CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE REGIONAL RHÔNE-ALPES

CENTRE D'ENSEIGNEMENT DE GRENOBLE

MÉMOIRE

présenté par **Laurent Poulenard**

en vue d'obtenir

LE DIPLÔME D'INGÉNIEUR C.N.A.M.

en INFORMATIQUE

Diffusion de l'information géographique avec les services Web : application au logiciel HyperAtlas

Soutenu le 7 avril 2011

JURY

Président : M. Eric Gressier-Soudan

Membres : M. Jean-Pierre Giraudin
M. André Plisson
M. Mathias Voisin-Fradin
M. Jérôme Gensel
Mme Hélène Mathian
M. Bruno Orsier

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE REGIONAL RHÔNE-ALPES

CENTRE D'ENSEIGNEMENT DE GRENOBLE

MÉMOIRE

présenté par **Laurent Poulenard**

en vue d'obtenir

LE DIPLÔME D'INGÉNIEUR C.N.A.M.

en INFORMATIQUE

Diffusion de l'information géographique avec les services Web : application au logiciel HyperAtlas

Soutenu le 7 avril 2011

Les travaux relatifs à ce mémoire ont été effectués au sein de l'équipe STEAMER du LIG (Laboratoire d'Informatique de Grenoble), sous la direction de M. Jérôme Gensel.

Remerciements

En tout premier lieu, je remercie toutes les personnes qui me font l'honneur de participer au jury de ce mémoire :
Monsieur Eric Gressier-Soudan, professeur au CNAM Paris et président du jury ;
Monsieur Jean-Pierre Giraudin, professeur à l'Université Pierre Mendès France de Grenoble ;
Monsieur André Plisson, directeur du centre d'enseignement CNAM de Grenoble ;
Monsieur Mathias Voisin-Fradin, directeur adjoint du centre d'enseignement CNAM de Grenoble ;
Monsieur Jérôme Gensel, professeur à l'Université Pierre Mendès France de Grenoble ;
Madame Hélène Mathian, ingénieure de recherche CNRS au laboratoire Géographie-Cités ;
Monsieur Bruno Orsier, responsable R&D chez Agilent Technologies France.

Je tiens à exprimer ma reconnaissance à mes tuteurs, Jérôme Gensel, professeur à l'Université Pierre Mendès France de Grenoble, et Paule-Annick Davoine, maître de conférences à l'INP de Grenoble, pour leurs conseils, pour leur soutien, et surtout pour la confiance qu'ils mont accordée tout au long de mon stage.

Je remercie également tous les membres de l'équipe STEAMER pour leur accueil et leur disponibilité. Je remercie tout particulièrement Christine Plumejeaud, Anton Telechev et Benoît Le Rubrus. Nos échanges furent passionnés et passionnants.

Merci aussi à Arnaud Pierrel et Bruno Orsier, de l'entreprise Agilent Technologies France, qui se sont montrés compréhensifs et m'ont encouragé tout au long de ma formation au CNAM.

Je remercie mon épouse et mes deux filles qui ont su m'entourer de leur affection tout au long de mon stage.

Je dédie ce mémoire à Georges Stamon, Maurice et Raphaël Poulenard, qui sont à l'origine de ma passion pour l'informatique.

Table des matières

Table des figures

Liste des tableaux

Acronymes

Ant : Another Neat Tool. Ant est principalement utilisé pour automatiser la construction de projets en langage Java, mais il peut être utilisé pour tout autre type d'automatisation, dans n'importe quel langage.

API : Application Programming Interface (interface de programmation). Une API est un ensemble de fonctions, procédures ou classes mises à disposition par une bibliothèque logicielle, un système d'exploitation ou un service.

AXIOM : Axis Object Model. Modèle d'objet du moteur de services Web Axis2.

CSV : Comma-Separated Values. CSV est un format informatique ouvert représentant des données tabulaires sous forme de «valeurs séparées par des virgules».

DBC : Design By Contract (conception par contrat). La programmation par contrat est un paradigme de programmation dans lequel le déroulement des traitements est régi par des règles. Ces règles, appelées des assertions, forment un contrat qui précise les responsabilités entre le client et le fournisseur d'un morceau de code logiciel.

DTD : Document Type Definition (définition de type de documents). DTD est un document permettant de décrire un modèle de documents XML. Le modèle est décrit comme une grammaire de classes de documents.

GML : Geography Markup Language. GML est un langage dérivé du XML pour encoder, manipuler et échanger des données géographiques.

HTTP : Hypertext Transfer Protocol. HTTP est un protocole de communication client-serveur développé pour le Web.

IDE : Integrated Development Environment. Un IDE est un environnement de développement logiciel.

JAXB : Java Architecture for XML Binding. JAXB est l'API Java d'Oracle (anciennement Sun Microsystems) permettant de créer des classes Java à partir de schémas XSD et inversement.

JAX-WS : Java API for XML Web Services. JAX-WS utilise les annotations pour créer des services Web.

JSDK : Java Servlet Development Kit. Kit de développement pour les Servlets.

OGC : Open Geospatial Consortium. L'OGC est un consortium, regroupant les principaux acteurs du domaine de la géomatique, dont l'objectif est de standardiser les services Web géographiques.

MIME : Multipurpose Internet Mail Extensions. MIME est un standard internet qui étend le format des données à d'autres encodages que l'ASCII dans les protocoles de communication du W3C.

REST : Representational State Transfer. REST est un style d'architecture pour les services Web, les services sont sans état (ils ne conservent aucune information entre deux appels).

ROA : Resource Oriented Architecture. Architecture orientée vers les ressources.

RPC : Remote Procedure Call. Appel de procédure à distance.

SDI : Spatial Data Infrastructure. Infrastructures de données spatiales.

SIG : Système d'Information Géographique. Un SIG est un système d'information capable d'organiser et de présenter des données alphanumériques spatialement référencées, ainsi que de produire des plans et des cartes.

SOA : Service Oriented Architecture. Architecture orientée vers les services.

SOAP : Simple Object Access Protocol. SOAP est un protocole de RPC orienté objet bâti sur XML.

TDD : Test Driven Development (développement dirigé par les tests). L'approche TDD est une pratique qui consiste à écrire le test unitaire du code avant le code.

URL : Uniform Resource Locator, désigne une chaîne de caractères utilisée pour adresser les ressources du Web.

W3C : World Wide Web Consortium. w3C est un organisme de standardisation chargé de promouvoir la compatibilité des technologies du Web.

WAR : Web ARchive. WAR est une extension de fichiers d'une application Web Java compressée.

WMS : Web Map Service. WMS est un service Web, dont l'API est un standard de l'OGC, qui permet de visualiser des cartes.

WSDL : Web Services Description Language. WSDL décrit une interface publique d'accès à un service Web, notamment dans le cadre d'architectures de type SOA (Service Oriented Architecture). C'est une description fondée sur XML qui indique «comment communiquer pour utiliser le service».

XML : eXtended Markup Language. XML est un langage informatique de balisage générique. Il sert essentiellement à stocker/transférer des données de type texte Unicode structurées en champs arborescents. Ce langage est qualifié d'extensible car il permet à l'utilisateur de définir les balises des éléments.

XSD : XML Schema Definition. XSD est un langage de description de format de documents permettant de définir la structure et le type de contenu d'un document XML. Cette définition permet notamment de vérifier la validité de ce document.

Chapitre 1

Présentation

La vérité est comme le meilleur coup aux échecs : elle existe, mais il faut la chercher.

Arturo Perez-Reverte

1.1 Introduction

L'information géographique suscite un intérêt croissant, de la part des particuliers, des entreprises ou des autorités publiques. On appelle information géographique une information contenant une référence à un lieu. Il peut s'agir d'un point précis du territoire, d'une infrastructure linéaire telle qu'une route ou encore d'un périmètre donné.

L'information géographique est divisée en deux parties [Dumolard *et al.*, 1998] :
– partie *géométrique*. Il s'agit de la localisation, la forme et la dimension. C'est cette partie qui permet sa représentation géographique, et elle suppose de disposer d'un référentiel spatial ;
– partie *sémantique*. Il s'agit d'un ensemble de descripteurs, qui tentent de la caractériser par rapport à un ensemble d'informations auquel elle appartient, et d'en exprimer la signification.

Les données géographiques concernées peuvent être de trois sortes :
– les référentiels géographiques (cartes, photographies aériennes, *etc.*), qui servent surtout de fond de plan pour la présentation des autres données ;
– les objets géographiques (bâtiments, routes, zones urbanisées, forêts, *etc.*), que l'on peut visualiser par superposition aux référentiels ;
– les données proprement dites, généralement rattachées à l'un de ces objets géographiques : la largeur ou le trafic d'une route, le nombre de logements ou d'emplois dans une zone.

Les besoins de données géographiques ont bien sûr toujours existé, mais ils étaient difficiles à satisfaire, car ce n'est que de façon récente que les systèmes d'information ont pu traiter ces données efficacement et massivement. Cela a permis à quelques grands acteurs de diffuser des informations géographiques sur Internet : en particulier Google (Google Maps), Microsoft (Bing Maps) et en France, l'IGN avec le Géoportail. L'avènement du Web 2.0, puis la démocratisation des *smartphones* ont fait exploser la demande. Le défi à relever pour les géomaticiens n'est donc plus seulement le traitement de l'information géographique, mais sa *diffusion*.

Cette problématique est vaste et revêt de nombreux aspects, mais il est un point sur lequel tous les acteurs du domaine s'accordent, c'est la brique fonctionnelle élémentaire sur laquelle s'appuie la diffusion de cette information : le service Web. Les systèmes d'information géographique (SIG) traditionnels sont généralement des applications mono-postes lourdes. Ils doivent évoluer en publiant

sous la forme de services Web les données et les processus qu'ils proposent actuellement sous la forme d'applications mono-postes.

Concrètement il s'agit de publier, grâce aux technologies du Web 2.0, des cartes interactives et les données qui leur sont associées.

1.2 Contexte

HyperCarte

Le projet de recherche HyperCarte a pour but de mettre au point des outils interactifs de représentation et d'interrogation cartographique des phénomènes socio-économiques, environnementaux, épidémiologiques, *etc.* [@Hypercarte].

Le concept «HyperCarte» repose sur l'hypothèse centrale suivante : toute *spatialisation d'un phénomène social peut faire l'objet d'un nombre infini de représentations*.

Ces représentations vont dépendre :
– de la nature intrinsèque des phénomènes ;
– des hypothèses du concepteur de la carte ;
– des objectifs, des demandes, des pratiques ou des croyances des utilisateurs finaux de l'information cartographique.

Toute représentation cartographique d'un phénomène social relève de choix qui président à son établissement (cadrage, orientation, échelle, maillages, seuils, trames visuelles), et qui expriment un parti pris dans la restitution scientifique ou politique de la réalité sociale.

Le projet Hypercarte propose d'ouvrir le champ des possibilités de représentation cartographique de ces phénomènes, et «d'explorer le monde cartographiquement» [Tobler, 2000].

Le projet repose donc à la fois sur des compétences en géographie et sciences sociales, ainsi que sur des compétences informatiques.

Il regroupe quatre partenaires :
– l'équipe PARIS de l'U.M.R. Géographie-cités ;
– l'U.M.S. RIATE ;
– l'équipe Mescal du Laboratoire d'Informatique de Grenoble (LIG) ;
– l'équipe Steamer du LIG.

Les principaux objectifs de l'équipe PARIS (acronyme de « Pour l'Avancement des Recherches en Interaction Spatiale ») sont de décrire, représenter et de modéliser l'organisation de l'espace géographique. Dans le cadre d'HyperCarte, cette équipe apporte son savoir-faire dans les domaines de la géographie et de la statistique (sciences sociales).

La mission de l'UMS RIATE (Unité Mixte de Service Réseau Interdisciplinaire du Territoire Européen) est le recensement et la valorisation des compétences scientifiques françaises en matière d'aménagement du territoire européen. Elle est le point focal français pour l'Observatoire en Réseau de l'Aménagement du Territoire Européen (en anglais *European Spatial Planning Observation Network* : ESPON).

La problématique de l'équipe MESCAL, quant à elle, a trait aux calculs distribués. Dans le cadre du projet HyperCarte, l'équipe MESCAL s'occupe de la parallélisation sur grappes de PC afin de pouvoir calculer en temps réel des cartes.

Enfin, l'équipe STEAMER apporte à ce groupe de recherche ses connaissances dans les systèmes d'information et le multimédia. Son rôle dans HyperCarte est de concevoir et développer les outils cartographiques. C'est au sein de cette équipe, dirigée par Jérôme Gensel, que nous avons effectué notre stage.

1.2.1 HyperAtlas

Une des réalisations du groupe Hypercarte est le logiciel HyperAtlas, un outil d'analyse spatiale.

HyperAtlas a connu de nombreuses améliorations au fil des années et la version sur laquelle est basé le travail présenté est la version 1.2.9. Le premier prototype a été réalisé dans le cadre du projet ESPON par Philippe Martin [Martin, 2004]. Son travail a ensuite été repris et enrichi par Olivier Cuenot [Cuenot, 2005], Christophe Chabert [Chabert, 2007], Christine Plumejeaud [Plumejeaud, 2007] et Raphaël Thomas [Thomas, 2008]. Aujourd'hui, Benoit Le Rubrus travaille sur une version 2.0 d'HyperAtlas [Le Rubrus, 2011].

L'application HyperAtlas permet de générer et de visualiser à la volée un ensemble de huit cartes, rendant compte de la distribution de différents indicateurs (démographiques ou socio-économiques) dans les limites d'un maillage territorial.

Modèle des Unités Territoriales

L'élément de base sur lequel est fondé un jeu de données d'HyperAtlas est l'unité territoriale. Ce sont les unités territoriales qui contiennent la partie géométrique de l'information géographique, qui sera utilisée pour dessiner les cartes. Les unités territoriales forment une hiérarchie qui est structurée sous la forme d'un arbre :
- le nœud racine correspond à l'espace étudié (par exemple l'Europe) ;
- les nœuds du premier niveau de l'arborescence correspondent aux unités principales (par exemple la France) ;
- les nœuds intermédiaires correspondent à des maillages intermédiaires (par exemple la région Rhône-Alpes) ;
- les nœuds feuilles correspondent aux plus petites unités (par exemple le département de l'Isère).

À partir de ce modèle, HyperAtlas va permettre de réaliser un certain nombre de cartes permettant d'observer une statistique pour un territoire donné.

Les indicateurs et le ratio

À chaque jeu de données est associée une liste d'indicateurs. Ces indicateurs décrivent une donnée numérique comme par exemple le nombre d'habitants ou la superficie. Pour chaque indicateur et pour chaque unité territoriale une relation correspondant à la valeur de l'indicateur est créée. Pour réaliser un jeu de cartes complet il faut en sélectionner deux. Le premier fera office de numérateur et le second de dénominateur. On peut par exemple combiner le nombre d'habitants à la superficie, le jeu de cartes correspondra alors à une étude de la densité de population. Le ratio, qui est la *thématique* représentée, se définit comme suit :

$$ratio = \frac{numérateur}{dénominateur}$$

Les indicateurs correspondent à une variable quantitative en valeur absolue (les économistes parlent de stock), c'est-à-dire à un dénombrement (nombre d'habitants) ou à une mesure (superficie). Ils

traduisent un état de l'unité territoriale à un instant donné. Le ratio, correspond à une variable quantitative en valeur relative.

Le contexte

La première notion est celle de *contexte*. Le contexte est l'ensemble des unités territoriales d'une aire d'étude pour un maillage donné. Une aire d'étude est une sélection d'unités principales (éventuellement l'aire totale d'étude). Un maillage correspond à un niveau de l'arborescence. Toutes les unités territoriales en dehors de cette sélection seront soit cachées, soit utilisées comme fond pour la carte. Par exemple, si sur le jeu de données Europe, on choisit comme contexte les quinze pays membres plus anciens, avec comme maillage «pays», les nouveaux membres (Pologne, Bulgarie, etc.) seront affichés comme les autres pays du fond de la carte (pays hors Union, comme la Suisse). Les unités territoriales correspondant aux maillages plus fins (régions, départements) ne seront pas dessinées sur la carte. La figure 1.1 représente la sélection que représente le contexte dans l'arbre du jeu de données : l'intersection d'un sous-arbre avec un niveau d'arborescence.

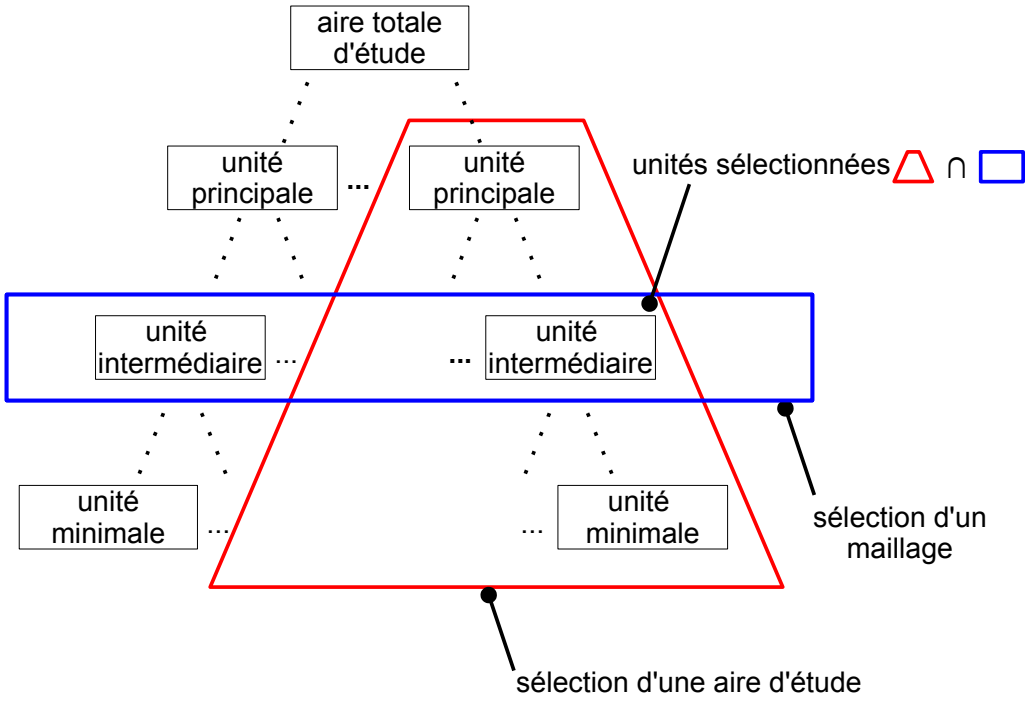


FIGURE 1.1 – Contexte HyperAtlas

Les déviations

La déviation d'un ratio par rapport à une valeur moyenne est calculée de deux façons, la déviation absolue et la déviation relative :

$$déviation_{absolue} = moyenne \times dénominateur - numérateur$$

$$déviation_{relative} = 100 \times \frac{ratio}{moyenne}$$

HyperAtlas calcule trois déviations pour chaque unité territoriale, par rapport aux trois moyennes suivantes :
– le ratio d'une des aires d'étude du jeu de données (par exemple on comparera la densité de la région Rhône-Alpes par rapport à celle de l'ensemble des pays de l'Europe Centrale), c'est la *déviation globale*. Pour toutes les unités du contexte cette moyenne a une valeur unique ;
– le ratio d'un maillage supérieur à celui du contexte (par exemple on comparera la densité de la région Rhône-Alpes par rapport à celle de la France), c'est la *déviation médiane*. Les unités ayant le même ancêtre dans l'arbre pour le niveau supérieur utiliseront la même moyenne, mais pas les autres ;
– la moyenne des ratios d'un groupe d'unités voisines (par exemple les unités situées à moins de six heures de camions), c'est la *déviation locale*. La déviation est calculée à partir d'une moyenne différente pour chaque unité.

La figure 1.2 permet de comprendre les relations entre la hiérarchie des unités territoriales et chacune des déviations. Pour une unité territoriale du contexte (grisée) donnée :
– la déviation globale correspond à la déviation par rapport à un groupe d'unités principales. Les unités principales sélectionnées ne font pas nécessairement partie du contexte ;
– la déviation médiane correspond à la déviation par rapport aux unités ancêtres ;
– la déviation locale correspond à la déviation par rapport à certaines des unités sœurs.

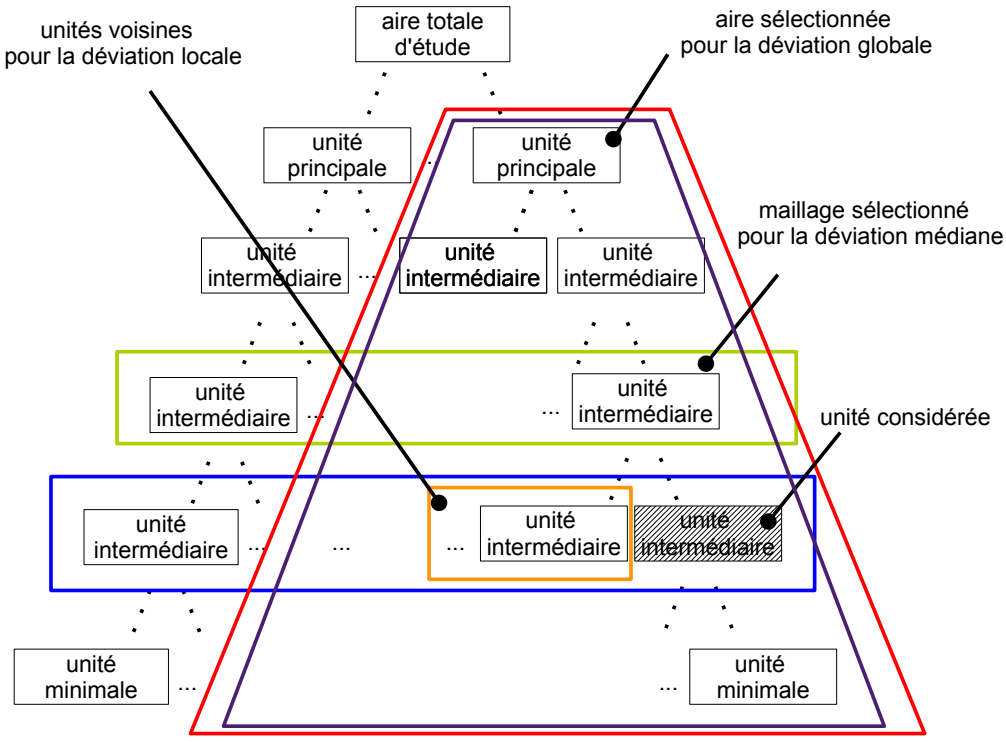


FIGURE 1.2 – Déviations HyperAtlas et hiérarchie

La distribution

La distribution vise à créer des classes d'équivalence à partir des unités territoriales du contexte. Lors du dessin de la carte, chaque classe sera associée à une couleur distincte, qui sera utilisée pour colorier toutes les unités appartenant à la classe. C'est ce qu'on appelle une carte choroplèthe [1].

1. Du grec *khorê*, qui signifie espace ou aire géographique et *plethos*, qui signifie quantité. La carte choroplèthe est le type le plus usuel de carte statistique.

Il y a deux types de distribution qui sont utilisés :
– distribution des ratios par quantiles : les quantiles sont des points essentiels pris à des intervalles réguliers d'une fonction de répartition d'une variable. La variable utilisée pour réaliser cette distribution peut-être le ratio, le numérateur ou le dénominateur. Les classes d'équivalence correspondant à ce mode de distribution ont toutes le même effectif ;
– distribution des déviations : l'ensemble des valeurs d'une déviation absolue (globale, médiane ou locale) forment un intervalle. Cet intervalle est découpé en sous-intervalles. La progression pour passer d'un intervalle à l'autre peut être arithmétique ou géométrique. Lorsque ce mode de distribution est choisi, une classe d'équivalence peut être vide.

Jeu de cartes

Un jeu de cartes est produit comme suit :
– sélection d'un espace d'étude ;
– sélection d'un maillage ;
– sélection d'un indicateur utilisé comme numérateur ;
– sélection d'un indicateur utilisé comme dénominateur ;
– sélection d'un espace d'étude utilisé pour le calcul de la déviation globale, la valeur moyenne utilisée correspondant au ratio des deux indicateurs pour l'espace d'étude sélectionné ;
– sélection d'un maillage utilisé pour le calcul de la déviation médiane, la valeur moyenne utilisée correspondant au ratio des deux indicateurs pour le maillage sélectionné. Le maillage sélectionné doit englober le maillage étudié, par exemple on considèrera la moyenne d'un pays pour chacune de ses régions ;
– sélection d'une relation de voisinage utilisée pour le calcul de la déviation locale, la valeur utilisée correspondant à la moyenne des ratios des unités territoriales satisfaisant la relation de voisinage.
– définition d'un seuil, utilisé pour classer les déviations relatives.

Le nombre de représentations possibles est proportionnel au nombre de possibilités offertes par le jeu de données. La première ligne du tableau 1.1 correspond au nombre de thématiques, les autres lignes au nombre de représentations pour une thématique donnée.

Choix possibles	**Représentations possibles**
I : nombre d'indicateurs	$I \times (I-1)$ thématiques
A : nombre d'aires d'études M : nombre de maillages	$A \times M$ contextes
A : nombre d'aires d'études	A possibilités pour la déviation globale
M : nombre de maillages	M possibilités pour la déviation médiane
V : nombre de relations de voisinage	V possibilités pour la déviation locale

TABLE 1.1 – Possibilités d'un jeu de données

L'atlas obtenu est composé des huit cartes suivantes :

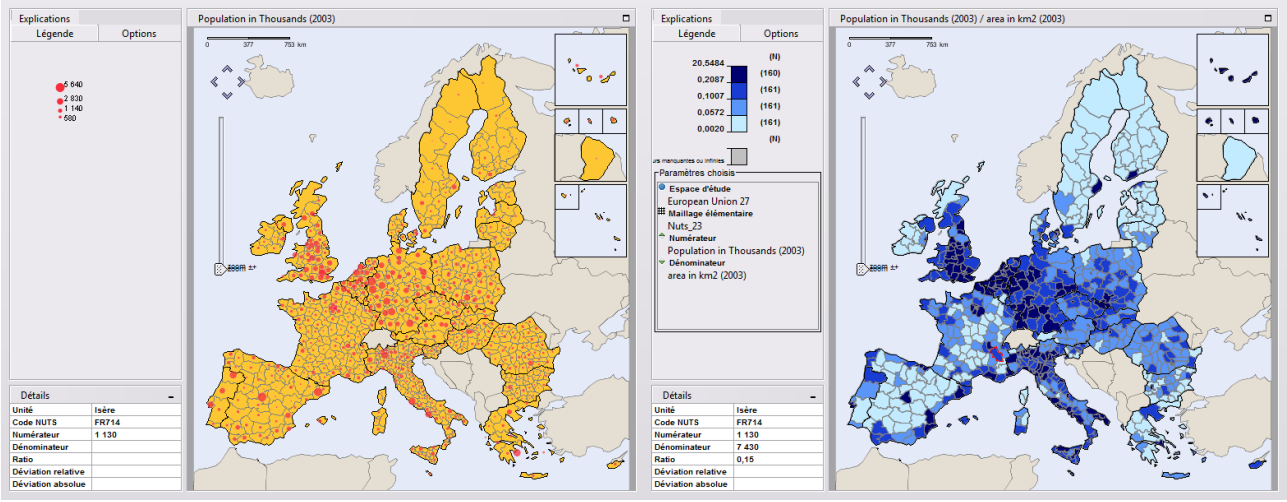
Espace d'étude : représentation du territoire sur lequel porte l'étude ;

Numérateur, dénominateur : deux cartes représentant les valeurs quantitatives absolues des indicateurs. Ces quantités sont représentées par des symboles (disques) dont la taille est proportionnelle à la valeur représentée. La figure 1.3-a correspond à ce type de cartes ;

Ratio : une carte choroplèthe représentant une distribution par quantile. Pour ce type de cartes on choisit un dégradé de couleurs allant du plus clair au plus foncé, la distribution traduisant une

variation unipolaire de la quantité représentée. La quantité représentée est relative si on choisit le ratio comme critère pour les quantiles, absolue si on choisit le numérateur ou le dénominateur. Ce type de cartes (voir figure 1.3-b) permet de visualiser la continuité de la distribution spatiale de la variable ;

Déviation globale, médiane et locale : trois cartes choroplèthes, représentant la distribution des déviations absolues (globale, médiane ou locale). Cette distribution est caractérisée par un nombre de classes (intervalles et donc de couleurs) et une progression entre ces classes, qui peut être arithmétique ou géométrique. Pour ce type de cartes on choisit en général un dégradé de couleur dans deux teintes (par exemple du bleu foncé vers le bleu clair, puis du rouge clair vers le rouge foncé) pour traduire graphiquement la variation bipolaire de la variable représentée : bleu/rouge traduit l'infériorité ou la supériorité par rapport à la valeur moyenne, foncé/clair l'éloignement par rapport à cette moyenne. La carte de la figure 1.3-c est un exemple de ce type de cartes ;



(a) carte à disque

(b) distribution des ratios (4 quartiles)

(c) distribution de la déviation (8 intervalles)

(d) carte synthèse

FIGURE 1.3 – Cartes d'HyperAtlas

Synthèse : une carte choroplèthe représentant une synthèse des trois déviations. Cette synthèse reprend les trois déviations relatives, et les compare à un seuil pour chaque territoire. Si le seuil vaut 100, et que la comparaison est basée sur la supériorité, la classification va s'établir à partir

des trois prédicats : la déviation globale relative est supérieure à 100 %, la déviation médiane relative est supérieure à 100 % et la déviation locale relative est supérieure à 100 %. Ceci offre huit possibilités, de Vrai-Vrai-Vrai à Faux-Faux-Faux. Ces huit possibilités sont associées à huit couleurs sur la carte de la figure 1.3-d. Le tableau suivant résume la classification établie pour la synthèse.

couleur de l'unité territoriale sur la carte							
prédicat pour la déviation globale	✔	✔	✔	✔			
prédicat pour la déviation médiane	✔	✔			✔	✔	
prédicat pour la déviation locale	✔		✔		✔		✔

TABLE 1.2 – Classification par couleur des déviations relatives par rapport à un seuil

Analyse multiscalaire

L'interprétation de ces cartes s'appuie sur la première loi de la géographie, énoncée par Waldo Tobler [Tobler, 1970] : «tout interagit avec tout, mais deux objets proches ont plus de chance de le faire que deux objets éloignés [2]». Elle s'appuie aussi sur les travaux de Claude Grasland de l'équipe UMS-RIATE [Grasland *et al.*, 2007]. L'objectif de ces cartes est donc de créer une proximité entre les unités territoriales, la proximité spatiale n'étant qu'une des dimensions de l'espace étudié, et de mettre en évidence, à l'aide des couleurs ou des tailles des symboles, les discontinuités de valeurs entre les unités.

La carte «Aire d'étude» ne présente aucun aspect sémiologique de l'information géographique contenue dans le jeu de données.

Les cartes «Numérateur», «Dénominateur» et «Ratio» reposent sur la proximité spatiale. Une unité territoriale se singularisera par rapport aux unités qui lui sont proches spatialement. Sur les cartes de la figure 1.3, dont la thématique est la densité de population, la région de Madrid se détache des unités qui lui sont proches (disque plus gros, couleur bleu marine sur la carte de ratio).

La carte «Déviation globale» illustre une proximité avec une aire d'étude. Par exemple si l'espace étudié est l'Europe des nouveaux entrants (PECO), choisir pour la déviation globale les membres plus anciens (Europe des quinze) crée une relation de proximité entre les unités territoriales de la PECO et la zone Europe des quinze.

La carte «Déviation médiane» illustre une proximité au sein d'un même maillage. Cette carte met en lumière les réseaux qui sont cachés dans une représentation spatiale. Cette proximité est à relier à l'unité sociale ou politique que représente le maillage. Cette carte est donc particulièrement précieuse lorsque le thème de l'étude est social. Par exemple, Cayenne, très éloignée de la France spatialement, fait partie de la même hiérarchie. Les frontières politiques, qui sont un facteur de distanciation peu visible sur une carte, si l'on excepte le fait de les dessiner de façon plus ou moins visible, sont accentuées par cette carte.

La carte «Déviation locale» redéfinit la proximité. La contiguïté est équivalente à la proximité spatiale (elle gomme tout à fait l'effet des frontières politiques), mais une relation de voisinage telle que «à moins de six heures de camion» fait la part belle aux infrastructures. Sur une telle carte, certaines unités territoriales peuvent se retrouver totalement isolées, comme Cayenne.

2. La phrase originale en anglais est : «*Everything is related to everything else, but near things are more related to each other*».

La carte «Synthèse» prend en compte toutes les proximités de l'espace étudié, qui peuvent être associées à différentes échelles de cet espace. On qualifie l'analyse produite par les cartes d'HyperAtlas d'analyse multiscalaire.

Par la suite l'analyse peut-être affinée en ajustant des paramètres propres à chaque carte, ce qui augmente d'autant le nombre de représentations du phénomène étudié :
– taille des disques ;
– palette de couleurs ;
– nombre de classes d'équivalence (égal au nombre de couleurs) ;
– type de quantile : par rapport au ratio, au numérateur ou au dénominateur ;
– mode de progression : arithmétique ou géométrique.

La dernière étape est la génération d'un rapport, à partir de l'analyse. Ce rapport est une page HTML qui contient les cartes générées ainsi qu'un tableau contenant les valeurs des indicateurs et les valeurs calculées.

Légendes

Les légendes d'HyperAtlas ne sont pas seulement des codes de couleurs décrivant les variables visuelles proposées par Jacques Bertin [Bertin, 1967]. Elles contiennent des informations qui n'apparaissent pas sur les cartes. Sans ces informations la carte est vidée de son sens. La taille des disques est la même sur la légende que sur la carte, et les valeurs indiquées au dessus des disques de la légende correspondent aux valeurs extrêmes de l'indicateur observé. Les valeurs indiquées au dessus du dégradé de couleurs correspondent aux bornes des intervalles. Sous le dégradé, l'effectif (cardinalité) de chaque classe est indiqué. La figure 1.4 montre deux exemples de légendes d'HyperAtlas.

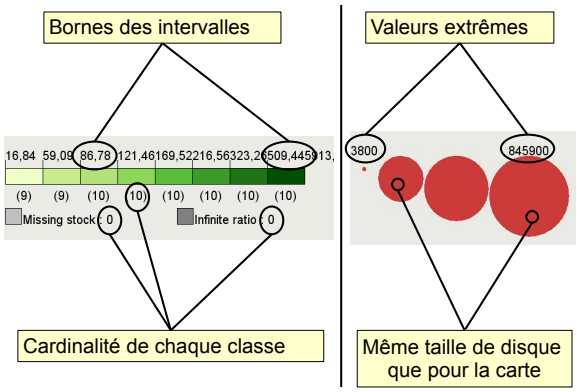


FIGURE 1.4 – Légendes HyperAtlas

La nécessité d'associer à une carte HyperAtlas sa légende sera un élément important de notre travail.

1.2.2 Diffusion des données géographiques

Les évolutions technologiques de l'information géographique numérique et des réseaux de communication bouleversent les relations entre les défis techniques et les acteurs susceptibles de les relever [Bucher, 2009]. Les vendeurs traditionnels de logiciels SIG ont raté le virage du Web, et laissé la place à de nouveaux acteurs, comme par exemple Google ou la communauté OpenStreetMap. Cette dynamique est d'ailleurs renforcée par l'essor des réseaux de positionnement.

La tendance est à la mise en place d'infrastructures de données géographiques (SDI, de l'anglais Spatial Data Infrastructures). Les géomaticiens désignent par ce terme les architectures de médiation. Elles facilitent l'accès à des sources de données géographiques variées.

Une application utilisant des données géographiques réutilise le plus souvent des données existantes. Le concepteur d'une telle application doit donc identifier ces données, les acquérir et éventuellement les intégrer. Le plus grand obstacle est d'ordre financier : le coût d'acquisition dépasse le prix que peuvent payer la plupart des utilisateurs, et il est donc nécessaire qu'il soit absorbé par un intermédiaire. Il peut s'agir d'un organisme d'état (ex : IGN), d'une entreprise privée dont le modèle économique s'accommode de cette «gratuité» (ex : Google) ou d'une communauté libre (ex : OpenStreetMap). Cette mutualisation bénéficie à tous : l'essor de l'information géographique citoyenne en atteste [Poulenard, 2010].

Une démarche phare dans le domaine de la géomatique est celle de l'Open Geospatial Consortium (OGC). L'OGC est une organisation internationale, à but non lucratif fondée en 1994 pour répondre aux problèmes d'interopérabilité des SIG. Elle regroupe aujourd'hui plus de 400 contributeurs, comprenant des entreprises (Google, IBM, EADS), des organismes d'état (NASA, IGN) ou des universités. Les missions du consortium consistent à regrouper tous les acteurs concernés afin de développer et promouvoir des standards ouverts garantissant l'interopérabilité dans le domaine de la géomatique et de l'information géographique et de favoriser la coopération entre développeurs, fournisseurs et utilisateurs [@OGC]. Ces standards comprennent notamment de nombreuses spécifications de services Web.

Par le biais de l'interopérabilité, la diversité des solutions correspond à la diversité des usages et des besoins, et non plus à la diversité des acteurs du domaine. C'est ce qu'illustre la figure 1.5 : le client qui utilise une SDI ne connait pas les fournisseurs de données de la SDI, il interroge la SDI par rapport à ses besoins.

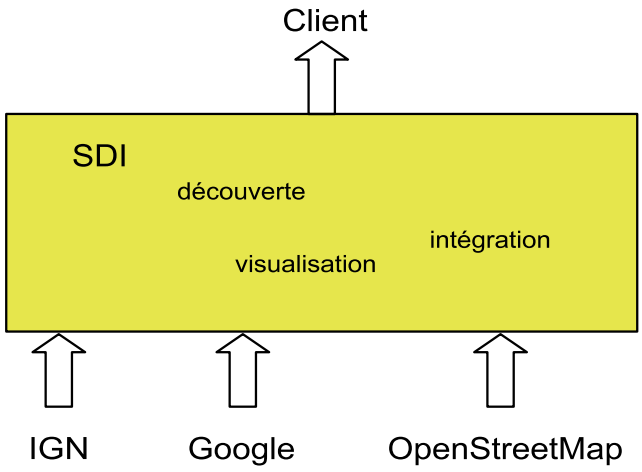


FIGURE 1.5 – SDI et médiation
Source : [Bucher, 2009]

1.3 Problématique

Aujourd'hui, le logiciel HyperAtlas est un logiciel mature. Le site du projet ESPON en propose le téléchargement [@ESPON, a], et le logiciel est utilisé dans plusieurs pays de la Communauté Européenne.

Toutefois, dans le contexte décrit dans la section précédente, le logiciel doit évoluer pour s'adapter aux contraintes liées à la diffusion de l'information géographique. Le projet de directive INSPIRE lancé par la DG Environnement et Eurostat a pour objectif de rendre disponible une information géographique pertinente et harmonisée au niveau européen. Elle prévoit en effet l'obligation pour les états membres de l'Union Européenne d'assurer l'harmonisation progressive de l'information géographique. ESPON incite de plus en plus ses partenaires à utiliser les standards de l'OGC [@ESPON, b] dans le cadre des projets qu'il finance.

Cette évolution vise à créer un certain niveau d'interopérabilité avec les services pré-existants, et donc constituerait un atout pour l'intégration d'HyperAtlas (cartes et calculs) dans des SDI, existantes ou à venir. Le laboratoire STEAMER collabore à d'autres projets avec ESPON, comme par exemple *ESPON 2013 DataBase Web Application*, qui sera la base de connaissances d'une SDI européenne.

HyperAtlas est une application mono-poste, et dans le contexte actuel de la géomatique, ce seul fait est un frein à sa diffusion. Par ailleurs, son utilisation impose d'installer, en plus du logiciel, les fichiers de données **.hyp* sur son poste. Enfin, le format propriétaire des fichiers empêche de combiner les données de ces fichiers à d'autres, permettant ainsi la création de cartes plus complètes, ni de partager les données que l'on produit avec des utilisateurs d'autres logiciels.

La publication sous forme de service Web des cartes que propose HyperAtlas permettrait de contourner ces obstacles. Si de plus ce service Web correspond à un standard de l'OGC, il pourra être combiné avec d'autres services de l'OGC, ce qui aura pour conséquence d'enrichir les possibilités d'HyperAtlas à moindre coût, à travers une application composite [3] par exemple. Le service Web permettrait aux utilisateurs d'HyperAtlas de créer de nouvelles cartes en incorporant dans leurs analyses spatiales des sources de données autres que celles des fichiers **.hyp*, et aux personnes qui n'utilisent pas le logiciel de profiter des méthodes d'analyse proposées, pour enrichir leur production cartographique.

Ce dernier point met en évidence la cohérence du projet par rapport au travail réalisé lors de l'épreuve TEST. Dans ce travail [Poulenard, 2010], nous avons abordé la problématique de la diffusion sous l'angle des consommateurs (les citoyens), peu experts de la géomatique, dans notre stage nous allons l'aborder du point de vue du fournisseur, expert du domaine.

1.4 Objectifs

En partant de la version 1.2.9 d'HyperAtlas, l'objectif principal du stage est la réalisation d'un service Web conforme aux spécifications de l'OGC, offrant les mêmes fonctionnalités que le logiciel.

Pour atteindre cet objectif, notre travail a été découpé en plusieurs étapes.

La première étape a consisté en une appropriation du projet afin d'en cerner le périmètre. Nous n'avions jamais utilisé avant ce stage le langage de programmation Java, ni programmé de services Web. De plus, le monde de la géomatique, même s'il avait été abordé dans le cadre de l'épreuve TEST [Poulenard, 2010], nous était inconnu. Enfin, nous avons découvert le logiciel HyperAtlas et le projet HyperCarte.

Cette appropriation s'est faite en trois temps, permettant chacun d'envisager des solutions :
– *appropriation WS :* réalisation d'un état de l'art de l'univers des services Web. Quelles solutions techniques avec quels outils pour un problème donné ?

3. En anglais *mashup*, une application composite permet de combiner des contenus ou des services plus ou moins hétérogènes. On peut par exemple combiner des images satellites avec la géolocalisation d'autres objets pour créer une nouvelle carte dont le fond est l'image satellite.

– *appropriation OGC :* réalisation d'une étude des différentes spécifications de services Web des standards OGC. Quelles spécifications pour quels besoins ?
– *appropriation HyperAtlas :* étude de l'existant. Quels sont les modules correspondant à la logique métier (celle que les services doivent restituer) ? Quels sont les modules correspondant à la logique de présentation ? Existe-t-il des couplages nécessitant de remanier le code [4] ?

La seconde étape a été le développement du service. Le standard de l'OGC qui s'est imposé est le Web Map Service (WMS), la fonctionnalité principale d'HyperAtlas étant de produire des cartes, et les spécifications du standard WMS étant particulièrement adaptées aux opérations mises en jeu par HyperAtlas. L'architecture du service est composée de trois couches :
– une couche de transport, responsable de l'acheminement des requêtes et des réponses ;
– une couche abstraite correspondant à l'utilisation des standards OGC. Cette couche est un environnement de développement qui permettra par la suite de réaliser d'autres services WMS ;
– une couche HyperAtlas, implémentation concrète de la couche précédente, et réutilisant dans une certaine mesure l'existant. Cette couche est l'objectif principal, le service HyperAtlasWMS.

La troisième étape a été la création d'un client pour le service. Le client a deux fonctions principales :
– utiliser un service WMS en général, et le service HyperAtlasWMS en particulier ;
– créer des documents cartographiques à partir des cartes générées.

1.5 Plan du mémoire

Après ce chapitre introductif, la suite de ce mémoire est composée de quatre chapitres.

Le chapitre 2 dresse un état de l'art des domaines scientifiques ou techniques impliqués dans la diffusion de l'information géographique sur le Web. L'accent est mis sur les services Web et sur les différents standards de l'OGC, qui constituent le socle technique de notre stage.

Le chapitre 3 présente notre proposition : les choix de conception et les choix techniques que nous avons faits pour pouvoir réaliser le projet que l'encadrement de l'équipe STEAMER nous avait confié. Ces choix ont été faits en nous basant sur l'état de l'art réalisé et sur l'analyse de l'existant, mais également d'après nos précédentes expériences dans le développement d'applications.

Le chapitre 4 est consacré au travail de développement que nous avons réalisé durant le stage, à partir des choix de conception et d'architecture proposés au chapitre 3. Ce chapitre présente également les outils et les méthodes utilisés.

Le dernier chapitre conclut ce mémoire de stage par une synthèse technique des réalisations présentées au chapitre 4, une ouverture vers les perspectives offertes par ces réalisations, et enfin un bilan personnel.

4. En anglais *refactoring* : il s'agit de retravailler le code source non pas pour ajouter une fonctionnalité supplémentaire au logiciel mais pour améliorer sa lisibilité, simplifier sa maintenance, ou changer sa généricité.

Chapitre 2

État de l'art

Je redoute l'homme qui n'a lu qu'un seul livre.

Thomas d'Aquin

Ce chapitre est divisé en trois parties.

La première partie aborde la problématique liée à la diffusion de l'information géographique sur le Web. Cette partie décrit le modèle des SDI, qui permet d'y répondre et qui a été adopté par la plupart des acteurs de ce domaine.

La seconde partie présente les standards de l'OGC, qui peuvent être considérés comme un cahier des charges pour la réalisation d'une SDI. Cette partie s'attarde sur le Web Map Service (WMS), la spécification que nous adopterons dans notre proposition. Cette partie se conclut par un banc d'essai de différents clients WMS.

La dernière partie présente les outils JAVA qui permettent de réaliser un service Web, la brique technique fondamentale des solutions implémentant les spécifications de l'OGC.

2.1 Diffusion de l'information géographique

L'information géographique est une représentation (graphique ou non) de phénomènes situés dans l'espace géographique. Sa diffusion vise à supporter un très grand nombre d'activités, ce qui nécessite de la tailler sur mesure pour son destinataire, en prenant en compte ses besoins. Cette adaptation aux besoins peut se résumer à deux cas de figure (volontairement très généraux) :

– l'information souhaitée n'existe pas et doit être construite sur mesure. Par exemple, on devra intégrer différents jeux de données, en modifiant éventuellement le schéma de ces données ou en en modifiant le niveau de détail ;
– l'information souhaitée existe, mais est difficile d'accès. L'effort portera sur l'interface de consultation. Par exemple, en associant des taxonomies compréhensibles pour l'utilisateur afin qu'il puisse rechercher les objets qui l'intéressent dans un catalogue de données géographiques.

Les SDI adressent cette double problématique tout en permettant de mutualiser les coûts de production de l'information géographique. Toutefois, ceci n'est possible qu'à travers un ensemble de conditions d'interopérabilité (normes, spécifications, protocoles, interfaces, *etc.*), et dans le cadre d'une coopération entre différents acteurs du domaine (administrations internationales, nationales ou locales, mais aussi entreprises du secteur privé).

2.1.1 Spatial Data Infrastructure (SDI)

Bénédicte Bucher distingue deux générations de SDI [Bucher, 2009] :
– la première génération de SDI (début des années 80) réalisait ces conditions par l'adoption par les différents acteurs des mêmes produits répondant aux spécifications adoptées. Le travail d'adaptation des producteurs de données participant à ce type de SDI consistait à adapter leurs données (à travers des processus plus ou moins automatisés) aux produits/spécifications de la SDI. On utilise parfois le terme de SDI *orientée produit*, car, de fait, la diffusion de l'information contenue dans une telle SDI va de pair avec la diffusion des produits/spécifications de la SDI ;
– les SDI de seconde génération essaient d'adopter un modèle plus *orienté vers les processus* décisionnels, qui vient compléter le modèle orienté produit.

Les défis à relever sont nombreux. Certains sont apparus avec la première génération de SDI et ont donc trait à la qualité des produits liés à ces SDI. Les autres sont apparus avec la seconde génération, et ont donc trait à la qualité des processus de prise de décision qui s'appuient sur ces SDI.

Défis

Standardisation : la standardisation des modèles et des méthodes est un élément incontournable, qui a reçu toute l'attention des premiers travaux dans le domaine. Techniquement, il s'agit de connecter les usages aux données. Les travaux du comité technique TC211 de l'ISO [1] et ceux de l'OGC convergent vers un modèle ISO/OGC qui est un atout lors de la mise en place de SDI. Ces recommandations sont davantage des normes de méta-modèles sur la nature des objets, et de description de leur positionnement que de description de l'espace géographique. S'y conformer permet *de facto* une certaine interopérabilité avec les données et méthodes d'autres SDI les ayant adoptées. Toutefois, ces standards ont des limites au niveau conceptuel. Par exemple, s'ils permettent de placer au bon endroit sur une même carte deux objets de type FeatureType provenant de deux SDI différentes, effectuer une analogie par rapport à la nature de ces deux objets ne sera pas possible, car en général les ontologies utilisées par deux SDI distinctes ne sont pas interopérables. On peut définir plusieurs sources d'hétérogénéité freinant l'interopérabilité [Balley, 2008] :
– hétérogénéité des formats (la plus visible) ;
– hétérogénéité des schémas ;
– hétérogénéité sémantique (un même mot ne définit pas toujours la même chose dans deux ontologies distinctes) ;
– hétérogénéité des structures (conjonction des trois précédentes).
Hélas, il n'existe pas de solution globale (une structure universelle convenant à toutes les applications) ;

Intégration des utilisateurs : l'intégration des utilisateurs dans les processus de développement n'est pas une problématique propre au développement des SDI. Toutefois, elle possède un aspect particulier : la grande diversité d'expertise des utilisateurs. On doit intégrer cette disparité des utilisateurs dans le modèle de la SDI [Kuhn, 2005]. C'est encore trop rarement le cas, il est par exemple symptomatique que dans le *cookbook* du GDSI [2] on ne trouve aucune mention des utilisateurs des SDI [GSDI, 2008]. Les SDI sont encore trop souvent conçues et évaluées par des experts pour des experts ;

Modèles plus ouverts : les standards de l'OGC reflètent une vision experte du domaine, et influencée par les modèles des logiciels cartographiques du monde des SIG. Par exemple, OpenStreetMap n'utilise pas les standards de l'OGC parce que la communauté les jugeait «trop complexes

1. OSI ou ISO : Organisation Internationale de Standardisation.
2. Global Data Spatial Infrastructure Association, une organisation regroupant des acteurs des SDI, dont l'OGC.

par rapport à ses besoins». Ce défi est corrélé au défi précédent : le niveau d'expertise des utilisateurs de la SDI ;

Catalogage : un défi pour les SDI de seconde génération est de proposer des méthodes de catalogage et d'intégration génériques et dynamiques ;

Contribution : le besoin de contribuer à des contenus géographiques est une tendance forte et également un enjeu stratégique pour les SDI [Poulenard, 2010]. Pour intégrer pleinement ces contributions, évaluer la qualité des contributions est impératif ;

Financement : le retour sur investissement n'est pas évident. Les SDI doivent s'appuyer sur les modèles économiques du Web 2.0 pour être viables. La mise en place d'indicateurs de performance pertinents (par exemple, le nombre d'utilisateurs ayant trouvé les données dont ils avaient besoin) est nécessaire pour convaincre les investisseurs ;

Diffusion : les fournisseurs de données sont souvent réticents à diffuser leurs données sans contrôle. Pour des raisons de sécurité nationale ou ne serait-ce que par crainte d'une prise de décision basée sur une mauvaise interprétation de ces données. L'amélioration des processus de décision qui est l'objectif des SDI de seconde génération contribue à relever ce défi ;

Attractivité : améliorer l'impact émotionnel des cartes [Bucher, 2009] augmente l'adhésion des utilisateurs, ainsi qu'en atteste le succès de GoogleEarth. D'une manière générale, le soin apporté à l'interface graphique du géoportail d'une SDI, qui en est la vitrine, est un élément décisif de son succès.

Modèle des SDI

Qu'elles soient orientées produit ou processus, les SDI ont des points communs. Elles doivent mettre en place un processus qui, partant des données collectées, va permettre de prendre des décisions. Ce processus (DIKW [3]), qui n'est pas propre au monde de la géomatique, est accompli en 3 phases :
– transformer les données en informations : qu'est-ce que c'est ?
– transformer les informations en connaissances : comment et pourquoi ?
– transformer les connaissances en sagesse : quelle est la meilleure solution ?

La dernière phase du processus est aujourd'hui dévolue aux utilisateurs des SDI, qui accédant aux connaissances de la SDI, pourront les transformer en sagesse.

Les SDI doivent donc permettre à des utilisateurs d'horizons variés d'obtenir et de retrouver des informations à la fois complètes et cohérentes [Davies, 2003]. Cette diversité des utilisateurs, combinée à la dissémination des données qui doivent être utilisées, nécessite de mettre en place une stratégie d'accès.

La clé du succès est l'interopérabilité, ce qui implique de la part des acteurs (fournisseurs et clients) l'adoption de standards convenant à tous, ou tout du moins au plus grand nombre. Le choix des standards est une étape critique dans l'implémentation d'une SDI, car il sera coûteux, voire impossible, d'y revenir plus tard : ce changement n'ajoute pas de contenus et il doit être accepté par tous les partenaires. Les standards doivent adresser toutes les dimensions techniques de la SDI. Les données et les modèles qui leur sont associés, mais aussi les métadonnées, qui permettent le catalogage, la recherche d'information, le transfert...

Pour atteindre cet objectif, une SDI regroupe différents partenaires. Bien souvent, ces partenaires appartiennent à des juridictions différentes (différents pays, secteur public/secteur privé, *etc.*), ce qui nécessite de mettre en place une gouvernance au sein de la SDI : confidentialité, propriété, sécurité...

3. DIKW est l'acronyme de Data Information Knowledge Wisdom, ce qui se traduit par Données Informations Connaissance Sagesse.

Toutes ces contraintes imposent d'établir des arrangements administratifs, des contrats commerciaux, et ce à la fois pour la collecte, la maintenance et la diffusion de l'information.

Ces trois nécessités, l'accès, la gouvernance, et les standards sont trois composants élémentaires du modèle SDI, les données (entrées du processus DIKW) et les utilisateurs (sorties du processus DIKW) étant les deux autres. Le modèle proposé par Abbas Rajabifard sur la figure 2.1 montre bien que l'utilisateur n'accède pas à la donnée, mais à l'information ou à la connaissance selon le degré de maturité de la SDI.

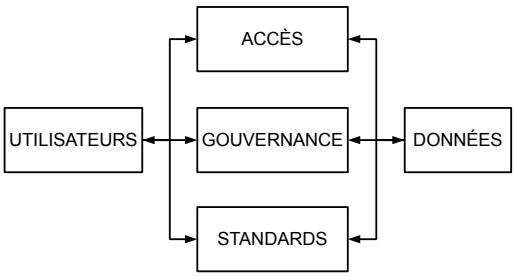


FIGURE 2.1 – Composants d'une SDI
Source : [Rajabifard et Williamson, 2001]

Ce modèle est une vision large des SDI, et les détails de l'implémentation de chacun des composants vont varier d'une SDI à l'autre.

Abbas Rajabifard établit une topologie des SDI [Rajabifard et Williamson, 2001], partant d'un niveau local (LSDI) en direction d'une SDI globale (GSDI), en passant par des SDI nationales (NSDI). Au sein de cette hiérarchie, la taille des régions couvertes par une SDI est inversement proportionnelle au niveau de détail des données qu'elle expose, c'est ce qu'illustre la figure 2.2.

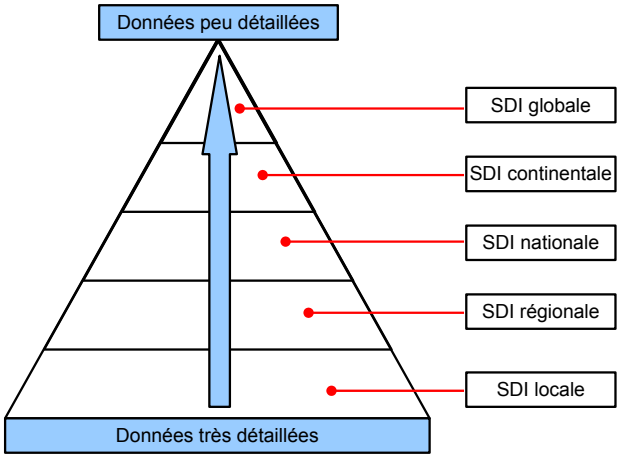


FIGURE 2.2 – Hiérarchie des SDI
Source : [Rajabifard et Williamson, 2001]

Cette topologie ne vise pas uniquement à distinguer les SDI par rapport aux territoires, mais aussi selon les objectifs qu'elle servent, et donc les organisations qu'impliquent ces SDI.

