



U.F.R. Informatique &  
Mathématiques Appliquées



LSR - SIGMA

I.M.A.G

## MAGISTERE INFORMATIQUE 3ème année

Réalisation d'un noyau logiciel pour la mise en oeuvre  
d'un accès progressif à l'information  
dans les applications KIWIS

Projet présenté par :  
BILASCO Ioan Marius

Effectué au laboratoire  
LSR (Logiciels Systèmes Réseaux) de Grenoble  
Au sein de l'équipe  
SIGMA (Systèmes d'Information : inGénierie et Multimédia)

Date : 10 octobre 2003

Jury :

M. Claude Puech

Mme. Marlène Villanova-Oliver

# REMERCIEMENTS

---

Je remercie Madame Marlène VILLANOVA-OLIVER, Maître de Conférences à l'Université Pierre-Mendès France de Grenoble et Monsieur Jérôme GENSEL, Maître de Conférences à l'Université Pierre-Mendès France de Grenoble, pour leurs conseils lors de l'élaboration de ce travail et pour leur encadrement, leur aide et leur amitié tout au long de l'année.

Je remercie Monsieur Hervé MARTIN, Professeur à l'Université Joseph Fourier de Grenoble, pour le soutien accordé tout au long de mon activité au sein de l'axe multimédia de l'équipe SIGMA.

Je tiens à remercier également Monsieur Jean Pierre GIRAUDIN de m'avoir accueilli au sein de son équipe, ainsi que Monsieur Farid OUABDESSELAM en tant que directeur du laboratoire d'accueil.

Je remercie tous les membres de l'équipe SIGMA de LSR pour cette année agréable passée ensemble.

# TABLE DES MATIÈRES

---

<b>Remerciements</b> .....	<b>i</b>
<b>Table des matières</b> .....	<b>iii</b>
<b>Table des images</b> .....	<b>vi</b>
<b>Introduction</b> .....	<b>1</b>
Structure d'accueil .....	1
Equipe d'accueil .....	1
Contexte .....	2
Travail à réaliser .....	3
Plan du mémoire .....	3
<b>Etat de l'art</b> .....	<b>5</b>
1.    Kiwis .....	6
1.1. Description UML des besoins fonctionnels.....	7
1.2. Organisation de la plateforme KIWIS .....	8
1.3. Architecture logicielle de la plateforme KIWIS.....	9
2.    Représentation des connaissances avec AROM .....	11
2.1. Représenter des connaissances en AROM .....	11
2.1.1    Classes et objets .....	11
2.1.2    Associations et tuples .....	12
2.2. Architecture de la plateforme AROM .....	12
2.3. Le module de représentation de connaissances .....	13
2.3.1    Classes et Associations.....	14
2.3.1.1    Slots .....	14
2.3.1.2    Facettes .....	15
2.3.2    Instances AROM.....	15
2.3.2.1    Objets AROM.....	15
2.3.2.2    Tuples AROM .....	15
2.4. Module de types.....	15
3.    Patrons de conception.....	18
3.1. Fabrique abstraite (Abstract Factory) .....	19
3.1.1    Problème.....	19
3.1.2    Solution architecturale.....	19

3.1.3	Conséquences .....	20
3.2.	Pont (Bridge) .....	20
3.2.1	Problème.....	20
3.2.2	Solution architecturale.....	21
3.2.3	Conséquences .....	21
3.3.	Adaptateur (Adaptor).....	21
3.3.1	Problème.....	21
3.3.2	Solution architecturale.....	22
3.3.3	Conséquences .....	23
3.4.	Chaîne de responsabilités (Chain of responsibilities).....	23
3.4.1	Problème.....	23
3.4.2	Solution architecturale.....	23
3.4.3	Conséquences .....	24
<b>Spécifications</b>	.....	<b>27</b>
4.	Les modèles dans KIWIS .....	28
4.1.	Le modèle d'accès progressif .....	28
4.1.1	Entité Masquable et Représentation d'Entité Masquable .....	28
4.1.2	Stratification .....	30
4.1.3	Fonctions pour l'accès progressif.....	30
4.2.	Modèle du Domaine .....	31
4.3.	Modèle des Fonctionnalités .....	31
4.3.1	Modèle des Fonctionnalités et Accès progressif .....	32
4.3.2	Fonctionnalité de consultation avec accès progressif .....	32
4.4.	Le modèle utilisateur .....	34
<b>Réalisation</b>	.....	<b>37</b>
5.	Le noyau KIWIS .....	38
5.1.	Module d'utilisateurs (kiwis.user) .....	38
5.2.	Module d'accès (kiwis.access) .....	40
5.2.1	Récupération des entités du système KIWIS .....	40
5.2.1.1	L'interface KContext.....	40
5.2.1.2	L'interface EntityLocator .....	41
5.2.2	L'accès au système.....	41
5.2.2.1	KAccess .....	42
5.2.3	Perspectives ouvertes par le module d'accès .....	42
5.3.	Module d'accès progressif (kiwis.pam).....	43

5.3.1	L'élément et l'entité masquable .....	43
5.3.2	Stratifications et Représentations d'entités masquables.....	44
5.3.3	Le PAM .....	46
5.3.4	Création des représentations effectives d'entité masquables .....	47
5.4.	Module des données (kiwis.data) .....	47
5.5.	Le module des fonctionnalités (kiwis.functionality).....	48
5.5.1	Fonctionnalités de consultation (ReportFunctionality) .....	49
5.5.2	Rôles Fonctionnels et Espace Fonctionnel (FunctionalRoles, FunctionalSpace).....	51
5.5.3	L'interface FunctionalModel.....	51
5.6.	Le système et les application KIWIS.....	52
5.7.	Implémentation AROM du noyau Kiwis.....	54
5.7.1	Organisation des bases de connaissances.....	55
5.7.2	Base de connaissances système.....	56
5.7.3	Base de connaissances pour le module des fonctionnalités .....	57
<b>Conclusion .....</b>		<b>59</b>
<b>Bibliographie.....</b>		<b>61</b>

# TABLE DES IMAGES

---

Figure 1.1 Diagramme des cas d'utilisation pour l'acteur CONCEPTEUR .....	7
Figure 1.2 Diagramme des cas d'utilisation pour l'acteur UTILISATEUR .....	8
Figure 1.3 Vue logique de l'architecture de KIWIS .....	10
Figure 2.1 Organisation de la plateforme AROM [AROM01] .....	13
Figure 2.2 Relation de composition entre les objets AROM [AROM01].....	14
Figure 2.3 Hiérarchie des classes de CType [AROM01].....	16
Figure 3.1 Modélisation UML pour le patron Fabrique abstraite [GAMM95].....	19
Figure 3.2 Modélisation UML pour le patron Pont [GAMM95] .....	21
Figure 3.3 Modélisation UML du patron Adaptateur (version héritage multiple) [GAMM95] .....	22
Figure 3.4 Modélisation UML du patron Adaptateur (version délégation) [GAMM95].....	22
Figure 3.5 Modélisation UML du patron Chaîne de responsabilités [GAMM95].....	24
Figure 3.6 Traitement d'une demande selon le patron Chaîne de responsabilités [GAMM95] .....	24
Figure 4.1 Représentations en intesion et extension d'un ensemble de nombres rationnels ...	29
Figure 4.2 Entité Masquable et Représentations d'Entité Masquable [VILL02].....	30
Figure 4.3 Mécanismes de masquage et de dévoilement appliqués aux REM [VILL01].....	30
Figure 4.4 Concepts du Modèle des Fonctionnalité obtenus à partir des cas d'utilisation .....	32
Figure 4.5 Modélisation d'une Fonctionnalité de Consultation avec Accès Progressif [VILL02] .....	33
Figure 4.6 Modèle des Utilisateurs : distinction entre Groupes et Utilisateurs basée sur les concepts du Modèle des Fonctionnalités [VILL02].....	34
Figure 4.7 – Le Modèles des Utilisateurs : profils décrits pour les groupes et les utilisateurs	35
Figure 5.1 Diagramme des classes pour le module d'utilisateurs .....	39
Figure 5.2 La relation entre l'élément et l'entité masquable dans le module d'accès progressif .....	44
Figure 5.3 Diagramme de classes concernant la définition d'une fonctionnalité de consultation (Report) mettant en place les requis pour l'accès progressif .....	49
Figure 5.4 Diagramme de composition de l'espace fonctionnel et des rôles fonctionnels supportant un mode d'accès progressif .....	51
Figure 5.5 Organisation logicielle d'un système KIWIS .....	53
Figure 5.6 Organisation des bases des connaissances d'un système KIWIS .....	55
Figure 5.7 Base de connaissances utilisée pour la sauvegarde du système.....	56

Figure 5.8 Base de connaissances pour le module des fonctionnalités KIWIS ..... 57

# INTRODUCTION

---

## Structure d'accueil

Créé le 1 janvier 1995, le Laboratoire Logiciels Systèmes Réseaux (LSR) a le statut d'Unité Mixte de Recherche (UMR 5526). Ses tutelles sont le CNRS, l'Institut National Polytechnique de Grenoble et l'Université Joseph Fourier. Le LSR est l'un des sept laboratoires de la fédération de recherche en Informatique et Mathématiques Appliquées de Grenoble (IMAG : FR 0071).

Le LSR organise ses activités autour des axes suivants :

- Ingénierie des grands logiciels
- Conception et validation des logiciels
- Applications de la programmation logique avec contraintes
- Services base de données sur le réseau
- Modélisation des systèmes d'information
- Ingénierie des réseaux et du multimédia

## Equipe d'accueil

L'équipe d'accueil au sein du LSR est l'équipe SIGMA qui réalise des recherches appliquées sur l'ingénierie des systèmes d'information. Ces recherches sont centrées sur la formalisation, la conception et l'organisation des Systèmes d'Information. Les résultats attendus de ces recherches sont des modèles, des langages, des outils et des démarches afin d'appuyer des raisonnements à différents niveaux qui favorisent principalement quatre aspects : présentation multimédia adaptée, flexibilité, traçabilité et réutilisation.

Les principaux thèmes de recherche de l'équipe sont :

1. Ingénierie des systèmes d'information,
2. Formalisation, multimodélisation et métamodélisation,
3. Réutilisation et traçabilité,
4. Systèmes d'information multimédias distribués et Web,
5. Applications industrielles et systèmes de gestion de données multimédias.

L'équipe SIGMA est organisée autour de 2 axes :

1. axe composant et réutilisation qui explore une approche génie logiciel à base de patrons dans la conception des systèmes d'information
2. axe multimédia qui explore la conception des systèmes d'information multimédia basés sur le Web

Le stage décrit dans ce rapport a été proposé par le deuxième axe. Il s'inscrit dans la continuation des travaux menés lors de mon stage de magistère première année autour de l'amélioration de l'interface de conception du système KIWIS, un environnement de conception et de déploiement de SIWs (Systèmes d'Information sur le Web).

## Contexte

Avec le développement de systèmes de plus en plus rapides et de plus en plus performants, les informations traitées qui, au départ, n'étaient que textuelles et linéaires, sont maintenant représentées sous forme d'ensembles constitués d'entités atomiques liées entre elles. On parle d'hypertexte. Ce concept fournit une méthode d'accès non séquentielle à l'information, contrairement aux approches traditionnelles. La nature des informations a également évolué. Le concept de multimédia permet de regrouper désormais des données jusqu'alors exploitées séparément : du texte, du son, de l'image, de la vidéo, etc.

Le Web permet d'accéder de façon simple à une multitude d'informations stockées dans des ordinateurs parfois dispersés géographiquement et reliés entre eux par un réseau mondial. Un tel environnement est aussi utilisé dans le cadre d'accès intranet au sein d'un réseau local d'ordinateurs. Le principal avantage d'une telle approche consiste en l'accès simple et standardisé à de nombreuses informations de sources différentes, grâce à des outils qui facilitent ainsi la communication entre les personnes.

Dans ce contexte, et face à l'intérêt grandissant des utilisateurs pour l'Internet, il est apparu nécessaire pour l'équipe SIGMA, de disposer d'une architecture et d'une méthode de conception de systèmes d'information (SI), d'une part, basés sur le Web et, d'autre part, prenant en compte la dimension *adaptabilité*.

L'adaptabilité d'un SI se définit par sa capacité à adapter les informations (contenu), leur forme (présentation, navigation, etc.), ainsi que les services (consultation, modification, etc.) qu'il propose aux besoins et aux caractéristiques individuelles et collectives des utilisateurs.

Le travail de l'axe multimédia de l'équipe SIGMA consiste à proposer à la fois différents modèles d'aide à la conception de Systèmes d'Information Multimédia basés sur le Web (SIMW), et à proposer un outil implantant la méthode. Notamment, l'accent est mis sur l'adaptabilité du SIMW à ses utilisateurs. L'idée est que le SIMW doit fournir non seulement une information pertinente mais que l'accès à celle-ci soit progressif afin d'éviter que l'utilisateur ne se perde dans un espace d'information hypermédia trop dense. C'est pourquoi l'équipe SIGMA propose que la conception d'un SIMW repose sur quatre modèles distincts mais liés : un *modèle de données* chargé de la description du domaine d'application, un *modèle de l'utilisateur* chargé de la description des droits d'accès et des préférences en matière de présentation de l'information, un *modèle pour l'accès progressif* et personnalisé à l'information, et un *modèle de présentation* permettant la personnalisation graphique des

pages Web qui sont dynamiquement générées par le SIMW en réponse aux requêtes de l'utilisateur.

Ces spécifications ont servi à l'élaboration du cahier des charges d'un générateur automatique de systèmes d'information multimédia basés sur le Web, appelé KIWIS. KIWIS se présente comme un serveur Web permettant de mettre en œuvre des modèles présentés ci-dessus. Une fois les modèles instanciés, KIWIS se charge du déploiement du serveur Web sur lequel sera accessible le SI ainsi conçu.

Le projet KIWIS (Knowledge for Improving Web Information System) a débuté en septembre 1999 au sein de l'équipe SIGMA du laboratoire LSR de l'IMAG, avec le démarrage de la thèse de Marlène Villanova. Son objectif est de fournir un ensemble de modèles et une architecture générique pour la conception de systèmes d'information (SI) multimédias adaptables et basés sur le Web. Dans le cadre de sa thèse [VILL02] Marlène Villanova, a proposé une démarche méthodologique de conception de SIMW reposant sur ces quatre modèles.

## **Travail à réaliser**

Le but de ce stage est de proposer un noyau logiciel qui correspond aux spécifications actuelles des modèles fondamentaux de KIWIS.

La principale caractéristique attendue de la réalisation de ce noyau est de permettre la construction d'applications KIWIS capables de mettre en place un accès graduel à l'information.

Cette capacité doit être doublée d'une flexibilité du système en ce qui concerne l'inclusion de nouveaux modèles dans le noyau logiciel.

Le respect de ces caractéristiques nous permettra de considérer le système KIWIS comme la plateforme centrale de validation et de déploiement des divers travaux réalisés au sein de l'équipe.

## **Plan du mémoire**

La première partie de ce mémoire, commence par la présentation de l'état de la plateforme KIWIS précédemment à ce travail. Elle continue avec la présentation d'une alternative en ce qui concerne la représentation interne des données véhiculées par le système KIWIS.

Cette alternative est constitué par la plateforme AROM qui offre une solution Java pour la gestion complète de cycle de vie des données : la modélisation, l'instanciation, l'utilisation et la destruction.

La troisième section de cette première partie du mémoire est consacrée à la présentation de quelques patrons de conception susceptibles d'aider à la construction d'un noyau KIWIS conforme à nos attentes : extensible et flexible. En effet, les patrons de conception offrent des solutions extensibles, flexibles et réutilisables pour des problèmes qui apparaissent souvent lors de l'étape de réflexion précédant une réalisation logicielle.

La deuxième partie de *Spécifications* correspond à la présentation des modèles que nous envisageons de mettre en place dans cette première version du noyau KIWIS : le modèle du domaine, le modèle des fonctionnalités, le modèle d'accès progressif et le modèle utilisateur.

La partie *Réalisation* du mémoire présente le noyau KIWIS, découpé dans une série de module correspondant en grande partie aux modèles fondamentaux. Chaque module met en place des interfaces qui dictent la manière dont les interactions se font entre les entités du systèmes, ou bien entre les entités du systèmes et des clients extérieurs.

Une présentation de d'implémentation réalisée pour ce noyau clôt cette dernière partie. L'implémentation s'appuie sur la plateforme AROM qui est utilisée pour stocker les données persistantes éparpillées dans les divers modules mis en place par le noyau.

Nous proposons un bilan du travail effectué, ainsi que les perspectives ouvertes par la réalisation du noyau KIWIS pour conclure ce mémoire.

# ETAT DE L'ART

---

Nous consacrons la première partie de l'état de l'art de ce mémoire à la présentation de l'état actuel de KIWIS et notamment à la mise en œuvre de l'accès progressif dans les applications générées par KIWIS.

Nous continuons l'état d'art en présentant le système AROM, un outil de modélisation et représentation de connaissances à objets, qui constitue une solution potentielle pour la représentation des informations dans les applications KIWIS.

Nous abordons, enfin, dans cet état d'art les patrons de conception (en anglais **Design patterns**). Ces patrons constituent des points d'appuis indispensables pour une réalisation logicielle qui doit être caractérisée par un degré élevé de flexibilité et de réutilisabilité. Ces propriétés doivent être observées dans le noyau logiciel que nous souhaitons réaliser, d'où notre intérêt pour les patrons de conception.

# 1. KIWIS

---

---

Avant de commencer les travaux dans le cadre de ce stage, la plateforme KIWIS se présentait comme une application Web, capable de générer sur le serveur d'origine des systèmes d'information accessibles sur le Web.

Les systèmes d'information sont définis par un ensemble de pages Web, générées dynamiquement en fonction de l'information diffusée et de l'utilisateur auquel elles sont adressées.

La réalisation de la plateforme repose sur cinq modèles, chacun régissant une dimension distincte d'un système d'information :

- le modèle du domaine
- le modèle de fonctionnalités
- le modèle utilisateur
- le modèle hypermédia
- le modèle d'accès progressif

Néanmoins, au niveau logiciel on ne dispose pas de cinq modules distincts chacun correspondant à un modèle. Cette implémentation de KIWIS souffre en termes de modularité et réutilisation, car on dispose d'une application fermée, qui n'est pas conçue de manière à ce que l'on puisse ajouter de nouveaux modules.

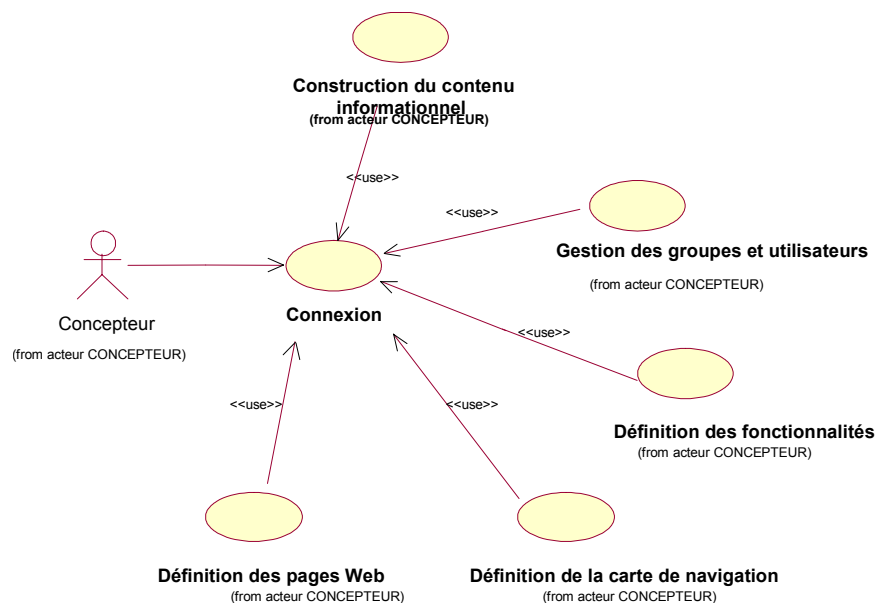
La mise en page des informations diffusées suit une charte de composition personnalisée par chaque utilisateur du système. Les pages Web disposent aussi de mécanismes (liens hypermédia, scripts JavaScript) de navigation permettant d'accéder à des pages présentant le même contenu sémantique, d'une manière plus ou moins détaillée selon le niveau d'accès.

