Université de Montréal

Intégration de services de raisonnement automatique basés sur les logiques de description dans les applications d'entreprise

par
Jacques Bergeron

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

décembre, 2012

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Intégration de services de raisonnement automatique basés sur les logiques de description dans les applications d'entreprise

présenté par:

Jacques Bergeron

a été évalué par un jury composé des personnes suivantes:

Michel Boyer,	président-rapporteur
Guy Lapalme,	directeur de recherche
Houari Sahraoui,	membre du jury

Mémoire accepté le: . . . . . .

RÉSUMÉ

Ce mémoire présente un patron d'architecture permettant, dans un contexte orienté-objet, l'exploitation d'objets appartenant simultanément à plusieurs hiérarchies fonctionnelles. Ce patron utilise un *reasoner* basé sur les logiques de description (web sémantique) pour procéder à la classification des objets dans les hiérarchies. La création des objets est simplifiée par l'utilisation d'un ORM (*Object Relational Mapper*). Ce patron permet l'utilisation effective du raisonnement automatique dans un contexte d'applications d'entreprise.

Les concepts requis pour la compréhension du patron et des outils sont présentés. Les conditions d'utilisation du patron sont discutées ainsi que certaines pistes de recherche pour les élargir. Un prototype appliquant le patron dans un cas simple est présenté. Une méthodologie accompagne le patron. Finalement, d'autres utilisations potentielles des logiques de description dans le même contexte sont discutées.

Mots clés: Génie logiciel, Orienté-objet, OWL, Web sémantique, ORM, Patron d'architecture, Classification, Polymorphisme, Ontologie, Hiérarchies de classes.

ABSTRACT

This master thesis presents a software architectural pattern for use in an object oriented environment to simultaneously access objects in multiple functional hierarchies. A Description Logics (Semantic Web) reasoner is used to classify the objects in the hierarchies. Object creation is simplifed by the use of an ORM - Object Relational Mapper. The pattern effectively allows automatic reasoning procedures to be used in an enterprise application context.

All concepts required to understand the architectural pattern and the tools are presented. Usage conditions for the pattern are discussed and research projects are presented to widen the pattern's applicability. A prototype applying the pattern on a simple problem is presented. A methodology is also presented. Finally, other potential uses of Description Logics based automatic reasoning procedures are discussed.

Keywords: Software Engeneering, Object orientation, OWL, Semantic web, ORM, Architectural pattern, Classification, Polymorphism, Ontology, Class hierarchies.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES

BD	Base de Données
BI	Business Intelligence - Intelligence d'affaires
DDD	Domain Driven Design
ERP	Enterprise Resources Planning - Progiciel de gestion d'entreprise
ETL	Extract Transform and Load
MAJ	Mise à jour
MSC	Most Specific Concept
O-O	Orienté-Objet
ORM	Object Relational Mapper
OWL	Web Ontology Language
RDF	Resource Description Framework
SGBD	Système de Gestion de Base de Données
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol (à l'origine)
W3C	World Wide Web Consortium
WSDL	Web Services Definition Language
XML	eXtensible Markup Language

NOTATION

[11]	Référence bibliographique
(2.3.1)	Référence à une section du document
reasoner	Terme anglais ou emphase
Etudiant	Nom de classe, de propriété ou d'individu
Etudiant(Jacques)	Prédicat
$\forall$	Quantificateur universel (pour tous)
$\exists$	Quantificateur existentiel (il existe)
$\vdash$	A pour conséquence logique (*entails*)
$\wedge$	Conjonction (et)
$\vee$	Disjonction (ou)
$\Rightarrow$	Implication
$\neg$	Négation
$\sqcap$	Intersection de classes
$\sqcup$	Union de classes
$\equiv$	Équivalence de classes
$\sqsubseteq$	Inclusion de classes (est sous-classe de)

À Dominic et Ginette

AVANT-PROPOS

Dès le début du processus visant l'obtention d'une maîtrise en informatique, nous nous sommes intéressés au concept du web sémantique. Notre curiosité s'est développée face aux mécanismes permettant de *raisonner* sur une ontologie.

Comme nous avions déjà plus de vingt (20) années d'expérience en développement et en gestion d'applications au sein de grandes organisations et que nous avions récemment terminé notre baccalauréat en génie logiciel, la possibilité d'utiliser les outils du web sémantique pour *raisonner* dans une application d'entreprise s'est avérée un sujet naturel pour nous.

Malheureusement, nous n'avons pas trouvé de travaux fournissant des pistes de recherche déjà défrichées. Ce travail n'est donc pas la suite ou le complément d'une thèse passée. Cela a rendu plus difficile l'identification d'un objectif précis et d'une démarche claire.

La découverte du rôle que le raisonnement automatique peut jouer pour rendre plus naturelle l'utilisation de hiérarchies fonctionnelles multiples en orienté-objet et de la simplification amenée par l'utilisation d'un ORM - *Object Relational Mapper* dans la création d'objets complexes en fut d'autant plus stimulante.

Nous espérons pouvoir vous transmettre une partie de notre enthousiasme dans la suite du document.

CHAPITRE 1

INTRODUCTION

« Programming is fun but developing quality software is hard » [10] *Philippe Kruchten*

1.1 Présentation du sujet

L'objectif de notre mémoire est d'étudier comment le raisonnement automatique, basé sur les logiques de description, pourrait être mis à profit pour traiter efficacement le problème des hiérarchies fonctionnelles multiples dans les applications d'entreprise sans changer les paradigmes de base de ces applications et en utilisant des outils de raisonnement existants.

Nous faisons référence à des applications d'entreprise pour restreindre notre sujet à des systèmes logiciels complexes. De tels systèmes amènent des problèmes particuliers et difficiles. Le génie logiciel est le domaine de l'informatique qui aborde ces problèmes, en particulier en amenant une structure dans ces systèmes et les méthodes employées pour les développer.

Avec les années, des façons d'aborder ces problèmes (des paradigmes) et des outils pour assister les concepteurs de logiciel ont été développés et permettent la réalisation de systèmes de grande envergure. Toutefois, les langages de programmation utilisés pour développer la plupart des applications d'entreprises (Java, C#) implantent une version limitée du paradigme orienté-objet (langages à classes avec types statiques) qui posent des limites importantes lorsque l'on veut travailler avec des applications traitant plusieurs fonctionnalités simultanément.

Parallèlement, le raisonnement automatique, une branche de l'intelligence artificielle, a connu des progrès importants vis-à-vis de la complexité des problèmes abordés et de la maturité des outils disponibles. Les logiques de description, une forme de raisonnement automatique, permettent de *raisonner* en contrôlant la complexité du calcul et sont donc

utilisables pour des problèmes « industriels ». Au cours des années 2000, les logiques de description ont connu une forte popularité, car elles représentaient le fondement théorique de ce qui a été appelé le web sémantique. Il s'agissait d'un projet pour interconnecter les *données* du web et ainsi permettre à des agents automatisés de *raisonner* pour répondre à des questions complexes.

Indépendamment du web sémantique, ce mémoire vise à utiliser les outils de raisonnement automatique basés sur les logiques de description pour repousser certaines limites des langages de programmation utilisés dans le développement d'applications d'entreprise. En ce sens, nous nous situons à la frontière du génie logiciel et de l'intelligence artificielle.

Un outil est *utile*, en génie logiciel, si, face à un problème satisfaisant des conditions bien définies, il peut être réutilisé de façon systématique pour résoudre ledit problème. Cette utilisation systématique de méthodes et d'outils se fait en ayant recours à des patrons de conception, pour les programmes, et à des patrons d'architecture, pour la structure de l'application.

Notre projet de mémoire élabore un patron d'architecture détaillé pour montrer de façon concrète *l'utilité* des logiques de description dans un contexte d'applications d'entreprise. Un prototype opérationnel a été développé pour illustrer que l'approche proposée pouvait être réalisée avec des outils normalement utilisés pour développer des applications d'entreprise. De plus, le mémoire étudie certaines conditions générales restreignant les architectures et les problèmes pouvant être envisagés pour l'utilisation des logiques de description.

L'approche proposée utilise les avantages de logiques de description mais en ayant le moins d'impact possible sur les ressources humaines et les processus de développement.

On peut faire un parallèle avec les bases de données relationnelles qui reposent sur un modèle théorique différent de l'approche orienté-objet et qui demandent l'intervention de ressources spécialisées pour l'intégration et l'optimisation des bases de données complexes. Cependant, des méthodes et outils ont été développés (ex. : les ORM) pour rendre compatibles l'utilisation des bases de données relationnelles avec l'approche orienté-objet sans que les développeurs n'aient à comprendre la théorie du modèle relationnel

ou soient des experts en configuration de bases de données. Nous visons le même genre de résultats dans le cas des logiques de description.

1.2 Structure du document

Le chapitre 2 définit ce que nous entendons par *application d'entreprise*, présente les concepts de base requis pour comprendre l'objectif de recherche et pose le problème, l'objectif, la question et la démarche de recherche.

Le chapitre 3 définit le cadre théorique en présentant les approches actuellement utilisées face au problème, certaines caractéristiques des ORM - *Object Relational Mapper* et surtout les logiques de description, dont certains aspects sont reportés en annexe.

Le chapitre 4 décrit le patron d'architecture principal élaboré dans ce mémoire : la classification en lots. On aborde également les étapes à réaliser (méthodologie) pour mettre en place le patron.

Le chapitre 5 aborde les conditions générales qui ont influencé les caractéristiques et les limites de notre patron. Nous évoquons également les extensions à envisager sous d'autres conditions initiales.

Le chapitre 6 évalue les résultats obtenus aux chapitres 4 et 5.

Le chapitre 7 décrit des suites possibles à ce projet et une présentation des pistes de recherche.

Le chapitre 8 conclut en rappelant notre contribution principale.

Le document inclut également cinq annexes détaillant certains sujets importants avec plus de détails sans être essentiels pour comprendre l'argumentation du mémoire.

La présentation du prototype développé avec ce mémoire est reportée aux annexes IV et V pour mettre l'emphase sur le patron d'architecture dans le corps du texte.

CHAPITRE 2

PRÉSENTATION DE LA PROBLÉMATIQUE

Le but du chapitre est de présenter l'objectif de recherche du mémoire. Nous présentons d'abord deux thèmes requis pour comprendre notre objectif, soit notre caractérisation d'une application d'entreprise et une première discussion des concepts d'objets et de classes, omniprésents dans la suite.

2.1 Applications d'entreprise

Le Gartner Group définit le terme « applications d'entreprise » de la façon suivante :

> « Software products designed to integrate computer systems that run all phases of an enterprise's operations to facilitate cooperation and coordination of work across the enterprise. The intent is to integrate core business processes (e.g., sales, accounting, finance, human resources, inventory and manufacturing). The ideal enterprise system could control all major business processes in real time via a single software architecture on a client/server platform. Enterprise software is expanding its scope to link the enterprise with suppliers, business partners and customers. [8] »

Pour notre propos, nous considérerons les applications d'entreprise comme toute application possédant un certain nombre de caractéristiques présentées au tableau 2.I.

2.1.1 Requis non-fonctionnels

Une application d'entreprise doit posséder un certain nombre d'attributs non-fonctionnels pour remplir sa mission. Voici un extrait de la liste du « Perl 5 Enterprise Project » comme exemple [1] :

> **Availability** - the assurance that a service/resource is always accessible

[1] Le texte n'est pas traduit puisqu'il s'agit d'une citation

Tableau 2.I – Caractéristiques des applications d'entreprise

	Caractéristique	Description
1	Grandes équipes de développement	Applications complexes demandant des processus structurés pour le développement et l'entretien.
2	Requis non-fonctionnels contraignants	En plus de réaliser les fonctionnalités requises les applications doivent posséder plusieurs caractéristiques essentielles (2.1.1) : disponibilité, robustesse, performance, intégrité, sécurité, etc.
3	Couvrent plusieurs domaines fonctionnels	Certaines informations auront des interprétations différentes à l'intérieur d'une même application (2.1.2).
4	Programmées selon le paradigme orienté-objet	Les environnements Java / Websphere et C# / .Net disposent chacun d'un langage orienté-objet pour l'écriture du code et d'environnements d'exécution sophistiqués fournissant tous les services requis.
5	Utilisent des bases de données relationnelles	Au moins un des SGBD - Système de Gestion de Bases de Données - Oracle ou SQL Server est utilisé dans la presque totalité des applications d'entreprise.

Scalability - the ability to support the required quality of service as the load increases

Reliability - the assurance of the integrity and consistency of the application and all of its transactions.

Security - the ability to allow access to application functions and data to some users and deny them to others

Interoperability - the ability of the system to share data with external systems and interface to external systems. [15]

Les composantes à utiliser dans le développement des applications d'entreprise doivent donc être choisies non-seulement selon leurs fonctionnalités mais également selon qu'elles remplissent ou non chacun des requis non-fonctionnels propres à l'application. Par exemple, le logiciel servant à conserver les données de façon permanente devra être sécurisé, performant, à haute disponibilité, devra isoler les transactions, etc. Ces contraintes limitent les composantes utilisables et nous forcent à construire des architectures applicatives plus complexes. Dans l'exemple précédent, un SGBD sera normalement utilisé pour conserver les données de façon permanente (persistance) pour s'assurer que l'application satisfasse ses requis non-fonctionnels.

2.1.2 Plusieurs domaines fonctionnels

Une application d'entreprise couvre plusieurs domaines fonctionnels à l'intérieur d'une organisation. Les mêmes informations ou données auront alors des interprétations différentes à l'intérieur d'un même système.

Un système de gestion des dossiers étudiants devra couvrir plusieurs domaines fonctionnels tels que l'admission, l'aide financière, l'inscription, les services aux étudiants, les résultats aux évaluations, etc.

Dans chacun de ces domaines, le concept d'étudiant sera analysé de façon différente. Pour l'inscription, le domaine d'étude et le programme seront essentiels (étudiants en médecine ou en administration). Pour l'aide spécialisée aux étudiants, on s'intéressera à connaître leur origine ; pour les bourses, il faudra savoir s'ils sont gradués ou non et ainsi de suite. Chacun de ces « types d'étudiants » viendra avec ses caractéristiques et ses traitements particuliers.

Le problème de l'appartenance « d'objets » à plusieurs « types » n'est pas nouveau. Le terme « Classification multiple et dynamique » est par exemple utilisé par James Odell pour décrire cette situation [13].

2.2 Objets et classes

2.2.1 Paradigmes de programmation

Le génie logiciel permet d'aborder les problèmes selon différentes approches appelées « paradigmes de programmation ». Plusieurs auteurs ont étudié les paradigmes de programmation. Nous suivons ici l'exposé de Peter Van Roy [18]. Un « paradigme de programmation » est une approche pour la programmation d'un ordinateur basée sur une théorie mathématique ou un ensemble cohérent de principes.

Van Roy identifie 27 de ces paradigmes. Les applications d'entreprises en utilisent essentiellement deux. Le premier est la famille de la « programmation logique et relationnelle » qui est utilisé par les SGBD des applications d'entreprises à travers le langage SQL. Le second est la « programmation orientée-objet » utilisée par les langages Java et

C# qui réalisent la logique d'affaire de la plupart des applications d'entreprise.

2.2.2 Orienté-objet

On se concentrant sur les langages de programmation orienté-objet, on peut identifier les langages basés sur les prototypes (JavaScript) et ceux basés sur les classes (Java, C#, Python). Ces derniers sont typiquement séparés à leur tour selon que la validation des types est faite de façon statique (Java) ou dynamique (Python).

Antero Taivalsaari présente bien l'avantage des langages basés sur des prototypes au niveau de la flexibilité et les limites des langages statiques basés sur les classes quand à la capacité de modélisation du monde réel [16]. Il ajoute cependant :

« The majority of business applications are such that they can fairly be represented with concepts that are defined in terms of shared properties. Furthermore, despite the inherent limitations, the modeling capabilities of the object-oriented paradigm are still much better then those of the other well-known programming paradigms. Consequently, the realization that object-oriented programming is not really capable of modeling the real world is more an observation than a real problem. »

Des concepts basés sur des propriétés partagées correspondent aux classes des langages basés sur des classes.

Les taxonomies des langages orientés-objet considèrent le concept de classe comme inhérent à celui de l'orienté-objet. Voir par exemple la taxonomie de Deborah Armstrong [2]. De plus, les présentations typiques des langages de programmation mettent l'accent sur les principaux langages qui demeurent basés sur les classes et avec des types statiques. Voir l'article de Aho par exemple [1].

Dans la suite de ce mémoire, nous aborderons l'approche orienté-objet dans le contexte des langages à classes avec types statiques. Les langages utilisés dans les applications d'entreprise font partie de cette catégorie. Nous verrons dans les paragraphes suivant que cette décision amène un problème au niveau du traitement des hiérarchies fonctionnelles multiples. Au lieu de « changer de paradigme » pour régler le problème nous proposerons un patron d'architecture permettant de traiter le problème en conservant les mêmes langages.

2.2.3 Caractéristiques des objets et classes

Les concepts de base sont utilisés tout au long du document.

- Un *objet* est la représentation d'une entité du monde réel pouvant être désignée (on peut l'identifier parmi les autres objets). Par exemple, Jacques et MaitriseEnInformatique sont des objets.
- Les objets peuvent être mis en relation à l'aide de *propriétés*, qui spécifient la nature de la relation. Par exemple, dans Jacques estInscrit MaitriseEnInformatique, estInscrit est une propriété. Nous appellerons ce type de propriétés des *propriétés référentielles*.
- D'autres *propriétés* sont utilisées pour rattacher des valeurs scalaires aux objets. Par exemple, dans Jacques Mesure 1,80m, Mesure est une propriété. Nous appellerons ce genre de propriétés des *propriétés scalaires*.
- Les objets peuvent également avoir des *comportements*, tout comme les objets du monde réel. Par exemple, une figure géométrique peut se dessiner sur un canevas.
- Les *classes* sont des ensembles d'objets ayant des propriétés et/ou des comportements communs.

Le fait de pouvoir désigner un objet permet de lui assigner un identifiant. C'est l'identifiant qui est utilisé par la suite pour référer à l'objet à travers des propriétés référentielles.

2.2.4 Trois façons de voir

Ce mémoire traite trois façons différentes de définir et d'utiliser les concepts d'objets et de classe.

2.2.4.1 Langages orienté-objet basés sur des classes avec types statiques

C'est l'approche utilisée pour programmer les applications d'entreprise. On définit des classes à priori en décrivant leurs propriétés et leurs comportements. Lorsque l'on créé un objet on doit préciser la classe à laquelle il appartient.

L'objectif visé est de regrouper tout le code se rapportant à une classe d'objets au

même endroit. Chaque classe devient une petite composante avec un but précis : modéliser une entité ou un processus du domaine de l'application. Ce regroupement a un effet structurant sur l'ensemble de l'application. Comme chaque morceau de code *est à sa place*, on peut envisager des applications plus complexes mais qui demeurent contrôlables.

2.2.4.2 Modèle relationnel

C'est l'approche utilisée pour conserver les données de façon permanente dans les applications d'entreprise. On définit des tables ayant un ensemble de colonnes correspondant aux propriétés d'objets. Les objets sont conservés comme des lignes des tables. L'une des colonnes de la table contient les identifiants des objets. L'intersection d'une ligne et d'une colonne donne donc la valeur d'une propriété pour un objet en particulier. Il s'agira d'une valeur scalaire ou de l'identifiant d'une ligne, selon le type de propriété. L'appartenance d'un objet à une classe est déterminée par l'écriture d'une ligne dans la table correspondant à cette classe.

Notons finalement que dans le modèle relationnel, une propriété ne peut avoir qu'une seule valeur pour un objet donné. Cette contrainte exige l'introduction de tables de liaison lorsque des valeurs multiples sont requises.

2.2.4.3 Logiques de description

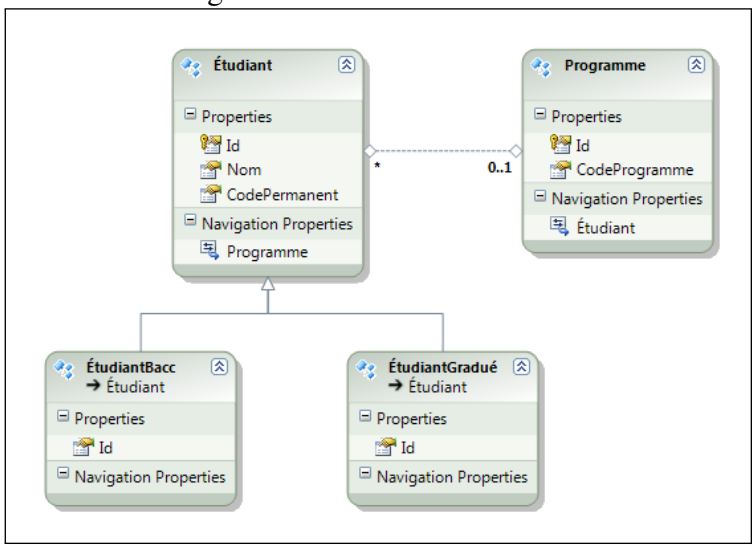
Contrairement aux approches précédentes, les logiques de decription définissent des objets comme des entités indépendantes des classes. On précise les propriétés d'un objet et on *déduit* son appartenance ou non à une classe donnée à partir de la valeur de ses propriétés et de la *définition* de la classe. Les logiques de description ne sont pas utilisées actuellement dans les applications d'entreprise. Le chapitre suivant abordera les logiques de description beaucoup plus en détail.

2.2.5 Hiérarchies fonctionnelles en orienté-objet

Tel que décrit plus haut, dans le modèle orienté-objet, un objet particulier (individu) n'existe que comme l'instanciation d'une classe. Ainsi, l'individu Jacques est une instance de la classe EtudiantGradue et les caractéristiques et comportements applicables à cette instance sont déterminés par sa classe de définition.

On a également des relations de dérivation entre les classes qui définissent une hiérarchie fonctionnelle permettant de déterminer que Jacques est également un Etudiant parce que tous les EtudiantGradue sont également des Etudiant (généralisation). Les classes dérivées *héritent* des caractéristiques et comportements de leurs généralisations ; ce qui permet d'élargir les caractéristiques que l'on peut associer à un individu. Ainsi Jacques possède à la fois les caractéristiques d'un EtudiantGradue et d'un Etudiant.