Une GSDI ou une SDI continentale correspondra à une entité internationale, dont les objectifs sont avant tout stratégiques. Il s'agit, par exemple, de prendre des décisions par rapport à une thématique environnementale. INSPIRE [4] est un exemple de ce type de SDI.

Une NSDI aura plutôt des objectifs fonctionnels, comme par exemple l'aménagement du territoire. Le géoportail de l'IGN [5] donne accès à une SDI de type national.

Les directions régionales ont chacune leur géoportail [6]. Les objectifs d'une telle SDI sont avant tout opérationnels, comme la visualisation du cadastre d'une commune.

La figure 2.3 met en parallèle la pyramide des organisations et celle de la hiérarchie des SDI, et les associe à un type de technologie.

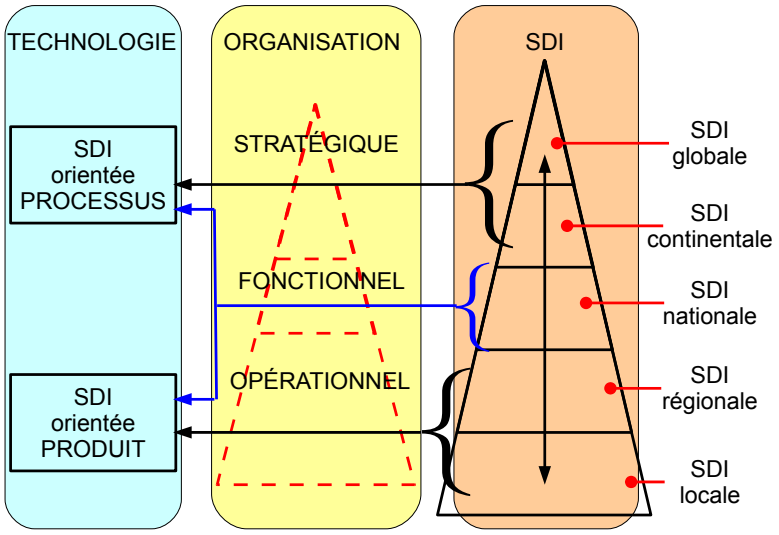


FIGURE 2.3 – SDI et organisations
Source : [Rajabifard et Williamson, 2001]

Les choix technologiques sont eux aussi liés à la hiérarchie des SDI. Ceci est expliqué par le fait suivant : selon le type de SDI, le poids des composants varie.

Les SDI interagissent les unes avec les autres. Un utilisateur commencera une recherche au niveau global, puis, au fur et à mesure, descendra dans la hiérarchie, obtenant des informations de plus en plus riches sur une localisation de plus en plus restreinte. Les données d'une GSDI sont donc avant tout des méta-données, qui décrivent les données pour chacune des SDI de niveau inférieur qu'elle abrite. Plus on descend dans la hiérarchie, plus la part du catalogue dans les données est réduite. Le composant «accès» sera donc plus lourd dans une GSDI que dans une LSDI.

Le composant «gouvernance» pèsera beaucoup plus dans une GSDI que dans une LSDI : le nombre des acteurs, la disparité des administrations impliquent énormément de travail sur ce composant. Au contraire, il sera très léger pour une LSDI, l'entité administrative étant souvent unique.

4. Le géoportail d'INSPIRE est accessible à l'adresse suivante : http://inspire.jrc.ec.europa.eu/index.cfm.

5. Le géoportail de l'IGN est accessible à l'adresse suivante : http://www.geoportail.fr/. Son nom, Géoportail, peut prêter à confusion avec le terme générique géoportail que nous utilisons comme traduction du mot anglais *geoportal*.

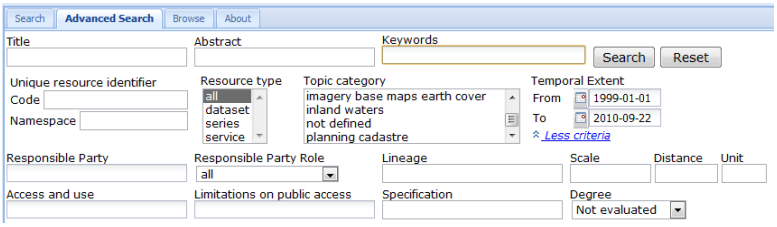
6. Le géoportail de la région Rhones-Alpes est accessible à l'adresse suivante : http://carmen.application.developpement-durable.gouv.fr/30/DREAL_DIFFUSION_GEN93.map.

Le composant «standard» sera également plus lourd dans une GSDI que dans une LSDI, puisqu'une GSDI doit être capable de manipuler les standards de toutes les SDI de niveau inférieur.

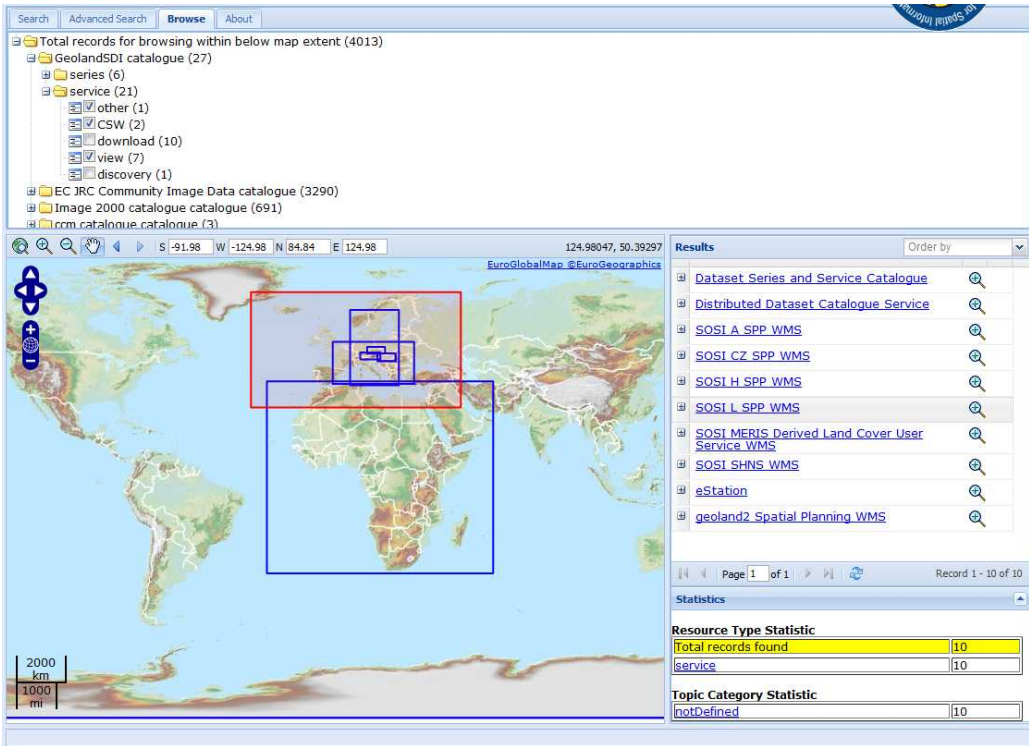
Le géoportail est un portail Web permettant de trouver et d'accéder à de l'information géographique. C'est un élément clé des SDI : c'est lui qui sert d'interface entre les utilisateurs et la SDI.

Les géoportails sont très différents graphiquement les uns des autres, mais il est une constante, c'est la possibilité d'effectuer des recherches parmi les informations de la SDI. Les interfaces graphiques varient là aussi, mais on peut les classer en trois catégories :
– champs textuels ;
– sélection (boîte, listes, arbres, *etc.*) ;
– carte interactive.

Les deux captures d'écrans faîtes à partir du géoportail d'INSPIRE sur la figure 2.4 montrent un exemple d'interface de recherche.



(a) Champs textuel et sélection dans une liste



(b) Sélection dans une arborescence et carte interactive

FIGURE 2.4 – Interfaces de recherche d'un géoportail

Cette interface de recherche correspond à l'interface du composant «accès». Si les fonctions filtrantes de l'édition de texte ou de la sélection sont évidentes, la carte interactive mérite quelques explications qui seront données dans la section suivante. Dans l'exemple de la figure 2.4, elle a deux rôles : limiter

la zone de recherche (en effectuant un zoom sur une zone précise) et afficher les zones correspondant aux résultats.

Le composant «standards» apparaît aussi partiellement dans cette interface de recherche : dans l'arborescence de la figure 2.4-b, les options de recherche correspondent aux types des informations (base de données, services Web). Dans l'onglet des résultats, les résultats sont classés en fonction du standard auquel ils correspondent.

Pour finir, le composant «gouvernance» est implicite du fait de la variété des sujets et des territoires que le géoportail permet d'explorer, et explicite à travers tout l'espace sur le géoportail dédié à la communication des partenaires de la SDI (cet aspect a été coupé des captures d'écran).

2.1.2 Carte interactive

L'interface utilisateur est un élément essentiel du succès d'une application à base de géographie. Le succès de Google Earth doit sans doute plus à son interface qu'à ses fonctionnalités. L'élément fondamental d'une interface est la carte interactive. La conception d'une carte est un domaine à part entière, qui tient autant de la science que de l'art : *l'essence de la modernité de l'appréhension scientifique et technique du monde s'y trouve mêlée avec la vie animale* [Houellebecq, 2010]. Néanmoins, il existe des règles (sémiologie), pas toujours respectées sur le Web, et des usages.

Les interactions offertes à l'utilisateur peuvent varier grandement selon les cartes, mais certaines d'entre elles sont si fondamentales qu'elles sont presque toujours présentes. Elles sont liées à l'exploration de la carte :
– le zoom et la translation, pour explorer la carte ;
– la sélection d'un objet de la carte et l'affichage d'informations supplémentaires le concernant ;
– la possibilité de filtrer les objets représentés sur la carte.

Fonctions d'une carte interactive La carte a plusieurs fonctions, qui dépendent du contexte dans lequel elle est utilisée. Parmi les contextes principaux, on peut citer :
– recherche de ressources sur le Web ;
– plan de situation, itinéraires ;
– information du citoyen, politiques environnementales, gestion du risque.

Dans le cadre de la recherche de ressources sur le Web, la carte servira à explorer et à évaluer la pertinence des résultats de la recherche. Selon l'objet de la recherche, les critères spatiaux peuvent être décisifs. Ce qui caractérise ces cartes, c'est la diversité et le nombre des ressources représentés sur la carte (photographies, établissements, personnes). Toute information peut-être indexée sur une carte à partir du moment où on peut lui associer une empreinte spatiale [Bucher, 2007].

Dans le cadre d'un plan de situation ou de la conception d'un itinéraire, l'interactivité, en plus de la phase d'exploration et de validation par l'utilisateur du résultat, devra permettre de modifier le résultat (différentes variantes d'itinéraires). Combinées à un dispositif de géolocalisation, ces cartes doivent interagir non seulement avec l'utilisateur, mais aussi par rapport au dispositif.

L'objectif de ce type de cartes est d'ancrer un phénomène dans la réalité, et éventuellement de susciter une prise de conscience. La sémiologie joue un rôle clef dans l'élaboration d'une telle carte. L'interactivité devra bien sûr permettre l'exploration de la carte, mais aussi permettre de filtrer les éléments affichés sur cette carte par rapport à la légende et aux informations qu'elle comprend, afin de permettre à l'utilisateur de mieux comprendre les phénomènes représentés.

Caractéristiques concrètes d'une carte interactive Une carte se distingue d'une autre par des différences dans des variables concrètes comme, par exemple, la couleur des signes ou l'épaisseur des traits pour une carte statique. Une carte écran héritera de ces variables et y ajoutera des variables relatives à l'affichage, comme un clignotement ou la gestion des zones interactives. La carte Web héritera des variables de la carte écran (et donc de la carte statique) et ajoutera des variables liées au Web : vitesse d'affichage, flux... La mise en cache d'une carte à plus petite échelle sur le serveur est une technique fréquemment utilisée pour pouvoir satisfaire plus rapidement les requêtes suivantes de l'utilisateur. La figure 2.5 illustre ces héritages successifs.

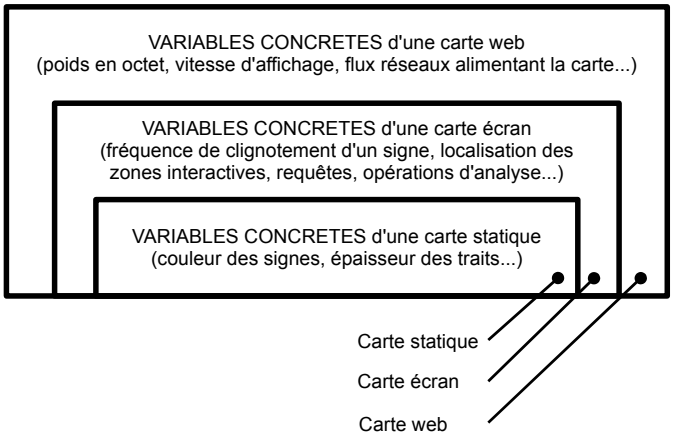
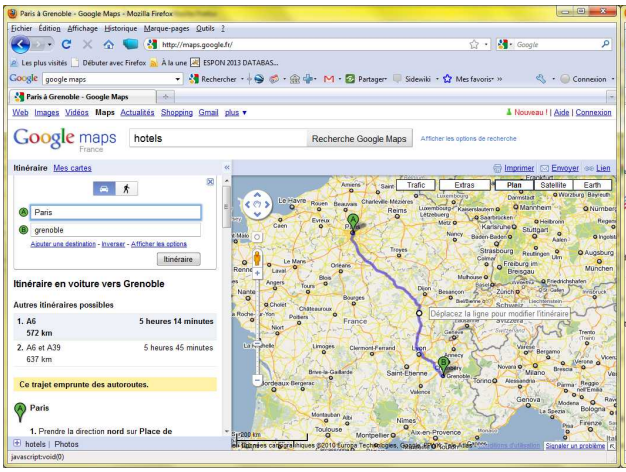


FIGURE 2.5 – Variables concrètes d'une carte Web
Source : [Bucher, 2007]

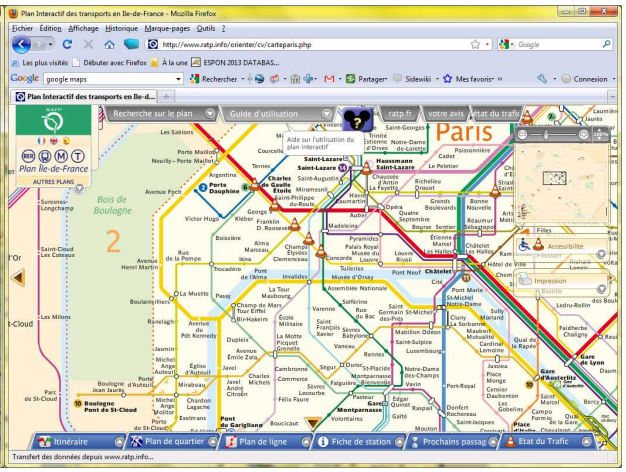
Caractéristiques abstraites d'une carte interactive Une carte a également des caractéristiques plus abstraites. Elles servent de niveau intermédiaire entre les fonctions de la carte et les variables concrètes qui la servent, comme par exemple les niveaux de lecture de la carte (une information apparaîtra ou pas selon l'échelle choisie). Elles doivent être définies en prenant en considération les variables concrètes et les fonctions attendues de la carte.

Parmi ces caractéristiques abstraites il y a celles héritées de la carte statique. Selon le type de carte (inventaire, communication...), les typologies varient et un même objet ne sera pas représenté de la même façon. Les cartes 3D posent à cet égard un problème : leur impact est lié à leur réalisme, qui cohabite mal avec la représentation d'un objet non visible (toponymie, variable statistique). D'autres caractéristiques sont héritées de la carte écran. Parmi celles-ci, les variables liées aux fonctions d'analyse automatique associées à la carte, ou alors celles liées à l'intégration de vues symboliques avec un fond de carte issu d'une vue satellite. Les caractéristiques propres à la carte Web sont variées. Il y a tout d'abord celles liées à l'affichage : taille d'écran variable, logiciel de navigation. D'autres sont liées à la navigation, qui ne doit pas décevoir l'utilisateur. La facilité de l'exploration (ergonomie), la fluidité sont des enjeux critiques pour le succès d'une carte. La nature des flux d'information qui peuvent être affichés sur la carte est un autre exemple de variables abstraites. Les flux RSS permettent par exemple d'automatiser la mise à jour d'une carte [Poulenard, 2010], les prévisions météorologiques sont un exemple de ce type de carte. Il existe de nombreuses autres variables liées à la possibilité d'éditer une carte (historique des modifications), de combiner cette carte avec d'autres (*mashup*) ou de la protéger d'utilisateurs malveillants (droits).

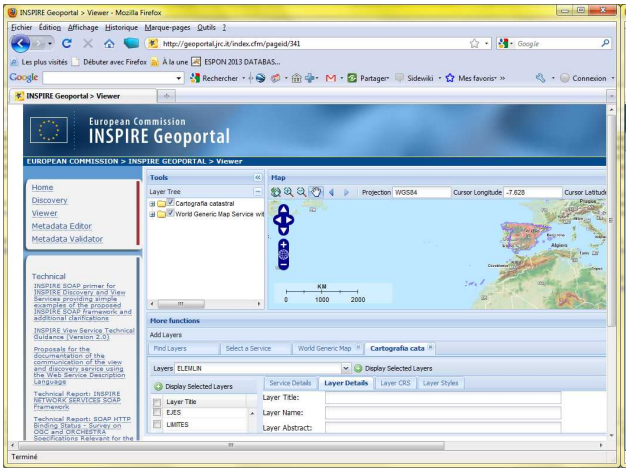
La figure 2.6 montre quelques exemples d'interfaces utilisateurs. La carte de google maps, même si on peut la critiquer d'un point de vue géographique, fait office de référence sur le web. La carte du métro est très fonctionnelle, mais sa prise en main n'est pas toujours intuitive. Les deux autres interfaces sont celles de sites destinés à des utilisateurs plus experts. et si l'on peut déplorer la taille et la lenteur de l'affichage de la carte sur le portail INSPIRE, l'interface d'Arc Explorer donne l'impression d'avoir été bâclée : elle est peu intuitive, pas très belle, et les contrôles de navigation disparaissent parfois complètement sous la carte, qui est très lente à s'afficher. C'est d'autant plus regrettable que l'application est riche en fonctionnalités.



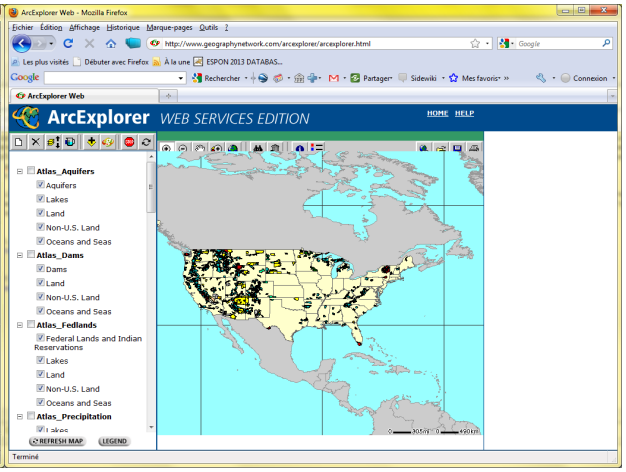
(a) itinéraire google maps



(b) Métro parisien.



(c) géo portail INSPIRE



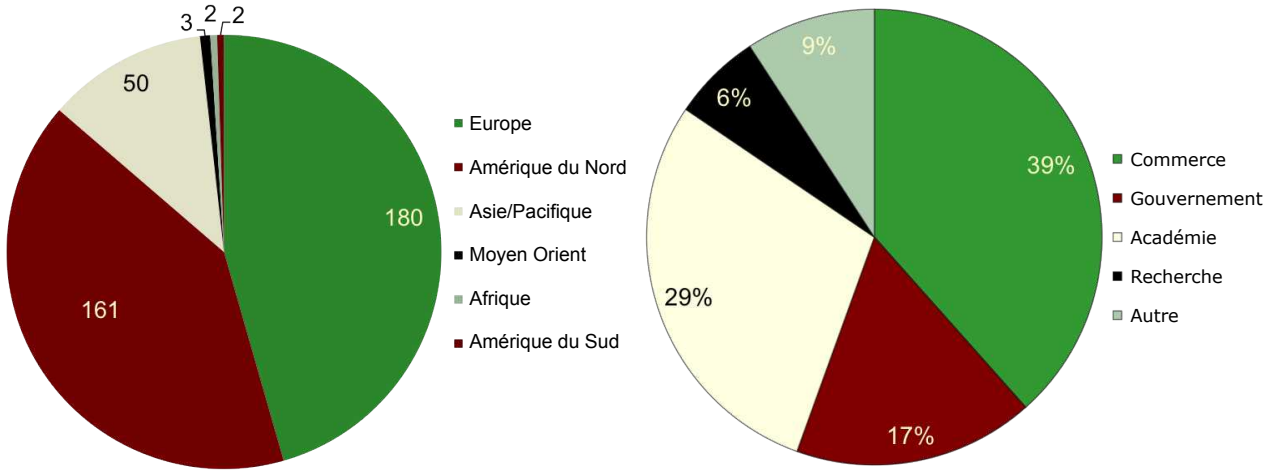
(d) Arc Explorer web

FIGURE 2.6 – Interfaces utilisateurs

2.2 Web services géographiques de l'OGC

2.2.1 L'organisation

L'Open Geospatial Consortium (OGC) est une organisation internationale à but non lucratif fondée en 1994. Elle regroupe aujourd'hui plus de 400 contributeurs d'horizons très variés. La figure 2.7 montre que l'Europe et l'Amérique du Nord constituent l'essentiel des partenaires, et que le secteur d'activité dominant est le secteur commercial. Toutefois, les membres n'ont pas tous le même poids dans l'organisation, et le comité stratégique n'est composé que de partenaires américains liés à des enjeux militaires (Lockheed Martin, Northrop Grumman Corporation, USGS [7], NASA [8] et NGA [9]) [@OGC].



(a) Répartition géographique

(b) Répartition par secteur d'activité

FIGURE 2.7 – Répartition des membres de l'OGC
Source : [@OGC]

L'OGC est un forum collaboratif cherchant à développer des standards internationaux pour favoriser l'interopérabilité des traitements de l'information géospatiale. Il existe aujourd'hui une trentaine de standards correspondant à des cas d'utilisation variés. Ces standards ne sont pas des solutions implémentées, mais des spécifications abstraites définissant l'API de services ou de structures de données. Ces standards sont ensuite implémentés par des solutions logicielles payantes ou non. En 2009, un accord a été conclu avec l'Open Source Geospatial Foundation [10] (OSGeo) pour implémenter des solutions concrètes de ces spécifications.

2.2.2 Standards de l'OGC

Les standards de l'OGC peuvent être répartis en cinq catégories. Quatre catégories correspondent à des services Web permettant :
– le catalogage (*Catalog Services*) ;
– le transfert d'information (*Data Services*) ;

7. US Geological Survey.
8. National Aeronautics and Space Administration.
9. National Geospatial-Intelligence Agency.
10. OSGeo (Open Source Geo) a été créée pour soutenir et construire une offre de logiciels en accès libre pour la géomatique de la plus grande qualité. http://www.osgeo.org.

– la visualisation (*Portrayal Services*) ;
– le traitement (*Processing Services*).

La cinquième catégorie correspond à des formats de données ou de description (*encodings*).

La figure 2.8 montre comment ces services et les formats peuvent servir dans la mise en place d'une SDI. Les services de catalogage opèrent l'indexation (flèches *Publish*) au sein de la SDI, pour permettre à l'utilisateur la recherche d'information à l'aide d'une application cliente, (flèches *Find*). Les services Web effectuant le transfert d'informations, la visualisation et le traitement (flèches *Bind*) permettront à l'utilisateur (toujours à travers une application cliente) d'exploiter l'information. Les formats et les différentes API joueront le rôle du composant «standards». Le composant «gouvernance» est délégué à l'application cliente, même si une partie de la gouvernance est contenue dans la logique des services (conditions d'accès, services payants, *etc.*).

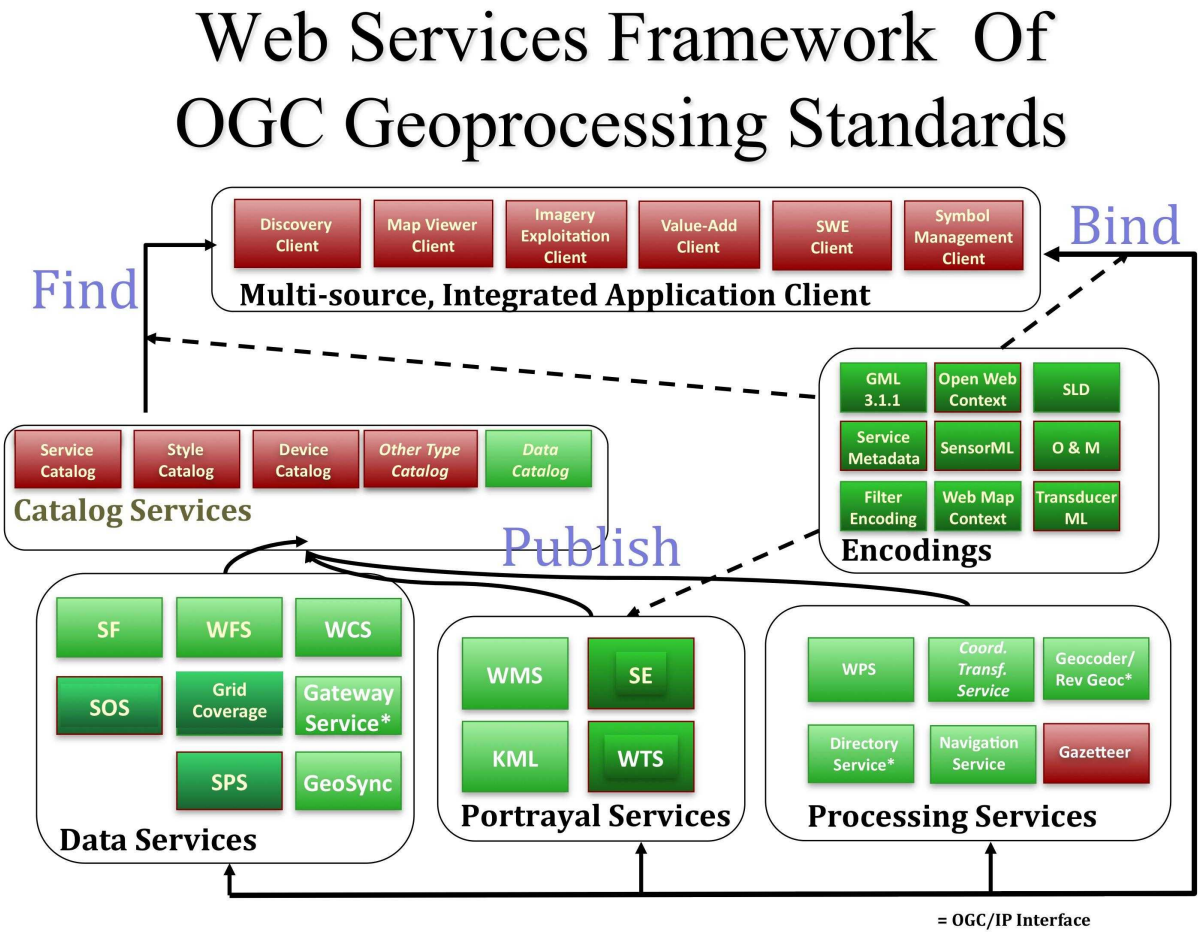


FIGURE 2.8 – Standards de l'OGC
Source : [@OGC]

Les services ou les formats de données sont définis à l'aide de schémas qui permettent aux différents intervenants d'éviter la phase de décryptage des différentes API. Les différents acteurs, lorsqu'ils émettent une requête, précisent le format désiré pour la réponse.

La publication (*publish*) se fait entre les services de type transfert de données, visualisation et traitement vers les services de catalogage. Le catalogue, outre bien sûr le type et la localisation du service publié, expose des métadonnées sur le service ou les données qu'il permet d'obtenir, de visualiser ou de traiter. Les métadonnées peuvent, par exemple, concerner l'organisme fournissant le service, la

région géographique ou le type des données du service, le langage de requête supporté par le serveur ou des informations sur les opérations du service.

La découverte est une transaction entre un client et un service de catalogage, qui a pour but de fournir aux clients la liste des services correspondant à ces besoins. Une requête *GetRecords* sur un service de catalogage avec comme paramètre une région géographique [11] retournera la liste des services catalogués (accès au service et métadonnées), possédant des informations/cartes/traitements relatifs à la région étudiée.

L'utilisation du service dépend de la nature du service. Toutefois, les services de l'OGC forment une hiérarchie de classes et ont des points communs.

Le diagramme de la figure 2.9 montre les héritages mis en jeu dans cette hiérarchie. Chaque service de l'OGC hérite du service abstrait *OGC Web Service*, c'est-à-dire qu'on peut lui adresser une requête *GetCapabilities*. Cette requête retourne un fichier basé sur XML qui décrit les capacités du service. Entre autres informations ce fichier décrit les méthodes du service et comment les invoquer. Dans la suite de notre mémoire nous appellerons ce fichier le fichier *capabilities*. Ensuite, au sein de la hiérarchie, certains services sont également spécialisés. Par exemple, le service StyledLayerDescriptorWMS spécialise le service WMS en ajoutant la possibilité d'effectuer une requête *DescribeLayer*.

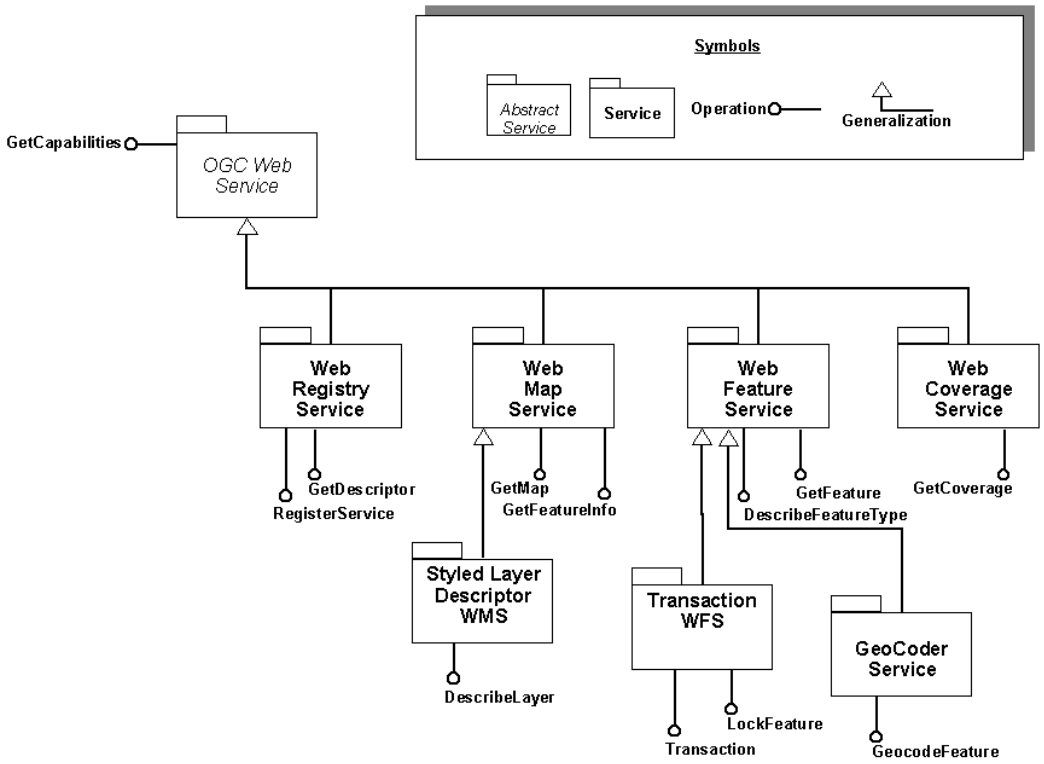


FIGURE 2.9 – Services de l'OGC
Source : [@OGC]

11. Une région géographique est définie par le paramètre BBOX (BoundingBox) qui est utilisé dans de nombreux services. Il définit une région à l'aide de quatre paramètres correspondant aux limites de la région (est, sud, ouest et nord). Ce paramètre n'a de sens qu'accompagné du paramètre SRS (Spatial Reference System) qui définit la projection utilisée pour la région donnée.

2.2.3 Transactions

Toute transaction avec un service de l'OGC commence par une requête *GetCapabilities*. En se basant sur les informations qu'il a obtenues par l'entremise du fichier *capabilities*, le client du service peut ensuite effectuer les requêtes correspondant à son besoin. Ensuite, selon le type de service, les possibilités diffèrent, mais une requête à un service OGC est toujours composée des éléments suivants :

– le paramètre SERVICE correspondant au type du service invoqué ;
– le paramètre REQUEST précisant l'opération que le client va déléguer au service ;
– le paramètre VERSION spécifiant la version du service. Selon les versions, l'API d'un service est différente. Le fichier renvoyé par un *GetCapabilities* pour un service WMS version 1.3.0 n'est pas tout à fait équivalent à celui renvoyé par la version 1.1.1 (la version la plus courante de ce service). L'encodage GML a lui aussi plusieurs versions (une dizaine). La première version ne comportait que des éléments de géométrie plane et des triplets «propriété» (type/nom du type/valeur) alors que les dernières versions permettent d'utiliser la dimension temporelle d'un objet géographique ou des éléments de topologie ;
– des paramètres obligatoires (définis par la spécification), dépendant de la requête. Parmi ces paramètres se trouve souvent le paramètre FORMAT si la spécification admet plusieurs formats pour la réponse. Si une requête *GetCapabilities* n'admet qu'un type de format comme réponse, la requête *GetMap* du service WMS est beaucoup plus ouverte (Bitmap, jpeg, png, gif, *etc.*) ;
– des paramètres optionnels, mais prévus par la spécification. Si les paramètres sont absents de la requête, celle-ci doit néanmoins aboutir. Concrètement cela signifie que ces paramètres ont une valeur par défaut, précisée ou non dans la spécification. Pour une requête GetMap, le paramètre TRANSPARENT est optionnel, et par défaut vaut FAUX selon la norme, le paramètre BGCOLOR, qui correspond à la couleur de fond de la carte est optionnel aussi, mais la norme n'impose pas de valeur par défaut, et donc celle-ci dépendra du service ;
– des paramètres VENDORSPECIFIC, c'est-à-dire des paramètres propriétaires, que la plupart des clients ne pourront pas *a priori* préciser. Comme dans le cas des paramètres optionnels, la requête doit aboutir si ces paramètres sont absents de la requête.

La figure 2.10 résume cette description, la séparation en pointillé séparant les paramètres communs à tous les services des paramètres spécifiques d'une requête.

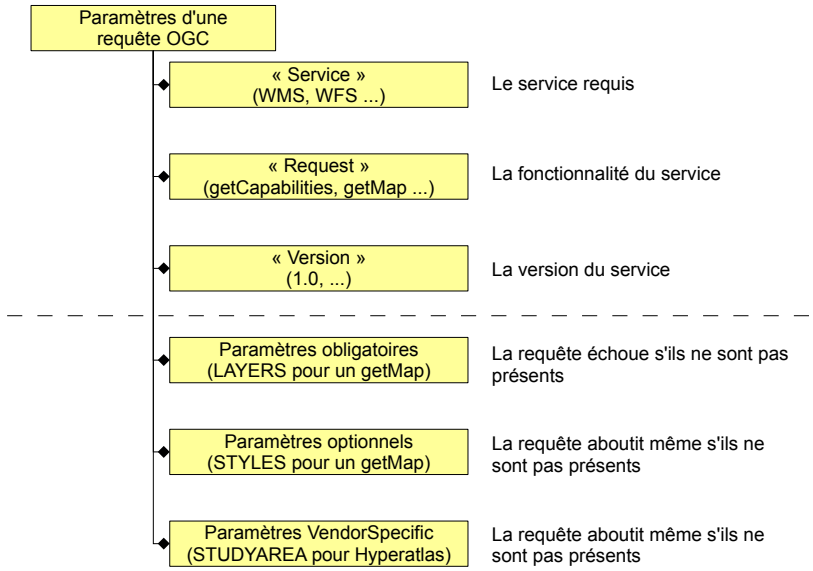


FIGURE 2.10 – Paramètres d'une requête à un service de l'OGC

2.2.4 L'interopérabilité

L' interopérabilité est la capacité que possède un produit ou un système, dont les interfaces sont intégralement connues, à fonctionner avec d'autres produits ou systèmes, existants ou futurs, et ce, sans restriction d'accès ou de mise en œuvre [@Wikipedia, c].

Pour les fournisseurs de service comme pour les clients, le bénéfice est clair :
– le fournisseur du service a déjà des clients *a priori* capables d'utiliser son service ;
– le nombre de services que le client peut exploiter ne fait qu'augmenter.

Dans le cadre des services de l'OGC, l'interopérabilité permet d'aller un peu plus loin : on peut combiner les services, et ainsi créer de la valeur ajoutée aux contenus accédés. Les possibilités sont virtuellement infinies. La combinaison peut être parallèle (la combinaison de cartes issues de différents WMS permet d'obtenir une carte plus riche), en série (la combinaison de plusieurs algorithmes de WPS), ou en cascade entre services de nature différente.

Les figures 2.11 et 2.12 illustrent ces deux façons de combiner les services.

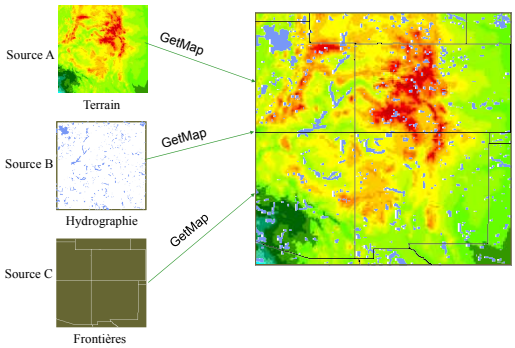


FIGURE 2.11 – Combinaison parallèle de plusieurs services WMS
Source : [@OGC]

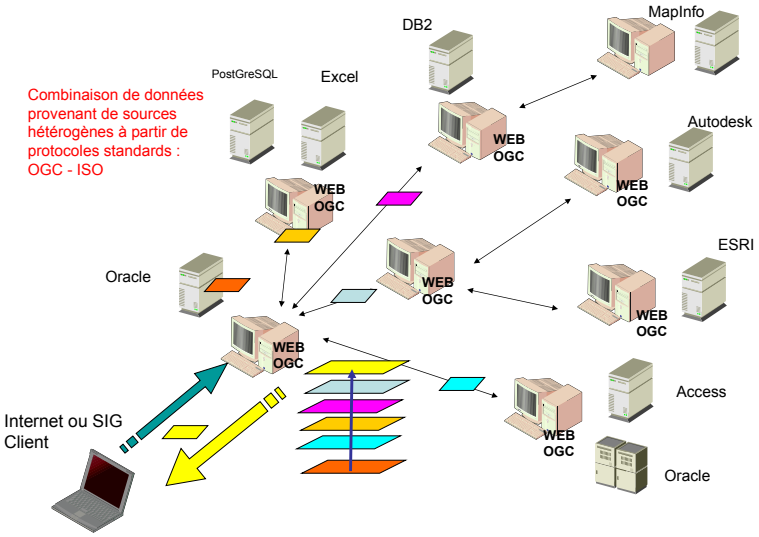


FIGURE 2.12 – Combinaison en cascade de plusieurs services OGC
Source : [@OGC]

L'orchestration des différents services permet de créer une infrastructure de données géographiques

à partir de données hétérogènes et distribuées. Ce flux peut ensuite être utilisé via un géoportail qui présentera les services en les organisant par thématiques ou par régions. Ce portail pourra être consulté directement, grâce à des clients intégrés qui permettent de visualiser les données qu'il référence, ou utilisé via des appels à ses services. La figure 2.13 décrit cette orchestration. Le géoportail permet d'atteindre les différents éléments de la SDI à travers l'Internet : les services de Catalogage dans un premier temps, puis les services de visualisation (*Portrayal*) et les services de données (*Data*).

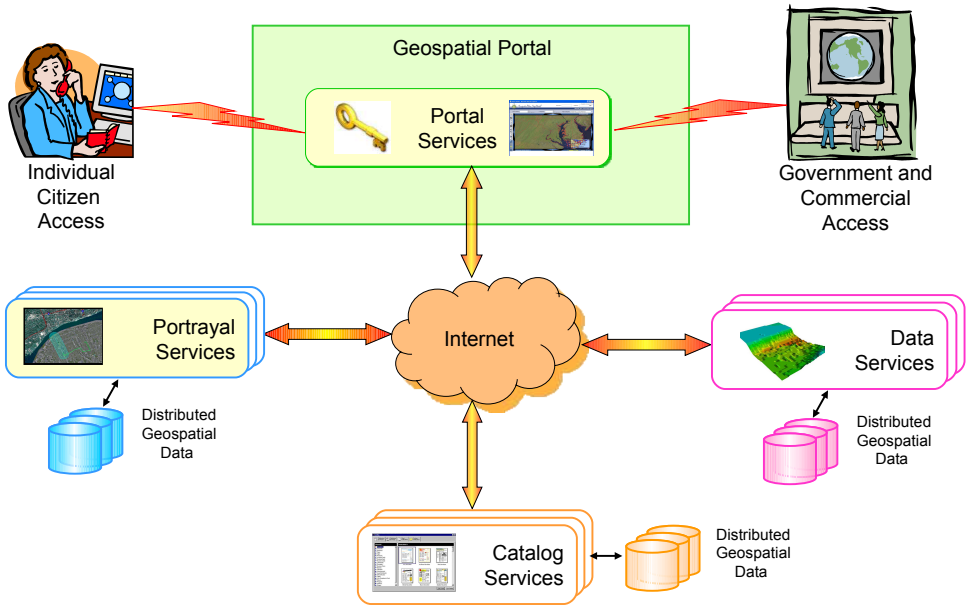


FIGURE 2.13 – Architecture d'un géoportail basé sur des services OGC
Source : [@OGC]

2.2.5 Exemples de spécifications

L'OGC *Reference Model* décrit le cadre de développement complet des spécifications de l'OGC. Il comprend des documents de spécification décrivant les modalités d'utilisation des différents types de service, et des schémas basés sur XML correspondant aux données échangées (les fichiers *capabilities* notamment). Résumer cette masse d'information [12] est impossible. Les services décrits dans la suite correspondent à ceux qui sont les plus implémentés (d'après les statistiques de l'OGC, qui ne prennent en compte que les services enregistrés auprès du consortium, qui sont la partie émergée de l'iceberg), ou à ceux qui sont caractéristiques d'un besoin.

WMS : Web Map Service. Le WMS, qui est le service que nous avons retenu pour notre proposition, fait l'objet de la section suivante où il sera abordé en détail. Il permet d'afficher et d'interagir avec des cartes ;

WFS : Web Feature Service. Le service WFS permet, au moyen d'une URL formatée, d'interroger des serveurs cartographiques afin de manipuler des objets géographiques (points, lignes, polygones). *GetCapabilities* permet de connaître les capacités du serveur (quelles opérations sont

12. Une dizaine de mégaoctets de fichiers compressés.

supportées et quels objets sont fournis). *DescribeFeatureType* permet de retourner la structure de chaque entité susceptible d'être fournie par le serveur. *GetFeature* permet de livrer des objets (géométrie et/ou attributs) en GML. *LockFeature* permet de bloquer des objets lors d'une transaction. Transaction permet de modifier l'objet (création, mise à jour, effacement). *DescribeFeatureType* et *GetFeature* sont les deux requêtes obligatoires. Un WFS avec ses deux seules requêtes peut être considéré comme un WFS en lecture seule. Une requête *GetFeature* sera faite à l'aide d'une région géographique (BBOX) et d'un système de références géographiques (SRS).

WPS et WCTS : Web Processing Service et Web Coordinate Transformation Service. Ces services différent des autres services car ils ne permettent pas d'accéder à des informations, mais à des traitements. Le WPS permettra, par exemple, d'appliquer différents algorithmes liés aux géométries des entités géographiques. *DescribeProcess* renvoie une description des traitements offerts, ainsi que des formats d'entrée et de sortie. *Execute* permet d'invoquer le traitement. WCTS permet lui de convertir un système de références spatiales en un autre grâce à la requête *Transform*.

SOS, SAS, SPS : Sensor Observation Service, Sensor Alert Service, Sensor Planning Service. L'OGC a spécifié plusieurs standards permettant d'utiliser des capteurs. L'orchestration de ces standards permet de créer un flux d'informations géolocalisées à partir de ces capteurs, ce flux alimentant éventuellement ensuite les services classiques de l'OGC. Cette suite de standards forme le SWE (Sensor Web Enablement). Les échanges sont basés sur le standard SensorML (Sensor Modeling Language), basé sur XML, qui décrit les capteurs, et le standard Observation and Measurements qui sert à décrire les mesures réalisées par ces capteurs. La figure 2.14 décrit ce flux : l'information est créée depuis les capteurs (prise de vue aérienne, satellitaire, balises Argos, *etc.*) puis dirigée soit directement vers des applications clientes, soit vers d'autres services de l'OGC qui vont la traiter et l'intégrer dans leur propre flux.

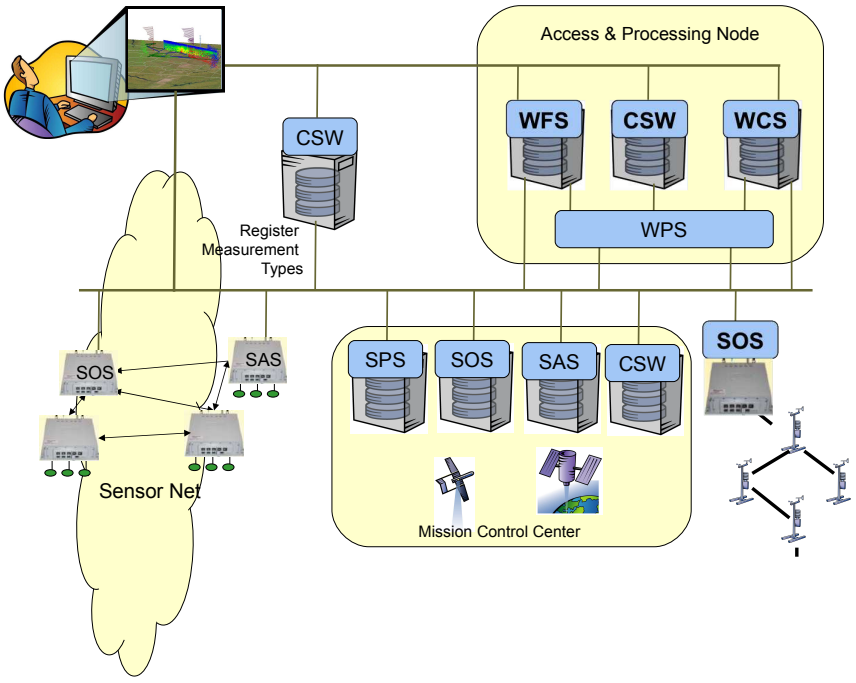


FIGURE 2.14 – SWE et le flux geoprocessing
Source : [@OGC]

2.2.6 Spécification WMS

Web Map Service. Le WMS permet de produire des cartes de données géo-référencées à partir de différents serveurs de données. Cela permet de mettre en place un réseau de serveurs cartographiques à partir desquels des clients peuvent construire des cartes interactives. Les modalités de cette interaction seront décrites dans la section suivante qui traite des clients WMS.

Il existe deux types de WMS : WMS et Styled Layer Descriptor WMS (SLD-WMS). Cette section traite essentiellement le WMS simple. SLD-WMS est une spécialisation de WMS (toutes les fonctionnalités de WMS sont présentes dans SLD-WMS) qui offre des fonctionnalités supplémentaires pour styliser les cartes. Un client peut traiter un service SLD-WMS comme si c'était un WMS (il n'utilise pas les fonctionnalités SLD).

Cependant, avant d'aborder les requêtes permettant d'exploiter les fonctionnalités du WMS, il faut d'abord, *comme pour tous les services Web de l'OGC*, émettre une requête *GetCapabilities* pour le découvrir.

La requête *GetCapabilities*

La requête *GetCapabilities* retourne un fichier *capabilities*. C'est un fichier basé sur le langage XML qui décrit le fichier à l'aide d'informations stockées dans une arborescence. La figure 2.15 montre que cette arborescence distingue deux types d'informations :

– les métadonnées du service, qui se trouvent sous le nœud Service. Dans ces métadonnées on trouve des informations concernant le fournisseur du service (le nœud ContactInformation et ses sous-éléments qui n'apparaissent pas sur la figure) ou les conditions d'accès au service (il peut être payant), et des éléments sur la nature des données que le service expose, comme par exemple une liste de mots-clés ;
– des éléments fonctionnels sous le nœud Capability, que nous allons détailler.

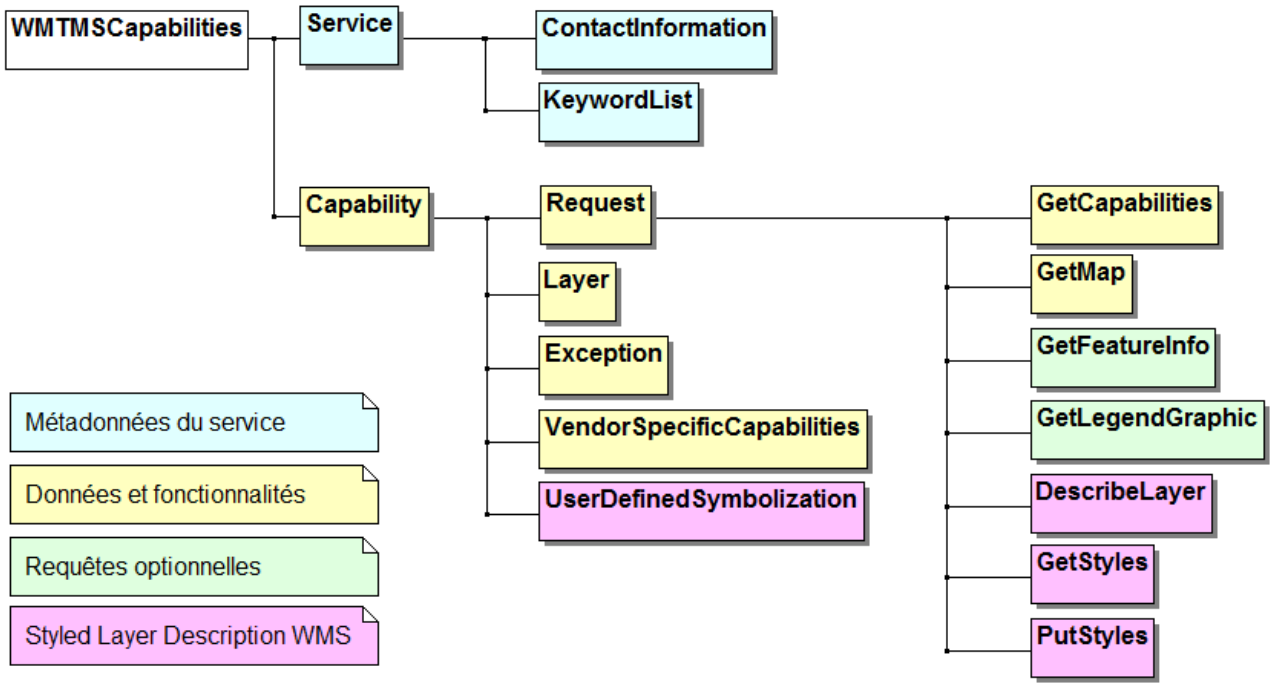


FIGURE 2.15 – Arborescence du fichier *capabilities*

Request : l'élément Request contient les informations relatives aux requêtes qui sont supportées par le service : la spécification WMS n'impose qu'une requête en sus de la requête *GetCapabilities* : la requête *GetMap*. Les autres requêtes sont optionnelles (en vert sur la figure 2.15, les éléments en rose sont optionnels et spécifiques du SLD-WMS). Pour signifier au client qu'une requête n'est pas supportée, il suffit de ne pas la mentionner dans le fichier *capabilities*. L'extrait suivant provient du fichier *capabilities* du service HyperAtlasWMS :

```
<GetMap>
  <Format>image/png</Format>
  <Format>image/jpg</Format>
  <Format>image/jpeg</Format>
  <Format>image/bmp</Format>
  <Format>image/gif</Format>
  <DCPType>
    <HTTP>
      <Get>
        <OnlineResource xlink:href=
          "http://localhost:8080/Europe_ESPON_2007/Hyperatlas?"/>
      </Get>
    </HTTP>
  </DCPType>
</GetMap>
```

Les balises <Format> correspondent aux formats supportés par la requête pour la réponse. La balise <DCPType> [13] correspond aux plates-formes supportées. Dans le cas du WMS, seul HTTP est supporté. La balise <Get> indique l'adresse à laquelle le client devra émettre la requête : la spécification n'impose pas que les différentes requêtes d'un service doivent être émises vers la même adresse.

Layer : la balise <Layer> correspond à une couche d'information du service. Elle peut elle même contenir d'autres couches. Une couche est tout d'abord décrite par une série d'attributs prenant des valeurs booléennes (0 : FAUX, 1 : VRAI). Les deux plus importants sont :

- opaque : la couche masquera les autres couches (c'est le cas lorsque la couche correspond à une photographie satellite servant de fond pour la carte). Lorsqu'une couche est opaque, pour éviter qu'elle ne masque les autres couches, il vaut mieux la placer en tête des couches de la requête *GetMap*. Pour la même raison, se limiter à une seule couche opaque est souhaitable ;
- queryable : cet attribut indique si l'on peut émettre des requêtes *GetFeatureInfo* pour cette couche.

Les sous-éléments de <Layer> sont :

- <Name>, <Title> et <Abstract> qui décrivent la couche. Le contenu de <Name> est la valeur qu'il faudra passer en paramètre d'une requête, les contenus de <Title> et <Abstract> sont plus compréhensibles pour un humain et pourront être utilisés par une interface graphique ;
- <Attribution> contient des éléments relatifs au propriétaire de la couche ;
- les attributs de <LatLonBoundingBox> permettent de connaître les coordonnées géographiques (latitude, longitude) de l'étendue de la carte ;
- <BoundingBox> est composé de deux éléments : les limites de l'étendue géographique et le système de références spatiales. Il peut y avoir plusieurs éléments <BoundingBox> ;
- <Style> contient la description d'un style pour la carte. Il peut y en avoir plusieurs. Pour les services WMS simples le style est identifié par une simple chaîne de caractères (default dans l'exemple suivant), pour les services SLD-WMS, les styles sont beaucoup plus élaborés, construit à partir d'éléments XML.

L'extrait de code XML est lui aussi tiré du fichier *capabilities* du service HyperAtlasWMS. La

13. DCP : Distributed Computing Platforms, ce qui se traduit par plate-formes informatiques distribuées.

description de la couche a été supprimée, ainsi que les éléments pouvant apparaître plusieurs fois pour alléger le texte.

```
<Layer opaque="0" cascaded="0" queryable="1">
  <Name>STUDY</Name>
  <Title>Study area</Title>
  <Abstract> ... description de la couche ... </Abstract>
  <Attribution>
    <Title>STEAMER</Title>
    <OnlineResource xlink:href="http://steamer.imag.fr/"
       xlink:type="simple"/>
  </Attribution>
  <LatLonBoundingBox maxy="90" maxx="180" miny="-90" minx="-180"/>
  <BoundingBox maxy="2518193.74356659" maxx="1783333.27849457"
    miny="-1723801.06972807" minx="-2217174.75620719" SRS="EPSG:4326"/>
  ... autres balises <BoundingBox>
  <Style>
    <Name>default</Name>
    <Title>Hypercarte plain map style.</Title>
    <Abstract>Map is displayed without legend.</Abstract>
  </Style>
<Style>
... autres balises <Style>
</Layer>
```

Exception : la balise <Exception> contient une liste de formats supportés par le service pour retourner au client des messages d'exception, comme par exemple lorsque la requête émise a été mal formée (paramètre manquant, valeur incorrecte, *etc.*). La spécification a un format par défaut qui est basé sur XML, mais dans le cas du WMS on peut utiliser des images affichant le message de l'exception. Pour chaque requête, le paramètre optionnel EXCEPTIONS permet de préciser le format souhaité pour l'exception.