## 1.1. Description UML des besoins fonctionnels

Ce prototype s'adresse principalement à deux acteurs : tout d'abord le concepteur pour définir et concevoir un nouveau SIW, ensuite l'utilisateur final pour accéder au nouveau SIW.

- Acteur « concepteur » : personne qui spécifie le SI en exécutant les actions suivantes :
  - Construction du contenu informationnel du SI (modèle et instances)
  - Déclaration de groupes et d'utilisateurs
  - Conception/Personnalisation des pages Web

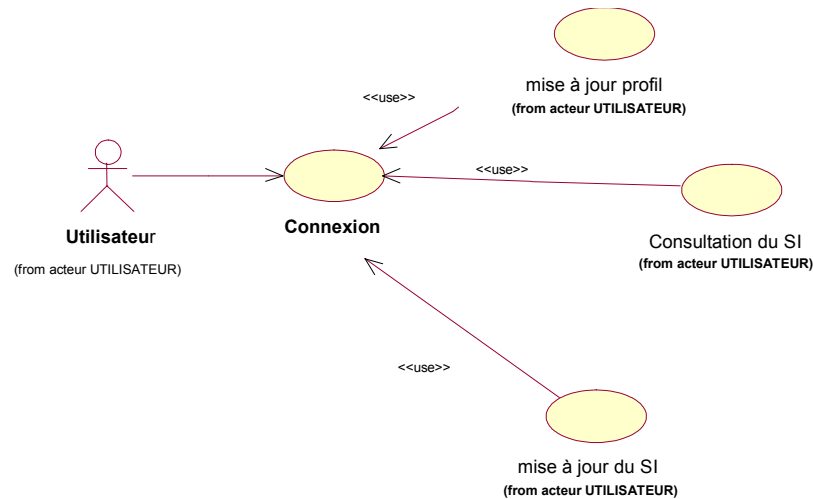
Le cas d'utilisation principal est l'entrée dans le système, après identification du concepteur du système. Ce cas d'utilisation est utilisé par les cas d'utilisation de la Figure 1.1.



**Figure 1.1 Diagramme des cas d'utilisation pour l'acteur CONCEPTEUR**

- Acteur « utilisateur » : personne qui utilise le système pour les objectifs suivants :
  - Consulter les données du système d'information
  - Mettre à jour le système, selon ses droits
  - Mettre à jour son profil

Comme pour le concepteur, le cas d'utilisation principal est l'entrée dans le système. Ce cas d'utilisation est utilisé par les cas d'utilisation de la Figure 1.2.



**Figure 1.2 Diagramme des cas d'utilisation pour l'acteur UTILISATEUR**

## 1.2. Organisation de la plateforme KIWIS

Dans la première version de KIWIS, les modèles se reflètent dans un ensemble d'objets JAVA, constituant les servlets [DAVI99], composantes logicielles côté serveur qui permettent de traduire une requête client en une suite d'opérations portant sur les divers modèles sous-jacents.

Généralement chacune des servlets correspond à un cas d'utilisation parmi ceux présentés dans la section précédente (cf. Figure 1.1 et Figure 1.2). Naturellement, il y a alors des servlets dont les actions portent sur plusieurs modèles du système. Par exemple, la servlet correspondant au cas d'utilisation *Définition des fonctionnalités* (cf. Figure 1.1) porte sur le modèle de fonctionnalités, sur le modèle utilisateur et sur le modèle d'accès progressif. Néanmoins, il n'existe pas de connexion au niveau logiciel (partage des objets, appels de méthodes statiques) entre les servlets dont les opérations atteignent le contenu de plusieurs modèles.

La concertation entre les différentes servlets n'est assurée que par des conventions de représentation et d'exploitation des données du système, conventions propres à cette implémentation particulière.

Les diverses données véhiculées par le système sont représentées en XML [XML98], chaque modèle disposant d'un fichier XMLSchema [XSCH01] caractérisant la représentation du format de sauvegarde correspondante. Le choix d'une représentation XML a comme principal avantage le fait que les données associées aux modèles sont facilement exploitables par d'autres applications. Ainsi, la publication des données dans les pages Web se fait de manière immédiate en appliquant des feuilles de styles XSL [XSLT99] en accord avec le modèle hypermédia (présentation), le modèle de fonctionnalités (contenu) et le modèle d'accès progressif (niveau de détail).

Ce couplage fort au niveau des données entre les différentes composantes (servlets) assurant le bon fonctionnement de l'application constitue une barrière dans l'évolution du prototype. En effet, par exemple :

- Changer la représentation du modèle des données à un impact sur tout le système et nécessite une réécriture des composantes l'exploitant.

- Ajouter de nouveaux modèles, dans le contexte actuel, nécessite la réécriture presque intégrale de la plateforme.
- Faire interagir d'autres systèmes avec KIWIS n'est pas une simple tâche. Le système KIWIS est un système clos, le seul moyen de communication est représentées par les pages Web générées par KIWIS, et donc récupérer les données qui sont mis à plat (leur structure interne est anéantie par les feuilles de styles) dans une page Web n'est pas évident.

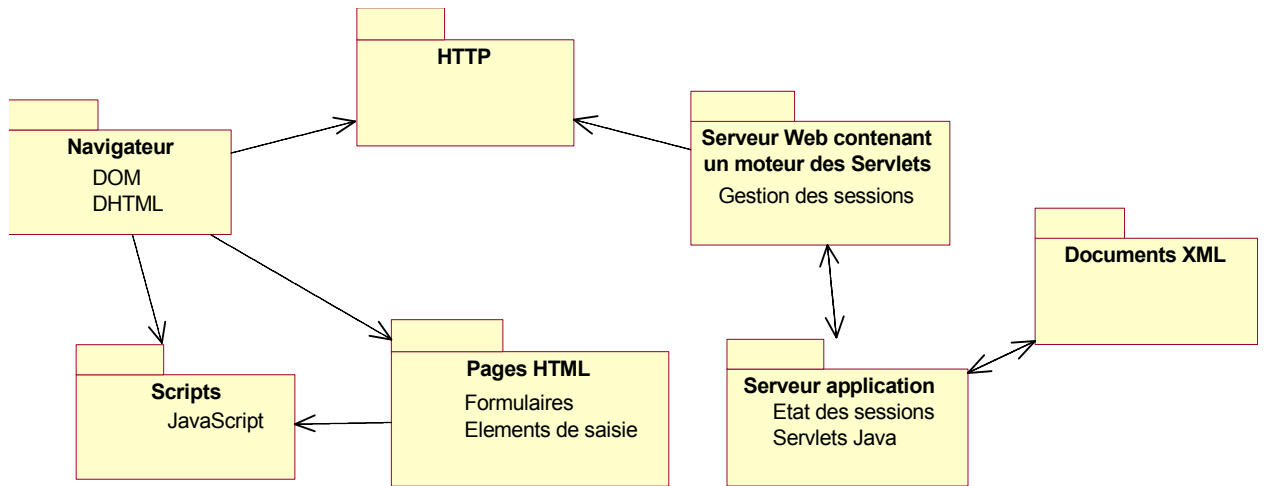
Avant de conclure cette présentation de l'état actuel de la plateforme KIWIS nous présentons son architecture pour compléter ce bref aperçu.

### 1.3. Architecture logicielle de la plateforme KIWIS

Le prototype KIWIS met en œuvre une architecture Web aussi bien pour la partie conception d'un nouveau SI (l'utilisation du DHTML pour la rendre plus conviviale et effectuer la validation des données saisies), que pour la partie utilisation du SI côté client et notamment la prise en compte de l'adaptabilité.

Les composants majeurs de cette architecture sont les suivants (cf. Figure 1.3) :

- le *navigateur client* est un navigateur HTML standard compatible avec les formulaires, acceptant l'utilisation de cookies et de JavaScript,
- le *serveur Web* est le point d'accès principal pour tous les navigateurs clients, et en fonction des requêtes des pages HTML, déclenche des traitements côté serveur (par exemple, la publication de données XML en HTML via l'outil de publication que nous avons conçu),
- le protocole *http* est le protocole utilisé pour réaliser la communication entre les navigateurs client et le serveur Web,
- les *pages HTML* sont des pages Web qui comprennent une interface utilisateur et du contenu (texte, des formulaires de saisie, etc)
- le *serveur application* est le principal exécuter de la logique métier et se trouve sur la même machine que le serveur Web,
- les *scripts* sont des scripts de type JavaScript inclus dans des pages HTML et interprétés par le navigateur,
- les *documents XML* contiennent des données brutes sans indication aucune pour la mise en page.



**Figure 1.3** Vue logique de l'architecture de KIWIS

Lors de ce chapitre nous avons mis en évidence le fait que les données associées aux divers modèles sont représentées directement en XML ce qui ouvre bien des perspectives d'exploitation (illustrée dans la section 1.2). En revanche, le contrôle de l'instanciation et la validation des données saisies dans les divers fichiers XML sont à la charge des servlets qui doivent assurer la cohérence des données. Il est souhaitable que cette charge supplémentaire soit gérée par une application a part entière. C'est dans cette optique que nous présentons dans le chapitre suivant le système AROM. Ce système de représentation de connaissances nous semble en effet pouvoir combler nos besoins en termes de gestion des informations du domaine d'application.

## 2. REPRÉSENTATION DES CONNAISSANCES AVEC AROM

---

---

Pour modéliser des connaissances et/ou des données dans un domaine d'application, les méthodes de conception proposent d'utiliser deux concepts différents:

- des classes d'entités pour décrire les individus ayant des caractéristiques similaires, et,
- des relations ou associations pour regrouper les liens similaires entre ces différents individus.

Dans le système de représentation de connaissances par objets AROM ces concepts sont représentés par des classes regroupant des objets et des associations définissant des liens entre les objets.

La plate-forme AROM est distribuée depuis Janvier 2000, dans sa version 1.0. La version 2.0 est essentiellement un travail de ré-écriture de la première version, destiné à rendre le système AROM plus modulaire et plus extensible.

### 2.1. Représenter des connaissances en AROM

#### 2.1.1 *Classes et objets*

Une classe décrit un ensemble d'objets ayant des propriétés communes. Une classe AROM ne dispose pas de méthode au sens des langages de programmation orientée objet.

Un ensemble de propriétés appelées variables caractérise chaque classe. Cet ensemble de propriétés définit *l'intension* de la classe.

Une variable correspond à une propriété dont le type ne constitue pas une référence vers une des classes de la base de connaissances. En effet, il n'existe pas de variable référençant un ou plusieurs objets; pour réaliser cela il est nécessaire de passer par une association.

Chaque variable est caractérisée par un ensemble de facettes qui peut être subdivisé en trois catégories :

- Les facettes de restriction du domaine des valeurs acceptées par la variable
- Les facettes d'inférence qui permettent d'inférer la valeur d'une variable lorsque celle-ci n'a pas été fixée dans l'objet à l'aide soit d'une valeur par défaut fixée, d'une équation du langage algébrique ou encore d'une méthode Java attachée à la variable chargée de calculer la valeur de celle-ci)
- Les facettes de documentation qui permettent d'associer diverses informations à la variable.

Les classes AROM sont structurées de manière hiérarchique par la relation de spécialisation qui est simple. La spécialisation d'une classe est réalisée soit par ajout ou modification d'une valeur de facette, soit par l'ajout d'une nouvelle variable.

Un objet AROM représente une entité distinguable du domaine modélisé. Chaque objet est attaché à une classe mais appartient également à tous les ancêtres de cette classe.

### **2.1.2 Associations et tuples**

En AROM, les associations jouent un rôle aussi important dans la représentation de connaissances que les classes. Une association représente un ensemble de liens similaires entre  $n$  ( $n \geq 2$ ) classes, distinctes ou non. Un lien est constitué d'un  $n$ -uplet d'objets appartenant aux instances des classes reliées par l'association.

En AROM, chaque association possède un nom. Une association est décrite par ses rôles et ses variables. Les rôles d'une association correspondant à des connexions entre l'association et les classes connectées et les variables caractérisent les propriétés du lien.

Chaque association  $n$ -aire possède donc  $n$  rôles et la valeur de chaque rôle est une instance de la classe correspondante. Chaque rôle possède un nom et une multiplicité. La multiplicité est un intervalle d'entiers indiquant le nombre d'instances que l'on peut associer à ce rôle dans le cadre d'un lien.

Les associations sont organisées en hiérarchies grâce à une relation de spécialisation. La spécialisation d'associations, comme celles des classes, est simple en AROM. Elle permet d'ajouter sur une hiérarchie d'associations un héritage de rôles, de variables et de facettes.

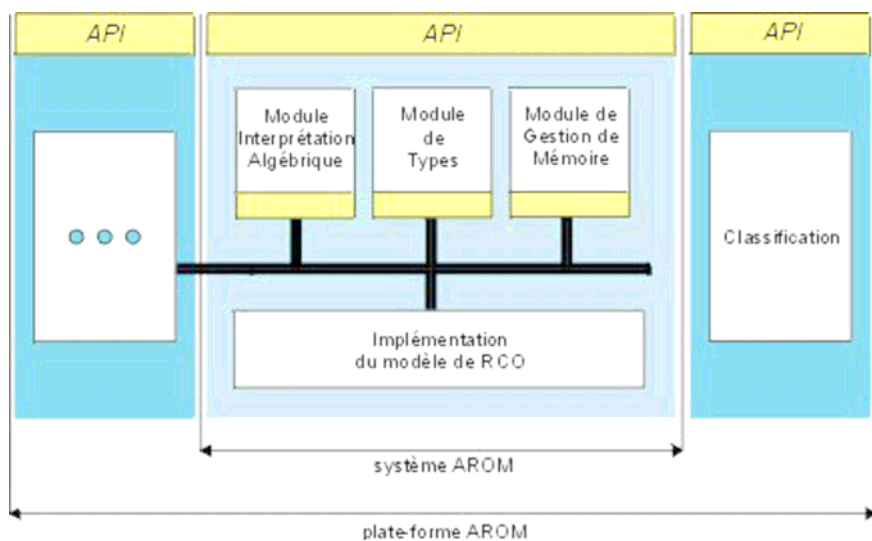
Après avoir donné un aperçu des notions que la plateforme AROM véhicule pour représenter des connaissances nous présentons, dans les sections suivantes, l'architecture et la réalisation logicielle de la plateforme AROM

## **2.2. Architecture de la plateforme AROM**

La plate-forme AROM englobe l'implémentation du modèle de représentation de connaissances que nous venons de brièvement présenter.

La plate-forme AROM est organisée en modules (ou composants). Chaque module ayant en charge la réalisation d'une fonctionnalité précise du système. La plate-forme actuelle est constituée des modules suivants :

- *Module de représentation de connaissances* : ce module gère l'ensemble des structures et des instances appartenant à une base de connaissances.
- *Module de type* : il définit l'ensemble des types reconnus dans une base de connaissances AROM et les opérations possibles sur ces types.
- *Module d'interprétation algébrique* : il assure l'interprétation d'équations algébriques dans AROM.
- *Module de gestion de la mémoire* : il assure le chargement et le déchargement des instances AROM depuis le disque vers la mémoire, afin d'optimiser l'occupation de la mémoire de la machine virtuelle Java lorsque les bases de connaissances comportent un grand nombre d'instances.



**Figure 2.1 Organisation de la plateforme AROM [AROM01]**

Tous ces modules étant indispensables au bon fonctionnement de la plate-forme, il est clair qu'ils dépendent plus ou moins directement les uns des autres. Cependant, les communications entre modules passent par des API définies pour chacun des modules de la plate-forme. De cette manière, la plate-forme AROM est configurable en changeant l'implémentation de l'un des modules par une autre implémentation.

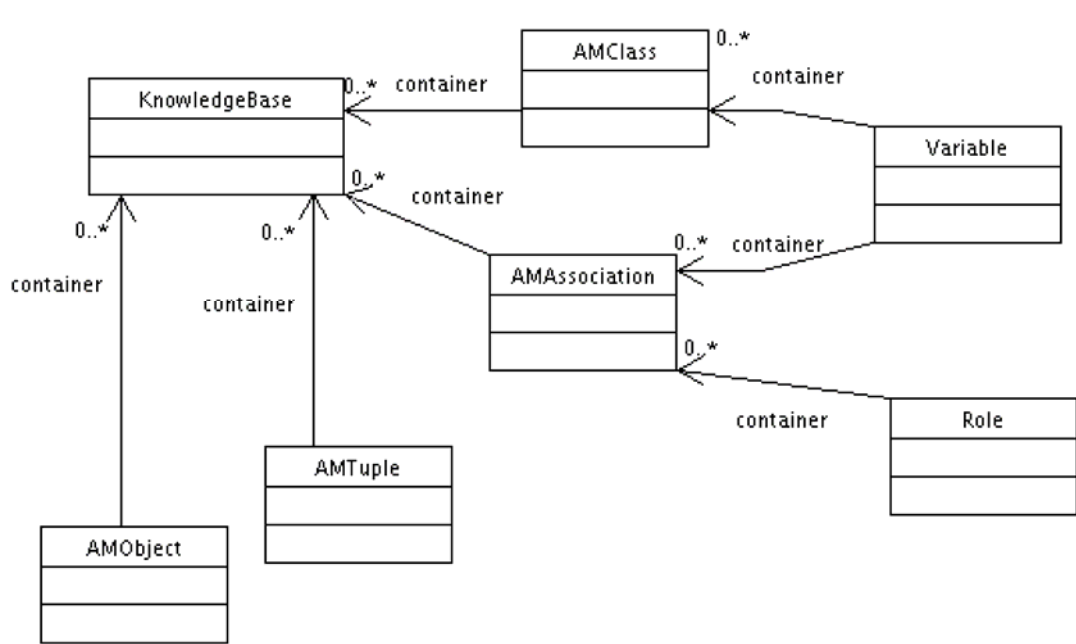
Dans les sections suivantes nous nous intéressons aux modules de représentation de connaissances et des types d'AROM en illustrant les points importants de chacun.

### 2.3. Le module de représentation de connaissances

La plateforme AROM propose à l'utilisateur de manipuler ce que l'on nomme des entités qui sont les éléments constitutifs des bases de connaissances AROM. C'est en créant et en supprimant des entités que l'utilisateur définit une base de connaissances.

Les entités AROM possèdent les deux propriétés suivantes: un *identificateur* et une *documentation*. Chaque entité possède un identificateur unique dans le système, désigné par celui-ci, pour permettre une identification sans équivoque.

La relation de composition qui s'établit entre les entités du système est organisée de la manière suivante:



**Figure 2.2 Relation de composition entre les objets AROM [AROM01]**

Une base de connaissances AROM (*KnowledgeBase*) regroupe les descriptions de toutes les structures (*AMClass*, *AMAssociation*) qui modélisent un domaine précis, ainsi que les instances de ces structures (*AMObject*, *AMTuple*). Chaque structure est constituée d'un ensemble de variables (*Variable*) et, seulement pour les associations, de rôles (*Role*) définissant les liens avec les classes voisines des associations.

### 2.3.1 Classes et Associations

Les entités structurant les bases de connaissances AROM sont les *classes* et les *associations*. Ces structures définissent leurs *intentions* au travers des variables et des rôles.

Les variables et les rôles sont caractérisés par un ensemble de propriétés communes: un nom, un type, etc. Seules les valeurs qui leur seront associées au niveau des instances (objets ou tuples) sont de natures différentes : des valeurs au sens propre pour les variables, respectivement des références vers les objets reliés pour les rôles.

Pour exploiter pleinement cette ressemblance, un concept unifiant les deux est introduit : le *slot*. Le slot est l'entité généralisant la variable et le rôle. Il sera ensuite décliné à l'aide des facettes qui lui seront associées.

#### 2.3.1.1 Slots

Un slot est l'entité AROM utilisée pour décrire l'intention d'une classe ou d'une association. Un slot est caractérisé par : un *nom*, une *valeur* et une *description*. Ces trois caractéristiques sont communes à tous les slots.

Un slot est décrit par un ensemble de *facettes* (type, inférence, documentation, etc ...).

Les descripteurs de slot sont organisés selon la même hiérarchie que les structures pour lesquelles ils décrivent le slot. Par conséquent, une facette étant liée à la description d'un slot, il existe alors plusieurs objets facettes différents pour un même slot : un pour chaque description du slot (donc un pour chaque niveau de la hiérarchie des structures où le slot apparaît).

