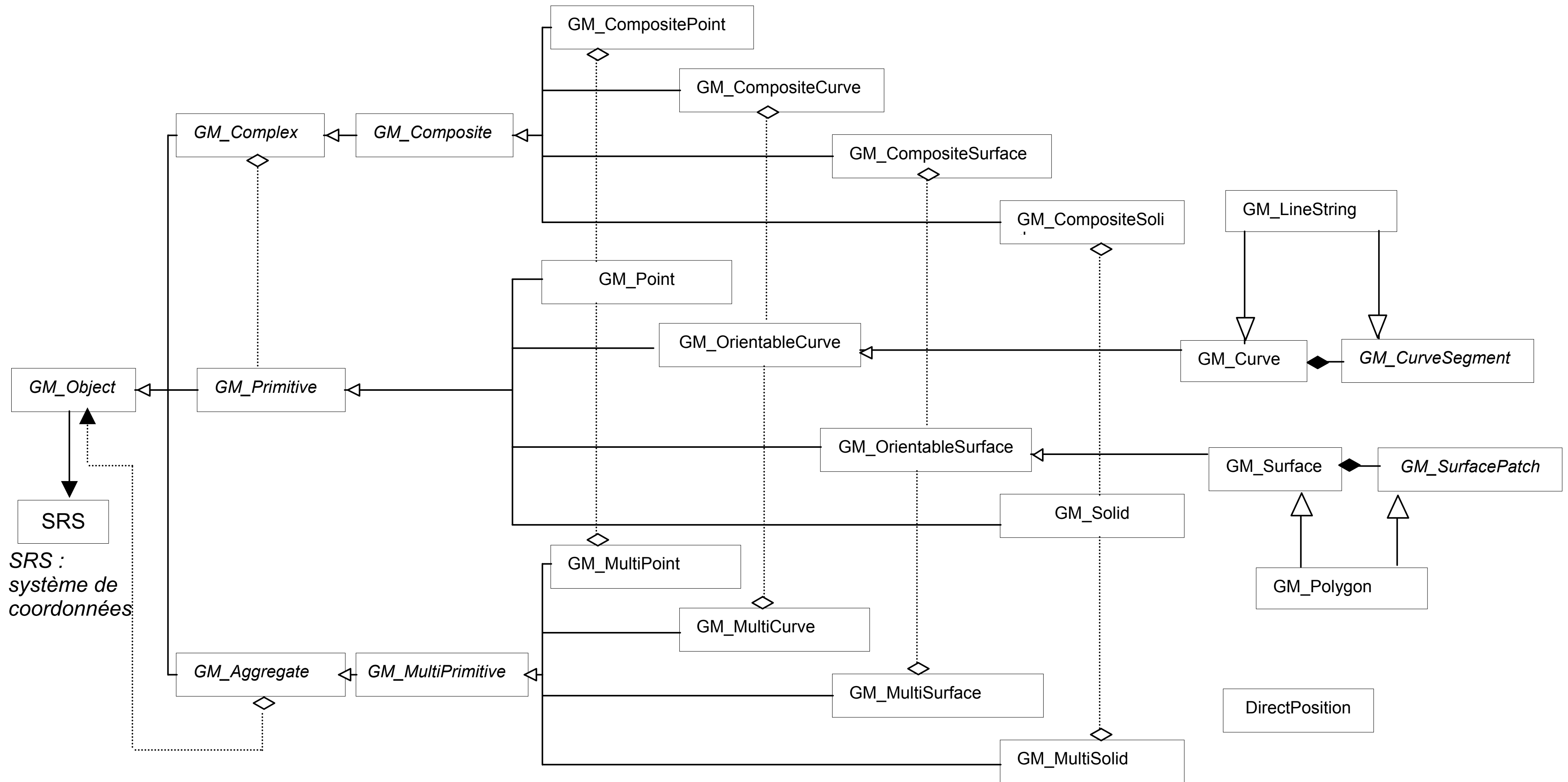
- Il existe des classes de primitives orientées (*GM_OrientableCurve, GM_OrientableSurface)*. On appelle « primitive » la primitive orientée positivement *(GM_Curve, GM_Polygon)*. Chaque primitive orientée est liée à sa primitive orientée positivement via le lien *primitive*. En fait, la primitive orientée positivement et la primitive sont le même objet. C'est elle qui est éventuellement liée à une primitive topologique. La primitive est liée à ses deux primitives orientées via le lien *proxy* (sachant que la primitive orientée positivement est elle-même …).