VendorSpecificCapabilities : le contenu de ces balises dépend du propriétaire du service. Il n'y a aucune contrainte ni aucun modèle imposés. Il peut décrire des requêtes additionnelles ou des paramètres. La seule contrainte est que le service doit fonctionner, même dans un mode dégradé, si les paramètres VendorSpecific sont omis lors de l'envoi de requêtes de la spécification.

La requête *GetMap*

La requête *GetMap* renvoie une carte sous forme d'image. Le tableau suivant décrit les paramètres obligatoires d'une requête (en plus de SERVICE=WMS, REQUEST=GETMAP et VERSION).

Nom	Description	Exemple
FORMAT	le format de l'image	FORMAT=image/png
WIDTH	la largeur de l'image (en pixels)	WIDTH=800
HEIGHT	la hauteur de l'image (en pixels)	HEIGHT=600
BBOX	limites de l'étendue de la carte, séparées par une virgule : abscisse minimale, ordonnée minimale,abscisse maximale, ordonnée maximale	BBOX=-180,-90,180,90
SRS	système de références spatiales	SRS=ESPG :4326
LAYERS	couches de la carte, séparées par une virgule	LAYERS=borders,labels

TABLE 2.1 – Paramètres d'une requête *GetMap*

Les paramètres optionnels sont TRANSPARENT, BACKGROUND, STYLES et EXCEPTIONS, qui a été décrit précédemment. TRANSPARENT définit si l'image retournée est transparente, la valeur par défaut est Vrai (TRASNPARENT=true). C'est cette valeur qu'un client doit utiliser s'il souhaite superposer le résultat de la requête au résultat d'une requête vers un autre service WMS. Il doit faire attention au format, car tous les formats ne permettent pas les images transparentes. Lorsque que le client choisit une image opaque, il peut préciser une couleur pour le fond de la carte : les zones de la carte sans information seront de cette couleur. L'encodage des couleurs pour les services de l'OGC est 0xRRGGBB, ou RR, GG et BB sont la valeur sous forme hexadécimale du rouge (*red*), du vert (*green*) et du bleu (*blue*). Le paramètre STYLES correspond aux choix des objet <Style> pour chaque couche, le paramètre a donc autant de valeurs qu'il y a de couches dans le paramètre LAYERS (par exemple, LAYERS=layer1,layer2 & STYLES=style1,style2).

La figure 2.16 montre le résultat d'une requête : le territoire est défini par les paramètres BBOX et SRS, la taille de l'image est définie par les paramètres WIDTH et HEIGHT, le contenu et son apparence correspond aux paramètres LAYERS et STYLES, le format retourné dépend des paramètres FORMAT, TRANSPARENT et BACKGROUND lorsque la requête est un succès, EXCEPTIONS dans les cas contraires.

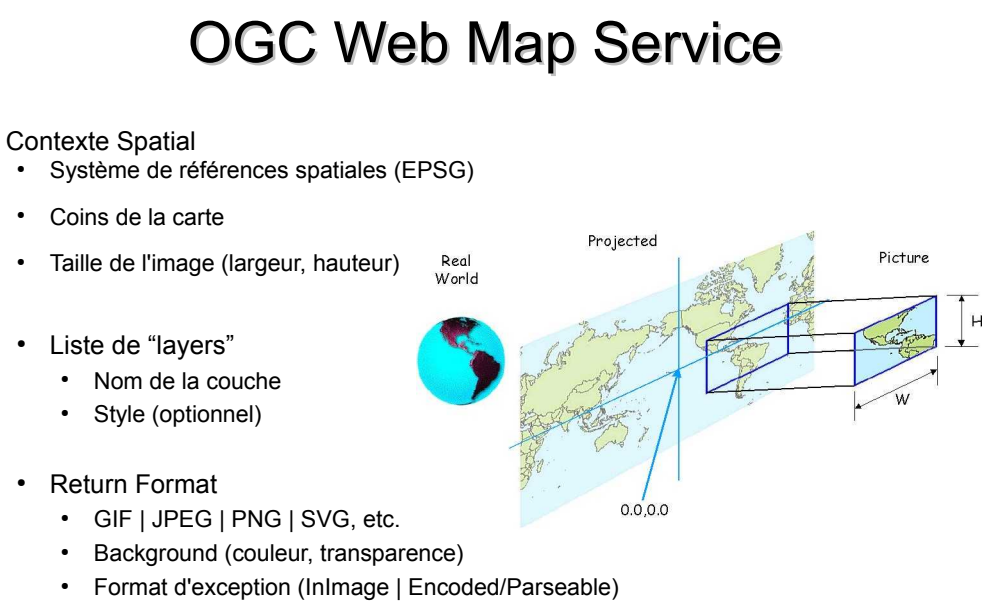


FIGURE 2.16 – Requête *GetMap*
Source : [@OGC]

La requête *GetFeatureInfo*

L'objectif d'une requête *GetFeatureInfo* est de renseigner le client sur des éléments d'une carte. Le service WMS est sans état, la requête *GetFeatureInfo* doit donc contenir tous les paramètres de la requête *GetMap*, à l'exception des paramètres REQUEST et VERSION, qui seront présents dans la requête, mais contiendront les valeurs correspondant à la requête *GetFeatureInfo*.

Les paramètres obligatoires à ajouter dans cette requête sont :
– QUERY_LAYERS est renseigné avec le nom de la couche à laquelle on s'intéresse : pour un même point de la carte plusieurs couches peuvent contenir des informations, il convient donc de préciser la couche sur laquelle porte la requête *GetFeatureInfo* ;
– X et Y correspondent aux coordonnées du point de la carte correspondant à la requête. Ces coordonnées sont celles d'un point de la carte, pas du territoire. Elles sont donc celles d'un pixel (valeurs entières) de l'image correspondant à la requête. Sur une image l'origine (0,0) est le coin supérieur gauche de l'image (le repère est indirect).

Le texte d'une requête *GetFeatureInfo* sera :

$$\underbrace{\text{http ://serveur :port/service ?SERVICE=WMS\&VERSION=1.1.1\&REQUEST=getfeatureinfo}}_{\substack{\textit{adresse du service} \\ \textit{type de la requête et version}}}$$

$$\underbrace{\text{\&LAYERS=layer1,layer2\&WIDTH=800\&...}\textit{etc.}\text{\&QUERY\_LAYER=layer1\&X=50\&Y=75}}_{\substack{\textit{paramètres de la requête GetMap} \\ \textit{paramètres de GetFeatureInfo}}}$$

Les paramètres optionnels sont, à nouveau EXCEPTIONS, puis INFO_FORMAT pour le format de la réponse et FEATURE_COUNT qui correspond au nombre d'éléments souhaités dans la réponse (un par défaut).

La requête *GetLegendGraphic*

Il y a deux façons pour un client d'obtenir la légende d'une carte. L'élément <Layer> peut contenir un élément <LegendURL> qui permet d'accéder directement à la ressource (un fichier image) correspondant à la légende ou émettre une requête *GetLegendGraphic*, qui retourne la légende associée à une couche de la carte.

Les seuls paramètres obligatoires sont LAYER et FORMAT, pour la couche concernée par la légende et le format retourné. Les paramètres optionnels sont très nombreux, parmi eux SCALE qui correspond à l'échelle de la carte, ou WIDTH et HEIGHT pour imposer au service une taille d'image. Les autres paramètres concernent le style.

2.2.7 Client WMS

Le client le plus naturel pour un service WMS est un navigateur Web puisque la communication entre service et client s'effectue au moyen d'une requête HTTP, mais l'interactivité est inexistante, et taper à la main des requêtes n'est ni évident, ni plaisant.

L'échange entre un client et un service WMS suit une séquence bien définie. Dans un premier temps, le client demande à l'utilisateur de lui indiquer l'adresse d'un service WMS. Ensuite, le client émet une requête *GetCapabilities* auprès du service. À l'aide du fichier retourné, le client extrait la liste des couches qu'il pourra afficher. À ce moment, l'utilisateur est sollicité pour sélectionner les couches qu'il souhaite afficher. La carte est alors affichée. L'utilisateur peut ensuite interagir avec cette carte via l'interface graphique. À chaque interaction une nouvelle requête est émise auprès du service, ce qui peut prendre du temps selon les performances du serveur et du client :
– ajouter/supprimer/réordonner [14] des couches, le paramètre LAYERS est modifié ;
– zoomer ou se déplacer sur la carte, le paramètre BBOX est modifié ;

14. Réordonner les couches est parfois nécessaire, car si une couche opaque a été dessinée en dernier, elle masque toutes les autres couches.

– agrandir/réduire la carte, les paramètres WIDTH et HEIGHT sont modifiés ;
– demande d'informations supplémentaires pour un point donné de la carte, une requête *GetFeatureInfo* est émise ;
– afficher la légende, une requête *GetLegendGraphic* est émise ;
– la carte peut être sauvegardée, imprimée, combinée à d'autres cartes, selon les capacités du client et l'objectif de l'utilisateur.

Les clients peuvent tout d'abord être classés en deux catégories, les clients en ligne, en général intégrés à un géoportail, et les applications sur poste, c'est-à-dire installées et s'exécutant sur la machine de l'utilisateur.

Nous avons effectué un banc d'essai de plusieurs clients WMS, afin de voir s'ils pouvaient être utilisés pour *remplacer l'interface graphique d'HyperAtlas dans le cadre de notre projet*. Tous les clients testés l'ont été avec les services `http://nsidc.org/cgi-bin/atlas_north` et.`http://services.sandre.eaufrance.fr/geo/zonage-shp`. Le premier, qui permet d'afficher des données relatives à l'Arctique, a été choisi car il possède des couches variées (imagerie satellitaire, polygones, lignes, points d'intérêt) et de nombreux référentiels géographiques. Le second, qui propose des données hydrographiques de la France, car il permet d'utiliser la requête *GetFeatureInfo* sur des objets superposés ce qui n'est pas nécessairement simple pour un client. De plus, les deux services implémentent *GetLegendGraphic*.

Tests des clients en ligne

Si les clients en ligne ont pour avantage de ne nécessiter aucune installation, ils ont des inconvénients :
– ils sont lents, car la requête est émise depuis le serveur d'application vers le serveur WMS, traité par le client sur le serveur d'application, et enfin affichée dans le navigateur ;
– ils sont prévus pour mettre en valeur les services du géoportail qui les héberge, et ne sont pas toujours capables d'utiliser d'autres services. D'ailleurs, peu d'entre eux proposent cette possibilité ;
– hormis l'exploration de la carte ils offrent très peu de fonctionnalités.

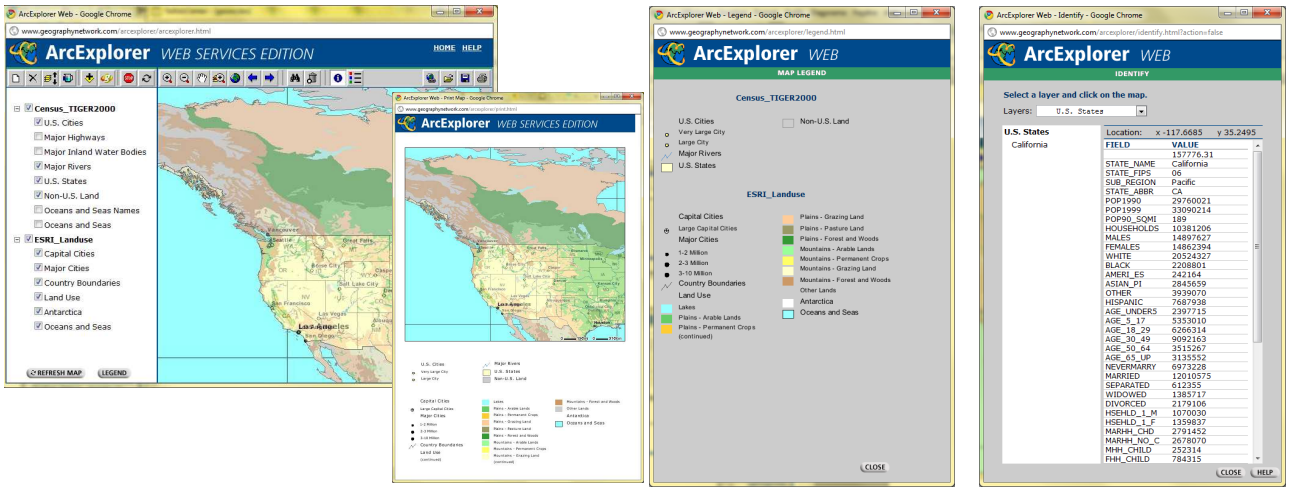
MapPlace : `http://webmap.em.gov.bc.ca/mapplacewms/servlet/com.autodesk.wmsviewer.WmsViewer?request=startviewer`. Ce client développé par *Autodesk, Inc* [15] n'a pu se connecter à aucun des deux services.

ArcExplorer Web : `http://www.geographynetwork.com/arcexplorer/arcexplorer.html`. Ce client développé par ESRI [16] ne fait pas partie d'un géoportail. Il a réussi à se connecter aux services, à présenter les couches du service, mais n'a pas réussi à afficher de carte. C'est d'autant plus regrettable que ce client, même s'il est esthétiquement critiquable, offre de nombreuses fonctionnalités par rapport aux autres clients en ligne :
– exploration de la carte ;
– afficher la légende (*GetLegendGraphic*) ;
– informations supplémentaires pour un point donné de la carte (*GetFeatureInfo*) ;
– impression de la carte ;
– sauvegarde de la carte (nécessite un compte) ;
– création de lien vers la carte (HTML à copier).

La figure 2.17 présente des captures d'écran réalisé avec ArcExplorer Web en combinant deux des services qu'il propose : Census_TIGER2000 et ESRI_Landuse. La première correspond à l'interface après avoir sélectionné les services (figure 2.17-a à gauche). La seconde au rapport créé, qui combine la carte et la légende (figure 2.17-a à droite). La troisième est le résultat d'une requête *GetLegendGra-*

15. Société spécialiste des logiciels de modélisation 3D.
16. Société spécialiste des systèmes d'information géographique.

phic (figure 2.17-b à gauche). La dernière est le résultat d'une requête *GetFeatureInfo* (figure 2.17-b à droite).



(a) interface et impression ArcExplorer Web (b) GetLegendGraphic et GetFeatureInfo ArcExplorer Web

FIGURE 2.17 – ArcExplorer Web

Portail Inspire : http://geoportal.jrc.it/index.cfm/pageid/341. Ce client est parvenu à se connecter aux services et à afficher les différentes couches. Toutefois, ce client ne gère que le système de références spatiales WGS84 [17], mais pas les projections. Cette limitation est très gênante pour la carte de l'Arctique, car elle apparaît très déformée. Ce client ne permet aucune autre fonctionnalité que l'exploration de la carte (zoom et translation), et l'affichage est très lent : deux minutes entre chaque interaction pour la carte ci-dessous, qui affiche cinq couches.

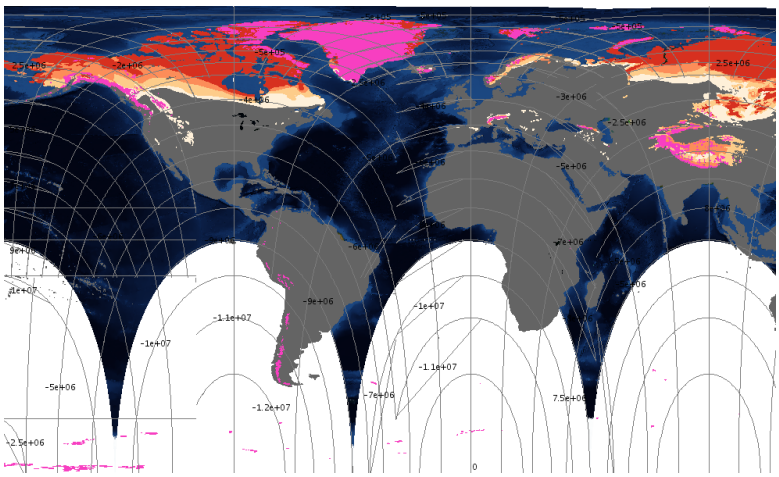


FIGURE 2.18 – Carte de l'Arctique en WSG 84

La conclusion de cette première série de tests est que les clients en ligne servent avant tout à présenter des implémentations de services WMS, mais ne peuvent pas véritablement être utilisées en tant qu'outils par rapport à leurs performances et/ou leur polyvalence.

17. World Geodetic System 1984 est le système géodésique associé au GPS.

Tests des clients sur poste

Avec les mêmes services quatre clients sur poste ont été testés. Le premier est GoogleEarth [18], que l'on ne présente plus. Le second est Quantum GIS [19], un projet officiel de l'OSGeo. Le troisième, uDig [20], est un projet *open source* [21] basé sur le cadriciel Eclipse RCP (Rich Client Platform), soutenu par la compagnie Refractions Research. Le dernier est Gaïa [22], un logiciel gratuit basé sur l'environnement de développement .NET.

GoogleEarth : Si GoogleEarth permet d'émettre des requêtes WMS, il n'a clairement pas été conçu pour cela. Tout d'abord, trouver la fonctionnalité est presque impossible sans consulter l'aide (1.menu Ajouter, 2.Superposition d'image, 3.onglet actualisation lorsque le dialogue s'affiche, 4. bouton «paramètres WMS»). De plus, GoogleEarth ne permet pas de choisir le référentiel, ce qui fait que certaines couches n'apparaissent pas. Enfin, GoogleEarth a une façon étrange d'utiliser les services : il fait une requête en fonction de l'affichage et «tente» ensuite de coller l'image qu'il a obtenue sur le globe. Le résultat est souvent mauvais, et a tendance à se dégrader à mesure que l'on explore la carte. Voici une liste des principaux problèmes rencontrés :
– les éléments de la carte ne coïncident pas avec les éléments affichés par GoogleEarth ;
– la transparence est mal gérée (une couche transparente devient blanche) ;
– la couche WMS n'est que partiellement affichée sur la zone sélectionnée ;
– aucune interaction avec le service (*GetLegendGraphic* et *GetFeatureInfo)* ;
– parmi les services pré-sélectionnés, plusieurs ne sont pas compatibles (échec lors de la lecture du *GetCapabilities)* ;
– la couche WMS n'est que partiellement affichée sur la zone sélectionnée.

Pour conclure, le fait d'utiliser WMS avec GoogleEarth ressemble plus à une concession faite par le géant à l'OGC qu'à une réelle fonctionnalité du logiciel.

Quantum GIS : Quantum GIS est un logiciel de cartographie SIG (Système d'Information Géographique) *open source* capable de gérer des images matricielles comme vectorielles provenant de bases de données, de services WMS ou WFS, et même de GPS. La prise en main de Quantum GIS est immédiate, l'interface principale est très bien conçue (chaque fonctionnalité peut-être déclenchée via un menu ou via une barre d'outils). Importer depuis un service WMS est simple, la liste des couches disponibles est affichée, et sous chaque couche un nœud permet de choisir le style associé à la couche. On peut aussi choisir parmi les référentiels disponibles. Cette phase de sélection terminée, la carte s'affiche, mais les couches non-disponibles pour le référentiel sont escamotées sans message. Dans la suite, l'ensemble des couches est considérée comme une seule couche, ce qui empêche de sélectionner/dé-sélectionner les couches, ou surtout de les réordonner. Pour pouvoir gérer les couches individuellement, il faut les sélectionner une par une, ce qui nécessite à chaque fois de refaire la séquence *GetCapabilities*/défilement de l'arbre des couches/choix du référentiel. L'exploration de la carte est efficace (outil de mesure de distance ou d'aire en plus de l'exploration classique), *GetFeatureInfo* interroge l'ensemble des couches sélectionnées et affiche les informations obtenues dans dialogue, mais *GetLegendGraphic* n'est pas exploité. Si la première impression est excellente, à l'usage on découvre que le logiciel comporte quelques bogues qui obligent l'utilisateur à redémarrer l'application, les modifications non sauvegardées étant perdues.

uDig : uDIG signifie User-friendly Desktop Internet GIS. Il est à la fois une application SIG et une plate-forme avec laquelle des développeurs peuvent créer de nouvelles applications dérivées car il

18. http://www.google.fr/intl/fr/earth/index.html.
19. http://qgis.org/.
20. http://udig.refractions.net/.
21. Les sources du code sont publiques.
22. http://www.thecarbonproject.com/

est basé sur le *framework* Eclipse RCP. Dans sa configuration initiale, uDIG autorise uniquement la lecture des données vecteurs au format ShapeFile. Le choix est par contre beaucoup plus vaste pour celles stockées dans un SGBD. Les bases que l'on peut interroger sont : ArcSDE, DB2, MySQL, Oracle Spatial et PostGis.

Pour les standards de l'OGC, uDig accède WFS et WMS. L'interface est exactement celle d'Eclipse, et l'on peut donc disposer chacun des éléments du projet (carte) en cours dans des panneaux que l'on peut disposer à sa guise. L'interactivité est riche (zoom, translation, outil de mesure de distance), les requêtes *GetFeatureInfo* sont affichées dans un onglet à part, triées en fonction des couches affichées. *GetLegendGraphic* est utilisé pour afficher une icône devant les nœuds des couches, mais pas de possibilité de voir véritablement la légende, ni de l'imprimer avec la carte. uDig est stable, mais quand on le relance toutes les cartes sont ré-initialisées avec le référentiel WGS84, ce qui oblige l'utilisateur à redéfinir la projection qu'il souhaite utiliser.

Gaïa : Gaïa est développé avec le *framework* .NET, ce qui signifie qu'il s'agit d'un ensemble de dll [23] et d'un exécutable.

Dans sa version gratuite, Gaïa permet, outre l'utilisation des services WMS/WFS/WCS, d'utiliser les cartes Yahoo Maps, Bing Maps (Microsoft) et OpenStreetMap, d'utiliser de nombreux fichiers contenant des géométries [24]. Ce qui différencie Gaïa des autres clients présentés, c'est que pour un utilisateur ayant une bonne connaissance des standards de l'OGC, il permet d'ajouter «manuellement» des paramètres aux requêtes, ce qui autorise un usage des paramètres optionnels ou *VendorSpecific*. De plus, Gaïa offre la possibilité de consulter le fichier *capabilities* de façon approfondie, ce qui, combiné avec la fonctionnalité précédente permet d'utiliser le service au mieux. L'interactivité est agrémentée par le fait que parallèlement à la nouvelle requête *GetMap*, Gaïa simule l'interaction avec l'image qu'il possède déjà, ce qui permet de donner l'impression que le zoom est animé. Comme pour uDig et Quantum GIS, la requête *GetLegendGraphic* n'est pas utilisée, l'onglet «Legend» étant réservé au WFS. Au niveau des fonctionnalités, en plus des impressions ou conversions de la carte, Gaïa permet d'ajouter des notes géoréférencées. Enfin, il est possible d'importer des extensions, ou même d'en développer avec l'outil CarbonTools PRO, mais cela nécessite une licence payante.

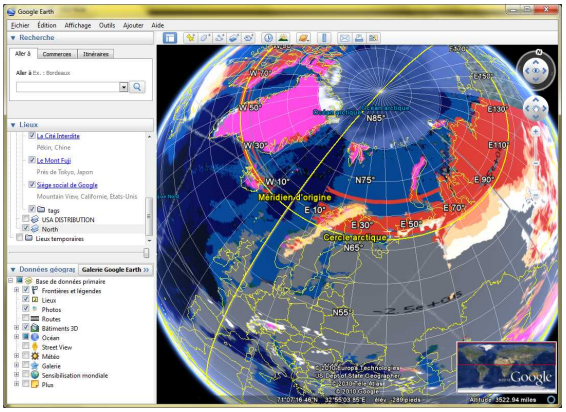
Conclusion du banc d'essai : GoogleEarth ne peut pas être utilisé comme un client WMS au sens de la définition de l'utilisabilité [@Wikipedia, d]. Les trois autres clients présentés peuvent eux être considérés comme tels, même s'ils ont en commun un défaut très important : la non exploitation de la requête *GetLegendGraphic*, ce qui a pour conséquence de créer des cartes sans légende, et donc privées de leur sens [Bertin, 1967]. Si, globalement, leur utilisation du standard WMS est comparable (Quantum GIS est tout de même moins abouti) le choix se fera en fonction du profil ou du besoin de l'utilisateur. Quantum GIS et uDig font partie d'une suite d'outils OSGeo4W, et dans ce cadre, les utiliser semble naturel. Quantum GIS correspondra mieux à des utilisateurs désireux de modifier les données géographiques, uDig à des utilisateurs/développeurs Java. Gaïa conviendra mieux aux utilisateurs travaillant avec Windows souhaitant créer des cartes à partir de sources de données hétérogènes.

23. Dynamic Link Library.
24. GML, MapInfo : Mif, Google : kml/kmz,ESRI : shp, AutoDesk : dxf.

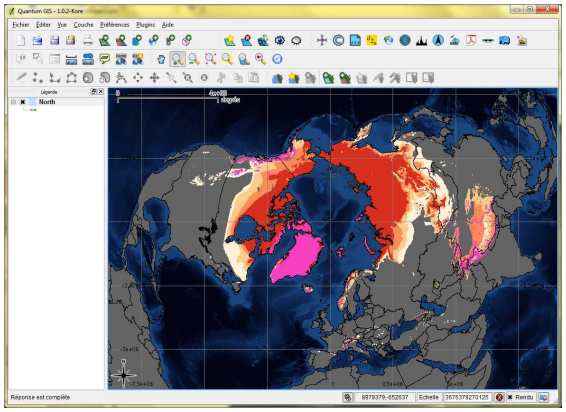
Le tableau suivant reprend les points principaux des trois clients. Suivent des captures d'écran réalisées durant les tests (figure 2.19).

	Quantum GIS	**uDig**	**Gaïa**
Ergonomie de la sélection des couches	mauvaise bogues	bonne icônes *legend*	bonne icônes *preview*
Zoom/Translation	oui	oui	oui
GetFeatureInfo	oui	oui	oui
GetLegendGraphic	non	partiellement	non
Export	fichier .QGis Image	projet uDig Image	fichier .GSF Image
Import	WFS shp/vrt (raster) PostGIS/GPS	WFS ArcGis/Oracle MySql/PostGIS	WFS/WCS/GML Mif/kml/kmz/shp/dxf Bing/Yahoo/OpenStreetMap
Impression	oui (bogues)	oui	oui

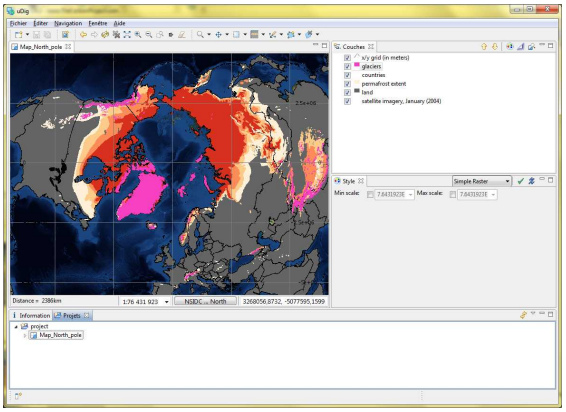
TABLE 2.2 – Comparatif des clients WMS Quantum GIS/uDig/Gaïa



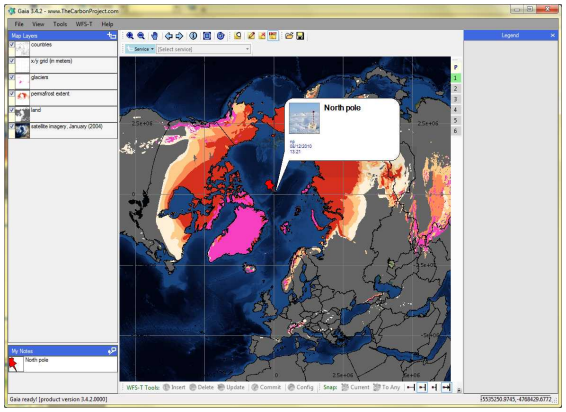
(a) GoogleEarth



(b) Quantum GIS



(c) uDig



(d) Gaïa

FIGURE 2.19 – Clients mono-postes avec le service *atlas north*

2.3 Services Web

Un service Web est un programme informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Plus concrètement, du point de vue d'un programmeur, il s'agit de faire appel à une méthode retournant ou non des informations à travers un réseau (internet ou intranet). L'utilisation des services Web doit répondre à trois problématiques :
– la publication du service (*publish*) ;
– la découverte du service (*find*) ;
– l'utilisation du service (*bind*).

De ces trois besoins découlent trois acteurs : le fournisseur de service (*service provider*), le client du service (*service requester*) et l'annuaire de services (*service broker*).

Le schéma de la figure 2.20, qui décrit le fonctionnement d'un service Web, est à rapprocher du schéma de la figure 2.8 qui décrit le modèle SDI de l'OGC.

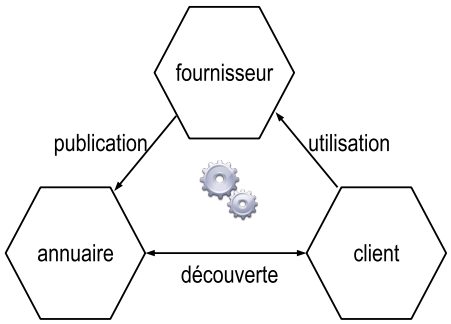


FIGURE 2.20 – Service Web

Ce schéma illustre bien un fait essentiel, le fournisseur de services ne connait pas ses clients *a priori*. Toutefois, cette vision des services Web doit être tempérée par le fait que si la normalisation des interactions entre le fournisseur et le client peut-être considérée comme établie, c'est loin d'être le cas en ce qui concerne les interactions impliquant l'annuaire, et, dans les faits, une solution à base de services Web se cantonne souvent à l'utilisation du service, la publication et la découverte étant remplacées par le fait que le client «connaisse déjà» le service. Dans la suite, l'exposé se focalise donc sur deux acteurs : le *client* et le *serveur* (= fournisseur), en principe indépendants, mais cependant liés de façon plus ou moins tacite par un contrat : l'API du serveur.

2.3.1 Application Programming Interface (API)

L'API d'un service permet l'interaction du service avec ses clients. C'est la description des fonctions, procédures et classes que manipulent le service. La plupart des standards décrivant l'API d'un service Web sont des formats basés sur le langage XML.

WSDL (Web Service Description Language)

La version WSDL 2.0 a été approuvé en 2007 en tant que recommandation du W3C [@Wikipedia, e]. WSDL sert à décrire :
– le protocole de communication (SOAP RPC ou SOAP orienté message) ;

– le format de messages requis pour communiquer avec ce service ;
– les méthodes (fonctions) que le client peut invoquer ;
– la localisation du service.

Une description WSDL est un document XML qui commence par la balise <definitions> et qui contient les balises suivantes :
– <binding> : définit le protocole à utiliser pour invoquer le service Web ;
– <port> : spécifie l'emplacement actuel du service ;
– <service> : décrit un ensemble de points finaux du réseau.

Les opérations et les messages sont décrits de façon abstraite, mais cette description renvoie à des protocoles réseaux et à des formats de messages concrets. Le protocole de communication associé à WSDL est le protocole SOAP (Simple Access Object Protocol), un protocole RPC [25] qui permet la transmission de messages entre objets distants. Un objet peut, grâce à SOAP, invoquer des méthodes d'objets physiquement situés sur un autre serveur. Le transfert se fait le plus souvent à l'aide du protocole HTTP, mais peut également se faire par un autre protocole, comme SMTP. Un message SOAP est constitué d'une enveloppe, qui en plus des informations nécessaires au transport du message, contient un en-tête (*header*) et un corps (*body*). L'en-tête contient des définitions et des règles relatives aux types de données (classe), le corps contient, lui, les valeurs transportées (instances).

XSD et DTD

Ces deux formats ne décrivent pas véritablement l'interface du service, mais sont plutôt des grammaires (structures et règles) décrivant les types des objets impliqués dans l'utilisation du service. Dans le cadre de services normalisés ces documents suffisent. Ces documents décrivent la structure des données échangées, ainsi que des règles. On peut imbriquer des documents les uns dans les autres afin de ne pas les rendre trop lourds. Ainsi le fichier A (.dtd ou .xsd) décrira un objet de type A, dans cette description, il sera fait mention d'un objet de type B (par exemple, A contient une liste vide ou non d'objets de type B), sans le décrire, mais en faisant référence à un fichier décrivant le type B.

Le bénéfice à retirer de l'utilisation de ce type de fichier est l'interopérabilité : le client et le serveur peuvent partager une connaissance du domaine métier à travers ces fichiers. On pourra, par exemple, programmer le serveur et le client avec deux langages de programmation différents : les objets échangés ne dépendront pas de ces deux langages. De plus, il existe des outils permettant de générer automatiquement, à partir de ces fichiers, les classes correspondantes pour un langage donné, ce qui facilite le travail des programmeurs et évite aussi des erreurs. Ces fichiers ne sont donc pas seulement un moyen d'échange entre le client et le serveur, mais aussi une spécification du modèle de données utilisé.

Dans le fichier XSD suivant, le type *personne* est un type complexe. Un objet de ce type est défini par un nom, un prénom, un établissement et un numéro de téléphone.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="personne">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="nom" type="xs:string" />
        <xs:element name="prenom" type="xs:string" />
        <xs:element name="date_naissance" type="xs:date" />
        <xs:element name="etablissement" type="xs:string" />
        <xs:element name="num_tel" type="xs:string" />
      </xs:sequence>
```

25. Remote Procedure Call : appel de procédure à distance.

```
      </xs:complexType>
    </xs:element>
</xs:schema>
```

Suivi d'un fichier XML valide :

```
<?xml version="1.0" encoding="UTF-8"?>
<personne xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:noNamespaceSchemaLocation="personne.xsd">
  <nom>DUPONT</nom>
  <prenom>Jean</prenom>
  <date_naissance>06/10/1975</date_naissance>
  <etablissement>CNAM</etablissement>
  <num_tel>764704140</num_tel>
</personne>
```

2.3.2 Architectures

Il existe deux approches des services web : REST et WS-*. On parle aussi de Ressource Oriented Architecture (ROA)[26] et Service Oriented Architecture (SOA)[27]. Si la finalité reste la même (un service obtenu via le Web), le besoin est différent. Dans un cas le client souhaite accéder à une ressource (par exemple, une carte), dans l'autre à un traitement (par exemple, calcul d'un itinéraire). Toutefois, distinguer les deux approches uniquement par le biais ROA/SOA, ou par la proximité du service avec l'utilisateur est périlleux, car la frontière est floue, comme on peut le voir sur la figure 2.21. C'est par rapport aux technologies impliquées que la distinction est plus nette. REST est basé sur le protocole HTTP, qui possède un vocabulaire riche et standardisé (URI, Internet media type ou MIME, response codes, *etc.*). De plus, comme le web (W3C) est basé sur ce protocole, l'utilisation de REST implique la gestion de *proxy*, de cache ou de sécurité. WS-* est lui basé sur les protocoles SOAP pour les échanges de messages, et WSDL pour la description du service. SOAP est plus lourd et moins riche que le protocole HTTP, par contre WSDL permet de redéfinir tout un vocabulaire lié à un contexte métier. Les performances ou la sécurité peuvent être un problème.

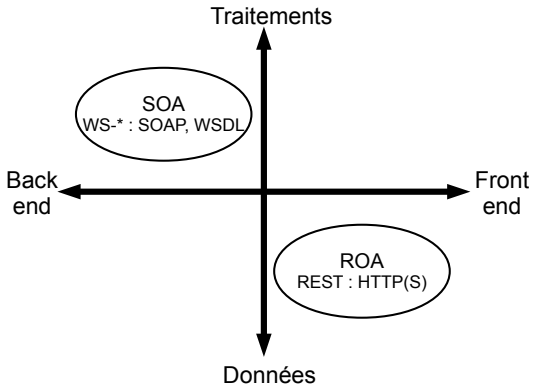


FIGURE 2.21 – Approches ROA / SOA

Les deux solutions ont leurs promoteurs et leurs détracteurs, et comme souvent il n'y a pas de choix juste ou faux, mais simplement un bon ou un mauvais choix, qui doit être fait en fonction des objectifs et des contraintes [Fielding, 2000] du contexte.

26. Architecture orientée données.
27. Architecture orientée traitements.

Ce ne sont bien sûr pas les seuls critères, mais voici quelques exemples de contextes pouvant guider ce choix [Tyagi, 2006]. On privilégiera l'utilisation de REST dans les cas suivants :
– le serveur est sans état ;
– une infrastructure de cache est nécessaire à cause de contraintes de performance. On peut utiliser SOAP avec un cache, mais la lourdeur inhérente au protocole est accrue, et des incohérences peuvent apparaître lorsque des changements d'états ont lieu au cours de l'interaction client/serveur ;
– la bande passante est un élément critique. RESTful est particulièrement indiqué pour des clients s'exécutant sur des PDA ou des *smartphones*.

Une conception basée sur SOAP sera plus appropriée dans les cas suivants :
– besoin d'un contrat formel décrivant l'interface proposée par le service Web. WSDL décrira les détails concernant les messages et les opérations ;
– spécifications nécessitant un vocabulaire métier commun relatif aux échanges entre le client et le serveur. Les transactions se font sur un mode conversationnel avec des informations contextuelles ce qui implique le maintien d'un état sur le serveur.

Le tableau suivant résume les différences entre les deux approches.

SOA WS-*	**ROA REST**
Complexe	Simple
URL non-significative	URL riche
Conservation d'état	RESTful (sans état)
Bus applicatif (ESB)	Réseau
SOAP, WSDL	HTTP (GET, PUT, POST, DELETE)

TABLE 2.3 – Approches ROA / SOA

2.3.3 Solutions techniques avec JAVA

On peut envisager deux approches pour créer un service Web :
– approche descendante (Top Down) : à partir de fichiers de spécifications (wsdl ou dtd/xsd), les outils de développement permettent de générer automatiquement les classes nécessaires à l'implémentation du service ;
– approche montante (Bottom Up) : à partir d'un code existant, on peut générer les couches de transport permettant de réaliser le service, et générer aussi les fichiers associés au service (fichier wsdl par exemple).

L'approche à utiliser dépend de l'existant et des contraintes du projet. Dans le cas d'un service Web géographique, on pourra choisir une approche Top Down à partir des fichiers dtd/xsd correspondant aux normes de l'OGC. Dans le cas de web service Bottom Up (Java vers XML, comme lorsque l'on publie le code d'un existant sous forme d'un service Web), il faudra prendre garde aux types des objets manipulés dans les classes [28].

Les outils présentés par la suite ont un point commun : ils permettent au développeur de faire abstraction de la couche de transport, pour se consacrer à la logique métier du service qu'il développe.

28. Voir correspondance entre XML Schema Type et Java Data Type : http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html#bnazs.

Servlet HTTP

Les Servlets sont des classes Java implémentant des classes et des interfaces provenant des paquetages suivants :
– javax.servlet, un paquetage générique indépendant du protocole utilisé ;
– javax.servlet.http, un paquetage spécifique au protocole HTTP 1.0.

Le paquetage *javax.servlet* est fourni dans le JSDK (Java Servlet Development Kit, c'est-à-dire l'API Java Servlet) de Sun [29]. Il faut également importer le paquetage java.io pour gérer les exceptions.

Ainsi, toutes les Servlets implémentent directement ou indirectement l'interface Servlet, en dérivant une classe qui l'implémente, généralement la classe HttpServlet, elle-même issue de GenericServlet.

La figure 2.22 décrit ces héritages. L'interface définit le contrat d'une Servlet, essentiellement rendre un service à l'aide d'un paramètre requête et d'un paramètre réponse. La classe abstraite met en place les attributs principaux d'une Servlet. La classe HttpServlet implémente la méthode *service()*, en en répartissant la logique entre des méthodes abstraites doGet(), doPost(), doPut() et doDelete() selon le type de requête HTTP [30].

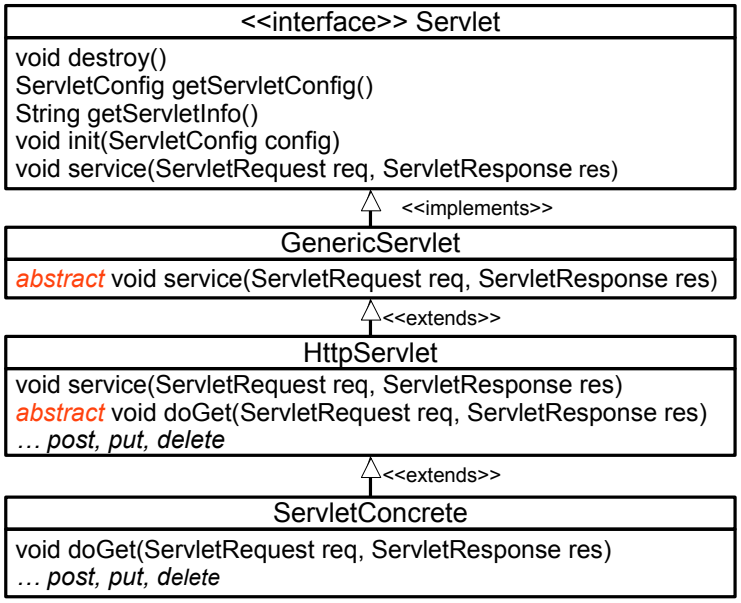


FIGURE 2.22 – Héritages mis en jeu dans l'implémentation d'une Servlet

Le cycle de vie d'une Servlet est assuré par le conteneur de Servlets. Ainsi, afin d'être à même de fournir la requête à la Servlet, récupérer la réponse ou bien tout simplement démarrer/arrêter la Servlet, cette dernière doit possèder une interface (un ensemble de méthodes prédéfinies) déterminée par le JSDK afin de suivre le cycle de vie suivant :
– le serveur crée un pool de *threads* (fil d'exécution) auxquels il va pouvoir affecter chaque requête ;
– La Servlet est chargée au démarrage du serveur ou lors de la première requête ;

29. Disponible sur http://java.sun.com/products/servlets.
30. Les quatre types de requête HTTP sont :
    – GET : obtenir une ressource ;
    – POST : ajouter une ressource ;
    – PUT : modifier une ressource ;
    – DELETE : supprimer une ressource.

– la Servlet est instanciée par le serveur ;
– la méthode *init()* est invoquée par le conteneur ;
– lors de la première requête, le conteneur crée les objets Request et Response spécifiques à la requête ;
– la méthode *service()* est appelée à chaque requête dans un nouveau *thread*. Les objets Request et Response lui sont passés comme paramètres ;
– grâce à l'objet Request, la méthode *service()* va pouvoir analyser les informations en provenance du client ;
– grâce à l'objet Response, la méthode *service()* va fournir une réponse au client ;
– la méthode *destroy()* est appelée lors du déchargement de la Servlet, c'est-à-dire lorsqu'elle n'est plus requise par le serveur.

Lorsqu'une Servlet est appelée par un client, la méthode *service()* est exécutée. Celle-ci est le principal point d'entrée de toute Servlet et accepte deux objets en paramètres :
– l'objet ServletRequest encapsulant la requête du client, c'est-à-dire qu'il contient l'ensemble des paramètres passés à la Servlet (informations sur l'environnement du client, *cookies* du client, URL demandée, ...) ;
– l'objet ServletResponse permettant de renvoyer une réponse au client (envoyer des informations au navigateur). Il est ainsi possible de créer des en-têtes HTTP (headers), d'envoyer des *cookies* au navigateur du client...

Afin de développer une Servlet fonctionnant avec le protocole HTTP, il «suffit» de créer une classe étendant HttpServletRequest (qui implémente elle-même l'interface Servlet. La classe HttpServletRequest permet de fournir une implémentation de l'interface Servlet spécifique à HTTP. La classe HttpServletRequest surcharge la méthode service en lisant la méthode HTTP utilisée par le client, puis en redirigeant la requête vers une méthode appropriée. Le travail du développeur de la servlet consiste finalement à surcharger les méthodes abstraites de la classe HttpServlet correspondant aux possibilités du protocole HTTP : GET,POST,PUT et DELETE avec la logique métier du service que la servlet doit rendre.

L'extrait de code suivant est un exemple de surcharge de la méthode doGet, qui correspond à la requête HTTP GET :

```
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletConcrete extends HttpServlet {
  @override// en anglais surcharger
  public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException {
        // lecture de la requête
        // traitements
        // envoi de la réponse
  }
}
```

A l'intérieur de la méthode *doGet()* (*doPost()*, *doDelete()* ou *doPut()*), la requête de l'utilisateur est passée en paramètre sous forme d'objet HttpServletRequest. La méthode extrait de cet objet les couples nom / valeur de chaque paramètre de la requête. Le résultat du traitement sera stocké dans l'objet HttpServletResponse.

Les deux tableaux suivants résument les possibilités de ces deux types d'objet.

Méthode	Description
String getMethod()	Retourne la méthode HTTP utilisée par le client
String getHeader(String Key)	Retourne la valeur de l'attribut Key de l'en-tête
String getRemoteHost()	Retourne le nom de domaine du client
String getRemoteAddr()	Retourne l'adresse IP du client
String getParameter(String Key)	Retourne la valeur du paramètre Key (clé) d'un formulaire. Lorsque plusieurs valeurs sont présentes, la première est retournée
String[] getParameterValues(String Key)	Retourne les valeurs correspondant au paramètre Key (clé) d'un formulaire, c'est-à-dire dans le cas d'une sélection multiple (cases à cocher, listes à choix multiples) les valeurs de toutes les entités sélectionnées
Enumeration getParameterNames()	Retourne un objet Enumeration contenant la liste des noms des paramètres passés à la requête
String getServerName()	Retourne le nom du serveur
String getServerPort()	Retourne le numéro de port du serveur

TABLE 2.4 – Méthodes de l'objet HttpServletRequest

Méthode	Description
String setStatus(int StatusCode)	Définit le code de retour de la réponse
void setHeader(String Nom, String Valeur)	Définit une paire clé/valeur dans les en-têtes
void setContentType(String type)	Définit le type MIME de la réponse HTTP, c'est-à-dire le type de données envoyées au navigateur
void setContentLength(int len)	Définit la taille de la réponse
PrintWriter getWriter()	Retourne un objet PrintWriter permettant d'envoyer du texte au navigateur client. Il se charge de convertir au format approprié les caractères Unicode utilisés par Java
ServletOutputStream getOutputStream()	Définit un flot de données à envoyer au client, par l'intermédiaire d'un objet ServletOutputStream, dérivé de la classe java.io.OutputStream
void sendredirect(String location)	Permet de rediriger le client vers l'URL location

TABLE 2.5 – Méthodes de l'objet HttpServletResponse

JAX-WS

JAX-WS [31] permet d'assurer l'indépendance du protocole SOAP et du protocole HTPP, qui n'est utilisé que pour le transport. Le développement de services Web avec JAX-WS est basé sur un système d'annotation des POJO [32] décrivant les fonctionnalités de base liées au service Web. Cette technique ne nécessite aucun fichier de configuration. On retrouve les deux types d'approche (Bottom/Up et Top/Down) à partir du format WSDL.

31. Java API for XML Web Services.
32. POJO : Plain Old Java Object (bon vieil objet JAVA), ce sont des classes JAVA simples (notamment pas d'héritage).

Dans son travail, le développeur du service Web, que ce soit lors de la réalisation du serveur ou de celle du client, ne manipule que du code Java, le code XML étant caché par JAXB [33], une API d'Oracle capable de transformer des schémas (xsd) en classes Java.

La figure 2.23 illustre ce principe : les messages WSDL ou SOAP sont transmis à JAX-WS, qui va faire correspondre les éléments XML à des classes JAVA.

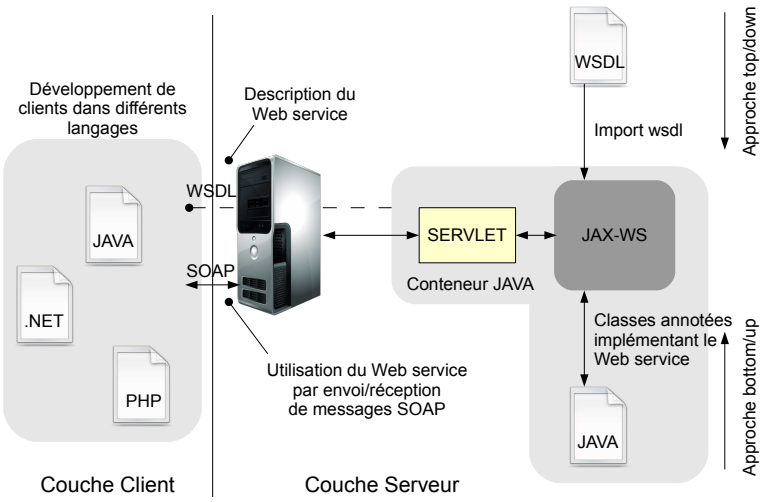


FIGURE 2.23 – JAX-WS : vue générale

Bottom/Up L'approche *Bottom/Up* consiste à annoter le code Java. Le code suivant, correspondant à un service affichant le message «Hello world», éventuellement accompagné du nom de la personne saluée.

```
package helloservice.endpoint;

public class Hello {
    private String message = new String("Hello, ");

    public void Hello() {}

    public String sayHello(String name) {
        return message + name ;
    }
}
```

Il est annoté de la façon suivante :

```
package helloservice.endpoint;

import javax.jws.WebService;

@WebService  // la classe Hello est annotée en tant que service Web
public class Hello {
    private String message = new String("Hello, ");
```

33. Java Architecture for XML Binding.

```
    public void Hello() {}

    @WebMethod  // la méthode sayHello est une méthode web
    public String sayHello(String name) {
        return message + name ;
    }
}
```

Il ne reste plus qu'à le publier pour que la transformation de la classe Hello en service Web soit complète. Cela peut-être fait par exemple à l'aide de la méthode statique *publish()* de la classe EndPoint :

```
import javax.xml.ws.Endpoint;

public static void main(String[] args) {
      Endpoint.publish(
         "http://localhost:8080/WebServiceExample/hello",
         //  adresse du service Web
         new Hello());
         //  type du web service
}
```

Après avoir compilé le service Web, il faut utiliser l'outil wsgen [34] de JAX-WS, qui génère les artefacts JAX-WS nécessaires à son exécution ainsi que le fichier WSDL qui lui est associé [35].

```
wsgen -cp helloService.Hello -keep -wsdl
```

Top/Down Dans le cas de l'approche *Top/Down*, le développement démarre en partant du ficher wsdl. De ce fichier, avec l'outil wsimport, on génère les classes liées à JAXB et des interfaces WS. C'est ce que l'on appelle le squelette du service. Par exemple à partir du fichier WSDL suivant (le fichier n'est pas complet), qui décrit un service correspondant à un carnet d'adresses :

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
  <types>...Person...</types>
  <portType name="Notebook">
    <operation name="addPersonWithComplexType">
      <input message="tns:addPersonWithComplexType"/>
      <output message="tns:addPersonWithComplexTypeResponse"/>
    </operation>
    <operation name="getPersonByName">
      <input message="tns:getPersonByName"/>
      <output message="tns:getPersonByNameResponse"/>
    </operation>
    <operation name="getPersons">
      <input message="tns:getPersons"/>
      <output message="tns:getPersonsResponse"/>
    </operation>
    <operation name="addPersonWithSimpleType">
      <input message="tns:addPersonWithSimpleType"/>
    </operation>
  </portType>
  <binding name="NoteBookPortBinding" type="tns:Notebook">...</binding>
  <service name="Notebook">...</service>
</definitions>
```

34. À partir du JRE 6, ce n'est plus nécessaire car la génération est implicite lors de l'exécution du service.
35. Dans l'exemple, ce fichier sera accessible à l'adresse suivante : http://localhost:8080/WebServiceExample/hello?wsdl.

L'outil génèrera automatiquement les classes et les interfaces (la figure 2.24 montre l'arborescence de ces classes dans un environnement de développement) correspondant aux définitions du fichier :
– les types de données (définies entre les balises<types>, comme Person, qui représente un contact) ;
– les opérations (addPersonWithComplexType,getPersonByName,...) ;
– les messages (tns :addPersonWithComplexType,tns :addPersonWithComplexTypeResponse,...) ;
– le service (NoteBook).

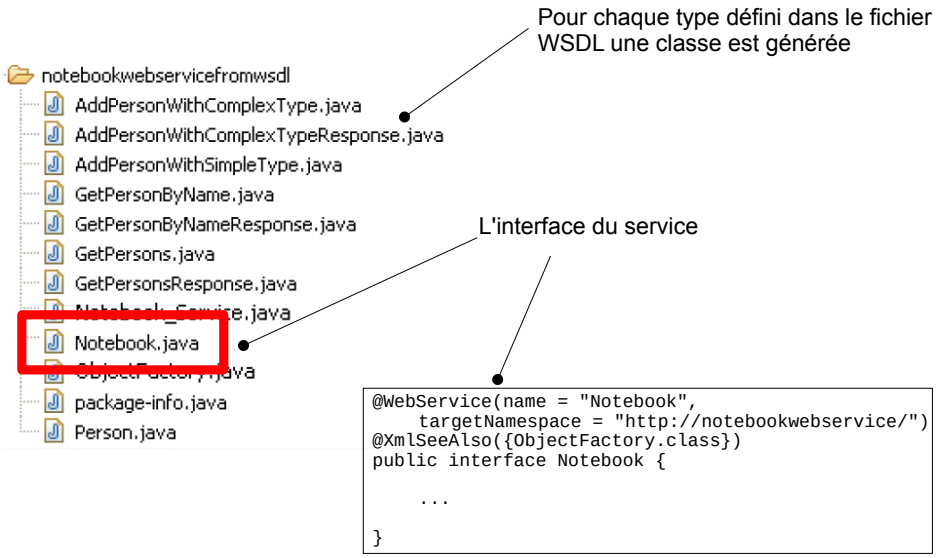


FIGURE 2.24 – JAX-WS : classes générées avec wsimport - côté serveur

Le travail de développement consistera à réaliser l'implémentation concrète de l'interface du service correspondant à son interface avec la logique de sa solution :

```
@WebService(endpointInterface = "notebookwebservicefromwsdl.Notebook")
public class NotebookServiceImpl {
  public boolean addPersonWithComplexType(Person newPerson) {
...
    // logique de la solution
...
    return true;
  }
...
}
```

On remarque que la classe est un POJO : elle n'est pas définie comme implémentant l'interface Notebook, c'est l'annotation endpointInterface = "notebookwebservicefromwsdl.Notebook" qui se charge d'assurer cette relation.

Client Le développement du client suit une procédure similaire à l'approche Top/Down, en partant du fichier wsdl (récupéré via une URL ou via un fichier physique). L'outil wsimport permet d'obtenir le même squelette de classes que pour le serveur. Le développement sera la réalisation des implémen-

tations des interfaces correspondant au service Web : PortType et Service, qui permettent l'appel à distance des opérations proposées par le service.