Figure 2.1 – Modèle de classes



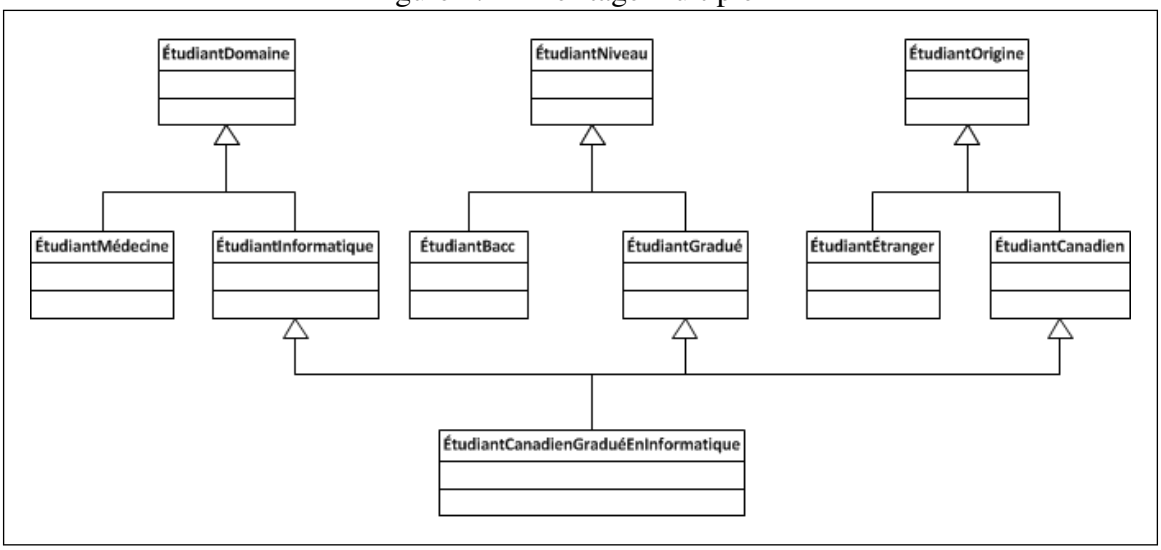
On illustre un modèle objet avec un diagramme de classes tel que la figure 2.1 où une ligne simple indique une association entre deux classes à travers une propriété référentielle et où l'utilisation du triangle indique une relation de dérivation (ÉtudiantGradué est une *sous-classe* de Étudiant qui est alors appelée sa *super-classe*).

La section 3.2.2 abordera le concept de polymorphisme, très lié à celui des hiérarchies de classes.

2.2.6 Limites

Si on a besoin de plusieurs hiérarchies fonctionnelles pour modéliser un domaine, on doit placer un individu dans chacune de ces hiérarchies. La façon d'y arriver est de définir une classe ayant une généralisation dans chacune des hiérarchies. Ainsi, si on veut définir des hiérarchies par domaine d'étude, par niveau et par origine, on peut définir la *classe d'intersection* des EtudiantCanadienGradueEnInformatique pour couvrir les trois hiérarchies et définir Jacques comme une instance de cette classe.

Figure 2.2 – Héritage multiple



La figure 2.2 illustre cette situation. À noter que nos hiérarchies sont simplifiées pour fin de présentation. Dans la réalité, une hiérarchie aura plusieurs niveaux et pourra compter des dizaines, voire des centaines de classes.

Le nombre de classes d'intersection à définir est exponentiel par rapport au nombre de hiérarchies fonctionnelles et à la complexité de ces hiérarchies. La nécessité de définir un individu comme une instance d'une classe produira une explosion de la hiérarchie de classes avec la complexification fonctionnelle de l'application.

Ce problème est exacerbé dans le contexte des applications d'entreprise puisque celles-ci couvrent plusieurs domaines fonctionnels et multiplient de ce fait, les hiérarchies fonctionnelles. De plus, une complexité supplémentaire est liée au fait que les

langages utilisés pour le développement des applications d'entreprise (Java, C#) ne supportent pas l'héritage multiple.

On ne peut pas directement *résoudre* ce problème dans le contexte des langages orienté-objet basés sur des classes avec types statiques puisqu'il est inhérent à l'approche elle-même. Plusieurs stratégies ont cependant été élaborées pour mitiger le problème et certaines sont utilisées dans la pratique. Ce mémoire considère les logiques de description comme l'une de ces stratégies de mitigation.

2.2.7 Rôle potentiel des logiques de description

Les logiques de description abordent le problème des hiérarchies fonctionnelles de façon différente. Les individus ne sont pas attachés à une classe en particulier. On définit les propriétés applicables à tel ou tel individu et quelle valeur (ou ensemble de valeurs) est affectée à chacune de ces propriétés pour cet individu.

Les classes quant à elles, sont définies comme étant l'ensemble des individus possédant tel ensemble de propriétés avec telles conditions sur les valeurs de ces propriétés.

Nous reviendrons sur ces concepts dans le chapitre suivant (cadre théorique) mais on peut déjà imaginer que cette approche simplifie la définition de la hiérarchie de classes car on peut *déduire* l'appartenance d'un individu à une classe à partir de la définition de la classe et de la valeur des propriétés de l'individu. On évite ainsi la définition de micro-classes à l'intersection des hiérarchies fonctionnelles. Les logiques de descriptions peuvent donc, en théorie, aider à gérer le problème des hiérarchies fonctionnelles multiples dans les applications d'entreprise.

2.3 Problème et objectifs de recherche

Dans ce mémoire, nous nous intéresserons au problème de la « complexification de la hiérarchie des classes dans le paradigme orienté-objet dans le contexte de hiérarchies fonctionnelles multiples ».

C'est un problème rencontré dès les premières étapes de la conception orienté-objet et qui s'aggrave rapidement avec la complexité fonctionnelle d'une application.

Notre objectif de recherche est d'étudier comment le raisonnement automatique, basé sur les logiques de description, pourrait être mis à profit pour traiter efficacement le problème des hiérarchies fonctionnelles multiples dans les applications d'entreprise sans changer les paradigmes de base de ces applications et en utilisant des outils de raisonnement existants.

Nous posons deux restrictions à notre approche :

- Nous supposons que les logiques de description ne remplaceront pas l'approche orienté-objet comme paradigme de base des applications d'entreprise.
- Nous utiliserons les *reasoners* [2] existants.

Nous définissons notre recherche en spécifiant des objectifs détaillés et une démarche à suivre pour les remplir. Le résultat de cette analyse est présenté au tableau 2.II.

Au chapitre suivant nous présenterons certaines stratégies actuelles pour le même problème. Nous n'essaierons cependant pas de comparer les logiques de description avec ces stratégies. Nous voulons donc établir que les logiques de description constituent une solution, sans tenter de montrer si elles sont la meilleure possible.

2.3.1 Pertinence du projet de recherche

Nous défendrons une contribution de notre mémoire selon quatre aspects :

1. Présentation d'une analyse originale de la pertinence des logiques de description dans le contexte des applications d'entreprise.
2. Poser les bases en vue de fournir à la communauté une interface avec les *reasoners* OWL couramment utilisés à partir de la plateforme .NET.
3. Fournir une méthodologie pour enrichir une application d'entreprise avec un *reasoner* OWL pour traiter le scénario de base.

[2] Comme nous le verrons plus loin, un *reasoner* est un outil permettant, entre autres, de faire du raisonnement automatique avec des logiques de description. La traduction *moteur d'inférence* est parfois utilisée mais ne couvre qu'une partie de la fonctionnalité des outils. De ce fait nous avons préféré conserver le nom original.

Tableau 2.II – Objectifs détaillés

	Objectif	Sommaire	Démarche
1	Établir la faisabilité.	S'assurer que les logiques de description peuvent effectivement être utilisées dans un contexte d'applications d'entreprise.	1. Caractéristiques des logiques de description qui seraient utilisables dans des applications d'entreprises. 2. Montrer qu'un aller-retour entre base de données et *reasoner* est faisable. 3. Requis non-fonctionnels satisfaits.
2	Identifier un scénario type.	Trouver une utilité significative aux logiques de description dans le contexte des applications d'entreprise.	1. Application possible aux hiérarchies multiples. 2. Limitation des approches actuelles. 3. Correspondance entre le treillis de subsomption et les hiérarchies multiples.
3	Architecture pour le scénario type.	Proposer une architecture applicative permettant d'intégrer les logiques de description dans le cas du scénario fonctionnel type.	1. Approche pour réintégrer le résultat de la classification dans l'application. 2. Alternatives pour la création d'objets. 3. Modèles du ORM.
4	Conception et réalisation du prototype.	Le prototype validera la pertinence du patron d'architecture.	1. Environnements et outils de développement. 2. Interface *reasoner*. 3. Traducteur de syntaxe de Manchester. 4. Convertisseur base de données vs axiomes.
5	Élaborer la méthodologie pour le scénario type.	Comment on s'y prendrait pour utiliser les logiques de description dans un scénario concret d'application d'entreprise en appliquant le patron d'architecture.	1. Requis pour patron d'architecture. 2. Structurer sous forme d'étapes.
6	Évaluer les forces et les faiblesses du patron d'architecture.	Bien caractériser dans quel cas le patron d'architecture est applicable.	1. Préciser les conditions d'utilisation. 2. Scénarios techniques applicables. 3. Scénarios fonctionnels applicables.
7	Étudier d'autres scénarios.	Voir si on peut étendre l'approche à d'autres scénarios, incluant la possibilité de ne pas renvoyer les résultats à l'application.	1. Synchronisation avec l'application. 2. Autres scénarios sans retour à l'application.

4. Identifier des pistes de réflexion pour utiliser un *reasoner* OWL pour aborder des scénarios plus complexes.

2.3.2 Ouverture de la question de recherche

Nous avons identifié quelques approches pour l'utilisation des logiques de description pour le développement d'applications, par exemple :

- Des outils et des méthodologies pour le développement d'applications ontologiques, c'est-à-dire d'applications utilisant le paradigme du web sémantique comme modèle de base [14].
- Des approches pour l'utilisation de OWL dans la composition de web services [7]. Nous avons inclus plus de détails sur cette approche en annexe pour le lecteur intéressé (Annexe I).

Nous n'avons cependant pas rencontré de tentative d'intégrer les logiques de description en minimisant l'impact sur les processus de développement actuellement utilisés pour les applications d'entreprise. La démonstration de la faisabilité d'une telle approche constitue donc une question de recherche ouverte.

***

Dans ce chapitre, nous avons présenté le concept d'application d'entreprise comme contexte de notre problématique. Nous avons montré comment les applications d'entreprise font ressortir un problème inhérent à l'approche orienté-objet et comment les logiques de description peuvent aider à mitiger ce problème. Finalement, nous avons présenté les éléments plus traditionnels d'un mémoire : problème de recherche, objectif de recherche, démarche, etc.

CHAPITRE 3

CADRE THÉORIQUE

Nous commençons par expliquer que le problème des hiérarchies fonctionnelles multiples est spécifique aux langages à base de classes en montrant comment on pourrait le régler en utilisant en dehors de ces langages.

Nous présentons ensuite différentes approches actuellement utilisées dans les langages à classes pour faire face au problème des hiérarchies fonctionnelles multiples. A noter que la section 3.2.2 introduit le concept de polymorphisme qui sera utilisé dans la suite.

On étudie ensuite certains aspects des ORM - *Object Relational Mapper* qui gèrent les passages du modèle relationnel au modèle orienté-objet et qui seront fortement utilisés dans le patron d'architecture présenté au prochain chapitre.

Finalement, nous commençons notre discussion des logiques de description. Nous abordons les éléments essentiels pour comprendre le patron d'architecture et nous reportons les autres aspects en annexe dans le but d'arriver le plus rapidement possible au prochain chapitre, qui arrime les logiques de description aux applications d'entreprise.

3.1 Approches générales de l'orienté-objet

On a vu au chapitre précédent 2.2 que nous nous limitions aux langages de programmation à base de classes avec types statiques du type Java et C# car ce sont ces langages qui sont utilisés pour la codification des règles d'affaires des applications d'entreprise. Nous développerons graduellement une approche pour traiter le problème des hiérarchies fonctionnelles multiples en conservant les outils actuellement utilisés. Cette approche prendra la forme d'un patron d'architecture intégrant un « reasoner » de logiques de description.

Nous commençons cependant par voir comment on pourrait aborder le problème sans la contrainte d'utiliser des langages à bases de classes avec types statiques tout en

restant dans le cadre de l'orienté-objet. Cette brève discussion permettra de bien préciser que l'approche prise dans ce mémoire n'est qu'une des façons d'aborder le problème.

La littérature caractérise l'orienté-objet de plusieurs façons [2]. Au niveau technique, on peut caractériser l'orienté-objet par la présence d'un mécanisme de répartition (dispatch) qui renvoi à l'objet la responsabilité de déterminer comment un appel de méthode ou un envoi de message sera effectivement traité [3]. Par exemple, un langage à base de prototype renverra un appel de méthode à l'objet prototype si l'objet destination ne surcharge la méthode appelée. De la même façon, un langage à base de classe avec types statiques pourra renvoyer l'appel à la classe de base en cas d'héritage sans surcharge de la méthode. Les techniques utilisées pour implanter ces répartitions sont basées sur des tables de pointeurs sur des fonctions. Ces tables sont plus ou moins dynamiques et spécifiques à chaque objet selon le type de langage utilisé.

On peut facilement imaginer des mécanismes de répartition plus complexes qui permettraient de régler le problème des hiérarchies fonctionnelles multiples directement au niveau du langage. Par exemple, Chambers [4] proposait le concept de « predicate classes » pour rendre le mécanisme de répartition des appels de méthodes dépendant de la valeur des propriétés de l'objet. Si les valeurs de propriétés de l'objet satisfont au prédicat associé à une classe, c'est la méthode de cette classe qui est exécutée.

Plus récemment, Bloom et al. [3] proposaient le langage Ferret implantant le même genre d'approche mais de façon optimisée pour le cas où les classes sont organisés sous forme de partitions. Ces partitions sont équivalentes aux hiérarchies fonctionnelles dont il est question dans le présent mémoire.

Nous aurions donc pu orienter notre travail en suggérant d'utiliser un langage du type de Ferret ou de continuer le travail sur un de ces langages. Ce genre de travail aurait possiblement été plus intéressant au niveau de la théorie du génie logiciel mais aurait complètement manquer notre objectif. Les applications d'entreprise ne seront pas développées avec Ferret ou un langage de ce type dans un avenir prévisible.

3.2 Approches actuelles

Décrivons certaines approches présentement utilisées pour mitiger le problème du traitement des hiérarchies multiples dans le cas des langages à base de classes avec types statiques. Ces exemples nous permettront de mieux comprendre le problème et les stratégies proposées dans la suite du mémoire.

3.2.1 Hiérarchies techniques et constructions génériques

Commençons par un des problèmes typiques de la programmation orientée-objet qui consiste à combiner une hiérarchie technique avec une hiérarchie fonctionnelle. Par exemple, pour définir une classe ListeEtudiantsGradues, on doit avoir les caractéristiques d'une liste, un type de collection, et les caractéristiques d'un étudiant gradué, un type d'étudiant.

La solution classique est d'utiliser l'héritage multiple, disponible dans certains langages (p.ex. C++), pour définir une classe ListeEtudiantsGradues dérivée à la fois de la classe Liste et de la classe EtudiantGradue. Cette solution est lourde car elle demande de définir une classe pour chaque type de liste et impossible dans les langages plus récents (Java, C#) où l'héritage multiple n'est pas disponible.

La solution apportée par tous ces langages consiste à permettre la définition de classes génériques référant à un type générique. Par exemple, Liste<T> serait une liste d'un type générique <T>, dérivée d'une classe Collection<T>. Le type générique est utilisé dans la définition des méthodes de la classe. Par exemple, la méthode void Add(T element) ajoutera un élément de type <T> à la liste.

On utilise cette classe en déclarant des objets d'un type particulier, par exemple une variable de type Liste<EtudiantGradue>. Concrètement en C# :

```
List<EtudiantGradue> inscrits = new List<EtudiantGradue>();
EtudiantGradue jacques = new EtudiantGradue();
inscrits.Add(jacques);
```

Cette construction syntaxique évite d'avoir à définir une classe dans le seul but de faire une intersection entre la hiérarchie technique des collections et la hiérarchie fonc-

tionnelle des étudiants. Dans les faits, la classe d'intersection est effectivement générée mais ce travail répétitif est effectué par le compilateur au lieu du programmeur.

Comme le cas de l'intersection d'une classe technique (Liste) et d'une classe fonctionnelle (EtudiantGradue) est très fréquent et ne peut être évité dans un programme orienté-objet, le problème a dû être résolu au niveau des langages de programmation.

Nous traitons un problème similaire dans ce mémoire, nous travaillons toutefois avec plusieurs hiérarchies fonctionnelles plutôt qu'avec une hiérarchie fonctionnelle et une hiérarchie technique.

3.2.2 Variables-type et indicateurs

La programmation orientée-objet permet de regrouper dans une même classe le comportement propre aux objets appartenant à cette classe (encapsulation) et à adapter le comportement d'un objet selon sa classe de définition (polymorphisme). Par exemple, si on a une liste d'étudiants certains étant des étudiants étrangers et d'autres des étudiants canadiens, la méthode ObtenirFraisBase aura un comportement différent sans que la logique du code traitant la liste d'étudiants n'ait à se préoccuper de la classe de l'objet. La méthode correspondant à la classe de l'objet sera appelée à l'exécution.

```
foreach (Etudiant etud in liste) {
    decimal frais = etud.ObtenirFraisBase();
    ...
}
```

Avant l'introduction de l'approche orienté-objet, on obtenait le même résultat à l'aide d'une variable-type ou d'un indicateur et d'un énoncé de branchement lors de l'utilisation des données. Une variable-type est une énumération qui précise la nature d'un objet. Par exemple, une variable sexe peut avoir les valeurs Masculin, Feminin ou Inconnu. Un indicateur est une variable-type avec deux valeurs encodées sous forme de Vrai ou Faux. Par exemple, un étudiant est-il un étudiant-étranger oui ou non.

```
foreach (Etudiant etud in liste) {
    decimal frais;
    if (etud.EstEtudiantEtranger) {
        frais = FraisEtudiantEtranger(etud);
```

```
    } else {
        frais = FraisEtudiantCanadien(etud);
    }
    ...
}
```

Ici EstEtudiantEtranger est un indicateur appliqué à la classe Etudiant qui permet de distinguer les étudiants étrangers. Elle joue le même rôle que les classes dérivées EtudiantCanadien et EtudiantEtranger. L'utilisation des variables-types et des indicateurs amène une multiplication des structures de branchement et une détérioration rapide du code avec l'augmentation de la complexité du système. C'est cette détérioration qui est résolue par la programmation orienté-objet dans le cas où on a une seule hiérarchie.

Si on est en présence de plusieurs hiérarchies fonctionnelles, la solution la plus simple est de ne garder qu'une seule hiérarchie et de remplacer les autres par des variables-types ou des indicateurs. Dans la pratique, un grand nombre de programmes « orienté-objet » sont en fait remplis d'indicateurs, de variables-type et de structures de branchement complexes (ex. : énoncés switch ou if imbriqués). Selon notre expérience, dans la plupart des cas, cette situation n'est pas due à une mauvaise compréhension du programmeur mais plutôt pour éviter d'avoir à définir une multitude de classes d'intersection entre les hiérarchies.

Les variables-type et les indicateurs sont traités en détail dans la théorie des bases de données relationnelles. Un des objectifs du processus de normalisation des bases de données est justement de se débarrasser de ces variables. Nous utiliserons plus loin ces variables-types et indicateurs, que nous appellerons champs de contrôle, mais de façon contrôlée pour éviter la dispersion du code.

3.2.3 Plusieurs hiérarchies à partir d'une même base de données

Une autre façon de traiter le problème des hiérarchies fonctionnelles multiples est de les considérer une à la fois. Dans l'exemple des étudiants (2.1.2), on développerait un module pour l'admission, un autre pour l'aide financière, un autre pour les résultats aux évaluations, etc. Chaque module classerait les étudiants selon la hiérarchie la plus pertinente pour la fonction considérée. Si d'autres hiérarchies sont requises par le module,

elles donneraient lieu à des variables-type et des indicateurs, selon le modèle présenté à la section précédente.

Toutes les méthodologies de développement d'applications comprennent une phase de découpage axée soit sur les fonctions (ex. : analyse structurée) ou bien sur les données (ex. : DDD - Domain Driven Design). L'idée de découper un système pour en faciliter la compréhension semble naturelle et est acceptée d'emblée.

Une fois le découpage réalisé, chaque module aura une « vue » particulière des données. Cette vue sera centrée sur la hiérarchie principale utilisée par les entités du système. Dans notre exemple, le module d'inscription abordera les étudiants selon leur domaine et programme d'étude.

Le découpage par module, traitant les entités sous différentes hiérarchies, demande que les données soient conservées sous un format pouvant être interprété selon ces vues.

Pour être utilisable dans une application d'entreprise, le modèle relationnel doit pouvoir être mis en correspondance avec les hiérarchies fonctionnelles utilisées par les modules. Ce travail, effectué historiquement par des « couches d'accès aux données » est maintenant réalisé de façon plus formelle par les ORM (*Object Relational Mapper*). Chacun de ces logiciels comprend son propre langage pour effectuer la correspondance entre le modèle de la base de données et un modèle objet, où chaque entité de base sera représentée par une hiérarchie fonctionnelle. Nous reparlerons de ces composantes à la section suivante.

Notons qu'il y a des limites à organiser le système pour traiter les hiérarchies une à la fois. Le module de facturation, dans l'exemple des dossiers étudiants, doit les considérer aussi bien par domaine d'étude que selon leur origine ou leur niveau. De tels modules sont reconnus pour avoir recours à plusieurs tables de taux et de règles, qui sont en fait des variables-type avec les procédures de branchement correspondantes. Les systèmes de gestion de ressources humaines sont d'autres exemples de ce type. On veut éviter d'avoir à découper en modules plus que ce qui est nécessaire selon le domaine de l'application.

3.3 ORM - Object Relational Mapper

Comme indiqué à (2.1) les applications d'entreprise utilisent le plus souvent des langages orienté-objet pour la programmation et des bases de données relationnelles pour conserver les *objets* de façon permanente (persistance).

Plusieurs projets ont tenté de réaliser un outil d'interface entre les modèles objet et les bases de données relationnelles. Le projet de logiciel libre *Hibernate*[9] a commencé, en 2001, le développement d'un outil autonome effectuant cette interface de façon efficace pour l'environnement Java. Le nom officiel donné à un tel logiciel est « *Object Relational Mapping library* ». L'environnement .NET possède un tel module appelé *Entity Framework*[11]

3.3.1 Fonctionnement général

La figure 3.1 décrit le fonctionnement général du *Entity Framework* de .NET. Ce fonctionnement est typique d'un ORM.

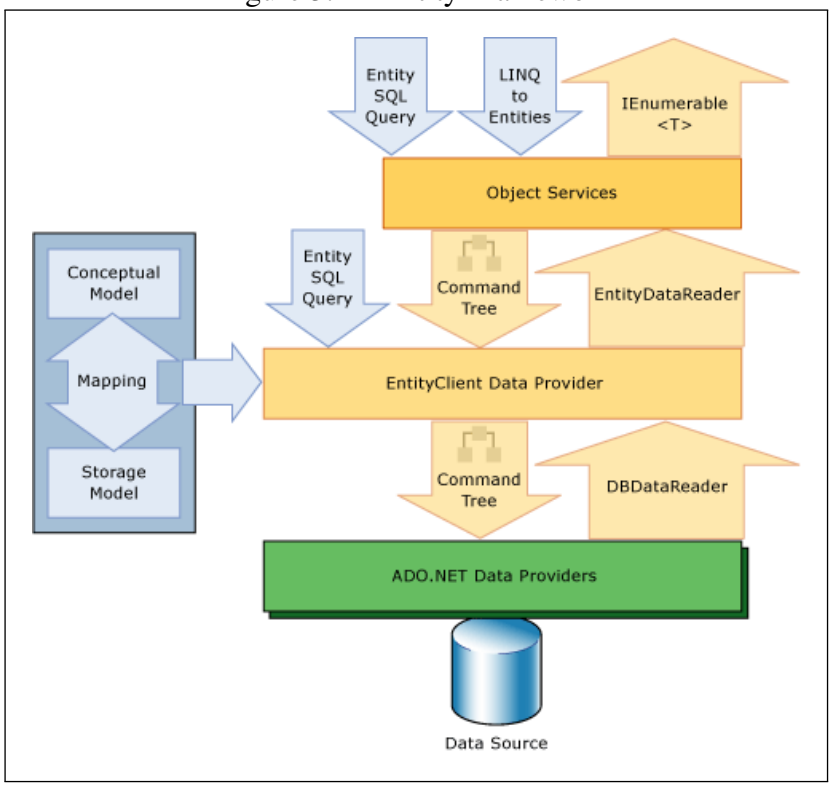
On doit définir trois (3) modèles :

- Un modèle objet au niveau conceptuel (*conceptual model*) décrivant les « entités » (classes) logiques que l'on désire manipuler dans l'application.
- Un modèle de la base de données (*storage model*) décrivant la structure des tables.
- Une correspondance (*mapping*) entre les deux modèles précédents.