- La classe *DirectPosition* représente un tableau de X,Y,Z, avec certaines méthodes associées (non détaillées ici). Si on travaille en 2D, le Z n'est pas renseigné. On ne parle pas des primitives 3D dans ce document.

- A noter qu'il existe des classes qui sont des structures pour représenter les frontières des objets géométriques (*GM_CurveBoundary* pour représenter la frontière d'une GM_Curve,

GM_SurfaceBoundary pour représenter la frontière d'un GM_Polygon). Leur utilisation n'est pas fondamentale.

- Quelques compléments sur le modèle géométrique :
    - Se fier à la description technique du modèle et à la norme ISO 19107.
    - Regarder le schéma page suivante.
    - Utiliser l'exemple *TestGeomCurve* du package *fr.ign.cogit.geoxygene.example*.

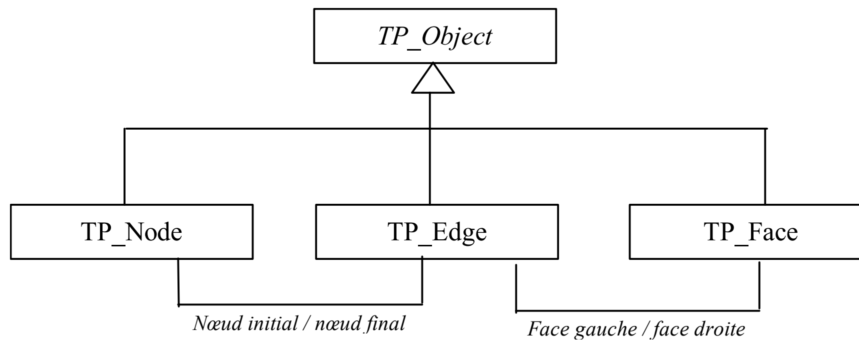


Open GIS (feature geometry) / ISO 19107 : diagramme des classes relatives à la géométrie
(en italique : classes abstraites)

A.3. LA TOPOLOGIE (package « fr.ign.cogit.geoxygene.spatial »)

- De même qu'il existe une classe mère abstraite pour la géométrie *GM_Object*, il existe une classe mère abstraite pour la topologie : *TP_Object.*

- La topologie de GeOxygene reprend la structure de carte topologique : il existe des brins *(TP_Edge)* définis par un nœud final et un nœud initial (*TP_Node)*, et des faces *(TP_Face)*, les brins ayant une face gauche et une face droite.



- On peut avoir une topologie de type "réseau" : on trouve uniquement des nœuds et des brins (sans faces).

- Il existe dans le modèle, beaucoup de classes et de méthodes pour représenter et manipuler la topologie : nœud orienté, brin orienté, face orientée ; on peut aussi manipuler les primitives topologiques sous forme d'expressions polynomiales. Ces notions ne sont pas indispensables à la compréhension du modèle et ne seront pas abordées ici.

- La topologie permet de stocker des relations spatiales, qui sont calculées lorsqu'on utilise la géométrie. Ainsi, on passe de calculs de géométrie algorithmique très longs, à des calculs plus simples.

- Il existe des classes de primitives orientées (*TP_DirectedNode*, *TP_DirectedEdge*, *TP_DirectedFace*). On appelle « primitive » la primitive orientée positivement (*TP_Node, TP_Edge, TP_Face)*. Chaque primitive orientée est liée à sa primitive orientée positivement via le lien *topo*. En fait, la primitive orientée positivement et la primitive sont le même objet. C'est elle qui est liée à une primitive géométrique. La primitive est liée à ses deux primitives orientées via le lien *proxy* (sachant que la primitive orientée positivement est elle-même …).