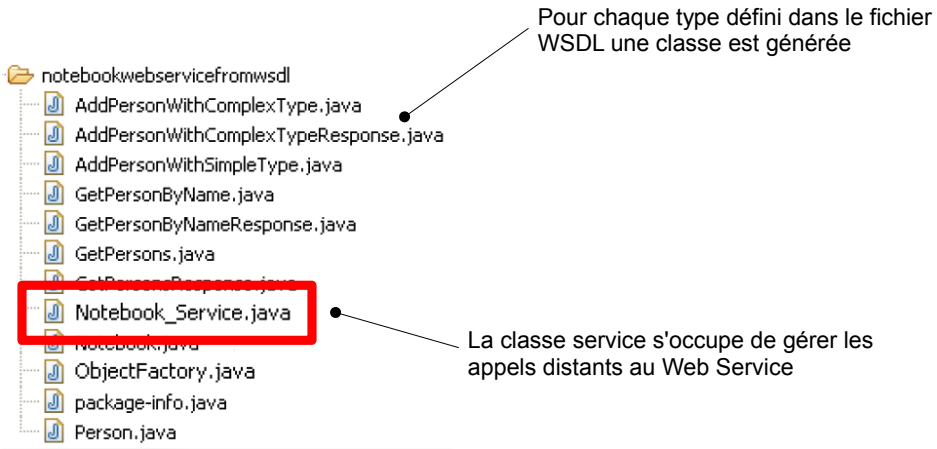


FIGURE 2.25 – JAX-WS : classes générées avec wsimport - côté client

Le code du client utilisera le service comme si c'était une classe «normale». Le fait que la logique s'exécute sur un serveur distant devient transparent pour le client, l'implémentation du PortType servant de *proxy*[36].

```
public class NotebookClient {
  public static void main(String[] args) {
    Notebook_Service notebook_Service = new Notebook_Service();
    Notebook noteBookPort = notebook_Service.getNoteBookPort();

    //logique du client utilisant des appels aux opérations du service
    //via l'objet noteBookPort

  }
}
```

36. Patron de conception *proxy* [Gamma *et al.*, 1994]

Axis 2

La solution Axis2 ressemble à la solution JAX-WS dans son principe. Là aussi il s'agit de *découpler la couche transport de la couche logique* du service Web. Toutefois les mécanismes utilisés sont très différents. L'architecture Axis2 repose sur plusieurs modules élémentaires (modules CORE). Pour Deepal Jayasinghe, Axis2 est un pur moteur de traitement SOAP [Jayasinghe, 2008]. Tous les messages échangés sont d'abord transformés en messages SOAP. Les modules CORE sont au nombre de six :

XML Processing Model : utilise le modèle d'objet AXIOM (Axis Object Model) qui permet la correspondance entre classes de données et classes de services avec des fichiers SOAP. Le modèle respecte les normes DOM du W3C ;

Information Model : ce modèle est composé de deux hiérarchies, une hiérarchie de description, qui contient les informations statiques relatives aux services déployés, et une hiérarchie de contexte qui contient les informations qui changent dynamiquement à mesure que les échanges ont lieu. La figure 2.26 expose le découpage en couches du modèle d'informations d'Axis2 : la configuration est composée de groupe de services, un service effectue des opérations à l'aide de messages ;

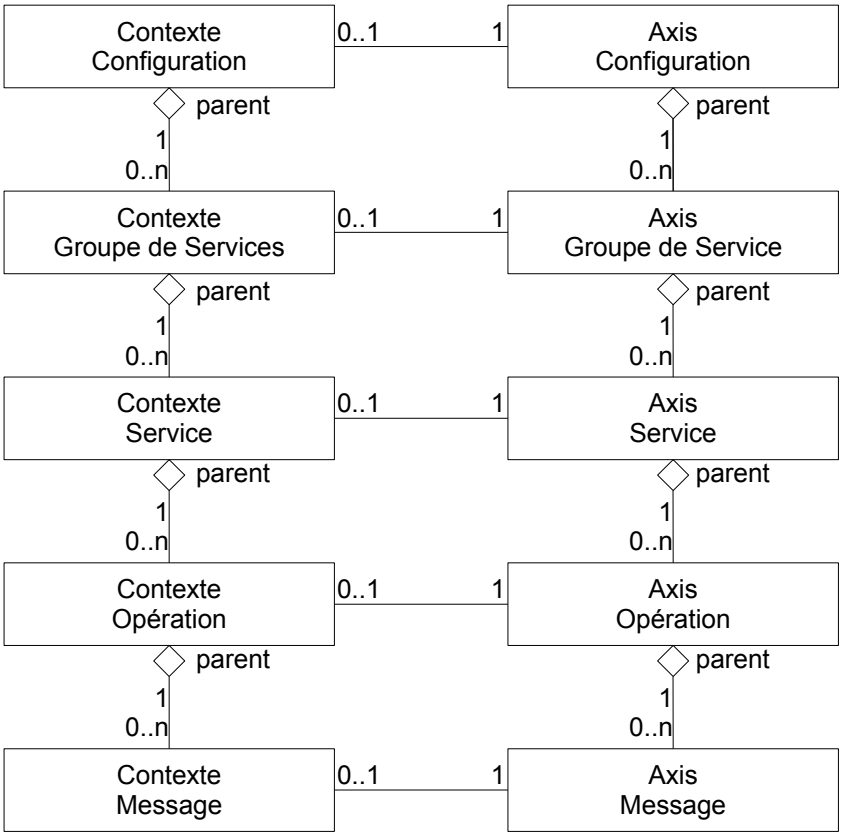


FIGURE 2.26 – Axis2 : Information Model
Source : [Jayasinghe, 2008]

SOAP Processing Model : ce module enchaîne les phases de traitement d'un fichier SOAP entrant. Le nombre de phases peut varier d'un service à l'autre. À travers un mécanisme de *handlers*, le message est intercepté et un traitement peut-être exécuté sur le message avant qu'il n'arrive dans la couche responsable de la logique du service. Ce traitement peut d'ailleurs se terminer

par une interruption de la transaction : au cours d'une des phases précédant la réception du message (voir la figure 2.27), le *handler* de la phase peut interrompre la transaction ;

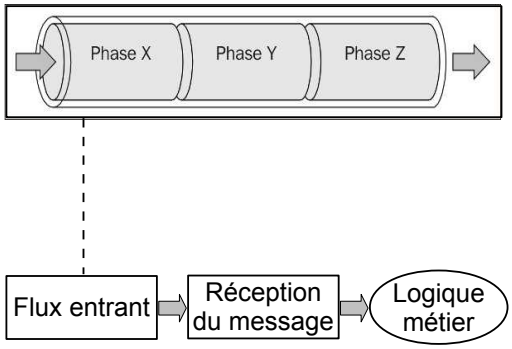


FIGURE 2.27 – Axis2 : SOAP Processing Model
Source : [Jayasinghe, 2008]

Deployment Model : ce modèle contient des informations relatives au déploiement du service. Axis2 est très étroitement intégré aux serveurs Apache (Tomcat par exemple), et permet même de renouveler l'implémentation d'un service en cours d'exécution sans l'interrompre (*hot update*) ;

Client API : l'invocation d'un service Web doit se faire de façon asynchrone, et surtout non-bloquante. L'API client permet cela. La classe ServiceClient possède dans son API quatre méthodes pour invoquer un service. La méthode *sendRobust* envoie une requête XML, et lève une exception en cas de problème. La méthode *fireAndForget* envoie une requête XML sans même lever une exception en cas de problème. La méthode *sendReceive*, la plus utilisée, invoque le service et reçoit une valeur de retour. La méthode *sendReceiveNonBlocking* invoque le service et reçoit la valeur de retour sous forme d'événement (*callback*) ;

Transports : il y a dans Axis2 deux transporteurs, l'émetteur et le récepteur. Le reste de l'architecture est complètement indépendant de la couche de transport. Axis2 supporte les protocoles HTTP/HTTPS, TCP, SMTP, JMS et XMPP.

Les deux autres modules concernent la génération de code et la liaison de données (*data binding*). La génération de code utilise des templates XSL ce qui permet de générer le code dans différents langages de programmation. Le code peut être généré en utilisant l'approche *TopDown* ou *BottomUp*, comme avec JAX-WS. Le générateur de code produit un squelette pour le service et un *proxy* pour le client. La différence avec JAX-WS est qu'Axis2 est complètement intégré à l'environnement de développement Eclipse avec le plug-in Web Tools Platform (WTP), ce qui facilite grandement le travail du développeur. Un projet «Web Service» est équivalent à un projet classique. Il existe de nombreux didacticiels permettant de mettre rapidement en place un service Web basé sur Axis2. De façon analogue le *data binding* a été sorti du cœur afin de permettre l'utilisation de différents environnements de développement :

– ADB : Axis2 Data Binding ;
– XMLBeans ;
– JaxMe ;
– JibX.

Le principe du *data binding* est basé sur une correspondance QName [37]/nom de classe.

37. De l'anglais Qualified Name : ce sont les noms correspondant aux définitions des éléments/attributs d'un fichier XML.

2.4 Conclusion

Le contexte présenté dans le premier chapitre amenait quelques interrogations et cet état de l'art a permis d'y répondre :

1. *Le modèle des SDI est-il une réponse à la problématique de la diffusion de l'information géographique ?*

Si l'on se base sur les réalisations implémentant ce modèle et leur succès (le géoportail de l'IGN est un modèle du genre) la réponse est oui. Toutefois le succès d'une SDI dépend de quelques éléments essentiels : la coopération entre les partenaires aux niveaux des organisations, et l'interopérabilité, qui découle partiellement de cette coopération, mais aussi de la conception d'interfaces séduisantes et fonctionnelles.

2. *Les services Web de l'OGC permettent-ils la diffusion de l'information géographique ?*

L'éventail des services Web de l'OGC encadre la mise en place d'une SDI, depuis la production de l'information (SWE) jusqu'à la carte interactive du géoportail (WMS). Ce n'est pas étonnant car l'OGC rassemble les acteurs principaux du monde de la géomatique et du monde du Web. La cohérence de l'ensemble des spécifications de l'OGC explique la décision d'ESPON d'inciter ses partenaires à les utiliser.

L'objection que l'on peut faire à cet ensemble de spécifications est sa complexité, mais elle reflète sa richesse d'usages. Le service SLD-WMS est très riche et donc très difficile à mettre en place, mais le WMS reste assez simple, ce qui explique sans doute son succès.

Cependant le banc d'essai des clients WMS a mis en lumière un point important, la légende des cartes n'est pas disponible dans ces clients [38], et, par rapport à l'importance des légendes dans le contexte HyperAtlas, cela pose un problème.

3. *Le langage JAVA est-il adapté au développement des services Web ?*

JAVA propose de nombreuses solutions permettant de faire abstraction de la couche de transport, que ce soit dans le cadre d'un service REST, ou dans celui d'un service WSDL/SOAP. Pour finir, cette étude nous a permis de nous familiariser avec le langage et avec les protocoles de transport du Web.

38. Le seul qui fasse exception est ArcExplorer Web, mais il semble ne pas être capable d'utiliser les services WMS qu'il ne «connait» pas : on peut se demander s'il utilise vraiment la spécification WMS.

Chapitre 3

Proposition

Le but n'est pas le but, c'est la voie.

Lao-Tseu

Notre proposition a été établie à partir du besoin qu'avait exprimé notre maître de stage au sein de l'équipe STEAMER du Laboratoire d'Informatique de Grenoble : *rendre les fonctionnalités d'HyperAtlas disponibles à travers un ou plusieurs services Web basés sur les spécifications de l'OGC*.

Comme cela a été montré dans les chapitres précédents, cette évolution vise à créer un certain niveau d'interopérabilité avec les services pré-existants, et donc constituerait un atout pour l'intégration d'HyperAtlas (cartes et calculs) dans des SDI, existantes ou à venir, comme, par exemple, dans le cadre du projet ESPON DataBase 2013 [@ESPON, b].

La première partie de ce chapitre expose les principaux choix de conception que nous avons faits au vu des besoins, et en fonction de l'analyse du logiciel HyperAtlas et de l'état de l'art menés lors de la préparation de cette proposition.

La seconde partie présente la méthode de travail que nous avons adoptée, et un cahier des charges simplifié. Les choix effectués dans cette partie sont basés sur notre expérience du développement informatique et sur les contraintes imposées par un stage Ingénieur CNAM : un seul développeur et un délai de 9 mois.

3.1 Conception

3.1.1 Choix de la spécification WMS

L'analyse de l'existant du logiciel HyperAtlas [1] et l' étude des standards de l'OGC [2] nous ont conduit à choisir la spécification WMS pour diffuser les cartes et les calculs d'HyperAtlas avec un service Web de l'OGC.

1. Voir 1.2 HyperCarte, page 2.
2. Voir 2.2.6 Spécification WMS, page 29.

Les méthodes de la spécification WMS recouvrent l'essentiel des fonctionnalités du logiciel HyperAtlas. Le tableau 3.1 et la figure 3.1 établissent cette adéquation.

Fonctionnalité d'HyperAtlas	**Spécification WMS**
affichage des différentes cartes, accessibles via des onglets	paramètre LAYERS de la requête GetMap
sélection des paramètres (aire d'étude, maillage, indicateurs, *etc.*)	paramètres VENDORSPECIFIC
affichage des résultats des différents calculs pour une unité territoriale, en la pointant à l'écran	requête GetFeatureInfo
affichage de la légende à gauche dans l'onglet	requête GetLegendGraphic
zoom et déplacement sur la carte	paramètre BBOX de la requête GetMap
modifications des options pour une carte (type de progression, couleurs, *etc.*)	paramètres VENDORSPECIFIC

TABLE 3.1 – Correspondances entre fonctionnalités d'HyperAtlas et de WMS

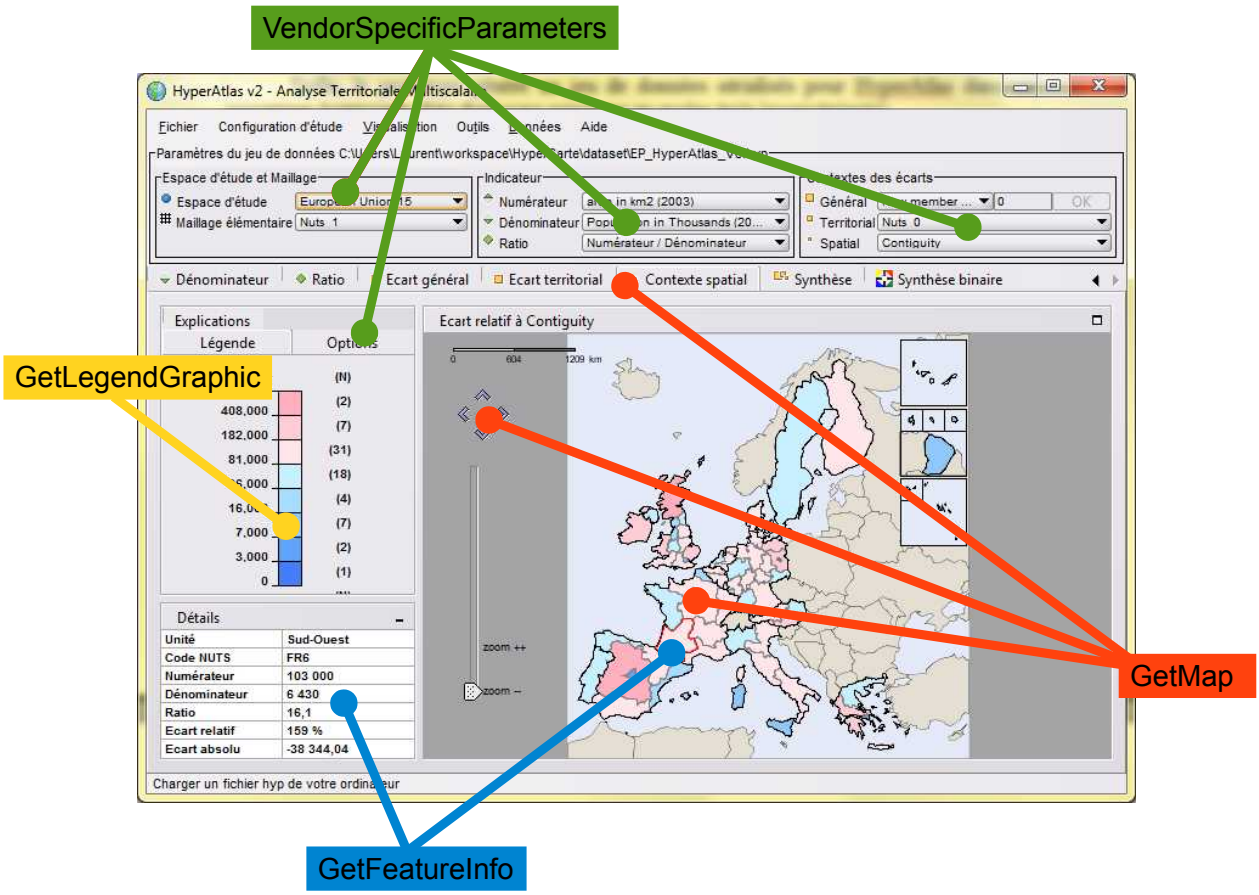


FIGURE 3.1 – Éléments WMS de l'interface HyperAtlas

3.1.2 Choix de la solution pour le service Web

Le second choix à effectuer fut celui de la solution technique Java.

La solution la plus séduisante lorsque l'on souhaite créer un service Web à partir d'un code existant est d'utiliser une approche montante [3], et de générer à partir de ce code la couche de transport. Toutefois cette solution n'a pas été possible. Pour pouvoir utiliser cette approche il aurait fallu n'utiliser que la logique métier de l'application HyperAtlas, mais elle est très fortement couplée à la logique de présentation, et cet état de fait, combiné à l'absence de tests automatisés [4], rendait les remaniements de code risqués. Nous reviendrons sur la réutilisation du code existant dans le chapitre suivant.

L'approche restante était donc une approche descendante : partir des spécifications du standard WMS pour implémenter ensuite le service. Cette approche nous a amené à choisir pour la réalisation du service Web une architecture en couches :
– une couche de transport ;
– une couche service Web WMS ;
– une couche service Web HyperAtlas.

La première couche correspond au transport des échanges entre service et client, et elle a été déléguée à la classe HttpServlet du paquetage Java *javax.servlet.http* [5]. Certains services Web de l'OGC sont spécifiés pour SOAP, mais ils sont rares, et ce n'est pas le cas du WMS à ce jour.

La seconde couche correspond à une couche abstraite WMS (elle ne contient pas d'implémentation concrète de service WMS, hormis quelques classes utilisées pour les tests unitaires). Le rôle de cette couche est de gérer les aspects du service liés à la spécification WMS. Elle décompose un service WMS en interfaces correspondant aux différentes fonctionnalités de WMS (GetMap, *etc.*), comme le montre la figure 3.2.

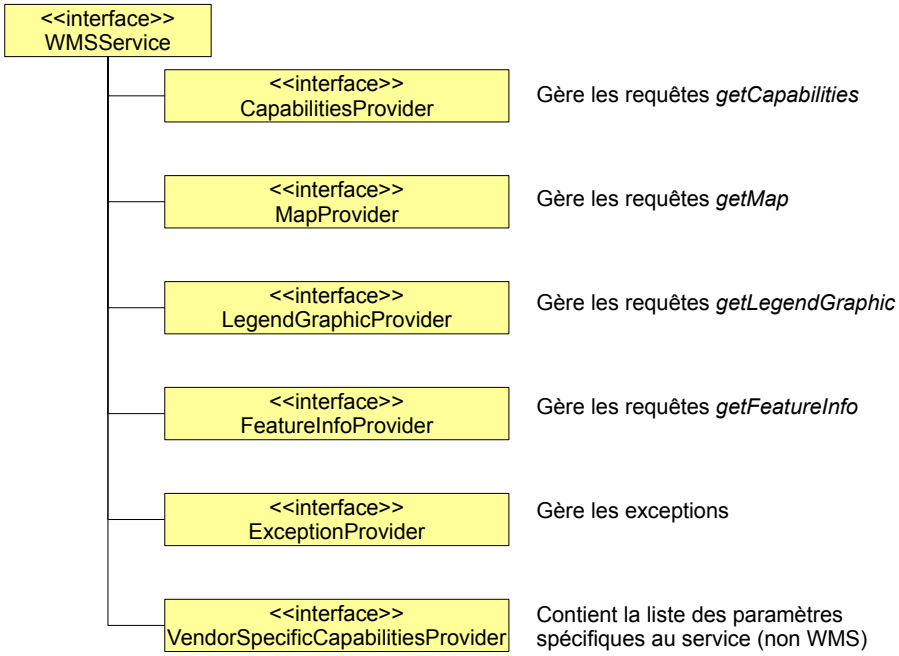


FIGURE 3.2 – Interfaces du service abstrait WMS

Son principe est à rapprocher du fonctionnement de la hiérarchie de classes des Servlets [6] : de même que la Servlet délègue les requêtes GET, POST, PUT et DELETE à une classe fille, la Servlet WM-

3. Voir 2.3.3 Solutions techniques avec JAVA, page 42.
4. Tests automatiques : exécutés par un programme, et non par un utilisateur.
5. Voir 2.3.3 Servlet HTTP, page 43.
6. Voir figure 2.22, page 43.

SServiceBase délègue le traitement des requêtes WMS à une classe fille, après avoir décrypté la requête HTTP.

Elle contient aussi des classes outils permettant, à partir des paramètres WMS, de dessiner une carte, ou de gérer les formats (image, *capabilities*, *etc.*).

Dans la suite de ce mémoire, cette bibliothèques de classes, qui correspond au paquetage *org.steamer.wms* du projet HyperCarte, sera appelée WMSBase.

La troisième couche correspond à l'implémentation concrète des différentes interfaces de la couche précédentes dans le contexte d'HyperAtlas. Le point d'entrée de cette couche est une implémentation concrète de la Servlet WMSServiceBase, la classe Service du paquetage *hypercarte.hyperatlas.wms*. La figure 3.3 décrit les héritages mis en jeu par la création du service Web HyperAtlas. Le nom de ce service est HyperAtlasWMS.

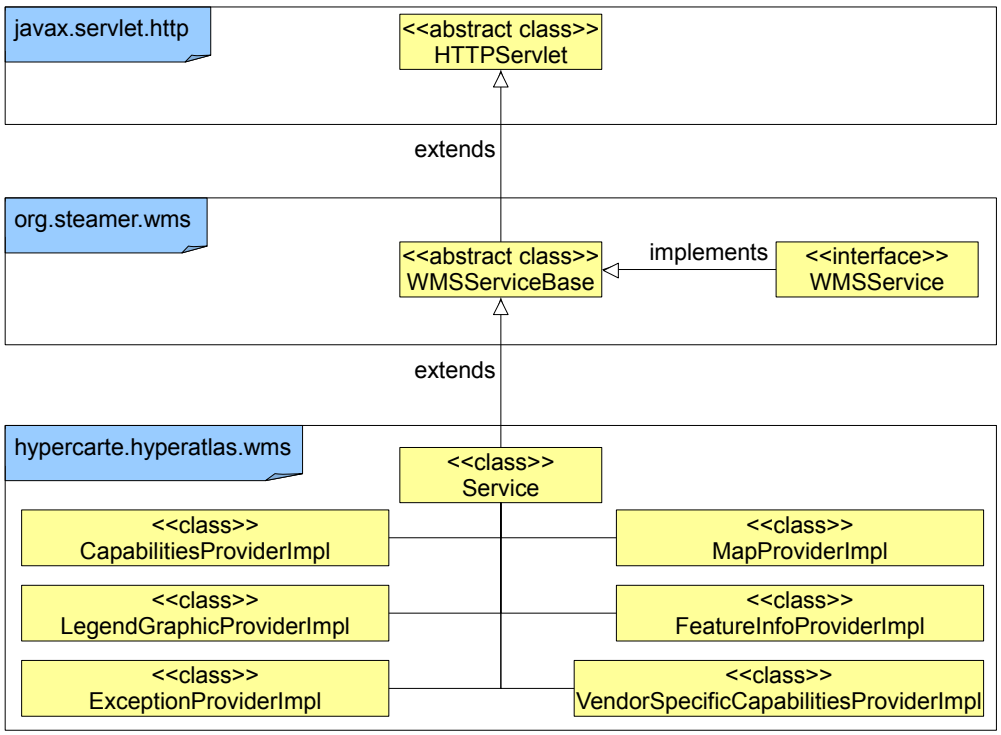


FIGURE 3.3 – Couches du service HyperAtlasWMS

Ce découpage est conforme aux principes fondamentaux de conception [Martin, 2000]. Par exemple :
– la couche abstraite WMS ne comporte aucune référence vers HyperAtlas et peut être réutilisée pour implémenter n'importe quel autre service WMS ;
– la couche HyperAtlasWMS s'appuie sur l'existant HyperAtlas, mais n'a introduit dans celui-ci aucun élément relatif à WMS.

Dans le cadre du développement, ce découpage a deux atouts principaux :
– possibilité d'un développement itératif qui a permis d'avoir un service très rapidement opérationnel, la norme OGC/WMS n'imposant que la requête *GetMap* comme fonctionnalité. Les autres requêtes ont été codées par la suite, en réutilisant les mêmes patrons de conception, voire en utilisant l'héritage des classes mises en place lors des itérations précédentes. Par exemple, la requête *GetFeatureInfo* contient tous les paramètres d'une requête *GetMap*, elle a donc été considérée comme une spécialisation de celle-ci ;
– chaque fonctionnalité peut être testée unitairement et automatiquement de façon très naturelle. Ce

point était d'autant plus critique que l'existant n'était pas testé automatiquement.

3.1.3 Utilisation du service HyperAtlasWMS

L'utilisation du service HyperAtlasWMS peut être résumée à un cas d'utilisation générique, l'émission d'une requête, son traitement sur le serveur, et l'envoi de la réponse, le contexte exact du cas d'utilisation relevant de la responsabilité du client. Cette séquence, qui comprend cinq étapes, valable pour tous les types de requêtes, est résumée par le schéma 3.4.

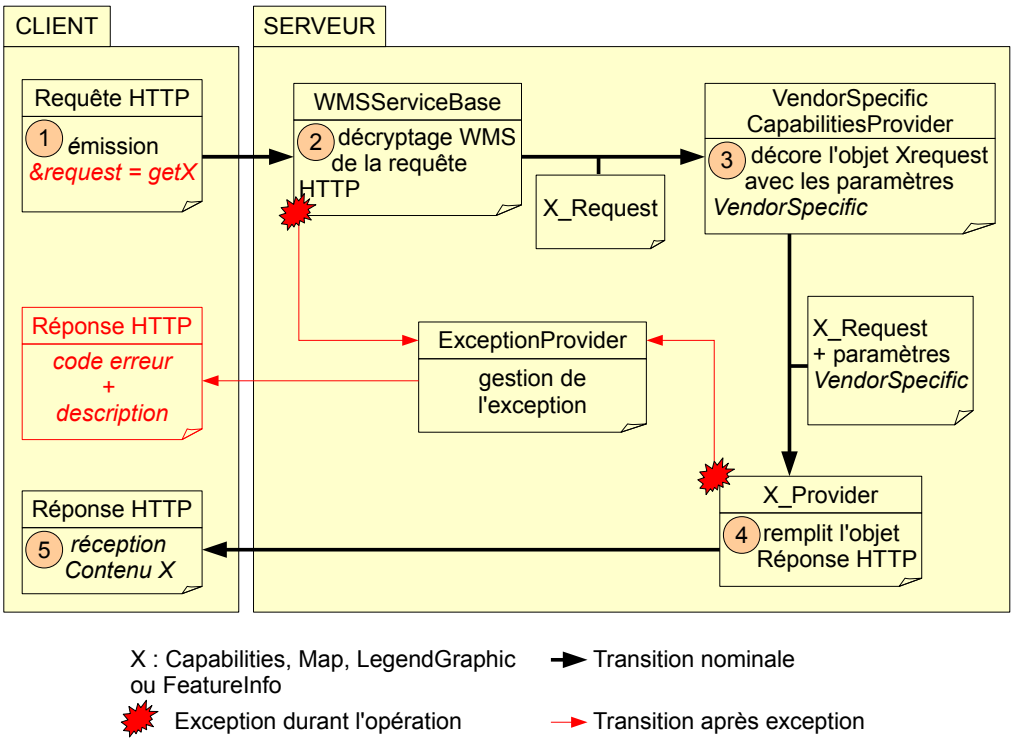


FIGURE 3.4 – Utilisation du service HyperAtlasWMS

étape 1 : émission d'une requête par le client.

étape 2 : réception de la requête par le serveur (le transport s'est déroulé normalement). La classe WMSServiceBase décortique la requête contenue dans l'objet HttpServletRequest [7]. Si cette requête contient tous les paramètres spécifiés par la norme OGC, WMSServiceBase crée un objet métier de la couche WMSBase correspondant à la requête [8]. Dans le cas contraire, l'implémentation de ExceptionProvider gèrera l'exception (ici, un paramètre manquant ou ne correspondant au type attendu, par exemple un caractère au lieu d'un entier).

étape 3 : l'objet requête est décoré [9] par VendorSpecificCapabilitiesProvider (par défaut, il ne se passe rien). Cette étape ne génère pas d'exception, ces paramètres étant optionnels.

étape 4 : la classe chargée du traitement de la requête [10] réalise le traitement approprié. Les exceptions sont gérées par ExceptionProvider.

étape 5 : le résultat du traitement est renvoyé vers le client via l'objet HttpServletResponse.

7. voir 2.3.3
8. MapRequest, FeatureInfoRequest ou LegendGraphicRequest, selon le type de requête.
9. Patron de conception Décorateur du GOF [Gamma *et al.*, 1994].
10. MapProvider, FeatureInfoProvider ou LegendGraphicProvider, selon le type de requête.

3.1.4 Architecture du client

Le fait d'inclure un client dans la proposition de l'implémentation d'un service OGC peut sembler paradoxal : l'un des bénéfices à retirer étant le fait que le service est alors utilisable par les clients existants. La capture d'écran suivante, réalisée en combinant une carte des USA produite avec le service HyperAtlasWMS (couche correspondant à la carte Ratio) avec les données du service utilisé dans la partie 2.2.7, le montre.

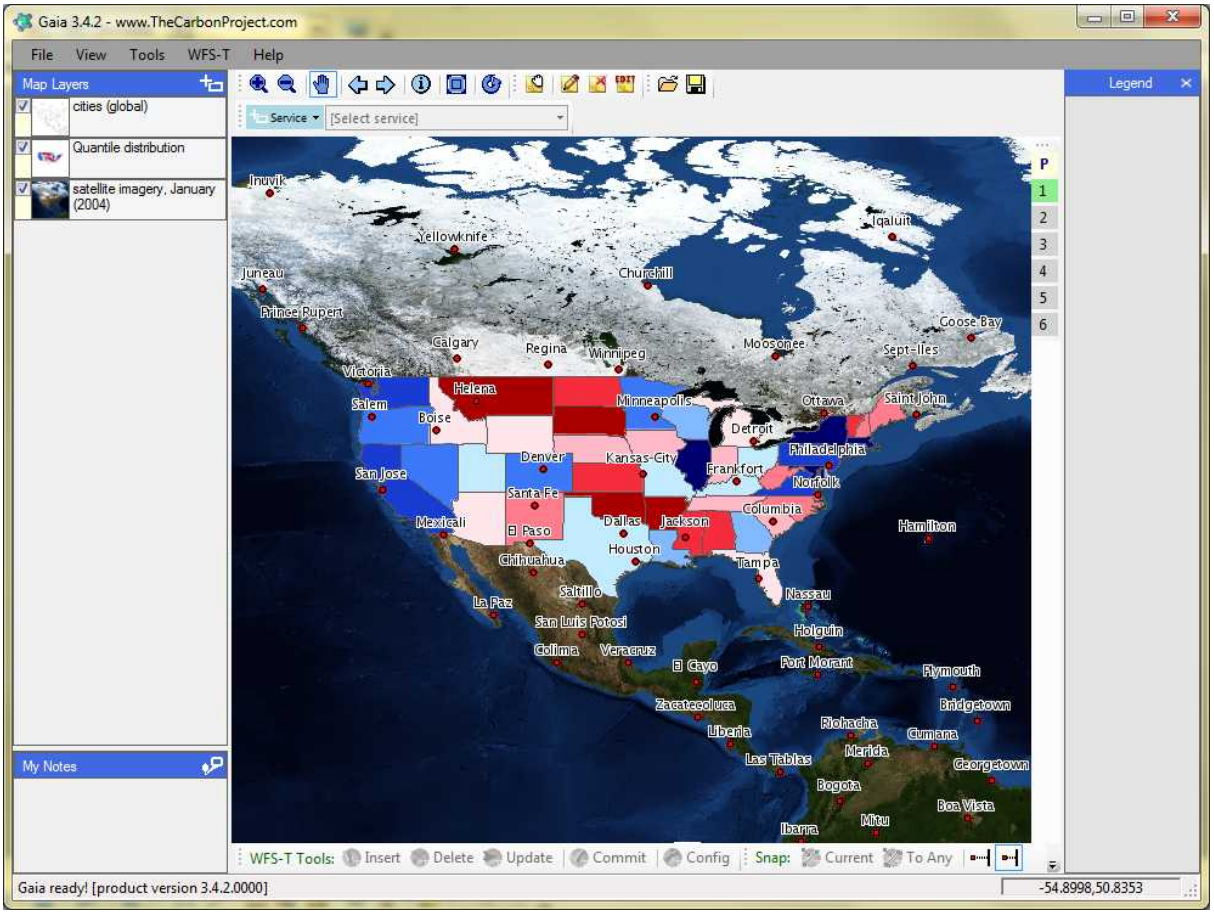


FIGURE 3.5 – Utilisation du service HyperAtlasWMS avec le client Gaïa

Ce choix a été motivé par 3 constats :

- il reste une fonctionnalité d'HyperAtlas qui n'est pas adressée par WMS : la génération d'un rapport reprenant les huit cartes. Une des fonctionnalités du client sera la possibilité pour l'utilisateur de réaliser un rapport à partir des cartes qu'il aura obtenues du service ;
- l'étude des différents clients WMS décrite au chapitre 2 a montré qu'ils avaient en commun un défaut majeur : ils n'utilisent pas ou très peu la requête GetLegendGraphic. Dans le contexte d'HyperAtlas c'est rédhibitoire : sans leur légende, les cartes HyperAtlas sont incompréhensibles. Le dégradé de couleurs n'a de sens que s'il est accompagné des valeurs correspondant aux intervalles, les disques que par rapport aux grandeurs qu'ils représentent ;
- le service HyperAtlasWMS utilise des paramètres VendorSpecific, qui sont indispensables pour exploiter la richesse des possibilités du logiciel HyperAtlas. Leur utilisation, même avec Gaïa, nécessite de comprendre le XML associé à la requête *GetCapabilities* du service, ainsi que de savoir éditer la requête à la main, deux obstacles qui seront contournés par un client dédié.

La conception du client repose sur les mêmes principes que celle du service, avec les mêmes objectifs

en ligne de mire : disposer très tôt dans le processus de développement d'une version opérationnelle et faciliter les tests unitaires.

Le client est divisé en trois modules :
– un client WMS générique, capable d'émettre des requêtes vers n'importe quel service WMS ;
– un greffon spécialisé pour le service HyperAtlasWMS. Ce greffon permet d'enrichir les requêtes génériques avec les paramètres VENDORSPECIFIC d'HyperAtlasWMS ;
– un module chargé de créer et de sauvegarder des rapports construits à partir des ressources consultées à l'aide des services WMS.

Dans la suite, nous appellerons BrowserWMS le client générique et HyperAtlasPlugin le greffon dédié à HyperAtlas.

3.1.5 Utilisation du client BrowserWMS

BrowserWMS comprend à trois cas d'utilisation principaux :
– connexion à un service WMS : ce cas d'utilisation comprend l'établissement d'une connexion via le réseau, mais aussi le décryptage du fichier *capabilities* et l'initialisation des éditeurs (par exemple, un éditeur permet de sélectionner les différentes couches du service) qui permettront ensuite à l'utilisateur d'émettre des requêtes (*GetMap*, *GetFeatureInfo* et *GetLegendGraphic*). Ces requêtes correspondent au cas d'utilisation suivant : l'acquisition ;
– acquisition de données grâce au service WMS auquel l'utilisateur s'est connecté. Chaque interaction de l'utilisateur avec ces éditeurs a pour effet d'émettre une requête et d'afficher le résultat (carte, légende ou informations concernant un point de la carte). L'utilisateur peut à tout instant décider d'acquérir ce résultat : il est alors conservé, et pourra être réutilisé pour fabriquer un document ;
– création/édition de documents à partir des données acquises. Les documents ainsi produits peuvent ensuite être sauvegardés, puis réouverts et modifiés plus tard.

Le diagramme UML suivant présente les cas d'utilisation de BrowserWMS.

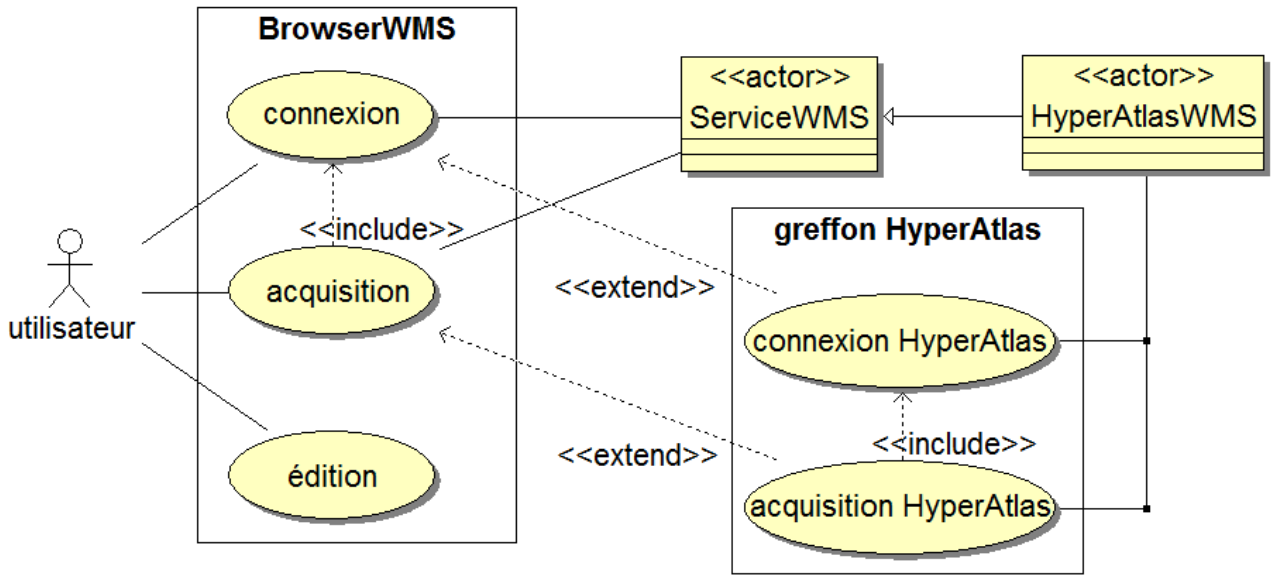


FIGURE 3.6 – Cas d'utilisation de BrowserWMS

Ces cas d'utilisation sont volontairement très généraux, et seront détaillés dans le chapitre Réalisations.

3.2 Cycle de développement

3.2.1 Méthode

Dans la vie d'un projet le choix d'une méthode de travail est un moment important, car il est souvent lourd de conséquences. Le format du projet, un stage de neuf mois, un seul développeur, fait que si l'on considère le triangle des trois contraintes d'un projet, Coût - Qualité - Délai, la seule qui soit ajustable est la qualité. Ce constat nous a amené à choisir un cycle de développement itératif et incrémental, chaque itération se terminant avec une version plus riche en fonctionnalités que la précédente, spécifiée et testée.

Il est important de préciser ici ce que signifie ajuster la qualité : diminuer le nombre de fonctionnalités, et non pas écrire un code de moindre qualité voire faire une impasse sur les phases de validation des spécifications. Ce choix, s'il ne fait pas disparaître le risque de ne pas réaliser le projet en temps et en heure a pour avantage de limiter les conséquences : la dernière version réalisée est une version acceptable, même si la fonctionnalité qu'elle propose est limitée par rapport aux prévisions.

Certaines fonctionnalités dépendent d'autres (une requête de la couche HyperAtlasWMS doit d'abord exister au niveau de la couche abstraite), elles doivent naturellement être réalisées d'abord, mais ce n'est pas systématique. Lorsque deux tâches sont indépendantes, il faut établir un ordre de priorité. Ceci est réalisé en attribuant à chacune des fonctionnalité une valeur client et un coût, qui correspond à une estimation du temps de développement que va prendre la tâche. Le rapport de ces deux valeurs définit la priorité de la tâche :

$$priorité = \frac{valeur}{estimation}$$

Par exemple, l'implémentation d'une couche (layer) géographique correspondant à la carte de synthèse a été repoussée à l'itération 14 :
– sa valeur ajoutée a été jugée assez faible (d'après les utilisateurs c'est la moins utile des cartes HyperAtlas) ;
– le temps de développement a été estimé comme relativement élevé. Les remaniements de code nécessaires sont assez conséquents, voire complexes, au niveau du service et au niveau de l'existant.

Cette méthode dépend beaucoup de la capacité des développeurs à estimer correctement la longueur et la difficulté des tâches liées au développement et de la capacité des utilisateurs (représentants de la maîtrise d'ouvrage) à estimer la valeur ajoutée d'une fonctionnalité. Dans ce cadre, la capacité des uns et des autres à communiquer et à convaincre joue un rôle important. À la fin de chaque étape, l'expertise des développeurs augmente et ils peuvent revoir à la baisse ou à la hausse leurs estimations. La valeur associée aux besoins peut évoluer également. Cela nécessite parfois de réorganiser les itérations. Fondamentalement cette méthode permet d'avoir une bonne visibilité du déroulement du projet :
– la maîtrise d'œuvre sait ce qui reste à faire, et dans quel ordre le faire ;
– la maîtrise d'ouvrage sait combien de temps cela va prendre.

Un autre avantage du cycle de développement itératif et incrémental est qu'il permet de s'adapter assez facilement au changement. D'une certaine façon, on peut considérer que le projet est terminé entre deux itérations, et si de nouveaux besoins apparaissent, on peut adapter la suite en intégrant ces nouveaux besoins aux fonctionnalités «restant à faire». Cela a été le cas pour l'itération 13 qui correspond à un besoin qui n'avait pas été envisagé au départ du projet, et non pas à un besoin avec une faible priorité.

3.2.2 Découpage

Le tableau 3.2 résume notre découpage du projet en itérations. La colonne «Description» correspond à un besoin, exprimé très brièvement, la colonne «Livrable» correspond à un livrable «point d'étape», qui sert de base à la démonstration de l'implémentation du besoin.

#	**Description**	**Livrables**
0	choix technique de la couche de transport	✔état de l'art services Web ; ✔classe WMSServiceBase.
1	*getcapabilities* au niveau abstrait	✔interface CapabilitiesProvider ; ✔paquetage *net.opengis.wms*[11]
2	*getmap* au niveau abstrait.	✔un premier service WMS : le service *FromOtherWMS*[12]
3	*getcapabilities* et *getmap* au niveau HyperAtlas, avec une seule couche géographique : l'aire d'étude	✔classes outils de la couche abstraite permettant le dessin d'une carte à partir des paramètres *GetMap* ✔première version du service HyperAtlasWMS
4	ajout des couches quantité (carte à disque) et ratio (choroplèthe)	✔remaniements de code[13] et ajout de tests dans l'existant HyperAtlas ✔version améliorée du service HyperAtlasWMS.
5	ajout des couches de déviation (choroplèthe)	✔remaniements de code et ajout de tests dans l'existant HyperAtlas ✔version améliorée du service HyperAtlasWMS
6	dessin de la légende sur la carte	✔version améliorée du service HyperAtlasWMS, le paramètre STYLES permet d'inclure le dessin de la légende sur la carte
7	requête *getfeatureinfo*	✔version améliorée du service HyperAtlasWMS
8	premier client	✔application permettant d'interroger un service WMS, d'afficher et d'interagir avec la carte
9	requête *getlegendgraphic*	✔version améliorée du service HyperAtlasWMS
10	client BrowserWMS	✔nouveau client WMS, gère toutes les requêtes
11	greffon HyperAtlas pour le client BrowserWMS	✔édition des paramètres VendorSpecific
12	éditeur de cartes dans le client BrowserWMS	✔possibilité d'utiliser le résultat des requêtes (image ou texte) dans un éditeur dédié
13	service HyperAtlasWMS à partir de fichier GML	✔application permettant d'extraire des données d'un fichier csv vers un format GML ✔remplissage du modèle HyperAtlas à partir d'un fichier GML
14	couche synthèse pour le service HyperAtlasWMS	✔version améliorée du service HyperAtlasWMS
15	ajout de fonctions de dessin dans l'éditeur de BrowserWMS	✔version améliorée du client, tracé de droite dans l'éditeur

TABLE 3.2 – Itérations du projet

Lors de l'exécution de l'itération 13, nous avons créé un utilitaire qui est décrit dans le chapitre Réalisations (p. 82). Toutefois nous ne le considérons pas comme un livrable définitif. Sa qualité et

11. Implémentation Java des spécifications OGC relatives aux schémas des fichiers *capabilities* de WMS.
12. À partir d'une requête, on en fait une nouvelle vers un service WMS déjà existant.
13. Diminution du couplage entre l'interface graphique et les traitements.

les fonctionnalités qu'il offre ne correspondent pas aux standards que nous nous étions imposés dans le reste de nos développements (tests, conception). Cet utilitaire a juste servi pour l'élaboration d'un format alternatif aux fichiers *.hyp*. Son code a été intégré au dépôt[14] de code du projet HyperCarte et pourra donc être éventuellement réutilisé et amélioré plus tard.

3.2.3 Versions WMS

Le service HyperAtlasWMS et le client BrowserWMS ne supportent que la version 1.1.1 du standard WMS. Cette limitation a été dictée par les contraintes de délai liées au stage. Elle est prévue par la norme WMS et n'a pas d'impacts fonctionnels, si l'on excepte que le service n'est compatible qu'avec les clients 1.1.1 et le client qu'avec les services 1.1.1.

Préférer la version 1.1.1 par rapport à la version 1.3.0 peut surprendre, mais les deux arguments suivants ont emporté notre décision :
– il y a peu de chances que la version 1.1.1 disparaisse au profit de la version 1.3.0 : la version 1.3.0 n'est pas une version récente[15], et peine à s'imposer car elle n'apporte rien par rapport à la version 1.1.1 en termes de fonctionnalités pour l'utilisateur (aucune nouvelle requête) ;
– la plupart des implémentations de services WMS certifiées conformes par l'OGC sont des versions 1.1.1[16], et peu de clients permettent l'utilisation de la version 1.3.0. Ce n'est pas le cas des versions gratuites que nous avons pu tester, ni celui de la bibliothèque JavaScript *OpenLayers*, très utilisée dans l'élaboration d'applications composites[17] à base de services Web.

Les développements réalisés l'ont été en ayant conscience de ce parti pris, et si dans l'avenir on devait assurer la compatibilité avec la version 1.3.0, le travail à réaliser ne sera pas difficile techniquement et ne remet pas en cause les choix de conception, ni pour le service HyperAtlasWMS ni pour le client BrowserWMS. De plus, la couche abstraite gère déjà toutes les versions de WMS, et si dans l'avenir l'équipe STEAMER souhaite la réutiliser, pour implémenter de nouveaux services WMS, elle pourra utiliser l'une ou l'autre version.

3.3 Conclusion

Notre proposition comporte cinq livrables qui sont détaillés au chapitre suivant :
– un paquetage de classes permettant la manipulation de standards d'échange de l'OGC[18] liés au service WMS. Le nom de ce paquetage est *net.opengis* ;
– un paquetage de classes *org.steamer.wms*,réutilisables, permettant d'implémenter des services Web WMS. Le nom de cette bibliothèque est WMSBase ;
– un service Web WMS réalisant les cartes du logiciel HyperAtlas. Le nom de ce service est HyperAtlasWMS ;
– un client pour les services Web WMS. Le nom de cette application est BrowserWMS. Elle peut intégrer des greffons dédiés à certains services WMS ;
– un greffon pour BrowserWMS, dédié au service HyperAtlasWMS.

14. *repository* en anglais.
15. Elle a vu le jour à la fin 2005, et l'OGC semble plutôt envisager des versions basées sur SOAP [@OGC] dans le futur.
16. 298 pour la version 1.1.1 contre 34 pour la version 1.3.0 en janvier 2011.
17. *Mashups* : combinaisons de sources géographiques dans une page HTML, très populaire dans l'univers de la *Volunteered Geographic Information* [Poulenard, 2010].
18. Toutes les versions des fichiers *capabilities* et des fichiers *exceptions* pour le WMS. La version 1.0 des fichiers GML.

Chapitre 4

Réalisations

La tactique, c'est ce que vous faites quand il y a quelque chose à faire. La stratégie, c'est ce que vous faites quand il n'y a rien à faire.

Xavier Tartakover

Ce chapitre présente l'ensemble des développements que nous avons réalisés au cours du stage. Cette présentation n'a pas pour vocation d'être une présentation exhaustive du code produit, mais plutôt de mettre en avant les différentes techniques que nous avons utilisées au cours de ce travail tout en essayant de présenter l'ensemble des fonctionnalités.

Ce chapitre est divisé en trois parties.

La première présente les différents outils utilisés, ainsi que les méthodes de développement.

La seconde présente le service :
- les bibliothèques du paquetage *net.opengis*, qui traduisent les spécifications des formats d'échange de l'OGC dans le cadre d'un service WMS en classes JAVA ;
- le service Web : WMSBase et sa spécialisation HyperAtlasWMS.

La dernière est consacré au client BrowserWMS et au greffon.

4.1 Outils et méthodes de développement

4.1.1 IDE Eclipse

Eclipse est un environnement de développement intégré, libre et surtout extensible, du fait que son architecture est développée autour de la notion de greffon. Eclipse IDE est un atelier logiciel permettant d'utiliser n'importe quel langage de programmation (grâce à ces greffons), mais du fait que Java est le langage qui a été utilisé pour le développer, le greffon Java est installé en même temps que n'importe quelle version.

Parmi les versions existantes, il existe une version Web Tools Platform [1](WTP) qui contient de nombreux outils adaptés au développement de services Web en Java.

L'interface graphique d'Eclipse présente le plan de travail (*workbench*). Le plan de travail est composé de perspectives, mais il ne permet d'en afficher qu'une à la fois. Une perspective présente une partie du projet de développement sous un certain angle, qui correspond aux vues et aux éditeurs nécessaires à un type de tâche : par exemple la perspective *Parcours des ressources* dissimulera la vue

1. Outils pour une plate-forme Web.

Débogueur, alors que la perspective *Débogueur* la mettra au premier plan. Les perspectives peuvent être personnalisées par l'utilisateur.

L'interface d'Eclipse propose de nombreux assistants pour permettre de faciliter la saisie des informations requises pour certaines tâches par l'utilisateur, la complétion de code notamment.

Pour finir, Eclipse intègre de nombreux outils qui facilitent le travail du développeur, comme par exemple *DRefactor*, qui facilite le remaniement de code.

4.1.2 Ant

Ant (ou Apache Ant) est un outil écrit en Java permettant d'automatiser des opérations répétitives tout au long du cycle de développement de l'application (compilation, exécution des tests, génération de documents, mises en production, *etc.*). L'exécution d'Ant est possible en mode ligne de commandes, ou grâce à l'IDE Eclipse.

Ant repose sur un fichier de configuration XML (*build.xml*) qui décrit les différentes tâches qui peuvent être exécutées par l'outil. Ant fournit un certain nombre de tâches courantes qui sont codées sous forme d'objets développés en Java. Ces tâches sont donc indépendantes du système sur lequel elles seront exécutées. De plus, il est possible d'ajouter ses propres tâches en écrivant de nouveaux objets Java respectant certaines spécifications.

Le fichier de configuration contient un ensemble de cibles (*target*). Chaque cible contient une ou plusieurs tâches. Chaque cible peut avoir une dépendance envers une ou plusieurs autres cibles pour pouvoir être exécutée. Le lancement d'une cible déclenche automatiquement le lancement des cibles dont elle dépend. Dans l'exemple décrit sur la figure suivante, le lancement de la cible créant l'archive Web (fichier .war) déclenchera la compilation d'HyperAtlasWMS. Cette seconde cible déclenchera la compilation de WMSBase, et ainsi de suite jusqu'au *setup* qui est une tâche d'initialisation. Le lancement de l'exécution des tests automatisés d'HyperAtlasWMS aura un comportement similaire.

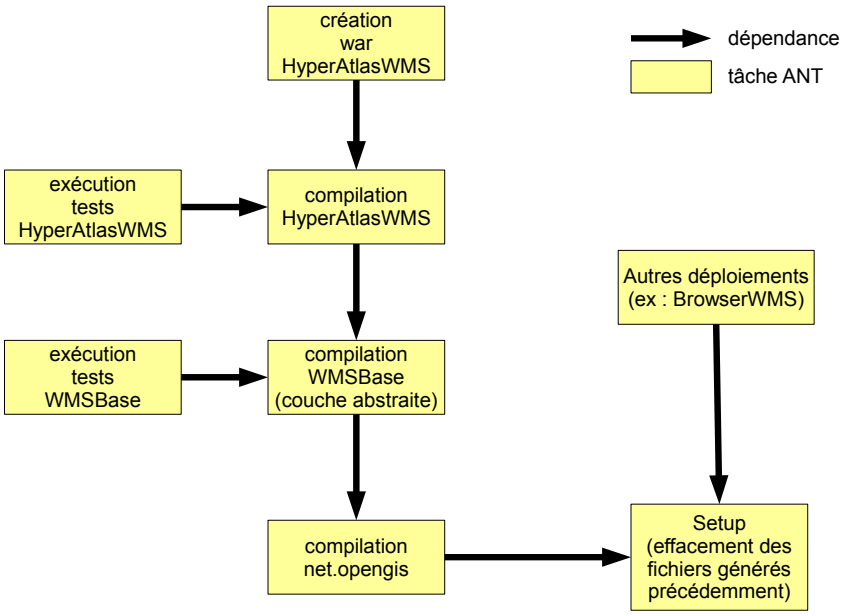


FIGURE 4.1 – Tâches Ant

4.1.3 Tomcat

Tomcat est un conteneur d'applications Web diffusé en *open source* sous une licence Apache. C'est aussi l'implémentation de référence des spécifications Servlets et JSP implémentées dans les différentes versions de Tomcat. Tomcat s'éxécute dans une machine virtuelle Java (JVM), autrement dit sur n'importe quelle plate-forme (LINUX, Windows, *etc.*).

En tant qu'implémentation de référence, facile à mettre en œuvre et riche en fonctionnalités, Tomcat est quasi incontournable dans les environnements de développements. Les qualités de ses dernières versions lui permettent d'être de plus en plus utilisé dans des environnements de production.

L'architecture de Tomcat est composée des éléments suivants, les noms anglais renvoyant à la figure 4.2 :

Serveur (Server) : le serveur encapsule tout le conteneur Web. Il ne peut s'exécuter qu'un seul serveur dans une JVM ;

Service : un service regroupe des connecteurs et un unique moteur de Servlet ;

Connecteur (Connector) : un connecteur gère les communications avec un client. Tomcat propose plusieurs connecteurs, notamment Coyote, pour les communications par le protocole HTTP ;

Moteur de Servlet (Engine) : un moteur de Servlet traite les requêtes des différents connecteurs associés au Service : c'est le moteur de traitements des Servlets ;

Hôte (Host) : un hôte est un nom de domaine dont les requêtes sont traitées par Tomcat. Un moteur peut contenir plusieurs hôtes ;

Contexte (Context) : un contexte permet l'association d'une application Web à un chemin unique pour un hôte. Un hôte peut avoir plusieurs contextes.

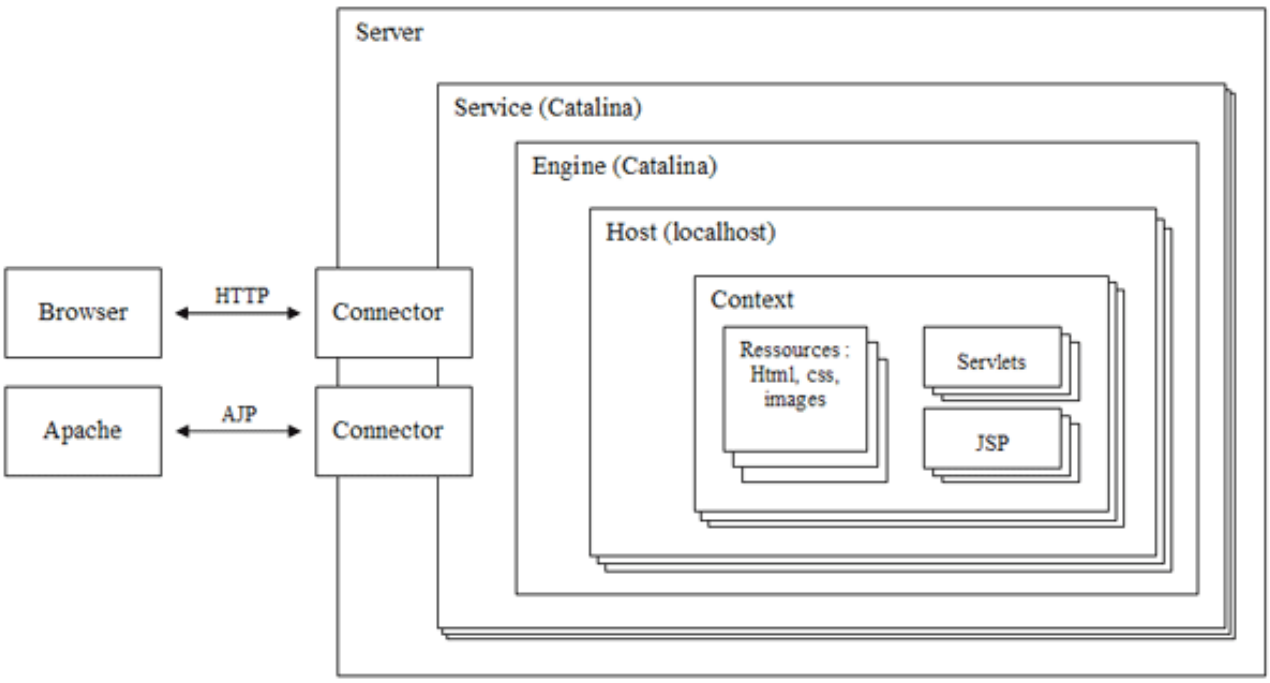


FIGURE 4.2 – Architecture Tomcat
Source : [Doudoux, 2010]

Les archives WAR sont déployées sur le serveur Tomcat dans un répertoire webapps. Une archive contient un fichier de configuration web.xml, qui décrit les modalités de l'exécution de la *Servlet*,

et deux répertoires : META-INF, qui peut contenir des fichiers décrivant le contexte de l'exécution du service, et WEB-INF qui contient tous les fichiers nécessaires à l'exécution du service (classes compilées, bibliothèques importées, fichiers de ressources).