Ces modèles sont en XML mais peuvent être générés à partir d'outils (rétro-ingénierie de la base de données, éditeur graphique du modèle conceptuel, etc.). Un outil génère également des classes correspondant au modèle conceptuel dans le langage utilisé par l'application (ex. C#).

À l'exécution, l'application utilise ces objets pour interagir avec les librairies (*runtime*) qui interprètent les modèles pour générer les commandes destinées à la base de données. L'application n'interagit plus directement avec la base de données.

Figure 3.1 – Entity Framework



En plus du service de « traduction » entre les modèles, la couche ORM rend toute une gamme de service à l'application (gestion de transactions, cache, lecture paresseuse - *lazy loading*, etc.).

3.3.2 *Mappings* particuliers

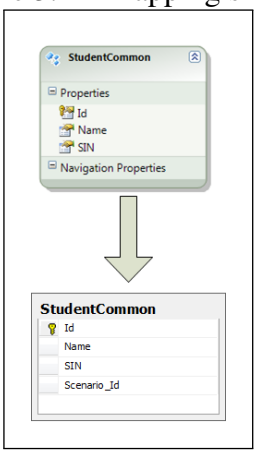
La caractéristique des ORM qui sera utilisée dans ce mémoire est le découplage qui est amené entre le modèle de classes utilisé par l'application et le modèle des tables de la base de données.

Nous utiliserons trois (3) *mappings* typiques dans la suite.

3.3.2.1 *Mapping* simple

C'est le *mapping* le plus direct (figure 3.2), une classe du modèle objet de l'application correspond à une table de la base de données.

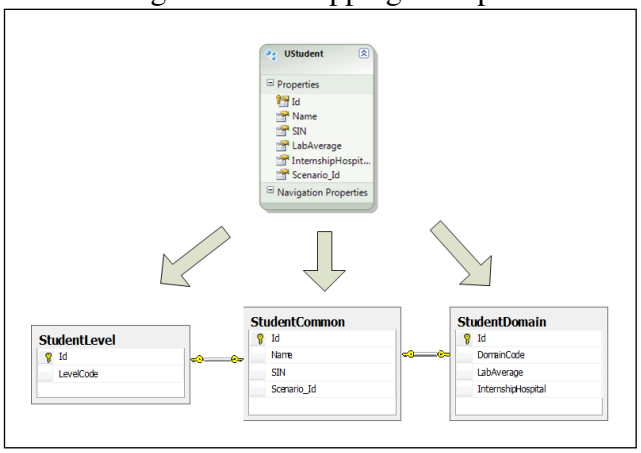
Figure 3.2 – Mapping simple



Le ORM n'apporte rien au plan du mapping mais offre tout de même les autres services (ex. transaction).

3.3.2.2 *Mapping* multiple

Figure 3.3 – Mapping multiple



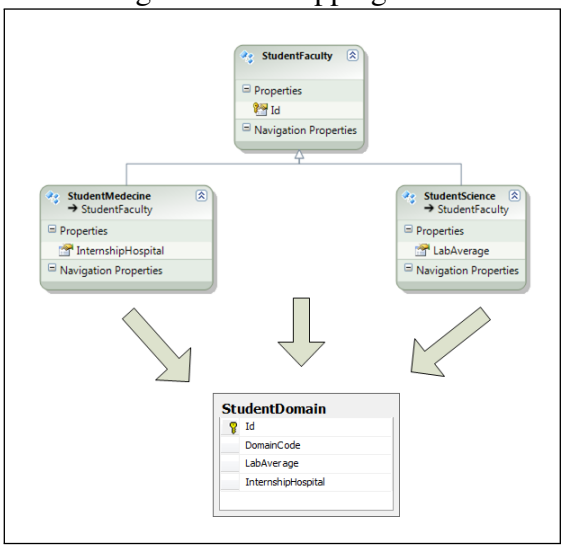
Ici, une classe (entité) du modèle objet correspond à plusieurs tables de la base de données. Les tables doivent toutes avoir la même clé primaire (identifiant) que la classe.

C'est le ORM qui s'occupera d'appliquer les mises à jour correctement à l'ensemble des tables. Par exemple, l'ajout d'un objet dans la classe impliquera l'ajout d'une ligne dans chacune des tables.

Le ORM fait donc une consolidation d'informations pouvant provenir de plusieurs modules gérant chacun leur partie de la base de données.

3.3.2.3 *Mapping* TPH

Figure 3.4 – Mapping TPH



Le *mapping* de base entre un modèle objet et un modèle relationnel est de faire correspondre une classe à une table. Un objet (instance d'une classe) correspond alors à une rangée dans la table.

Le problème principal provient du fait que le modèle relationnel n'a pas de concept d'héritage entre les tables. Plusieurs patrons de correspondances sont supportés par les ORM. Dans ce mémoire nous utiliserons seulement le *mapping* TPH pour *Table Per Hierarchy* ou Table Par Hiérarchie. L'idée est que toutes les classes d'une hiérarchie sont liées à une même table.

La table comprend des champs pour toutes les propriétés de toutes les classes de la hiérarchie. Comme certaines propriétés ne sont définies qu'à partir d'un niveau donné de la hiérarchie, les champs de la table ne sont pas nécessairement applicables pour tous les objets. Il n'y a pas de problème pour l'écriture d'un objet dans la base de données en autant que toutes les classes de la hiérarchie aient le même identifiant, qui doit être un champ de la classe à la racine de la hiérarchie.

Lors de la lecture, le ORM doit créer le nouvel objet dans la bonne classe. Le *mapping* doit inclure des conditions qui permettent de déterminer la classe de l'objet. La plupart du temps, il s'agit simplement d'un test sur un champ de contrôle de la table. Si tel champ de contrôle a telle valeur, alors l'objet doit être instancé par une telle classe. Par exemple, dans la figure 3.4, c'est le champ DomainCode qui sert de champ de contrôle. Si DomaineCode a la valeur « StudentMedecine » alors le ORM crée l'objet dans la classe StudentMedecine.

3.4 Logiques de description et classification

Dans cette section, nous montrons que le raisonnement automatique basé sur les logiques de description permet de réaliser une classification d'objets dans des hiérarchies fonctionnelles multiples. Nous procédons en quatre étapes en montrant :

- une introduction sur les logiques ;
- le principe général des logiques de description ;
- des exemples de descriptions de classe en utilisant le langage OWL (Annexe II) ;
- le mécanisme permettant de *raisonner* sur les hiérarchies de classes et sur l'appartenance d'un individu à une classe (Annexe III).

Le but du mémoire n'est pas d'expliquer en détail le fonctionnement des logiques de description mais plutôt d'analyser leur utilisation dans un contexte particulier. Nous vous référons à « The Description Logic Handbook - An introduction to description logics » [12] et aux articles suivants du même ouvrage pour une couverture plus complète. Nous nous limitons aux points suivants :

- L'objectif et la syntaxe de base du langage OWL qui sera utilisé pour les exemples dans la suite du mémoire.
- L'utilité d'un *reasoner* et le mécanisme permettant de remplacer la classification manuelle par une procédure automatique.

- La nécessité d'élaborer des définitions formelles des classes de la hiérarchie dans laquelle on désire classifier des objets.

3.4.1 Logiques et représentation des connaissances

Nous commençons notre exposé des logiques de description par quelques remarques sur les logiques, la représentation des connaissances et le raisonnement automatique.

Le concept de *raisonnement* réfère à deux types de processus : le raisonnement inductif et le raisonnement déductif.

Le raisonnement inductif correspond à découvrir des règles générales à partir de l'observation de cas particuliers, on parlera de régression dans le domaine des mathématiques ou d'apprentissage machine dans le domaine de l'informatique. Ce mémoire s'intéresse plutôt au raisonnement déductif.

Lorsque nous raisonnons de façon déductive, nous appliquons des règles de raisonnement, appelées règles d'inférence, sur de l'information connue, pour obtenir de la nouvelle information. Ainsi si l'on sait que :

1. une proposition logique peut être soit vraie, soit fausse mais pas les deux ;
2. $A \Rightarrow B$ (lire A implique B), c'est-à-dire si la proposition A est vraie, alors la proposition B doit aussi être vraie ;
3. la proposition A est vraie.

Alors on peut conclure, par la règle d'inférence du *modus ponens*, que la proposition B doit également être vraie. On dira que B est une *conséquence logique* de 2 et 3. Plus formellement :

$$A \Rightarrow B,\ A \vdash B$$

La clé pour comprendre le raisonnement automatique est d'observer que le fait que la proposition B soit vraie était déjà connu implicitement dès que l'on a énoncé 2 et 3 et à la condition que l'on accepte le *modus ponens* comme une règle de raisonnement valide. Vu sous cet angle, le raisonnement est un processus visant à expliciter des caractéristiques d'un domaine, déjà présentes implicitement dans la description de ce dernier.

La formalisation d'un langage permettant de représenter la connaissance et de faire des inférences s'appelle une logique. La structuration de l'ensemble de notre connaissance d'un domaine s'appelle la représentation des connaissances. L'exécution par un ordinateur d'une procédure pour extraire de l'information contenue implicitement dans notre connaissance du domaine s'appelle du raisonnement automatique.

On peut caractériser une logique donnée par :

- Ce que l'on décrit.
- Comment ces descriptions sont réalisées de façon formelle.
- Le type de connaissance pouvant être explicitée.
- Les méthodes de preuve pouvant être utilisées.

La prochaine section, ainsi que les annexes II et III, analyseront les logiques de description selon ces caractéristiques.

3.4.2 Les logiques de description

Les logiques de description sont une famille de formalismes de représentation des connaissances qui décrivent un domaine d'application en définissant les concepts du domaine (sa terminologie) et en utilisant ces concepts pour spécifier les propriétés des objets faisant partie du domaine (sa description du monde). La particularité de ce formalisme est qu'il constitue une logique, telle que décrite à la section précédente, et qu'il permet ainsi de faire des inférences pour expliciter des conclusions à partir de la description du domaine.

Les logiques de description ont été développées pour être utilisables dans des situations pratiques. Les opérateurs logiques permis sont donc soigneusement choisis pour s'assurer que les méthodes de preuve associées donnent des résultats dans un temps raisonnable pour des problèmes typiques. De plus, un grand soin a été apporté à l'optimisation des services de raisonnement. Différentes logiques de description ont été développées pour obtenir des compromis différents entre l'expressivité du langage et l'efficacité

des services de raisonnement. Les logiques de description sont des sous-ensembles de la logique des prédicats du premier ordre auquel on a ajouté de façon contrôlée certaines caractéristiques pour tenir compte de situations particulières au mode de représentation.

La représentation des connaissances est décomposée en deux parties. La TBox qui correspond à la définition de la terminologie et la ABox qui comprend les assertions correspondant aux propriétés des objets du domaine. L'union des déclarations de la TBox et de la ABox constituent les axiomes de la logique. Une tâche fondamentale de raisonnement sera de s'assurer, comme dans toute logique, que l'ensemble des axiomes n'est pas contradictoire.

Trois types de symboles sont utilisés (l'alphabet de la logique) :

- Les concepts qui correspondent à des ensembles d'objets du domaine. Les concepts correspondent à des prédicats à une variable de la logique des prédicats du premier ordre. Ainsi si Personne est un concept alors Personne(Marie) sera vrai si Marie est une personne dans le domaine étudié.

- Les rôles qui correspondent à des propriétés des objets du domaine. Il s'agit de prédicats à deux arguments. Ainsi si aEnfant est un rôle alors aEnfant(Marie, Pierre) sera vrai si Pierre est l'enfant de Marie dans le domaine étudié.

- Les objets du domaine qui sont représentés par des constantes (Marie, Pierre).

La terminologie (TBox) est constituée de définition de concepts. Elle décrit de nouveaux concepts à partir de concepts *primitifs* et de concepts déjà définis. Ces définitions utilisent des opérateurs logiques qui caractérisent chacune des logiques de description.

Par exemple, l'opérateur $\sqcap$ représente l'intersection de concepts et permet de définir :

$$\mathsf{Femme} \equiv \mathsf{Humain} \sqcap \mathsf{Femelle}$$

L'opérateur universel de restriction de valeur $\forall R.C$ permet de définir le concept d'un Hsf - un homme avec seulement des fils comme :

$$\mathsf{Hsf} \equiv \mathsf{Homme} \sqcap \forall \mathsf{aEnfant}.\mathsf{Homme}$$

L'opérateur universel de restriction de valeur $\forall R.C$ est une forme restreinte de l'opérateur universel de la logique des prédicats du premier ordre. Cette restriction permet une bonne expressivité tout en simplifiant beaucoup la méthode de preuve. De la même façon on a un opérateur de négation $\neg$, un opérateur existentiel de restriction de valeur $\exists R.C$, un opérateur d'union $\sqcup$, un opérateur de décompte de valeurs sur un rôle $(> nR.C)$ (ex. femme ayant plus de 3 enfants), etc. L'utilisation de définitions formelles est caractéristique des logiques de description et elle permet de conclure l'appartenance d'un individu à une classe dès que le membre de droite de la définition est satisfait.

Les assertions (ABox) sont des restrictions sur les valeurs des prédicats qui correspondent à spécifier les propriétés d'un objet du domaine.

Les connaissances pouvant être explicitées d'une logique de description avec une base de connaissances (ensemble d'axiomes) concernent à la fois la terminologie et la description du monde.

Du côté de la terminologie on a :

- Un concept donné peut-il être satisfait (satisfiabilité) ?
- Un concept donné est-il plus général qu'un autre concept (subsomption) ?
- Deux concepts sont-ils équivalents ?
- Deux concepts sont-ils disjoints ?

De plus, un service de raisonnement de base consiste à répéter le test de subsomption de façon efficace pour construire un treillis de subsomption pour l'ensemble des concepts de la terminologie. On obtient un treillis au sens mathématique en ajoutant un concept vide et un concept correspondant à l'ensemble du domaine.

Du côté de la description du monde (ABox) :

- Non-contradiction des axiomes.
- Appartenance d'un individu à une classe.

3.4.3 Correspondance avec les applications d'entreprise

Malgré notre présentation jusque-là très sommaire des logiques de description, nous pouvons déjà faire le lien avec les applications d'entreprise.

Pour appliquer les logiques de description, il nous faut d'abord des définitions formelles des classes pour constituer le TBox. Les classes à décrire sont les classes de dernier niveau (feuilles) de chacune des hiérarchies fonctionnelles.

Ces définitions sont basées sur les valeurs de propriétés des objets du domaine, qui correspondent à des rôles dans le langage des logiques de description. Elles utilisent également l'appartenance d'objets à des classes. Ces définitions sont fixes.

On doit ensuite obtenir la description du domaine pour constituer la ABox. Les assertions du ABox sont construites à partir de l'état de la base de données qui donne l'appartenance connue de certains objets à des classes (ici les tables) et la valeur des propriétés des objets.

La combinaison de nos définitions de classes et des assertions extraites de la base de données donnent une série d'axiomes constituant une logique. Nous pouvons ensuite extraire de l'information implicite de cette logique, ici l'appartenance des objets dans les différentes classes des hiérarchies fonctionnelles. L'outil pour effectuer le raisonnement automatique est appelé un *reasoner* dans le cas des logiques de description.

***

À partir d'ici le lecteur peut se satisfaire, pour l'instant, de notre explication du fonctionnement des logiques de description et procéder au chapitre suivant qui décrit le patron d'architecture permettant d'intégrer les outils vus jusqu'à maintenant.

Le lecteur peut également aller à l'annexe II pour apprendre les bases du langage OWL [1] qui sera utilisé pour codifier les définitions de classe.

L'autre possibilité est l'annexe III qui continue à décrire le fonctionnement d'un *reasoner* de façon beaucoup plus détaillée.

[1] La version de OWL utilisée dans le mémoire est celle basée sur les logiques de description, ils s'agit de OWL-DL pour selon la classification de la première version de OWL.

CHAPITRE 4

PRÉSENTATION DU PATRON D'ARCHITECTURE

Dans ce chapitre, nous utilisons les concepts présentés dans les trois premiers chapitres, ainsi que dans les annexes II et III, pour montrer l'utilité des logiques de description face au problème de l'utilisation simultanée de plusieurs hiérarchies fonctionnelles dans une application orienté-objet.

L'objectif est de développer un patron d'architecture, permettant d'approcher le problème de façon systématique. Comme vu précédemment, un patron d'architecture est une façon d'organiser et d'utiliser des composantes dans le but d'atteindre un objectif précis. Ici les composantes sont les hiérarchies de classes, les processus de création d'objets, la structure des tables de la base de données relationnelle, les modèles des ORM, la définition des classes en logiques de description, etc.

Nous commençons par introduire le modèle de classe qui sera utilisé dans l'application pour permettre l'utilisation simultanée de hiérarchies fonctionnelles multiples sans avoir recours à des classes d'intersection et à l'héritage multiple. Nous utilisons des objets satellites pour remplacer les classes d'intersection.

Nous verrons ensuite comment générer automatiquement les objets de ce modèle en utilisant un ORM et un *reasoner*. Nous procèderons en trois étapes :

1. Comment synchroniser le cycle de vie des objets satellites avec les objets de base.
2. Comment créer dynamiquement les objets satellites une fois que l'on connaît l'appartenance d'un objet à une classe dans chacune des hiérarchies fonctionnelles.
3. Comment déterminer la classe d'appartenance d'un objet dans les hiérarchies.

Nous résumerons ensuite le processus décrit sous forme d'étapes plus formelles qui constitueront le corps du patron d'architecture - classification en lots. Finalement, nous présenterons quelques considérations méthodologiques pour l'utilisation concrète du patron d'architecture.

Comme nous travaillons avec des composantes couvrant plusieurs technologies, ce chapitre se limite à une la présentation du principe du patron d'architecture et de son or-

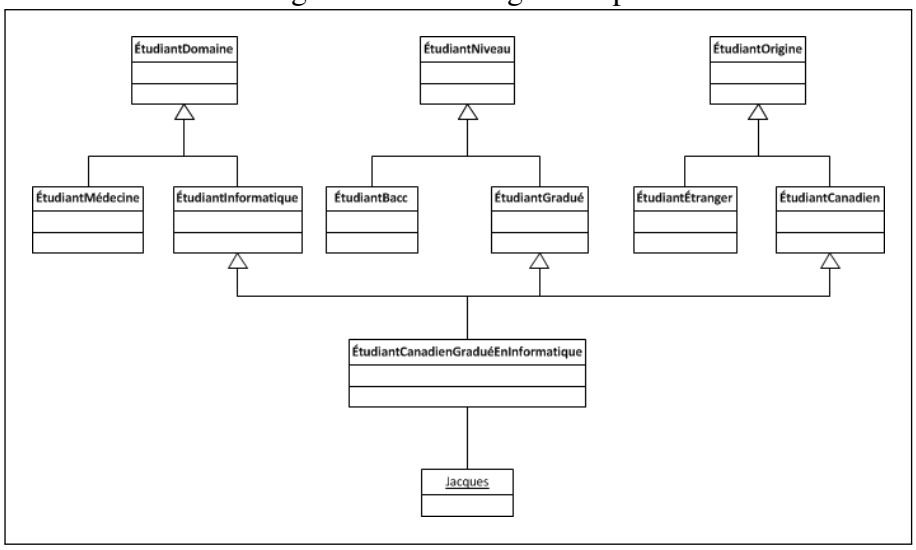
ganisation pour alléger le propos. Le chapitre suivant contiendra une analyse de plusieurs aspects du patron d'architecture telles que ses conditions d'utilisation et ses généralisations.

Notons également que les annexes IV et V décrivent le prototype que nous avons développé dans le cadre de ce projet de mémoire pour illustrer la possibilité d'utiliser ce patron avec des outils actuellement disponibles.

4.1 Objets satellites

Comme nous l'avons vu à la section 2.2.6, nous tentons de mitiger le problème des hiérarchies fonctionnelles multiples dans le paradigme orienté-objet.

Figure 4.1 – Héritage multiple



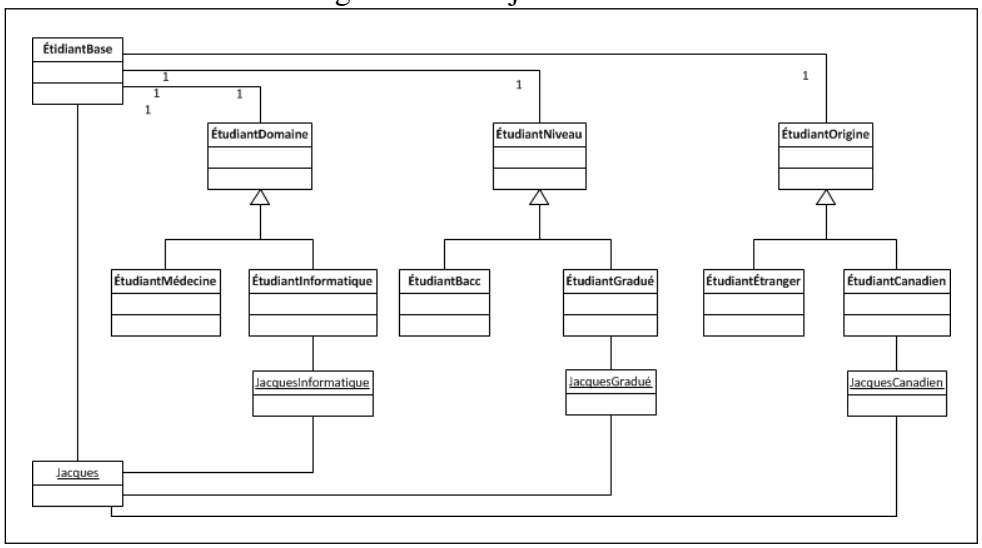
La figure 4.1 rappelle le problème, qui consiste à tenter d'éviter d'avoir à définir une classe d'intersection pour chaque combinaison de fonctionnalités dans les hiérarchies. Ici l'objet Jacques est une instance de la classe d'intersection EtudiantCanadienGradueEnInformatique ; ce qui lui permet d'acquérir des fonctionnalités spécifiques dans chacune des 3 hiérarchies.

À la section 3.2, nous avons vu trois approches visant à mitiger ce problème. Nous présentons ici une quatrième idée qui apparaît plus complexe à prime abord mais que

nous simplifierons par la suite en utilisant un ORM et un *reasoner* de logiques de description.