### 2.3.1.2 Facettes

Les facettes permettent de représenter des propriétés dans AROM. Elles peuvent apparaître à différents niveaux dans AROM :

- La facette de documentation peut être définie pour toute entité AROM.
- La facette de type permet de décrire les slots.
- La facette d'inférence permet de spécifier des valeurs ou méthodes d'inférence pour les variables.
- La facette de contrôle d'instanciation permet de contrôler l'instanciation des structures AROM.

La modification de *l'état d'une facette* se fait en appliquant ou en annulant un *modificateur de facette*. La facette, si elle reconnaît ce modificateur, modifie son état interne pour refléter la modification demandée.

### 2.3.2 Instances AROM

En AROM, chaque structure spécifie l'intention, un ensemble des propriétés la caractérisant au travers de slots.

En parallèle, l'*extension* d'une structure correspond à l'ensemble des individus qui appartiennent à la structure. Ces individus, les *instances*, doivent satisfaire les propriétés de l'intention afin d'appartenir à l'extension d'une structure.

Les *instances* représentent les individus des entités : classes ou associations. Plus particulièrement, les *objets* sont les instances des classes et les *tuples* représentent les instances des associations.

#### 2.3.2.1 Objets AROM

Un objet AROM appartient à une classe spécifique dont l'intention n'est définie qu'au travers de variables. Les objets AROM acceptent des valeurs pour ces variables. Ainsi, un objet qui appartient à une structure définit un ensemble de couples <variable ; valeur>.

Au vu des règles de spécialisation définies dans le modèle AROM, on peut dire que les objets d'une classe sont également objets des super-classes de cette classe.

#### 2.3.2.2 Tuples AROM

On retrouve les mêmes caractéristiques pour les tuples que celles décrites ci-dessus pour les objets. En revanche, l'intention des associations AROM est définie au travers de variables et de rôles. En conséquence, les tuples doivent non seulement définir des valeurs pour les variables, mais ils doivent également référencer les objets pour les différents rôles.

## 2.4. Module de types

Le module de types définit l'ensemble des types de données qui pourront être utilisés dans une base de connaissances AROM, pour caractériser une structure. Des opérations sur ces types de données sont également proposées par le module de types. Le module de types d'AROM considère deux niveaux de représentation des types :

- Un *C-type* (Classe de types) représente l'ensemble, fini ou non, des valeurs partageant une même structure, tel que l'ensemble des réels ou l'ensemble des chaînes de caractères. Chaque C-type définit des opérations qui lui sont applicables.
- Les  $\delta$ -types, quant à eux, permettent de représenter des sous-ensembles des C-types. Chaque  $\delta$ -type est associé à un C-type qui représente le type principal de celui-ci. Les  $\delta$ -types contiennent des informations concernant la restriction du domaine défini par le C-type.

Des classes de C-types sont prédéfinies et organisées hiérarchiquement afin de regrouper les C-types selon leurs caractéristiques (cf. Figure 2.3).

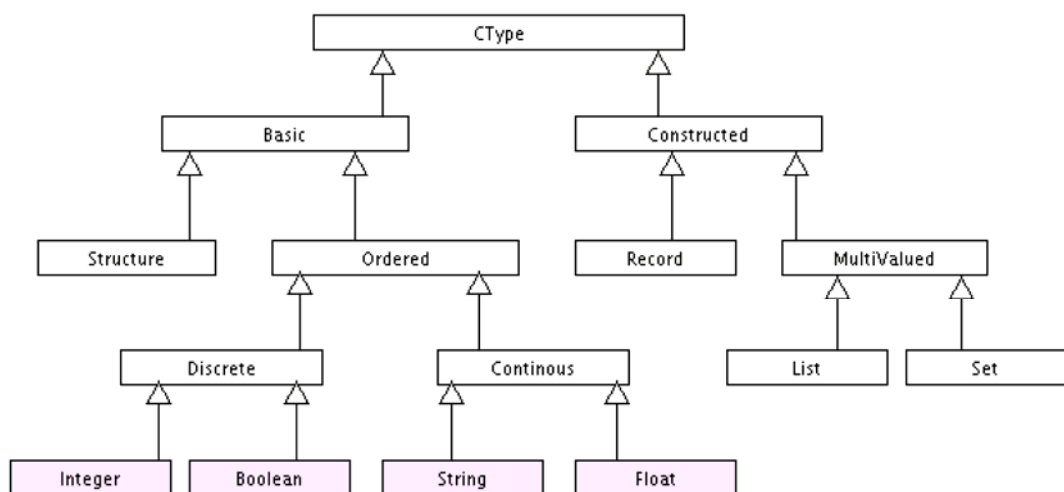


Figure 2.3 Hiérarchie des classes de CType [AROM01]

Seuls les C-types simples, sont définis dans le module de types et peuvent être utilisés afin de typer les variables AROM et ils sont accessibles via leur nom :

- *string* qui représente l'ensemble des chaînes de caractères.
- *boolean* qui représente l'ensemble des booléens.
- *float* qui représente l'ensemble des réels.
- *integer* qui représente l'ensemble des entiers.

Des C-types construits peuvent également être utilisés dans AROM. Les C-types construits principaux sont les *ListCT* et les *SetCT* qui représentent respectivement des listes et des ensembles de  $\delta$ -types.

La restriction d'un ensemble de valeurs, ce qui correspond à l'introduction de nouveaux  $\delta$ -types, est réalisée par l'intermédiaire de deux descripteurs: *set* ou *interval*. Le nouvel ensemble de valeurs, associé à une variable, correspondra au domaine spécifié par le descripteur d'intervalle ou d'ensemble. Un seul de ces descripteurs est pris en compte. L'application d'un descripteur entraînera la suppression du descripteur précédent.

La plateforme AROM constitue un candidat sérieux pour représenter les données véhiculées par le système KIWIS. Les capacités de la plateforme lui permet de gérer le cycle de vie des données : la modélisation et l'instanciation, l'utilisation et la destruction. Ainsi, un implémentation du système KIWIS qui s'appuie sur la plateforme AROM n'est plus concernée par la gestion de cycle de vie de données, puisque les tâches afférentes à cette gestion peuvent être déléguées à la plateforme AROM.

Après avoir statuer sur la représentation des données d'un futur système KIWIS, dans le chapitre suivant nous nous intéressons à des techniques de conceptions. Les patrons de conceptions constituent une source de connaissances pour les concepteurs des réalisations logicielles. Ils proposent des solutions extensibles et réutilisables pour des classes de problèmes auxquelles on se confronte couramment dans l'effort de conception d'une application. C'est dans cette idée que nous étudions dans ce mémoire quelques patrons de conception susceptibles de nous être utiles lors de l'ébauche du noyau KIWIS.

### 3. PATRONS DE CONCEPTION

---

---

En général, un patron de conception est caractérisé par les éléments suivants :

1. Le *nom* du patron permet de décrire un problème de conception, les solutions envisagées et leurs conséquences de la manière la plus succincte qui soit. Le nom fait ainsi référence à des décisions de conception et à leurs conséquences, facilitant le processus de réflexion autour des patrons, en simplifiant notamment la communication.
2. Le *problème* décrit de manière abstraite une situation où le patron est applicable. Il peut souligner l'existence des problèmes de conception tel que : comment représenter des algorithmes en tant qu'objets, ou mettre en évidence une structuration des classes et des objets ayant un degré élevé d'inflexibilité. Le *problème* inclut une liste des conditions qui doit être vérifiée avant qu'on applique la solution proposée par le patron de conception.
3. La *solution* regroupe l'ensemble de décisions de conception, des structures et objets mise en oeuvre, leurs relations, leurs responsabilités et leur collaboration. La *solution* ne décrit pas une réalisation particulière, puisque le patron doit jouer le rôle d'un modèle générique qui peut être appliquée dans des diverses situations. Un patron fournit une description abstraite d'un problème de conception et une description générale des éléments et de leur organisation permettant d'offrir une réponse, caractérisée par la flexibilité et la réutilisable, au problème en question.
4. Les *conséquences* sont les résultats des compromis issus de la mise en oeuvre de la solution générale du patron. La plupart de temps, elles ne sont pas discutées en compte de manière approfondie, mais elles représentent des points critiques dans l'évaluation des alternatives de décision de conception. En termes logiciels, les conséquences se traduisent en espace mémoire et temps d'exécution. la réutilisation est un facteur important dans la conception orientée objet, les conséquences d'un patron incluent aussi l'impact sur la flexibilité, extensibilité et la portabilité du système.

Le *nom* d'un patron de conception abstrait et identifie les aspects clés d'une décision de conception qui correspond à la création d'un système orienté objet réutilisables. Les patrons de conception dérivent les classes et les instances, leurs rôles et les relations qu'elles entretiennent, aussi bien que leurs responsabilités.

Pour résumer cette brève introduction sur les patrons, un patron décrit :

- les situations dans lesquelles il peut s'appliquer
- s'il peut s'appliquer compte tenu d'autres décisions de conception présentes
- les conséquences et les compromis induits par son usage.

Dans la suite de la section, nous présentons les patrons de conception, susceptibles de nous servir pour la création d'un noyau logiciel du système KIWIS réutilisable et extensible.

### 3.1. Fabrique abstraite (Abstract Factory)

La Fabrique abstraite (Abstract Factory) fournit une interface capable de créer une famille d'objets sans qu'on soit amené à spécifier les classes concrètes auxquelles ces objets appartiennent. Cet aspect est très important car il permet de découpler le comportement proposé par un objet (ses méthodes) de sa réalisation effective, ce qui se traduit par un important gain en terme de flexibilité.

#### 3.1.1 Problème

La mise en œuvre du patron de conception Fabrique Abstraite est recommandé, notamment, dans des situations où :

- le système doit être indépendant de la manière dont ses produits (instances et classes) sont créés, assemblés et représentés
- on veut fournir une bibliothèque des classes en rendant public seulement les interfaces implémentées, et pas leurs implémentations.

#### 3.1.2 Solution architecturale

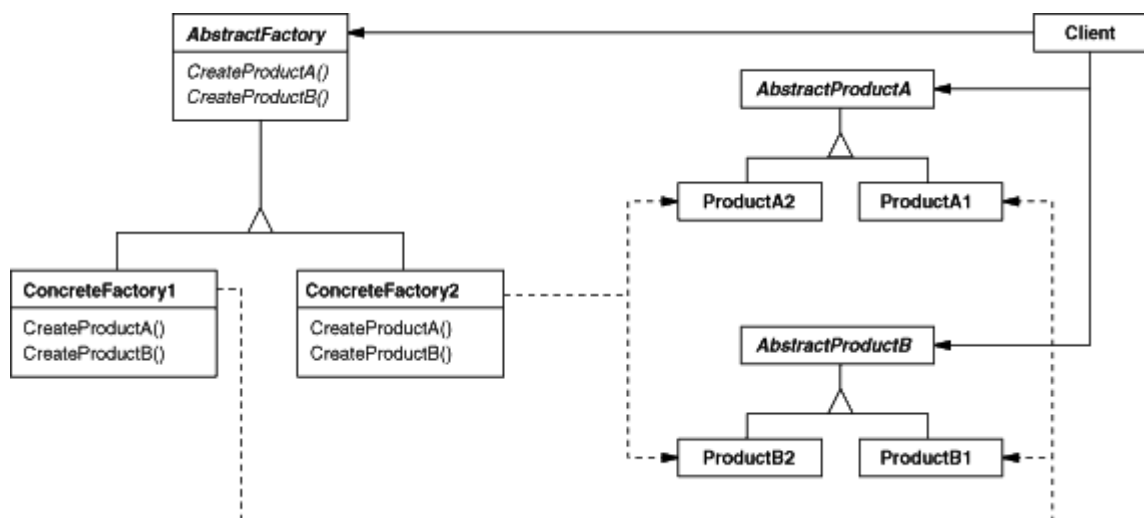


Figure 3.1 Modélisation UML pour le patron Fabrique abstraite [GAMM95]

La fabrique abstraite *AbstractFactory* déclare une interface contenant les opérations qui créent des objets conformes aux descriptions abstraites, c'est-à-dire aux interfaces (*AbstractProductA*, *AbstractProductB*), de chaque produit pris en charge par la fabrique. Des réalisations concrètes de la fabrique abstraite (*ConcreteFactory1*, *ConcreteFactory2*) permettent de basculer entre les différentes réalisations (*ProductA1*, *ProductA2*) d'un produit abstrait (*AbstractProductA*).

### 3.1.3 Conséquences

Cette architecture permet d'isoler les classes concrètes implémentant les produits abstraits véhiculés par les clients. On obtient ainsi un plus en termes de flexibilité et portabilité. Les applications client ne souffrent d'aucune manière si des changements au niveau du processus de fabrication apparaissent (remplacement de la fabrique *ConcreteFabrique1* par *ConcreteFabrique2*) ou si de nouvelles versions (familles de produits) pour de nouvelles plateformes matérielles sont créées, puisque les objets sont décrits selon leurs interfaces abstraites.

En général, une seule instance d'une classe fabrique existe au moment de l'exécution. Cette fabrique est responsable de la création des objets selon une implémentation particulière. La fabrique peut créer elle-même ces objets ou peut déléguer la responsabilité à des sous-fabriques enregistrées auprès d'elle.

L'un des peu rares inconvénients de ce patron est le manque d'extensibilité. En effet, un fois l'interface de la fabrique abstraite définie, ajouter de nouveaux types de produits n'est pas une tâche facile. Puisque l'interface de la fabrique fige l'ensemble des produits, ajouter de nouveaux types, implique d'étendre cette interface et toutes les classes fabriques concrètes sous-jacentes.

## 3.2. Pont (Bridge)

Ce patron de conception introduit un découplage entre l'abstraction d'un concept et son implémentation de façon à ce que les deux puissent être modifiées de manière indépendante.

En général, lorsque pour une interface on dispose de plusieurs implémentations, celles-ci dernières sont réalisées en héritant de l'interface. L'héritage lie une implémentation à l'interface de manière définitive ce qui rend difficilement réalisables les modifications, les extensions et les réutilisations de l'abstraction de celles de l'implémentation indépendamment.

### 3.2.1 Problème

La mise en oeuvre du patron Pont est recommandée notamment lorsque:

- On veut éviter une liaison figée entre les interfaces et les implémentations, notamment dans les situations où l'implémentation est sélectionnée ou remplacée à l'exécution.
- Les interfaces et les implémentations doivent être extensibles par héritage
- Les changements dans l'implémentation ne doivent pas induire des changements au niveau des clients

### 3.2.2 Solution architecturale

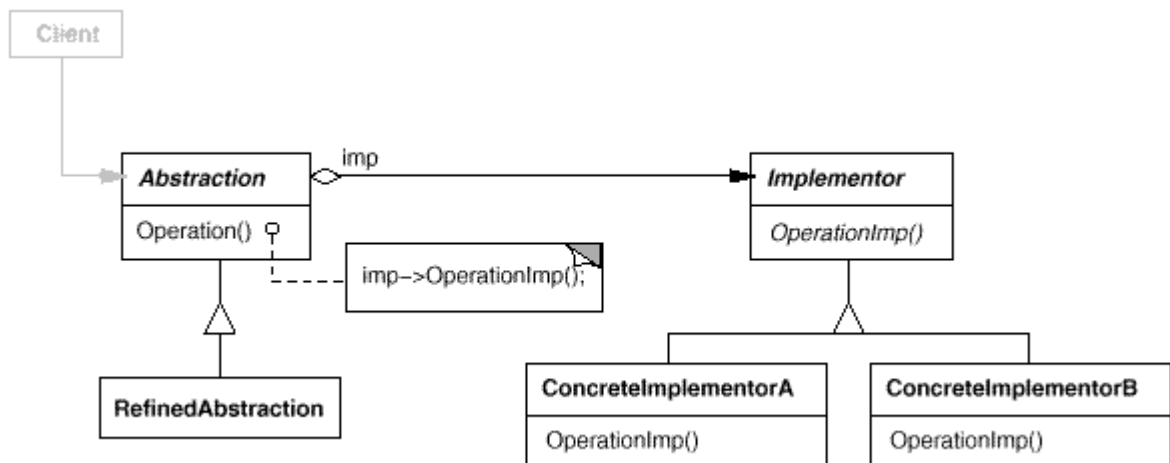


Figure 3.2 Modélisation UML pour le patron Pont [GAMM95]

L'interface *Abstraction* définit les opérations spécifiques aux produits. Les objets de type *Abstraction* maintiennent une référence à un objet de type *Implementor*. L'interface *Implementor* définit l'interface des classes d'implémentations. Celle-ci n'est pas tenue à correspondre de manière exacte à l'interface *Abstraction*. Il est souhaitable que l'interface *Abstraction* corresponde à la définition des opérations de haut niveau et l'interface *Implementor* à des primitives.

Dans cette configuration, l'inexistence d'un lien d'héritage entre les deux interfaces permet d'étendre l'une ou l'autre de manière indépendante sans que cela se répercute sur les applications clientes.

En pratique les objets implémentant l'interface *Abstraction* délègue l'exécution des opérations à l'objet *Implementor* sous-jacent.

### 3.2.3 Conséquences

Cette solution architecturale, le patron Pont, permet de découpler les interfaces et les implémentations. Ainsi on peut changer l'implémentation associée à un objet même à l'exécution, éliminant les contraintes de dépendances au moment de la compilation.

## 3.3. Adaptateur (Adaptor)

Ce patron de conception offre une Solution architecturale pour convertir l'interface d'une classe en une autre conforme aux attentes du client. Les adaptateurs permettent aux classes ayant des interfaces incompatibles de travailler ensemble.

### 3.3.1 Problème

La mise en oeuvre du patron Adaptateur se prête dans les situations suivantes :

- réutilisation une classe existante, dont l'interface ne correspond pas à celle escomptée

- on a besoin d'utiliser plusieurs sous-classes existantes, mais l'adaptation de leur interface par dérivation de chacune d'entre elles est impraticable. Un adaptateur objet peut adapter l'interface de sa classe parente.

### 3.3.2 Solution architecturale

Une première variante (adaptateur des classes) de ce patron est illustrée dans la Figure 3.3.

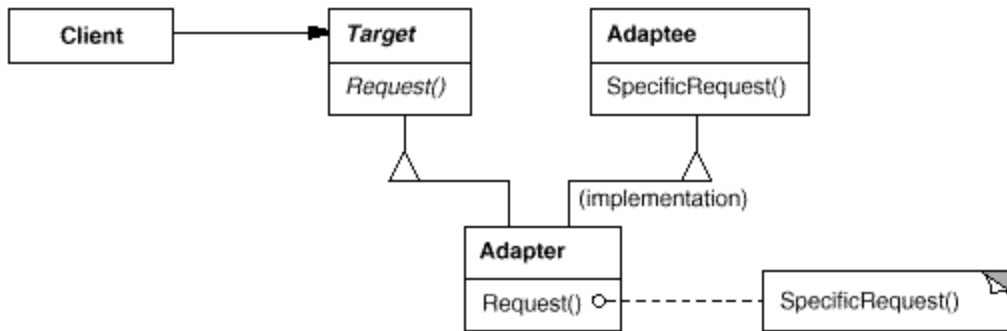


Figure 3.3 Modélisation UML du patron Adaptateur (version héritage multiple) [GAMM95]

Un adaptateur pour les classes utilise l'héritage multiple pour adapter une interface à l'autre. L'interface *Target* correspond à l'interface spécifique requise par les clients. La classe *Adaptee* correspond à l'interface spécifique que l'on veut rendre utilisable auprès des clients.

Les instances de cette dernière classe sont enveloppées dans des objets de la classe *Adapter* qui implémente les deux interfaces en question.

Les appels des clients sur une instance de la classe **Adapter** sont délégués aux opérations de l'interface *Adaptee* qui se charge de les mener à bien.

Une deuxième variante adaptateur des objets de ce patron, spécifique aux langages qui ne supporte pas l'héritage multiple aux niveaux de classes est présentée dans la Figure 3.4.