La figure 4.3 montre l'arborescence de fichiers correspondant à un service Web, tel qu'HyperAtlasWMS, lorsqu'il est déployé sur un serveur Tomcat.

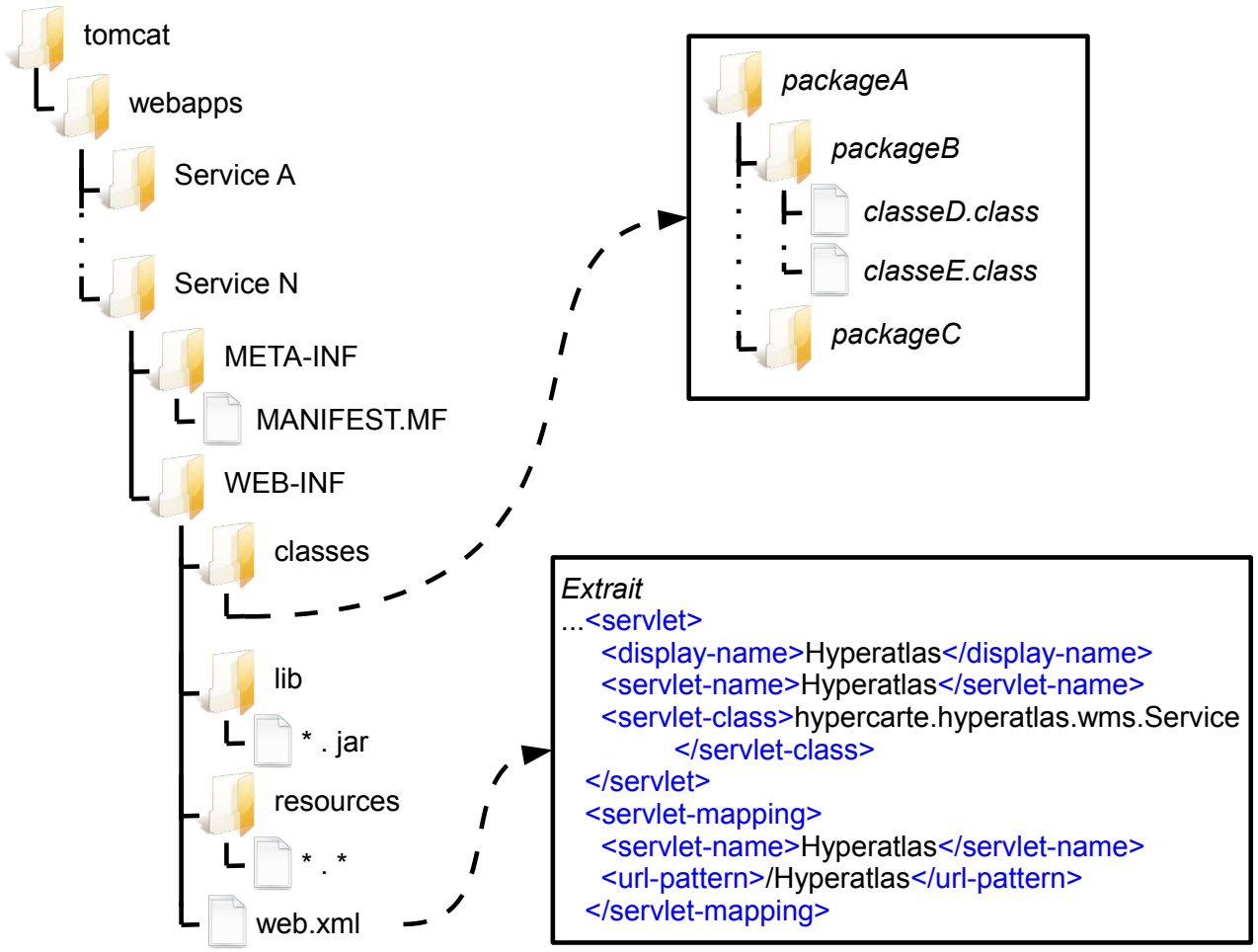


FIGURE 4.3 – Arborescence des fichiers d'un service Web dans Tomcat

4.1.4 Tests

Démarche

Les tests sont un élément fondamental du génie logiciel. Ils sont à un programme ce que la démonstration est à un théorème en mathématiques. Chaque spécification correspond à un contrat, et les tests permettent de valider que ce contrat est rempli. Les tests sont répartis en plusieurs catégories :
- tests unitaires, qui valident le contrat d'une classe ;
- tests d'intégration, qui valident le contrat d'un groupe de classes ;
- tests fonctionnels, qui valident la fonctionnalité ;
- tests de charge, qui permettent de valider les spécifications relatives à la performance du logiciel : rapidité, espace mémoire...

Des tests rédigés sous forme de scripts qu'un testeur déroule peuvent remplir ce rôle, mais des tests automatisés offrent deux avantages :
- exécution rapide ;

– exécution gratuite (on peut continuer à travailler tandis que la machine les exécute).

Concernant les tests, dans le cadre du projet, la pratique qui a été adoptée est le *Test Driven Development* [2] (TDD) [Beck, 2002] :
– écrire le test unitaire avant le code ;
– vérifier qu'il échoue (cette étape permet de valider que le test correspond à la fonctionnalité que l'on s'apprête à coder) ;
– écrire le code ;
– vérifier que le test passe (la fonctionnalité est codée) ;
– vérifier que tous les tests passent (le code ajouté a-t-il introduit des régressions dans le code préexistant ?) ;
– optimiser le code [Fowler et Beck, 1999], par exemple factoriser du code dupliqué [3].

Le diagramme suivant montre que cette activité est en fait un cycle de développement, puisque lorsque le codage d'une fonctionnalité est achevé (point de sortie de l'activité), on recommence pour la fonctionnalité suivante.

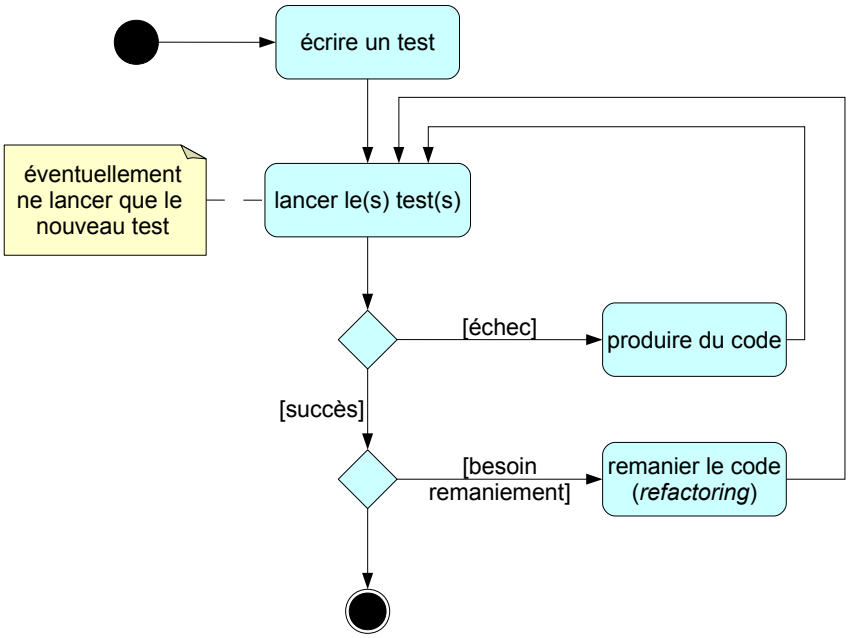


FIGURE 4.4 – Cycle de développement TDD

Écrire les tests avant le code plutôt qu'après n'apporte rien par rapport à la validation finale du projet, mais cela permet de détecter au plus tôt certains problèmes, généralement liés à une mauvaise conception.

Les symptômes [4] suivants doivent inciter à la vigilance :
– le test passe d'emblée (dans ce cas, c'est souvent le test qui est mal écrit) ;
– un test difficile à écrire ;
– un test difficile à faire passer ;

2. Développement piloté par les tests, cette méthode préconise d'écrire le code des tests avant le code des fonctionnalités.
3. DRY : *Don't Repeat Yourself* (ne vous répétez-pas), le code dupliqué est difficile à maintenir.
4. On les appelle souvent des *code smells* (odeurs de code).

– nombreux tests pré-existants qui échouent après l'écriture du code (possibilité de régression).

La démarche TDD, qui fait partie de la méthode *Extreme Programming* (XP) [Beck, 1999], est particulièrement adaptée au cycle de développement itératif et incrémental adopté pour la conduite du projet. On ne peut toutefois pas affirmer que le projet a été conduit en suivant la méthode XP *stricto sensu* : cette méthode correspond aux pratiques d'une équipe, et dans le cadre du projet il n'y a qu'un développeur (pas de programmation en binôme, ni de revues de code par exemple).

Parmi les pratiques recommandées par XP, voici celles que nous avons adoptées au cours du stage :
– client sur site qui doit avoir les connaissances de l'utilisateur final et avoir une vision globale du résultat à obtenir : les deux encadrants du stage ont joué ce rôle tout au long du projet ;
– petites livraisons : dès la quatrième itération le service HyperAtlasWMS pouvait être considéré comme opérationnel ;
– tests de recette (ou tests fonctionnels) ;
– tests unitaires ;
– conception simple ;
– remaniement du code.

Outils Java

JUnit est une bibliothèque pour le développement et l'exécution de tests unitaires automatisables. Cette bibliothèque a été créée par Erich Gamma [5] et Kent Beck [6]. Le principal intérêt est de s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications. JUnit fait partie de la famille des xUnit, descendant de SUnit, qui fut créé pour le langage SmallTalk. Le principe est assez simple : dans un langage fonctionnel, un test est une fonction comme une autre, alors dans un langage objet, un test est un objet (une classe). Toutes les classes de test sont des spécialisations de la classe *junit.framework.TestCase*. Une classe de test va contenir plusieurs méthodes de test dont le nom doit commencer par test. Le nom d'un test doit être choisi avec soin, car plus le nom est significatif, plus vite on peut identifier la cause d'un problème (c'est-à-dire sans même regarder le code du test). Les noms du genre testCase1,..., testCase3 doivent être évités, car ils n'indiquent pas ce que fait le test.

TestCase possède plusieurs méthodes qui facilitent l'écriture des tests. Tout d'abord deux méthodes qui permettent de créer un contexte pour les tests :

setup : permet d'initialiser le test (par exemple, l'instanciation d'objets qui seront utilisés tout au long des différents tests)

teardown : «fait le ménage» après l'exécution du test (par exemple, la destruction d'objets créés lors de cette exécution).

Ensuite les méthodes de type *assert* [7], qui suivent le schéma suivant : assert(<Message>,<Valeur attendue>,<Valeur obtenue>). Lorsque le test comparant la valeur attendue (*expected* en anglais) et la valeur obtenue (*actual* en anglais) échoue, une exception est générée avec le message argument. L'argument message est optionnel, mais comme pour les noms de test, le message permet de comprendre les raisons de l'échec du test sans même regarder le code. Cela peut sembler anecdotique, mais une des raisons d'être du TDD est de ne plus déboguer le code, ces séances de débogage étant remplacées par l'exécution systématique des tests entre chaque modification du code. Le tableau suivant résume les différentes méthodes de ce type.

5. Erich Gamma est un membre du «*gang of four*», les quatre créateurs du concept *design pattern* (patron de conception) [Gamma *et al.*, 1994].
6. Kent Beck est une figure du monde de l'agilité[Beck, 1999, Beck, 2002, Fowler et Beck, 1999].
7. Affirmer ou faire valoir en anglais.

Méthode	Rôle de vérification
assertTrue	la valeur fournie en paramètre est vraie
assertFalse	la valeur fournie en paramètre est fausse
assertEquals	l'égalité de deux valeurs. Il existe de nombreuses surcharges de cette méthode : – pour chaque type primitif – pour un objet de type Object – pour un objet de type String
assertNull	l'objet fourni en paramètre est *null* (pas initialisé)
assertNotNull	l'objet fourni en paramètre n'est pas null
assertSame	les deux objets fournis en paramètre font référence à la même entité, les deux exemples suivants sont équivalents : – assertSame("Les deux objets sont identiques",obj1, obj2) ; – assertTrue("Les deux objets sont identiques",obj1 == obj2) ; deux objets peuvent être égaux sans être identiques, comme par exemple deux chaînes de caractères ayant la même valeur.
assertNotSame	les deux objets fournis en paramètre ne font pas référence à la même entité

TABLE 4.1 – Méthodes de type *assert* de la classe TestCase

La dernière méthode de TestCase est la méthode *fail*() qui déclenche automatiquement l'échec du test. Elle est utilisée pour valider le fait que l'appel qui la précède a déclenché une exception. Le test suivant est une illustration de son usage. Dans ce test, qui passe malgré l'appel à la fonction *fail()*, le fait que seuls les types MIME valides (d'autres tests vérifient lesquels) soient supportés par *createGraphicTarget* est validé. Lors de l'exécution du test, *createGraphicTarget* a levé une exception du type FormatNotSupported, et l'appel à *fail()* a été contourné.

```
public void testOnlyValidMimeTypeNamesAreSupportedByCreateGraphicTarget(){
  int width = 10;
  int height = 100;
  try {
    BufferedImage image = Tools.createGraphicTarget("NotValidMimeTypeName",
        width, height);
    fail();//cet appel n'est pas exécuté lorsque le test réussit
  }
  catch (FormatNotSupported e) {
  }
}
```

Il existe d'autres bibliothèques, basées sur JUnit, adaptées au contexte que l'on teste. La bibliothèque *httpunit* permet de tester les Servlets comme si elles étaient installées sur un serveur. Dans le test suivant, la classe ServletRunner qui est issue du paquetage *httpunit* joue le rôle de Tomcat et permet en plus d'instancier un client virtuel.

```
public void testGetMapWithVendorSpecificForgottenIsOK() throws IOException,
    SAXException{
    ServletRunner sr = new ServletRunner();
    //l'objet ServletRunner joue le rôle du serveur
    sr.registerServlet( "myServlet", SimpleWMS.class.getName() );
    //SimpleWMS est un service avec des paramètres VendorSpecific
    ServletUnitClient client = sr.newClient();
    //l'objet ServletUnitClient joue le rôle du client
```

```
    WebRequest request = getRequestWithWMSParameters();
    try{
      //le client émet la requête, qui ne contient pas les paramètres
          VendorSpecific
      client.getResponse( request );
    }
    catch(com.meterware.httpunit.HttpException ex){
      //si le service n'arrive pas à gérer l'absence des paramètres
          VendorSpecific, le test échoue
      fail();
    }
}

//cette méthode renvoie une requête GetMap
private WebRequest getRequestWithWMSParameters() {
    WebRequest request = new GetMethodWebRequest( "http://test.meterware.com/
        myServlet" );
    request.setParameter("SERVICE", "wms");
    request.setParameter("REQUEST", "getMap");
    request.setParameter("VERSION", "1.3.0");
    request.setParameter("WIDTH", "10");
    request.setParameter("HEIGHT", "10");
    request.setParameter("BBOX", "1.0,2.0,1.0,2.0");
    request.setParameter("LAYERS", "layer1,layer2");
    request.setParameter("STYLES", "style1,style2");
    request.setParameter("FORMAT", "image/gif");
    request.setParameter("SRS", "default");
    return request;
}
```

L'annexe «Tests» contient d'autres exemples de tests unitaires, fonctionnels ou de performance.

Pour clore cette section consacrée aux outils de test Java, il existe un mot-clé **assert** dans le langage Java (depuis la version 1.4) qui permet d'inclure des affirmations dans le code. Cela ne remplace pas les tests automatisés, mais permet d'introduire dans le code des éléments de programmation par contrat (DBC : Design By Contract), ce qui renforce sa qualité et sa robustesse [Meyer, 1988, Plessel, 1998]. La syntaxe est la suivante assert <Condition> <Message>, et, comme dans les tests de JUnit, si la condition n'est pas vérifiée, une exception (textttAssertionError) est levée. Utiliser ce mécanisme permet, comme les tests unitaires, de simplifier le débogage. Cette vérification ne ralentit pas l'exécution du code : elle peut être activée ou non (par défaut elle est désactivée), on peut donc économiser ce coût supplémentaire lors de l'exécution du code lorsqu'il est distribué chez un client. Par contre on peut l'activer lors de phases de validation, comme par exemple lors de l'exécution des tests.

Ce mécanisme s'avère précieux lors de :
– l'élaboration d'algorithmes complexes ;
– traitements avec des pré-conditions ou des post-conditions fortes ;
– traitements que l'on sait sensible au changement ;
– code peu lisible.

Dans l'annexe «Programmation par contrat dans HyperAtlas» se trouvent des exemples issus du code. Il existe en Java des mécanismes d'annotation permettant le DBC en Java [Wang *et al.*, 2003], mais dans le cadre du projet, le DBC a plus été utilisé ponctuellement comme un outil de travail que comme une méthode de conception.

4.2 Service

Le diagramme suivant présente les paquetages principaux impliqués dans le service HyperAtlasWMS et leurs dépendances. Les sections suivantes de ce chapitre vont détailler le contenu de chacun d'entre eux. Les sections 4.2.1 Standards OGC et 4.2.2 Couche WMS abstraite décrivent le travail effectué par rapport à la spécification WMS, la section 4.2.3 Couche HyperAtlas celui lié à HyperAtlasWMS.

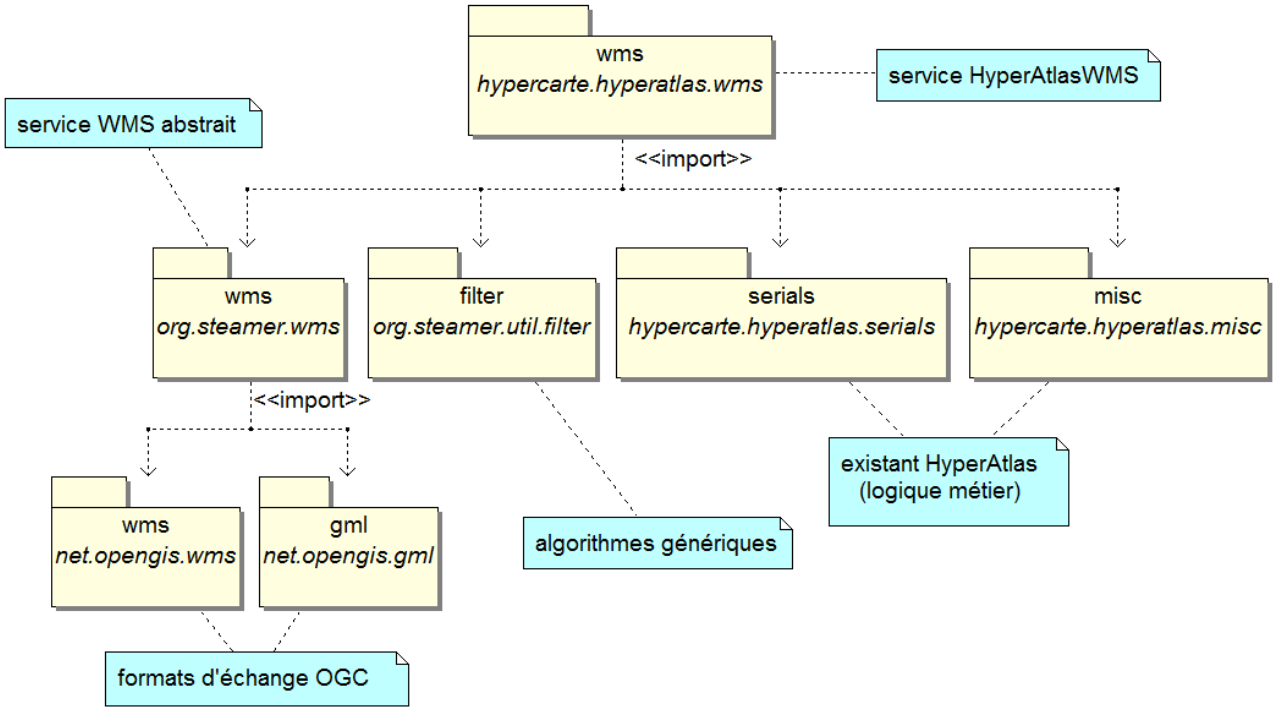


FIGURE 4.5 – Paquetages de l'architecture HyperAtlasWMS

4.2.1 Standards OGC

Les spécifications de l'OGC ne sont pas des solutions logicielles, mais des schémas détaillant l'API des différents services ou des formats d'échange. Ces API sont spécifiées à l'aide de fichiers dtd ou xsd. Pour pouvoir les implémenter, le développeur doit les traduire en un code correspondant au langage de programmation qu'il utilise, Java dans le cadre du projet HyperAtlas.

L'outil Ant permet de générer à partir de ces schémas un paquetage contenant l'ensemble des classes Java équivalentes. La tâche Ant qui permet cela est la tâche XJCTask du paquetage *com.sun.tools.xjc*. Les extraits suivants issus du fichier *build.xml* du projet HyperCarte (projet qui contient HyperAtlas) montrent la syntaxe permettant d'utiliser ce type de tâches :

– le premier extrait est la description d'une tâche de génération. Elle s'appelle xjc, et elle utilise la classe *com.sun.tools.xjc.XJCTask* ;

```
<typedef name="xjc" classname="com.sun.tools.xjc.XJCTask" />
```

– le second extrait correspond à la création des classes qui seront utilisées pour le fichier *capabilities* (version 1.3.0) à partir du fichier *capabilities_1_3_0.xsd*. Ces classes seront créées dans le paquetage *net.opengis.wms.capabilities130* ;

```
<target name="generateWMSCapabilities_1_3_0" description="generates
    WMSCapabilities java files from .xsd schema">
```

```
        <xjc destdir="${basedir}/src" extension="true" package = "net.opengis.
            wms.capabilities130">
                <schema dir="${basedir}/etc/wms/1.3.0" includes="
                    capabilities_1_3_0.xsd" />
        </xjc>
</target>
```

– le troisième extrait correspond à la création des classes du fichier GML (version 1.0.0) à partir du fichier *gmlfeature.dtd*. Ces classes seront créés dans le paquetage *net.opengis.gml.gml100*.

```
<target name="generateGML" description="generates GML java files from .dtd
    schema">
        <xjc destdir="${basedir}/src" extension="true" package = "net.opengis.
            gml.gml100">
                <arg value="-dtd" />
                <schema dir="${basedir}/etc/wms/gml/1.0.0" includes="
                    gmlfeature.dtd" />
        </xjc>
</target>
```

Le paquetage produit par une tâche de génération va contenir les classes correspondant aux éléments du schéma. Le principe de JAXB [8], l'outil sur lequel s'appuie ce type de tâches, est similaire à celui de JAX-WS présenté dans la partie 2.2.3. Les différents éléments XML du schéma sont liés aux classes par un mécanisme d'annotation. Le tableau suivant présente les annotations les plus fréquemment utilisées par JAXB, suivi des éléments XML et Java qu'il fait se correspondre.

Annotation	**XML**	**élément JAVA annoté**
@XmlRootElement	nœud racine	classe
@XmlElement	nœud simple ou enfant	attribut d'une classe (souvent une autre classe)
@XmlAttribute	attribut	attribut d'une classe (en général une chaîne de caractères normalisée)

TABLE 4.2 – Annotations JAXB

Le fragment de code suivant, traduit en XML donnera un nœud WMTMSCapabilities avec un attribut version (ex : <WMT_MS_Capabilities version="1.1.1">), un attribut séquence et deux nœuds enfants, Service et Capability. L'élément *required* signifie que l'objet annoté doit obligatoirement être présent dans le XML, et l'annotation @XmlJavaTypeAdapter indique la classe utilisée pour transformer la valeur contenue dans le XML en type Java.

```
@XmlRootElement(name = "WMT_MS_Capabilities")
public class WMTMSCapabilities {
    @XmlAttribute
    @XmlJavaTypeAdapter(NormalizedStringAdapter.class)
    protected String version;
    @XmlAttribute
    @XmlJavaTypeAdapter(NormalizedStringAdapter.class)
    protected String updateSequence;
    @XmlElement(name = "Service", required = true)
    protected Service service;
    @XmlElement(name = "Capability", required = true)
    protected Capability capability;
    ...
```

8. Java Architecture for XML Binding.

En plus des classes correspondant au schéma, la tâche de génération crée une classe ObjectFactory qui permet d'instancier toutes les classes du paquetage. Ce type de tâche est d'autant plus précieux qu'en plus de représenter un gain de temps considérable, il permet au développeur d'éviter des erreurs en traduisant le schéma en classes. Les paquetages *net.opengis.wms* (qui contient les classes correspondant aux *capabilities* et exceptions du wms) et net.opengis.gml ont été générés ainsi. Le tableau suivant établit les correspondances entre les différents formats d'échange standards de l'OGC et les classes générées.

Spécification OGC	**paquetage**
Capabilities 1.0.0	*net.opengis.wms.capabilities100*
Capabilities 1.0.7	*net.opengis.wms.capabilities107*
Capabilities 1.1.0	*net.opengis.wms.capabilities110*
Capabilities 1.1.1	*net.opengis.wms.capabilities111*
Capabilities 1.3.0	*net.opengis.wms.capabilities130*
Exceptions 1.1.0	*net.opengis.wms.exceptions110*
Exceptions 1.1.1	*net.opengis.wms.exceptions111*
Exceptions 1.3.0	*net.opengis.wms.exceptions130*
GML 1.0.0	*net.opengis.gml.gml100*

TABLE 4.3 – Correspondance entre les normes OGC et paquetages de *net.opengis*

En plus des classes générées, ces paquetages contiennent des classes outils permettant de remplir la partie géométrie d'un fichier GML directement à partir des géométries com.vividsolutions.jts.geom, un paquetage très utilisé dans l'univers des SIG (PostgreSQL/PostGIS) et une classe Marshaller [9] qui permet d'utiliser simplement le mécanisme permettant de passer du contenu XML (fichier ou flux de données) aux classes à partir d'un nom du paquetage (équivalent au *namespace*), ou l'inverse. Le schéma suivant illustre ce mécanisme.

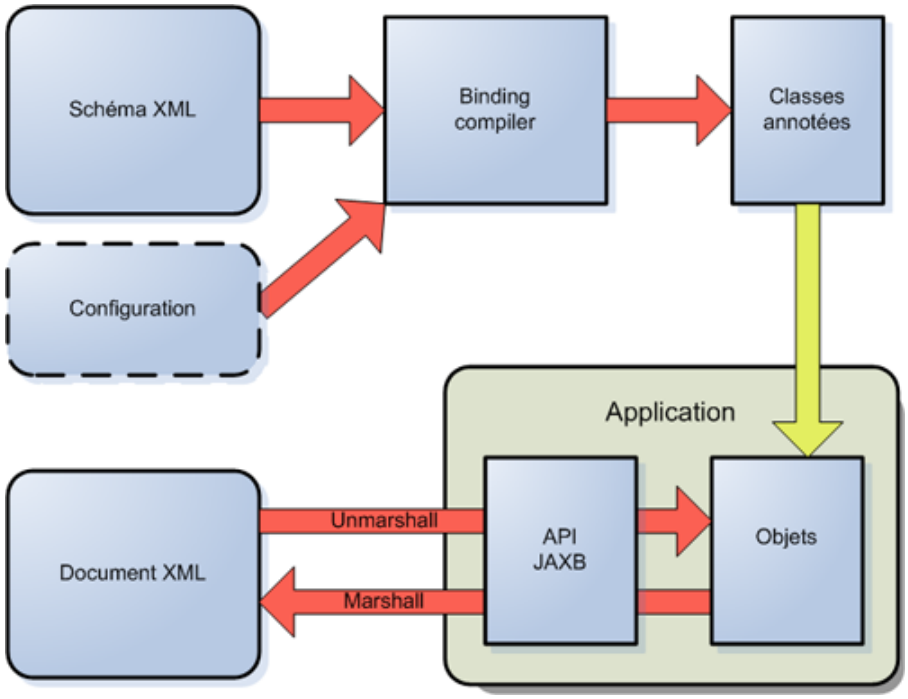


FIGURE 4.6 – Sérialisation / désérialisation avec l'API JAXB
Source : [Doudoux, 2010]

9. Sérialisation se dit *serialization* ou *marshalling* en anglais.

4.2.2 Couche WMS abstraite

La couche abstraite correspondant au paquetage *org.steamer.wms* est une bibliothèque de classes permettant la réalisation d'un service WMS.

Elle a été écrite dans le but de réaliser le service HyperAtlasWMS, mais elle peut être *réutilisée pour implémenter n'importe quel autre service de type WMS* : elle ne référence aucun des paquetages *HyperCarte.**.

Traitement d'une requête HTTP GET

La classe principale du paquetage est la classe WMSServiceBase. Elle étend la classe HttpServlet et implémente l'interface WMSService. Son rôle est de réaliser les étapes 2 et 3 du cas d'utilisation illustré par la figure 3.4 Utilisation du service HyperAtlasWMS présenté dans notre proposition, c'est-à-dire :
– valider la requête GET reçue par rapport à la norme WMS ;
– analyser la requête GET reçue et à partir de celle-ci construire un objet métier correspondant au type de la requête ;
– transmettre cet objet à la classe chargée de traiter ce type de requête.

Cette action est réalisée en quatre étapes :
– analyse et validation des paramètres communs à toutes les requêtes : SERVICE, REQUEST et VERSION. À ce stade le service a identifié le type de requête. À ce jour, seul le service WMS est traité, mais on pourrait par la suite implémenter d'autres types de services : tous les services de l'OGC sont basés sur le triplet SERVICE/REQUEST/VERSION ;
– analyse et validation des paramètres définis par la norme pour le type de requête (seule l'absence d'un paramètre obligatoirement présent déclenche une exception). La validation ne concerne pas que la présence d'un paramètre, elle peut aussi vérifier que la valeur associée au paramètre est cohérente par rapport à ce qui est représenté : nombre entier (WIDTH), coordonnées spatiales (BBOX), nombre de couches (LAYERS) et de styles (STYLES) égaux. À ce stade l'objet métier correspondant à la requête est instancié ;
– à partir de la requête, le membre en charge des paramètres propriétaires (VendorSpecificCapabilitiesProvider), s'il existe, ajoute ces paramètres dans l'objet métier. Tous ces paramètres sont optionnels, et la couche abstraite n'a aucune connaissance de ces paramètres autre que leur nom, cette étape n'effectue donc aucune vérification ;
– l'objet requête est transmis à la classe *provider* (fournisseur) du type de requête identifié lors de la première étape.

Le code suivant correspond à l'extraction d'un paramètre, fait à l'aide de la méthode *getParameter(HttpServletRequest request, String possibleName, boolean optional)*. La traduction de la requête est donc une suite d'appels à cette méthode, avec comme arguments la requête GET reçue (HttpServletRequest), le nom du paramètre (par exemple LAYERS) et un booléen valant vrai si le paramètre est optionnel.

Le principe est assez simple : à chaque appel, la méthode parcourt les paramètres de la requête, si le paramètre correspond au nom passé en argument (on ne tient pas compte des majuscules, ce qui est une spécification de l'OGC) on renvoie la valeur qui lui est associée, sinon, on ne renvoie rien (null) si le paramètre est optionnel. On génère une exception (MissingParameter) dans le cas contraire. Cette exception est alors redirigée vers le membre ExceptionProvider. Celui-ci renverra au client un message d'erreur, en utilisant le format correspondant à la version du service, ou au format

d'exception paramètre [10].

```
public static String getParameter(HttpServletRequest request,
    String possibleName, boolean optional) throws MissingParameter {
    Iterator<?> it = request.getParameterMap().keySet().iterator();
    while(it.hasNext()){//parcours des clés paramètre/valeur
      String key = it.next().toString();
      if(key.equalsIgnoreCase(possibleName) )
        return request.getParameter(key);
    }
    if(optional)//paramètre optionnel, la méthode retourne null
      return null;
  throw new MissingParameter(possibleName);//paramètre obligatoire, lancement d'
      une exception paramètre manquant
}
```

Les objets métiers correspondant aux requêtes (MapRequest, FeatureInfoRequestet LegendGraphicRequest) n'ont pas d'autre utilité que de présenter les arguments de la requête à la couche concrète du service en lui *masquant la couche de transport*. Le seul qui fasse exception à cette règle est BBox, qui possède en plus quelques fonctionnalités gérant la relation entre étendue géographique et image. Ils spécialisent tous la classe RequestVendorSpecificCapabilities, qui gère les paramètres VendorSpecific.

Le diagramme de classes suivant présente ces objets.

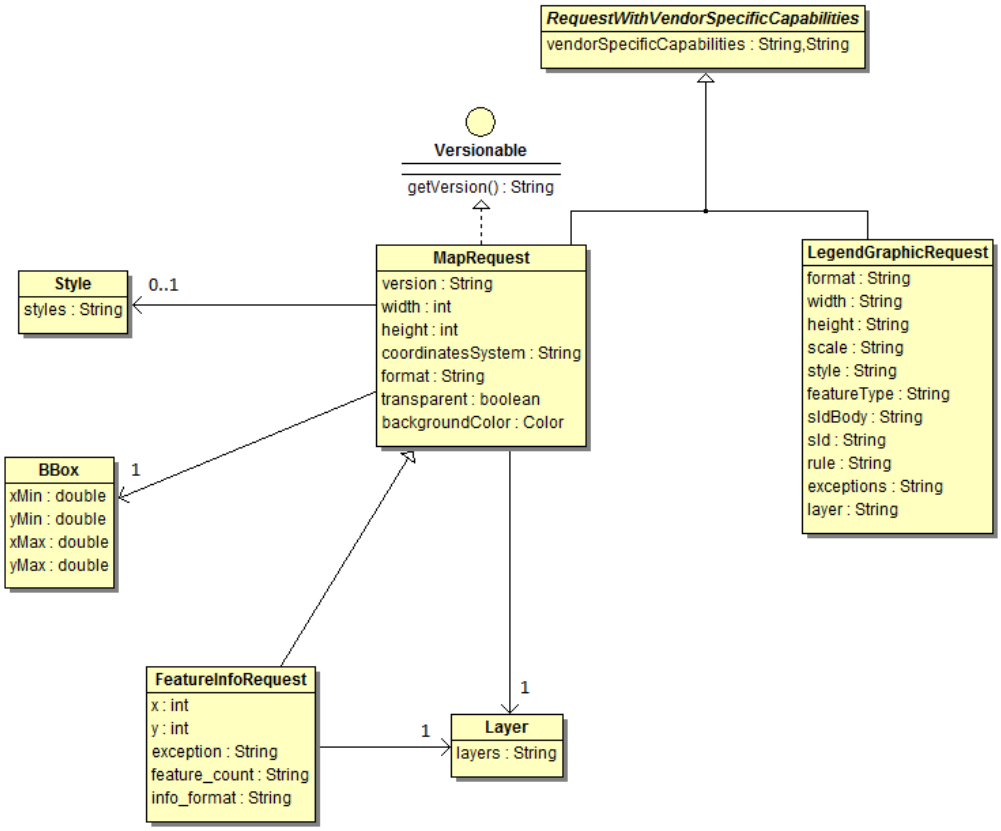


FIGURE 4.7 – Classes requêtes de la couche abstraite

10. Pour toute requête, on peut spécifier un format pour les exceptions. Les formats supportés par un service sont indiqués dans le fichier *capabilities*. Le format par défaut correspond aux schémas OGC.

Classes outils

En dehors des formats basés sur XML correspondant aux *capabilities*, aux *exceptions* ou aux *featureInfo*, le format d'échange est l'image. Le nombre de formats d'image est immense et la couche abstraite s'est limitée à cinq d'entre eux, les plus courants :
– JPG
– JPEG
– GIF
– BMP
– PNG

La classe utilitaire *org.steamer.wms.map.Tools* permet d'instancier un objet Map à partir des paramètres d'une requête *GetMap*. L'objet Map gèrera les transformations géométriques permettant de passer de la géométrie d'une donnée géographique à la géométrie qui sera finalement dessinée sur la carte. Pour transformer un objet décrit depuis le repère géographique il faut effectuer trois transformations :
– une translation de l'origine de la BBox vers (0,0) ;
– une mise à l'échelle horizontale et verticale, faisant coïncider la taille de la BBox avec celle de l'image ;
– une symétrie axiale, car le point (0,0) se trouve en haut à gauche dans une image, et l'axe vertical est descendant (le repère d'une image est indirect).

La figure 4.8 décrit les repères intermédiaires correspondant à ces trois transformations. Les coordonnées passées en paramètres sont :
– xMin, yMin, xMax et yMax pour le paramètre BBOX ;
– width et height pour les dimensions de l'image.

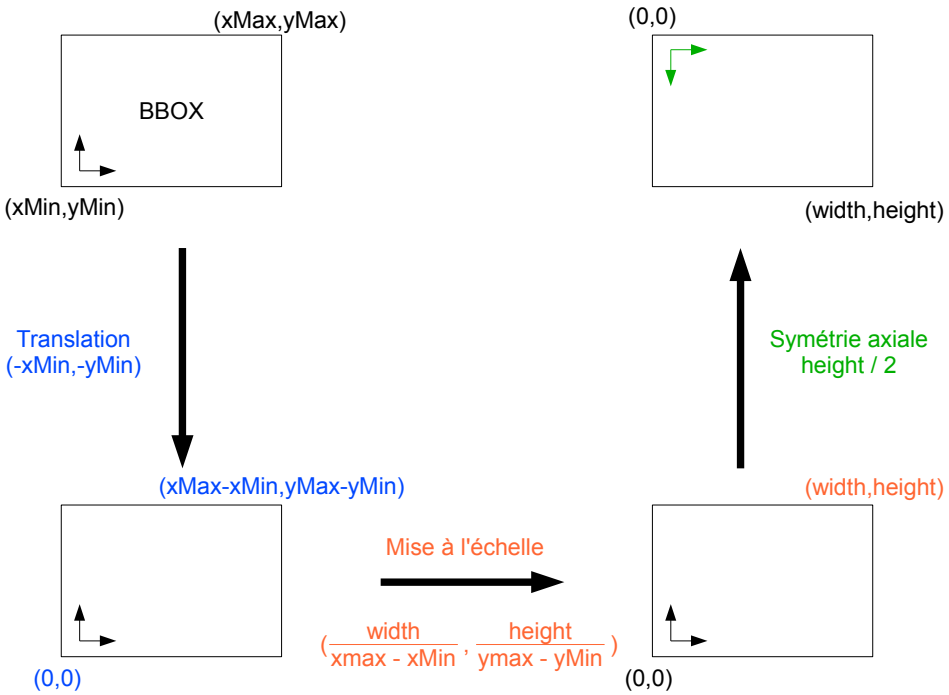


FIGURE 4.8 – Changement de repère : géographie vers image

La classe de la couche concrète du service n'a plus qu'à appeler les méthodes de Map en utilisant directement les géométries des entités géographiques qu'elle manipule, elle n'a pas à se soucier des

contraintes liées à l'image, qu'il s'agisse du format de l'image ou de sa taille.

Le tableau 4.4 décrit les méthodes proposées par Map qui ont été utilisées par le service HyperAtlasWMS, mais, comme Map encapsule la classe *java.awt.Graphics2D*, les possibilités sont beaucoup plus nombreuses que celles qui sont décrites...

Méthode	**Description**
init(MapRequest mapRequest)	initialisation (utilise FORMAT, BBOX, WIDTH et HEIGHT)
draw(Shape shape,Color color)	dessin d'un polygone (tracé de frontières)
fill(Shape shape,Color color)	remplissage d'un polygone (couleur d'un territoire, utilisé pour les cartes choroplèthes)
drawOval(int x, int y, int width, int height, Color color)	dessin d'un disque (cartes à disque)
fillOval(int x, int y, int width, int height, Color color)	remplissage d'un disque (cartes à disque)
setStroke(float thickness)	épaisseur du trait (les frontières des unités principales peuvent être un peu plus épaisses)
setComposite(AlphaComposite ac)	transparence

TABLE 4.4 – Méthodes de l'interface Map utilisées par HyperAtlasWMS

Une autre classe outil est la classe BBox. Elle est construite à partir des limites d'une zone géographique (paramètre BBOX par exemple). Elle offre de nombreuses fonctions permettant ensuite d'établir des relations entre cette zone et une image ou entre différentes zones géographiques.

Dans la suite, nous verrons que cette classe est utilisée par la couche HyperAtlasWMS, mais aussi par le client WMS. Voici quelques exemples :
– retrouver les coordonnées géographiques d'un point à partir des coordonnées d'un point d'une image (requête *GetFeatureInfo*) ;
– calculer la hauteur (resp. largeur) optimale pour une largeur (resp. hauteur) d'image donnée en respectant la correspondance entre échelle verticale et échelle horizontale (utilisé lors du zoom) ;
– calculer des intersections/unions de zones géographiques (utilisé lors du zoom).

4.2.3 Couche HyperAtlas

Exploitation de l'existant

Réutiliser le code existant de l'application HyperAtlas dans le cadre d'un service Web n'a pas été une tâche aisée, car l'application n'avait pas été conçue pour cela. Dans le cadre du WMS, il aurait été souhaitable de réutiliser la logique métier de l'application sans utiliser la logique de présentation :
– le modèle des données, ce qui a été possible ;
– l'établissement du contexte (sélection des unités territoriales d'une carte), ce qui n'a pas été possible ;
– les calculs de déviation, ce qui a été possible ;
– les calculs liés aux différentes façons de distribuer les résultats, ce qui n'a pas été possible.

Les difficultés rencontrées provinrent essentiellement de couplages forts entre les différentes logiques du projet. Ces couplages ont pour origine une utilisation abusive du patron de conception singleton [11]. L'absence de tests automatisés rendait les remaniements de code risqués.

La classe Settings est un exemple typique de l'anti-patron «*God Class*» [12] [Brown *et al.*, 1998], aggravé par le fait que c'est un singleton : un appel à cette classe, par ricochets, a pour conséquence d'instancier presque toutes les classes de l'application. Par exemple, dans le contexte du service, ouvrir un fichier de données *.hyp* produisait des erreurs (absence des fichiers de configuration de l'application) à cause d'un appel à Settings.getInstance(). Ceci a pour conséquence de créer des dépendances très fortes entre les différents composants de l'application. Remanier le code de cette classe (presque 3000 lignes) pour séparer les responsabilités et réduire les dépendances aurait représenté une quantité de travail incompatible avec les contraintes du projet. Notre travail s'est donc limité à utiliser la technique de l'inversion de dépendances [Martin, 2000] pour neutraliser cet appel lors de l'ouverture du fichier de données et à écrire des tests d'anti-régression pour prévenir la ré-introduction de dépendances lors d'évolutions futures d'HyperAtlas.

Les classes servant à stocker les données nécessaires à la création des cartes d'HyperAtlas sont contenues dans un paquetage *hypercarte.hyperatlas.serials*. Ce paquetage doit son nom au fait que toutes les classes de données qu'il contient implémentent l'interface Serializable. Le contenu de chacune des classes est créé lors de la *désérialisation* du fichier *hyp*. De plus, ces classes dérivent d'une classe SerialElement. Ceci a pour conséquence qu'un élément d'information d'HyperAtlas peut être retrouvé grâce à son code (chaîne de caractères) ou à son identifiant (nombre entier). La figure 4.9 montre ces héritages successifs.

Les principales classes structurant les données sont :

SerialUnitImpl : la classe centrale du modèle HyperAtlas, elle correspond à une entité géographique. C'est elle qui contient les éléments géographiques : géométries de points géolocalisés.

SerialStock : définit un indicateur, par exemple le nombre d'habitants.

SerialZoning : les unités sont organisées en hiérarchie. Une unité peut appartenir à plusieurs niveaux de maillage simultanément dans le modèle d'HyperAtlas (par exemple, Andorre sera prise en compte en tant que pays, région, *etc.*).

SerialArea : définit une aire d'étude. On pourra limiter l'étude d'un phénomène statistique à une partie seulement des entités géographiques, par exemple l'Europe des quinze, alors que le jeu de données contient l'Europe dans son ensemble.

Le tableau et la figure suivants montrent les relations entre ces différentes classes.

11. Voir annexe «Patron de conception singleton», page 123.
12. Une classe qui a tellement de responsabilités qu'elle référence et est référencée par de très nombreuses autres classes, ce qui a pour conséquence d'introduire un couplage fort dans l'architecture.

Méthode	Description
Geometry get_ outline()	Renvoie la géométrie associée
Point get_centroid()	Renvoie le point central
Hashtable<String, Float> get_stocks()	Renvoie les valeurs associées à un stock (clé : code du stock).
ArrayList<String> getAreaCodesList()	Renvoie la liste des codes des SerialArea comprenant l'unité
ArrayList<String> getZoningCodesList()	Renvoie la liste des codes des SerialZoning associés à l'unité

TABLE 4.5 – Principales méthodes de la classe SerialUnitImpl

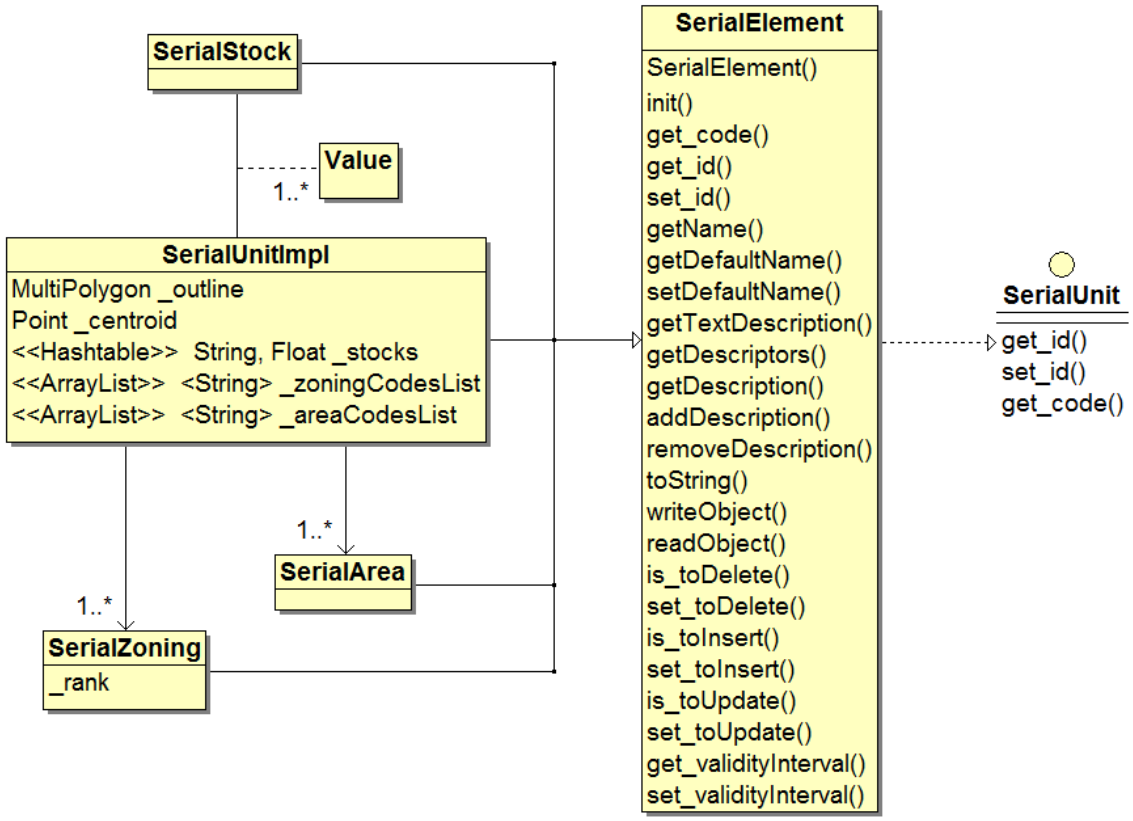


FIGURE 4.9 – Diagramme de la classe SerialUnitImpl

HierarchyRelations : stocke les relations de hiérarchie entre unités : elle permet, pour une unité donnée, de retrouver son unité parent ou ses unités filles, si elles existent.

Contiguity : définit une fonction de distance entre unités d'un même maillage et d'une même aire d'étude.

ContiguityRelations : stocke les valeurs pour une instance de Contiguity donnée sous forme d'une matrice.

Neighbourhood : cette classe sert à définir une notion de voisinage à partir d'une instance de Contiguity. Un voisinage est composé, en plus de cette instance d'un opérateur de comparaison et d'une valeur de comparaison. Par exemple, «à moins (opérateur de comparaison $\leqslant$) de six (valeur de comparaison 6.0) heures de camion (fonction de distance)». Les jeux de données définissent tous un voisinage par défaut, l'adjacence : les unités ayant une frontière en commun.

À l'ouverture du fichier toutes ces classes sont dé-sérialisées et contenues dans des collections, qui sont elles-mêmes contenues dans une classe appelée DataSerialver.

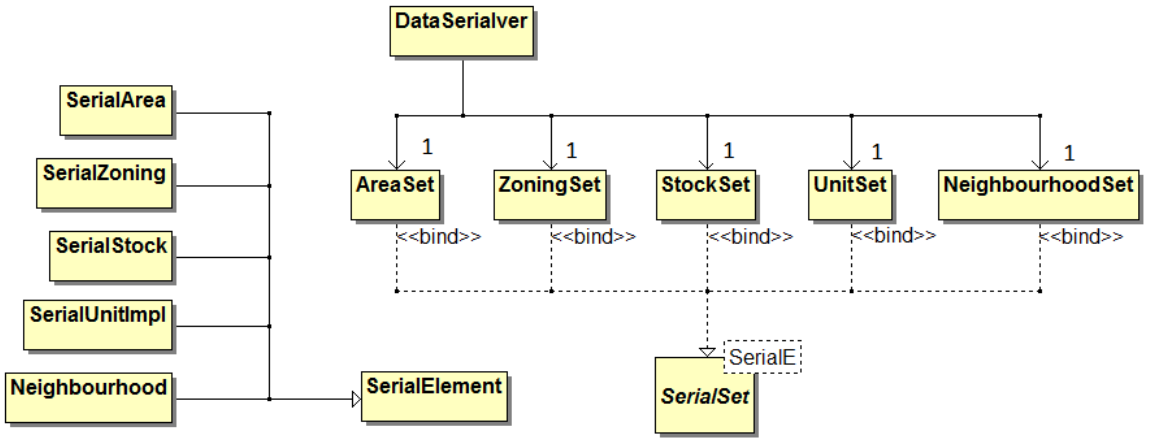


FIGURE 4.10 – Données des fichiers *.hyp

Le seul remaniement de code effectué dans cette partie de l'existant est la création de l'abstraction SerialSet, qui gère une collection d'objets SerialElement (voir la figure 4.10). Ce remaniement a permis de supprimer une grande quantité de code qui était dupliqué dans les classes collections (AreaSet, ZoningSet, StockSet, UnitSet et NeighbourhoodSet) et a été utile aussi dans la réécriture de la logique d'établissement du contexte HyperAtlas.

La sélection des unités territoriales a été réécrite, et est présentée dans la description de la requête GetMap, section 4.2.3, page 90.

Concernant les calculs de déviation, les algorithmes ont été extraits pour être regroupés dans une classe outil DeviationLogic qui ne contient que la logique de ces calculs.

La logique de distribution était trop fortement liée aux différents composants de l'application pour pouvoir être réutilisée simplement. Il faut préciser qu'il y a une différence importante entre l'utilisation du service et celle du logiciel HyperAtlas : pour un contexte donné, dans le cadre du service on ne doit en général créer qu'une carte, alors que l'application dessinera toutes les cartes. Cette différence implique que l'application doit effectuer beaucoup plus de travail lors de l'évolution du contexte. Le service est sans état, et doit au contraire n'effectuer que le strict minimum pour rester performant. Une requête *GetMap* sur la carte correspondant à l'aire d'étude ne fait aucun calcul, alors que dans l'application, cette carte est accompagné de toutes les autres cartes et doit donc effectuer tous les calculs nécessaires à leur établissement.

Les fonctions assurant le dessin des cartes n'ont pas pu être réutilisées : elles étaient fortement couplées aux composants graphiques chargés d'afficher ces cartes, ce qui les rendait incompatibles avec notre architecture, où une partie de cette tâche est déléguée à la couche WMSBase.

Ressources

La première requête au service, quelle qu'elle soit, a pour effet de charger les données depuis un fichier contenu dans le répertoire «*resources*» (voir figure 4.3). Ceci a pour conséquence de rendre la première requête plus longue que les autres.

Un service HyperAtlasWMS s'appuie sur deux types de fichiers *.hyp* ou GML. Il faut donc créer un service par fichier. Le fichier utilisé et le chemin de l'URL du service sont paramétrés lors de la

création de l'archive web (*.war*) à l'aide du fichier *build.properties*.

```
wms.hyperatlas.name=USA
wms.hyperatlas.build.home = ${basedir}/wms_hyperatlas_build
wms.hyperatlas.classes.dir = ${wms.hyperatlas.build.home}/WEB-INF/classes
wms.hyperatlas.lib.dir = ${wms.hyperatlas.build.home}/WEB-INF/lib
wms.hyperatlas.hypfile = usa.gml
wms.hyperatlas.resources.dir = ${wms.hyperatlas.build.home}/WEB-INF/resources
```

L'extrait précédent correspondra à un service dont l'URL sera :
http ://<nom du domaine> :<port>/USA/Hyperatlas :

nom du domaine : localhost pour un serveur Tomcat installé en local ;

port : 8080 par défaut sur Tomcat ;

USA : correspond à *wms.hyperatlas.name* dans l'extrait ;

Hyperatlas : chemin de la *Servlet*, indiqué dans le fichier *web.xml* (voir figure 4.3).

Le fichier de ressource du service sera le fichier *usa.gml*, qui correspond au paramètre *wms.hyperatlas.hypfile* de l'extrait.

Utiliser un format autre que les fichiers *.hyp* a été implémenté pour deux raisons :
- valider la généricité de la conception du service HyperAtlas, c'est-à-dire qu'il puisse être par la suite réutilisé avec des sources de données autres les fichiers *.hyp ;
- utiliser le service avec des systèmes de référence standard : les fichiers *.hyp* ne donnent pas d'indications sur le référentiel géospatial utilisé pour les géométries des entités géographiques. Si la norme OGC mentionne que c'est envisageable, dans les faits, les clients que nous avons testé ne gèrent pas ce cas. Le GML contenant des données sur les États-Unis avec un SRS standard, ce qui a permis de tester HyperAtlasWMS avec des clients WMS (voir figure 3.5).

Par la suite, l'origine des données est masquée pour l'ensemble des classes responsables de l'exécution du service (implémentations concrètes des interfaces Get*Provider de WMSBase). Ces classes utilisent deux interfaces, HCapabilitiesAdapter et HDataAdapter, qui permettent à la logique de traitement de n'utiliser que le modèle des données, sans tenir compte des aspects liés à leur persistance, ce que montre le diagramme suivant.