La figure 4.2 présente cette nouvelle idée. On définit une classe qui contiendra les propriétés et les méthodes qui ne dépendent pas des hiérarchies fonctionnelles. Les objets sont créés comme des instances de cette *classe de base*. Dans notre exemple Jacques est une instance de ÉtudiantBase. Ici le nom de l'étudiant serait une propriété de ÉtudiantBase.

Figure 4.2 – Objets satellites



La classe de base est ensuite mise en relation 1 à 1 avec les classes à la racine de chacune des hiérarchies fonctionnelles. Cela a pour effet de créer des propriétés référentielles vers chacune des hiérarchies pour les instances de la classe. Ainsi Jacques aura une propriété de type ÉtudiantDomaine, que nous appellerons Domaine, une propriété de type ÉtudiantNiveau, que nous appellerons Niveau et ainsi de suite.

On supposant que l'on connaisse la classe feuille à laquelle l'objet appartient dans chacune des hiérarchies, on pourra créer des objets dans chacune de ces classes et les affecter aux propriétés correspondantes. Comme Jacques est un étudiant canadien on pourra créer l'objet JacquesCanadien comme instance de la classe ÉtudiantCanadien et l'affecter à la propriété Origine de Jacques. Jacques obtiendra donc un comportement polymorphe en regard de la hiérarchie ÉtudiantOrigine à travers sa propriété Origine. En

reprenant l'exemple de la section 3.2.2 on obtient :

```
public class ÉtudiantOrigine
{
    public virtual decimal ObtenirFrais()
    { return 0; }
}

public class ÉtudiantCanadien : ÉtudiantOrigine
{
    public override decimal ObtenirFrais()
    { return 1; }
}

public class ÉtudiantBase
{
    public ÉtudiantOrigine Origine;
}

Etudiant Jacques = new EtudiantBase();
Jacques.Origine = new EtudiantCanadien();
decimal frais = Jacques.Origine.ObtenirFrais();
```

Et c'est la version de ObtenirFrais propre à la classe ÉtudiantCanadien (qui retourne la valeur 1) qui sera appelée même si elle est accédée à travers la propriété Origine qui est de type ÉtudiantOrigine et non de type ÉtudiantCanadien.

On obtient ainsi un comportement équivalent à la définition de la classe d'intersection EtudiantCanadienGradueEnInformatique à l'exception du fait que la fonctionnalité propre à chacune des hiérarchies est accédée à travers une propriété (dans notre exemple la propriété Origine) au lieu d'être directement des propriétés et méthodes de l'objet. Cette dernière contrainte ne cause pas problème.

Nous appelons *objets satellites* les objets qui font le lien entre les objets de base et les hiérarchies fonctionnelles (JacquesCanadien dans l'exemple). L'avantage d'utiliser ces objets satellites est évidemment d'éliminer le recours aux classes d'intersection, ce qui est l'objectif visé.

Jusqu'à maintenant nous ne faisons qu'utiliser les concepts de l'orienté-objet de façon créative. La prochaine étape, en restant dans l'orienté-objet, serait d'isoler la création des objets dans un *object factory* (littéralement une usine d'objets) et on obtiendrait du

code simultanément polymorphe sur plusieurs hiérarchies fonctionnelles dans le reste de l'application.

Nous voulons maintenant aller plus loin et automatiser la création des objets dans l'approche que nous venons d'énoncer. Pour y arriver nous devons :

1. Pouvoir synchroniser le cycle de vie de l'objet de base et de ses objets satellites.
2. Savoir dans quelle classe créer l'objet satellite dans chacune des hiérarchies fonctionnelles.
3. Pouvoir créer dynamiquement un objet dans la bonne classe d'une hiérarchie une fois que l'on connaît cette classe cible.

Dans les sections suivantes, nous présenterons comment utiliser un ORM pour remplir les objectifs 1 et 3 (synchronisation et création dynamique d'objets). Puis à la section 4.4 nous présenterons comment utiliser un *reasoner* pour déterminer à quelle classe appartient l'objet dans chacune des hiérarchies.

4.2 Cycle de vie des objets satellites

La section 3.3 a décrit le fonctionnement général d'un ORM et certains *mappings* particuliers entre le modèle objet et le modèle de la base de données.

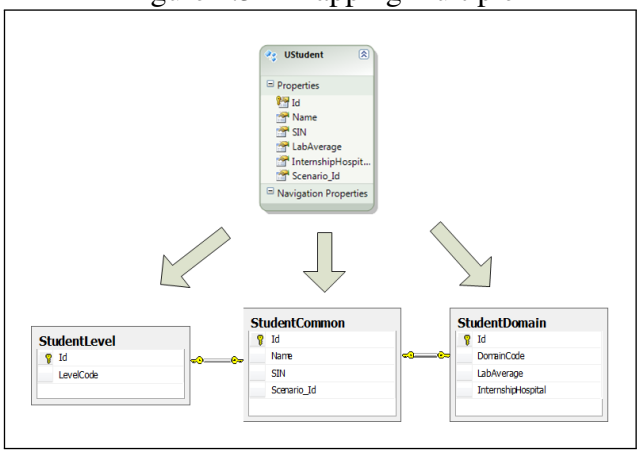
Assumons d'abord que la création des objets et leur utilisation soient effectués à des moments différents. Au prochain chapitre, nous poserons cette hypothèse comme une condition d'utilisation du patron d'architecture.

On peut utiliser cette contrainte pour amener notre première idée : utiliser des modèles objets et des *mappings* différents pour les étapes d'écriture et de lecture des objets dans la base de données (le modèle de la base de données est évidemment le même). La présente section couvre la phase d'écriture et la prochaine, la phase de lecture et d'utilisation.

Pour l'écriture, on utilisera un *mapping* multiple tel que décrit à la figure 3.3 que nous reproduisons ici (figure 4.3).

Nous devons donc prendre le concept que l'on veut utiliser dans plusieurs hiérarchies fonctionnelles (ici les étudiants) et créer un modèle de données avec une table principale

Figure 4.3 – Mapping multiple



contenant les propriétés communes à toutes les hiérarchies (ici Id, Name et SIN). On crée ensuite une table satellite pour chacune des hiérarchies dans lesquelles on doit classifier notre concept (ici domaine et niveaux d'étude).

Chaque table satellite inclura le même identifiant que la table principale, les champs qui sont spécifiques à la hiérarchie considérée (ex. LabAverage pour la table StudentDomain qui n'est pertinente que pour les étudiants de science) ainsi qu'un champ de contrôle permettant de déterminer à quelle classe de la hiérarchie appartient l'objet (ex. DomainCode pour la hiérarchie des domaines d'étude).

Lors de l'écriture le ORM s'assurera que les lignes soient créées dans toutes les tables et les champs pertinents mis à jour dans la bonne table. Cette opération de synchronisation est transparente pour l'application qui se contente de créer des instances de la classe du modèle conceptuel et demande au ORM de les sauvegarder.

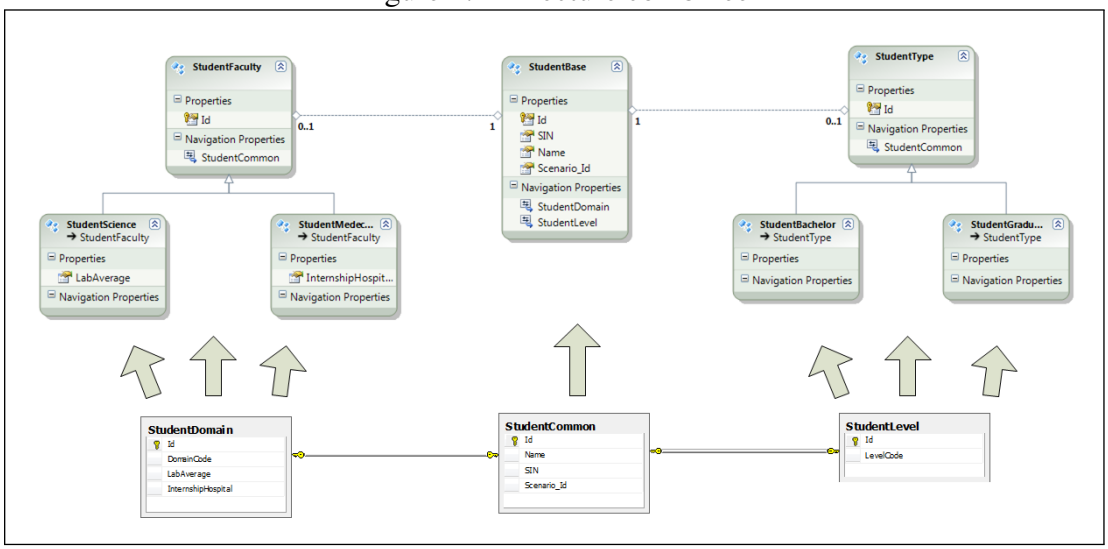
Ainsi la synchronisation de la création des objets se fait à travers la couche de persistance à l'aide d'un ORM.

Entre l'opération d'écriture et celle de lecture, nous assumons qu'un processus quelconque viendra peupler les champs de contrôle des tables secondaires. Nous verrons plus loin (4.4) que c'est ici que s'intégrera le *reasoner* de logiques de description mais un module codé en dur pourrait tout aussi bien être utilisé.

4.3 Création dynamique des objets

L'opération de lecture utilisera un modèle objet différent (voir figure 4.4).

Figure 4.4 – Lecture combinée



Ici la table principale correspondra à une classe (ici StudentBase) qui sera interrogée pour obtenir les étudiants désirés. La classe principale inclura des propriétés ayant le type des objets racine de chacune des hiérarchies (par exemple StudentDomain qui est de type StudentFaculty). En accédant à la propriété correspondant à une des hiérarchies fonctionnelles, le ORM va lire la ligne de la table satellite correspondante et créer un objet de la classe de la hiérarchie correspondant à la valeur du champ de contrôle.

Un appel à une propriété de l'objet racine de la hiérarchie sera traité de façon polymorphe (c'est la version de la classe réellement créée qui sera appelée à travers une référence à un objet à la base de la hiérarchie).

Puisque l'on peut ainsi *pointer* sur autant de hiérarchies que l'on veut, on a créé un objet étudiant ayant un comportement polymorphe sur des hiérarchies fonctionnelles multiples ; ce qui est l'objectif recherché par notre patron d'architecture. Il reste seulement à peupler les champs de contrôle des tables satellites.

4.4 Classification par logiques de description

Comme nous pouvons maintenant coordonner le cycle de vie des objets satellites et que nous pouvons les créer dans les bonnes classes une fois celles-ci identifiées, il ne nous reste qu'à déterminer la classe d'appartenance d'un objet dans les hiérarchies fonctionnelles. Nous voulons évidemment réaliser la classification en question à l'aide d'un *reasoner* de logiques de description.

Nous présentons d'abord les étapes du processus à haut niveau. Nous voyons ensuite quelques considérations et exemples permettant de mieux comprendre comment le cycle pourrait être réalisé dans la pratique.

4.4.1 Étapes

Décrivons d'abord succinctement les étapes du processus :

1. Définir les classes et lire l'ontologie OWL résultante : Pour chacune des ontologies, il faut définir les classes formant la partition des classes feuilles de la hiérarchie. On pourra avoir à déclarer certains objets utilisés comme cibles de propriétés-référentielles. Il faut utiliser un langage qui pourra être lu et envoyé au *reasoner*.
2. Traduire les objets et la valeur de chacune de leurs propriétés pertinentes : On doit utiliser le même langage que les définitions de classes, les identifiants des objets référencés doivent être les mêmes et il faut éviter de créer des incohérences.
3. Combiner les axiomes précédents pour former une ontologie OWL : La combinaison des axiomes de différentes origines est triviale à cause de l'hypothèse du monde ouvert (voir section suivante). Il faut simplement éviter de créer une incohérence dans la logique correspondant à l'ontologie.
4. Effectuer la classification : Par classification, on entend la création du treillis de subsomption de l'ontologie et l'affectation de chaque objet à son MSC - *MostSpecificConcept*. La classification est effectuée par un *reasoner*. On doit donc avoir accès à un *reasoner* à partir de l'application d'entreprise.
5. Retourner le résultat à l'application : Le résultat désiré pour chaque objet est la

classe à laquelle il appartient dans chacune des hiérarchies. On construira ce résultat à partir de la liste des membres de chaque classe de chacune des hiérarchies. On devra donc pouvoir demander au *reasoner* la liste des membres d'une classe de l'ontologie combinée construite à l'étape 3.

4.4.2 Considérations

4.4.2.1 Le langage du *reasoner* est OWL

Jusqu'à maintenant on a parlé de logiques de description sans parler de comment on exprimait de telles logiques.

Pour les articles techniques, une notation s'est graduellement formée à travers les différents articles de recherche traitant du sujet. Nous utilisons cette notation à l'annexe III pour illustrer le fonctionnement d'un *reasoner*.

Pour les applications, l'évolution progressive des langages a mené à la recommandation du W3C de 2004 [20] qui définit le langage OWL - *Web Ontology Language* pour la définition des ontologies dans le contexte du web sémantique. Ce langage : *a*) correspond à différentes logiques de description en fonction des constructions utilisées ; *b*) est supporté par tous les *reasoners* ; *c*) permet une certaine interopérabilité à travers le protocole OWL-Link ; et *d*) s'exprime en plusieurs syntaxes dont la syntaxe de Manchester que l'on utilisera dans nos exemples.

L'annexe II donne plus de détail sur le langage OWL.

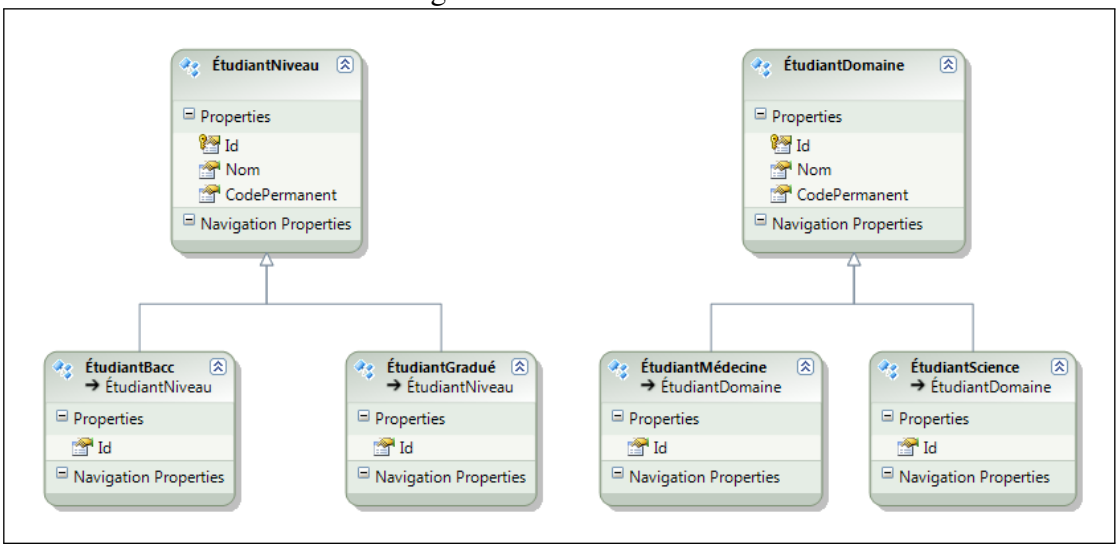
4.4.2.2 Que doit-on définir au juste ?

Dans une hiérarchie fonctionnelle, seul le dernier niveau est intéressant car l'appartenance d'un objet à une classe d'un niveau intermédiaire peut être déduite de son appartenance au dernier niveau. Par exemple, prenons la classe ÉtudiantMaitrise dérivée de la classe ÉtudiantGradué, elle-même dérivée de la classe Étudiant. Si un objet est une instance de la classe ÉtudiantMaitrise, il sera automatiquement une instance de ses deux classes de base. On n'a pas besoin d'expliciter cette information.

Nous appellerons *classes feuilles*, les classes au dernier niveau de la hiérarchie. Nous

appellerons la *partition* [1] d'une hiérarchie, l'ensemble de ses classes feuilles. Il s'agit bien d'une partition au sens mathématique : un objet doit être membre d'une et une seule classe de la partition. Pour les fins de la discussion, on supposera deux hiérarchies fonctionnelles simples (figure 4.5)

Figure 4.5 – Hiérarchies



Ici ce sont les classes ÉtudiantBacc et ÉtudiantGradué qu'il faut définir pour la hiérarchie des niveaux et les classes ÉtudiantMédecine et ÉtudiantScience pour la hiérarchie des domaines d'étude.

On doit définir les classes en spécifiant des contraintes sur les propriétés des objets. Ainsi la classe ÉtudiantGradué est définie en OWL Manchester comme :

```
Class : sc:ÉtudiantGradué
    SubClassOf : sc:ÉtudiantNiveau
    EquivalentTo :
        sc:estInscrit (some sc:ProgrammeMaitrise) or
                    (some sc:ProgrammeDoctorat)
```

On peut aussi avoir à ajouter des axiomes pour préciser les contraintes utilisées dans les définitions de classes.

[1] L'utilisation du terme *partition* dans ce contexte est un abus de langage. Comme chaque objet doit faire partie d'une classe de la hiérarchie et que seules les classes feuilles ne sont pas abstraites alors les classes feuilles, vues comme des ensembles d'objets, forment une partition de l'ensemble des objets.

```
ObjectProperty: sc:estInscrit
Class : sc:ProgrammeMaitrise
Individual : sc:MaitriseInformatique
    Types : sc:ProgrammeMaitrise
```

4.4.2.3 Exemple simple d'extraction d'objets

En continuant l'exemple du paragraphe précédent, l'application pourrait avoir une table d'étudiants et une table de programmes.

En supposant que l'étudiant avec le code permanent *SIN101* soit inscrit au programme de maîtrise en informatique, il faudrait extraire cette information de la base de données et produire les axiomes correspondant à :

```
Individual : sc:SIN101
    Facts: sc:estInscrit sc:MaitriseInformatique
```

4.4.2.4 Exemple simple de résultat

En continuant notre exemple, si on demande au *reasoner* la liste des étudiants membres de la classe ÉtudiantGradué alors SIN101 fera partie de la liste.

4.4.2.5 Hypothèse du monde ouvert

La combinaison d'ontologies en OWL ne pose pas de problème particulier car elles ne se comportent pas comme des bases de données.

Supposons que deux ontologies peuvent définir des caractéristiques des mêmes entités. Pour prendre un exemple simple, une ontologie pourrait inclure un axiome estInscrit (Jacques,MajeureMaths) et une deuxième ontologie pourrait inclure l'axiome estInscrit (Jacques,MaitriseInfo). Une interprétation traditionnelle, correspondant à ce que nous appellerons « l'hypothèse du monde fermé », conclurait que Jacques n'est pas un étudiant gradué si on ne fournissait que la première ontologie. En ajoutant la seconde ontologie, la conclusion serait différente et Jacques deviendrait un étudiant gradué. C'est l'interprétation standard utilisée avec les bases de données relationnelles. Une information qui n'est pas dans la base de données est réputée ne pas exister. Combiner des bases

de données peut donc contredire nos conclusions précédentes par rapport au monde. Ce comportement est très problématique si on a besoin de combiner des ontologies.

Une des grandes différences entre les bases de données et les logiques de description est que ces dernières utilisent plutôt « l'hypothèse du monde ouvert » c'est-à-dire qu'on considère que ce qui n'est pas dans l'ontologie est inconnu et non simplement faux. Par exemple, un *reasoner* ne va pas conclure que Jacques n'est pas un étudiant gradué à partir de la première ontologie car il y a toujours la possibilité que Jacques soit inscrit à un programme gradué mais que cette information soit inconnue. Selon cette hypothèse, la combinaison des ontologies nous permet maintenant de conclure que Jacques est un étudiant gradué mais ne contredit pas les conclusions déduites à partir de la première ontologie.

L'hypothèse du monde ouvert est cohérente avec le fonctionnement des logiques de description et est une des principales raisons pourquoi celles-ci ont été choisies comme support de raisonnement pour le web sémantique. Dans un contexte de web sémantique c'est le web en entier qui est la base de données et on ne peut assumer que l'on connaît l'ensemble des données qui s'y trouvent.

4.4.2.6 Possibilité d'incohérence

Si la combinaison d'ontologies ne cause pas de problème, il faut toutefois s'assurer de ne pas introduire d'incohérence. Une incohérence viendrait d'une contradiction dans les axiomes. Si on affirmait qu'un objet est et n'est pas membre d'une classe, par exemple, on empêcherait tout le processus de raisonnement de s'effectuer, jusqu'à ce que l'incohérence soit résolue.

4.5 Classification en lots

Le patron d'architecture « classification en lots » est la combinaison des étapes décrites précédemment pour *a*) synchroniser le cycle de vie des objets satellites ; puis *b*) déterminer l'appartenance des objets dans les hiérarchies ; puis *c*) instancer les objets dans les bonnes classes.

Les principales étapes sont :

1. Écriture dans la base de données des objets à classifier : Utilise le modèle d'écriture du ORM et précise les propriétés requises pour la classification.
2. Convertir les objets existants en assertions OWL : Utilise le modèle d'écriture du ORM (en mode lecture).
3. Lire la définition des hiérarchies cibles : Les définitions ont été élaborées selon la procédure de la section 4.4 et sont disponibles en OWL.
4. Combiner les deux ontologies : Assurer l'équivalence des identificateurs produits aux deux sections précédentes et éviter les incohérences. La combinaison en tant que telle ne cause pas de problème, à cause de l'hypothèse du monde ouvert.
5. Effectuer la classification à l'aide du *reasoner* : Cette étape exige de pouvoir accéder à un *reasoner*.
6. Initialiser les champs de contrôle de la Base de données : Utilise le modèle de mise à jour des types du ORM. Effectué à partir du résultat de la classification.
7. Lecture des objets classifiés : Lecture avec le modèle hiérarchique du ORM permet une utilisation polymorphe.

Si les propriétés des objets utilisées dans la classification sont modifiées ou si de nouveaux objets sont créés, il faut recommencer le cycle pour obtenir une nouvelle classification et relire les objets pour avoir les résultats dans l'application.

4.6 Méthodologie

Le tableau 4.I donne les principales étapes de la méthodologie à suivre pour réaliser le module de classification d'une application.

Seules les étapes 4 et 8 demandent une expertise de OWL et des logiques de description ce qui donne un impact minimal sur le processus de développement.

Tableau 4.I – Étapes de la méthodologie

	Étape	Description
1	Vérifier les conditions d'utilisation.	Réaliser une analyse à haut niveau du problème pour s'assurer que l'approche soit utile et réalisable.
2	Élaborer les hiérarchies fonctionnelles.	Les hiérarchies : *a*) doivent être construites à partir des requis de l'application ; et *b*) doivent identifier clairement la partition de chaque hiérarchie.
3	Identifier les propriétés nécessaires pour définir les partitions.	Il s'agit aussi bien de propriétés scalaires (ex. Mesure) que de liens entre objets (ex. estInscrit).
4	Ontologie OWL.	Il faut valider les propriétés identifiées en élaborant l'ontologie et en réalisant des tests de classification pour validation.
5	Intégrer la saisie des propriétés dans la liste de requis.	On doit s'assurer que les propriétés requises pour effectuer la classification peuvent être intégrés dans les requis de l'application sans remettre en cause la faisabilité ou la rentabilité.
6	Modèle de la base de données.	
7	Élaboration des trois modèles ORM.	Il faut bien s'assurer que les requis de l'application soient satisfaits et que les assertions peuvent être extraites.
8	Annotation des modèles pour extraction.	On assume que l'extraction des valeurs des propriétés et leur mise à jour se feront par un module générique utilisant des annotations aux modèles des ORM comme entrée.
9	Développer l'application.	Reste à créer des entités et utiliser les hiérarchies fonctionnelles dans l'application.