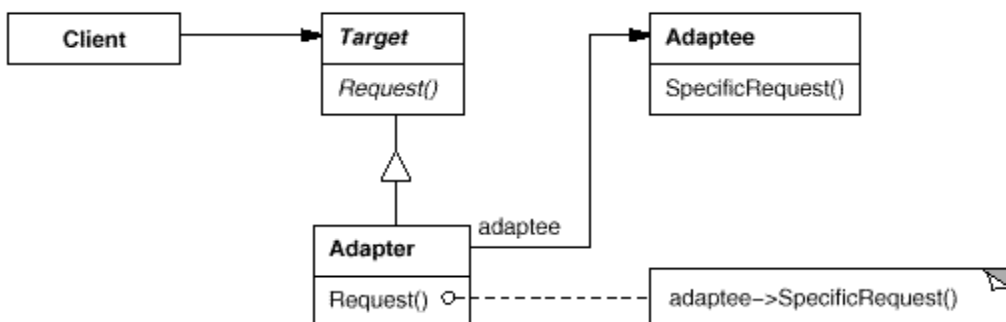


Figure 3.4 Modélisation UML du patron Adaptateur (version délégation) [GAMM95]

Dans ce cadre la classe adaptateur *Adapter* hérite seulement de la interface but **Target** et donc elle n'a pas de lien d'héritage avec la classe à adapter *Adaptee*. En revanche, elle détient une référence vers une instance de la classe *Adaptee* qui est chargé d'assurer la prise en charge d'appels adressés aux adaptateurs.

### 3.3.3 *Conséquences*

Les deux variantes de mise en oeuvre de ce patron que nous venons décrire présentent chacune des avantages et des inconvénients.

Dans le cas d'une solution adaptateur de classe il est impossible avec une seule classe adaptateur de rendre conforme à l'interface *but* la classe et ses sous-classes en même temps. Une classe adaptateur doit être créée pour chaque sous-classe.

Pour ce genre de situation la solution adaptateur d'objets est plus souple. Il suffit de créer une classe adaptateur qui contient une référence vers un objet de la classe racine de la hiérarchie de classes que l'on veut adapter. En fonction du type effectif de l'objet au moment de l'exécution on obtient un adaptateur pour n'importe quelle sous-classe.

Néanmoins, puisqu'il n'y pas de lien hiérarchique entre la classe à adapter et l'adaptateur, cette deuxième solution ne permet pas d'altérer le comportement de la classe à adapter.

Un autre point à prendre en compte est la flexibilité de la construction. L'adaptateur de classe est peu flexible aux changements de l'interface *but* (ajout de méthodes, changement de signature), lorsqu'on est en présence d'une hiérarchie de classes. En revanche, l'adaptateur d'objets résout plus facilement cette situation, puisqu'il n'y a qu'à étendre le seul adaptateur défini pour la hiérarchie entière.

## 3.4. Chaîne de responsabilités (Chain of responsibilities)

Le patron chaîne de responsabilités évite de coupler directement un objet demandeur et celui qui satisfait la requête. Le découplage est réalisé en introduisant un ou plusieurs objets intermédiaires ayant chacun la possibilité de traiter la requête. La requête passe d'un objet au suivant dans la chaîne des objets tant qu'elle n'est complètement résolue.

### 3.4.1 *Problème*

Le patron Chaîne de responsabilités est recommandé dans les situations suivantes :

- Un ou plusieurs objets pourrait avoir à traiter une requête et a priori on ne connaît ni leur nombre, ni leur nature, les objets traitants étant déterminés à la volée
- On veut formuler une requête sans dire explicitement à quelle(s) composante(s) précise(s) du système elle s'adresse.
- L'ensemble potentiel d'objets capable de traiter une requête est constitué de manière dynamique

### 3.4.2 *Solution architecturale*

Une architecture typique, spécifique à ce patron est illustrée dans la Figure 3.5.

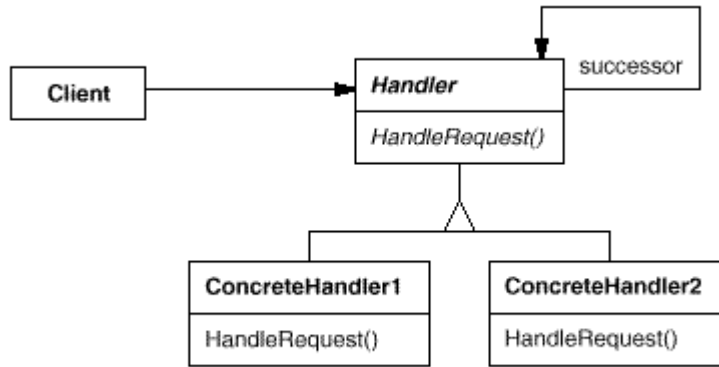


Figure 3.5 Modélisation UML du patron Chaîne de responsabilités [GAMM95]

L'interface *Handler* définit les opérations nécessaires pour traiter une demande. Pour transmettre la demande et assurer les clients qu'une réponse appropriée leur est fournie, les objets formant la chaîne partagent la même interface pour traiter les demandes et font tous référence (*successor*) à l'objet qui les suit dans la chaîne.

Les classes (*ConcreteHandler1*, *ConcreteHandler2*) implémentant l'interface *Handler* répondent aux demandes qu'elles peuvent traiter seules, les autres demandes sont transmises au successeur dans la chaîne (voir Figure 3.6).



Figure 3.6 Traitement d'une demande selon le patron Chaîne de responsabilités [GAMM95]

### 3.4.3 Conséquences

Parmi les avantages de ce patron, on peut inclure le fait que les objets demandeurs ne sont pas tenus de savoir de manière précise quel objet traite explicitement sa demande. D'ailleurs l'objet traitant n'a lui-même aucune connaissance sur l'objet qui a émis la demande. Ainsi, on simplifie les connexions explicites entre objets, connexions qui nuisent à la flexibilité et à la réutilisation d'un système.

Un autre gain correspond au fait que les objets traitants peuvent se distribuer entre eux les responsabilités de manière transparente pour les clients. Cette distribution de responsabilité peut se faire aussi bien de manière dynamique au moment de l'exécution en créant de nouveaux objets traitants, que de manière statique en spécialisant par extension/héritage les classes traitantes (*ConcreteClass1* dans la Figure 3.5).

Ce qui manque à ce patron est une garantie que la demande est résolue. Elle peut être laissée irrésolue en raison d'une mauvaise configuration de la chaîne.

Lors de ce chapitre et du chapitre précédent nous avons présenté les outils logiciels et les techniques que nous envisageons de mettre en pratique pour aboutir à un noyau logiciel flexible et extensible supportant le modèle d'accès progressif que nous avons succinctement

présenté dans le premier chapitre. Nous présentons dans le chapitre suivant de manière plus détaillée nos attentes par rapport à ce noyau, sous forme des spécifications en termes de modèles.



# SPÉCIFICATIONS

---

## 4. LES MODÈLES DANS KIWIS

---

---

Dans ce chapitre nous présentons les modèles qui constituent la fondation de la plateforme.

Au centre de ces modèles se trouve le modèle d'accès progressif qui assure un accès graduel aux informations portées par les différentes entités du système. Dans la suite de ce chapitre nous présentons le modèle d'accès progressif et les modèles qui s'articulent autour de celui-ci : le modèle du domaine, le modèle des fonctionnalités, le modèle utilisateur. Le modèle hypermédia qui complète la proposition ne sera pas discuté ici. Puisqu'il ne constitue pas, à ce jour, un modèle stable, il ne se retrouvera pas dans cette première version du noyau KIWIS.

### 4.1. Le modèle d'accès progressif

La notion d'accès progressif repose sur l'hypothèse suivante: l'utilisateur n'a pas tout le temps besoin d'accéder à toute l'information disponible. A travers l'accès progressif, on peut permettre à l'utilisateur d'accéder, dans un premier temps, aux informations qui sont essentielles pour lui, puis, de façon graduelle, lui donner la possibilité, s'il le souhaite, d'accéder à autres informations.

L'objectif de l'introduction de ce modèle est de tendre vers des systèmes capables de mettre en oeuvre des modalités d'accès progressif.

Le modèle d'accès progressif est défini indépendamment de toute référence aux concepts spécifiques d'un SIW. Nous présentons les notions permettant de comprendre les principes d'une approche basée sur l'accès progressif.

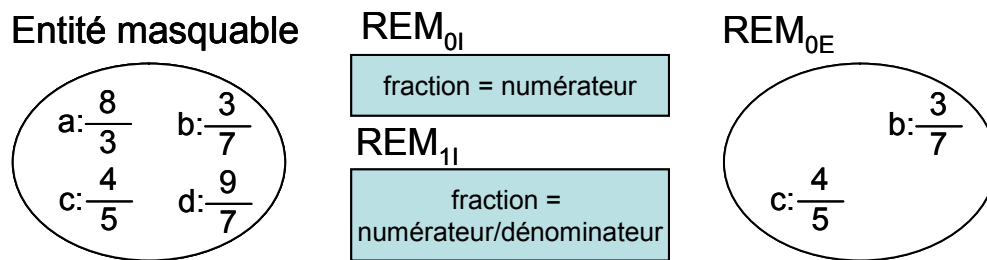
#### 4.1.1 *Entité Masquable et Représentation d'Entité Masquable*

Une Entité Masquable (EM) est un ensemble d'au moins deux éléments (la cardinalité de l'ensemble EM est donnée par  $|EM| \geq 2$ ) sur lequel on souhaite mettre en place un accès progressif. L'accès progressif sur une EM repose sur la définition de Représentations d'Entité

Masquable (REM) associées à cette EM. Les REM sont des sous-ensembles de l'EM ordonnés par la relation d'inclusion ensembliste.

Les REM sont dites extensionnelles lorsqu'elles agissent sur l'extension de l'EM. L'extension est ici définie comme l'ensemble d'éléments de l'Entité Masquable. Dans le cas où l'EM correspond à un ensemble structuré de données, on peut également lui associer un second type de REM, les REM intensionnelles, qui agissent sur la structure de l'entité masquable (i.e. son *intension*).

Pour illustrer ces deux concepts, considérons l'exemple de la Figure 4.1



**Figure 4.1 Représentations en intension et extension d'un ensemble de nombres rationnels**

L'entité masquable EM est constituée des nombres rationnels ( $a$ ,  $b$ ,  $c$  et  $d$ ) représentés sous la forme des couples numérateur/dénominateur. On peut identifier l'intension de la structure nombre rationnel comme étant : le numérateur et le dénominateur. La  $REM_{0I}$ , représente une REM intensionnelle et elle ne présente que le numérateur d'une fraction. La  $REM_{0E}$  représente une REM extensionnelle est elle présente un sous-ensemble de nombres rationnel ( $b$  et  $c$ ) de l'entité masquable.

Chaque REM, indépendamment de sa nature - intensionnelle ou extensionnelle -, est associée à un niveau de détail. On note  $REM_i$  la représentation de niveau de détail  $i$  avec  $1 \leq i \leq max$ , où  $max$  est le niveau maximum de détail défini pour l'EM. Pour une REM, les éléments de l'extension d'une EM présents à un niveau de détail  $i$  sont également présents au niveau supérieur de détail  $i+1$ . De même pour les REM intensionnelles, les éléments structurels d'une EM présents à un niveau de détail  $i$  sont également présents au niveau supérieur  $i+1$ . La différence entre deux niveaux successifs est réalisée par l'ajout d'au moins un élément. Ces éléments ajoutés dans  $REM_{i+1}$  par rapport à  $REM_i$  sont choisis parmi les éléments de EM n'appartenant pas à  $REM_i$ .

La Figure 4.2 illustre les concepts que nous venons de décrire dans le cas de REM extensionnelle, le même principe étant appliqué dans le cas de REM intensionnelles. L'Entité Masquable présentée à gauche de la figure est constituée de divers éléments. A partir de cette EM, trois Représentations d'Entité Masquable, correspondant à autant de niveau de détail sont définies. Au centre, nous symbolisons les ensembles d'éléments ajoutés par chaque REM. On note que l'élément symbolisé par le losange reste toujours masqué : même au plus haut niveau de détail, il ne participe pas à la représentation de l'entité.

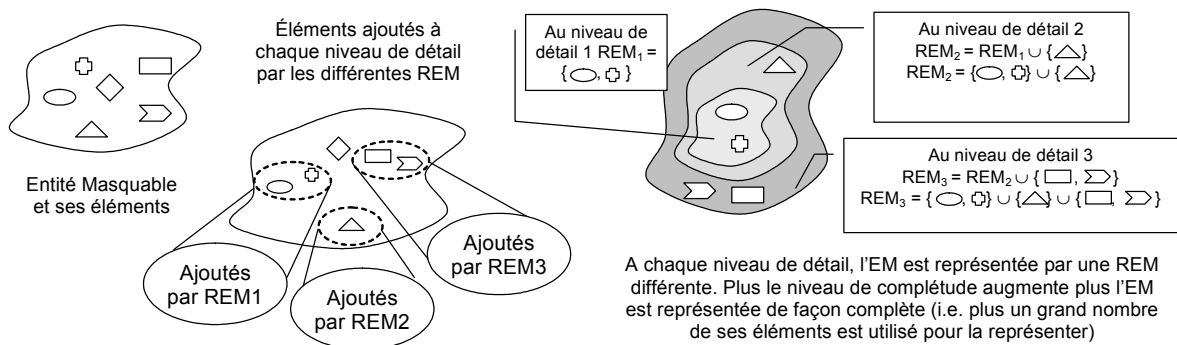


Figure 4.2 Entité Masquable et Représentations d'Entité Masquable [VILL02]

Les REM peuvent donc être considérées comme des masques sur une EM : elles offrent une visibilité sur les éléments à un certain niveau de détail, mais également, elles cachent une partie des éléments. Dans ce sens, les REM présentent une ambivalence leur permettant à la fois de supporter l'accès progressif et d'assurer une certaine confidentialité de l'information.

#### 4.1.2 Stratification

Une *stratification* représente le processus, et son résultat, qui vise à définir une organisation particulière d'un ensemble d'éléments, constituant l'Entité Masquable à travers les REMs.

La *stratification* est construite de telle manière que les Représentations d'Entités Masquables :

- sont ordonnées,
- sont liées par une relation ensembliste d'inclusion stricte,
- sont associées à des mécanismes spécifiques pour un accès progressifs
- permettent de donner une représentation de l'EM à différents niveaux de détail

#### 4.1.3 Fonctions pour l'accès progressif

Deux fonctions d'accès progressif permettent de passer d'un niveau de détail à l'autre. Etant donnée une REM, le masquage donne accès à la REM de niveau de détail immédiatement inférieur lorsqu'elle existe. A l'inverse, le dévoilement permet d'atteindre la représentation de niveau de détail immédiatement supérieur, lorsqu'elle existe.

La Figure 4.3 illustre les principes des fonctions de masquage et de dévoilement en reprenant l'exemple schématique introduit précédemment par la Figure 4.2.

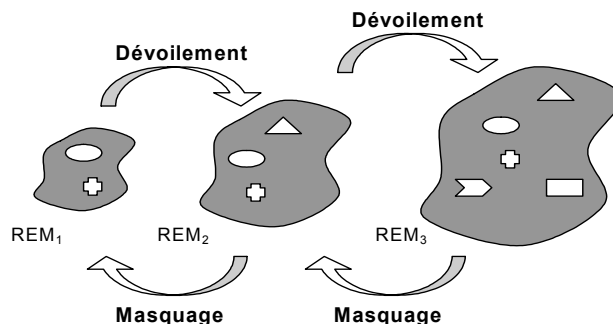


Figure 4.3 Mécanismes de masquage et de dévoilement appliqués aux REM [VILL01]

La suite de ce chapitre présente les différents modèles pour la conception de SIW

Le Modèle du Domaine représente les objets du monde réel qui selon manipulés dans les SIW. Il met en évidence les concepts et relation entre les concepts du domaine d'application.

Le second modèle, le Modèle des Fonctionnalités représente et organise au niveau conceptuel les différentes tâches qu'un utilisateur peut effectuer dans le SIW. Guidée par les besoins des utilisateurs, sa définition est par ailleurs conduite avec un objectif de mise en œuvre de l'accès progressif à différents niveaux. Ces niveaux sont portés par des concepts de granularités différentes : l'Espace fonctionnel, le Rôle Fonctionnel et la Fonctionnalité.

Le dernier modèle que nous discutons ici est le modèle des utilisateurs. Il distingue les notions de groupe d'utilisateurs et d'utilisateur individuel. Ils sont caractérisés par des profils qui constituent le support d'une adaptabilité portant sur l'accès progressif.

## 4.2. Modèle du Domaine

Les objets du monde réel qui participent à la définition du domaine d'application visé par le SIW sont représentés dans le Modèle du Domaine. Ce modèle est avant tout *conceptuel* dans la mesure où il est destiné à mettre en évidence les concepts du domaine et les relations entre ces concepts. Il est ainsi indépendant de toute considération relative à la dimension hypermédia du futur SIW.

De façon similaire, le Modèle du Domaine est ici décrit indépendamment de toute caractéristique liée à l'accès progressif.

Néanmoins, le processus de stratification doit être guidé en vue de proposer des accès progressifs intéressants pour l'utilisateur. Le Modèle de Fonctionnalités, présenté dans la section suivante, établit le lien manquant entre stratifications du Modèle du Domaine et besoins de l'utilisateur.

## 4.3. Modèle des Fonctionnalités

Le Modèle des Fonctionnalités représente et organise, au niveau conceptuel les fonctionnalités qu'un utilisateur peut mener à bien dans le SIW.

Le Modèle des Fonctionnalités repose sur différents concepts qui permettent d'organiser les activités de chaque utilisateur, et ce, de façon adaptée à ses besoins. Cette organisation repose sur trois niveaux et exploite une approche à base de cas d'utilisation pour la définition des trois concepts décrits à la suite de la Figure 4.4

Au plus bas niveau, les *Fonctionnalités* sont décrites. Les fonctionnalités sont identifiées à partir des cas d'utilisation dont les acteurs sont des personnes utilisant le système. À un niveau intermédiaire, les fonctionnalités sont regroupées en différents Rôles Fonctionnels. Les fonctionnalités définies pour un même rôle d'acteur permettent d'obtenir un Rôle Fonctionnel. Au plus haut niveau, un Espace Fonctionnel, défini pour chaque utilisateur, réunit les différents rôles fonctionnels auxquels cet utilisateur peut prétendre. Un utilisateur donné peut exercer le rôle de plusieurs acteurs. L'Espace Fonctionnel d'un utilisateur donné regroupe l'ensemble des Rôle Fonctionnels qu'il est susceptible de remplir.

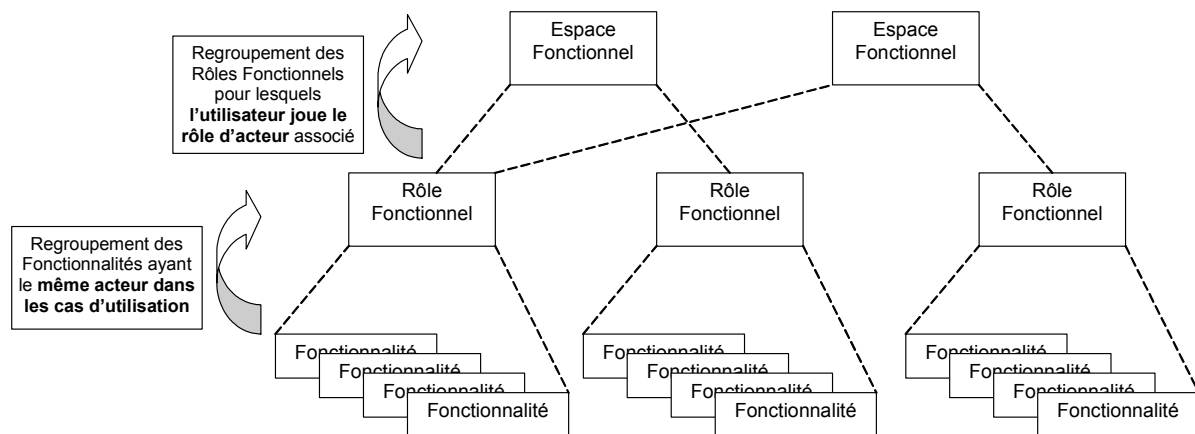


Figure 4.4 Concepts du Modèle des Fonctionnalités obtenus à partir des cas d'utilisation

### 4.3.1 Modèle des Fonctionnalités et Accès progressif

Une telle organisation nous permet d'exploiter les principes de l'Accès Progressif défini par le MAP. Ainsi, un Espace Fonctionnel est vu comme une EM constituée de Rôles Fonctionnels. Les Rôles Fonctionnels sont utilisés pour constituer les représentations d'un même espace à des niveaux de détail différents.