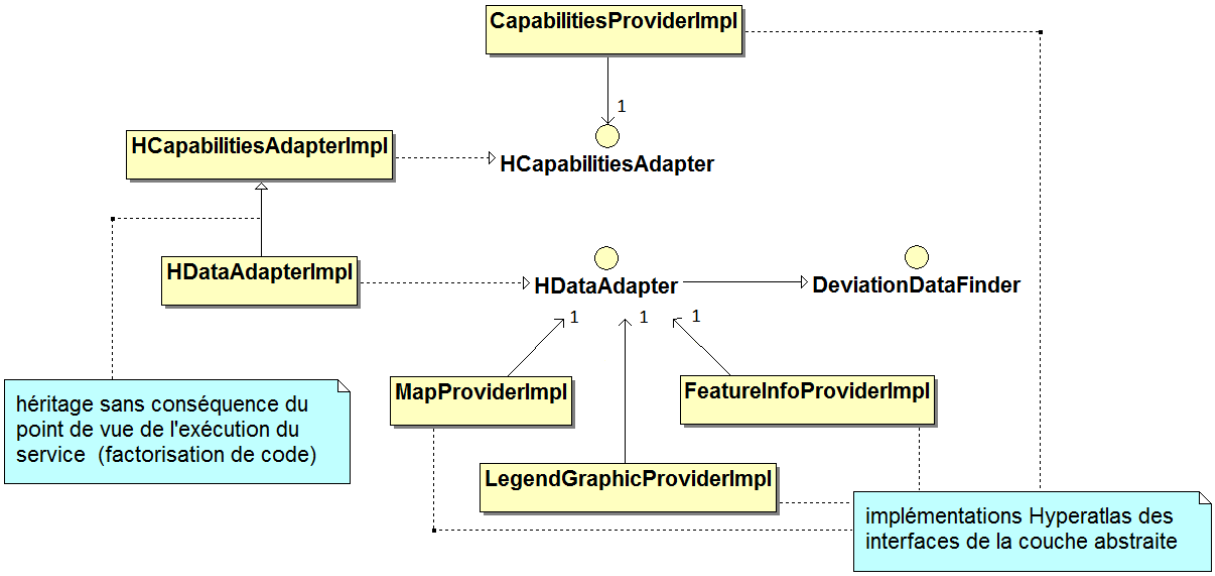


FIGURE 4.11 – Abstraction de la persistance des données

Les méthodes de ces deux interfaces seront détaillées dans les sections consacrées aux différentes requêtes du service. L'interface DeviationDataFinder, qui apparaît sur le schéma, a été créée dans

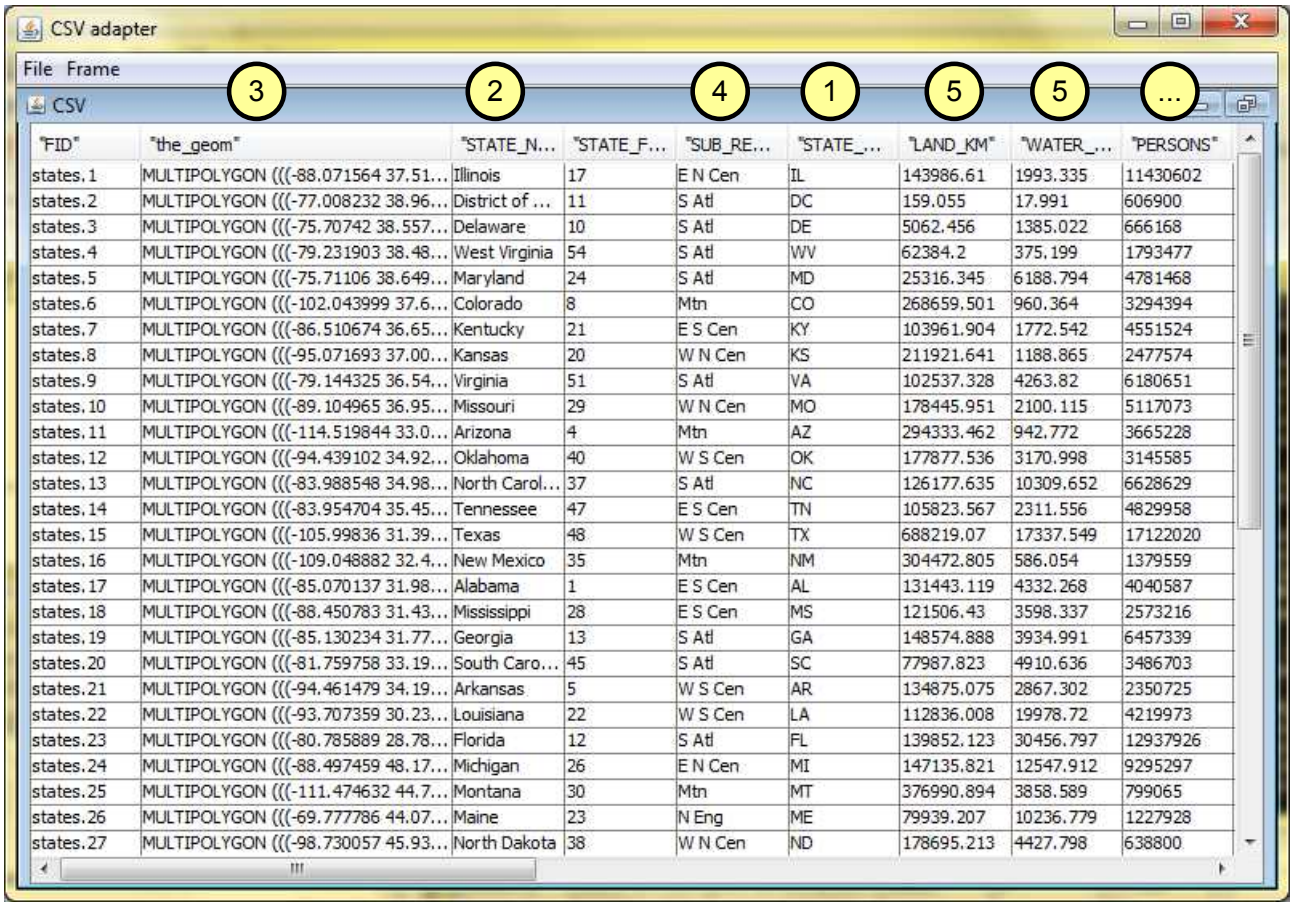
l'application HyperAtlas lors des remaniements de code effectués pour pouvoir réutiliser le code existant permettant le calcul des différentes déviations.

CSV adapter

Dans le cadre de l'itération 13 [13], nous avons développé un utilitaire appelé CSVAdapter qui permet, à partir d'un fichier **.csv* [14] de générer un fichier GML qui sera utilisé avec le service HyperAtlasWMS. On peut rapprocher cet utilitaire du logiciel HyperAdmin, qui permet de créer les fichiers *.hyp* [Plumejeaud, 2007] à partir de fichiers Excel et de fichiers MIF/MID.

Cet utilitaire peut utiliser un fichier csv quelconque, à condition que les lignes correspondent à des unités géographiques. Pour chaque ligne du fichier :
– une colonne doit identifier les unités de façon unique ;
– une colonne doit contenir la géométrie de l'unité ;
– une ou plusieurs colonnes doivent contenir des informations statistiques concernant l'unité.

Le processus de construction du fichier de données GML se fait en trois étapes :
– premièrement, l'ouverture d'un fichier **.csv*. L'interface graphique présente le contenu du fichier sous forme tabulaire, comme sur la figure 4.12 ;



"FID"	"the_geom"	"STATE_N...	"STATE_F...	"SUB_RE...	"STATE_...	"LAND_KM"	"WATER_...	"PERSONS"
states.1	MULTIPOLYGON (((-88.071564 37.51...	Illinois	17	E N Cen	IL	143986.61	1993.335	11430602
states.2	MULTIPOLYGON (((-77.008232 38.96...	District of ...	11	S Atl	DC	159.055	17.991	606900
states.3	MULTIPOLYGON (((-75.70742 38.557...	Delaware	10	S Atl	DE	5062.456	1385.022	666168
states.4	MULTIPOLYGON (((-79.231903 38.48...	West Virginia	54	S Atl	WV	62384.2	375.199	1793477
states.5	MULTIPOLYGON (((-75.71106 38.649...	Maryland	24	S Atl	MD	25316.345	6188.794	4781468
states.6	MULTIPOLYGON (((-102.043999 37.6...	Colorado	8	Mtn	CO	268659.501	960.364	3294394
states.7	MULTIPOLYGON (((-86.510674 36.65...	Kentucky	21	E S Cen	KY	103961.904	1772.542	4551524
states.8	MULTIPOLYGON (((-95.071693 37.00...	Kansas	20	W N Cen	KS	211921.641	1188.865	2477574
states.9	MULTIPOLYGON (((-79.144325 36.54...	Virginia	51	S Atl	VA	102537.328	4263.82	6180651
states.10	MULTIPOLYGON (((-89.104965 36.95...	Missouri	29	W N Cen	MO	178445.951	2100.115	5117073
states.11	MULTIPOLYGON (((-114.519844 33.0...	Arizona	4	Mtn	AZ	294333.462	942.772	3665228
states.12	MULTIPOLYGON (((-94.439102 34.92...	Oklahoma	40	W S Cen	OK	177877.536	3170.998	3145585
states.13	MULTIPOLYGON (((-83.988548 34.98...	North Carol...	37	S Atl	NC	126177.635	10309.652	6628629
states.14	MULTIPOLYGON (((-83.954704 35.45...	Tennessee	47	E S Cen	TN	105823.567	2311.556	4829958
states.15	MULTIPOLYGON (((-105.99836 31.39...	Texas	48	W S Cen	TX	688219.07	17337.549	17122020
states.16	MULTIPOLYGON (((-109.048882 32.4...	New Mexico	35	Mtn	NM	304472.805	586.054	1379559
states.17	MULTIPOLYGON (((-85.070137 31.98...	Alabama	1	E S Cen	AL	131443.119	4332.268	4040587
states.18	MULTIPOLYGON (((-88.450783 31.43...	Mississippi	28	E S Cen	MS	121506.43	3598.337	2573216
states.19	MULTIPOLYGON (((-85.130234 31.77...	Georgia	13	S Atl	GA	148574.888	3934.991	6457339
states.20	MULTIPOLYGON (((-81.759758 33.19...	South Caro...	45	S Atl	SC	77987.823	4910.636	3486703
states.21	MULTIPOLYGON (((-94.461479 34.19...	Arkansas	5	W S Cen	AR	134875.075	2867.302	2350725
states.22	MULTIPOLYGON (((-93.707359 30.23...	Louisiana	22	W S Cen	LA	112836.008	19978.72	4219973
states.23	MULTIPOLYGON (((-80.785889 28.78...	Florida	12	S Atl	FL	139852.123	30456.797	12937926
states.24	MULTIPOLYGON (((-88.497459 48.17...	Michigan	26	E N Cen	MI	147135.821	12547.912	9295297
states.25	MULTIPOLYGON (((-111.474632 44.7...	Montana	30	Mtn	MT	376990.894	3858.589	799065
states.26	MULTIPOLYGON (((-69.777786 44.07...	Maine	23	N Eng	ME	79939.207	10236.779	1227928
states.27	MULTIPOLYGON (((-98.730057 45.93...	North Dakota	38	W N Cen	ND	178695.213	4427.798	638800

FIGURE 4.12 – Colonnes du fichier USA.csv

13. Voir tableau 3.2 Itérations du projet, p.61.
14. Comma-separated values : un format informatique ouvert représentant des données tabulaires sous forme de «valeurs séparées par des virgules». On peut l'assimiler à une vue d'une base de données.

La seconde étape consiste à sélectionner des colonnes de la table selon le type d'informations qu'elle contiennent (noms, géométries, statistiques).

Cette sélection se fait par une succession de dialogues, celui de la figure 4.13 correspond à la sélection de la colonne qui contient le nom de l'unité géographique :
– un premier dialogue permet de sélectionner une colonne correspondant aux codes identifiants des entités géographiques (colonne 1 sur la figure 4.12) ;
– un second dialogue permet de sélectionner une colonne qui contient le nom des entités géographiques (colonne 2 sur la figure 4.12) ;
– un troisième dialogue permet de sélectionner une colonne correspondant aux géométries des entités géographiques (colonne 3 sur la figure 4.12). À l'aide de ces informations, l'outil construit une carte et une matrice contenant les distances entre unités ;
– un quatrième dialogue permet de sélectionner des colonnes correspondant à des aires d'études (colonne 4 sur la figure 4.12), c'est-à-dire des regroupements d'unités territoriales. Cette étape est optionnelle, l'outil construit une aire regroupant toutes les unités ;
– un dernier dialogue permet de sélectionner des colonnes correspondant à des indicateurs (colonnes 5 sur la figure 4.12).

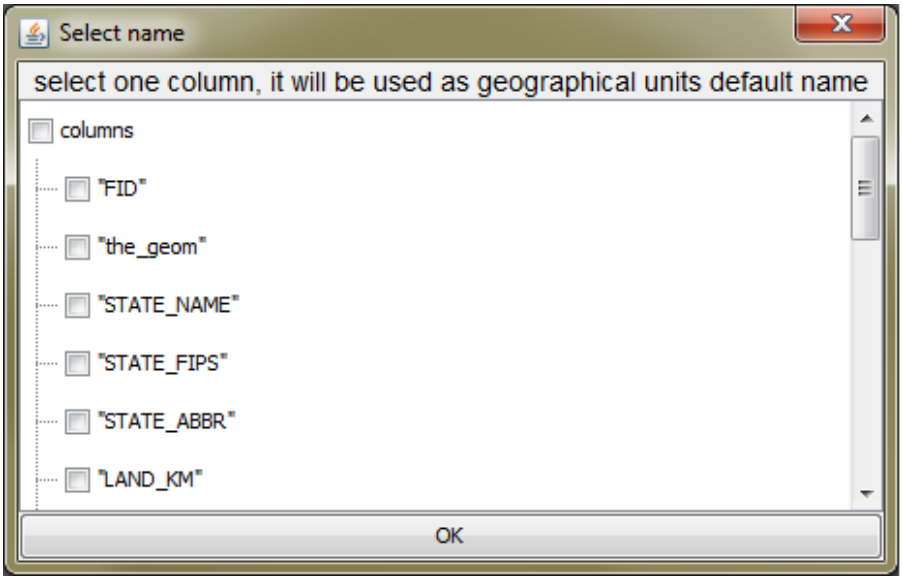


FIGURE 4.13 – Dialogue de sélection de colonnes

À la suite de ces sélections, l'utilitaire construit à l'aide des géométries :
– une carte ;
– une matrice des distances entre unités géographiques, si deux unités sont adjacentes, cette distance vaut zéro.

La troisième étape consiste pour l'utilisateur à raffiner les données dont il dispose, grâce aux différents éditeurs que montrent la figure 4.14.

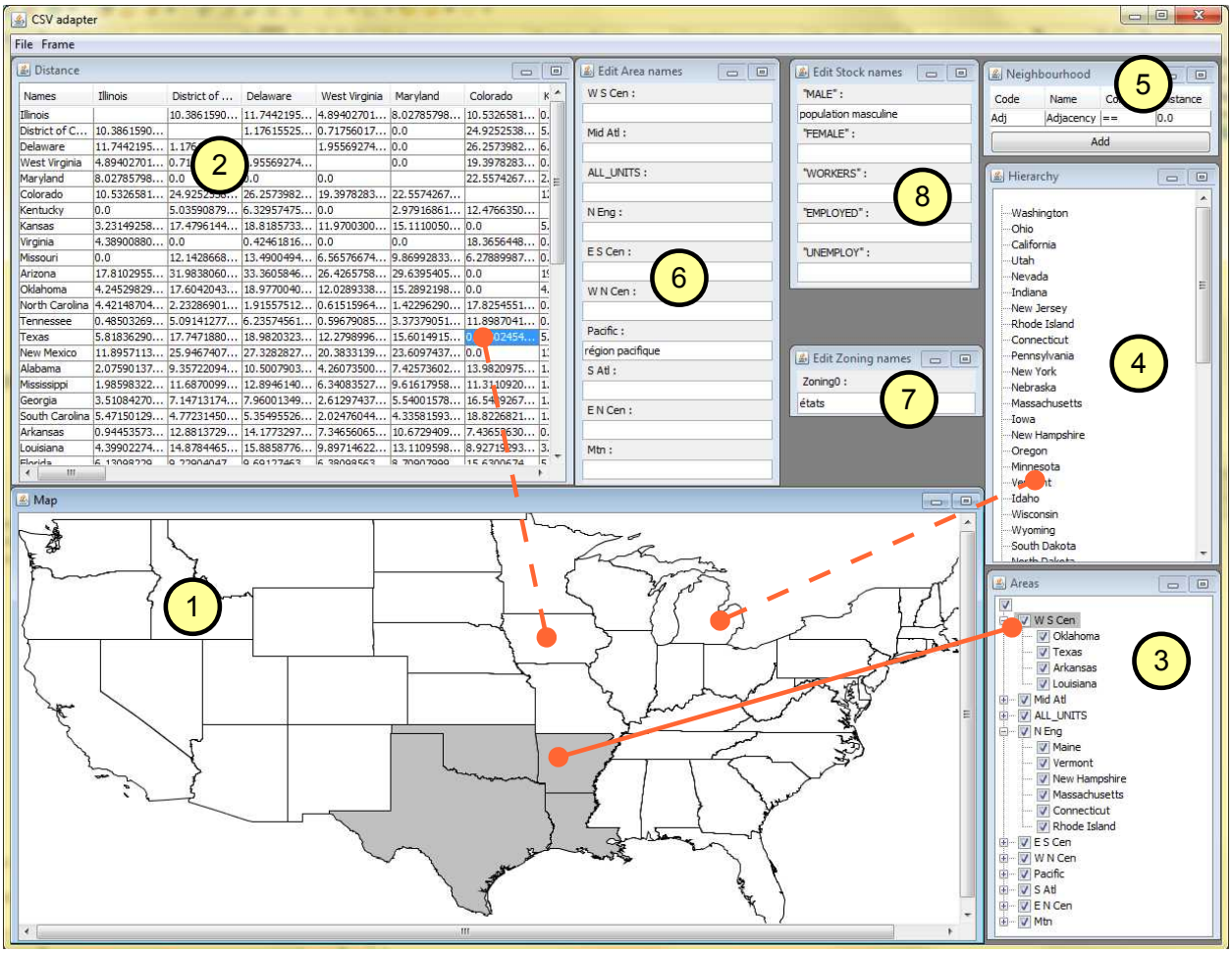


FIGURE 4.14 – Interface graphique de CSVadapter

1 : affiche la carte, la sélection d'un élément dans les écrans 2,3 et 4 grise les unités correspondant à la sélection (sur la figure 4.14, la sélection du nœud correspondant à la région «W S Cen») ;

2 : matrice des distances entre unités ;

3 : arborescence des aires d'études : l'utilisateur peut supprimer certaines aires, ou supprimer des unités d'une aire en cochant/décochant les nœuds correspondants ;

4 : arborescence des hiérarchies ;

5 : éditeur des voisinages, cet éditeur permet d'ajouter des relations de voisinage basées sur les distances, qui seront utilisées pour les cartes de déviation (voir 3.1). L'adjacence est ajoutée par défaut (distance = 0) ;

6 : éditeur des noms d'aires d'étude : l'utilisateur peut donner aux aires des noms plus significatifs ;

7 : éditeur des noms de maillages : l'utilisateur peut donner aux maillages des noms plus significatifs ;

8 : éditeur des noms d'indicateurs : l'utilisateur peut donner aux indicateurs des noms plus significatifs.

Lorsque l'utilisateur estime que les données sont prêtes, elles sont sauvegardées dans un fichier GML qui peut-être utilisé par un service HyperAtlasWMS.

Ce fichier est d'abord constitué d'une suite de triplets [15] décrivant les données autres que les unités territoriales :
– une liste d'aires d'études, décrites par un triplet Propriété correspondant au nom ;
– une liste de maillages, décrites par deux triplets : un pour le nom, l'autre pour le niveau du maillage ;
– une liste de stocks (indicateurs statistiques) décrits par leur nom ;
– une liste de voisinages, décrits par leur nom, le comparateur et une valeur pour la comparaison.

Après cette suite de définitions, on trouve les unités territoriales. Les triplets correspondent aux associations Stock/Valeur du stock, aux maillages et aux aires d'études auxquels l'unité appartient.

Le code XML suivant est extrait du fichier USA.gml :

```
<FeatureCollection>
  <!-- Aires d études -->
  <property type="CODE_AREA_NAME" typeName="Pacific">Côte Pacifique</property>
  ...
  <!-- Maillages des unités -->
  <property type="CODE_ZONI_NAME" typeName="Zoning0">States</property>
  <property type="CODE_ZONI_RANK" typeName="Zoning0">0</property>
  ...
  <!-- Stocks -->
  <property type="CODE_STOC_NAME" typeName=""PERSONS"">Nombre d habitants</
      property>
  ...
  <!-- Voisinages -->
  <property type="CODE_NEIG_NAME" typeName="Adj">Adjacency</property>
  <property type="CODE_NEIG_COMP" typeName="Adj">==</property>
  <property type="CODE_NEIG_DIST" typeName="Adj">0.0</property>
  ...
  <!-- Liste des unités -->
  <featureMember typeName="UNITS">
    <Feature identifier="CA" typeName="SerialUnitImpl">
      <name>California</name>
        <!-- Valeurs des stocks -->
        <property type="STOCK" typeName=""PERSONS"">4866692.0</property>
        <!-- Aires contenant les unités -->
        <property type="AREA" typeName="ALL_UNITS"/>
        <property type="AREA" typeName="Pacific"/>
        <!-- Maillage -->
        <property type="ZONING" typeName="Zoning0"/>
          <!-- Géométrie -->
          <geometricProperty>
            <MultiPolygon>
              <polygonMember>
                <Polygon srsName="EPSG:4326">
                <outerBoundaryIs>
                  <LinearRing>
                    <!-- Liste des coordonnées -->
                    <coordinates ts=" " cs="," decimal=".">
                        -122.40074899999999,48.22539499999999
                        -122.461586,48.228542000000004
                    ...
```

15. Les triplets Type/Nom de Type/Valeur sont une base des fichiers GML. Dans notre format, type correspond au type d'un objet, *typeName* correspond à un identifiant.

GetCapabilities

La première requête qu'effectue un client vers un service la requête GetCapabilities. C'est le fichier retourné par cette requête qui permet au client d'utiliser le service.

Le fichier *capabilities* du service HyperAtlasWMS est rempli par l'implémentation de l'interface CapabilitiesProvider de la couche abstraite : la classe CapabilitiesProviderImpl.

Ce fichier décrit les capacités du service, ainsi que les formats qui leur sont associés :
– *GetMap*, avec les formats images de la couche abstraite ;
– *GetFeatureInfo*, format GML 1.0 ;
– *GetLegendGraphic*, avec les formats images de la couche abstraite.

Les autres éléments du fichier *capabilities* dépendent de l'interface HCapabilitiesAdapter. Cette interface a deux rôles :

1. renseigner le client sur les couches supportées par le service. Selon les données, certaines couches ne sont pas supportées, par exemple pour la carte des USA, il n'y a pas de hiérarchie (toutes les unités correspondent à des états), donc la couche de la déviation médiane n'est pas supportée, et n'apparaît pas dans le fichier de *capabilities*. Le service offre 7 couches de données, correspondant aux cartes de l'existant (en majuscules, le nom de la couche tel qu'il doit apparaître dans la requête avec le paramètre LAYERS) :
   – STUDY : carte de l'aire d'étude ;
   – QUAN : carte à disques. Le service n'ayant pas d'état, il ne distingue pas le numérateur ou le dénominateur pour les cartes à disques ;
   – DIST : carte de distribution des ratios ;
   – GLOBAL : carte de déviation globale ;
   – MEDIUM : carte de déviation médiane ;
   – LOCAL : carte de déviation locale ;
   – SYNTHESIS : carte de synthèse.

   Pour chacune de ces couches, il y a 5 styles possibles, le style par défaut (*default*), qui ne dessine rien d'autre que la carte, et 4 styles ajoutant le dessin de la légende dans un coin de la carte (*topleft*, *topright*, *bottomleft* et *bottomright*). Ces styles peuvent sembler faire double emploi avec la requête *GetLegendGraphic*, mais, comme on a pu l'observer, la plupart des clients du marché ne gère pas cette requête ;

2. renseigner le client sur les paramètres *VendorSpecific*. De la même façon que pour les couches de données, certains paramètres n'apparaissent pas selon les données.

   Tous les paramètres ont le même schéma, toutefois certains d'entre eux peuvent être liés, par exemple le nombre de classes pour une distribution et la liste des couleurs à utiliser pour le coloriage d'une carte choroplèthe. Le service est robuste au fait qu'il puisse y avoir des incohérences entre les paramètres liés : la norme imposant que leur présence dans la requête n'est pas obligatoire, le client peut par exemple indiquer le nombre de classes sans mentionner la liste des couleurs. La distinction entre paramètres liés et paramètres libres doit être considérée plutôt comme une indication, pas comme un critère de validité de la requête. Par exemple dans notre client, l'interface synchronise l'éditeur du nombre de classes avec l'éditeur du nombre de couleurs. Cet exemple soulève un point important de la relation entre un service et son client : le service exécute, sans se soucier du bien fondé de la requête qu'on lui soumet, c'est au client de guider l'utilisateur. Un exemple classique est l'échelle d'une carte : rien dans la norme WMS n'impose que la taille de l'image soit proportionnelle à la taille de la région demandée. Le résultat d'une telle requête sera une carte déformée, pas une erreur. C'est à l'application cliente de prévenir l'envoi d'une telle requête.

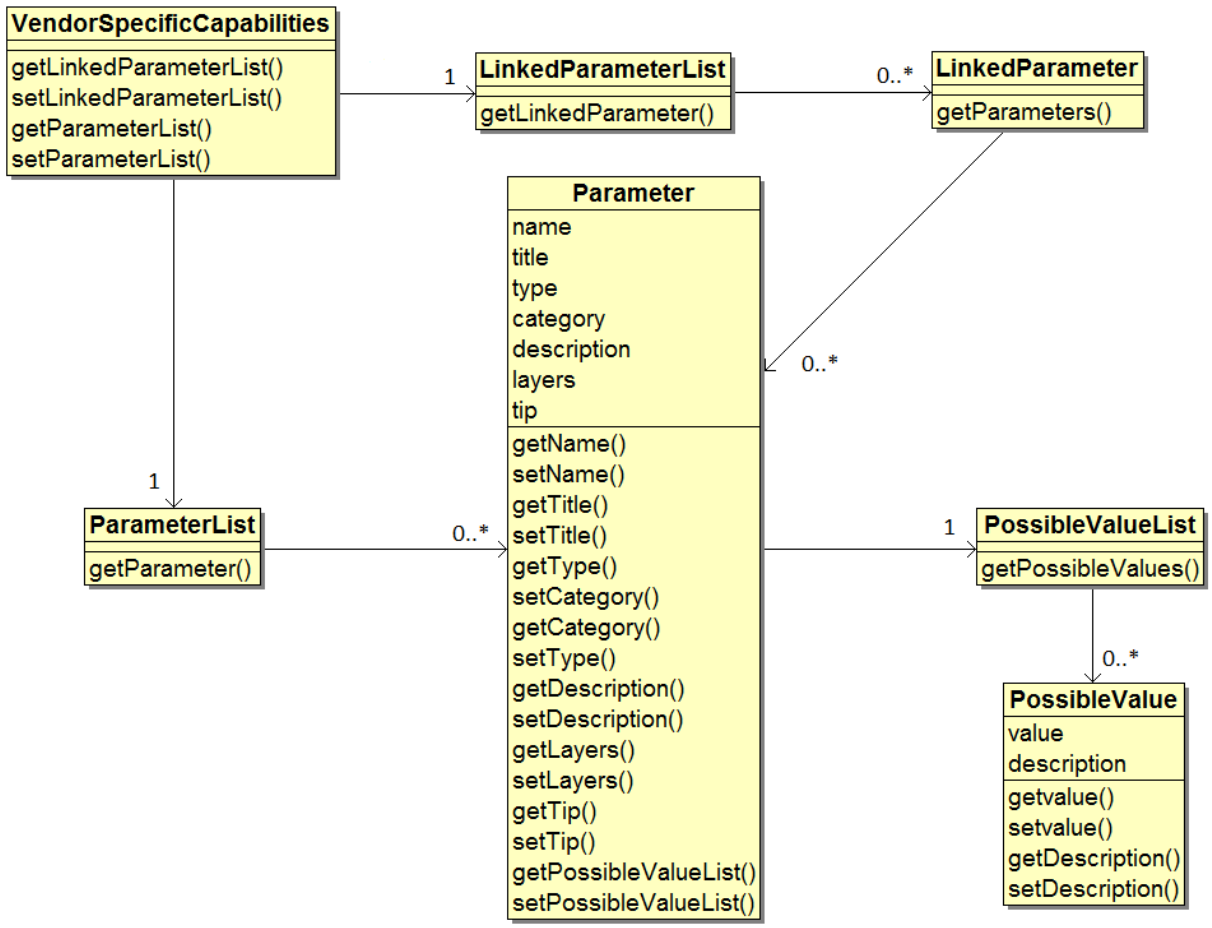


FIGURE 4.15 – Paramètres propriétaires du service (*VendorSpecific*)

Le diagramme 4.15 est la transcription en classes JAVA de la structure des paramètres VendorSpecific du service HyperAtlasWMS :

name : nom du paramètre, tel qu'il doit apparaître dans une requête ;

title : titre du paramètre, tel qu'il sera présenté à l'utilisateur ;

category : la catégorie du paramètre. Il y trois catégories de paramètres, «geo» pour les paramètres correspondant à des critères géographiques (par exemple maillage), «data» pour les informations affichées sur la carte (par exemple numérateur), et «style» pour les paramètres correspondant à des choix graphiques (par exemple couleur des disques) ;

type : le type du paramètre(par exemple enum, int, *etc.*). Le type est un type Java ;

description : description du paramètre ;

tip : indice sur la façon d'utiliser le paramètre ;

layers : la liste des couches pour lesquelles ce paramètre est utilisé ;

possibleValueList : liste de valeurs possibles pour un paramètre. Dans le cas d'une énumération (voir exemple suivant), la liste contient toutes les valeurs acceptables pour ce paramètre. S'il s'agit d'une valeur numérique, ces valeurs pourront correspondre au minimum et au maximum pour ce paramètre. Une valeur possible est un couple valeur/description. Si tous les éléments précédents sont communs à tous les services, les valeurs possibles proviennent du fichier de données utilisé par le service.

L'exemple suivant correspond au paramètre définissant l'aire d'étude de la carte, à partir de données d'une carte de l'Europe. La figure montre comment le client exploite les différentes informations pour proposer l'édition du paramètre, afin d'ensuite effectuer des requêtes vers le service.

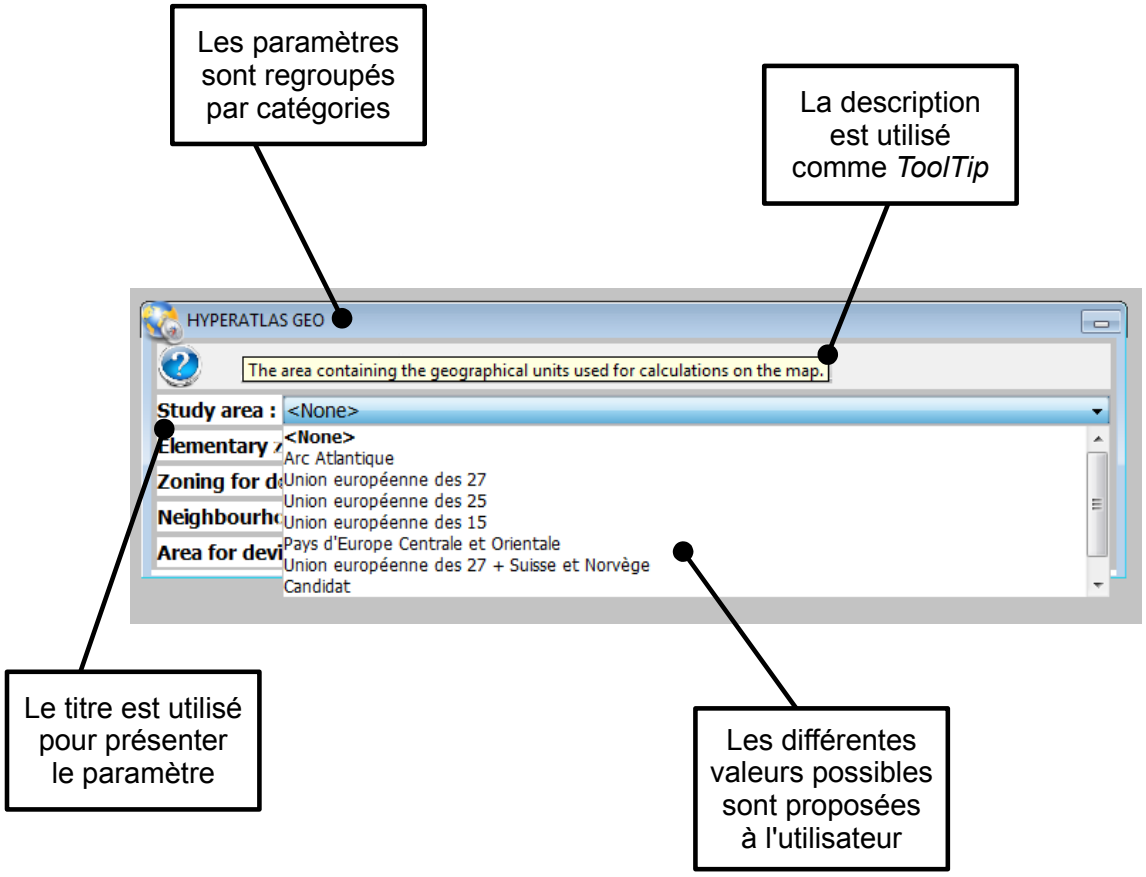


FIGURE 4.16 – Du XML à l'interface graphique

Le code XML correspondant au paramètre, à partir duquel le client a tiré ces informations.

```
<Parameter category="geo" type="enum">
  <Name>STUDYAREA</Name>
  <Title>Study area</Title>
  <Description>
    The area containing the geographical units used for calculations on the map.
  </Description>
  <Tip>pick one of the following values</Tip>
  <Layers>STUDY,QUAN,DIST,MEDIUM,LOCAL,GLOBAL,SYNTHESIS</Layers>
  <PossibleValueList>
    <PossibleValue description="Arc Atlantique" value="355"/>
    <PossibleValue description="Union européenne des 27" value="354"/>
    <PossibleValue description="Union européenne des 25" value="357"/>
    <PossibleValue description="Union européenne des 15" value="359"/>
    <PossibleValue description="Pays d'Europe Centrale et Orientale" value="358"
        />
    <PossibleValue description="Union européenne des 27 + Suisse et Norvège"
        value="361"/>
    <PossibleValue description="Candidat" value="362"/>
    <PossibleValue description="Roumanie et Bulgarie" value="360"/>
  </PossibleValueList>
```

```
</Parameter>
```

Pour finir, ce tableau rassemble la liste des paramètres et ce qu'ils représentent.

Paramètre	**Description**
STUDYAREA	Identifiant de l'aire d'étude.
ELEMENTARYZONING	Identifiant du niveau de maillage.
VAL1	Identifiant de l'indicateur utilisé comme numérateur ou comme quantité pour une carte à disque.
VAL2	Identifiant de l'indicateur utilisé comme dénominateur.
MEDIUMDEVIATION	Identifiant du niveau de maillage utilisé pour la déviation médiane.
LOCALDEVIATION	Identifiant du voisinage utilisé pour la déviation locale.
GLOBALDEVIATION	Identifiant de l'aire d'étude utilisée pour la déviation globale.
THRESHOLD	Seuil pour la carte de synthèse.
COLORS	Les couleurs correspondant aux classes. L'encodage est celui de l'OGC : 0xRRGGBB où RR (pour le rouge),GG (pour le vert) et BB (pour le bleu) est une valeur comprise entre 0 et 255 en hexadécimal (de 00=0 à FF=255). Les différentes couleurs sont séparées par des virgules. Tous les autres paramètres correspondant à des couleurs utilisent cet encodage.
CLASSNUMBER	Le nombre de classes (et donc de couleurs) pour une distribution. C'est une valeur entière comprise entre 2 et 10.
PROGRESSION	Type de progression (arithmétique ou géométrique) utilisée pour la distribution pour les cartes de déviation (globale, médiane ou locale).
QUANTILE	Quantile utilisé pour la distribution pour les cartes Ratio : numérateur, dénominateur ou le ratio.
THICK	Épaisseur du trait pour les frontières des unités principales.
QUAN_COLOR	Couleur des disques pour la carte à disque.
QUAN_SIZE	Taille des disques. La somme des surfaces des disques est égale à un pourcentage de l'aire totale de la carte. Une valeur entière entre 1 et 100.
BORDER	Couleur des frontières.
UNIT	Couleur de fond des unités lorsque la carte n'est pas choroplèthe.
BGUNIT	Couleur des unités en dehors de l'aire d'étude.
FONTN	Police de caractères utilisée pour la légende.
FONTS	Taille des caractères pour la légende.
LEG_COLOR	Couleur du fond pour la légende.

TABLE 4.6 – Paramètres *VendorSpecific* du service

GetMap

L'envoi d'une requête *GetMap* est géré par l'implémentation de l'interface MapProvider : la classe MapProviderImpl. La méthode GetMap de cette implémentation va construire un objet de type Style à l'aide des paramètres de la requête. La classe qui construit cet objet est la classe StyleFactory [16], dont le fonctionnement est décrit par la figure 4.17. L'interface Style permet de dessiner la carte grâce à sa

16. Patron de conception Fabrique [Gamma *et al.*, 1994].

méthode *draw(Map map)*, où map est une instanciation de la classe outil Map de la couche WMSBase. Il existe de nombreuses spécialisations de l'interface Style (voir figure 4.19), correspondant à des cartes ou à des algorithmes de dessin de cartes différents.

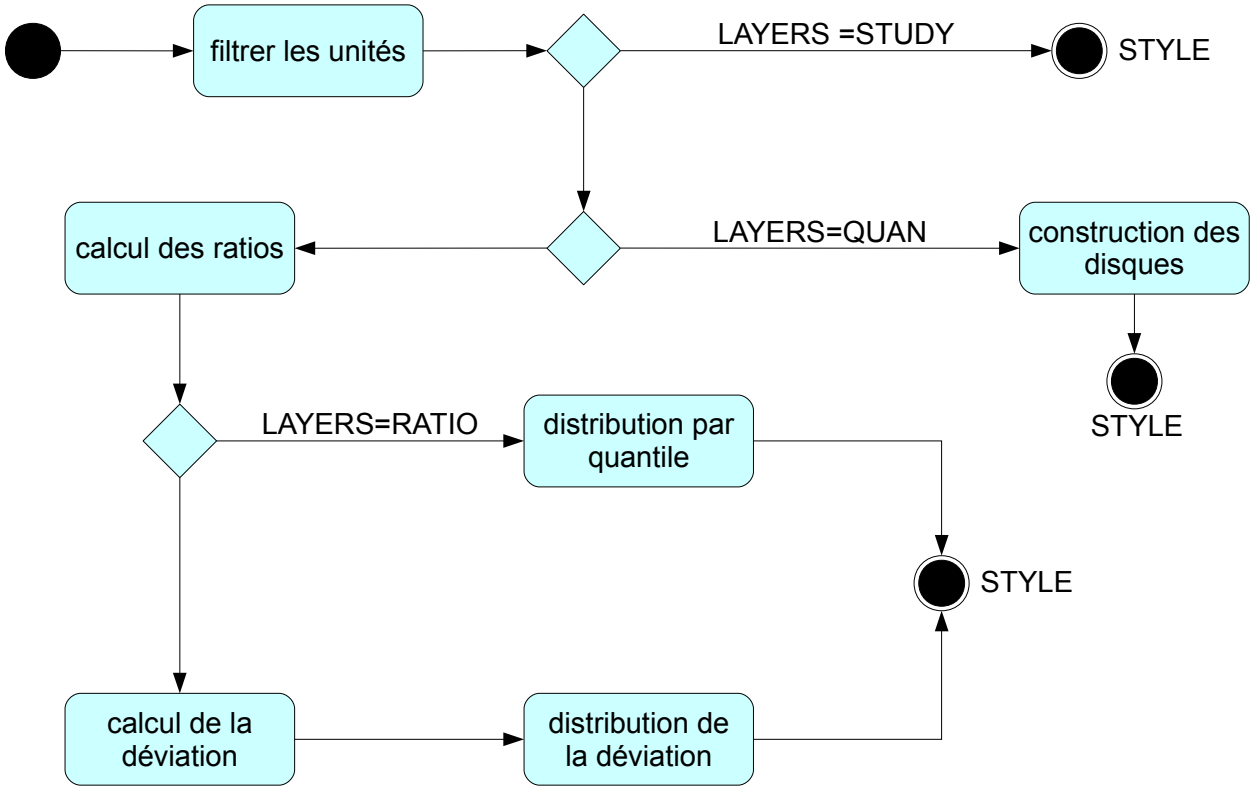


FIGURE 4.17 – Construction du style par la fabrique StyleFactory

La première étape est le filtrage des unités. De ce filtrage on obtiendra le contexte de la carte. L'interface en charge de filtrer les unités est l'interface générique Filter, qui n'a qu'une méthode, *accepts()*, qui correspond en fait à un prédicat sur un type d'objet quelconque :

```
public interface Filter<T> {
//interface générique, T peut-être de n'importe quel type

  boolean accepts(T object);
}
```

Sur le diagramme suivant, la classe AreaCodeFilter filtre les unités territoriales en fonction du code identifiant d'une aire d'étude, ZoningCodeFilter en fonction du code identifiant d'un maillage.

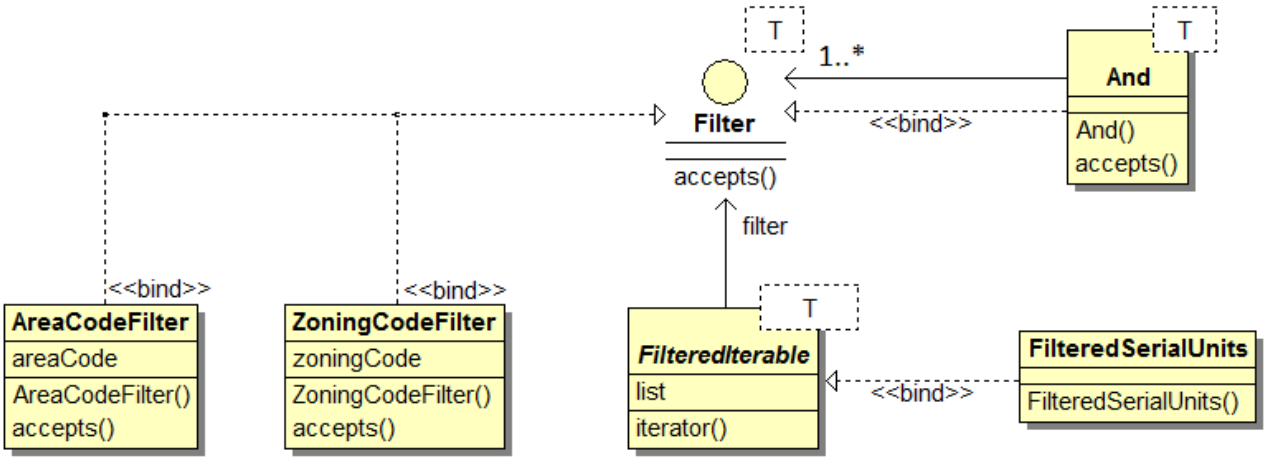


FIGURE 4.18 – Diagramme des classes implémentant Filter

On peut combiner les filtres comme des prédicats, c'est ce que fait And, qui est toujours du type Filter, et qui applique le ET logique sur une liste d'objets Filter.

```
public class And<T> implements Filter<T>{//classe générique, T peut-être de n'
    importe quel type

  protected Filter<T>[] filters ;

  public And(Filter<T>... filters) {//le constructeur prend un nombre indéfini de
      filtres
    this.filters = filters;
  }

  @Override
  public boolean accepts(T object) {
    for(Filter<T> f : filters){
      if(! f.accepts(object)){
        return false;//on parcourt l'ensemble des filtres, et si l'un d'entre
            eux renvoie FAUX, And rejette l'objet
      }
    }
    return true;//tous les filtres ont accepté l'objet T, And accepte l'objet
  }
}
```

On peut également appliquer un prédicat à une collection, et obtenir une sous-partie de cette collection. C'est ce que fait la classe abstraite FilteredIterable.

```
public abstract class FilteredIterable<T> implements Iterable<T> {

  protected Collection<T> list;//la collection à filtrer
  protected Filter<T> filter;//le filtre

  //itérateur sur la collection filtrée
  public Iterator<T> iterator() {
    return new FilteredIterator(list, filter);
  }

  private class FilteredIterator implements Iterator<T> {

    private final Iterator<T> state;
```

```
    private final Filter<T> filter;
    private boolean hasNextItem = false;
    private T nextItem = null;

    public FilteredIterator(Collection<T> coll, Filter<T> filter) {
      this.state = coll.iterator();
      this.filter = filter;
      findNext();//le premier élément qui sera renvoyé
    }

    public boolean hasNext() {//VRAI s'il y a encore des éléments
      return hasNextItem;
    }

    public T next() {//retourne le dernier élément trouvé par la méthode
       findNext
      T ret = nextItem;
      calcNext();
      return ret;
    }

    private void findNext() {//la méthode findNext parcourt la liste jusqu'à ce
       que le filtre accepte un élément. Cet élément sera retourné par la
       méthode next. La méthode hasNext retournera VRAI.
      hasNextItem = false;
      while (!hasNextItem && state.hasNext()) {
      T temp = state.next();
      if (filter.accepts(temp)) {
          nextItem = temp;
          hasNextItem = true;
        }
      }
    }
  }
}
```

Ce mécanisme est complètement générique (le type T des extraits de code n'est pas défini). Pour créer le contexte de la carte il suffit de spécialiser Filtered et FilteredIterable.

AreaCodeFilter n'accepte que les unités correspondant à une aire d'étude et ZoningCodeFilter celles qui correspondent à un maillage. Ces deux classes combinées avec And permettent de filtrer l'ensemble de toutes les unités territoriales pour le réduire au contexte de la carte.

```
public class AreaCodeFilter implements Filter<SerialUnitImpl>{

  String areaCode;
  public AreaCodeFilter(String areaCode) {
    this.areaCode = areaCode;
  }
  @Override
  public boolean accepts(SerialUnitImpl unit) {
    return unit.getAreaCodesList().contains(areaCode);
  }
}
```

```
public class ZoningCodeFilter implements Filter<SerialUnitImpl>{

  String zoningCode;
  public ZoningCodeFilter(String zoningCode) {
```

```
    this.zoningCode = zoningCode;
  }
  @Override
  public boolean accepts(SerialUnitImpl unit) {
    return unit.getZoningCodesList().contains(zoningCode);
  }
}
```

La classe FilteredSerialUnits est une spécialisation de la classe abstraite. Elle ne fait que lier (*«bind»* sur la figure 4.18, page 91) le type générique T au type du modèle de données SerialUnitImpl, la classe correspondant à une unité territoriale.

```
public class FilteredSerialUnits extends FilteredIterable<SerialUnitImpl> {
  public FilteredSerialUnits(Filter<SerialUnitImpl> filter, List<SerialUnitImpl>
      list){
    this.filter = filter;
    this.list = list;
  }
}
```

La méthode suivante est une méthode de l'interface HDataAdapter.

```
  @Override
  public Iterable<SerialUnitImpl> getIterable(Filter<SerialUnitImpl> filter) {
    return new FilteredSerialUnits(filter,DataSerialver.unitSet.getAllUnit());
  }
```

Une fois le filtrage des unités effectué, on peut créer le Style qui aura pour tâche de les dessiner. Le diagramme suivant expose les classes implémentant l'interface Style et est suivi par une description de leurs spécificités.

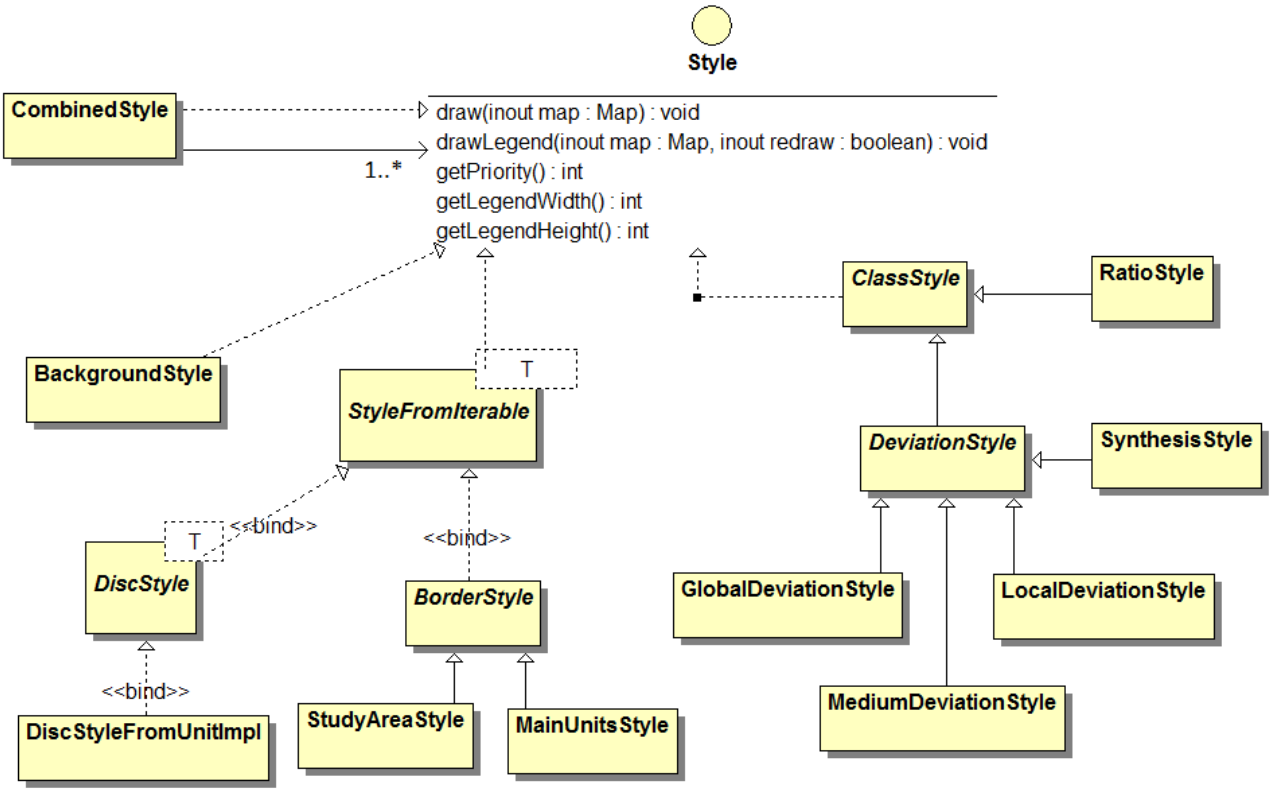


FIGURE 4.19 – Diagramme des classes implémentant Style

BackGroundStyle : dessine les unités territoriales en dehors de l'aire d'étude. Ce style gère également le dessin des cartouches [17], pour les cartes de l'Europe par exemple ;

StyleFromIterable : ce style abstrait va parcourir une collection d'objets et dessiner chaque élément de la collection selon un algorithme que spécifiera la spécialisation. Les implémentations de ce style abstrait devront surcharger les trois méthodes abstraites *beforeDraw*, *afterDraw* et *drawSingleUnit* ;

```
@Override
public void draw(Map map) {
        beforeDraw(map);
        Iterator<T> iterator = iterable.iterator();
        while(iterator.hasNext()){
                drawSingleUnit(map, iterator.next());
        }
        afterDraw(map);
}
protected abstract  void beforeDraw(Map map) ;//phase d'initialisation
protected abstract  void afterDraw(Map map) ;//phase finale
protected abstract void drawSingleUnit(Map map , T next);//dessin d'une
    unité
```

BorderStyle : dessine une unité sur la carte. Il faut d'abord remplir le fond, et dessiner la frontière ensuite. Ce style est une implémentation de StyleFromIterable et surcharge la méthode *drawSingleUnit* ;

```
@Override
protected void drawSingleUnit(Map map, SerialUnitImpl singleUnit) {
        if(singleUnit.hasGeometry()){
                Area a = singleUnit.getArea();//Area : géométrie associée
                draw(map, a);//dessin de la géométrie associée
        }
}

protected void draw(Map map, Area area) {
        if(unitBackgroundColor != null)
                map.fill(area, unitBackgroundColor);//on colorie le fond si
                    la couleur \textit{unitBackgroundColor} est définie
        if(borderColor != null)
                map.draw(area, borderColor);//dessin des frontières
}
```

MainUnitsStyle : dessine les frontières des unités principales de l'aire d'étude avec un trait qui peut être plus épais (unitBackgroundColor vaut null dans l'extrait de code précédent), en fonction du paramètre THICK. Ce style surcharge BorderStyle et surcharge les phases d'initialisation et finale en modifiant l'épaisseur du trait ;

```
@Override
protected void beforeDraw(Map map) {
        if(thick > 1)//thick : épaisseur du trait
                map.setStroke(thick);
}
@Override
protected void afterDraw(Map map) {
        map.resetStroke();
}
```

17. Un cartouche est une zone encadrée qui peut contenir la légende ou une région éloignée (les DOM-TOM pour une carte de la France).

StudyAreaStyle : dessine la carte correspondant à l'aire d'étude, la couleur du fond correspond au paramètre UNIT, et la couleur des frontières au paramètre BORDER (ce paramètre est utilisé pour le dessin des frontières, quel que soit le style) ;

DiscStyle, DiscStyleFromUnitImpl : dessine des disques proportionnellement à une quantité associée, les disques sont centrés sur l'unité territoriale ;

ClassStyle, DeviationStyle ou RatioStyle : tous les styles dérivant de ClassStyle correspondent aux cartes choroplèthes. Le principe est commun, il s'agit de construire une collection (classe QualifiedSet) à partir des unités territoriales. Tout d'abord on filtre celles correspondant au contexte (aire d'étude et maillage). On calcule le ratio val1/val2 quand c'est possible : les unités pour lesquelles une valeur n'est pas définie ou celles pour lesquelles val2 vaut 0 sont stockées à part. À ce stade, si la carte est une carte Ratio, on répartit les ratios par quantile. Dans le cas où il s'agit d'une carte de déviation, on calcule la déviation pour chaque unité. À la fin de ces calculs on obtient un intervalle, que l'on divisera en un nombre d'intervalles équivalent au paramètre CLASS. Le style parcourt ensuite les unités de l'objet QualifiedSet en demandant l'index de l'intervalle, qu'il associe à une couleur. La figure 4.20 reprend les différents éléments de cette construction, et quels sont les paramètres *VendorSpecific* intervenant lors de cette construction ;

GlobalDeviationStyle, MediumDeviationStyle, LocalDeviationStyle ou SynthesisStyle : sont des spécialisations de ClassStyle. Elles ne diffèrent que dans la façon dont elles calculent les déviations ;

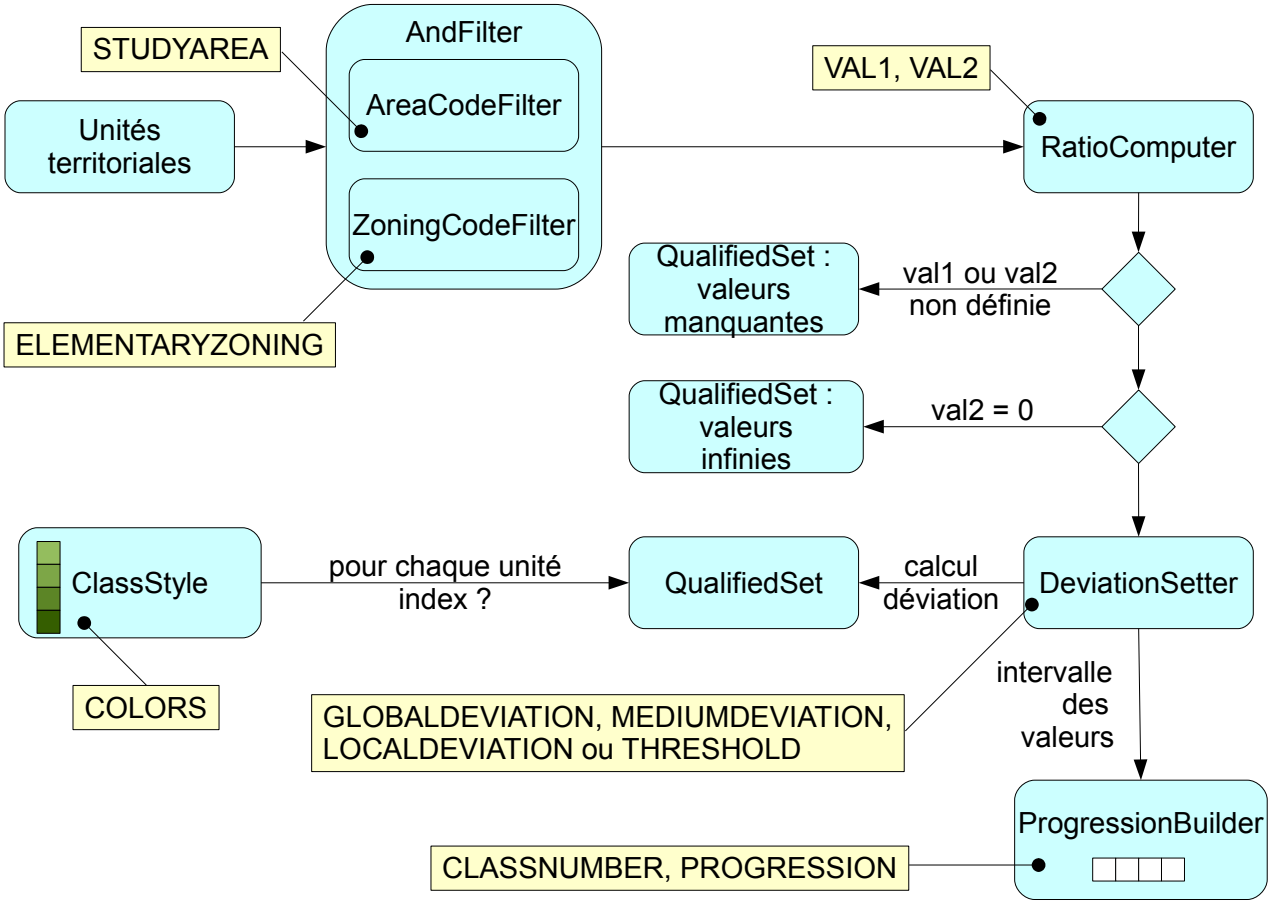


FIGURE 4.20 – Style ClassStyle

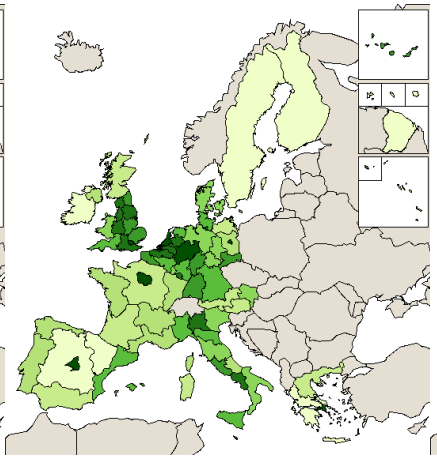
CombinedStyle : ce style n'a pas de capacités de dessin à proprement parler, mais il est une combinaison de plusieurs autres objets de type Style. La méthode *draw(Map map)* va appeler succes-

sivement les méthodes *draw(Map map)* de chacun des objets de type Style, par ordre de priorité, de la plus faible à la plus forte. Ceci aura pour résultat d'écraser les détails correspondant aux priorités plus faibles.