CHAPITRE 5

ANALYSE DU PATRON D'ARCHITECTURE

Nous présentons quatre aspects laissés de côté jusqu'à maintenant pour pouvoir présenter le patron de base le plus rapidement possible.

On verra : *a*) les conditions d'utilisation du patron d'architecture ; *b*) pourquoi avoir utilisé une série d'environnements et d'outils existants plutôt qu'une approche plus radicale basée uniquement sur les logiques de description ; *c*) l'explication de certaines des caractéristiques fondamentales du patron de base ; *d*) comment on pourrait restreindre certaines conditions d'utilisation du patron de base ; et *e*) quels scénarios techniques pourraient être envisagés si on ne requiert pas un retour du résultat de la classification dans l'application.

5.1 Conditions d'utilisation

Que ce soit pour un patron de conception ou un patron d'architecture, la première étape est toujours de préciser dans quelles conditions le scénario sera applicable. Ici nous avons identifié des conditions générales applicables dès que l'on introduit le *reasoner* de logiques de description. Nous avons également des conditions particulières au patron « classification en lots ».

5.1.1 Conditions générales

Les conditions d'utilisation générales pour introduire un *reasoner* comme solution à un problème de classification dans une application d'entreprise sont :

1. Application d'entreprise : Application complexe tant au niveau fonctionnel que technique.
2. Hiérarchies fonctionnelles complexes : La tâche de classification doit être complexe. Inutile si quelques énoncés conditionnels peuvent faire le travail.
3. Accès simultané à plusieurs hiérarchies : La tâche devient difficile sans utiliser une

usine d'objets (*Object Factory*) complexe. Le ORM (au minimum) et le *reasoner* deviennent attrayants.

4. Bonne connaissance OWL : Modélisation logique très particulière par rapport au modèle relationnel.

5.1.2 Conditions particulières

Les conditions particulières attachées au patron « classification en lots » sont :

1. Création et utilisation séquentielle : Il faut valider que l'on peut procéder en trois étapes :
   (a) Création d'un lot d'objets
   (b) Classification
   (c) Utilisation des hiérarchies fonctionnelles
2. Cycles peu fréquents : À chaque cycle on doit refaire la classification pour l'ensemble des objets. Si cycles fréquents, beaucoup d'objets au total et peu de changements à chaque cycle alors grosse perte d'énergie.

L'idée est de figer les valeurs des propriétés des objets pendant la classification. En plus de simplifier l'implantation, cette contrainte ramène le problème à l'utilisation principale autour de laquelle les *reasoner* ont évolué, soit le développement des ontologies. On reste donc dans les services les plus matures des *reasoners*.

5.2 Pourquoi les outils existants ?

Depuis le début du mémoire, nous assumons que nous utilisons une série d'outils existants (Base de données relationnelles, ORM, langages orienté-objet, *reasoner*). Cette hypothèse nous a amené à développer un patron d'architecture, qui n'est rien d'autre qu'une façon d'utiliser des outils existants pour résoudre un type de problème bien déterminé. Mais pourquoi conserver tous ces outils ? Ne serait-il pas plus simple d'aborder le problème d'une façon complètement nouvelle ? Cette section répondra à ces questions.

5.2.1 Pourquoi garder le paradigme orienté-objet

La plupart des articles et outils traitant du développement d'applications en utilisant des logiques de description supposent que la modélisation des données est faite dans le cadre du paradigme du web sémantique (réseau d'objets sous forme de graphe). Nous qualifierons les applications développées selon cette approche d'applications ontologiques.

Nous supposons quant à nous, que bien que certaines applications particulières puissent être envisagées comme des applications ontologiques, le paradigme de base ne sera pas modifié à court terme pour la plupart des applications d'entreprise.

Nous appuyons notre hypothèse sur plusieurs facteurs :

- Le manque de maturité des outils liés aux logiques de description dans le contexte d'applications transactionnelles. On peut évaluer cette maturité par rapport à la liste de requis non-fonctionnels caractérisant les applications d'entreprise. Par exemple, une *proposition* de protocole existe pour accéder à des *reasoners* à partir de différentes plateformes (OWL-Link). Il s'agit donc d'une proposition visant l'interopérabilité. Cependant, cette proposition n'utilise pas l'infrastructure des Services Web XML (SOAP et standards WS*) comme transport. Elle ne remplit donc aucun des requis non-fonctionnels typiques d'une application d'entreprise.

- Remplacer le paradigme de base aurait pour effet de perdre certains avantages de l'approche orienté-objet. Par exemple, le fait de lier le concept d'objet à celui d'instance d'une classe a pour effet de déterminer les caractéristiques et comportements applicables à un objet et donc, à le rendre plus prévisible. Un système composé de ce type d'objets sera plus facilement *testable* et donc robuste.

- L'inertie des systèmes en termes d'investissements en outils, plateformes et ressources humaines est bien trop grande pour penser à un changement radical à court ou moyen terme.

Nous sommes convaincus que la plupart des applications d'entreprise continueront à suivre l'approche orienté-objet pour encore longtemps.

Nous nous restreignons donc à des approches pouvant être utilisées en complément du modèle orienté-objet et non en remplacement de ce dernier.

Nous cherchons à utiliser les avantages de logiques de description mais en ayant le moins d'impact possible sur les ressources humaines et les processus de développement.

On peut faire un parallèle avec les bases de données relationnelles qui reposent sur un modèle théorique différent de l'approche orienté-objet et qui demandent l'intervention de ressources spécialisées pour l'intégration et l'optimisation des bases de données complexes. Cependant, des méthodes et outils ont été développés (ex. : les ORM) pour rendre compatibles l'utilisation des bases de données relationnelles avec l'approche orienté-objet sans que les développeurs n'aient à comprendre la théorie du modèle relationnel ou soient des experts en configuration de bases de données.

C'est donc ce dernier type d'approche et d'outils que nous visons à développer dans le cadre de ce mémoire.

5.2.2 Pourquoi utiliser un *reasoner* existant ?

Le principal outil nécessaire pour exploiter les logiques de description est le *reasoner* qui sert à réaliser la classification des classes et des individus.

Des *reasoners* de ce type existent depuis la fin des années 1990 et ont été utilisés pour plusieurs projets importants liés à la réalisation d'ontologies et au web sémantique. Leur principale force est donc leur relative maturité et leur utilisation passée dans des projets réels dans ces contextes d'application. Les algorithmes utilisés ainsi que leurs implantations sont matures. Les déductions réalisées par les principaux *reasoners* sont donc correctes.

Plusieurs problèmes se posent cependant dans le cadre d'une utilisation dans les applications d'entreprise. Nous en retiendrons quatre principaux pour la suite de la discussion :

1. La couverture fonctionnelle des principaux outils n'est pas uniforme. Par exemple, les fonctionnalités du protocole OWL-Link, que nous utiliserons dans le développement du prototype, ne sont pas couvertes de la même manière pour chacun des

principaux *reasoners*.

2. Les *reasoners* sont très sensibles aux axiomes du domaine car toute contradiction dans l'ensemble des axiomes empêchera de réaliser la classification. Dans ce sens, la vérification de la cohérence des axiomes est un service de base du *reasoner*. Si on a une incohérence, il faut diagnostiquer le problème et apporter des correctifs. Ce genre d'interruption ne peut avoir lieu dans le cadre d'une application d'entreprise d'où la nécessité de pouvoir garantir la cohérence des axiomes.

3. Les *reasoners* ont été développés pour effectuer rapidement l'ensemble des étapes de raisonnement sur une ontologie. Les modifications ne sont typiquement pas fréquentes. Dans le contexte des applications d'entreprise les modifications sont typiquement très fréquentes et on voudrait avoir les résultats du raisonnement mis à jour à chaque modification. Un ajout récent aux *reasoners* permet la modification des axiomes et l'exécution d'une classification sans reconsidérer l'ensemble des axiomes. Le comportement de ces nouvelles fonctionnalités sous un grand volume de transactions reste à être déterminé.

4. Et finalement le problème le plus important, les *reasoners* actuels n'implantent pas le concept de transaction et ses caractéristiques standards (ACID - *Atomicité*, *Concurrence*, *Isolation* et *Durabilité*). Si on exécute en même temps plusieurs MAJ provenant de sources différentes, rien ne nous garantira qu'un raisonnement sera effectué sur un ensemble cohérent d'axiomes. L'absence d'un concept de transaction implique que le raisonnement doit avoir lieu sur une image figée du domaine prise entre deux transactions. Pour obtenir cette image on doit pouvoir sérialiser les transactions lors de leurs applications dans le modèle d'un *reasoner*.

Créer un *reasoner* ayant les caractéristiques requises serait un tout autre projet, en assumant que ce soit possible.

Nous n'avons donc pas tenté, dans le cadre de ce mémoire, de développer un *reasonner* ayant les caractéristiques non-fonctionnelles nécessaires pour supporter directement des applications d'entreprise.

Nous cherchons plutôt à définir une méthodologie pour utiliser les outils existants tout en obtenant les caractéristiques non-fonctionnelles désirées pour le système.

5.2.3 Pourquoi les bases de données relationnelles ?

La section précédente a souligné le manque de gestion des transactions d'un *reasoner*. Cela nous amène donc à garder une base de données relationnelles comme dépôt principal, à moins de faire intervenir une autre technologie. Notons ici que l'utilisation potentielle des bases de données déductives aurait pu être envisagée car elles allient capacité de raisonnement et gestion de transactions. Cela nous aurait toutefois amené sur une toute autre voie.

L'utilisation d'un ORM n'est pas requise pour le développement d'une application d'entreprise. L'alternative consiste à développer une couche d'accès aux données rendant des services essentiellement similaires mais de façon plus efficace dans certains cas. La description du sous-patron « polymorphisme par ORM » (4.2) a cependant montré l'importante simplification amenée par cet outil, d'où son intégration dans notre liste d'outils à la base de l'élaboration de notre patron d'architecture.

5.3 Élaboration du patron de base

La section précédente a expliqué pourquoi nous utilisons des outils existants. Nous avons donc élaboré une recette pour utiliser ces outils pour obtenir le résultat désiré et nous avons appelé cette recette notre patron d'architecture de base. Revenons maintenant sur certaines des caractéristiques intéressantes de ce patron.

5.3.1 Pourquoi interface avec la BD ?

Le processus de classification se connecte à l'application en passant par la base de données et non directement (voir figure IV.1). Une connexion directe est impossible puisqu'en orienté-objet un objet ne peut pas changer de classe. La classe doit être déterminée préalablement à la création de l'objet pour que celui-ci soit créé comme une instance de sa classe. Un objet pouvant être classifié doit donc être détruit puis recréé

dans la bonne classe. Cette recréation des objets lors d'une reclassification découle de la nature de l'orienté-objet, pas du patron d'architecture.

En écrivant le résultat de la classification dans la base de données, recréer l'objet revient à le relire dans une nouvelle classe.

5.3.2 Pourquoi création des objets par le ORM ?

Comme nous l'avons vu à la section 2.2.6 sur les limites des hiérarchies fonctionnelles en orienté-objet nous avons trois problèmes principaux :

- On doit définir une classe pour chaque élément du produit cartésien des feuilles de hiérarchies concernées. Ainsi si on a 10 domaines d'études et 10 origines d'étudiants, on devra définir 100 sous-classes d'intersection (étudiants marocains en mathématiques, étudiants chinois en médecine, etc.).
- On devra codifier la logique pour créer un objet dans la sous-classe précise à laquelle il appartient.
- Les langages les plus utilisés pour le développement des applications d'entreprise (Java, C#) ne supportent pas l'héritage multiple requis pour qu'une sous-classe fasse partie de plusieurs hiérarchies.

Pour traiter ces problèmes avec des logiques de description, on réalise une classification à l'aide d'un *reasoner* en utilisant toutes les hiérarchies fonctionnelles à la fois et en suivant les étapes décrites aux sections précédentes. Lorsqu'il construit son treillis de subsomption le *reasoner* doit assigner chacun des individus à une seule classe, on appelle cette classe le « concept le plus spécifique » de l'individu (*MSC - most specific concept*). Lorsque l'individu appartient à deux classes, le *reasoner* créé une classe d'intersection anonyme et assigne l'individu à cette classe.

Les MSC de l'ensemble des individus constituent la liste des classes d'intersection qui comptent effectivement au moins un individu. Dans un problème type, le nombre de classes ayant au moins un individu sera beaucoup plus petit que la cardinalité du produit cartésien des classes au dernier niveau des différentes hiérarchies. Le produit cartésien donne lieu à une matrice creuse (*sparse matrix*). Ce phénomène est dû au fait que la plupart des catégorisations naturelles concentrent les individus dans un petit nombre de

catégories. Par exemple, si on divise la grandeur des personnes dans un ensemble d'intervalles de un centimètre, la plupart des hommes canadiens se retrouveront dans les catégories autour de la moyenne qui est de 1,760m. À l'opposé La plupart des catégories auront peu d'individus. Peu d'hommes canadiens mesureront 1,2m ou 2,2m. En faisant l'intersection de plusieurs catégorisations on se retrouve donc avec la plupart des intersections vides.

En utilisant le résultat de la classification on pourrait régler les deux premiers problèmes évoqués plus haut. On ne définirait que les classes d'intersection identifiées par le *reasoner* en construisant son treillis. On pourrait assigner les individus aux bonnes classes d'intersection à partir du résultat de la classification (section précédente).

Le dernier problème devrait faire l'objet d'un contournement en utilisant des hiérarchies d'interfaces si l'application est développée en Java ou en C#. La création des classes et l'affectation des individus se faisant automatiquement on produira le même effet que les constructions génériques évoquées à la section 3.2.1. C'est l'infrastructure qui aura la responsabilité de gérer les classes d'intersection.

Le scénario que nous venons de décrire serait une alternative à l'utilisation d'un ORM pour la création des objets. C'est l'approche que nous visions utiliser initialement mais elle est beaucoup plus complexe sur le plan technique que l'utilisation d'un ORM.

5.3.3 Pourquoi classifier l'ensemble des objets ?

Il y a deux problèmes ici :

1. Dans la section 5.2.2 on a vu que les *reasoners* ne traitaient pas les modifications aux axiomes jusqu'à tout récemment.
2. Même si on permet les modifications, nous voulons identifier les classifications qui ont changé. La seule façon simple de le faire serait de refaire sortir la liste des objets membres de chaque classe et de comparer avec le dernier résultat obtenu.

On traite donc tous les objets de toute façon. Nous avons décidé qu'il était plus simple de s'en tenir aux fonctionnalités de base des *reasoners*, qui ont été testées intensivement avec les années avec le traitement des ontologies. À la section 5.4.2 on verra que le traitement de l'impact de la mise à jour d'un nombre restreint d'axiomes constitue une

piste de recherche intéressante.

5.3.4 Comment valider l'ontologie produite ?

Il y a deux problèmes demandant de s'interroger sur la validité de l'ontologie produite par la procédure de classification : la possibilité d'incohérence dans l'ontologie et les classes définies doivent former une partition pour chaque hiérarchie.

On a déjà parlé (4.4) de la sensibilité d'un *reasoner* aux incohérences de l'ontologie. Il y a deux sources d'incohérences potentielles : les assertions des objets et la définition des classes.

On règle les assertions des objets en remettant cette responsabilité à l'application qui doit, de toute façon s'assurer d'avoir une base de données intègre. Si les axiomes sont bâtis à partir d'une base de données intègre, ils seront cohérents. [1]

Pour s'assurer que la définition ne contient pas d'incohérences, on utilise un outil comme Protégé pour le développement. Cet outil peut se connecter à un *reasoner* pour valider la cohérence de l'ontologie.

Le deuxième problème consiste à vérifier que les partitions sont bien définies.

On peut utiliser Protégé pour faire une telle validation en ajoutant des axiomes et en s'assurant que ces nouveaux axiomes n'amènent pas d'incohérence. Par exemple, l'union des classes d'une partition doit représenter l'ensemble universel. Dit autrement, chaque objet sera dans au moins une classe de la partition. En ajoutant un axiome affirmant que l'ensemble universel (owl :Thing) est équivalent à l'union des classes de la partition, on obtiendra une incohérence si la condition n'est pas satisfaite.

De la même façon, les intersections deux à deux des classes doivent être vides. On ajoute donc les axiomes correspondants et on valide la cohérence.

[1] Nos parlons ici du ABox, soit les assertions concernant les objets. Nous devons nous assurer, par exemple, qu'un objet n'a pas deux valeurs pour une propriété si celle-ci est définie comme fonctionnelle dans OWL. Ce genre de contraintes est mise en force par les SGBD relationnels. Le patron d'architecture prévoit le développement d'un module de conversion des relations de la base de donnée relationnelle vers le ABox. Cette opération est effectuée manuellement dans le cadre du prototype présenté en annexe.

5.4 Autres scénarios avec retour direct à l'application

Dans les conditions d'utilisation du patron de base (5.1) nous insistons pour que tout se passe de façon séquentielle : *a*) on crée ou on modifie des objets ; puis *b*) on procède à la classification ; puis *c*) on utilise les hiérarchies fonctionnelles. Cette section analysera la possibilité de réaliser la classification de façon plus synchronisée avec l'application.

5.4.1 Mode synchrone

On voudrait ici qu'un objet soit reclassifié dès qu'il est modifié et que l'on puisse utiliser sa nouvelle position dans les hiérarchies immédiatement (après une relecture, tel qu'il a été indiqué à la section 5.3.1).

Pour arriver à ce résultat une transaction devrait pouvoir incorporer la base de données et le *reasoner*. Puisque le *reasoner* ne supporte pas les transactions, cette approche est impossible.

5.4.2 Mode asynchrone

Le système de journalisation des SGBD permet d'extraire les transactions appliquées par le système une à la suite de l'autre. Réappliquer de façon séquentielle ces transactions donnerait le même résultat que les mises à jour concurrentes appliquées en direct. On pourrait obtenir un effet similaire en produisant manuellement un journal de transactions à partir de l'application.

Cette approche de journalisation permettrait d'implanter le scénario suivant :

1. on extrait les informations de la base de données de l'application ;
2. on génère une ontologie de départ et on exécute les procédures de raisonnement sur cette ontologie ;
3. applique continuellement les MAJ à l'ontologie à partir de la journalisation de la base de données ;
4. on exécute le raisonnement sur les changements apportés à chaque application des MAJ et on extrait les conséquences des MAJ sur les conclusions déduites ;

5. on applique ces nouvelles conclusions dans la base de données à mesure qu'elles sont produites.

Un tel cycle permettrait de disposer des conclusions mises à jour presque instantanément dans l'application. Ce mode presque en direct (*near real-time*) couvre la plupart des situations dans une application d'entreprise. Malheureusement, la quatrième étape cause problème. Les *reasoners* permettent depuis un certain temps de faire des modifications à une ontologie et de refaire le processus de raisonnement sans recommencer du début. Cependant, nous sommes intéressés à savoir ce qui change dans les conclusions. À l'heure actuelle, il faut poser les questions une nouvelle fois et comparer avec la dernière série de réponses pour obtenir les modifications qu'il faudrait apporter à la base de données de l'application. Une telle comparaison n'est pas compatible avec un mode presque en direct à cause du volume de traitement à effectuer à chaque mise à jour.

Un *reasoner* ne peut donc pas être utilisé à l'heure actuelle en mode de mises à jour asynchrones. Nous reviendrons sur ce point dans la conclusion de ce mémoire car la résolution de ce problème nous apparaît comme une piste de recherche importante.

5.5 Scénarios sans retour direct à l'application

Jusqu'à maintenant nous avons fait l'hypothèse que les logiques de description étaient utilisées pour effectuer une classification d'objets et que le résultat de cette classification était retourné à l'application pour permettre d'exploiter des hiérarchies fonctionnelles multiples.

Plusieurs autres scénarios sont possibles. D'abord la classification n'est pas le seul service de raisonnement offert par un *reasoner*. Par exemple, si certains requis de l'application sont de nature ontologiques, l'utilisation d'un *reasoner* peut être beaucoup plus directe. L'exemple des pizzas est très utilisé dans le domaine des ontologies. Si une application a besoin de savoir si une pizza Margherita (tomate, mozzarella, basilique) est une pizza végétarienne, un *reasoner* est l'outil tout indiqué (le nombre d'ingrédients est fixé et aucun n'est de la viande alors c'est une pizza végétarienne). Dans ce mémoire, nous nous sommes limités à la classification car celle-ci a une portée plus large pour les

applications d'entreprise.

Mais la classification peut être utile à autre chose qu'à produire des comportements polymorphes sur plusieurs hiérarchies simultanément. Par exemple, la classification de services sous forme de niveaux de service (ex. le nombre d'étoiles associées à une chambre d'hôtel) peut faire intervenir un grand nombre de règles qu'il serait possible de faire exécuter par un *reasoner*. Notre objectif est qu'une fois que le lecteur aura compris le genre de contribution potentielle du raisonnement automatique, il pourra commencer à identifier des utilisations potentielles dans les listes de requis des applications.

Finalement, et c'est ce que nous regardons plus en détail dans cette section, le résultat de la classification peut être utile sans retourner les résultats au niveau opérationnel de l'application. Nous analyserons brièvement les connexions possibles vers l'intelligence d'affaire (BI).

5.5.1 Intelligence d'affaires - BI

Initialement constitué des entrepôts de données, le domaine du BI - *Buisiness Intelligence* ou Intelligence d'affaires comprend aujourd'hui toute une série de techniques d'analyse, effectuées par des outils spécialisés, sur des données autant internes qu'externes à l'organisation pour générer de l'information pertinente à la gestion d'une organisation.

L'utilisation du raisonnement automatique en ce qui concerne le BI serait un sujet de recherche en soi. Nous utiliserons un exemple simple provenant des entrepôts de données pour illustrer la connexion avec ce que l'on a dit jusqu'à présent sur les logiques de description.

La technique de base des entrepôts de données consiste à construire des *star schemas* ou schémas en étoile. On analyse des transactions selon un certain nombre d'axes. Chaque axe a une énumération de valeurs possibles. Certains axes peuvent organiser les valeurs possibles de façon hiérarchique. Par exemple, un axe peut être la région géographique du client (pays, province), un autre la date de la transaction (année, mois), un autre le type de produit vendu, etc. On cumule toutes les transactions partageant les mêmes valeurs sur tous les axes. Par exemple, on aurait une entrée pour les ventes de

janvier 2012 de rasoir à 8 lames au Québec. Il peut y avoir des millions d'entrées de la sorte. On envoie ensuite le fichier produit, avec la définition des axes, à un analyseur de données à plusieurs dimensions pour extraire les informations pertinentes.

Certains axes pourraient être sont moins « évidents », par exemple on pourrait vouloir séparer les clients par « type » avec une procédure de détermination du « type » référant à plusieurs propriétés et associations de l'objet. On retombe ici dans un problème de classification où le résultat est chargé à l'entrepôt de données plutôt que d'être retourné au système opérationnel.