En descendant d'un niveau de granularité, un Rôle Fonctionnel, constitué de Fonctionnalités, peut à son tour être considéré comme une EM. Ses représentations associées à des niveaux de détail différents donnent accès à plus ou moins de Fonctionnalités.

A un niveau encore plus bas de granularité, une Fonctionnalité peut également être stratifiée selon les principes du MAP, mais à quelques particularités près. Nous consacrons la suite de cette section à présenter l'approche que nous proposons pour concevoir des fonctionnalités de consultation avec accès progressif.

### 4.3.2 Fonctionnalité de consultation avec accès progressif

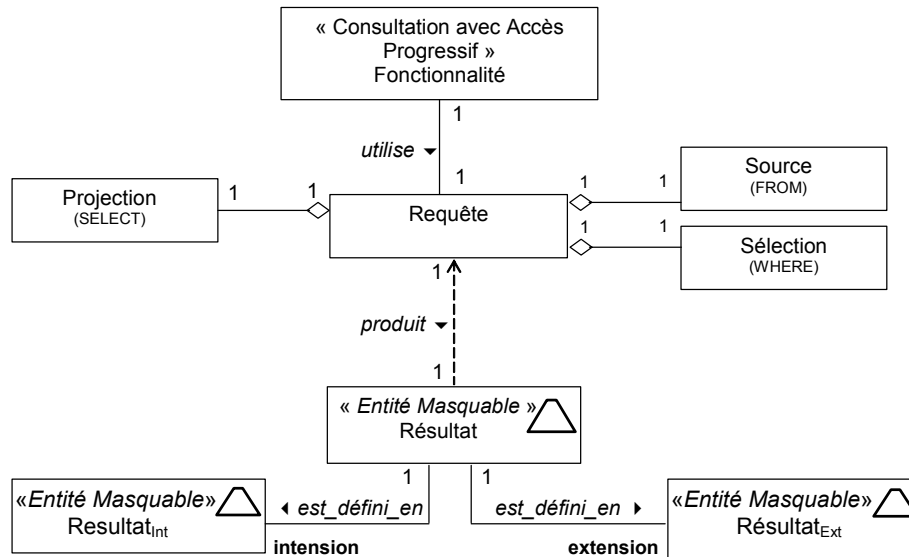
L'accès progressif se révèle essentiellement pertinent pour des fonctionnalités qui permettent à l'utilisateur de *consulter* des informations disponibles dans le SIW. L'accès progressif intervient dans le fait de stratifier l'ensemble d'informations retourné à l'utilisateur afin de lui éviter d'être submergé par celui-ci.

La Figure 4.5 présente la partie du modèle décrivant une fonctionnalité de consultation associée à des mécanismes d'accès progressif. Une telle fonctionnalité de consultation (classe stéréotypée « *Consultation avec Accès Progressif* ») utilise une requête (classe *Requête*) qui permet de sélectionner et de structurer les informations à présenter. La classe *Requête* est modélisée par une composition impliquant plusieurs classes représentant les différentes clauses d'une requête simple.

- La clause *Select* est représentée par la classe *Projection* qui définit la structure attendue du résultat de la requête.
- La clause *From* est modélisée par la classe *Source* qui introduit les classes et les associations dont les instances sont impliquées dans la construction du résultat.

- La clause *Where* est décrite par la classe *Sélection* qui spécifie un prédicat permettant de filtrer les instances.

Une requête produit un résultat structuré selon une classe du même nom.



**Figure 4.5 Modélisation d'une Fonctionnalité de Consultation avec Accès Progressif [VILL02]**

La particularité de ce modèle réside dans le fait de considérer la classe *Résultat* comme une EM. Conformément aux spécifications du MAP, cette classe peut être stratifiée. Ainsi on peut reporter le processus de stratification sur la classe *Résultat* permettant à la fonctionnalité de proposer un accès progressif au contenu délivré à l'utilisateur.

La définition en intension du résultat est donnée directement par la clause *SELECT* de la requête. En effet, la classe *Projection* décrit la structure attendue du résultat au moyen d'attributs et de rôles issus des classes et associations décrivant le Modèle du Domaine. L'accès progressif, défini pour un *Résultat* considéré en intension est donc basé sur la stratification de la classe *Projection*.

La définition en extension du résultat est donné par les extensions (ensembles d'instances), indiquées par la source de la requête, auxquelles est appliqué le filtrage défini par la sélection de cette requête.

Pour résumer, on modélise une fonctionnalité en tenant compte de la requête qu'elle utilise et surtout, en considérant le résultat de cette requête comme une Entité Masquable. Le but de cette modélisation est d'accroître les possibilités d'expression en vue de l'Accès Progressif au moyen de structures de données qui permettent de retrouver au sein d'un même structure des informations répandues en classes et associations distinctes.

La modélisation proposée autorise ainsi la construction d'EM plus complexes en bénéficiant de mécanismes inhérents aux requêtes tels que la définition de vues, l'utilisation des jointures, les filtrages sur des valeurs, etc. Ainsi modélisée, une fonctionnalité permet de répondre à des besoins de consultation d'informations plus élaborés, i) en autorisant des clauses de projection et sélection complexes d'informations et ii) en donnant les moyens de stratifier de différentes façons le résultat d'une même requête pour garantir un Accès Progressif personnalisé au contenu qu'elle délivre.

#### 4.4. Le modèle utilisateur

Compte tenu de l'hétérogénéité du public d'un SIW dont les individus ne présentent pas les mêmes besoins à l'égard du système, n'ont pas les mêmes droits sur les données de celui-ci en faisant preuves d'expériences ou de connaissances variées, l'identification de différents groupes d'utilisateurs est un moyen de réduire la complexité de la gestion d'une telle variété.

Néanmoins, les utilisateurs dans le cadre d'un groupe, présentent des caractéristiques individuelles variant parfois dans une large mesure. La prise en compte d'un utilisateur en tant qu'individu est fort nécessaire. L'objectif est de proposer en priorité des adaptations basées sur des spécifications personnelles ; en l'absence de celles-ci, les spécifications des groupes sont alors utilisées.

Un utilisateur donné peut également, selon le contexte d'utilisation du SIW, avoir des besoins différents, préférer une option à une autre, etc. A cet égard, on doit prévoir le fait que les utilisateurs disposent de profils différents.

L'organisation du modèle des utilisateurs est illustrée dans la Figure 4.6.

Le concept de *Groupe d'utilisateurs* (ou *groupe*) permet de fédérer les informations relatives à plusieurs utilisateurs qui partagent certaines caractéristiques. Dans notre approche, le critère de regroupement des utilisateurs individuels est fondé sur le concept de *Rôle Fonctionnel*.

Un *groupe* est associé à un *rôle fonctionnel*, ce dernier étant constitué de fonctionnalités. Tous les membres de ce groupe ont ainsi en commun le fait d'avoir accès à l'ensemble des Fonctionnalités qui constituent le *Rôle Fonctionnel* associé à ce groupe. L'identification d'un groupe dans le Modèle des Utilisateurs relève donc de celle des différents *Rôles Fonctionnels*.

Un *utilisateur* peut appartenir à un ou plusieurs groupe(s), et peut, en conséquence, remplir les différents *Rôles Fonctionnels* associés aux groupes dont il est membre. *L'Espace Fonctionnel* dont un *Utilisateur* est propriétaire est constitué de ces *Rôles Fonctionnels*.

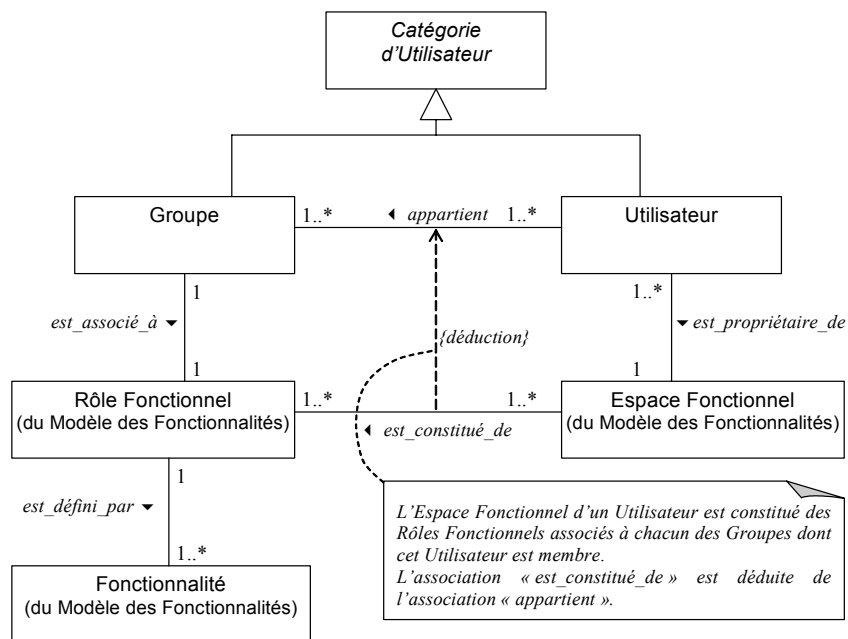


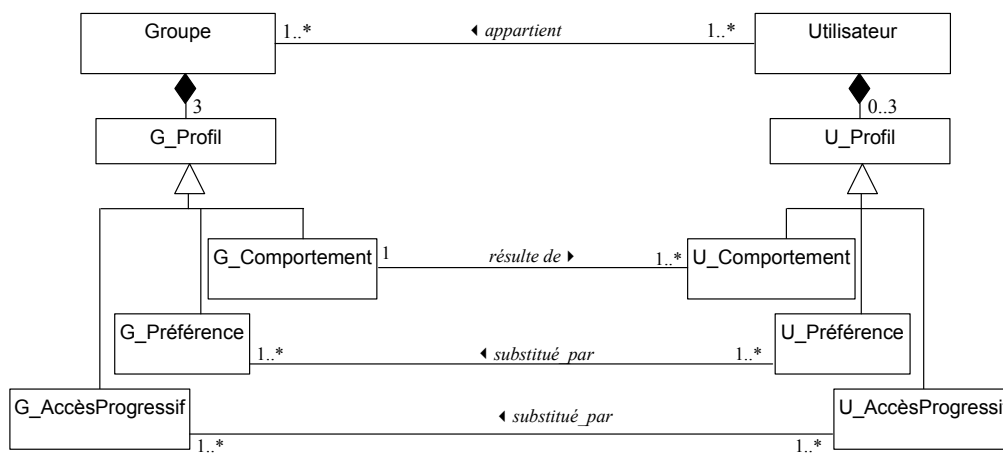
Figure 4.6 Modèle des Utilisateurs : distinction entre Groupes et Utilisateurs basée sur les concepts du Modèle des Fonctionnalités [VILL02]

Les groupes et les utilisateurs détiennent un certain nombre d'informations qui permettent de les identifier et les qualifier. Il s'agit classiquement d'attributs tels que *nom* et *description*, d'attributs pour l'authentification (login, mot de passe), l'identification (nom, prénom de l'utilisateur), etc.

Les autres types d'informations modélisées sont répartis en différentes catégories que nous appelons *profils*. La Figure 4.7 illustre la prise en compte de tels profils dans le Modèle des Utilisateurs. Les profils sont directement en rapport avec les modèles personnalisable : le Modèle des Fonctionnalités et le Modèle de l'Hypermédia.

La classe Groupe est liée par une composition à trois *profils de groupe* modélisés par la classe *G\_Profil*. Celle-ci se spécialise en *G\_AccèsProgressif*, *G\_Préférences* et *G\_Comportement*.

De façon similaire, la classe Utilisateur est liée par une composition à, au plus, trois *profils d'utilisateurs* que modélise la classe *U\_Profil*. Une spécialisation de cette classe, similaire à celle de la classe *G\_Profil* est proposée : on distingue les classes *U\_AccèsProgressif*, *U\_Préférences* et *U\_Comportement*.



**Figure 4.7 – Le Modèles des Utilisateurs : profils décrits pour les groupes et les utilisateurs**

Le profil d'accès progressif garantit l'adaptation des stratifications. L'adaptation des aspects relatifs à la présentation de l'hypermédia est supportée par les profils de préférences. Et enfin, les principes de l'adaptativité<sup>1</sup> sont introduits par les profils de comportement.

Dans cette première version de noyau nous incluons dans le noyau seulement les profils d'accès progressif.

<sup>1</sup>Adaptativité capacité d'adaptation automatique de système à l'utilisateur en analysant son comportement exprimé à travers les interactions utilisateur-système



# RÉALISATION

---

Cette dernière partie du mémoire est consacrée à la présentation du noyau logiciel réalisé au cours de ce stage.

Le noyau est composé d'une hiérarchie d'interfaces Java, organisées en paquetage. Chaque paquetage correspond à un des modèles présentés dans ce mémoire comme faisant partie intégrante des applications KIWIS. Le regroupement en paquetages des interfaces a été réalisé afin d'isoler les modules, augmentant ainsi la flexibilité du noyau.

Le chapitre 5 décrit la hiérarchie d'interfaces et l'organisation en paquetages du noyau. La présentation sera réalisée en consacrant une section à chaque module de base (Module du Domaine, Module des Fonctionnalités, Module d'Accès Progressif, Module des Utilisateurs). En plus de ces modules qui résultent direct des spécifications théoriques présentées dans le chapitre précédent des modules supplémentaires, sont introduits :

- module d'accès – assure la gestion de l'authentification et de l'accès aux applications
- module application KIWIS – régit les interactions entre modules au sein d'une application KIWIS
- module système KIWIS – régit les interactions entre les différentes applications KIWIS partageant un même serveur KIWIS

Le chapitre 6 propose une première implémentation de ce noyau logiciel. Celle-ci s'appuie pour la représentation des données, sur la plateforme AROM que nous avons présentée lors de deuxième chapitre. Les informations caractérisant les divers modèles discutés, ainsi que leurs instances sont stockées dans des bases de connaissances AROM. L'accès à ces informations : consultation, modification, création, se réalise en utilisant l'API AROM dans sa version 2.0.

## 5. LE NOYAU KIWIS

---

Dans ce chapitre nous présentons le noyau KIWIS, qui se révèle être une réalisation logicielle modulaire, flexible et ouverte, capable d'interagir avec divers clients logiciels.

Le noyau constitue une véritable base de départ pour toute application pour laquelle il est souhaitable de mettre en œuvre un accès graduel à l'information. La description de noyau ne portera que sur les aspects spécifiques à l'organisation d'une application en vue de la mise en œuvre d'un accès progressif.

Nous ne souhaitons pas que le noyau prenne en charge des problèmes liés à la présentation des informations extraites des bases de connaissances sous-jacentes au modèle de données. Il vise avant tout à fournir un support logiciel pour les applications mettant en œuvre un accès progressif. Ainsi, on peut envisager de construire des applications Web complexes, qui ne se limitent pas à une suite de pages Web, mais qui mettent en œuvre un ensemble varié de techniques et réalisations logicielles.

Dans ce qui suit, nous présentons le *module des utilisateurs*, car l'utilisateur constitue une des notions centrales de notre approche. La description des autres modules du noyau sera faite par rapport à ce modèle.

Nous présentons un module propre à l'implantation logicielle, le *module d'accès* aux informations contenues dans les différents modules, ainsi qu'aux ressources du système.

Nous poursuivons par la présentation des modules correspondant aux modèles fondamentaux de notre approche introduits dans le chapitre précédent : le modèle d'accès progressif, le modèle du domaine, le modèle des fonctionnalités.

En dernier lieu nous introduisons les interfaces qui regroupent des instances des modules présentés dans des applications KIWIS (*KApplication*) organisées à leur tour en systèmes KIWIS (*KSystem*).

### 5.1. Module d'utilisateurs (kiwis.user)

Le module utilisateur est chargé de la gestion des utilisateurs et des groupes d'utilisateurs dans le système KIWIS.

L'implantation logicielle de ce module suit les spécifications du modèle des utilisateurs décrit de la section 4.4 à quelques différences près. Chaque notion du modèle utilisateur se reflète dans une interface spécifique du module utilisateur. Ainsi, le diagramme de classes de la Figure 5.1 correspond au module d'utilisateurs :

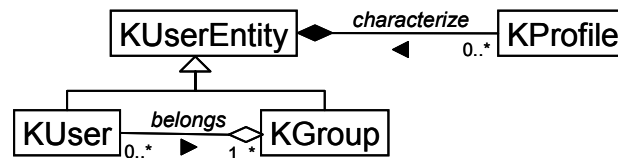


Figure 5.1 Diagramme des classes pour le module d'utilisateurs

Par rapport au modèle d'utilisateurs illustré dans les Figure 4.6 et Figure 4.7 nous avons implémenté l'association de manière implicite, puisque cela convient mieux à la réalisation en Java du modèle.

Une deuxième différence provient du fait que la notion de profil n'est pas déclinée selon les différents types indiqués lors de la spécification : profil d'accès progressif, profil de préférence, profil de comportement. Ceci constitue un choix délibéré que nous assumons pour permettre l'extension des types de profils exploitables par les futurs modules qui seront intégrés au noyau. L'interface *KProfil* considère les profils comme des objets contenant une série de couples <identifiant de la propriété, valeur de la propriété>.

Cette interface permet la récupération *Object getValueOf(Object propertyID)*, la modification *Object setValueOf(Object propertyID, Object propertyValue)* ou l'effacement *Object remove(Object propertyID)* d'une propriété. L'interface assure, également, la mise en oeuvre d'une méthode qui fournit la liste des propriétés dont le profil dispose *Iterator properties()*.

La méthode *String getProfileType ()* permet de distinguer entre les différentes types de profils qui caractérisent une entité utilisateur *KUserEntity*.

Pour accéder aux profils la caractérisant une instance de l'interface *KUserEntity* dispose des deux méthodes :

- la première permet de récupérer un profil selon son type *KProfile* *getProfile(String type)*;
- la deuxième fournit un itérateur sur l'ensemble des profils associés à l'entité.

De manière générale, une entité utilisateur est caractérisée par un identifiant – qui permet de la repérer de manière unique au sein du modèle des utilisateurs. En plus de cet identifiant *id* qui peut être porteur de peu d'informations descriptives, nous ajoutons un attribut de description *description*, ainsi que l'attribut *name* désignant un nom plus significatif que l'identifiant pour l'entité utilisateur en question. Mis à part l'identifiant d'une entité, tous les autres attributs sont à la fois consultable (*get\*()*) et modifiables (*set\*(\* value)*).

Cette interface est déclinée dans les interfaces *KUser* et *KGroup* qui spécialise le concept d'entité utilisateur selon les notions d'utilisateur et groupe d'utilisateurs. Globalement, les extensions prévoient l'implantation de l'association d'appartenance *belongs\_to* entre les groupes et les utilisateurs

Ainsi, l'interface *KUser* dispose des méthodes :

- *Iterator groups()* – fournit un itérateur sur la liste des groupes dont l'utilisateur fait partie

- *KGroup getMainGroup()* – récupère le groupe principal de cet utilisateur
- *boolean belongsTo(KGroup group)* – permet de confirmer l'appartenance à un certain groupe

De son côté, l'interface *KGroup* implante les méthodes :

- *Iterator users()* – fournit un itérateur sur la liste des utilisateurs appartenant à ce groupe
- *boolean contains(KUser user)* – permet de confirmer l'appartenance d'un utilisateur à ce groupe

Un système KIWIS peut déployer plusieurs applications KIWIS. Par rapport à cet aspect, nous sommes obligés d'envisager l'existence de deux types des modules au sein d'un système KIWIS :

- le module des utilisateurs systèmes : qui se concentre sur les droits (modification, consultation, création, etc.) associées aux utilisateurs en tant que clients du systèmes
- le module des utilisateurs au niveau d'une application : qui est plutôt orientée vers le regroupement de centre d'intérêts par rapport à l'utilisation de l'application. Dans l'approche de conception proposée par [VILL02] les groupes d'utilisateur sont créés par rapport aux fonctionnalités qui les intéressent en non pas leur droit sur le système qu'ils utilisent.

Dans la section suivante nous présentons l'implantation du module d'accès qui offre les moyens de consulter ou de modifier un système KIWIS ou ses applications.

## 5.2. Module d'accès (kiwis.access)

Le module d'accès introduit une série d'interfaces qui définissent les modalités d'accès aux ressources des applications KIWIS installées sur un serveur KIWIS.