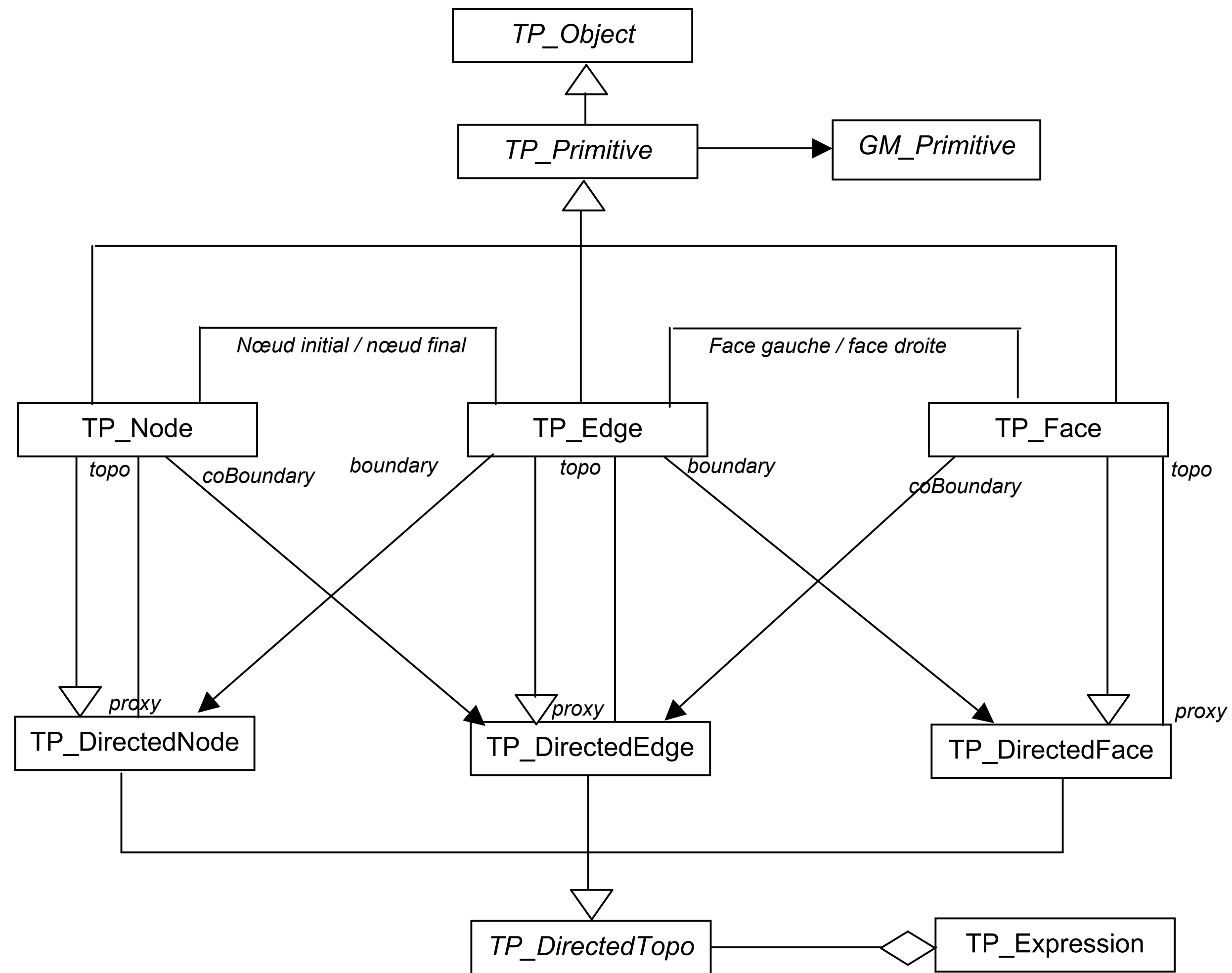
- Les liens nœud initial / nœud final et face gauche / face droite définissent une carte topologique. Ils ne sont pas présents strictement dans la norme, mais servent à stocker efficacement les relations.

- Sur les primitives, il existe des méthodes « boundary( ) » et « coBoundary( ) » qui renvoient des ensembles de TP_DirectedTopo. Il existe aussi de méthodes arc suivant, arc précédent, etc.