CHAPITRE 6

ÉVALUATION DES RÉSULTATS

Nous évaluerons dans ce chapitre si l'objectif de recherche du mémoire a été atteint. Nous passerons ensuite en revue les principales limites et forces du patron d'architecture.

6.1 Atteinte des objectifs

Reprenons encore une fois l'objectif de recherche.

Notre objectif de recherche est d'étudier comment le raisonnement automatique, basé sur les logiques de description, pourrait être mis à profit pour traiter efficacement le problème des hiérarchies fonctionnelles multiples dans les applications d'entreprise sans changer les paradigmes de base de ces applications et en utilisant des outils de raisonnement existants.

En montrant que chaque élément de démarche associé à chaque objectif (voir tableau 2.II) a été rempli correctement, nous montrerons que l'objectif, tel que formulé, a été atteint dans le cadre du projet de mémoire.

6.1.1 Objectifs détaillés

Cette section rappelle comment chacun des objectifs a été atteint.

6.1.1.1 Établir la faisabilité

La caractéristique retenue pour établir une contribution potentielle des logiques de description est leur capacité à classifier des objets. Comme il s'agit d'un de leurs services de base, nous avons diminué le risque d'avoir des problèmes avec le *reasoner* lui-même.

Pour monter qu'un aller-retour entre l'application et les logiques de description était possible, nous avons élaboré le patron d'architecture « Classification en lots ». Ce patron d'architecture n'utilise que des outils standards pour effectuer des tâches qui leur sont normalement assignées.

Pour s'assurer que les requis non-fonctionnels étaient remplis nous avons : *a*) minimisé le risque vis-à-vis des *reasoners* en utilisant leur mode de fonctionnement standard ; *b*) conservé les approches relationnelles et orienté-objet pour l'application d'entreprise ; et *c*) assuré que la classification ne soit pas sur le chemin critique d'une transaction en direct.

6.1.1.2 Identifier un scénario type

En rapprochant la flexibilité de la fonction de classification des logiques de description et la rigidité des hiérarchies fonctionnelles en orienté-objet, particulièrement dans le cas des hiérarchies multiples, un potentiel est rapidement apparu. L'idée de viser le problème des hiérarchies multiples a stabilisé l'objectif de recherche du mémoire.

Nous avons vu à la section 3.2 que des solutions sont actuellement appliquées au problème des hiérarchies fonctionnelles. Ces solutions ont cependant leurs limites :

- Les constructions génériques sont applicables dans le cas des hiérarchies techniques où le comportement d'un type d'objet est appliqué sur un autre type (ex. listes de personnes). Elles ne sont pas applicables dans le cas de deux hiérarchies fonctionnelles multiples qui sont toutes deux basées sur les données plutôt que sur les comportements et qui n'agissent pas l'une sur l'autre.

- Le *mapping* entre les bases de données et les modèles objets utilisés dans les ORM se limite à des correspondances directes. Dès que la correspondance est plus complexe des couches d'accès aux données doivent être développées.

- Les variables-types et les indicateurs sont très présents dans les applications d'entreprise. Ces techniques annulent les gains potentiels de l'approche orienté-objet liés à l'encapsulation.

Nous avons graduellement précisé la relation entre les hiérarchies multiples en orienté-objet et le treillis de subsomption d'un *reasoner*. La clé est que pour chaque hiérarchie, seule la partition définie par les classes feuilles doit être définie.

6.1.1.3 Architecture pour le scénario type

On a vu que la façon de ramener les résultats de la classification dans l'application d'entreprise doit passer par l'initialisation de champs de contrôle de la base de données car on ne peut pas changer un objet de classe en orienté-objet.

Pour la création des objets, on a étudié la possibilité de ne créer (dynamiquement) qu'un sous-ensemble des classes d'intersection pour finalement se rabattre sur l'approche des objets satellites, chacun étant défini dans une des hiérarchies fonctionnelles.

L'approche des objets satellites pose le problème de synchronisation de ces objets. Ce genre de synchronisation est cependant un service de base des ORM, qui deviennent alors une composante naturelle et centrale de la solution. La découverte, en cours de projet, du rôle potentiel des ORM, a amené une simplification substantielle dans le développement du prototype. Notre démarche est alors apparue comme le développement d'un patron d'architecture.

Nous avons également décrit les trois modèles objets requis pour paramétrer l'ORM : Mise à jour, Hiérarchique et Types.

6.1.1.4 Conception et réalisation du prototype

Les annexes IV et V décrivent le prototype réalisé dans le cadre de ce mémoire. Ce prototype implante le patron d'architecture décrit et produit le comportement polymorphe recherché. Certaines remarques importantes sur le prototype suivent.

Le choix de l'environnement de développement reposait entre Java et .NET. Nous avons choisi .NET car nous l'utilisons déjà dans le cadre d'autres mandats. Cela a demandé le développement d'un traducteur pour la syntaxe de Manchester et d'un client Owl-Link.

L'interface avec le *reasoner* a été réalisé avec OWL-Link. Nous avons également élaboré la séquence de questions à poser au *reasoner* pour obtenir la classification.

Pour l'interface avec la base de données nous avons codé en dur un module d'extraction et un module de mise à jour des types. Une solution opérationnelle exigerait de rendre ce module générique. Nous le proposons comme piste de recherche.

6.1.1.5 Élaborer la méthodologie pour le scénario type

Les requis pour mettre en place le patron de base sont essentiellement des livrables de modélisation soit : *a*) les trois modèles destinés au ORM ; et *b*) l'ontologie OWL définissant formellement les partitions de chacune des hiérarchies.

L'élaboration de la méthodologie n'a pas représenté de problème particulier.

6.1.1.6 Évaluer forces et faiblesses du patron d'architecture

Les conditions d'utilisation ont été précisées avec les limitations que l'utilisation doit être *en lots* et que la classification doit être faite sur tous les objets à chaque cycle.

En termes techniques on doit viser des applications de type *workflow* qui ont le comportement séquentiel requis et idéalement des situations où les données peuvent être partitionnées pour s'assurer qu'une classification d'un groupe d'objet ne demande pas de reprendre la classification sur l'ensemble.

Malgré les contraintes de l'approche, plusieurs situations réelles ont cependant les caractéristiques requises. [1]

6.1.1.7 Étudier d'autres scénarios

On ne peut envisager un traitement synchrone avec l'application car les *reasoner* ne supportent pas le concept de transaction. Pour un traitement asynchrone, où la base de données sérialiserait les transactions pour le *reasoner*, plus de recherche est requis pour voir si le fait de limiter les changements aux assertions sur les objets et les impacts à l'appartenance à des classes précises pourrait permettre de limiter le travail requis pour identifier les changements aux classifications de façon continue. On propose ce scénario comme piste de recherche.

On a également vu que l'on pouvait considérer la classification comme une étape préliminaire pour un entrepôt de données.

[1] Les *workflows* sont omniprésents dans les applications d'entreprise. Par exemple, traitement des dossiers des documents, élaboration et révision de toutes sortes de documents et formulaires, mise-à-jour d'inventaires de produits, etc.

6.2 Limites et forces

L'objectif de recherche étant atteint, essayons de caractériser les limites ainsi que les forces du patron d'architecture élaboré.

6.2.1 Limites

Si les approches actuelles, face au problème des hiérarchies fonctionnelles en orienté-objet, ont leurs problèmes, le patron d'architecture que nous avons développé n'en est pas exempt. Notons toutefois que notre objectif n'était pas d'être *meilleurs* que les approches actuelles, simplement de montrer que l'utilisation des logiques de description était possible.

Les principales limites de notre patron d'architecture :

1. À chaque cycle de classification, il faut relire les objets qui auraient pu *changer de classe*. Ce genre de requis de relecture n'est pas typique en orienté-objet.
2. On demande une expertise supplémentaire (modélisation OWL) sans en enlever en contrepartie. Le *mix* d'expertises requises pour un projet est donc plus complexe.

6.2.2 Forces

Malgré les limites de la démarche, le patron d'architecture a des forces importantes qui ne manqueraient pas d'avoir un impact positif sur une application d'entreprise.

1. L'utilisation de hiérarchies fonctionnelles multiples dans l'application devient très naturelle une fois la classification effectuée. Le code devient tout aussi *propre* que pour les syntaxes génériques par exemple. En ce sens, on peut dire que le problème est correctement résolu par le patron d'architecture proposé.
2. On pourra faire un nettoyage des indicateurs et structures de branchement actuellement utilisés pour faire le même travail. Donc, gain de *maintenabilité* à prévoir.
3. La logique pour déterminer le *type* d'un objet est clairement isolée, d'où un autre gain quant à la maintenance.
4. Et le plus important, tout ce qui a trait à la définition et à la maintenance des hiérarchies fonctionnelles passe du côté des modèles et sort de l'application. Les

modèles objets de l'ORM, incluant les hiérarchies, deviennent le mode de communication au lieu du modèle de données. Ces modèles sont plus près du modèle conceptuel du domaine. L'application peut ainsi être conçue en lien plus direct avec le domaine, un gain majeur pour les coûts de développement et de maintenance ainsi que la pertinence de l'application.

CHAPITRE 7

TRAVAUX FUTURS ET EXTENSIONS

7.1 Prochaines activités

Beaucoup de travail a été investi dans ce projet de mémoire. Nous sommes très satisfaits d'avoir pu définir un objectif de recherche à la fois conséquent et original et d'avoir pu le satisfaire en suivant la démarche formelle associée à un mémoire de maîtrise. De plus, la réalisation du prototype a été très motivante en permettant de *voir* les idées se réaliser dans la réalité.

La simplicité et la clarté du code obtenu en bout de ligne pour manipuler des hiérarchies fonctionnelles multiples dans un environnement orienté-objet a grandement dépassé nos attentes. Nous avons donc le sentiment d'avoir non seulement rempli notre objectif, mais de l'avoir fait de façon *naturelle*.

Nous possédons maintenant une bien meilleure compréhension aujourd'hui du rôle majeur que peut jouer un ORM, non seulement comme intermédiaire mais également comme couche d'abstraction. Nous transférerons cette expertise dans notre pratique professionnelle. Par exemple, notre patron d'architecture peut être utilisé sans avoir recours aux logiques de description. On pourrait utiliser des déclencheurs (*triggers*) sur la base de données pour mettre à jour les champs de contrôle des tables liées aux hiérarchies.

L'état d'avancement des modules sérialiseur OWL et client OWL-Link (voir annexe IV), développés en .NET, permettent d'envisager la production d'une librairie d'accès aux *reasoners* OWL pour le monde .NET. Comme une telle librairie n'existe pas à l'heure actuelle, il s'agirait d'une contribution au monde du logiciel libre. Nous planifions terminer une telle librairie au cours des prochains mois car il s'agissait d'un de nos objectifs personnels dans le cadre de ce mémoire.

Finalement, du côté de l'objectif de recherche en tant que tel, la suite est de réaliser quelques projets-pilotes pour détailler le mode opératoire du patron et de commencer à faire des comparaisons avec les autres approches possibles.

Une utilisation plus large de l'approche impliquerait une plus grande synchronisation entre le *reasoner* et la base de données. Une expertise devrait donc être développée autour d'approches visant également l'ajout de composantes logiques aux bases de données, notamment le domaine des bases de données déductives.

7.2 Pistes de recherche

Nous proposons deux pistes de recherche pour cette section pour étendre notre travail à des projets-pilotes plus conséquents et pour comparer notre approche à d'autres façons de traiter le problème des hiérarchies multiples dans les applications orienté-objet complexes.

Une perspective beaucoup plus large s'ouvre également autour de l'idée d'ajouter des composantes existantes de raisonnement automatique à des applications et de développer des patrons d'architecture et des outils permettant leur utilisation pour des types de problèmes spécifiques.

7.2.1 Convertisseur OWL générique

Tel que mentionné à la section IV, le convertisseur OWL est la composante qui : *a*) extrait les objets de la base de données et la valeur de leurs propriétés ; *b*) produit les assertions OWL décrivant ces objets ; *c*) interprète les résultats provenant du *reasoner* pour déterminer la classe de chaque objet dans chaque hiérarchie fonctionnelle ; et *d*) met à jour les champs de contrôle dans la base de données pour conserver la classe à laquelle appartient chaque objet dans chaque hiérarchie.

Les deux premières activités préparent le classement par le *reasoner* alors que les deux dernières appliquent le résultat du classement dans la base de données.

Pour obtenir un bénéfice maximal de la classification par un *reasoner*, le convertisseur devrait être générique. Il n'y aurait alors aucun code requis dans l'application pour mettre en place le patron de base, sauf l'appel au processus de classification lui-même.

Pour effectuer l'extraction des objets et de leurs propriétés, nous utilisons déjà le modèle objet du ORM qui est utilisé pour créer les objets et mettre à jour leurs propriétés.

Nous appelons ce modèle le modèle de Mise à jour. L'idée serait de développer un mécanisme d'annotation de ce modèle pour guider un extracteur générique dans l'extraction des données et la génération des axiomes.

L'environnement .NET utilise un environnement d'exécution (*runtime*). Beaucoup d'information sur le code est disponible à l'exécution à travers un mécanisme de *Réflexion*, du même type que celui de l'environnement Java. On peut ajouter des annotations au code, classes, méthodes, etc. qui seront disponibles à l'exécution pour changer le comportement du système. Il s'agit d'une implantation du concept de programmation orienté-aspect. Par exemple, le ORM de .NET, le *Entity Framework*, a déjà un mécanisme d'annotation des modèles pour guider les validations à appliquer par des modules de saisie génériques.

Annoter le modèle de Mise à jour pour guider un extracteur générique semble donc un objectif réalisable. De la même façon, on pourrait annoter le modèle des Types pour guider le convertisseur pour appliquer les résultats de la classification.

On pourrait également extraire des modèles une partie de l'information requise pour produire l'ontologie de définition des partitions des hiérarchies fonctionnelles. L'information de base sur les classes, les propriétés et les objets est déjà présente dans les modèles. En partant de cette base, serait-il possible d'entrer les définitions comme des annotations aux modèles, éliminant ainsi le besoin d'avoir une ontologie OWL séparée ?

7.2.2 Utilisation asynchrone

On a déjà parlé de l'utilisation potentielle de *reasoner* en mode asynchrone (5.4.2 et 6.1.1.7). L'objectif serait d'obtenir une reclassification d'un objet presque en temps réel (*near real time*) lors de sa création ou de la modification de ses propriétés.

Les *reasoners* permettent depuis peu de modifier des axiomes déjà envoyés et de répondre à de nouvelles questions sans avoir à recréer tout le treillis de subsomption. On sait également qu'un système de gestion de base de données moderne peut sérialiser ses transactions lors de la journalisation.

À partir de la journalisation, on pourrait envoyer les mises à jour au *reasoner* de façon séquentielle, simulant ainsi un traitement transactionnel. Il ne resterait que le délai de

journalisation et d'application des modifications. On obtiendrait ainsi presque du temps réel sans que le *reasoner* n'ait à supporter le concept de transaction.

Le problème ici est qu'une modification à une propriété d'un objet peut changer la classification des autres objets. Par exemple, si Paul n'est plus le fils de Marie alors Marie peut très bien ne plus être une Mère [1]. Il faudrait donc, en théorie, vérifier la classification de tous les objets à chacune des modifications d'un objet quelconque, ce qui brise le fonctionnement presque en temps réel.

L'objectif de recherche serait donc de développer une façon d'identifier les impacts des modifications aux axiomes sur le résultat de la classification, sachant que les seules modifications possibles concernent les propriétés des objets (assertions) et non la définition des classes.

[1] À noter qu'il n'y a rien de dynamique dans le concept de classification. Les propriétés des objets déterminent leur classification et si ces propriétés changent alors la classification change. Ce changement est considéré comme instantané. Cette hypothèse d'instantanéité des changements est implanté dans les système de bases de données relationnelle qui répondent aux requêtes comme si toutes les transactions étaient traitées en séquence. Le problème noté ici est que la classification devrait théoriquement être refaite à chacune des transactions, ce qui pause évidemment un problème de performance.

CHAPITRE 8

CONCLUSION

Ce mémoire a démontré qu'il est possible d'utiliser les logiques de description pour traiter efficacement le problème des hiérarchies fonctionnelles multiples dans les applications d'entreprise sans changer les paradigmes de base de ces applications et en utilisant des outils de raisonnement existants.

Sans changer le paradigme de base veut dire que nous cherchons à utiliser les avantages de logiques de description mais en ayant le moins d'impact possible sur les ressources humaines et les processus de développement.

Cette démonstration a été effectuée par le développement d'un patron d'architecture utilisant un ORM (*Object Relational Mapper*) pour construire une structure d'objets satellites évitant d'avoir recours à l'héritage multiple et à des classes d'intersection. Un *reasoner* existant est utilisé pour déterminer dans quelle classe créer les objets satellites dans les hiérarchies. On utilise la capacité des logiques de description à classifier les objets mais dans une approche générale compatible avec l'orienté-objet et le modèle relationnel.

Ce patron d'architecture constitue notre contribution principale et permet de remplir l'objectif du mémoire.

BIBLIOGRAPHIE

[1] Alfred V. Aho. Software and the future of programming languages. *Science*, 303 (5662):pp. 1331–1333, 2004. ISSN 00368075. URL http://www.jstor.org/stable/3836392.

[2] Deborah J. Armstrong. The quarks of object-oriented development. *Commun. ACM*, 49(2):123–128, février 2006. ISSN 0001-0782. URL http://doi.acm.org/10.1145/1113034.1113040.

[3] Bard Bloom, Paul Keyser, Ian Simmonds et Mark Wegman. Ferret : Programming language support for multiple dynamic classification. *Computer Languages, Systems and Structures*, 35(3):306 – 321, 2009. ISSN 1477-8424. URL http://www.sciencedirect.com/science/article/pii/S1477842408000225.

[4] Craig Chambers. Predicate classes. Dans *ECOOP '93 Conference Proceedings*, pages 268–296. Springer-Verlag, 1993.

[5] O. Lassila M. Paolucci T. Payne D. Martin, M. Burstein et S. McIlr aith. *Describing Web Services using OWL-S and WSDL*, novembre 2004. URL http://www.daml.org/services/owl-s/1.1/owl-s-wsdl.html.

[6] David Martin et al. *OWL-S : Semantic Markup for Web Services*, novembre 2004. URL http://www.w3.org/Submission/OWL-S.

[7] David Martin et al. Bringing semantics to web services : The OWL-S approach. Dans J. Cardoso et A. Sheth, éditeurs, *SWSWPC 2004, Semantic Web Services and Web Process Composition, First International Workshop*, pages 26–42. Springer-Verlag, 2005.

[8] Gartner Group. *Technology glossary*, octobre 2011. URL http://www.gartner.com/technology/it-glossary/.

[9] Gavin King. *Hibernate website*, février 2012. URL `http://www.hibernate.org`.

[10] Graig Larman. *Applying UML and patterns*. Pearson Education, 2005.

[11] Microsoft MSDN. *Entity Framework website*, février 2012. URL `http://msdn.microsoft.com/en-us/library/bb399572.aspx`.

[12] D. Nardi et R. J. Brackman. An introduction to description logics. Dans F. Baader, D. Calvenese, D. L. McGuinness, D. Nardi et P. F. Patel-Schneider, éditeurs, *The Description Logic Handbook*, pages 1–44. Cambridge University Press, 2007.

[13] JamesJ. Odell. Dynamic and multiple classification. Dans *Object-Oriented Behavioral Specifications*, volume 371 de *The Springer International Series in Engineering and Computer Science*, pages 193–196. Springer US, 1996. ISBN 978-0-7923-9778-6. URL `http://dx.doi.org/10.1007/978-0-585-27524-6_12`.

[14] Agostino Poggi. Ol3 : An OWL object-oriented library for the realization of ontology based applications. Dans *Proceedings of the 13th WSEAS International Conference on Computers*. WSEAS, 2009.

[15] Perl 5 Enterprise Project. *Definitions*, octobre 2011. URL `http://www.officevision.com/pub/p5ee/definitions.html`.

[16] Antero Taivalsaari. Classes vs. prototypes - some philosophical and historical observations. Dans *Journal of Object-Oriented Programming*, pages 44–50. SpringerVerlag, 1996.

[17] James Hendler Tim Berners-Lee et Ora Lassila. *The Semantic Web*, mai 2001. URL `http://www.scientificamerican.com/article.cfm?id=the-semantic-web`.

[18] Peter Van Roy. Programming Paradigms for Dummies : What Every Programmer Should Know. Dans G. Assayag et A. Gerzso, éditeurs, *New Computa-*

tional Paradigms for Computer Music. IRCAM/Delatour, France, 2009. URL http://www.info.ucl.ac.be/VanRoyChapter.pdf.

[19] W3C. *Web Services Description Language (WSDL) 1.1*, mars 2001. URL http://www.w3.org/TR/wsdl.

[20] W3C. *OWL Web Ontology Language - Overview*, février 2004. URL http://www.w3.org/TR/owl-features/.

Annexe I

Ontologies et web services : OWL-S

Notre recherche des autres tentatives d'intégration des logiques de description dans les applications d'entreprise ne nous a permis de trouver qu'un seul exemple que nous présentons brièvement.

Une proposition du début des années 2000 appelée OWL-S ou *Web Ontology Language for Services* [7] visant à ajouter des descriptions sémantiques aux services web et à utiliser ces descriptions pour automatiser certaines tâches associées à l'exécution, à la recherche et à la composition de ces services. Ces travaux visaient des services web faisant partie du web sémantique. Ils sont toutefois pertinents pour notre propos parce que les applications d'entreprise sont souvent structurées comme un ensemble de services interconnectés (approche SOA ou *Service Oriented Architecture*).

Les services web sont le plus souvent décrits à l'aide du langage WSDL ou *Web Services Definition Language* [19], un standard du W3C, qui est un langage XML définissant le format des messages en entrée et en sortie ainsi que l'environnement d'exploitation (protocoles, adresses, etc.) d'un service web. WSDL se contente de décrire le format des messages en séparant chacun en « parties » et en se référant à un schéma XML pour définir le type de chaque partie. Cette description détaillée facilite l'interopérabilité entre des services web développés par différents auteurs sur différentes plateformes technologiques. Une documentation textuelle séparée du web service est cependant requise pour décrire l'intention de chacun des types utilisés ainsi que l'enchainement entre les différents messages. Cette documentation doit être interprétée par un programmeur qui doit codifier manuellement un programme-client pour pouvoir utiliser le service web. C'est cette étape que les auteurs de OWL-S visaient à automatiser.

OWL-S [6] est une extension à OWL pour décrire des web services au niveau sémantique. Une description OWL-S comprend un *Profile* qui décrit ce que fait le web service, un *Process model* qui décrit comment le web service fonctionne et un *Grounding* qui fait le lien entre les descriptions abstraites du *profile* et du *process model* et la

description concrète du service en WSDL. L'idée est de mettre en correspondance entre les types des schémas XML du WSDL et les types OWL abstraits de OWL-S [5]. Cette correspondance permet de réaliser des déductions en utilisant un *reasoner* et ensuite de reconvertir les résultats dans le monde concret des services web.