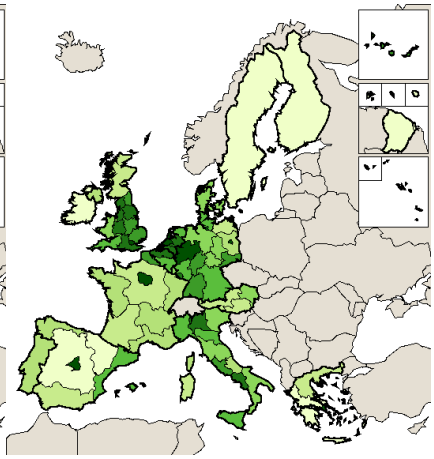
La méthode *GetMap* va donc construire un style combinant (CombinedStyle) les différentes couches de la requête. Lorsque l'analyse de la requête est terminée, la carte est dessinée à travers l'appel à *draw(Map map)*. Si le paramètre de la requête STYLES le spécifie, les légendes sont dessinées. Le résultat est une image de la carte, qui est renvoyée au client avec l'objet réponse.

La figure 4.21 montre les étapes du dessin d'une carte par un style combiné.

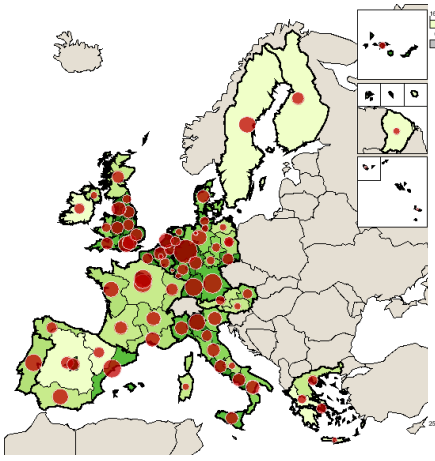
(a) BackGroundStyle : dessin du fond de la carte



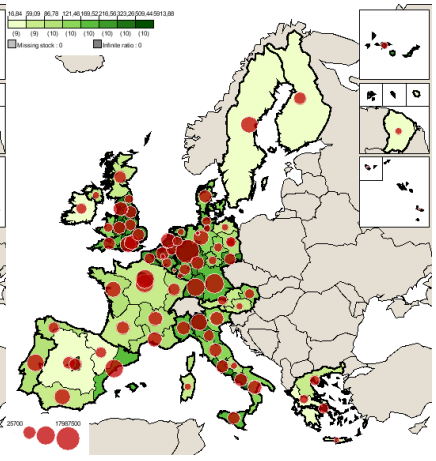
(b) RatioStyle : dessin des unités territoriales



(c) MainUnitsStyle : dessin des frontières principales



(d) DiscStyleFromUnitImpl : dessin des disques



(e) Ajout des légendes

FIGURE 4.21 – Combinaison des styles

Cette carte correspond à la requête HTTP suivante.

http ://localhost :8080/Europe/Hyperatlas ?SERVICE=WMS&REQUEST=getmap

adresse du service *type de la requête*

&VERSION=1.1.1 *version* &SRS=EPSG :4326&BBOX=-2217174,-1723801,1783333,2518193 *paramètres géographiques (référentiel et limites))*

&FORMAT=image/png&WIDTH=538&HEIGHT=571
paramètres de l'image (format, largeur et hauteur)

&LAYERS=DIST,QUAN&STYLES=topleft,bottomleft
couches et styles (légendes)

&STUDYAREA=359&ELEMENTARYZONING=73 *contexte HyperAtlas* &VAL1=437&VAL2=438 *indicateurs*

&CLASSNUMBER=8&COLORS=0xF0FFC8,0xC8EB8C,0xB4E178,0x8CD75A,0x5ABE3C ...
nombre et couleurs des classes de la distribution

&QUAN_SIZE=25 *disques* &FONTS=8&LEG_COLOR=0xFFFFFF *style de la légende* &THICK=2.5&BORDER=0x000000 *frontières*

&BGCOLOR=0xFFFFFF&BGUNIT=0xE5DFD3&QUAN_COLOR=0xC00000
autres couleurs : fond et disques

GetFeatureInfo

La requête WMS *GetFeatureInfo* reprend les mêmes mécanismes que la requête *GetMap*. La classe FeatureInfoFactory (qui est l'équivalent de StyleFactory) fabrique un objet FeatureInfo qui a pour charge de remplir la réponse avec un format GML. Les GML produit par une spécialisation de FeatureInfo ont en commun la géométrie de l'unité territoriale. Ce qui les différencie sont les propriétés associées à cette géométrie : seuls les résultats des calculs associés à la couche pour laquelle on effectue la requête (paramètre WMS «QUERY_LAYERS», qui est un paramètre propre à *GetFeatureInfo*) sont ajoutés au fichier GML. Par exemple, il n'y a aucun calcul pour la couche STUDY, seules figureront les informations comme le nom de l'unité, alors que si la couche est SYNTHESIS, tous les calculs sont faits, et donc tous les résultats seront ajoutés au GML. Le tableau suivant récapitule les équivalences entre Style et FeatureInfo.

LAYER	**Style**	**FeatureInfo**
STUDY	StudyAreaStyle	FeatureInfoFromSerialUnitImpl
QUAN	DiscStyleFromUnitImpl	QuantitativeFeatureInfo
DIST	RatioStyle	RatioFeatureInfo
GLOBAL	GlobalDeviationStyle	GlobalDeviationFeatureInfo
MEDIUM	MediumDeviationStyle	MediumDeviationFeatureInfo
LOCAL	LocalDeviationStyle	LocalDeviationFeatureInfo
SYNTHESIS	SynthesisStyle	SynthesisFeatureInfo

TABLE 4.7 – Équivalences Style et FeatureInfo

Le diagramme suivant, qui montre les héritages entre ces classes, est à comparer avec le diagramme 4.19 Diagramme des classes implémentant Style, page 93.

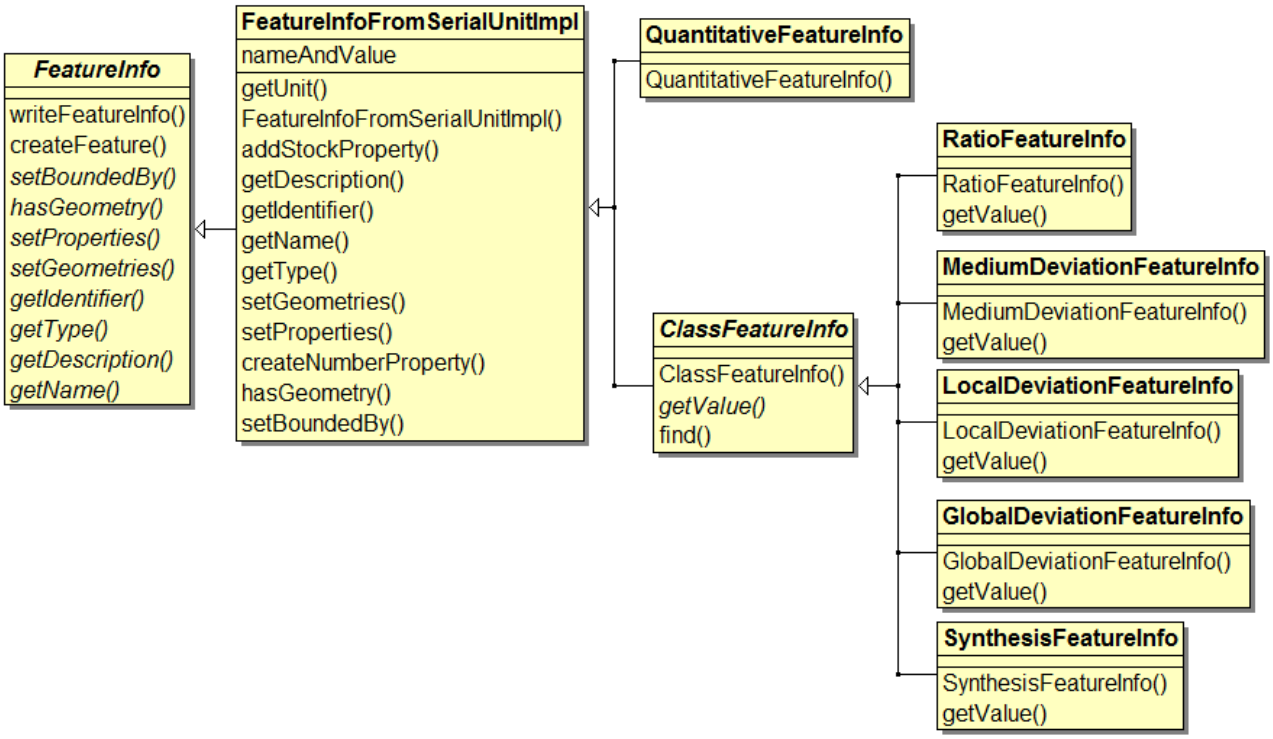


FIGURE 4.22 – Diagramme de classes FeatureInfo

La responsabilité de faire correspondre les coordonnées d'un point de l'image (paramètres WMS X et Y) avec un point géographique est déléguée à la classe BBox [18] de la couche abstraite. Il suffit ensuite de rajouter au filtre du contexte un filtre par rapport à ce point pour obtenir l'unité territoriale correspondant à la requête.

GetLegendGraphic

L'équivalence entre la requête *GetLegendGraphic* et la requête *GetMap* est encore plus immédiate. Les mécanismes mis en jeu sont exactement les mêmes. Cela peut sembler paradoxal de devoir recalculer toute la carte pour simplement dessiner sa légende, mais les légendes d'*HyperAtlas* ne sont pas seulement des codes de couleurs, elles contiennent des informations nécessitant que tous les calculs soient faits.

Cette richesse d'informations nécessite que lors d'une requête *GetLegendGraphic*, en plus des noms des paramètre WMS correspondant à la couche demandée (LAYER et non pas LAYERS ou QUERY_LAYERS) et au format (FORMAT), qui d'après la spécification WMS sont les seuls obligatoires, la requête devra comporter les paramètres optionnels de la norme WMS WIDTH, HEIGHT et SCALE, qui correspondent à la taille de l'image de la carte et à son échelle.

Avec ces trois paramètres, la classe outil BBox permet de simuler une requête *GetMap* avec un style dessinant la légende. Ensuite l'image est découpée afin de ne contenir que la partie contenant la légende. Ce découpage est effectué à l'aide des méthodes *getLegendWidth* et *getLegendHeight* de l'interface Style.

18. Voir 4.2.2 classes outils, page 77.

La figure suivante montre cette opération.

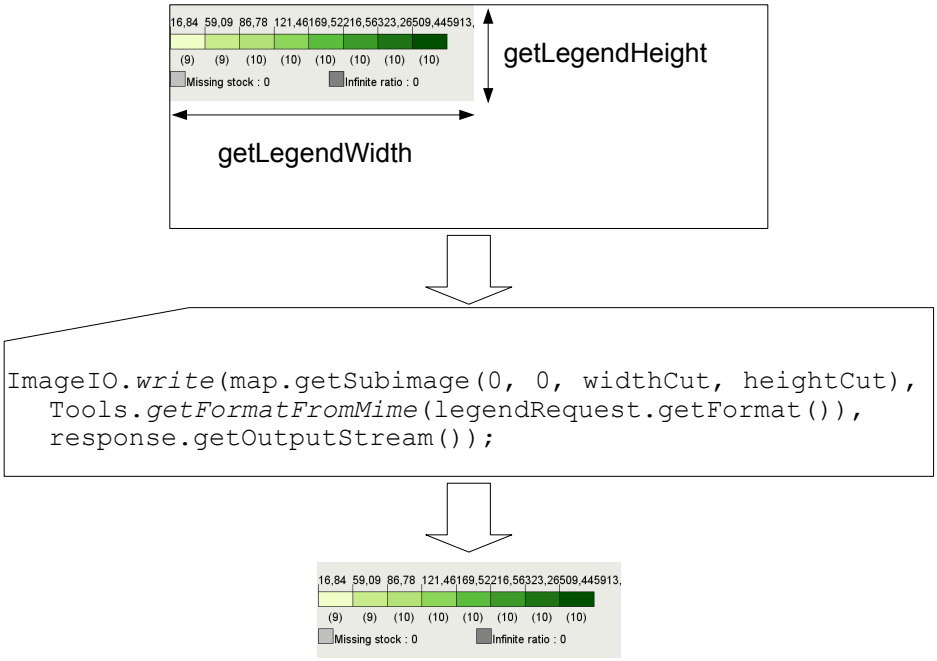


FIGURE 4.23 – Dessin de la légende

4.3 Client : BrowserWMS

Pour mieux comprendre le fonctionnement de BrowserWMS, on peut établir une analogie entre son utilisation et celle d'un site de commerce électronique avec un navigateur Web.

Lorsqu'un consommateur visite un site de commerce électronique, il commence par se connecter au site à l'aide de l'URL du site. Ensuite, le consommateur consulte le catalogue. Lorsqu'un article lui plait, il l'ajoute à son panier, après l'avoir éventuellement regardé de plus près : zoom sur la photo de l'article, consultation des caractéristiques techniques. Lorsqu'il a terminé de sélectionner ses articles, il ouvre son panier, ne conserve que les articles qu'il compte effectivement acheter, et effectue le règlement de ses achats.

BrowserWMS propose la même chose, mais les articles sont remplacés par le résultat des requêtes WMS : *GetMap*, *GetFeatureInfo* et *GetLegendGraphic*. L'utilisateur consulte les cartes/informations/-légendes en éditant les paramètres du service. Il peut zoomer sur la carte et obtenir des informations sur un point de la carte. Il peut ajouter le résultat d'une requête à son panier entre deux requêtes, et lorsqu'il le souhaite il ouvre son panier, et utilise les cartes/informations/légendes en créant un rapport, ce qui est analogue à l'achat.

La figure suivante permet de visualiser l'analogie entre l'interface du site commercial avec un navigateur Web et BrowserWMS.

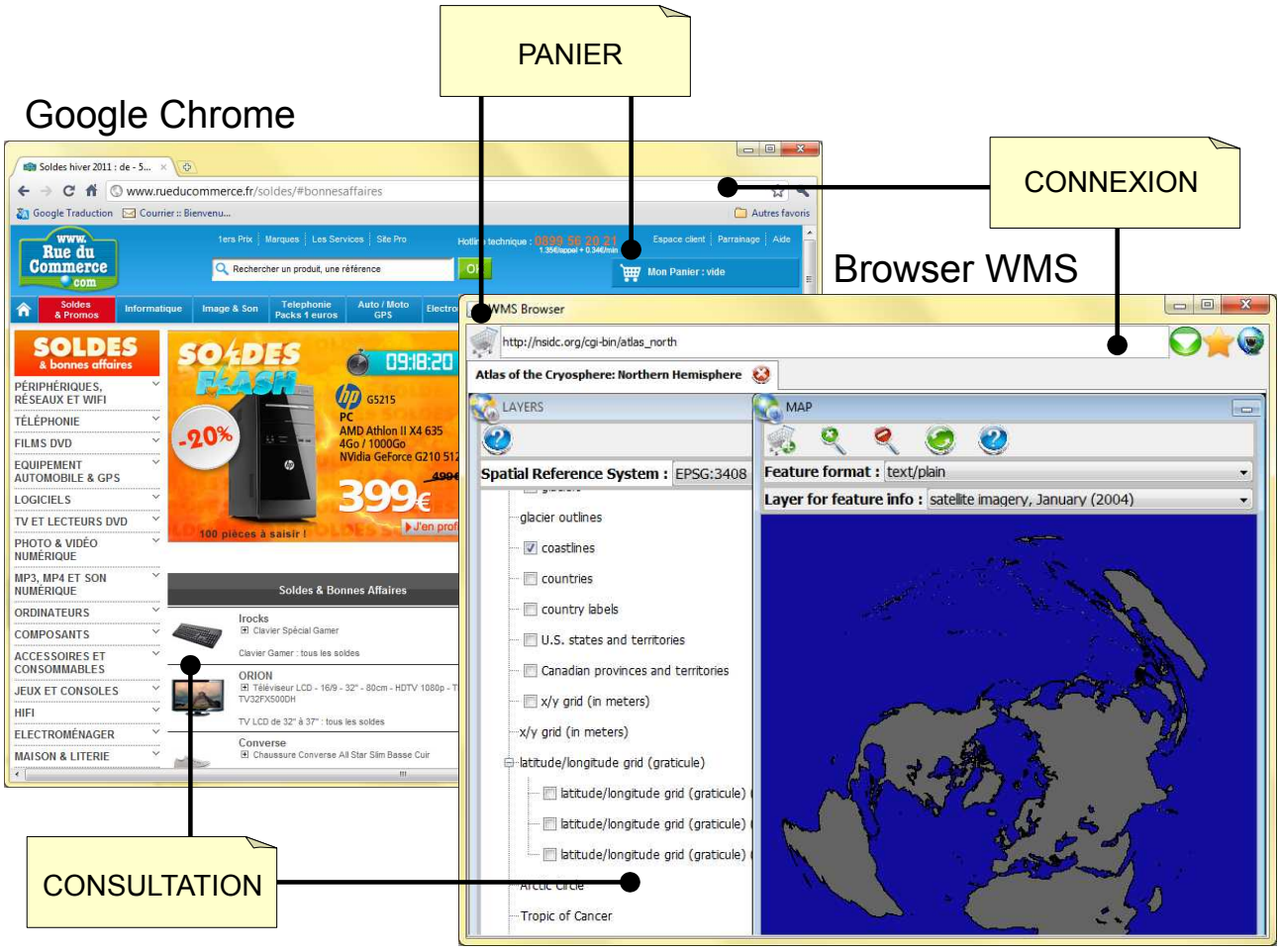


FIGURE 4.24 – Interfaces similaires

4.3.1 Architecture

Le patron de conception Modèle-Vue-Contrôleur [19] (MVC) est omniprésent dans l'architecture de BrowserWMS :

Modèle : gère la logique métier. Dans l'application BrowserWMS, le modèle est divisé en deux sous-modèles, l'un correspondant à la connexion, et l'autre aux données ;

Vue : transmet au contrôleur les entrées de l'utilisateur ;

Contrôleur : gère les interactions entre la vue et le modèle.

BrowserWMS est découpé en trois modules, chacun correspondant aux cas d'utilisation de la proposition. De plus, il peut héberger des greffons dédiés à certains services WMS, comme HyperAtlasWMS, par exemple.

Le diagramme suivant présente l'arborescence des paquetages de BrowserWMS, que nous allons détailler dans la suite de ce chapitre.

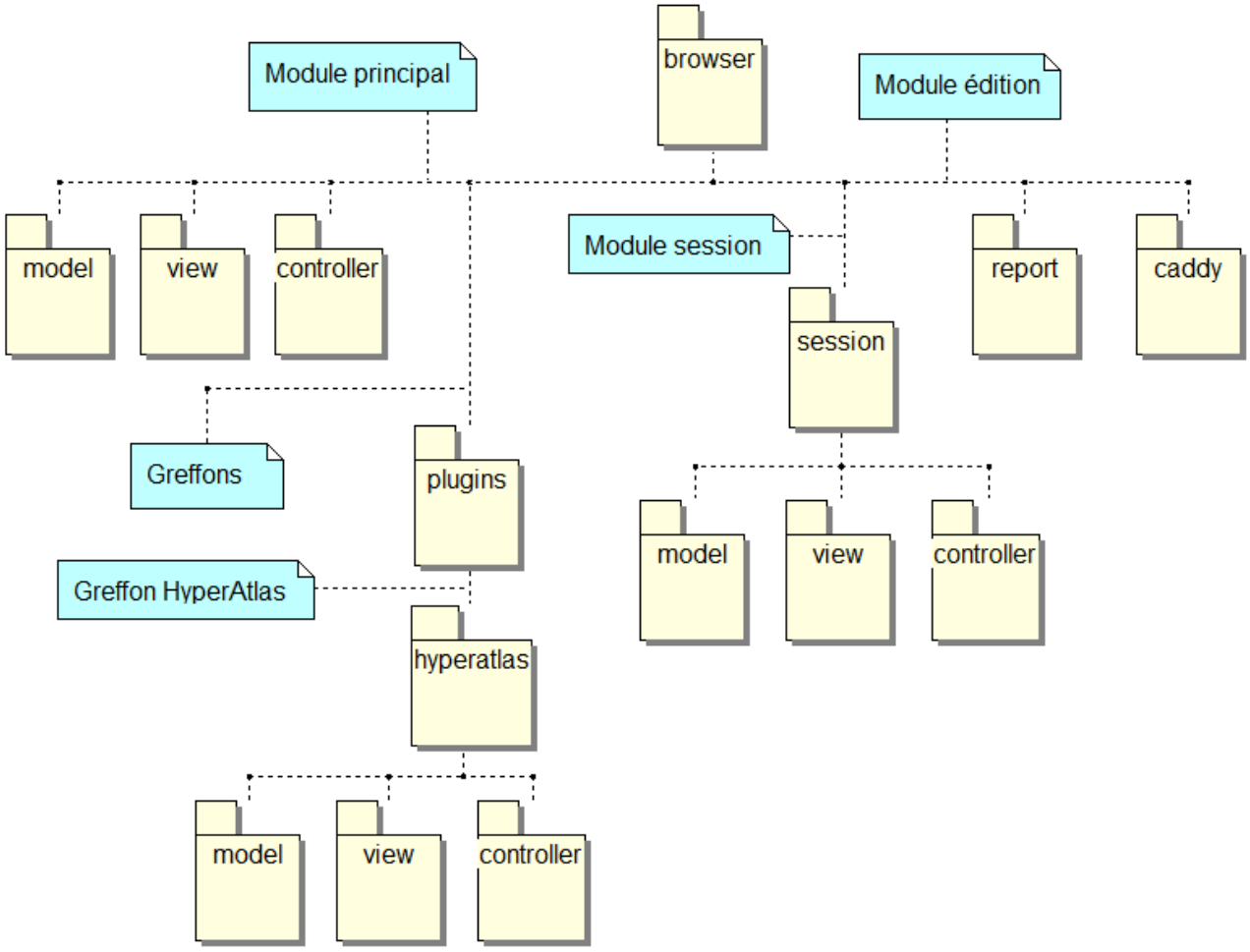


FIGURE 4.25 – Paquetages de l'application BrowserWMS

Module principal

Ce module gère la première connexion à un service. Pour ce module, le métier correspond à la gestion de l'URL d'un service. La vue correspond à la barre d'outils et aux onglets (un pour le panier et un pour chaque service auquel l'application est connectée).

19. En anglais Model View Controller [Gamma *et al.*, 1994].

Le diagramme suivant expose les classes de ce paquetage implémentant le MVC.

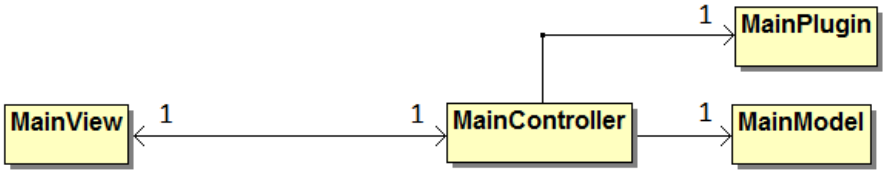


FIGURE 4.26 – Classes principales du MVC de BrowserWMS

MainController : cette classe gère les interactions entre la vue (l'utilisateur) et le modèle (service WMS) ;

MainPlugin : gère la connexion avec le service. C'est cette classe qui a la responsabilité de déléguer la session à un greffon ;

MainModel : contient la logique métier des URL, notamment une liste de services pré-sélectionnés ;

MainView : transmet au contrôleur les actions de l'utilisateur : lancement d'une connexion, ouverture/fermeture de l'onglet Caddy et des onglets de session.

Module session

Ce module gère l'acquisition. Le diagramme 4.27 montre les principales classes actrices d'une session avec un service WMS.

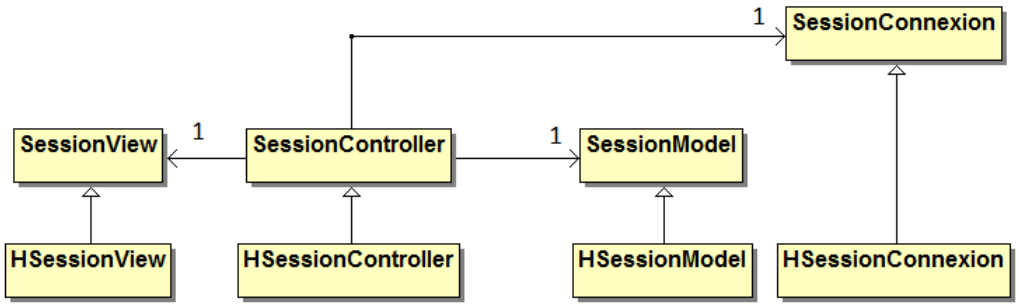


FIGURE 4.27 – Classes principales du MVC de BrowserWMS

SessionController : cette classe gère les interactions entre la vue (l'utilisateur) et le modèle (paramètres de requêtes WMS) ;

SessionConnexion : gère les envois de requêtes au service ;

SessionModel : contient la logique métier spécifique au service, découverte grâce à la requête GetCapabilities initiale ;

SessionView : transmet au contrôleur les entrées de l'utilisateur : édition de paramètres, envoi de requêtes, ou ajout dans le panier des objets acquis via ces requêtes.

Pour chacune de ces classes, il existe une spécialisation pour le greffon HyperAtlasPlugin (H+nom de la classe). Lors de la première connexion, le contrôleur principal de l'application (qui n'apparaît pas sur le diagramme) est capable de lancer une session «spécialisée» si le service est un service HyperAtlasWMS. Cette session spécialisée remplace alors la session générique. HyperAtlasPlugin gère les paramètres VendorSpecific du service HyperAtlasWMS. La classe HSessionModel est capable d'interpréter le schéma *VendorSpecificCapability* du service, la classe HSessionView présente

des éditeurs supplémentaires pour ces paramètres, la classe HSessionController coordonne ces deux spécialisations.

Module édition

Le module édition correspond à l'édition des cartes. Le principe est très simple : chaque requête (*GetMap*, *GetFeatureInfo* ou *GetLegendGraphic*) correspond à une image ou à un texte et peut être ajouté au panier. L'éditeur permet ensuite de les agencer et de les styliser : ajout de cadres autour des différents éléments, police de caractères, couleurs. La carte est ensuite sauvegardée dans un fichier XML, avec un schéma propriétaire.

4.3.2 Interactions avec un service WMS

Connexion

La connexion au service est une étape préliminaire à son utilisation. L'élément de base de la connexion est l'URL du service, que l'utilisateur fournit.

La figure 4.28 décrit les étapes de cette première connexion au service.

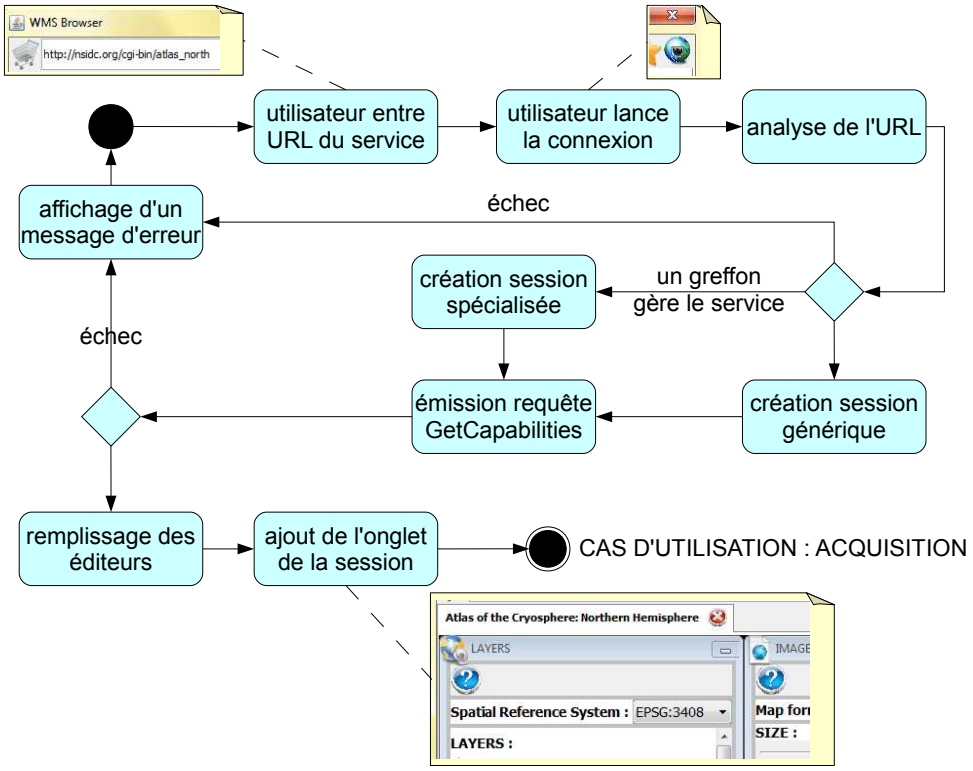


FIGURE 4.28 – Connexion à un WMS

L'enchaînement nominal de cette connexion se termine par l'ouverture d'un onglet qui contient les éditeurs qui permettent une session d'échanges avec le service :

1 l'utilisateur entre une URL ;

2 l'utilisateur lance la connexion ;

3 l'URL est analysée ;

4 émission vers le service d'une requête *GetCapabilities* ;

5 le fichier *capabilities* est analysé, et les éditeurs sont remplis à partir des informations qui s'y trouvent (couches, systèmes de référence, *etc.*) ;

6 un onglet contenant les éditeurs est affiché.

Ce scénario contient trois scénarios alternatifs :

3.a l'URL n'est pas valide, un message d'erreur est affiché ;

3.b le service est géré par un greffon [20]. Cela ne change rien à la suite du déroulement de la connexion, il y aura plus d'éditeurs (paramètres *VendorSpecific*) ;

4.a le service ne peut pas être joint, un message d'erreur est affiché ;

4.b le format de la réponse n'est pas valide, un message d'erreur est affiché.

4.3.3 Session générique

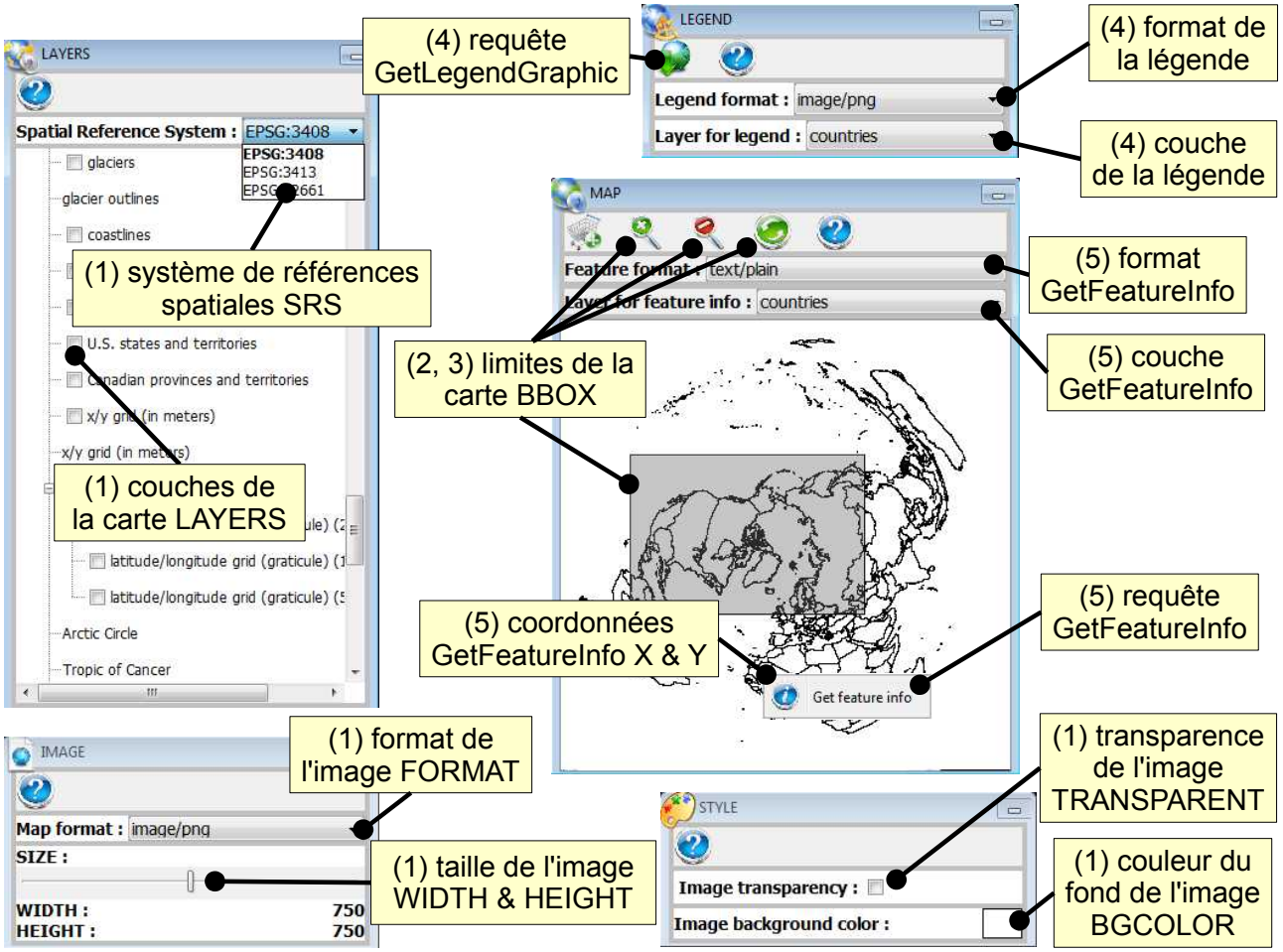


FIGURE 4.29 – Édition des paramètres WMS

Lorsque la session est lancée, les actions de l'utilisateur émettent des requêtes vers le service. Les éditeurs sont à l'intérieur de fenêtres flottantes qui se trouvent dans l'onglet. Il y a cinq façons pour l'utilisateur d'émettre une requête (les numéros correspondent avec ceux de la figure 4.29) :
1 l'utilisateur modifie la valeur d'un paramètre. Une requête *GetMap* est émise automatiquement ;
2 l'utilisateur effectue un zoom en traçant un rectangle sur la carte. Une requête *GetMap* est émise ;

20. Actuellement uniquement HyperAtlasPlugin.

3 l'utilisateur modifie le niveau de zoom : zoom avant, zoom arrière, pleine échelle. Cette action est effectuée à l'aide des boutons de la barre d'outils de la fenêtre Carte. Une requête *GetMap* est émise ;
4 l'utilisateur lance une requête *GetLegendGraphic* avec le bouton de la fenêtre Légende. La fenêtre permet de sélectionner le format et la couche de la légende ;
5 l'utilisateur lance une requête *GetFeatureInfo*. Cette action est réalisée à l'aide d'un clic droit sur la carte. Un bouton apparaît et permet d'émettre la requête. Comme pour la légende, l'utilisateur peut sélectionner le format et la couche.

Le diagramme 4.30 contient les classes mises en jeu lors de la modification du paramètre BGCOLOR, qui correspond à la couleur du fond de la carte. Les classes spécialisées qui apparaissent sur ce diagramme (IFStyle, ColorEditor et RPColor) ont leur équivalent pour les autres types de paramètres.

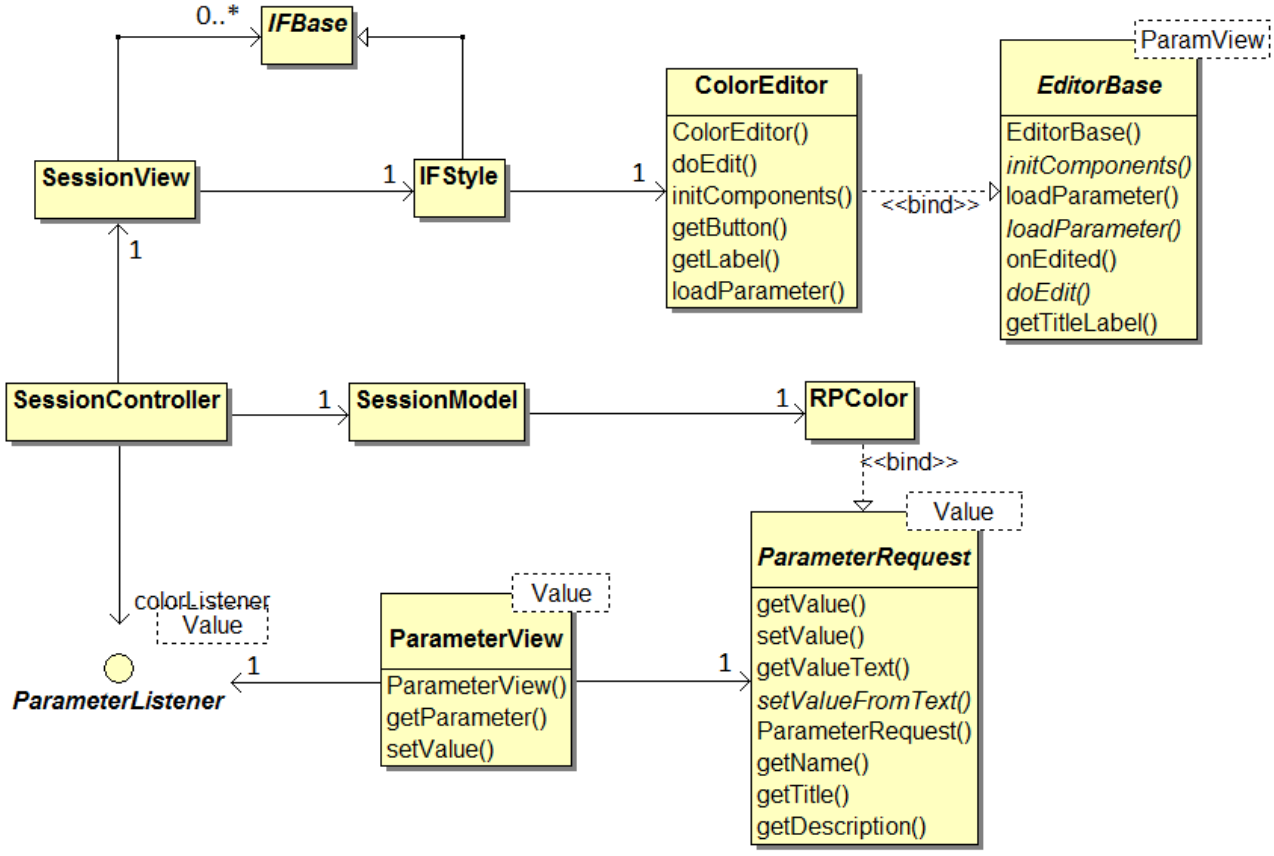


FIGURE 4.30 – Classes mises en jeu lors de l'édition d'un paramètre

La classe SessionView gère une liste de fenêtres flottantes (classe IFBase qui dérive elle même de la classe Java InternalFrame). Ces fenêtres peuvent être déplacées, agrandies, réduites, cela n'a pas d'influence sur les autres classes du diagramme.

Chaque fenêtre contient un ou plusieurs éditeurs de type (EditorBase). Chaque objet EditorBase gère un paramètre : il synchronise les actions de l'utilisateur (boutons, boîtes de sélection, *etc.*) avec ce paramètre. À chaque interaction, la valeur du paramètre est modifiée et la classe SessionController est notifiée avec la méthode *setValue* de ParameterView. EditorBase n'a aucune connaissance du type ou de la logique du paramètre qu'il édite, il n'a qu'une vue de ce paramètre : les classes ParameterView et ParameterRequest. Ce qui est lié au type et aux composants graphiques est délégué aux spécialisations (RPColor et ColorEditor), ce qui est lié à la logique à la classe SessionModel.

SessionModel contient la liste des paramètres à proposer et les valeurs qu'il peuve prendre. Il sait aussi comment «rédiger» la requête HTTP correspondant à leur état.

Lorsque SessionController est notifié, il interroge le modèle métier (SessionModel et SessionConnexion) afin de lancer une nouvelle requête *GetMap*. Il synchronise la vue avec la nouvelle carte. Dans certains cas la nouvelle requête peut ne pas produire de carte (aucune couche n'est sélectionnée), dans ce cas la fenêtre affichant la carte est vide (c'est d'ailleurs le cas au début de la session, aucune couche n'est sélectionnée par défaut).

Cinq paramètres, même s'ils utilisent le même mécanisme au niveau abstrait sont un peu particuliers :
- SRS et LAYERS sont liés : les couches ne sont pas toutes disponibles pour tous les systèmes de référence. Lorsque l'utilisateur modifie la valeur de SRS, les couches qui ne sont plus disponibles sont désactivées ;
- WIDTH et HEIGHT, qui donnent la taille de l'image sont édités simultanément à l'aide d'un curseur, afin d'assurer que l'échelle verticale et l'échelle horizontale coïncident. Les calculs sont effectués à l'aide de la classe BBox ;
- le paramètre BBOX a pour valeur par défaut sa valeur maximale (union des BBOX spécifiées dans le fichier *Capabilities*). Ce paramètre est modifié lorsque l'utilisateur effectue un zoom. Là encore, ce sont les méthodes de la classe BBox qui permettent d'assurer la cohérence entre la résolution [21] de l'image, et celle du territoire observé. À partir du rectangle que l'utilisateur dessine sur la carte lorsqu'il zoome, on calcule si la zone correspondant à sa sélection doit être étendue en hauteur ou en largeur. Ensuite, on centre par rapport à la sélection, ainsi l'intention de l'utilisateur est respectée. La figure 4.31 illustre ce mécanisme.

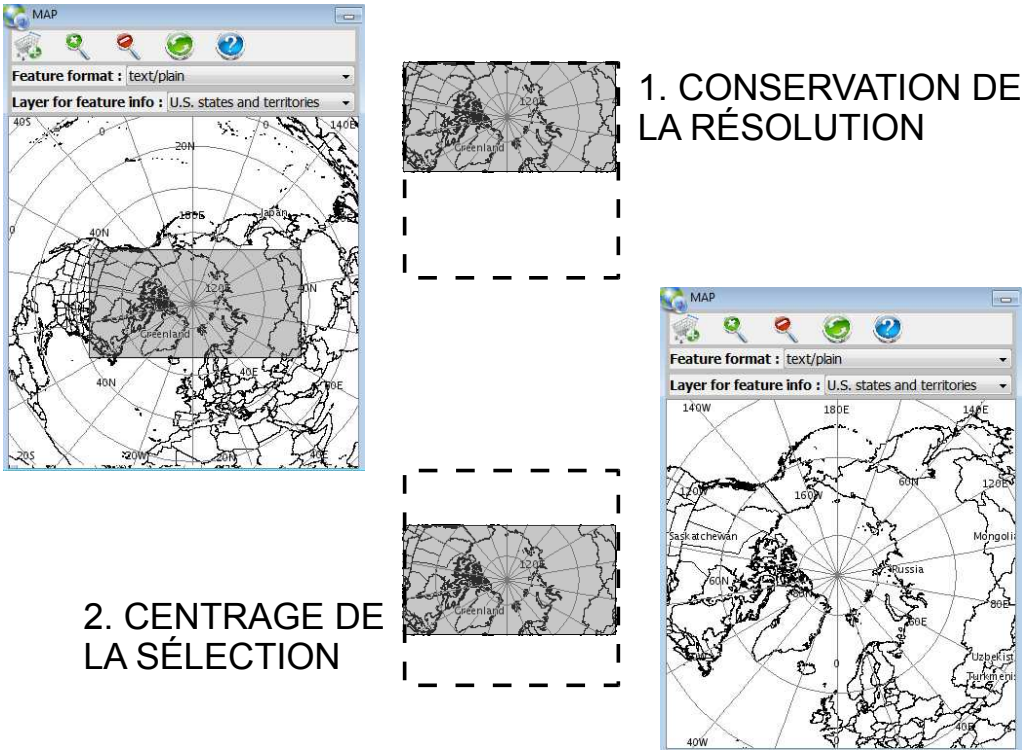


FIGURE 4.31 – Modification du paramètre BBOX avec le zoom

Les deux autres requêtes, GetLegendGraphic et GetFeatureInfo sont lancées à l'initiative de l'utilisateur en appuyant sur un bouton pour la légende, en sélectionnant un point depuis la carte (voir figure 4.29). À chacune de ces requêtes, une nouvelle fenêtre flottante est affichée avec le contenu

21. AspectRatio en anglais et dans le code

de la réponse. Ces fenêtres ne sont pas des éditeurs, et contrairement aux éditeurs elles peuvent être fermées (définitivement) si le contenu ne satisfait pas l'utilisateur, ou après avoir ajouté ce contenu au panier. La fenêtre de la carte a aussi cette fonctionnalité. La figure 4.32 montre comment l'utilisateur peut interagir avec le panier depuis la session.

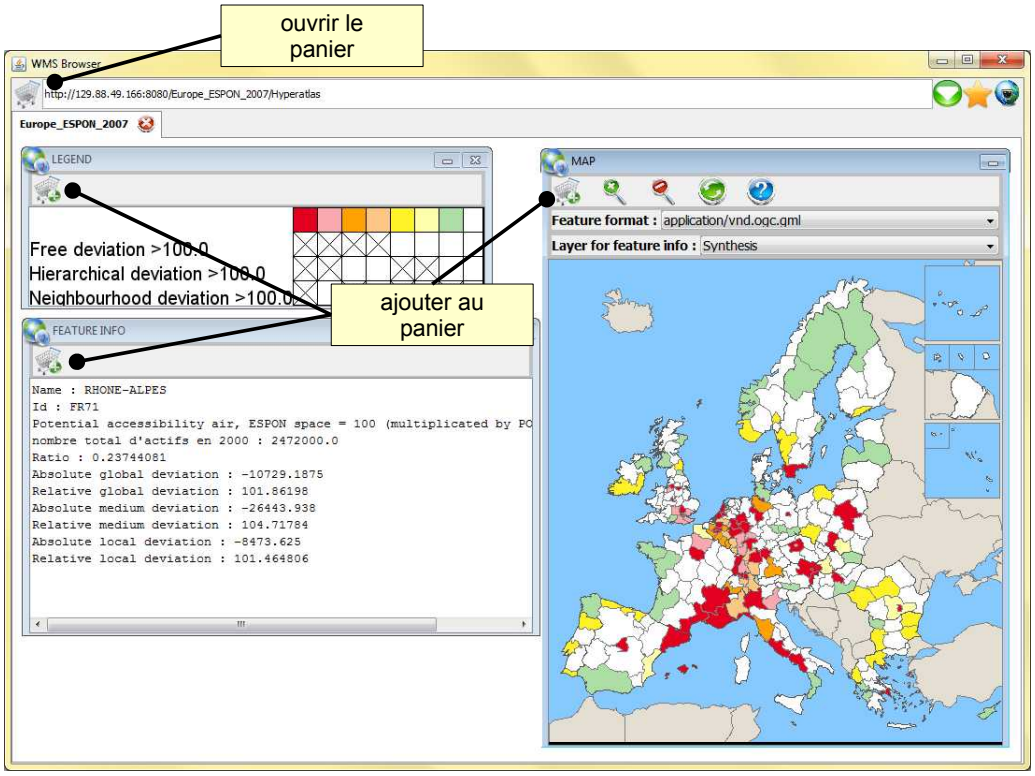


FIGURE 4.32 – Interactions entre la session et le panier

L'utilisateur peut, à tout instant, fermer sa session, en lancer une autre, ou passer au dernier cas d'utilisation, l'édition.

4.3.4 Greffon HyperAtlasPlugin

La généricité du code du paquetage *wms.browser.session*, combinée au fait que les paramètres *VendorSpecific* du service aient tous le même schéma [22], a rendu très simple l'implémentation du paquetage *wms.browser.plugin.hyperatlas*. La vue, le modèle, la connexion et le contrôleur héritent de la logique de leurs équivalents de la session et utilisent les attributs du paramètre :

- name : sera utilisé pour les requêtes ;
- category : trois nouvelles fenêtres, une pour chaque catégorie. Le contrôleur assigne aux éditeurs la bonne fenêtre grâce à cet attribut ;
- type : le type du paramètre permet au modèle et au contrôleur d'instancier les spécialisations de ParameterRequest et de EditorBase qui conviennent ;
- title, description, possibleValueList : permettent à la vue de présenter le paramètre à l'utilisateur (voir la figure 4.15 p.87) ;
- layers : seuls les paramètres utiles par rapport aux couches sélectionnées sont inclus dans les requêtes.

22. voir la description de ces paramètres p.87

Un autre avantage de cette conception est que le client est robuste aux changements effectués au niveau de HyperAtlasWMS. Au cours des itérations 13 et 14, le service a beaucoup évolué :
– la possibilité que certaines couches ne soient pas proposées (le jeu de données des USA n'a pas de hiérarchie) ;
– de nouvelles couches sont apparues ;
– de nouveaux paramètres ont été créés, mais *sans changer le schéma des paramètres*.

Toutefois, ces évolutions du service n'ont nécessité aucun travail au niveau du code du client.

Le diagramme de classes 4.33 présente les différentes spécialisations des fenêtres, éditeurs et paramètres (seules les relations d'héritage figure sur le diagramme).

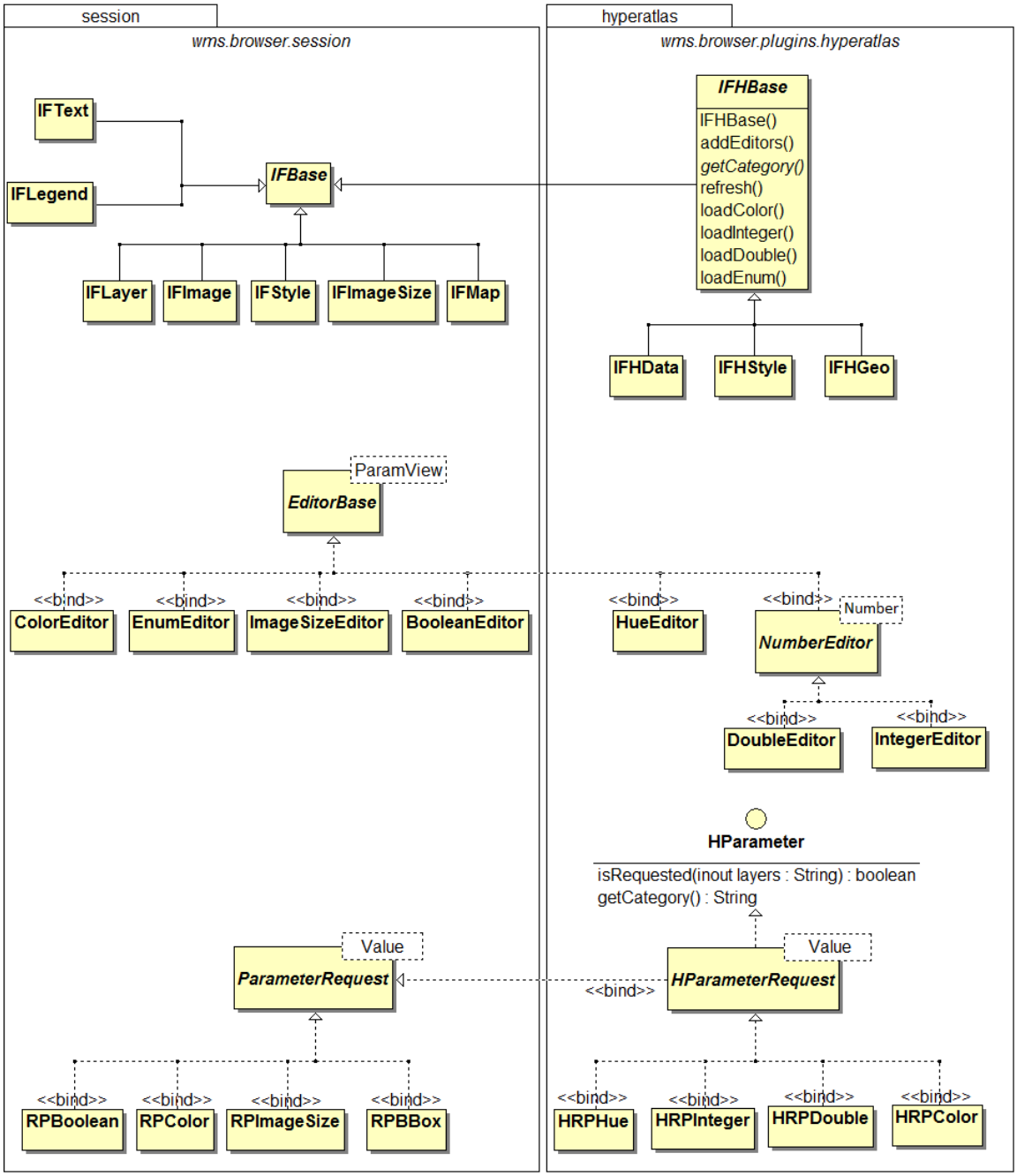


FIGURE 4.33 – Héritage de classes

Détailler chacune des classes serait fastidieux, mais le code suivant permet de comprendre le bénéfice retiré de la conception en couches du projet et de la généricité des paramètres HyperAtlasWMS. Pour les deux classes le code est présenté en intégralité, seule la documentation a été enlevée et remplacée par des commentaires en français.