La première interface que nous présentons, l'interface *EntityLocator* définit la manière générale dont on accède aux objets du système.

### 5.2.1 Récupération des entités du système KIWIS

Chaque entité du système est caractérisée par au moins deux propriétés : son type et son identifiant qui l'identifie de manière unique.

Les caractéristiques et les droits de celui qui fait une demande en tant qu'utilisateur du système constitue le *contexte d'accès*. En plus du type et de l'identifiant, le contexte constitue le troisième facteur du processus de récupération d'une entité, car en fonction de ce contexte le système peut refuser l'accès à l'entité en question.

#### 5.2.1.1 L'interface KContext

L'interface *KContext* correspond à la notion de contexte. La définition de cette interface est fort liée au module utilisateur que nous avons présenté précédemment.

Cette interface contient des méthodes permettant d'accéder :

- à l'objet utilisateur *KUser* associé à ce contexte
- son profil et groupe courant *KGroup* *getGroup()*, *KProfile* *getProfile(String profileType)* (le paramètre *profileType* permet de récupérer un type précis de profil : profil de préférences, profil de droits d'accès, etc.)
- ainsi qu'à la totalité des profils définis pour cet utilisateur *Iterator profiles()*,
- et à tous les groupes dont cet utilisateur fait partie *Iterator groups()* ;

En plus de ces informations liées à la modélisation de la notion d'utilisateur dans le système KIWIS, le contexte peut contenir une série d'informations supplémentaires donnant la possibilité de particulariser un peu plus le contexte. Ces informations sont stockées sous la forme de couples <nom de propriété, valeur> dans un tableau de stockage (*java.util.Map*). La récupération de ce tableau de propriétés s'effectue en appelant la méthode *Map getAdditionalProperties()* de l'interface *KContext*.

En plus des méthodes de consultation, l'interface *KContext* permet aussi de changer la configuration courante du contexte. Ainsi, les méthodes *changeCurrentGroup(KGroup g)*, *changeCurrentProfile(KProfile profile)* font changer respectivement le groupe ou le profil courant associé à l'utilisateur.

#### 5.2.1.2 L'interface *EntityLocator*

L'interface *EntityLocator* définit la manière générale dont on accède aux objets du système.

La récupération d'une entité se réalise suite à l'appel de la méthode *get(Object entityType, Object entityID, KContext context)*. La récupération se fait par rapport au type de l'entité, son identifiant et les caractéristiques de celui qui en fait la demande.

Indiquant de manière explicite le type d'entité que l'on recherche, les objets de type *EntityLocator* ont la possibilité de cibler la recherche.

Un objet *EntityLocator* a la possibilité d'enregistrer d'autres objets de type *EntityLocator* qui peuvent prendre en charge la récupération de l'entité demandée. La méthode *registerEntityLocator(Object[] entityTypes, EntityLocator entityLocator)* informe l'objet localisateur qu'un deuxième objet localisateur peut lui rendre service pour récupérer des entités des types indiqués par le vecteur *entityTypes*. On obtient ainsi une chaîne de responsabilités dont la configuration peut être changée de manière dynamique à n'importe quel moment conférant ainsi à ce système d'accès un maximum de flexibilité.

Dans le cas où plusieurs objet localisateurs sont enregistrés comme délivrant des entités de même type, celui qui s'est enregistré en premier a la priorité. Dans le cas où il ne parvient pas à satisfaire la demande, le prochain localisateur dans la liste est alors employé.

L'annulation d'un enregistrement effectué par un objet en tant que objet localisateur est réalisée par la méthode *unregisterEntityLocator(Object[] entityTypes, EntityLocator entityLocator)*.

### 5.2.2 L'accès au système

Dans la section précédente nous avons montré comment au niveau du système la récupération d'une entité est réalisée. Cette partie doit rester cachée aux clients des systèmes. Puisque l'accès aux entités de systèmes doit être limité, nous considérons nécessaire

l'introduction d'un nouveau type d'objet qui permet la récupération des entités tout en vérifiant les droits d'accès à celles-ci. Dans ce but nous avons introduit l'interface *KAccess*.

### 5.2.2.1 KAccess

Cette interface qui régit directement l'accès au système. La récupération des entités se fait approximativement de la même manière que dans le cas d'un objet *EntityLocator* (*get(Object entityType, Object entityID)*). Cette fois, néanmoins, le contexte est implicite à la requête, puisqu'il est fourni par la méthode *getContext()* de l'objet *KAccess*.

L'interface dispose aussi d'une méthode *logout()* qui réalise la déconnexion de l'utilisateur associé à un objet *KAccess*.

Les objets *KAccess* sont instanciés par l'objet de type *KSystem* désignant le système *KIWIS* suite à une opération d'authentification suivant un nom de *login* et un mot de passe.

### 5.2.3 Perspectives ouvertes par le module d'accès

Ce découplage entre le client et l'entité recherchée permet aux systèmes de mettre en place une politique des droits d'accès très pointue. Dans un premier temps, le droit d'accéder à telle ou telle entité est requis et en second, à un niveau plus fin de granularité on peut envisager d'exiger des droits spécifiques pour les divers types d'opérations : consultation, modification, création, etc.

Ce deuxième aspect trouve la solution dans la création d'un objet de même type que l'entité initiale. Cet objet délègue l'exécution de tous les appels de ses méthodes vers l'entité tout en vérifiant auparavant si le client en a le droit. Du point de vue de l'implémentation de cette proposition, nous envisageons deux solutions :

- Etendre les classes pour lesquelles on veut avoir un contrôle d'accès au niveau des méthodes ;

```
class EntitéAvecControleDAcces extends Entité {
    EntitéAvecControleDAcces(Entité entité, DroitDAcces da) {
        this.entité=entité ;
        this.da=da ;
    }
    methode1() {
        if (aLesDroitsRequis(da))
            this.methode1;
        else
            throw new DroitsNonAquisException()
    }
}
```

- Utiliser la programmation par aspects (AOP) [KICZ97] qui permet de créer des objets disposant de méthodes soumises à un contrôle à la volée. Pour minimiser la description AOP, cette deuxième solution nécessite une rigueur accrue dans l'écriture de code et notamment dans le choix des noms pour les méthodes. Les noms doivent bien refléter les types d'opérations, car la description AOP qui permet de mettre en place le contrôle d'accès se fait par rapport aux noms des méthodes (*get\**, *set\**, *create\**, *remove\**). Le code ci-dessous, écrit en AspectJ [KICZ01], illustre comment implémenter un droit d'accès en consultation. D'abord on identifie les points d'accès aux méthodes qui nous intéressent et

ensuite on décrit la manière dont le contrôle d'accès se réalise avant que l'appel ne soit effectué.

```
aspect VerificationDroits {
    pointcut get():
        (call(* Entité.get*(*));

    before():get() {
        if (aLesDroitsRequis())
            throw new DroitsNonAquisException();
    }
    ...
}
```

Après avoir présenté le principe d'accès aux entités d'un système KIWIS nous présentons tour à tour en commençant par le modèle d'accès progressif, les modèles fondamentaux de notre proposition.

### 5.3. Module d'accès progressif (kiwis.pam)

Le module d'accès progressif est chargé de la mise en œuvre du modèle d'accès progressif. Dans le même esprit que le modèle, le module que nous proposons est défini de manière générique sans qu'il fasse référence à une entité particulière appartenant aux modèles autres que le modèle d'accès progressif. Nous avons défini une famille d'interfaces JAVA pour représenter, au niveau logiciel, les divers concepts introduits par le modèle d'accès progressif. Ces interfaces décrivent le comportement attendu des objets correspondants aux concepts tels que : entité masquable, représentation d'entité masquable, etc.

L'entité masquable constitue la notion fondamentale de ce modèle. Néanmoins pour permettre l'exploitation des entités masquables il est nécessaire de considérer aussi le concept qui correspond au composants d'une entité masquable: l'élément d'une entité masquable, que l'on appelle par la suite *élément masquable*.

#### 5.3.1 L'élément et l'entité masquable

L'élément masquable constitue la notion fondamentale dans l'organisation et la mise en œuvre des autres concepts du modèle. Il contribue à la caractérisation de l'entité masquable et sa présence permet de construire les diverses représentations de cette même entité.

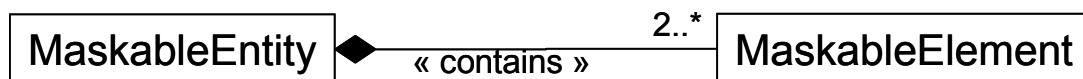
Un élément masquable est caractérisé par sa valeur, par une description facultative qui lui est attribuée, et par un nom.

L'interface qui matérialise l'élément masquable ne dispose que de méthodes de consultation (*getXXX*), la création de ces éléments et la modification ne relevant pas du modèle d'accès progressif. Nous tenons à préciser que la description que nous faisons ici, et la nature éventuelle des éléments masquables, sont totalement ignorées par le modèle d'accès progressif. De manière générale, les objets constituant des éléments masquables sont des instances (i.e. une variable d'une classe, une fonctionnalité, etc.) appartenant aux modèles connexes (i.e. modèle du domaine, modèle des fonctionnalités, etc.) dans le cadre d'une

application. Ce sont les modules correspondant à ces modèles qui régissent l'état et la durée de vie d'un élément masquable et non pas le module d'accès progressif.

Cette remarque, portant sur l'intangibilité de l'état interne des objets éléments masquables, reste valable dans le cas des entités masquables. Une entité masquable ne constitue qu'une vue supplémentaire que l'on définit sur la représentation d'un objet ; elle permet ainsi d'accéder à l'objet qu'elle représente selon un degré plus fin de granularité. Par exemple, définir une entité masquable sur un objet *ensemble* permet d'avoir accès aux éléments composant l'ensemble et de traiter l'ensemble en tant que composition d'objets *éléments* en non seulement en tant qu'objet *ensemble*.

La relation de composition entre les interfaces correspondant aux entités et éléments masquables est illustrée ci-dessous :



**Figure 5.2 La relation entre l'élément et l'entité masquable dans le module d'accès progressif**

Une entité masquable *MaskableEntity* est composée d'au moins deux éléments masquables.

L'interface *MaskableElement* met à disposition des méthodes permettant d'accéder à :

- l'identification de l'élément
- la valeur de l'élément

L'interface *MaskableEntity* dispose des méthodes permettant d'accéder à :

- l'ensemble d'éléments composant l'entité au travers d'un *itérateur* défini sur l'ensemble (*extension()*)
- dans le cas d'une entité à éléments structurés, l'ensemble définissant l'*intension* de la structure (*intension()*)

Les éléments et les entités masquables ainsi représentés permettent la construction des représentations d'entités masquables et l'organisation de celles-ci en stratifications.

### 5.3.2 Stratifications et Représentations d'entités masquables

Les éléments masquables appartenant à une entité masquable servent à créer ce que l'on appelle des représentations d'entités masquables. Selon la nature des éléments masquables, éléments masquables intensionnels (les slots *dénominateur* et *numérateur* de la Figure 4.1 page 29) ou extensionnels (les fractions *a*, *b*, *c*... Figure 4.1 page 29) on crée des représentations en intension ou en extension d'une entité.

Dissociant les éléments intensionnels des éléments extensionnels on obtient deux stratifications parallèles pour la même entité masquable :

- une stratification en intension constituée des représentations en intension de l'entité masquable, représentations construites sur les éléments masquables intensionnels
- une stratification en extension constituée des représentations en extension de l'entité masquable, représentations construites avec les éléments masquables extensionnels

Dans ce qui suit le terme *représentation d'entité masquable* fait référence à une représentation qui combine une représentation en *intension* et une représentation en *extension*. Elle détermine les instances visibles ainsi que l'aperçu que l'on a de leur structure à différents niveaux de détail. Dans certain cas il est possible que seulement un des deux types de représentation soit défini, ainsi on peut avoir des représentations de l'entité masquable de *type=intension*, *type=extension*. On parle de représentation de *type=mix* lorsqu'on fait référence à une *représentation d'entité masquable* comme présentée en début de paragraphe.

L'interface *ROME (Representation Of a Maskable Entity)* traduit la notion de représentation d'une entité masquable en langage informatique. Les objets implémentant cette interface permettent d'obtenir l'*intension* et l'*extension* visibles d'une entité pour une représentation courante. L'*intension* et l'*extension* à délivrer à un niveau de détail donné sont déterminées à l'aide d'itérateurs sur ces ensembles.

Avant de rentrer dans la description détaillée des opérations portant sur les ROMEs, nous soulignons le fait que la gestion des représentations se réalise sur les ROME de type simple *intension* ou *extension* car les ROMEs mixtes ne sont que le fruit d'une combinaison entre les ROMEs de type simple. Elles sont donc vues par le système comme le résultat d'une composition entre des entités basiques (et pas comme des composantes du système à part entière).

L'interface *Stratification* que nous mettons en place, permet de gérer les représentations associées à une entité masquable.

Les opérations de consultation de l'interface *Stratification* permettent d'accéder aux représentations de manière directe : *getROMEatDetailLevel(int detailLevel)*, ou bien progressive : *getFirstROME()*, *getLastROME()*, *getPrevROMEInt(ROME rome)*, *getPrevROMEExt(ROME rome)*, *getNextROMEExt(ROME rome)*, *getNextROMEExt(ROME rome)*.

Les couples *getPrevROME\**, *getNextROME\** permettent respectivement de masquer, de dévoiler la représentation d'une entité masquable soit sur la dimension intensionnelle, soit sur la dimension extensionnelle.

Les méthodes *getFirstROME()* renvoie la représentation qui combine la première représentation en *intension* et la première représentation en *extension*. A l'inverse la méthode *getLastROME()* renvoi la représentation qui combine la dernière représentation en *intension* et la dernière représentation en extension.

La création des représentations d'entités masquables au sein d'une stratification est réalisée par les méthodes *createROMEInt(Object []elements)*, *createROMEExt(Object []elements)* selon le type de représentation qu'on vise. Toute ROME ainsi créée s'ajoute à celles existantes, et correspond au niveau de détail le plus élevé au moment de sa création. Le vecteur d'*éléments* à rajouter à la nouvelle représentation contient dans le cas d'une ROME soit des objets *MaskableElement* soit des chaînes de caractères (objets *String*) désignant les noms des éléments qu'on soit dans le cas de représentations intensionnelles ou extensionnelles.

Si cette façon de créer de nouveaux niveaux de détail convient très bien au cas intensionnel, dans le cas extensionnel la plupart de temps on est confronté au fait que les éléments extensionnels ne sont pas connus au moment de la conception. C'est seulement au moment de l'exploration de l'extension que on les connaît (i.e. les instances d'une classe ne sont pas connues à la définition de la classe, et leur nombre évolue au cours du temps).

Nous considérons qu'il est nécessaire de fournir un mécanisme complémentaire plus générique pour la création des représentations en extension. A notre avis, formuler des expressions algébriques portant sur la structure des éléments (leur intension) satisfait bien les besoins en termes de mécanisme générique pour effectuer une sélection d'éléments appartenant à l'extension d'une entité masquable.

Ainsi, nous rajoutons une troisième méthode pour la création des ROMEs *createROMEExt(String algExpr)*.

L'interface propose aussi des méthodes qui permettent de réorganiser une stratification. La méthode *migrateElementFromROME2ROME(Object []elements, ROME rome1, ROME rome2)* enlève l'élément spécifié comme paramètre de la *rome1* et l'ajoute à la *rome2*. La méthode *migrateElementFromROME2ROME(String algExpr, ROME rome1, ROME rome2)* a été introduit pour adapter le mécanisme de migration des éléments extensionnels d'une ROME à une autre. La motivation de l'introduction de cette deuxième méthode est justifiée par le fait que, dans le cas des éléments extensionnels, on ne peut pas indiquer les éléments un à un, et il est souhaitable d'avoir un mécanisme de désignation générique, qui dans notre cas correspond à une expression algébrique.

Il faut remarquer que les ROMEs participant à cette opération doivent être de même *type* simple, c'est à dire *type=intension* ou *type=extension*.

Cette méthode met à la disposition du système une manière simple d'envisager des stratifications qui évoluent selon les interactions entre le système et le contexte d'utilisation (y compris le comportement de l'utilisateur).

La suppression d'une représentation au sein d'une stratification se réalise en appelant la méthode *removeROME(ROME rome1)*, où *rome1* appartient à la stratification et elle est de type simple.

Nous avons vu que la création de représentations d'entités masquables est assurée par la stratification dont elles seront membres. Les stratifications sont à leur tour gérées par l'interface PAM qui gère le modèle d'accès progressif dans son ensemble.

### 5.3.3 Le PAM

L'interface PAM joue à la fois le rôle de fabrique d'objet stratification, offrant en même temps des fonctionnalités d'accès à ses composantes étendant l'interface EntityLocator présentée dans la section 5.2.1.

Cette interface permet la création des stratifications définies sur une entité masquable et l'association de chaque stratification à un certain *contexte*.

Le *contexte* permet d'associer une certaine *stratification* d'une entité masquable avec un contexte d'utilisation bien précis. Par exemple, le *contexte* qu'on associe à une *stratification* contient le nom de l'utilisateur pour lequel on définit la stratification en question, ou un nom d'un groupe si on définit la stratification pour un groupe d'utilisateur. L'abstraction d'un contexte à un objet Java générique (i.e. *java.lang.Object*) permet d'imaginer une multitude de manières pour décrire le contexte et les propriétés de celui-ci. La création des stratifications est assurée par la méthode *createStratification(MaskableEntity me, Object context)*. L'exécution de cette méthode a comme résultat à la fois l'instanciation d'un nouvel objet *Stratification* et l'association de cette stratification avec le *context* indiqué.

La récupération d'une stratification définie pour une entité masquable peut par la suite être effectuée, en indiquant le contexte pour lequel elle a été définie, et éventuellement nécessite la désambiguïsation du contexte. La désambiguïsation du contexte est réalisé par

l'introduction d'un objet (i.e *java.lang.Comparator*) permettant de comparer les contextes associés aux différentes stratifications disponibles et le contexte associé à la requête. La signature de cette méthode est la suivante : *getStratificationFor(MaskableEntity me, Object context, Comparator contextComparator)*

### 5.3.4 Création des représentations effectives d'entité masquables

Les ROMEs constituent simplement une description de l'aperçu que l'on peut avoir dans un certain contexte de l'entité masquable sous-jacente. Néanmoins, le client s'adresse à ce module pour lui fournir un accès graduel à l'information et en conséquence il doit disposer d'objets qu'il peut manipuler de la même manière que l'objet désignant l'entité masquable. Autrement dit, il faut que les objets qu'on lui fournit de manière graduelle se conforment à la même interface que l'objet désignant l'entité masquable, tout en offrant des mécanismes pour passer au niveau de détail supérieur ou inférieur.

Pour faciliter le passage entre les objets conformes aux différentes représentations d'une stratification nous introduisons une nouvelle interface *EffectiveROME* (*Effective Representation of a Maskable Entity*) qui permet d'accéder aux représentations effectives de l'entité masquable de niveau de détail supérieur (*getNext{Int/Ext}EROME()*) et inférieur (*getPrevious{Int/Ext}EROME()*) pour une entité masquable.

La création de ces objets ne peut pas être prise en charge en mode direct par le modèle d'accès progressif, car cela impliquera de disposer de constructeurs pour l'ensemble des types d'objets éléments masquables pour lesquels on souhaite offrir un accès progressif.

Pour résoudre ce problème tout en préservant le caractère générique de notre approche, nous donnons la possibilité d'enregistrer auprès de l'interface PAM les objets capables de construire des nouveaux objets à partir d'une entité masquable. On assure également que ces objets soient conformes au niveau de détail correspondant à une ROME particulière. Cet enregistrement se fait par rapport à un type particulier d'entité masquable : *registerEROMECreator(Class meType, EROMECreator)*.

L'interface *EROMECreator* constitue une fabrique abstraite disposant d'une seule méthode *create(MaskableEntity me, ROME rome)* qui crée un nouvel objet de même type que l'objet *me* et qui respecte la description fournie par l'objet *rome*. De manière générale les modules qui souhaitent disposer d'un accès progressif à leurs entités doivent fournir des implémentations de cette fabrique abstraite *EROMECreator* et les enregistrer auprès du PAM. Ainsi, le PAM peut construire les représentations effectives associées d'une entité masquable à un niveau détail précis.