- Les TP_Expression permettent une manipulation algébrique des primitives orientées. C'est utile par exemple pour des calculs d'itinéraires. Se référer à l'exemple *TestTopo* du package *fr.ign.cogit.geoxygene.example.*

- Quelques compléments sur le modèle topologique :

    - Se fier à la description technique du modèle et à la norme ISO 19107.
    - Regarder le schéma page suivante.
    - Utiliser l'exemple *TestTopo* du package *fr.ign.cogit.geoxygene.example*.



Open GIS (feature geometry) / ISO 19107 : diagramme des classes relatives à la topologie
(en italique : classes abstraites)

ANNEXE B : CONVENTIONS DE CODAGE

Si vous contribuez au développement de GeOxygene, nous vous remercions d'essayer de suivre les quelques conventions de codage suivantes, que nous avons adoptées afin de rendre nos codes cohérents et plus faciles à lire et à maintenir.

STRUCTURE ET DOCUMENTATION

- Organisez votre code en package. Un fichier "package.html" dans le répertoire source du package doit brièvement résumer le contenu et la structure du package. Il apparaîtra dans la javadoc.
- Une classe par fichier. Présentation des classes :
    - Mention de licence, de provenance et de non garantie
    - Le nom du package
    - Une ligne blanche
    - Les import
    - Le commentaire javadoc /** … */. Ne pas oublier les tags @author et @version.
    - Le code
- Commentez chaque méthode et attribut public en javadoc /** ... */.
  On se dispense de la description des paramètres (tag @param) quand ils sont intuitifs.
- Commentez comme ceci /* ... */ ce qui est inutile à la documentation mais utile à la compréhension générale du code (fonctionnement d'un algorithme, notes, lien vers un document). Ne dites pas seulement ce que vous faites, mais pourquoi vous le faites !
- Utiliser // pour commenter les parties de code peu claires. Dans l'idéal, c'est à éviter ! Plutôt que d'écrire du code peu clair et de le commenter, écrivez du code clair ! ;-)

  Exemple :

  int index = -1; // -1 serves as flag meaning the index isn't valid

  Ceci est mieux :

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

- Une centaine de caractères maximum par ligne.
- Aérez votre code ! Sautez des lignes, mettez des espaces.

CONVENTIONS DE NOTATION

- *package* : tout en minuscule *commeca*
- *classe* : première lettre en majuscule impérativement *CommeCa*
- *attribut* : première lettre en minuscule impérativement *commeCa*
- *constante (*attribut *final*) : tout en majuscule, éventuellement avec des « underscores » *COMME_CA*
- *méthode :* première lettre en minuscule impérativement *commeCa()*
- *méthode créant un objet de type Truc : void createTruc()* ou *void newTruc( ).*
- *méthode convertissant un objet en type Truc* : toTruc( )
- *soit un attribut attr de type Type*
  méthode renvoyant l'attribut attr : *getAttr( )*
  méthode affectant l'attribut attr : *void setAttr(Type value)*
- *soit une collection (set ou list) machin contenant des objets de type Type*
  méthode renvoyant toute la collection : *Collection getMachinList( )*
  méthode renvoyant un élément indexé (si liste) : *Type getMachin(int i)*
  méthode affectant une valeur à l'indice i : *void setMachin(int i, Type value)*
  méthode ajoutant un élément à la collection machin à la fin: *void addMachin(Type value)*
  méthode ajoutant un élément à la collection à l'indice i: *void addMachin(int i, Type value)*

méthode enlevant un élément à la collection machin :	*void removeMachin(int i)* *removeMachin(Type value)*
méthode vidant la collection machin :	*void clearMachin( )*
méthode renvoyant la taille de la collection machin :	*int sizeMachin( )*
initialisation d'un itérateur sur la collection :	*void initIteratorMachin( )*
prochain élément :	*boolean hasNextMachin( )* *Type nextMachin()*

■ *QUELQUES RECOMMANDATIONS*

- Les imports : évitez les *import package.**, dites précisément ce que vous importez. Cela aidera les lecteurs de votre code (utilisez la fonction « organize imports » sous Eclipse par exemple).
- Préférez les double au float.
- Gérez les exceptions proprement, i.e. en utilisant les éléments du langage le permettant ou en créant vos propres exceptions !
- La classe contenant la méthode main doit juste servir à tester ou à faire une démonstration.
- Pour comparer des Object, utiliser la méthode *equals* et pas ==.