Comme exemple simple de l'utilisation de cette approche, supposons que nous ayons un répertoire de services web décrits à l'aide de OWL-S et que toutes les définitions réfèrent à la même ontologie OWL. Si nous recherchons un service web permettant d'acheter des articles de sports alors un service web permettant d'acheter des équipements de hockey sera sélectionné car on pourra *raisonner*, à partir de l'ontologie, qu'un équipement de hockey et en fait un article de sport. Les outils accompagnant OWL-S permettaient évidemment de réaliser des tâches beaucoup plus complexes mais il n'est pas nécessaire d'aller plus en détails ici.

Ce qu'il faut retenir de OWL-S c'est qu'un projet majeur du début des années 2000 est parvenu à utiliser un *reasoner* basé sur OWL et les logiques de description pour réaliser des tâches complexes dans un environnement de services web. Certaines tâches étaient réalisées par le *reasoner* dans un environnement OWL alors que d'autres étaient réalisées par les services web eux-mêmes, donc avec des méthodes traditionnelles. On faisait cohabiter ces deux environnements en faisant une correspondance entre les types traditionnels, ici des types définis dans des schémas XML associés aux descriptions WSDL, et les classes OWL définies par les descriptions OWL-S et en traduisant dans les deux sens entre ces deux modes de représentation. Cette démarche n'est pas très éloignée de l'approche adoptée dans le présent mémoire et en valide en partie la faisabilité.

Annexe II

Décrire des classes en OWL

Le *World Wide Web* ou simplement le web est un ensemble de pages inter-reliées par des hyperliens. Cet ensemble peut être *navigué* par des humains en utilisant un fureteur. Les humains comprennent le sens des pages et des liens et peuvent donc retrouver l'information recherchée. Dès la naissance du web, ses instigateurs ont travaillé à construire le même genre de réseau au niveau des données pour permettre à des machines de faire une partie du travail de recherche et de synthèse de l'information. Le terme de *web sémantique* a été choisi pour dénommer cet ensemble de données inter-reliées [17].

Le langage RDF (*Resource Description Framework*) a été développé pour identifier les données à l'intérieur des pages web et pour les lier. Le langage SPARQL a ensuite été développé pour interroger des réseaux RDF. L'étape suivante a consisté à augmenter RDF pour permettre de définir des concepts et permettre d'extraire l'information implicite des réseaux sémantiques. RDF-S a été une première extension en ce sens. Plus tard, le langage OWL (*Web Ontology Language*) a été créé pour réaliser les inférences requises à partir de définitions formelles. OWL est basé sur les logiques de description. L'effervescence du début des années 2000 autour du web sémantique a amené beaucoup de recherche et le développement de plusieurs outils dans le domaine des logiques de description.

Dans ce mémoire, nous utilisons OWL comme langage même si nous ne traitons pas directement du web sémantique, et ce, principalement à cause des standards et outils développés autour de ce langage. Nous présentons ici les principales correspondances entre les concepts des logiques de description et la syntaxe de OWL.

Le tableau II.I fait la correspondance entre les concepts de base présentés à la section 2.2 avec leurs équivalents en logiques de description et en OWL.

Une des particularités de OWL est que le langage a été défini avec plusieurs syntaxes équivalentes satisfaisant des besoins particuliers. Nous utilisons dans ce mémoire la syntaxe de Manchester qui est à la fois compacte et facile à comprendre pour des humains.

Tableau II.I – Correspondance des concepts en OWL

Concept	Logiques de description	OWL
Objet	Object	Individual
Propriété référentielle	Role	ObjectProperty
Propriété scalaire	Role	DataProperty
Comportement	N/A	N/A
Classe	Concept	Class
Identifiant	N/A	IRI *International Resource Identifier*

Les prochains paragraphes donnent des exemples de base de la syntaxe de Manchester. L'annexe III présente un exemple d'une ontologie complète.

On peut déclarer explicitement l'existence d'un individu et faire une assertion de son appartenance à une classe (un type en syntaxe de Manchester) et préciser la valeur de ses propriétés.

```
Individual : Mary
    Type : Person
    Facts :
        hasHusband John
        hasAge "51"^^xsd:integer
```

À noter que l'affectation d'une valeur scalaire à une propriété est une extension des logiques de description (*concrete worlds*).

On peut déclarer l'existence d'une classe et définir une hiérarchie explicite entre les classes.

```
Class : Woman
    SubClassOf : Person
```

Ou poser des définitions pour les classes. Ces définitions sont la base des inférences comme pour les logiques de descriptions.

```
Class : Mother
    EquivalentTo : Woman and Parent
Class : Parent
    EquivalentTo : hasChild some Person
```

L'ensemble des définitions de classes et d'assertions concernant les individus constituent une ontologie, ce qui correspond à définir une TBox et une ABox en logique de description et donc à spécifier les axiomes de la logique représentant le domaine étudié.

Annexe III

Raisonnement automatique et logiques de description

On a vu à la section 3.4.1 qu'une méthode de preuve complète, valide et monotone permet de vérifier la validité d'un énoncé de façon systématique et peut donc être mécanisée. La procédure exacte à utiliser dépend des opérateurs de la logique considérée et peut s'avérer très complexe. Nous présentons ici un cas simple pour introduire l'approche générale. Nous visons à convaincre le lecteur que la classification d'un individu peut effectivement s'effectuer de façon automatique.

Exemple d'ontologie et notation

On se servira de l'exemple de la figure III.1 qui est une ontologie OWL suivant la syntaxe de Manchester (voir annexe II), produite avec le logiciel Protégé et à laquelle on a enlevé les espaces blancs pour sauver de l'espace.

L'ontologie est très simple, elle définit :

- 4 types de programmes (lignes 06 à 09) ;
- le concept de programme est l'union de ces 4 types (lignes 10 à 12) ;
- un étudiant est un individu inscrit à un programme (lignes 13-14) ;
- un étudiant gradué est un étudiant inscrit à un programme de maîtrise ou de doctorat (lignes 15 à 18) ;
- même principe pour les autres types d'étudiants (lignes 19 à 24) ;
- on identifie ensuite quatre programmes particuliers. Par exemple, la maîtrise en informatique est un programme de science ainsi qu'un programme de maîtrise. (lignes 27 à 32) ;
- finalement, Jacques est identifié comme étant inscrit à la maîtrise en informatique et Marie au bacc. en nutrition (lignes 33 à 36).

Dans le langage des logiques des description, introduit à la section 3.4.2, on séparerait cette ontologie dans un ensemble de définitions de classes comme à la figure III.2

Figure III.1 – Ontologie de classement des étudiants

```
Note: Les numéros de lignes sont pour référence seulement.

01- Prefix: owl: <http://www.w3.org/2002/07/owl#>
02- Prefix: ce: <http://www.dogico.com/ontologies/2011/12/ClassementEtudiants.owl#>
03- Ontology: <http://www.dogico.com/ontologies/2011/12/ClassementEtudiants2.owl>
04- ObjectProperty: ce:estInscrit
05- Class: owl:Thing
06- Class: ce:ProgrammesDoctorat
07- Class: ce:ProgrammesMedecine
08- Class: ce:ProgrammesMaitrise
09- Class: ce:ProgrammesScience
10- Class: ce:Programmes
11-     EquivalentTo: ce:ProgrammesDoctorat or ce:ProgrammesMedecine
12-       or ce:ProgrammesMaitrise or ce:ProgrammesScience
13- Class: ce:Etudiants
14-     EquivalentTo: ce:estInscrit some ce:Programmes
15- Class: ce:EtudiantsGradues
16-     EquivalentTo:
17-         (ce:estInscrit some ce:ProgrammesDoctorat)
18-          or (ce:estInscrit some ce:ProgrammesMaitrise)
19- Class: ce:EtudiantsScience
20-     EquivalentTo: ce:estInscrit some ce:ProgrammesScience
21- Class: ce:EtudiantsMedecine
22-     EquivalentTo: ce:estInscrit some ce:ProgrammesMedecine
23- Class: ce:EtudiantsFacultaires
24-     EquivalentTo: ce:EtudiantsMedecine or ce:EtudiantsScience
25- Individual: ce:MaitriseInformatique
26-    Types: ce:ProgrammesScience, ce:ProgrammesMaitrise
27- Individual: ce:DoctoratOrthophonie
28-    Types: ce:ProgrammesDoctorat, ce:ProgrammesMedecine
29- Individual: ce:BaccMathematiques
30-    Types: ce:ProgrammesScience
31- Individual: ce:BaccNutrition
32-    Types: ce:ProgrammesMedecine
33- Individual: ce:Jacques
34-    Facts:  ce:estInscrit  ce:MaitriseInformatique
35- Individual: ce:Marie
36-    Facts:  ce:estInscrit  ce:BaccNutrition
```

pour constituer la terminologie (TBox) et dans un ensemble d'assertions comme à la figure III.3 pour constituer la description du monde (ABox).

Pour le lecteur plus à l'aise avec la notation de la logique des prédicats du premier ordre, on peut reformuler la définition (D2) :

$$\mathsf{Etudiant}(x) \equiv \exists y(\mathsf{estInscrit}(x,y) \wedge \mathsf{Prog}(y)) \quad \text{ou bien}$$

$$\mathsf{Etudiant}(x) \equiv \exists \mathsf{estInscrit}.\mathsf{Prog}(x)$$

Nous utiliserons cette dernière variante dans la description de la méthode de preuve

Figure III.2 – Définition des classes d'étudiants (TBox)

$(D1)$ Prog $\equiv$ ProgDoc $\sqcup$ ProgMed $\sqcup$ ProgMaitrise $\sqcup$ ProgSci
$(D2)$ Etudiant $\equiv \exists$estInscrit.Prog
$(D3)$ EtudiantGradue $\equiv \exists$estInscrit.ProgDoc $\sqcup \exists$estInscrit.ProgMaitrise(x)
$(D4)$ EtudiantSci $\equiv \exists$estInscrit.ProgSci
$(D5)$ EtudiantMed $\equiv \exists$estInscrit.ProgMed
$(D6)$ EtudiantFac $\equiv$ EtudiantSci $\sqcup$ EtudiantMed

Figure III.3 – Assertions du monde des étudiants (ABox)

$(A1)$ ProgSci(MaitriseInfo)
$(A2)$ ProgMaitrise(MaitriseInfo)
$(A3)$ ProgDoc(DocOrtho)
$(A4)$ ProgMed(DocOrtho)
$(A5)$ ProgSci(BaccMath)
$(A6)$ ProgMed(BaccNutri)
$(A7)$ estInscrit(Jacques,MaitriseInfo)
$(A8)$ estInscrit(Marie,BaccNutri)

dans les sections suivantes. Dans tous les cas il faut lire qu'un individu est un étudiant s'il existe un programme auquel il est inscrit.

Pour revenir à l'ontologie, notons qu'elle ne spécifie pas les relations entre les différentes classes d'étudiants, elle ne fait que définir chaque classe en fonction des individus qui la compose. Par exemple, il n'est pas spécifié que les étudiants gradués sont en fait des étudiants. Un humain n'aurait aucun problème à déduire les relations entre les classes d'étudiants présentées à la figure III.4. Nous montrerons comment faire ces déductions de façon automatique.

Notre ontologie n'utilise volontairement que les opérateurs de base union et existentiel sur une restriction de valeur. La négation, l'intersection et l'opérateur universel sur une restriction de valeur seront aussi utilisés dans le processus de raisonnement (voir

Figure III.4 – Relation entre les classes d'étudiants

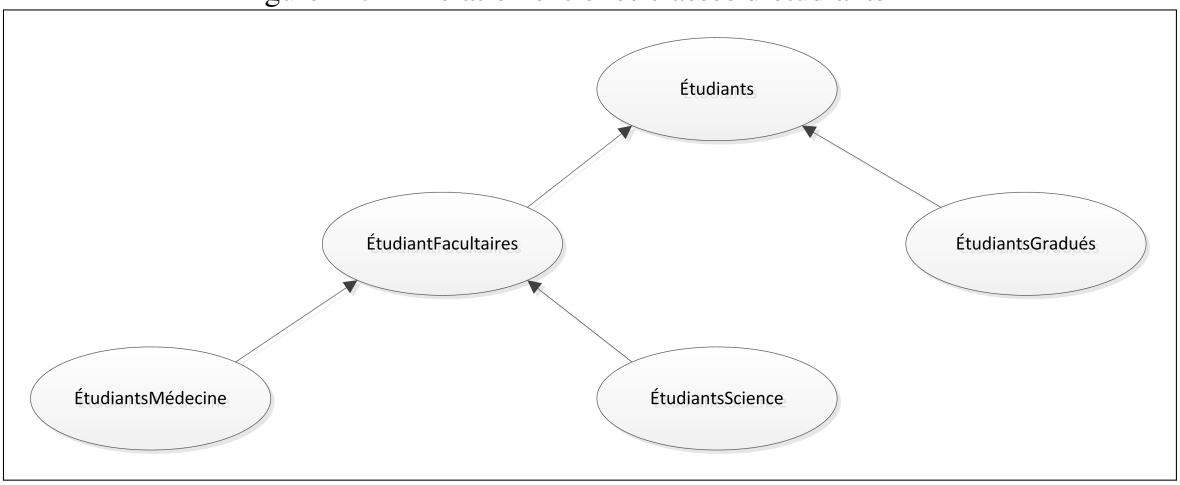


tableau III.I).

Tableau III.I – Opérateurs

Opérateur	Formule	Description
Union (disjonction)	EtudiantSci ⊔ EtudiantMed	L'individu est un étudiant en science ou un étudiant en médecine.
Intersection (conjonction)	EtudiantSci ⊓ EtudiantMed	L'individu est un étudiant en science et en médecine.
Négation	¬EtudiantSci	L'individu n'est pas un étudiant en science.
Existentiel sur restriction de valeur	∃estInscrit.Prog	Il existe un programme auquel l'individu est inscrit.
Universel sur restriction de valeur	∀estInscrit.Prog	L'individu est inscrit seulement à des programmes.

Une logique de description supportant ces cinq opérateurs peut utiliser l'approche des tableaux sémantiques comme méthode de preuve. La section suivante présente brièvement l'approche des tableaux sémantiques et son application dans notre logique à cinq opérateurs. Nous illustrerons ensuite cette approche avec deux exemples simples utilisant l'ontologie que nous venons de présenter.

Approche des tableaux sémantiques

L'approche des tableaux sémantiques est applicable dans le cas où les opérateurs de la logique étudiée sont accompagnés de règles de décomposition permettant de ramener

les énoncés à des formes simplifiées où il est trivial d'identifier si l'énoncé contient une contradiction. On parle alors de logiques monotones.

Dans le cas d'une logique monotone on peut déterminer si un énoncé est valide en décomposant sa négation jusqu'à obtenir un énoncé trivial. Si cet énoncé décomposé contient une contradiction, on en conclura alors que la négation de notre énoncé initial ne peut être vrai et donc que notre énoncé initial est toujours vrai, donc valide.

Dans le contexte des logiques de description deux règles concernant le TBox et le ABox viennent s'ajouter. Voici donc une explication des règles requises pour traiter les deux exemples des prochaines sections :

- *R1- Règle du ABox.* Comme on essaie de montrer si la négation de l'énoncé peut être satisfaite *dans le contexte de notre logique*, il faut prendre la conjonction de la négation de l'énoncé avec toutes les assertions du ABox et montrer que cette conjonction peut être satisfaite.
- *R2- Règle du TBox.* Il faut remplacer chaque concept composé par sa définition dans le TBox pour obtenir éventuellement un énoncé qui ne contient que des concepts de base. Nos exemples ne contiennent pas de cas où la définition d'une classe réfère à la classe elle-même (une boucle). Ces complexités sont traitées par l'approche des points-fixes mais les détails dépassent grandement le but de notre propos.
- *R3- Règle de la négation.* Il faut distribuer les négations jusqu'à ce qu'elles s'appliquent à des concepts de base. On utilise les règles bien connues de la logique :

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$
$$\neg\exists R.C \equiv \forall R.\neg C$$
$$\neg\forall R.C \equiv \exists R.\neg C$$

- *R4- Règle de la conjonction.* Pour que $A \wedge B$ puisse être satisfait alors A doit être satisfait et B doit être satisfait. Si un des deux contient une contradiction alors l'ensemble ne peut être satisfait.

- *R5- Règle de la disjonction.* Pour que $A \vee B$ puisse être satisfait alors A doit être satisfait ou bien B doit être satisfait. L'ensemble ne peut pas être satisfait seulement si les deux branches contiennent chacune une contradiction.
- *R6- Règle du quantificateur universel de restriction.* Si tous les individus liés à un individu de départ par un rôle donné sont d'une classe donnée et qu'un individu particulier est lié à notre individu de départ par ce rôle, alors notre individu particulier est de la classe en question. Formellement :

$$\forall R.C(x),\ R(x,y) \vdash C(y)$$

- *R7- Règle du quantificateur existentiel de restriction* S'il existe un individu, lié à un individu de départ par un rôle donné, alors on peut nommer cet individu avec un identifiant qui n'a pas été utilisé jusqu'à maintenant. Formellement :

$$\exists R.C(x) \vdash R(x,z),\ C(z)$$

Test d'appartenance à une classe

Figure III.5 – Test d'appartenance à une classe

(1) EtudiantGradue(Jacques)
(2) ¬EtudiantGradue(Jacques)
    ∧ estInscrit(Jacques,MaitriseInfo) ∧ ProgMaitrise(MaitriseInfo)
(3) ¬(∃estInscrit.ProgDoc(Jacques) ∨ ∃estInscrit.ProgMaitrise(Jacques))
    ∧ estInscrit(Jacques,MaitriseInfo) ∧ ProgMaitrise(MaitriseInfo)
(4) ∀estInscrit.¬ProgDoc(Jacques) ∧ ∀estInscrit.¬ProgMaitrise(Jacques)
    ∧ estInscrit(Jacques,MaitriseInfo) ∧ ProgMaitrise(MaitriseInfo)
(4*A*) ∀estInscrit.¬ProgDoc(Jacques) ∧ estInscrit(Jacques,MaitriseInfo) ⇒ ¬ProgDoc(MaitriseInfo)
(4*B*) ∀estInscrit.¬ProgMaitrise(Jacques) ∧ estInscrit(Jacques,MaitriseInfo) ⇒ ¬ProgMaitrise(MaitriseInfo)
(5) ∀estInscrit.¬ProgDoc(Jacques) ∧ ∀estInscrit.¬ProgMaitrise(Jacques)
    ∧ estInscrit(Jacques,MaitriseInfo) ∧ ProgMaitrise(MaitriseInfo)
    ∧ ¬ProgDoc(MaitriseInfo) ∧ ¬ProgMaitrise(MaitriseInfo)

Le premier exemple concerne le test d'appartenance d'un individu à une classe et constitue le test de base d'une opération de classification. On se servira de l'ontologie de la figure III.1 et on essaiera de savoir si Jacques est un étudiant gradué. Cela revient à vérifier si l'énoncé EtudiantGradue(Jacques) est un énoncé valide. Pour faire ce test le *reasoner* suivra une procédure équivalente à la figure III.5. Nous utiliserons les numéros d'étapes de la figure (x). Les références [Ry] renvoient aux règles de la section précédente.

Voici donc les étapes :

- on pose d'abord l'énoncé duquel on veut vérifier la validité (1) ;
- on prend ensuite la négation de cet énoncé (2) en lui associant les assertions contenues dans le ABox [R1] (nous nous limitons ici aux assertions pertinentes pour économiser l'espace). Si on réussit à satisfaire l'énoncé (2) on aura montré que (1) n'est pas valide car on aura trouvé un cas qui le contredit. Si on ne peut satisfaire l'énoncé (2) alors (1) ne peut être contredit et est donc valide ;
- on remplace ensuite dans (2) EtudiantGradue(Jacques) par sa définition dans la TBox [R2] pour obtenir l'énoncé (3) ;
- on distribue ensuite la négation [R3] et on inverse les quantificateurs [R3] pour obtenir (4). S'il est faux de dire que Jacques est inscrit à au moins un programme de doctorat alors tous les programmes auxquels Jacques est inscrit ne sont pas des programmes de doctorat ;
- on utilise ensuite en (4A) la règle du quantificateur universel [R6]. Si tous les programmes auxquels Jacques est inscrit ne sont pas des programmes de doctorat et Jacques est inscrit à la maitrise en mathématiques alors la maitrise en mathématiques n'est pas un programme de doctorat ;
- on fait la même chose en (4B) pour les programmes de maitrise ce qui amène à l'énoncé (5) ;
- mais l'énoncé (5) est impossible puisqu'il affirme qu'une maîtrise en informatique est et n'est pas un programme de maîtrise. Comme l'énoncé (5) est équivalent à l'énoncé (2) alors Jacques ne peut pas ne pas être un étudiant gradué, il est donc un étudiant gradué.

Test de subsomption

Pour répondre efficacement aux demandes de classification, un *reasoner* OWL commence toujours par construire un treillis de subsomption à partir de l'ontologie. On compare les classes deux à deux pour savoir si l'une est une sous-classe de l'autre et on optimise le processus en utilisant le treillis partiellement bâti pour éviter les comparaisons inutiles.

Pour procéder à la décomposition de l'énoncé, on doit avoir la signification en logique du fait qu'une classe soit incluse dans une autre classe (subsomption). On utilise la logique standard pour y arriver :

$$\begin{aligned} A \sqsubseteq B &\equiv \forall x(A(x) \Rightarrow B(x)) \\ &\equiv \forall x(\neg A(x) \vee B(x)) \end{aligned}$$

Supposons que nous sommes dans le processus de construction du treillis et que nous voulons savoir si EtudiantGradue est une sous-classe de Etudiant (voir figure III.6). Nous procéderons comme pour le test d'appartenance :

- En (1) on pose que EtudantGradue est une sous-classe de Etudiant.
- En (2) on prend la négation que l'on va essayer de satisfaire par la suite. Que les étudiants gradués soient inclus dans les étudiants revient à dire que tour tout étudiant x, x est un étudiant ou bien x n'est pas un étudiant gradué, ce qui donne (3) en conservant la négation.
- En (4) on distribue la négation [R3] et on obtient qu'il existe un étudiant gradué x qui n'est pas un étudiant. C'est ce x que l'on tente de trouver dans la suite.
- En (5) on remplace EtudiantGradue et Etudiant par leurs définitions [R2].
- En (6) on remplace Programme par sa définition [R2] et on inverse le quantificateur [R3] pour insérer la négation.
- En (7) on distribue la négation sur les différents types de programmes [R3].
- En (8) on applique la règle de la disjonction [R5] pour obtenir deux fils d'exécution

Figure III.6 – Test de subsomption

(1) $\mathsf{EtudiantGradue} \sqsubseteq \mathsf{Etudiant}$
(2) $\neg(\mathsf{EtudiantGradue} \sqsubseteq \mathsf{Etudiant})$
(3) $\neg\forall x(\neg\mathsf{EtudiantGradue}(x) \vee \mathsf{Etudiant}(x))$
(4) $\exists x(\mathsf{EtudiantGradue}(x) \wedge \neg\mathsf{Etudiant}(x))$
(5) $(\exists\mathsf{estInscrit.ProgDoc}(x) \vee \exists\mathsf{estInscrit.ProgMaitrise}(x)) \wedge \neg\exists\mathsf{estInscrit.Prog}(x)$
(6) $(\exists\mathsf{estInscrit.ProgDoc}(x) \vee \exists\mathsf{estInscrit.ProgMaitrise}(x))$
$\wedge\forall\mathsf{estInscrit}.\neg(\mathsf{ProgDoc} \vee \mathsf{ProgMaitrise} \vee \mathsf{ProgMed} \vee \mathsf{ProgSci})(x)$
(7) $(\exists\mathsf{estInscrit.ProgDoc}(x) \vee \exists\mathsf{estInscrit.ProgMaitrise}(x))$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgDoc}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgMaitrise}(x)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgMed}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgSci}(x)$
($A8$) $\exists\mathsf{estInscrit.ProgDoc}(x)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgDoc}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgMaitrise}(x)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgMed}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgSci}(x)$
($A9$) $\exists\mathsf{estInscrit.ProgDoc}(x) \wedge \mathsf{estInscrit}(x,z) \wedge \mathsf{ProgDoc}(z)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgDoc}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgMaitrise}(x)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgMed}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgSci}(x)$
($A10$) $\exists\mathsf{estInscrit.ProgDoc}(x) \wedge \mathsf{estInscrit}(x,z) \wedge \mathsf{ProgDoc}(z)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgDoc}(x) \wedge \neg\mathsf{ProgDoc}(z) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgMaitrise}(x)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgMed}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgSci}(x)$
($B8$) $\exists\mathsf{estInscrit.ProgMaitrise}(x)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgDoc}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgMaitrise}(x)$
$\wedge\forall\mathsf{estInscrit}.\neg\mathsf{ProgMed}(x) \wedge \forall\mathsf{estInscrit}.\neg\mathsf{ProgSci}(x)$

(A8) et (B8). On cherche à satisfaire une des deux branches.