## 5.4. Module des données (kiwis.data)

Le module des données est chargé de la mise en œuvre du modèle du domaine. Nous avons souligné lors du chapitre 4 que nous ne sommes pas liés à une modélisation très spécifique de ce modèle.

Néanmoins, nous suggérons le recours à une modélisation de type Entité-Relation, déclinée sous la forme d'une modélisation classe – association. Ceci permet d'obtenir un

degré de généralité suffisant, aussi bien en termes de possibilités d'expression de modèle, qu'en termes de mise en place de réalisations logicielles.

Dans le cadre de ce module nous avons identifié deux activités principales:

- La gestion des structures – le terme structure fait référence aux notions de classe et association, ainsi qu'à leurs composants : variables et rôles
- La gestion des instances de ces structures : tuples et objets

Nous considérons le noyau de la plate-forme AROM que nous avons présenté lors du chapitre 2, comme une solution logicielle qui répond à toutes nos attentes en termes de gestion des structures et de gestion des instances.

Pour inclure le noyau AROM en tant que module dans la plate-forme KIWIS nous introduisons l'interface *DataModel* qui régit les interactions entre le client et le noyau AROM sous-jacent à notre réalisation.

L'interface *DataModel* constitue la porte d'entrée dans le module du domaine, de manière similaire à l'interface PAM pour le module d'accès progressif que nous avons présenté dans la section précédente. Ainsi, cette interface joue aussi le rôle d'un localisateur des entités qui sont véhiculées par le modèle sous-jacent. Dans sa mise en œuvre, elle étend donc, les fonctionnalités de l'interface *EntityLocator*.

En même temps, *DataModel* est l'interface qui régit les interactions au niveau du module. Ainsi, il est souhaitable qu'elle dispose des mêmes caractéristiques qu'une base de connaissances (*au sens AROM* du terme), étendant aussi l'interface *KnowledgeBase* fournie par AROM. Ceci donne la possibilité aux clients de faire référence à un objet *DataModel* caractérisant le modèle du domaine d'une application KIWIS, comme s'il s'agissait d'une base de connaissances AROM.

En effet, un objet *DataModel* représente un objet KnowledgeBase AROM qui en plus est capable de jouer aussi le rôle d'un localisateur comme nous l'avons défini dans la section 5.2.

Dans la suite de ce chapitre, nous présentons le dernier module que nous avons décidé d'inclure dans cette première version du noyau de KIWIS et qui correspond au modèle des fonctionnalités.

## 5.5. Le module des fonctionnalités (kiwis.functionalties)

Cette première version de ce module se concentre sur les fonctionnalités de consultation qui portent sur le modèle des données.

En plus d'assurer la description du module des fonctionnalités cette section constitue un premier exemple dans la construction d'un module supportant le modèle d'accès progressif. Cette description suit deux axes :

- Offrir un accès progressif concernant la structure de modèle (Espace Fonctionnel, Rôles Fonctionnels, Fonctionnalités). A ce niveau on réalise seulement un accès progressif en *extension* ; c'est-à-dire, on masque/dévoile certains rôles de l'espace fonctionnel et certaines fonctionnalités des rôles fonctionnels
- Offrir un accès progressif sur les résultats obtenus suite à l'exécution d'une fonctionnalité – ceci suppose un accès progressif croisé : en *intension* (l'aperçu de structure des instances résultats) et en *extension* (les instances résultats)

Nous commençons par présenter la réalisation logicielle d'une fonctionnalité de consultation. Ensuite, nous décrivons celle d'un rôle fonctionnel avant de présenter la définition d'un espace fonctionnel. En dernier, nous présentons l'interface principale du module de fonctionnalités qui en régit le bon fonctionnement *FunctionalitiesModel*. Cette interface permet de réaliser des interactions avec des entités extérieures au module (i.e. clients du noyau, d'autres modules du noyau).

### 5.5.1 Fonctionnalités de consultation (*ReportFunctionality*)

Une fonctionnalité de consultation est décomposée en plusieurs parties:

- une partie désignant les sources de données (classes ou associations de modèle du domaine, où même d'autres fonctionnalités de consultation)
- une partie indiquant l'ensemble *intensionnel* de la fonctionnalité, c'est-à-dire la projection définie sur les sources de données
- et, finalement une partie réalisant une sélection sur les instances candidates.

En plus de ces attributs une fonctionnalité de consultation doit jouer à la fois:

- le rôle d'élément masquable en tant qu'élément d'un rôle fonctionnel, et aussi
- le rôle d'entité masquable en tant qu'ensemble de réalisations de la fonctionnalité

Cette définition, en termes de modélisation logicielle se traduit dans la relation de composition suivante :

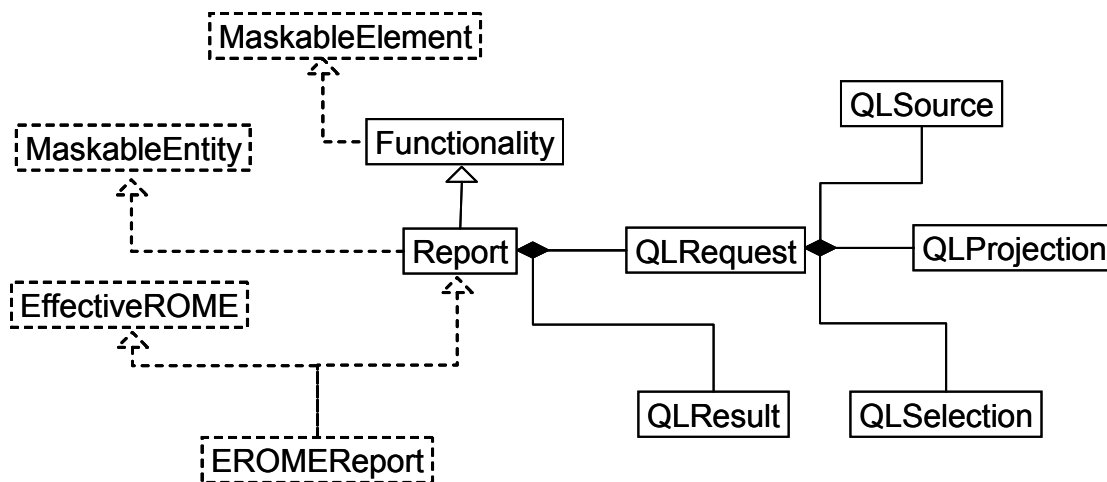


Figure 5.3 Diagramme de classes concernant la définition d'une fonctionnalité de consultation (*Report*) mettant en place les requis pour l'accès progressif

Dans un premier temps, nous décrivons la partie droite de la Figure 5.3, qui caractérise la notion de fonctionnalité de consultation représentée par l'interface *Report*. En second lieu, la partie gauche (dessinée en pointillés) propre à la mise en place de l'accès progressif sur une fonctionnalité est présentée.

Au centre de la figure, l'interface *Report* correspond à une fonctionnalité de consultation. Elle étend l'interface *Functionality* qui correspond à une fonctionnalité de type générique.

Une fonctionnalité de type *Report* est associée à une requête *QLRequest* portant sur le domaine des données propres à l'application (le modèle de données).

La formulation d'une requête suit la notation SQL : `SELECT {attribut1, attribut2, ..., attributn} FROM {source1, source2, ..., sourcem} WHERE conditions`, ce qui se traduit dans le diagramme de classes :

- la projection est modélisée par la classe `QLProjection` qui contient les noms des attributs caractérisant l'intension du résultat ;
- les sources de données sont modélisées par l'interface `QLSource`. Une source de données peut être associée à des structures : classes ou associations du modèle de domaine, ainsi qu'au résultat d'autres fonctionnalités. Les structures sont identifiées par leurs noms et les fonctionnalités par une notation fonctionnelle incluant leur identifiant *functionality(idFonctionality)*. A ce niveau, nous considérons que les éventuelles stratifications définies sur une structure ou sur une fonctionnalité deviennent transparentes, c'est-à-dire, elles ne s'appliquent pas ;
- les conditions à respecter pour l'ensemble des données résultantes se retrouvent dans `QLSelection` ;

Les résultats offerts par la fonctionnalité de consultation suite à l'exécution de la requête afférente sont rendus accessibles via l'interface `QLResult` qui permet d'accéder tour à tour aux éléments du résultat.

Dans ce qui suit nous présentons les apports amenés à ces structures pour supporter la mise en place d'un accès graduel à l'information renvoyée par une fonctionnalité de consultation. Ceux-ci figurent en pointillés dans la Figure 5.3.

L'interface *Functionality* étend l'interface *MaskableElement*, car les fonctionnalités sont considérées comme des éléments masquable au sein des rôles fonctionnels.

A son tour, l'interface *Report* étend l'interface *MaskableEntity*, puisque l'on veut la considérer comme une entité masquable. Les éléments de ces entités masquables spécifiques aux fonctionnalités de consultation sont constitués par les résultats de la requête associée à la fonctionnalité :

- l'*extension* d'une telle entité masquable est constitué par les éléments de *QLResult* ;
- l'*intension* correspondant aux éléments caractérisant la projection *QLProjection*.

Les notions que nous venons de présenter permettent au module d'accès progressif *kiwis.pam* de construire des stratifications pour les fonctionnalités de consultation. En revanche, ces stratifications ne sont que de nature descriptive, elles ne permettent pas aux clients d'obtenir les aperçus escomptes sur les fonctionnalités.

Pour parvenir à obtenir les objets attendus, il est nécessaire d'étendre les interfaces *Report* et *EffectiveROME* (de *kiwis.pam*) pour obtenir l'interface *EROMEReport* qui permettra d'avoir des réalisations des fonctionnalités conformes à telle ou telle représentation de la fonctionnalité.

Les implantations de ce noyau devront fournir des fabriques *EROMECreator* à enregistrer auprès du module d'accès progressif. Ceci permettra au module de gestion de stratification de fournir aux clients des objets conformes aux *ROMEs*, munis de modalités d'accès graduel (plus d'information/moins d'information).

### 5.5.2 Rôles Fonctionnels et Espace Fonctionnel (*FunctionalRoles*, *FunctionalSpace*)

Les rôles fonctionnels organisent les fonctionnalités, afin que l'utilisateur puisse se repérer plus facilement dans l'espace fonctionnel dont il dispose.

La diagramme de composition qui modélise les concepts d'Espace Fonctionnel et Rôle Fonctionnel est présenté dans la Figure 5.4.

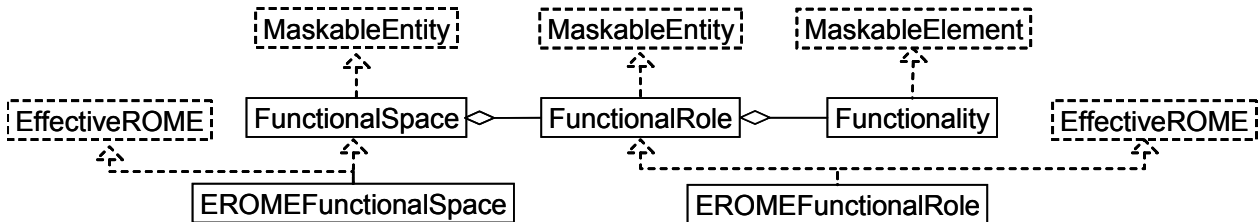


Figure 5.4 Diagramme de composition de l'espace fonctionnel et des rôles fonctionnels supportant un mode d'accès progressif

La partie inférieure de la figure présente la composition des interfaces spécifiques à la modélisation des concepts du module des fonctionnalités. Un rôle fonctionnel *FunctionalRole* regroupe une ou plusieurs fonctionnalités *Functionality*. Chaque fonctionnalité appartient à un rôle fonctionnel. L'espace fonctionnel associé à un utilisateur est constitué des rôles fonctionnels désignés par rapport aux groupes dont l'utilisateur fait partie.

La partie de la Figure 5.4 qui est dessinée en pointillée correspond aux ajouts permettant la mise en place de l'accès progressif pour ces structures.

Les interfaces *FunctionalSpace* et *FunctionalRole* étendent l'interface *MaskableEntity*, puisque toutes les deux définissent des ensembles ; et l'interface correspondant à une fonctionnalité hérite seulement du comportement d'un élément masquable.

Les interfaces *EROMEFunctionalSpace* et *EROMEFunctionalRole* sont introduites pour permettre la manipulation des représentations effectives des objets correspondant respectivement à l'espace fonctionnel et aux rôles fonctionnels. Ainsi, ces objets doivent respecter les niveaux de détail établis par les stratifications auxquelles ils correspondent.

### 5.5.3 L'interface *FunctionalModel*

L'interface *FunctionalModel* prend en charge la gestion du module des fonctionnalités. De manière similaire à d'autres interfaces régissant les modules : *kiwis.pam.PAM*, *kiwis.data.DataModel*, etc., elle joue à la fois le rôle de fabrique des entités spécifiques aux modules et le rôle de localisateur des entités au sein du module.

Ainsi, des méthodes pour la création des diverses entités manipulées sont disponible :

- *Functionality createFunctionality(String functionalityType, String functionalityID, Object parameters[])*

Cette méthode crée une fonctionnalité de type *functionalityType*. Cette chaîne de caractères indique le type de fonctionnalité qu'on veut créer. Par exemple, elle vaut "Report" lorsqu'on veut créer une fonctionnalité de consultation *Report*. Nous avons préféré cette manière plus générale (sans introduire des méthodes spécifiques : *createReport*, etc.), pour permettre l'extensibilité du noyau sans que les clients déjà

construits soit affectés. Ainsi, l'introduction d'un nouveau type de fonctionnalité ne se traduit pas par l'ajout d'une nouvelle méthode dans l'interface et donc les modifications induites sont transparentes pour les clients.

- *FunctionalRole createFunctionalRole(String roleName, Functionality []functs, Object context)*

Cette méthode crée un rôle fonctionnel regroupant un certain nombre de fonctionnalités. Plusieurs rôles fonctionnels peuvent avoir le même nom, car en plus de leur nom ils sont associés à un certain contexte. Ce contexte, de manière générale, est constitué par le(s) nom(s) de groupe(s) pour lesquels ce rôle est conçu. Néanmoins, cette caractérisation du contexte peut contenir d'autres paramètres, comme par exemple le nom d'un utilisateur auquel cas on parle de rôle personnalisé au niveau d'utilisateur.

En plus de ces méthodes pour la création des entités, l'interface met en place des méthodes pour leur gestion après leur instanciation :

- *void removeFunctionality(Functionality functID)* – efface la fonctionnalité en question ;
- *FunctionalRole duplicate(FunctionalRole role, Object context)* – duplique un rôle fonctionnel en lui associant un autre contexte (i.e. la personnalisation, c'est-à-dire le passage d'un rôle associé à un groupe, à un rôle associé à un utilisateur) ;
- *FunctionalRole[] getFunctionalRolesNamed(String name)* – fournit un vecteur contenant les rôles *nameRoles* indépendamment des contextes qui leurs sont associés
- *FunctionalRole[] getFunctionalRolesForContext(Object context)* – fournit un vecteur contenant les rôles associés à un certain contexte
- *void addFunctionalitiesToRole(FunctionalRole role, Functionality[] functs)* – ajoute de nouvelles fonctionnalités pour le rôle fonctionnel en question
- *void removeFunctionalitiesFromRole(FunctionalRole role, Functionality[] functs)* – retire les fonctionnalités indiquées de la liste des fonctionnalités atteignables par le rôle fonctionnel
- *void removeRoles(FunctionalRoles[] roles)* – retire du module des fonctionnalités les rôles indiqués. Néanmoins, les fonctionnalités qui les composaient continuent d'exister.

Une dernière méthode qui permet d'obtenir l'espace fonctionnel correspondant à un certain contexte d'utilisation est incluse dans cette interface *FunctionalSpace computeFunctionSpaceFor(Object context)*.

Suite à la présentation des modèles fondamentaux et, dans cette dernière section, de l'illustration de la préparation d'un module pour supporter l'accès progressif, nous introduisons les interfaces régissant les applications KIWIS et le système KIWIS.

## 5.6. Le système et les application KIWIS

Après avoir présenté les modules fondamentaux du noyau KIWIS dans les sections précédentes, dans cette section nous présentons ce que l'on appelle un système KIWIS, ainsi qu'une application KIWIS.

Cette section résume les différentes idées concernant le travail coordonné de ces modules, coordination qui aboutit à la définition d'une application de système KIWIS.

La Figure 5.5 illustre l'organisation en module d'un système KIWIS.

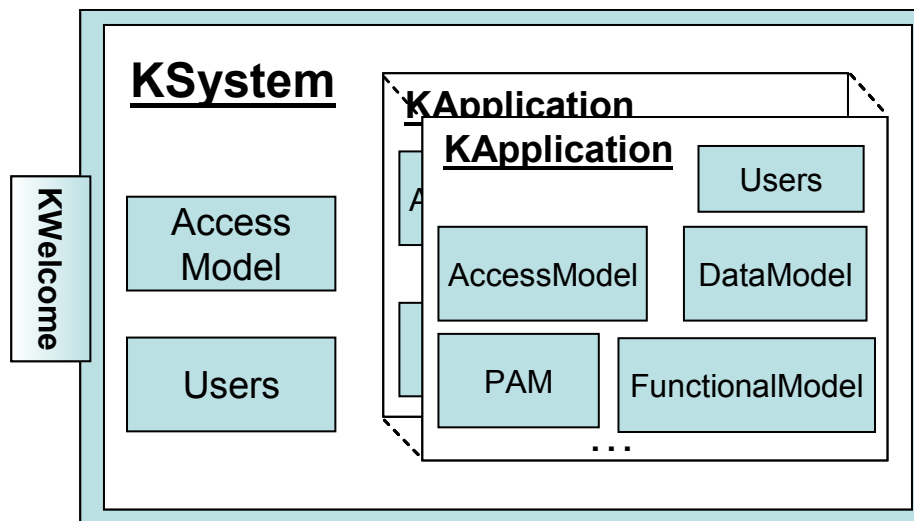


Figure 5.5 Organisation logicielle d'un système KIWIS

Dans cette première version toutes les applications KIWIS disposent :

- d'un module d'accès *AccessModel*,
- le module utilisateur *Users*
- d'un module d'accès progressif *PAM*,
- d'un module des fonctionnalités *FunctionalModel* et
- d'un module de données du domaine de l'application *DataModel*.

A ces modules standard peuvent s'ajouter d'autres modules au moment de la création d'une application. La seule contrainte imposée sur les modules supplémentaires est de mettre en place des mécanismes de localisations, c'est-à-dire étendre l'interface *EntityLocator*. Les interactions avec les autres modules se réalisent en faisant appel à l'instance *Application* qu'ils les instancient.

Le principal rôle d'un objet application *KApplication* est de permettre les interactions de ses modules entre eux, ainsi que les interactions client de l'application avec ses modules. Pour réaliser ces interactions une application doit en premier lieu authentifier les demandes (en se servant du module d'accès) et en second, localiser les instances des entités nécessaires aux échanges.

Ainsi l'interface *KApplication* doit hériter de l'interface *EntityLocator* du module d'accès qui permet la récupération des entités des modules enregistrés de l'application. C'est la responsabilité de l'application d'enregistrer d'autres objets *EntityLocator* ou de déléguer la tâche de manière explicite au module concerné.

Chaque application dispose en plus d'un nom qui l'identifie au sein du système, d'un objet de type *ApplicationResume* représentant une présentation plus détaillée de ces caractéristiques.

L'interface *ApplicationResume* est constituée d'une série des méthodes fournissant des informations sur une application KIWIS :

- *URL getLogo()* – fournit l'adresse du logo de l'application ;
- *String getName()* – fournit le nom de l'application ;
- *String getDescription()* – fournit une description courte de l'application
- *URL getURLDescription()* – fournit une adresse contenant une description plus détaillée de l'application

Les applications à leur tour sont enregistrées par le système (*KSystem*) auxquelles elles appartiennent. L'enregistrement leur permet d'accéder aux ressources du système et notamment au module utilisateur du système pour avoir la possibilité d'authentifier ses clients.

La création d'une nouvelle application *KApplication* est réalisée par le système KIWIS, suite à un appel *KApplication createApplication(String name)* réalisé par un client autorisé par le système. Les clients qui sont authentifiés directement par le système sont appelés des *clients systèmes* et ils constituent les administrateurs du système.