La classe IFHBase est la classe abstraite générique des fenêtres de HyperAtlasPlugin. Elle hérite de IFBase. Son seul rôle est d'empiler les éditeurs, comme on peut le voir sur la figure 4.34.

```
public abstract class IFHBase  extends IFBase  {

  //nom de la fenêtre, nom du fichier pour l'icône
  public IFHBase(String name, String resource) {
    super(name, resource);
  }
  //listes des éditeurs
  private ArrayList<ColorEditor> colors = new ArrayList<ColorEditor>();
  private ArrayList<IntegerEditor> integers = new ArrayList<IntegerEditor>();
  private ArrayList<DoubleEditor> doubles = new ArrayList<DoubleEditor>();
  private ArrayList<EnumEditor> enums = new ArrayList<EnumEditor>();

  //les éditeurs sont empilés au fur et à mesure qu'ils sont ajoutés
  @Override
  protected void addEditors(JPanel editionPanel) {
    editionPanel.setLayout(new GridLayout(0,1));
  }
  //la seule fonction à redéfinir pour les fenêtres filles est cette méthode qui
      sera utilisé par le contrôleur pour distribuer les paramètres
  public abstract String getCategory();

  //les méthodes load* crée l'éditeur adapté au type du paramètre
  public void loadColor(HParameterView<Color> colorView) {
    ColorEditor ed = new ColorEditor();
    ed.setBorder(getMiddleBorder());
    ed.loadParameter(colorView);
    colors.add(ed);
    getEditionPanel().add(ed);
  }
  public void loadInteger(ParameterView<Integer> integerView) {
    IntegerEditor ed = new IntegerEditor();
    ed.setBorder(getMiddleBorder());
    ed.loadParameter(integerView);
    integers.add(ed);
    getEditionPanel().add(ed);
  }
  public void loadDouble(ParameterView<Double> doubleView) {
    DoubleEditor ed = new DoubleEditor();
    ed.setBorder(getMiddleBorder());
    ed.loadParameter(doubleView);
    doubles.add(ed);
    getEditionPanel().add(ed);
  }
  public void loadEnum(CompositeParameterView enumView) {
    EnumEditor ed = new EnumEditor();
    ed.setBorder(getMiddleBorder());
    ed.loadParameter(enumView);
    enums.add(ed);
    getEditionPanel().add(ed);
  }
  //instructions pour l'utilisateur (bouton AIDE)
  @Override
```

```
  protected String getInstructions() {
    return super.getInstructions()+"Use editors to set the different style
        parameters of your map.\n";
  }
}
```

La classe IFHGeo (première fenêtre sur la figure 4.34) n'a plus qu'à définir le nom de la fenêtre, le choix d'une icône, et la catégorie des paramètres qu'elle affiche. Ces classes sœurs, IFHData et IFHStyle font la même chose, si ce n'est l'ajout de l'éditeur des dégradés de couleur pour IFHStyle.

```
public class IFHGeo  extends IFHBase {

  //le nom et l'icône sont donnés en paramètres à la classe mère
  public IFHGeo() {
    super("   HYPERATLAS GEO", Icons.IF_LAYER);
  }

  //la catégorie correspond aux paramètres GEO : Contexte et déviations
  @Override
  public String getCategory() {
    return Constants.CATEGORY_GEO;
  }
}
```

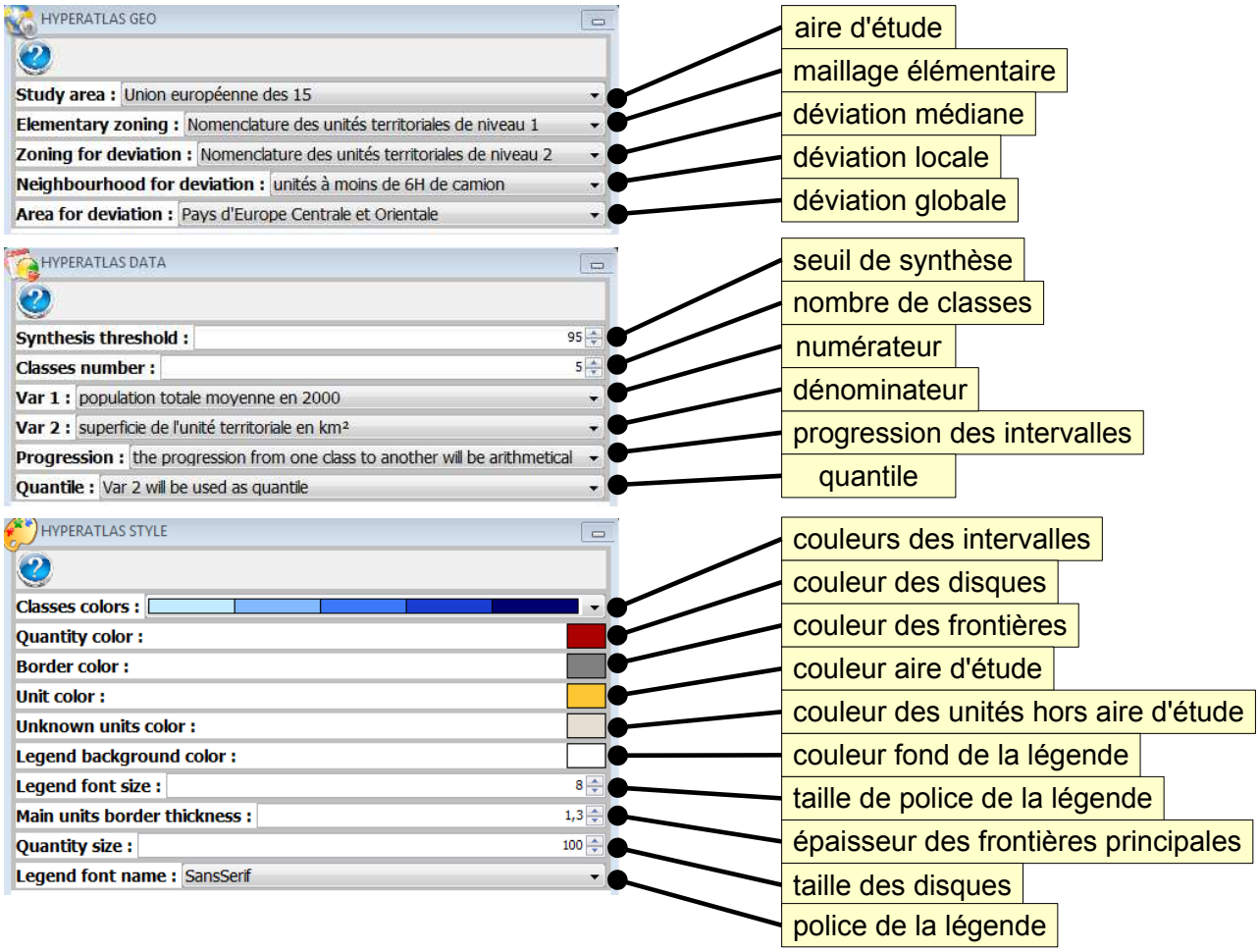


FIGURE 4.34 – Édition des paramètres HyperAtlas

4.3.5 Éditeur de cartes

L'édition se décompose en deux étapes :
– création d'un rapport ;
– sauvegarde du rapport.

Interface

La figure 4.35 et les explications qui la suivent montrent les fonctionnalités disponibles à ce jour.

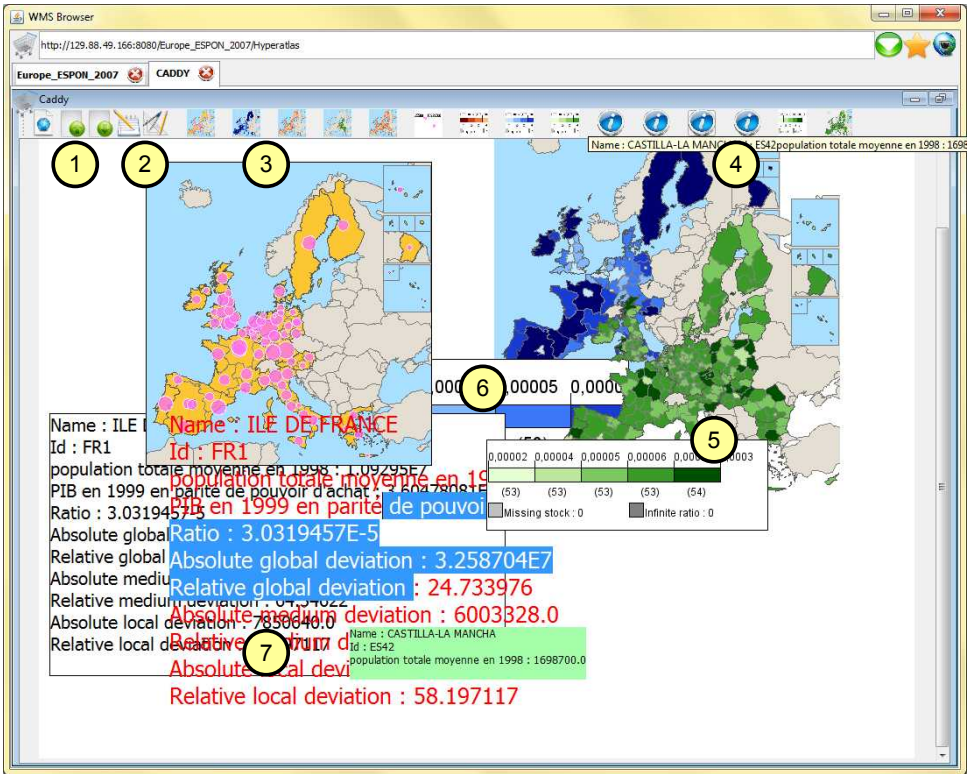


FIGURE 4.35 – Interface édition

1 : les premiers boutons de la barre d'outils correspondent à la persistance du rapport. On peut le sauver en temps qu'image (pour, par exemple, le réutiliser dans une présentation), ou sous un format propriétaire. Si le rapport est sauvegardé sous le format propriétaire, il pourra être réouvert lors d'une session et modifié ;

2 : ces deux boutons correspondent à des fonctionnalités permettant d'enrichir le rapport. Les deux fonctionnalités disponibles à ce jour sont l'ajout d'éléments de texte et le tracé de droite (par exemple, pour relier un élément de texte à un point d'une carte) ;

3 : pour chaque résultat de type image (requête GetMap ou GetLegendGraphic), un bouton est ajouté à la barre d'outil. Ce bouton est stylisé par une miniature de l'objet qu'il représente. D'une manière générale, tous les boutons de BrowserWMS sont stylisés par un image. Lorsque l'utilisateur appuie dessus, l'élément est ajouté au centre du rapport, au dessus de tous les autres. Si c'est le premier à être ajouté, le rapport est créé ;

4 : les boutons avec le symbole information correspondent à des éléments de texte obtenus à l'aide d'une requête *GetFeatureInfo*. Lorque l'utilisateur les survole avec la souris, le texte est affiché dans une bulle d'aide ;

5 : la transparence des images (paramètre TRANSPARENT de la norme WMS) est gérée. La carte verte de l'Europe est transparente. C'est très précieux si l'on souhaite superposer les résultats de plusieurs requêtes

6 : l'utilisateur peut déplacer, organiser (profondeur par rapport aux autres) ou supprimer un élément à tout instant. Il peut aussi en modifier le style, par exemple ajouter un cadre autour d'une image ou modifier la couleur de fond lorsqu'elle est transparente. Les cadres autour des légendes ou de la carte à disques ont été ajoutés de cette façon ;

7 : les éléments de texte peuvent être stylisés (couleur du fond ou des caractères, transparence, taille, police, cadre) et modifiés (l'utilisateur peut souhaiter ne garder qu'une partie des informations obtenues auprès du service). Ils peuvent aussi être déplacés, organisés, supprimés.

Format de sauvegarde

Le format de sauvegarde est basé sur XML. Il stocke tous les choix qu'a fait l'utilisateur :
- lorsque le rapport est créé, un dialogue invite l'utilisateur à choisir une taille et un style par défaut pour son document. Ces choix correspondent aux attributs de la balise <Report> ;
- pour les éléments de type image, l'attribut request de la balise <image> correspond à la requête HTTP qui a permis de l'obtenir ;
- pour les éléments de type texte, la balise <text> contient une liste d'éléments <line> (les lignes du texte), dont l'attribut content correspond au texte ;
- les éléments contiennent tous un élément <location> dont les attributs correspondent à la position et la profondeur dans le document de l'élément ;
- ce n'est pas le cas dans l'exemple suivant, mais chaque élément peut redéfinir son style, qui écrase les valeurs par défaut du rapport.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- le document fait 1600x900 pixels-->
<Report height="1600" width="900">
  <!-- éléments de style par défaut (cadres, couleurs) -->
  <imageStyle border="0" foreground="0x000000" background="0xFFFFFF"/>
  <textStyle fontSize="12" fontName="Tahoma" border="0" foreground="0x000000"
      background="0xFFFFFF"/>
  <!-- image correspondant à une requête GetMap -->
  <image request="http://nsidc.org/cgi-bin/atlas_south?SERVICE=WMS&amp;VERSION
      =1.1.1&amp;REQUEST=getmap&amp;LAYERS=land_excluding_antarctica,
      antarctic_islands,antarctic_continent,lat_long_grid_20,glaciers,
      coastlines_excluding_antarctica,antarctic_ice_shelves_outline,
      antarctic_coastline,antarctic_islands_coastlines&amp;SRS=EPSG:3409&amp;
      FORMAT=image/png&amp;BBOX=-9980476.8,-1.0184159E7,1.015021279E7,9946529.6&
      amp;WIDTH=750&amp;HEIGHT=750&amp;TRANSPARENT=false&amp;BGCOLOR=0xFFFFFF&amp
      ;STYLES=">
    <!-- position dans le document -->
    <location zorder="4" y="848" x="141"/>
  </image>
  <image request="http://nsidc.org/cgi-bin/atlas_north?SERVICE=WMS&amp;VERSION
      =1.1.1&amp;REQUEST=getmap&amp;LAYERS=land,lat_long_grid_20,glaciers,
      country_borders&amp;SRS=EPSG:3408&amp;FORMAT=image/png&amp;BBOX
      =-8569959.475,-7919064.00967,86.3180373,926.783509&amp;WIDTH=750&amp;HEIGHT
      =750&amp;TRANSPARENT=false&amp;BGCOLOR=0xFFFFFF&amp;STYLES=">
    <location zorder="2" y="57" x="139"/>
  </image>
  <!-- image correspondant à une requête GetLegendGraphic -->
  <image request="http://nsidc.org/cgi-bin/atlas_north?SERVICE=WMS&amp;VERSION
      =1.1.1&amp;REQUEST=getlegendgraphic&amp;LAYER=glaciers&amp;FORMAT=image/png
      ">
```

```
    <location zorder="0" y="818" x="637"/>
  </image>
  <!-- élément de texte -->
  <text>
    <location zorder="3" y="821" x="32"/>
    <!-- ligne de texte -->
    <line content="GLACIERS : hémisphère SUD"/>
  </text>
  <text>
    <location zorder="1" y="15" x="14"/>
    <line content="GLACIERS : hémisphère NORD"/>
  </text>
</Report>
```

Ce format permet de sauvegarder un document en utilisant un minimum d'espace mémoire, le prix à payer étant que lorsqu'on le charge pour l'afficher ou le modifier dans le client, l'application effectue à nouveau toutes les requêtes. La version XML de notre exemple utilise 2Ko contre 122 Ko pour la version au format PNG.

La figure suivante montre le document au format PNG.

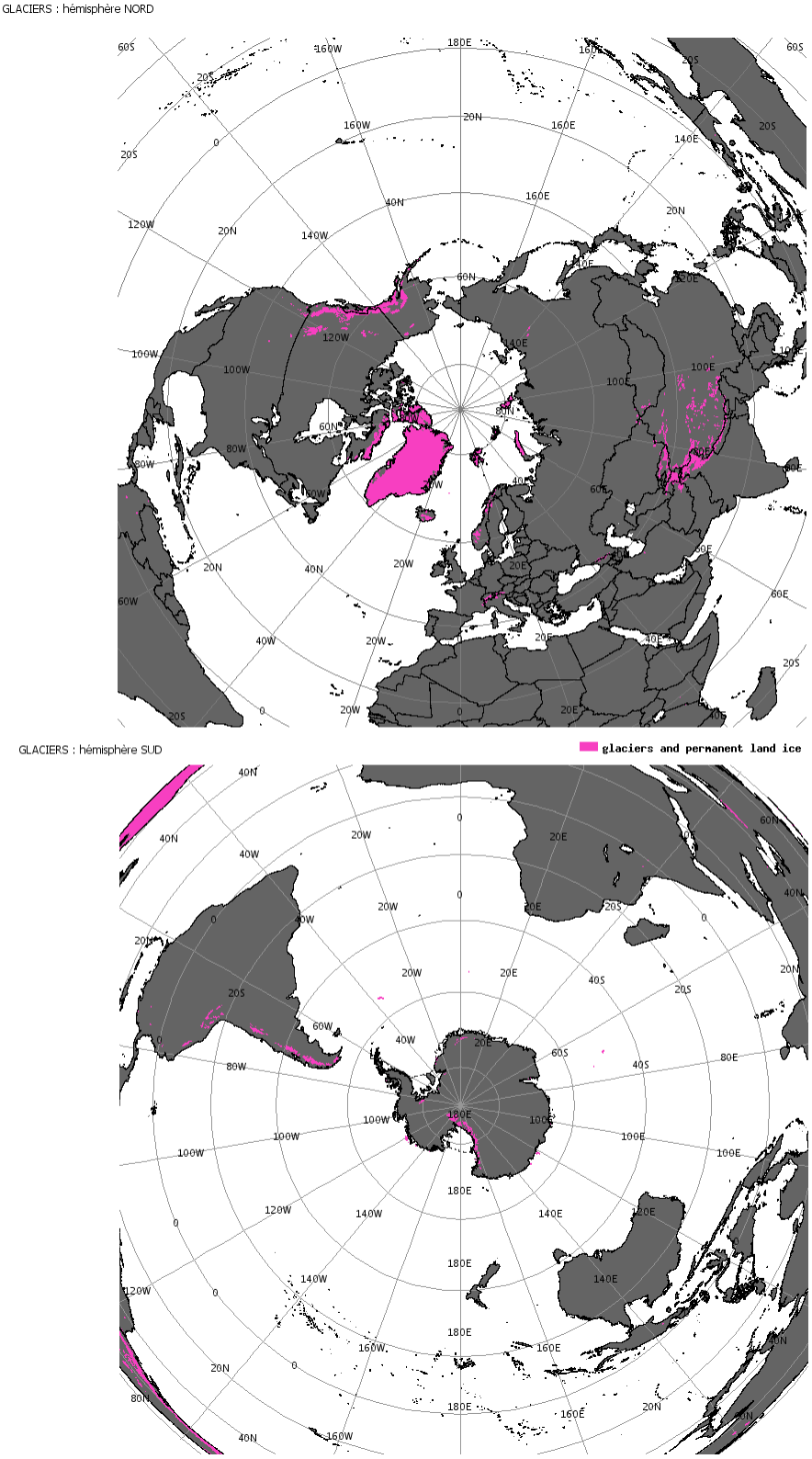


FIGURE 4.36 – Glaciers

Chapitre 5

Conclusion

L'expérience c'est le nom que chacun donne à ses erreurs.

Oscar Wilde

5.1 Rappel des objectifs

L'application HyperAtlas est une application géographique pour l'analyse spatiale, développée dans le cadre d'HyperCarte, un projet réunissant des partenaires européens issus du monde de la géographie et de celui de l'informatique.

Aujourd'hui, face à l'explosion de la demande d'information géographique, la tendance est à la mise en place d'infrastructures de données spatiales (SDI). Notre objectif était de préparer l'utilisation des informations et des traitements offerts par l'application HyperAtlas, dans le cadre d'une SDI, en les proposant à travers un service Web de l'OGC.

5.2 Synthèse

À présent, les fonctionnalités d'HyperAtlas sont disponibles à travers un Web Map Service (WMS) : HyperAtlasWMS. Ce service permet à un client WMS d'obtenir des cartes d'HyperAtlas, et de les rendre interactives à travers une interface graphique conçue dans ce sens.

Ce service correspond aux spécifications de l'OGC et est compatible avec les clients existants. Toutefois, pour pouvoir exploiter au mieux les possibilités offertes par ce service, nous avons créé un client dédié qui peut exploiter les paramètres propriétaires d'HyperAtlasWMS. Ce client peut utiliser HyperAtlasWMS, mais aussi les autres services WMS.

Notre travail comporte quatre modules. Les deux premiers, dans la liste suivante, sont génériques et adressent la spécification WMS, les deux autres sont dédiés à la logique d'HyperAtlas accessible à travers WMS :
– WMSBase permet de réaliser un service WMS ;
– BrowserWMS est un client WMS ;
– HyperAtlasWMS est un service WMS retournant les cartes d'HyperAtlas ;
– HyperAtlasPlugin est un greffon pour BrowserWMS dédié à HyperAtlasWMS.
Nous avons apporté un soin particulier à l'architecture du projet, afin de faciliter au maximum la réutilisation de notre travail dans le futur. La couche WMSBase pourra, par exemple, servir pour l'implémentation d'autres services WMS. Les tests automatisés faciliteront également ces évolutions,

en permettant de prévenir d'éventuelles régressions. La section suivante va montrer comment nous avons imaginé ces évolutions.

5.3 Perspectives

Notre travail est un premier pas vers la mise en place d'une SDI basée sur les services Web de l'OGC. Une telle SDI peut être comparée à un mille-feuilles de services et de clients, notre travail doit alors être considéré comme la première part du gâteau. On peut donc envisager de rendre cette part un peu plus épaisse, ou alors envisager d'ajouter d'autres parts.

Chaque couche de notre architecture propose des pistes pour évoluer.

WMSBase La couche WMSBase est un ensemble d'outils permettant de créer des services WMS. À partir de cette couche on peut envisager trois axes de travail :
- améliorer la prise en compte de la spécification WMS : à ce jour WMSBase ne gère pas les requêtes SLD (*DescribeLayer*, *etc.*) de WMS. Le besoin n'est pas apparu au cours de notre travail, mais c'est néanmoins l'étape nécessaire pour que WMSBase puisse être considérée comme un SDK pour WMS complet ;
- ajouter à cette couche la prise en compte d'autres services de l'OGC. Le WCS [1] ou le WFS [2] ont de nombreux paramètres en commun avec le WMS, comme BBOX par exemple. De plus, ils partagent avec WMS certains formats, comme GML. Les classes outils déjà présentes pourraient être mutualisées dans ce cadre ;
- créer de nouveaux services WMS, comme cela a été fait pour HyperAtlas, à partir de WMSBase et d'autres applications que l'équipe STEAMER développe.

HyperAtlasWMS Pour le service HyperAtlasWMS nous proposons deux axes :
- l'intégration au service du travail que Benoît Le Rubrus a mené parallèlement au nôtre : version 2.0 d'HyperAtlas. Par exemple, on pourrait ajouter à HyperAtlasWMS une carte «Synthèse Binaire» ou une carte «Redistribution» ;
- l'autre piste consisterait à utiliser d'autres sources de données que les fichiers *.hyp. C'est ce que nous avons fait à partir des fichiers *.CSV, mais on pourrait réutiliser ce travail et multiplier les possibilités. Le projet ESPON DataBase 2013 pourrait bénéficier des cartes HyperAtlas à partir d'une requête sur cette base de données : les données d'un fichier *.CSV sont le résultat d'une telle requête.

BrowserWMS Pour l'application BrowserWMS, nous proposons deux axes :
- amélioration de l'éditeur de rapport, pour l'instant il est très simple et va à l'essentiel, mais comme c'est cet outil qui ajoute de la valeur aux données, il serait souhaitable de l'enrichir d'un point de vue fonctionnel ;
- création de nouvelles sessions génériques à partir d'autres spécifications de l'OGC. L'application BrowserWMS deviendrait progressivement une application BrowserOGC.

HyperAtlasPlugin HyperAtlasPlugin n'est pas amené à évoluer en temps que tel. Même si le service HyperAtlasWMS évolue par l'ajout de nouvelles couches, il a été conçu pour s'y adapter sans que

1. Web Coverage Service
2. Web Feature Service

cela nécessite d'en modifier le code. Le seul cas où cela s'avérerait nécessaire serait la modification du modèle des paramètres VendorSpecific. Par contre l'application BrowserWMS a été conçue pour pouvoir intégrer plusieurs greffons. HyperAtlasPlugin pourrait servir de source d'inspiration dans ce cadre, et certains éditeurs pourraient alors être mutualisés.

La figure 5.1 reprend les différentes évolutions envisageables pour faire évoluer notre travail afin de réaliser une SDI. En vert figure le travail correspondant à des améliorations de nos réalisations, en rose celui correspondant à son enrichissement.

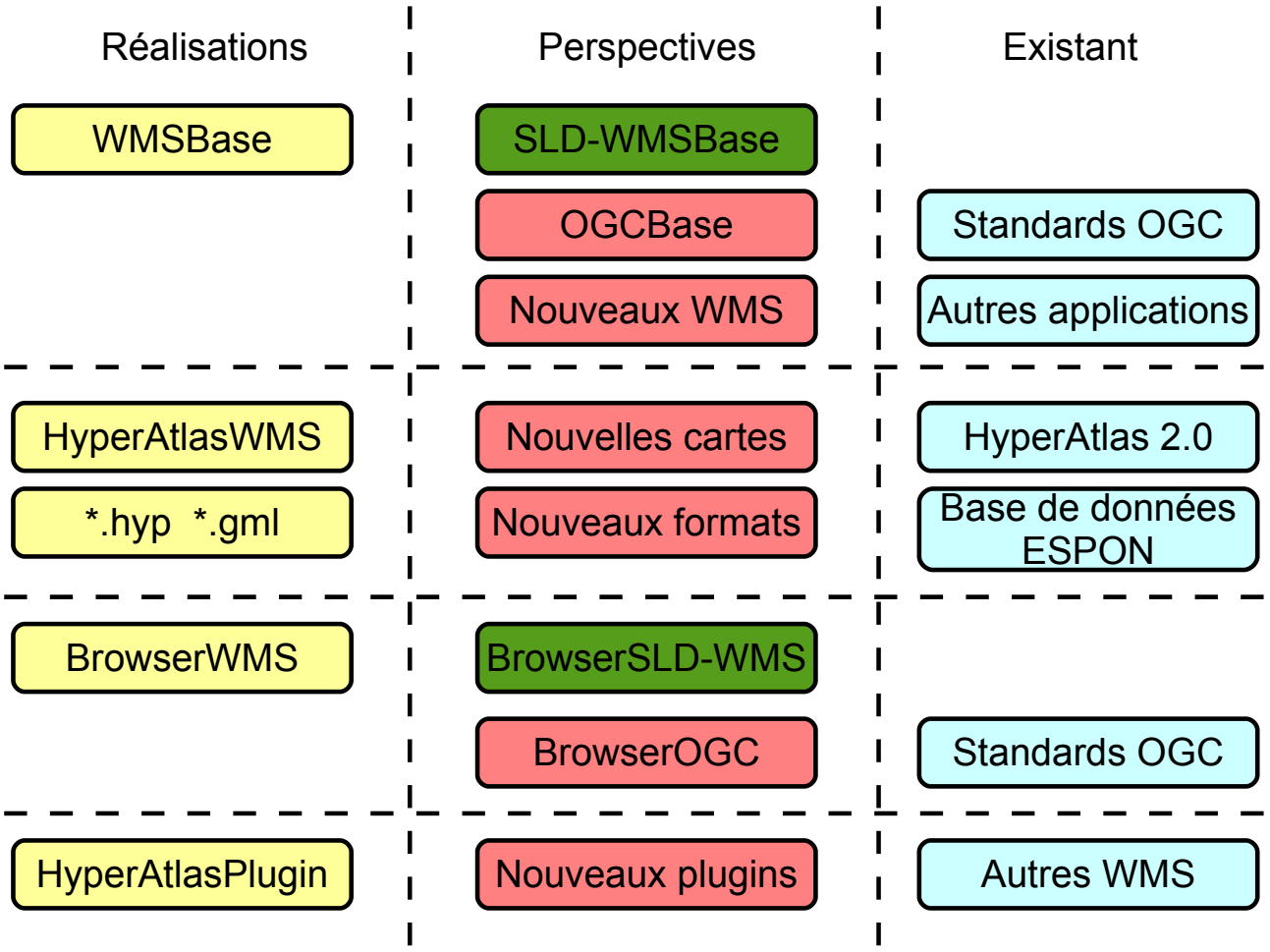


FIGURE 5.1 – Perspectives

5.4 Bilan personnel

Mon cursus CNAM est une démarche que j'ai commencée en 2005 à Montpellier. À cette époque, je participais à la création de l'entreprise LPREditor, et conscient de mes limites, j'avais décidé de suivre le cursus du CNAM afin de progresser.

Par la suite, j'ai changé d'emploi et j'ai travaillé pour l'entreprise Varian Data Systems à Grenoble. J'ai interrompu ma formation pendant un temps car l'acclimatation à la région et à ma nouvelle entreprise ont été consommatrices de temps et d'énergie. J'ai, par la suite, repris le cursus CNAM avec une ambition supplémentaire, faire évoluer ma carrière : je travaillais au sein d'équipes exclusivement composées d'ingénieurs diplômés, et le fait de ne pas posséder le diplôme était parfois un handicap.

Cette ambition était d'ailleurs soutenue par mes dirigeants et mes collègues, plusieurs d'entre eux ayant suivi ou suivant le cursus.

Suivre des cours le soir après une journée de travail demande de la ténacité et de l'organisation, mais cela crée une dynamique, apprendre, encore et toujours, qui permet de s'améliorer constamment. Les cours nous font découvrir des domaines que l'on n'aborde pas nécessairement dans le cadre professionnel. C'est par exemple, le cours NFE102 [3] de Jean-Michel Rodriguez qui m'a donné l'envie d'explorer les services Web. De plus, le contenu des cours permet d'avoir du recul et d'aborder les défis qui se posent dans une journée de travail sous des angles différents. Réciproquement, le contenu des cours est enrichi par la journée de travail : j'avais la chance de faire déjà un travail d'ingénieur, et de ce fait, les thèmes abordés en cours prenaient du relief par rapport à mon quotidien.

Le cursus CNAM se termine sur un stage de neuf mois, qui, si on le combine avec l'épreuve TEST qui le précède, représente presqu'un an de travail. Cela a été pour moi l'occasion de beaucoup progresser et d'aborder des domaines que je voulais connaître depuis longtemps.

Techniquement, je pense avoir atteint les objectifs que je m'étais fixés au départ de ce stage : appropriation du langage Java, et de la programmation des services Web. Il y a encore bien des points à améliorer, notamment en ce qui concerne les outils, comme Ant ou le serveur Tomcat, mais je me sens à présent à l'aise dans ces deux domaines. Un autre aspect, qui était également important pour moi, était d'avoir la maîtrise du projet. Mon tuteur a fait preuve d'une grande confiance en moi, et cela m'a été grandement profitable, car, d'un point de vue professionnel, le manque de confiance en soi est sans doute mon plus grand défaut. J'ai eu grand plaisir à réaliser ce travail, et surtout pris goût à l'autonomie que l'on m'a accordée pour le faire. La formation que j'ai suivie est celle d'architecte-concepteur, mais c'était la première fois que je pouvais faire tous les choix relativement à l'architecture et à la conception.

Humainement aussi, ce stage a vraiment été très riche. J'ai eu l'occasion de côtoyer des chercheurs venant de plus d'une dizaine de pays, et ils m'ont transmis leur goût pour la recherche. Ce stage Ingénieur CNAM aurait dû être l'aboutissement de ce cursus, et c'est dans cet état d'esprit que je l'ai abordé, mais il s'est avéré qu'il n'était en fait que la qu'une étape : j'aimerais aujourd'hui m'orienter vers la recherche, et c'est ce stage qui m'en a donné l'envie. C'est avec beaucoup de regrets que j'ai quitté le laboratoire et l'équipe STEAMER, et je souhaite avoir rapidement l'occasion de travailler à nouveau avec eux.

3. Infrastructures technologiques pour le commerce électronique.

Annexe A

Tests

A.1 Test unitaire

Ce test teste la méthode *intersects* de la classe BBox. Lorsqu'une BBox A appelle sa méthode *intersects* avec une BBox B comme paramètre, A est remplacée par l'intersection de A et B :

$$A.intersects(B) : A \leftarrow A \cap B$$

La méthode *same* retourne vrai si les deux BBox représentent la même région.

```
public void testIntersect() {
  BBox a = new BBox(0,0,10,20);
  BBox b = new BBox(5,-5,15,18);
  BBox actual = BBox.Max();//BBox.Max() est équivalente au plan en entier
  actual.intersects(a);
  assertTrue(actual.same(a, epsilon));//l'intersection de a et du plan vaut a
  actual.intersects(b);
  BBox expected = new BBox(5,0,10,18);//intersection de a et b
  assertTrue(actual.same(expected,epsilon ));
}
```

A.2 Test fonctionnel

MapForTest est un objet Mock. C'est une technique courante dans les tests fonctionnels. On remplace une interface métier par un objet de test. MapForTest implémente l'interface Map utilisée par un objet Style pour dessiner la carte. Il compte le nombre d'appels faits aux méthode de dessin de l'interface Map, cela permet de tester que tous les éléments de la carte sont bien dessinés.

```
public void testNumeratorStyle() throws InvalidParameterValue {
  MapForTest map = new MapForTest(bbox);
  MapRequest request = MapRequest.getReadyMapRequestForTest("1.1.1", Constants.
      LAYER_QUANTITY,bbox);
  Style style = StyleFactory.createStyle(new HMapRequest(request,data, Constants
      .LAYER_QUANTITY,null,null), data);
  style.draw(map);
  assertEquals(2*backgroundUnitCount+departementCount+4*communeCount,map.
      getCallCount());
  //les unités background ont été tracées et colorées, les frontières des unités
       principales ont été tracées, et pour chaque commune, (unités du maillage
      du contexte) : tracée, colorée et un disque a été tracé et coloré
}
```

A.3 Test de performance

La classe Benchmark permet de chronométrer l'exécution de certaines portions de codes. Le test suivant répète vingt fois la création et le dessin d'une carte. À la fin il vérifie que cela a pris moins de 5 secondes. Ce genre de tests permet dans un premier temps d'établir que le code est conforme à un certain niveau de performance, dans un deuxième temps, il alertera en cas de perte de performance au fur et à mesure des évolutions du code.

```
public void testRatioStyleSpeed() throws InvalidParameterValue {
  Benchmark.getInstance(EuropeTest.class.getCanonicalName()).start();
  for(int i = 0; i < 20;i++) {//la carte est calculée et dessinée 20 fois
    MapForTest map = new MapForTest(bbox);
    MapRequest request = MapRequest.getReadyMapRequestForTest("1.1.1", Constants
        .LAYER_RATIO, bbox);
    Style style = StyleFactory.createStyle(new HMapRequest(request,data,
        Constants.LAYER_RATIO, null, null), data);
    style.draw(map);
  }
  Benchmark.getInstance(EuropeTest.class.getCanonicalName()).stop();
        assertTrue(Benchmark.getInstance(EuropeTest.class.getCanonicalName()).
            getTotal()<5000);//les temps sont mésurés en millisecondes
}
```

Annexe B

Programmation par contrat dans HyperAtlas

Le code de la boucle suivante[1] est difficile à comprendre et mériterait d'être remanié. Le contrat *assert* sur l'invariant à la fin de la boucle garantit que ce code ne va pas boucler indéfiniment.

```
while(selectAgain && (!distribution.isEmpty() )) {
  int before = distribution.getPossibleValues();
  float key = distribution.firstKey();
  int count = distribution.getUnitsCount(key);
  if(selectedValues+count == preferedSize) {//remove key because preferedSize is
      reached
    selectAgain = false;
    maximum = key;
    distribution.remove(key);
  }
  else if(selectedValues+count < preferedSize) {//keep selecting
    selectedValues+=count;
    maximum = key;
    distribution.remove(key);
  }
  else {//select or not ?
    if(selectedValues == 0) {//select because there must be at least one
        selection
      selectAgain = false;
      maximum = key;
      distribution.remove(key);
    }
    else {
      //choose which is best : too many values, or not enough
      //it depends on the distance with preferedSize
      int tooMany = selectedValues + count - preferedSize;
      int notEnough = preferedSize - selectedValues;
      if(tooMany > notEnough) //current value is not selected
        selectAgain = false;
      else {//current value is selected
        selectAgain = false;
        maximum = key;
        distribution.remove(key);
      }
    }
  }
  //INVARIANT : selection is stopped or selectedValues has increased
  assert !selectAgain || (distribution.getPossibleValues()<before) : "POSSIBLE
     INFINITE LOOP";
```

1. extrait de l'algorithme effectuant la distribution par quantile

```
}
```

Certaines méthodes allaient évoluer au cours des itérations, pour être certain de ne pas les oublier par la suite, nous avons crée un contrat. Dans l'exemple suivant, lors de l'ajout de nouvelles couches, cette méthode ne sera pas oubliée. Lorsque la couche de synthèse a été ajoutée, tous les tests appelant cette méthode se sont mis à échouer.

```
public static final boolean isLayerChoropleth(String layerName) {
  if(layerName.equals(LAYER_STUDY_AREA)) {
    return false;
  }
  else if(layerName.equals(LAYER_QUANTITY)) {
    return false;
  }
  else if(layerName.equals(LAYER_RATIO)) {
    return true;
  }
  else if(layerName.equals(LAYER_GLOBAL_DEVIATION)) {
    return true;
  }
  else if(layerName.equals(LAYER_MEDIUM_DEVIATION)) {
    return true;
  }
  else if(layerName.equals(LAYER_LOCAL_DEVIATION)) {
    return true;
  }
  assert false : "MISSING CASE";
  return false;
}
```

Cette fonction d'initialisation doit faire beaucoup de travail. Lors de son élaboration, la première ligne de code écrite a été l'*assert*, qui correspond à la post-condition. Les tests (*écrits en premier*), échoueront jusqu'à la réalisation de cette condition. Il aurait été possible de la supprimer plus tard, mais elle ne rendait pas le code moins lisible, au contraire.

```
private void initUnits(Iterable<SerialUnitImpl>iterable, Computer<SerialUnitImpl
    ,HCUnit>computer, Progression progression, DeviationSetter deviationSetter){
  Iterator<SerialUnitImpl> it = iterable.iterator();
  while(it.hasNext()) {
    SerialUnitImpl sunit = it.next();
    HCUnit hunit = computer.compute(sunit);
    if(Float.isNaN(hunit.getRatio()))
      unitsWithMissingStock.add(hunit);
    else if(Float.isInfinite(hunit.getRatio()))
      unitsWithInfiniteRatio.add(hunit);
    else
      units.add(hunit);
  }
  MinMaxContainer minMaxContainer = deviationSetter.setDeviation(units);
  this.progressionBuilder = new ProgressionBuilder(progression, minMaxContainer)
      ;
  isArithmetic = progression.isArithmetic();
  assert progressionBuilder.getRangePossibilities()==progression.getClassesNb()
      : "INIT FAILED";
}
```

Annexe C

Patron de conception singleton

C.1 Définition

Le patron de conception singleton est défini comme suit : le singleton garantit qu'une classe n'a qu'une seule instance ET fournit un point d'accès global à cette instance.

Le code Java suivant réalise les deux éléments de cette définition :
- le constructeur est privé, donc aucune autre classe que la classe Singleton ne peut créer d'instance de Singleton. Pour disposer d'une instance de cette classe, il faut passer par la méthode *getInstance*, qui assure qu'une seule instance de la classe puisse exister, et qu'elle ne sera créée que si c'est utile ;
- la méthode *getInstance* est publique et statique, c'est le point d'entrée global. N'importe quelle classe peut disposer de l'objet Singleton à travers l'appel Singleton.*getInstance*().

```
public class Singleton{
  private static Singleton uniqueInstance;

  //le constructeur est privé, aucune autre classe ne peut créer d'objet
     Singleton
  private Singleton(){...}

  //le point d'entrée globale, n'importe quelle classe peut faire appel au
     singleton
  public static Singleton getInstance(){
    if(uniqueInstance == null){
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }

  //méthodes de la classe
  ...
}
```

Le patron de conception singleton pose plusieurs problèmes, liés aux couplages forts qu'il introduit :
- toutes les classes dépendant du singleton sont couplées entre elles : dès que l'une d'entre elles influe sur l'état du singleton, elle influe indirectement sur les autres ;
- tester unitairement les classes dépendant du singleton est complexe. Outre les problèmes de couplage, du fait de l'unicité, la même instance est utilisée tout au long des tests, et donc l'état du singleton au début d'un test dépend des tests le précédant ;
- l'unicité de l'objet Singleton crée des problèmes en cas d'accès concurrents ;
- l'accès global engendre une conception paresseuse : des Singleton.*getInstance*() apparaissent dans

le code, créant de plus en plus de couplages.

Les deux derniers points mettent en lumière l'origine de la plupart des problèmes liés à l'utilisation du patron de conception singleton : le ET de la définition. Le patron de conception singleton est la solution de deux besoins. Dans certains cas l'accès global suffirait, dans d'autres l'unicité.

Bannir l'utilisation du singleton comme le préconisent certains développeurs est exagéré, car il est la bonne solution à certains problèmes. Il faut donc se demander avant d'utiliser le singleton s'il est la solution au problème posé (l'accès global ET l'unicité de l'objet), et ensuite se demander pour chaque appel à Singleton.*getInstance*() s'il est justifié, malgré les couplages qu'il introduit. Une bonne pratique consiste à séparer dans la classe l'implémentation du singleton de la logique de service, en faisant implémenter à la classe une interface correspondant aux services qu'elle rend :

```
public class Singleton extends Service{
  private static Singleton uniqueInstance;

  //logique singleton
  private Singleton(){...}

  public static Service getInstance(){
    if(uniqueInstance == null){
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }

  //méthodes de l'interface Service

  doService(){
  ...
  }
}
```

Il faudra se demander quelle est la meilleure des deux solutions suivantes. La première correspond à la nécessité d'utiliser le point d'entrée global, la seconde à celle des fonctions (services) de la classe :

```
public class A {

  doJob(){
  ...
  Singleton.getInstance().doService();
  ...
  }
}

OU

public class A {

  doJob(Service service){
  ...
  service.doService();
  ...
  }
}
```

La meilleure solution étant presque toujours la seconde, puisqu'elle diminue les couplages. C'est cette technique, appelée l'inversion de dépendances, que nous avons utilisée pour réutiliser la couche de persistance d'HyperAtlas avec le service HyperAtlasWMS.

C.2 Exemple d'inversion des dépendances dans HyperAtlas

La méthode *readDataFromFile* ouvre un fichier, et stocke certaines données dans le singleton Settings, qui seront réutilisées par d'autres classes. Elles correspondent à des variables globales d'état. Le code existant était le suivant (dans les extraits n'apparaît que le code utile à l'explication, la classe Settings comporte presque 3000 lignes de code) :

```
//singleton Settings
public class Settings {
  //code patron de conception

  public void setRatioPertinent(boolean ratioPertinent) {
    this.ratioPertinent = ratioPertinent;
  }

  ...
}

public class DataSerialver {

  public static void readDataFromSerialFile(ObjectInputStream ois) throws
      IOException, ClassNotFoundException{
    ...
    Settings.getInstance().setRatioPertinent(true);
    ...
  }
}
```

Il a été remanié de la manière suivante :

```
//singleton Settings
public class Settings implements SettingsProvider {//à présent Settings
    implémente l'interface SettingsProvider
  ...
}

public interface SettingsProvider {

  void setRatioPertinent(boolean ratioPertinent);

}

public class DataSerialver {

  public static void readDataFromSerialFile(ObjectInputStream ois,
      SettingsProvider settingsProvider) throws IOException,
      ClassNotFoundException{
    ...
    settingsProvider.setRatioPertinent(true);
    ...
  }
}
```

Dans l'application HyperAtlas, les appels à *readDataFromSerialFile*(*ois*) ont été remplacés par l'appel *readDataFromSerialFile*(*ois*,Settings.*getInstance*()). Dans le service Web HyperAtlasWMS, l'appel est le suivant : *readDataFromSerialFile*(*ois*,new EmptySettings()). La classe EmptySettings remplace le singleton, et les couplages existant entre DataSerialver et les autres classes d'HyperAtlas à travers le singleton disparaissent (c'est l'inversion de dépendances).

Le code de la classe EmptySettings est le suivant, il ne sert qu'à neutraliser la logique de la classe Settings lors de la réutilisation du code HyperAtlasWMS, le service est un service REST, et il est sans états :

```
public class EmptySettings implements SettingsProvider{

  public void setRatioPertinent(boolean ratioPertinent){
    //il ne se passe rien, cet appel est inutile pour le service \hyperwms
  }
}
```

Glossaire

Analyse spatiale : l'analyse spatiale met en évidence des structures et des formes d'organisation spatiale récurrentes, et analyse des processus qui sont à l'origine de ces structures, à travers des concepts comme ceux de distance, d'interaction spatiale, de territorialité, *etc*. Une attention particulière est apportée en analyse spatiale à la définition de l'échelon géographique considéré, du niveau d'observation, qu'il s'agisse du niveau «microscopique» des acteurs individuels, ou d'agrégats spatiaux définis à des niveaux macro géographiques. L'analyse des processus par lesquels s'effectue le passage d'un niveau à un autre, l'émergence qualitative des structures qui rend pertinent le changement d'échelle, est l'un des grands problèmes posés actuellement à la réflexion théorique et à la modélisation en géographie [@Wikipedia, a].

Couplage : le couplage est une métrique indiquant le niveau d'interaction entre deux ou plusieurs composants logiciels (fonctions, modules, objets ou applications). Deux composants sont dits couplés s'ils échangent de l'information. On parle de couplage fort ou couplage serré si les composants échangent beaucoup d'information. On parle de couplage faible, couplage léger ou couplage lâche si les composants échangent peu d'information [@Wikipedia, b].

Classe : en informatique, la classe est un des concepts de base de la programmation orientée objet. On appelle classe un ensemble d'objets partageant certaines propriétés (les méthodes et les attributs).

Paquetage : en programmation, et plus particulièrement dans la terminologie de certains langages (Ada, Java, Common Lisp), le terme paquetage est le nom donné aux bibliothèques logicielles et servent aussi à la partition des espaces de noms.

Patron de conception : en informatique, un patron de conception, motif de conception ou modèle de conception est un concept de génie logiciel destiné à résoudre les problèmes récurrents suivant le paradigme objet. Les trois termes sont des traductions de l'expression anglophone design pattern, tentant d'exprimer la réutilisabilité d'un concept qu'on retrouve dans les patrons et les motifs. Les patrons de conception décrivent des solutions standard pour répondre à des problèmes d'architecture et de conception des logiciels. Le Gof [Gamma *et al.*, 1994] est un ouvrage de référence, qui regroupe les principaux patrons de conception.

Remaniement de code : remanier le code (en anglais *refactoring*) est une opération de maintenance du code informatique. Elle consiste à retravailler le code source non pas pour ajouter une fonctionnalité supplémentaire au logiciel mais pour améliorer sa lisibilité, simplifier sa maintenance, ou changer sa généricité. Cela consiste à :
- s'assurer que toute l'information nécessaire est disponible ;
- supprimer toute information redondante ou duplication de code ;
- simplifier l'algorithmique des méthodes ;
- limiter la complexité des classes ;
- limiter le nombre de classes.

Service Web : un service Web est un programme informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Il

s'agit donc d'un ensemble de fonctionnalités exposées sur internet ou sur un intranet, par et pour des applications ou machines, sans intervention humaine.

Servlet : une Servlet est une classe qui s'exécute dynamiquement sur un serveur Web et permet l'extension des fonctions de ce dernier, typiquement : accès à des bases de données, transactions d'e-commerce, *etc*. Une Servlet peut être chargée automatiquement lors du démarrage du serveur Web ou lors de la première requête du client. Une fois chargées, les Servlets restent actives dans l'attente d'autres requêtes du client.

Bibliographie

[Balley, 2008] BALLEY, S. (2008). «Aide à la restructuration de données géographiques sur le Web - vers la diffusion à la carte d'information géographique». Thèse - Paris-Est Marne la Vallée.

[Beck, 1999] BECK, K. (1999). «Extreme Programming Explained». Addison-Wesley.

[Beck, 2002] BECK, K. (2002). «Test Driven Development : By Example». Addison-Wesley.

[Bertin, 1967] BERTIN, J. (1967). «Sémiologie Graphique. Les diagrammes, les réseaux, les cartes». EHESS.

[Brown *et al.*, 1998] BROWN, W. J., MALVEAU, R. C., MCCORMICK, H. W. et MOWBRAY, T. J. (1998). «Anti Patterns : refactoring software, architecture and projects in crisis». Wiley Computer Publishing.

[Bucher, 2007] BUCHER, B. (2007). «La carte à la carte sur le Web». revue du CFC, n° 193.

[Bucher, 2009] BUCHER, B. (2009). «Vers la diffusion en ligne d'information géographique sur mesure». Habilitation à diriger des recherches Université Paris-Est Marne-la-Vallée.

[Chabert, 2007] CHABERT, C. (2007). «HyperAtlas et HyperAdmin : des outils cartographiques pour l'analyse de phénomènes sociaux». Mémoire EICNAM Rhône-Alpes.

[Cuenot, 2005] CUENOT, O. (2005). «Modélisation spatiale multiscalaire de phénomènes sociaux». Mémoire EICNAM Rhône-Alpes.

[Davies, 2003] DAVIES, J. (2003). «Expanding the Spatial Data Infrastructure model to support spatial wireless application». Département of Geomatics - University of Melbourne.

[Doudoux, 2010] DOUDOUX, J.-M. (2010). «Développons en Java». http://www.jmdoudoux.fr

[Dumolard *et al.*, 1998] DUMOLARD, P., ALLIGNOL, F., PAUL, E. et QUESSEVEUR, E. (1998). «L'outil informatique en géographie». Wiley Computer Publishing.

[Fielding, 2000] FIELDING, R. T. (2000). «Architectural Styles and the Design of Network-based Software Architectures». Thèse, University of California.

[Fowler et Beck, 1999] FOWLER, M. et BECK, K. (1999). «Refactoring : Improving the Design of Existing Code». Addison-Wesley.

[Gamma *et al.*, 1994] GAMMA, E., HELM, R., JOHNSON, R. et VLISSIDES, J. (1994). «Design Patterns : Elements of Reusable Object-Oriented Software». Addison-Wesley.

[Grasland *et al.*, 2007] GRASLAND, C., LAMBERT, N., YSEBAERT, R. et ZANIN, C. (2007). «Observing the structure of European territory in relative terms». Parlement Européen - Rapport.

[GSDI, 2008] GSDI (2008). «SDI Cookbook». http://www.gsdi.org/gsdicookbookindex.

[Houellebecq, 2010] HOUELLEBECQ, M. (2010). «La carte et le territoire». Flammarion.

[Jayasinghe, 2008] JAYASINGHE, D. (2008). «Quickstart Apache Axis2 : a practical guide to creating quality Web services». Packt publishing.

[Kuhn, 2005] KUHN, W. (2005). «Introduction to Spatial Data Infrastructures». http://www.docstoc.com/docs/2697206/Introduction-to-Spatial--Data-Infrastructures.

[Le Rubrus, 2011] LE RUBRUS, B. (2011). «HyperAtlas et HyperAdmin version 2». Mémoire EICNAM Rhône-Alpes.

[Martin, 2004] MARTIN, P. (2004). «Interface cartographique pour l'analyse territoriale multiscalaire de phénomènes sociaux». Mémoire EICNAM Rhône-Alpes.

[Martin, 2000] MARTIN, R. C. (2000). «Design Principles and Design Pattern». http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

[Meyer, 1988] MEYER, B. (1988). «Object-Oriented Software Construction». Prentice Hall.

[Plessel, 1998] PLESSEL, T. (1998). «Design By Contract : A Missing Link In The Quest For Quality Software». Lockheed Martin, US EPA.

[Plumejeaud, 2007] PLUMEJEAUD, C. (2007). «Acquisition de données et cartes de potentiel pour l'analyse spatiale». Mémoire EICNAM Rhône-Alpes.

[Poulenard, 2010] POULENARD, L. (2010). «Web 2.0 et Information Géographique Citoyenne (Volunteered Geographic Information)». Épreuve test EICNAM Rhône-Alpes.

[Rajabifard et Williamson, 2001] RAJABIFARD, A. et WILLIAMSON, I. P. (2001). «Spatial Data Infrastructures : concept, SDI hierarchy and future directions». GEOMATICS'80 Conference.

[Thomas, 2008] THOMAS, R. (2008). «Évolutions d'outils dédiés à l'analyse territoriale et à l'analyse spatiale dans le cadre du projet HyperCarte». Mémoire EICNAM Rhône-Alpes.

[Tobler, 1970] TOBLER, W. (1970). «A computer movie simulating urban growth in the Detroit region». Economic Geography n° 46.

[Tobler, 2000] TOBLER, W. (2000). «The Development of Analytical Cartography». Cartography and Geographic Information Science.

[Tyagi, 2006] TYAGI, S. (2006). «RESTful Web Services». http://www.oracle.com/technetwork/articles/javase/index-137171.html.

[Wang *et al.*, 2003] WANG, J., QIN, L., VEMURI, N. et JIA, X. (2003). «A toolset for Design By Contract for Java». DePaul University.

Les sites Web suivants ont tous été consultés en janvier 2011.

[@ESPON, a] @ESPON.
http://www.espon.eu/.

[@ESPON, b] @ESPON. «ESPON 2013 Database, Second Interim Report».
http://www.espon.eu/export/sites/default/Documents/Projects/ScientificPlatform/ESPONDatabase2013/Second_Interim_Report.pdf

[@Hypercarte] @HYPERCARTE.
http://hypercarte.imag.fr/.

[@OGC] @OGC
http://www.opengeospatial.org/.

[@Wikipedia, a] @WIKIPEDIA. Analyse spatiale.
http://en.wikipedia.org/wiki/Spatial_analysis.

[@Wikipedia, b] @WIKIPEDIA. Couplage.
http://en.wikipedia.org/wiki/Coupling_(computer_science).

[@Wikipedia, c] @WIKIPEDIA. Interopérabilité.
http://en.wikipedia.org/wiki/Interoperability.

[@Wikipedia, d] @WIKIPEDIA. Utilisabilité.
http://fr.wikipedia.org/wiki/Usability.

[@Wikipedia, e] @WIKIPEDIA. Wsdl.
http://fr.wikipedia.org/wiki/WSDL.

MÉMOIRE D'INGÉNIEUR C.N.A.M. en INFORMATIQUE

Diffusion de l'information géographique avec les services Web : application au logiciel HyperAtlas

Laurent Poulenard **Grenoble, le 7 avril 2011**

Résumé

Les Infrastructures de Données Spatiales (SDI) permettent la coopération entre fournisseurs de données géographiques, et peuvent, à cet égard, être considérées comme une nouvelle génération de Systèmes d'Information Géographique (SIG). L'OGC, un consortium regroupant les acteurs principaux des SIG traditionnels et du Web, propose un ensemble de spécifications permettant la mise en place de SDI basées sur les services Web. L'objectif de ces spécifications est de créer l'interopérabilité nécessaire entre les différents fournisseurs d'informations géographiques et leurs clients.

HyperAtlas est un logiciel d'analyse spatiale que l'on peut actuellement télécharger depuis le site d'ORATE/ESPON (Observatoire en Réseau de l'Aménagement du Territoire Européen). Notre travail a consisté à lui donner le niveau d'interopérabilité des spécifications de l'OGC en rendant ses fonctionnalités disponibles à travers un Web Map Service (WMS). Notre travail a également consisté à créer un client compatible avec les WMS existant et apte à tirer le meilleur profit des fonctionnalités d'HyperAtlas.

L'ensemble de nos développements a été réalisé en mettant l'accent sur les principes avancés de conception objet : évolutions, dépendances, organisation modulaire et stabilité.

Mots-clés : Infrastructures de Données Spatiales - Services Web - Interopérabilité

Abstract

Spatial Data Infrastructures (SDI) allow cooperation between geographic data suppliers, and in this regard can be considered as the new generation of Geographic Information Systems (GIS). OGC, a consortium of major players from traditional GIS and Web economics, offers a set of specifications enabling the establishment of Web services-based SDIs. The purpose of these specifications is to create the necessary interoperability between different suppliers of Geographic Information and their consumers.

HyperAtlas is a spatial analysis software that can currently be downloaded from the website of ESPON/ORATE (European Observation Network for Territorial Development and Cohesion). Our job was to give the level of interoperability of OGC specifications in making its features available through a Web Map Service (WMS). Our work has also been to create a client application compatible with existing WMS and able to take full advantage of the features of HyperAtlas.

All our developments have been achieved focusing on advanced principles of design : trends, dependencies, modular organization and stability.

Keywords : Spatial Data Infrastructures – Web Services - Interoperability