La demande initiale d'authentification d'un client est réalisée à travers l'interface *KWelcome*. Cette interface dispose de deux méthodes qui réalisent respectivement des authentifications habituelles *KAccess login(Map userIdTable)* ou des authentifications de type système *KAccess systemLogin(Map userIdTable)*. Le tableau de stockage *userIdTable* contient une liste de couples <nom propriété, valeur> qui permettent aux module d'accès de réaliser l'authentification. Habituellement, parmi les propriétés on retrouve l'identifiant de l'utilisateur au sein du système, son mot de passe, etc.

L'interface prévoit aussi une méthode qui fournit un tableau de stockage standard contenant les noms des propriétés requises pour réaliser l'authentification (*Map getStandardUserIdTable()* ou *Map getSystemUserIdTable()*).

A travers cette interface, le système KIWIS présente aux clients potentiels la liste des applications KIWIS publiques du système. *ApplicationResume[] applications()*.

Cette interface a été introduite pour éviter que tout client détienne une référence vers le cœur de système.

L'introduction de cette interface est nécessaire pour assurer la sécurité des systèmes KIWIS, car nous avons fait l'hypothèse qu'une référence vers une entité de système est obtenue seulement par les clients habilités. Cette hypothèse qui paraît un peu restrictive est cependant très efficace, car celle-ci permet aux implémentations de ce noyau de ne pas être concernées par des aspects liés aux droits des utilisateurs aux niveaux d'appels de méthodes. Ainsi, nous considérons que les efforts implémentations des interfaces supplémentaires, comme c'est le cas de l'interface *KWelcome*, sont très bien compensés au niveau de l'effort d'implémentation d'une politique d'accès pour chacune des entités du système.

## 5.7. Implémentation AROM du noyau Kiwis

La première implémentation du noyau KIWIS réalisée lors de ce stage est fortement liée à la plate-forme AROM, surtout en ce qui concerne la représentation des informations véhiculées.

En effet, nous avons reconsidéré notre choix concernant la représentation interne des données en passant d'une représentation en format XML à une représentation AROM. Nous avons choisi de représenter les données à l'aide de la plateforme AROM car cette plateforme offre un noyau Java pour la gestion des structures des données, ainsi que pour celle des instances. Ainsi, toutes les tâches découlant de la gestion des données n'alourdissent plus l'implémentation du noyau KIWIS, puisqu'elles sont prises en charge par la plateforme AROM.

Compte tenu de ces remarques nous avons adopté la plateforme AROM aussi pour assurer la persistance des données internes aux différents modules du noyau.

Cette dernière section du chapitre est consacrée à la description des bases de connaissances AROM utilisées par l'implémentation que nous avons réalisée. Nous n'insistons pas sur les réalisations logicielles effectives des modules puisque leur création n'a pas posé de difficultés particulières.

### 5.7.1 Organisation des bases de connaissances

Un système KIWIS doit sauvegarder, entre les moments où il est instancié, les applications qu'il déploie, ainsi que les diverses caractéristiques du système. Cette sauvegarde est réalisée dans des bases de connaissances AROM.

L'organisation des bases de connaissances caractérisant les modules d'une application suit le schéma de la Figure 5.6

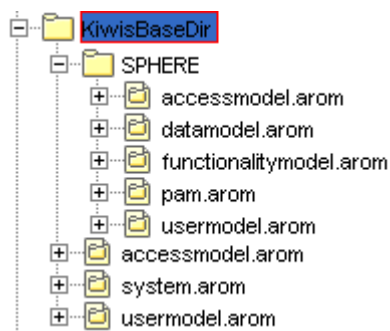


Figure 5.6 Organisation des bases des connaissances d'un système KIWIS

Les bases de connaissances d'un système Kiwis se trouvent dans ce que l'on appelle le répertoire principal de l'implantation logicielle (KiwisBaseDir). Ce répertoire contient les bases de connaissances concernant l'organisation du système :

- *system.arom* – caractérise l'état de système
- *accessmodel.arom* – caractérise les règles d'accès aux ressources système
- *usermodel.arom* – caractérise les utilisateurs disposant d'un accès au système

A part ces informations concernant l'état du système, chaque application (dans le cas présent il n'y a qu'une seule *SPHERE*) conserve l'état interne de ces modules dans des bases de connaissances comme suit :

- *accessmodel.arom* – caractérise les règles d'accès aux ressources de l'application
- *datamodel.arom* – décrit le modèle du domaine
- *functionalitymodel.arom* – caractérise les modules des fonctionnalités
- *pam.arom* – caractérise le module d'accès progressif

- *usermodel.arom* – caractérise les utilisateurs au niveau de l’application

La suite de cette section est consacrée à la présentation d’un des modules système, *system.arom*, et à un des modules application, *functionalitymodel.arom*. Nous choisissons d’illustrer ces deux bases de connaissances car elles constituent des exemples représentatifs au sein du système.

### 5.7.2 Base de connaissances système

La base de connaissances *system.arom* décrit de manière générale les propriétés du système et les applications contenues. Elle est modélisée conformément à la Figure 5.7

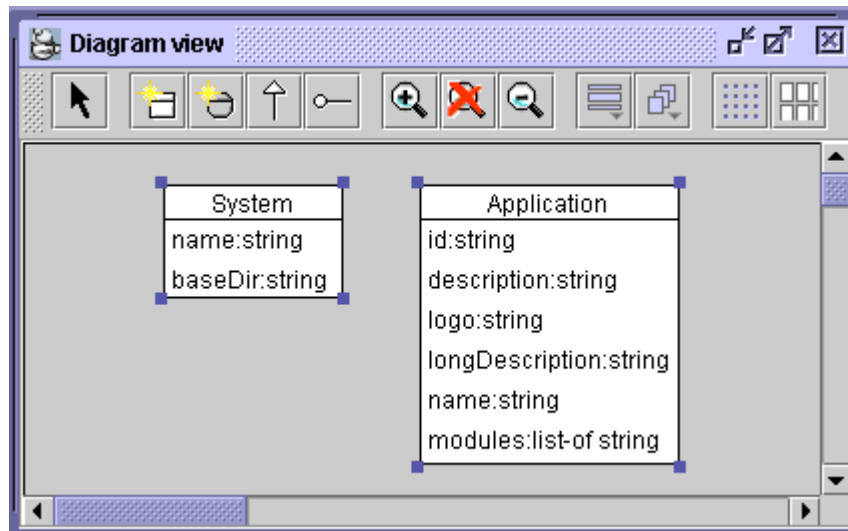


Figure 5.7 Base de connaissances utilisée pour la sauvegarde du système

La classe *System* définit la structure gardant les propriétés du système KIWIS. Cette classe constitue un *singleton* car il n’y a qu’une seule instance de cette structure. Ceci est tout à fait normal car cette base de connaissances ne décrit qu’un seul système KIWIS.

La classe *Application* décrit de manière succincte une application KIWIS. A ce niveau, une application est caractérisée par son identifiant, sa description, des adresses faisant référence au logo et à une description plus détaillée de l’application. A part ces propriétés, une liste de chaînes de caractères identifiant chacune de manière unique les modules est incluse dans les instances de la classe *Application*. Ce choix d’indiquer les modules présents dans l’application peut sembler inutile, car nous avons précisé que, selon cette implémentation, les applications contiennent toutes les mêmes modèles. Néanmoins, nous avons opté pour ce choix afin de limiter le nombre d’adaptations à faire pour le passage à des applications à modules variables.

Nous considérons inutile la modélisation d’une association entre la classe Système et la classe Application, car de manière implicite, toutes les instances de la classe Application seront associées à la seule instance de la classe Système.

Les instances de cette classe correspondent, en effet, aux applications installées au sein du système *KIWIS*. L’identifiant de l’application ainsi que la liste des modules présents permet d’accéder à la représentation AROM des modules en question.

Les identifiants contenus dans de la liste correspondant aux modules mis en œuvre permettant d'accéder aux bases de connaissances contenant les données persistantes de ces modules. Par exemple, le module PAM de l'application SPHERE retrouve sa sauvegarde à l'emplacement suivant `.\SPHERE\PAM.arom`

### 5.7.3 Base de connaissances pour le module des fonctionnalités

La base de connaissances `functionalitymodel.arom` contient des structures permettant de représenter les informations et les entités persistantes du module des fonctionnalités d'une application KIWIS.

Le diagramme des classes de la Figure 5.8 présente les structures que nous utilisons pour réaliser la sauvegarde du module.

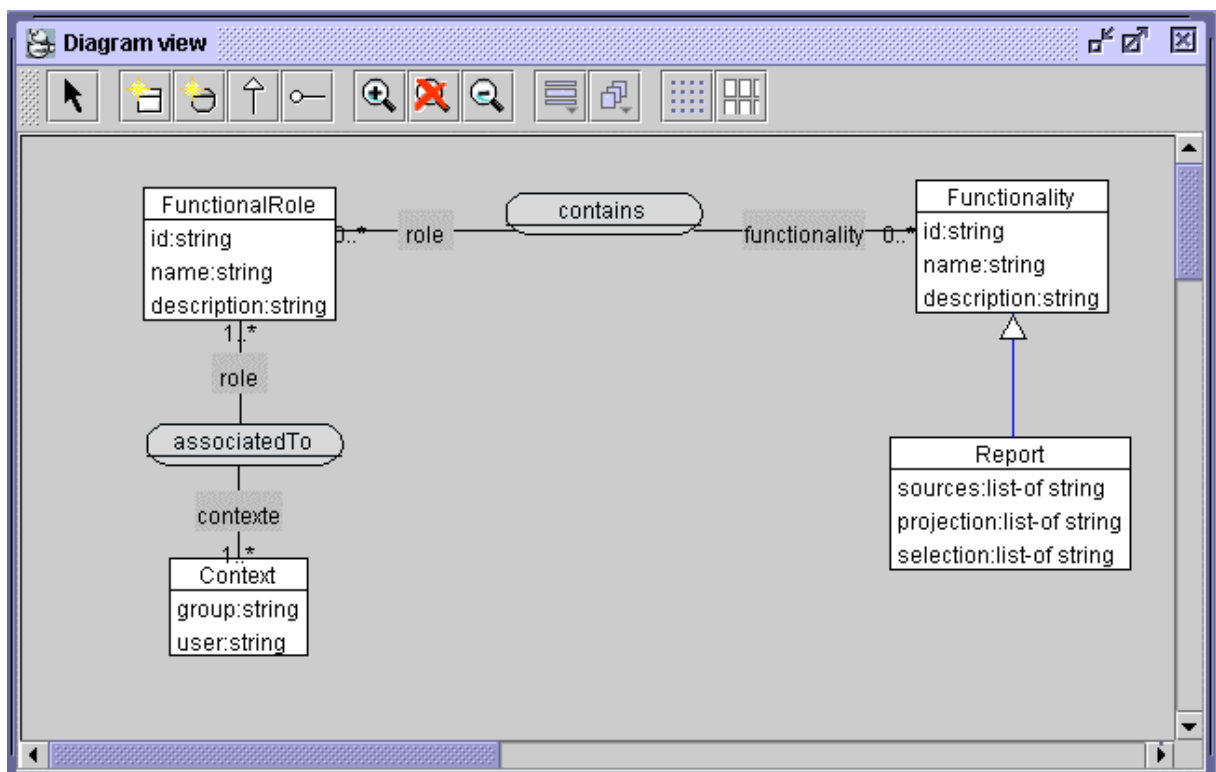


Figure 5.8 Base de connaissances pour le module des fonctionnalités KIWIS

La classe *Functionality* retient pour une fonctionnalité son identifiant, son nom, ainsi qu'une courte description textuelle. Cette classe est spécialisée par la classe *Report* qui caractérise les réalisations des fonctionnalités de consultation.

La classe *Report* sauvegarde la requête associée comme suit :

- la variable *sources* – contient les identifiants des sources d'informations de la requête (la clause *FROM* d'une requête *SQL*). Ces identifiants correspondent aux noms des structures du module des données ou à d'autres fonctionnalités (*funct(idFunct)*)
- la variable *projection* – contient des chaînes des caractères correspondant aux noms d'attributs caractérisant les différentes *sources* d'informations (la clause *SELECT* d'une requête *SQL*)

- la variable *selection* – contient une liste des expressions algébriques portant sur les attributs des sources de données (la clause *WHERE* d'une requête *SQL*). Cette liste définit les caractéristiques attendues des instances résultats.

Le regroupement des fonctionnalités dans les différents rôles fonctionnels du module est modélisé par l'association *contains*.

Les tuples de l'association *associatedTo* sauvegardent les correspondances entre l'ensemble des rôles fonctionnels et l'ensemble des contextes définis dans le cadre du module des fonctionnalités.

En plus des classes et des associations qui reflètent de manière directe la structure du modèle des fonctionnalités, ce diagramme indique aussi qu'elles sont les informations que nous considérons comme caractérisant un contexte associé à un rôle fonctionnel. Ainsi, dans cette implémentation nous considérons que le contexte est décrit par un nom de groupe, et éventuellement, un nom d'utilisateur utilisé pour personnaliser certains rôles fonctionnels au niveau d'utilisateur.

# CONCLUSION

---

Par rapport aux objectifs que nous nous sommes fixés, nous considérons le travail réussi. Nous sommes parvenus à identifier une série de modules et d'interfaces Java définissant l'organisation d'un système KIWIS et l'usage qu'on peut en faire.

L'attention portée à garder la description d'un module la plus indépendante possible des autres entités du système nous a permis de construire un noyau flexible et extensible.

L'implémentation que nous avons réalisée pour permettre l'utilisation immédiate de ce noyau s'appuie sur la plateforme AROM. En plus des avantages d'avoir une plateforme capable de gérer tout le cycle de vie des données qui soulage l'effort d'implémentation, le choix de cette plateforme a été motivé par l'adoption de celle-ci dans le cadre de plusieurs projets menés au sein de l'équipe. Cette implémentation AROM du noyau joue également un rôle de validation des capacités de la plateforme AROM.

Néanmoins, le temps n'a pas permis de réaliser une validation de cette implémentation. La validation nous aurait permis d'étudier les limites de cette première implémentation du noyau que nous avons proposé et de reconsidérer les décisions de conception qui se relèveraient mauvaises.

Parmi les perspectives immédiates de ce travail nous envisageons l'amélioration et la redéfinition de l'interface de conception et mise en œuvre des SIWs proposée par [BILA02] et présentée au chapitre 1. Ce premier travail que nous souhaitons réalisé constitue également une première validation de l'implémentation réalisée à l'aide d'AROM.

Une autre perspective ouverte par la réalisation de ce noyau est l'intégration de cette réalisation logicielle avec la plateforme BW Awareness Framework [KIRS02] mettant en œuvre des concepts tels que : le travail coopératif, la conscience de groupe, etc. Cette plateforme est développée par Manuele Kirsch, qui poursuit les travaux autour de cette plateforme dans le cadre de sa thèse réalisée au sein de l'équipe SIGMA. BW permettra de munir les acteurs des applications KIWIS avec des outils de communications spécifiques au travail coopératif.

Une troisième perspective est la réécriture de certaines applications dans le domaine des Systèmes d'Information Géographiques réalisées au sein de l'équipe, notamment les applications SPHERE [SPHE00] (surveillance des crues sur le bassin de l'Isère) et SIDIRA[SIDI00] (consultation de zone à risques avalanches).

Ces applications aux contenus informationnels très denses ne mettent pas en place un accès graduel à l'information ce qui rend assez difficile l'utilisation des systèmes par ceux qui consultent les informations pour la première fois. Le passage direct vers des applications disposant d'un accès progressif est assez difficile, puisqu'en ce moment les deux applications récupèrent les informations dans des bases de données en format propriétaire, ainsi la mise en place de cet accès ne peut pas se faire uniquement au niveau du code.

Dans notre vision, la mise en place de l'accès progressif au sein de ces applications en affectant au minimum les codes déjà écrits correspond au remplacement des bases de données propriétaires par une application KIWIS. Les fonctionnalités de cette application seront constituées par les requêtes qui sont réalisées sur la base de données. Néanmoins, il n'est pas forcément nécessaire d'exporter toute la base de données en AROM. Une solution dans ce

sens est l'implémentation d'un module de données spécifique constituant une interface entre les autres modules du système et la base de données en question.

La dernière perspective que nous discutons ici, constitue le déploiement des applications KIWIS en tant que services Web [WEBS03]. Ceci permettrait aux clients logiciels actifs sur le WEB de réaliser librement des interactions avec les applications installées sur un système KIWIS. Pour parvenir à cela nous envisageons l'extension des interfaces constituant le noyau KIWIS afin qu'elle soit conformes avec les recommandations W3C dans ce domaine.

# BIBLIOGRAPHIE

---

- [AROM01] M. Page, J. Gensel, C. Capponi, C Bruley, P. Genoud, D. Ziebelin, D. Bardou, V. Dupierris, *A new approach to Object-Based Knowledge Représentation: the AROM Systeem*, in Proceeding of the 14<sup>th</sup> International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEAAI&ES 2001), Budapest, Hungary, LNAI, juin 2001
- [BILA02] I.M. Bilasco, L'amélioration du prototype KIWIS, mémoire de Magistère informatique 1<sup>ère</sup> année à l'Université Joseph Fourier
- [GAMM95] E. Gamma., R. Helm, R.E. Johnson, J. Vlissides *Design Patterns Elements of Reusable Object-Oriented Software*, 1995, Addison-Wesley.
- [VILL01] M. Villanova-Oliver, J. Gensel, H. Martin, *Progressive Access to Knowledge in Web Information System through Zooms*, In Proceedings of Object-Oriented Information Systems (OOIS2001à, Calgary, Canada, August 2001
- [VILL02] M. Villanova-Oliver, *Adaptabilité dans les systèmes d'informations sur le Web : modélisation et mise en œuvre de l'accès progressif*, Thèse de doctorat de l'Institut National de Polytechnique de Grenoble, 2002
- [DAVI99] J.D. Davidson and D. Coward, *Java Servlet Specification*, v2.2, 1999, disponible à <http://javamug.org/Java>
- [XML98] T. Bray, J. Paoli and C.M. Sperberg-McQueen, Eve Maler. W3C Recommendation *Extensible Markup Language (XML) 1.0 (Second Edition)*, disponible à <http://www.w3.org/TR/REC-xml>, 10 February 1998
- [XSCH01] W3C Recommendation, *XML Schema, XML Schema Part 1: Structures*, disponible à <http://www.w3.org/TR/xmlschema-1/>, 2 May 2001
- [XSLT99] J. Clark, W3C Recommendation *XSL Transformation*, disponible à <http://www.w3.org/TR/xslt>, 1999
- [KICZ97] G. Kiczales, J. Lampingn A. Mendhekan C. Maeda, C.Videira Lopes, JM Loingtier, J. Irwin, *Aspect Oriented Programming*, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP) Finland, Spinger-Verlag LNCS 1241, june 1997
- [KICZ01] Gregor Kiczales<sup>1</sup>, Erik Hilsdale<sup>2</sup>, Jim Hugunin<sup>2</sup>, Mik Kersten<sup>2</sup>, Jeffrey Palm<sup>2</sup> and William G. Griswold<sup>3</sup>, *An overview of AspectJ*, LNCS vol. 2072, pages 327-355, 2001
- [KIRS02] M. Kirsch-Pinheiro, J.V.Lima; Borges, M.R.S. *A Framework for Awareness Support in Groupware Systems*. Proceedings of 7th International Conference on Computer Supported Cooperative Work in Design, UFRJ, Rio de Janeiro, Brésil, 2002. pp. 13-18.
- [SPHE00] D. Coeur, P.-A. Davoine, M. Lang, H. Martin, *Intégration de l'information historique dans un système d'information : le projet SPHERE*, Colloque SIRNAT'2000, Systèmes d'Information pour les Risques Naturels, Grenoble, Septembre 2000

- [SID100] Davoine P.A., Brunet R., Charrier P., Clavandié G., Favier R., Martin H, *Le projet SIDIRA : une approche pluridisciplinaire pour une meilleure appropriation des risques naturels via les nouvelles technologies de l'information*, Actes du colloque SIRNAT 2001 (Systèmes d'Information et Risques Naturels), 6 et 7 décembre 2001, Sophia-Antipolis..
- [WEBS03] D. Booth, H. Haas, F. McCabe, M. Champion, C. Ferris, D. Orchard, W3C Recommendation, *Web Services Architecture*, August 2003, disponible à <http://www.w3.org/TR/2003/WD-ws-arch-20030808/>