- Dans la branche (A) on applique d'abord la règle du quantificateur existentiel [R7] sur une restriction de valeur (A9). S'il existe un programme de doctorat auquel x est inscrit alors créons un de ces programmes que l'on appelle z. x est inscrit à z et z est un programme de doctorat.
- En (A10) on applique la règle du quantificateur universel [R6] sur les restrictions de valeur comme à la section précédente. Si tous les programmes auxquels x est inscrit ne sont pas des programmes de doctorat et que x est inscrit à z alors z n'est pas un programme de doctorat.
- On a donc déduit que z est un programme de doctorat et que z n'est pas un pro-

gramme de doctorat, ce qui est impossible et donc la branche (A) ne peut être satisfaite.

- Par un raisonnement similaire la branche (B) est aussi impossible ce qui fait que (2) est impossible et donc que (1) est valide.

***

Cette annexe était un peu plus technique que le reste du document au niveau de la logique. Elle a cependant bien montré qu'il est possible d'effectuer des opérations de classement de façon automatique à partir de définitions de classes et de la description d'un état. Le langage OWL et les *reasoners* actuels supportent des opérateurs plus complexes que les cinq opérateurs de notre logique de base. Il demeure que l'approche présentée correspond à ce qui est effectué par ce genre d'outils.

Annexe IV

Architecture du prototype

Prototype

Comme nous l'avons déjà mentionné, un patron d'architecture est une recette pour la structure d'une application quand des conditions spécifiques sont remplies. Pour illustrer la faisabilité du patron, nous avons développé une application satisfaisant les conditions tout en suivant le patron suggéré.

Nous discutons du sujet traité par l'application puis de l'environnement et des outils utilisés. Les sections principales couvrent ensuite l'architecture du prototype et ses principales composantes. Finalement nous faisons quelques remarques sur les composante Client-OWL et Sérialiseur-OWL.

L'annexe V présente une exécution du prototype qui inclue des explications plus détaillées sur les modèles et les résultats intermédiaires.

Application

Résumons les aspects importants de notre application :

- Nous entrons des dossiers étudiants pour lesquels nous conservons quelques information générales (nom, code permanent).
- Nous inscrivons les étudiants dans l'un des quatre (4) programmes d'étude disponibles.
- Nous classifions les étudiants, en nous basant sur leur inscription, dans deux hiérarchies fonctionnelles : l'une par domaine d'étude (Médecine ou Science) et l'autre par niveau (Bacc ou Gradué).
- Chaque étudiant fait partie d'un scénario et la classification se fait scénario par scénario. Les scénarios permettent à plus d'un utilisateur de travailler simultanément le prototype.

Environnement et outils

Comme nous utilisons l'environnement C# / .NET dans nos mandats professionnels et que cet environnement peut être utilisé pour développer des applications d'entreprise, nous avons décidé de l'utiliser pour le prototype. L'environnement .NET contient un ORM adéquat appelé *Entity Framework* qui s'intègre facilement avec le SGBD relationnel SQL Server. Ces deux outils ont donc également été utilisés.

Un problème avec cette approche vient du fait que plusieurs outils traitant OWL sont développés en Java (le *reasoner* Pellet, l'outil de développement d'ontologies Protégé, le protocole d'accès aux *reasoners* OWL-Api, etc.). Nous avons résolu le problème en développant trois (3) composantes :

- Une librairie de classes C# représentant chacune des constructions de la syntaxe fonctionnelle de OWL.
- Un sérialiseur permettant de lire la syntaxe de Manchester de OWL et de produire une ontologie interne en utilisant la librairie de classes OWL. Ce sérialiseur comporte certains aspects particuliers discutés plus loin.
- Une composante agissant comme un client du protocole OWL-Link. Ce protocole est une proposition, non adoptée comme recommandation, déposée aux W3C par certains intervenants du milieu des logiques de description. L'objectif est d'avoir un protocole neutre permettant d'interagir avec un *reasoner*. Une des façons d'utiliser ce protocole est de faire des appels de procédures XML sur le protocole Http, ce que nous utilisons pour passer de l'environnement .NET à l'environnement Java pour accéder au *reasoner* Pellet.

Le *reasoner* utilisé est donc Pellet car il est disponible gratuitement et facilement pour des projets académiques et qu'il existe un *plugin* pour l'outil de développement Protégé qui permet de comparer les résultats des raisonnements.

L'environnement de développement d'ontologies Protégé a aussi été utilisé pour valider les ontologies dans les différentes syntaxes de OWL.

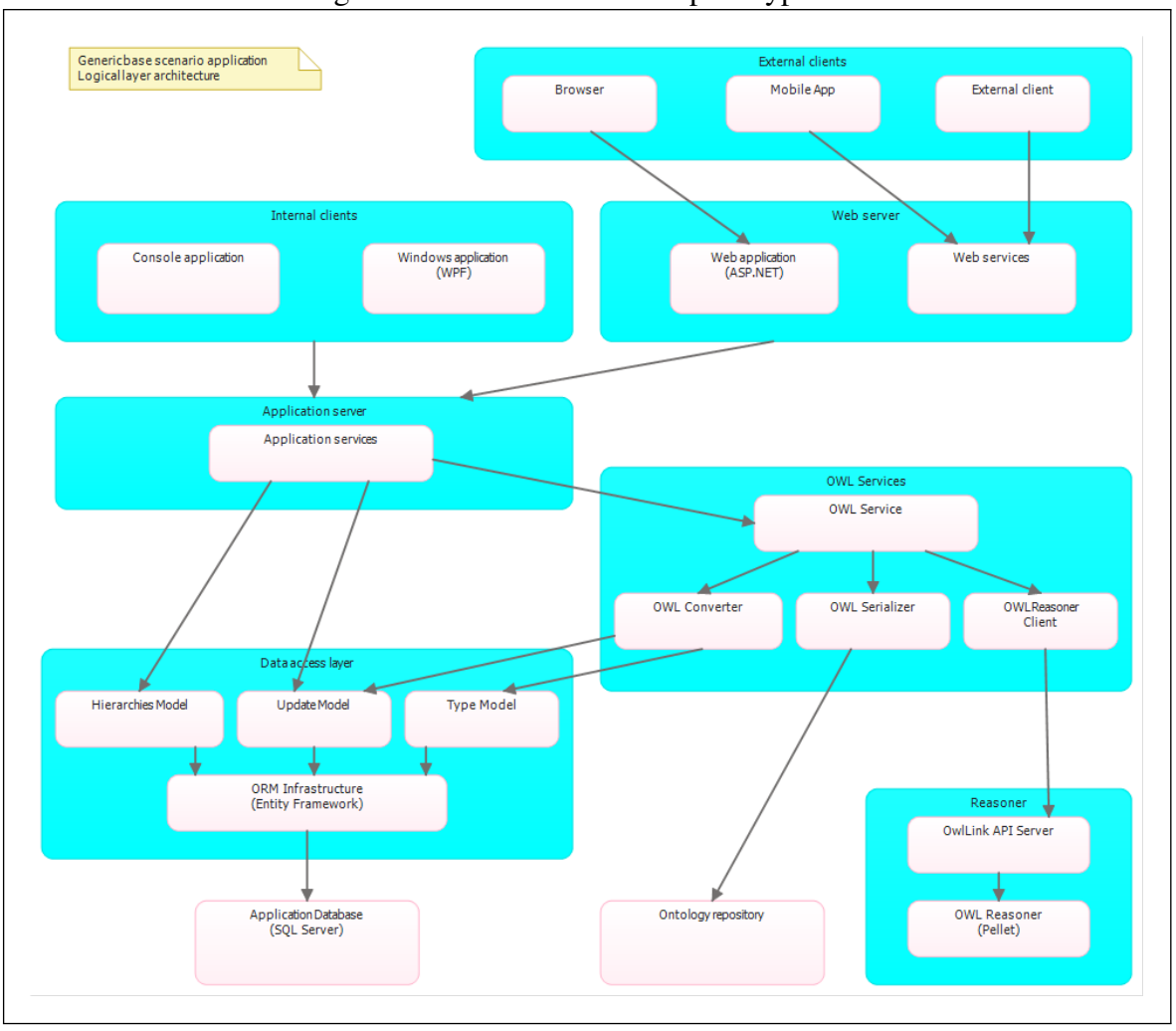
Notons également que nous avons toujours programmé en anglais d'où l'utilisation de termes anglais dans le texte des programmes ainsi que dans certains diagrammes. Une

copie du code source a été remis au directeur de recherche.

Architecture

La figure IV.1 décrit l'architecture par niveau (*layer architecture*) d'une application type qui implanterait le patron d'architecture. C'est une architecture typique à plusieurs niveaux (n-tiers).

Figure IV.1 – Architecture du prototype



Soulignons quelques remarques :

- La couche d'accès aux données (*data access layer*) est effectuée par le ORM mais avec trois (3) modèles d'accès distincts tel qu'il a été expliqué à la section 4.2.

- L'application appellera une couche générique pour effectuer la classification (OWL service). Cette couche communique à son tour avec la couche du *reasoner*, en plus de lire l'ontologie OWL-Manchester et d'interagir avec la base de données pour lire les objets et écrire les résultats. La couche OWL Service implante le cycle d'utilisation des logiques de description de façon générique.

Principales composantes

À partir de la description de l'architecture on peut identifier les principales composantes de l'application à développer (voir tableau IV.I).

Tableau IV.I – Principales composantes

	Composante	Rôles
1	Interface utilisateur	Mécanisme de saisie d'information et de commandes. Affichage des résultats des requêtes.
2	Serveur applicatifs	Création et mises à jour des étudiants Enclencher le processus de classification Lecture des hiérarchies fonctionnelles
3	Service OWL	Coordonner l'étape de classification
4	Convertisseur OWL	Lire les étudiants et leur inscription et produire les axiomes correspondants. Mettre à jour les champs de contrôle de la BD à partir de la classification.
5	Sérialiseur OWL	Interpréter une ontologie en syntaxe de Manchester pour produire une sérialisation OWL-XML pouvant être transmise au *reasoner*.
6	Client OWL-Link	Interfacer avec le *reasoner* via l'interface OWL-Link. Mener le dialogue permettant d'envoyer l'ontologie au *reasoner* et de reconstruire la classification résultante pour chaque hiérarchie.
7	*Reasoner*	Serveur HTTP pour recevoir les requêtes OWL-Link. Traduction OWL-Link en OWL-Api du côté Java Effectuer les tâches de raisonnement (Pellet via requêtes en OWL-Api)
8	Modèles d'accès aux données	Paramétrisation de l'accès à la base de données. Outil générique (pas de développement)
9	ORM - *Object Relationnal Mapper*	Accès à la base de données. Outil générique (pas de développement)

Voici quelques notes relatives à ces composantes :

- Dans notre cas la seule interface utilisateur développée est un site web.
- On a 3 modèles d'accès aux données : *a*) *Mise à jour* - pour la création des étudiants ; *b*) *Hiérarchique* - pour l'accès aux étudiants avec les hiérarchies fonctionnelles ; et *c*) *Types* - pour la mise à jour des champs de contrôle de la BD.
- Le convertisseur OWL est codé en dur dans le prototype. Le développement de sa version générique a été identifié au chapitre 7 comme une piste de recherche.
- Le sérialiseur OWL et le client OWL-Link sont expliqués plus loin..

Client OWL

Dans le corps du document, nous avons assumé qu'il était possible de demander à une *reasoner* d'effectuer une classification sur une ontologie et de nous retourner le résultat de cette classification dans un format nous permette de mettre à jour les champs de contrôle de la base de données, permettant ainsi au ORM de créer les objets dans la bonne classe pour chacune des hiérarchies.

L'annexe III a permis de montrer comment un *reasoner* peut effectivement classifier des individus dans des classes à partir des propriétés des individus et des définitions des classes. La présente section décrit avec plus de détails comment s'effectue le dialogue avec *reasoner* pour obtenir le résultat attendu.

La composante Client OWL-Link (*OWL-Link Reasoner Client*) de l'architecture du prototype (voir figure IV.1) réalise l'interface entre l'application et le *reasoner*. Une méthode de cette composante reçoit l'ontologie combinant les définitions des classes avec les propriétés des individus provenant de la base de données et retourne une structure décrivant la classe d'appartenance de chacun des objets dans chacune des hiérarchies.

La définition des classes contient :

- La définition, en OWL, de chacune des classes au dernier niveau de chacune des hiérarchies fonctionnelles où on veut classifier les objets. Ces classes forment une partition pour chaque hiérarchie.
- Une classe pour chacune des hiérarchies. Les classes formant la partition d'une hiérarchie sont définies comme des sous-classes de la classe représentant cette

hiérarchie.

- La classe spéciale dgc :hiérarchy. Les classes représentant les hiérarchies sont définies comme des sous-classes de cette classe spéciale.

Les systèmes de gestion des connaissances fonctionnent typiquement avec un système Tell / Ask. Un Tell ajoute ou modifie des éléments dans la base de connaissance alors qu'un Ask demande d'extraire des connaissance de la base de connaissance. Le langage OWL défini le contenu de la base de connaissance, dans le contexte des logiques de description, il sert donc essentiellement de contenu pour les opérations de Tell. Le langage OWL-Link est une extension à OWL qui spécifie :

- Une façon standard de demander la création et la destruction d'une ontologie.
- La méthode de transmission des axiomes OWL (Tell).
- Le format des questions (Ask) que l'on peut poser au *reasoner*.
- Le format des réponse attendues pour chaque question (ex. une liste de classes ou une liste d'individus) qui ne sont pas définis en OWL.
- La transmission de paramètres généraux de l'ontologie (ex. utiliser ou non l'hypothèse des noms uniques).

La méthode principale du Client OWL-Link implante la séquence suivante en réalisant des appels au *reasoner* en utilisant le protocole OWL-Link :

1. Demande au *reasoner* de créer un base de connaissance.
2. Transmission des axiomes de l'ontologie contenant les définitions de classes et les propriétés des objets (Tell).
3. Demande (Ask) la liste des sous-classes de la classe dgc :hierarchy qui donne une classe pour chacune des hiérarchies (ex. sc :StudentType).
4. Pour chacune des hiérarchies, on demande (Ask) la liste des sou-classes qui correspondent à la partition de cette hiérarchie (ex. ex. sc :StudentGraduate).
5. Pour chacune des classes de chacune des hiérarchies, on demande (Ask) la liste des individus membres de cette classe (ex. sc :SIN101).
6. On libère la base de connaissance.
7. On construit la structure par hiérarchie décrivant l'appartenance des individus à chacune des classes de chaque hiérarchie et on retourne le résultat.

Sérialiseur OWL

Tel que spécifié précédemment, la définition des classes est codée en utilisant la syntaxe de Manchester du langage OWL (voir annexe II). Il a fallu développer, dans le cadre du prototype, une composante pour lire une ontologie en syntaxe de Manchester et la traduire dans le format OWL interne du prototype car une telle composante n'existait pas dans l'environnement .NET.

Cette composante s'est finalement montrée assez complexe à développer car la syntaxe de Manchester contient des ambiguïtés au niveau des types d'objets et des propriétés. Par exemple dans la définition suivante

```
Class : Parent
    EquivalentTo : hasChild some Person
```

la propriété hasChild peut être une propriété-référentielle (ObjectProperty) ou une propriété-scalaire (DataProperty) et l'identifiant Person peut référer aussi bien à une classe (Class) qu'à une définition de données scalaire (DataRange).

Il faut donc faire une première analyse du fichier de définitions en utilisant une syntaxe intermédiaire pour identifier les types des propriétés de classes (ici hasChild et Person) pour ensuite traduire avec les bonnes expressions OWL.

Annexe V

Exécution du prototype

Les pages suivantes constituent une exécution de l'interface web du prototype développé dans le cadre de ce mémoire.

Cette interface est une série de pages HTML qui expliquent le processus du patron d'architecture de façon moins formelle que dans le texte du mémoire. Le but est de publier éventuellement le site pour permettre au lecteur d'avoir une idée du sujet sans avoir à lire l'ensemble du mémoire.

À travers les pages des boutons permettent d'effectuer les opérations importantes ou de montrer des résultats intermédiaires. Les pages suivantes inclues donc également un exemple d'une ontologie de définition de classes, une extraction de propriétés de la base de données, etc. En ce sens, il s'agit d'un complément au texte du mémoire.

Démo - Logiques et applications d'entreprise

[Log In]

Acceuil | Démo | À propos

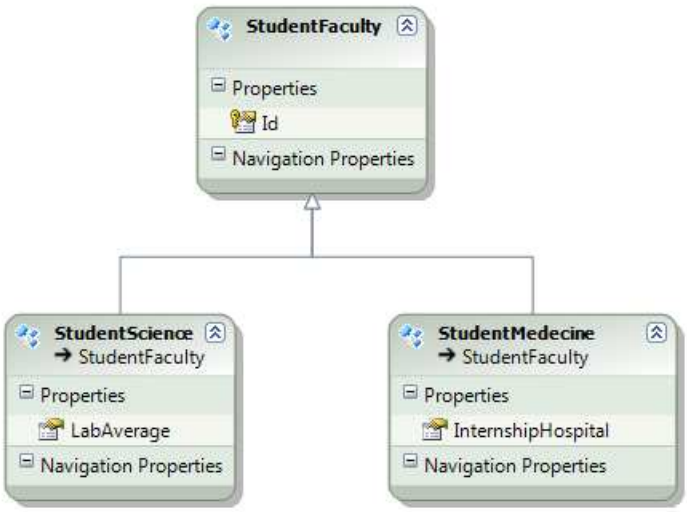
Page de la démonstration

Le problème

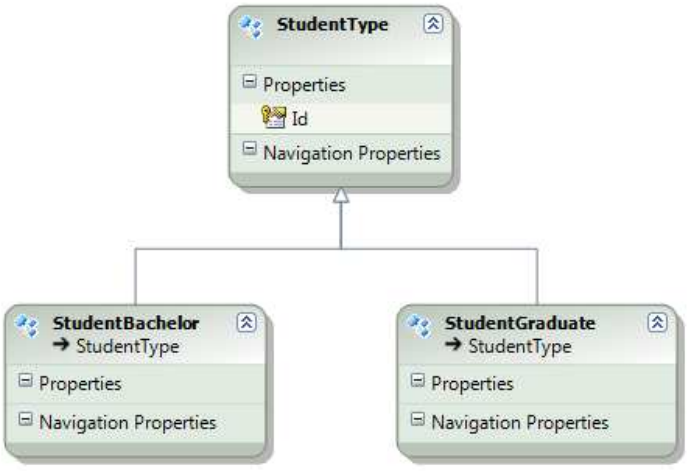
Le but de ce prototype est d'illustrer une approche permettant de contourner une limitation de l'approche orientée-objet à l'aide des logiques de description.

Le problème consiste à utiliser le polymorphisme simultanément sur plusieurs hiérarchies fonctionnelles.

Supposons que l'on veuille classifier des étudiants selon leur domaine d'étude. On aurait, par exemple, des étudiants en médecine et étudiants en science. D'où une hiérarchie par domaine d'étude.



D'un autre côté, si on voulait classifier les étudiant selon leur niveau on obtiendrait des étudiants de Bacc. et des étudiants gradués. D'où une hiérarchie par niveau d'étude.



Démo - Logiques et applications d'entreprise

[Log In]

Acceuil | Démo | À propos

Page de la démonstration

Les outils

Notre solution à l'exploitation simultanée d'objets dans plusieurs hiérarchies fonctionnelles utilisera de façon astucieuse et coordonnée deux outils existants.

Les ORM - Object Relational Mapper

Ces outils utilisent une correspondance (mapping) entre la structure d'une base de données relationnelle et un modèle object pour automatiser la lecture et l'écriture d'objets dans une base de données.

Toute la démonstration utilise l'environnement .Net de Microsoft. Le ORM de .Net s'appelle le Entity Framework que nous appellerons EF dans la suite.

La principale astuce sera d'utiliser plusieurs modèles objets différents définis sur une même base de données...

Les "reasoners" de logiques de description

Les reasoners sont des outils permettant d'effectuer du raisonnement automatique à partir d'ontologies qui suivent les règles de logiques de description supportées. Plus de détails dans la suite.

Le reasoner utilisé dans cette démo est Pellet.

La principale force des reasoners est la facilité avec laquelle ils réalisent des classifications complexes d'objets dans des classes à partir de définition formelles de ces classes. On voudrait justement classifier des objets dans plusieurs hiérarchies...

Précédent Suivant

Démo - Logiques et applications d'entreprise

[Log In]

Acceuil | Démo | À propos

Page de la démonstration

Choix du scénario

La démonstration consistera à créer et classifier des étudiants pour ensuite utilser plusieurs hiérarchies d'étudiants simultanément.

Nous allons créer et manipuler de vraies données. Pour permettre à chaque utilisateur de la démo de travailler de façon indépendant nous allons associer nos étudiants à une "scénario".

Vous avez deux choix. Vous pouvez entrer un nouveau nom de scénario et cliquez sur le bouton soumettre soumettre

Soumettre

Ou bien choisir un scénario déjà existant dans la liste suivante.

base
abc
abd
abe

Les scénarios sont détruits périodiquement.

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

SAISIE DES ÉTUDIANTS

Voici la liste des étudiants de votre scénario

Nom	Code permanent	Programme
Jacques	SIN100	MasterComputerScience
Marie	SIN101	DoctorateOrthophony
Ginette	SIN102	BachelorMathematics
Martin	SIN103	BachelorNutrition

Vous pouvez maintenant entrer d'autres étudiants si désiré.

Nom : :

Code permanent : :

Programme : MasterComputerScience

Ajouter étudiant

Ou ajouter une liste d'étudiants par défaut pour gagner du temps

Ajouter étudiants par défaut

Précédent Suivant

Démo - Logiques et applications d'entreprise

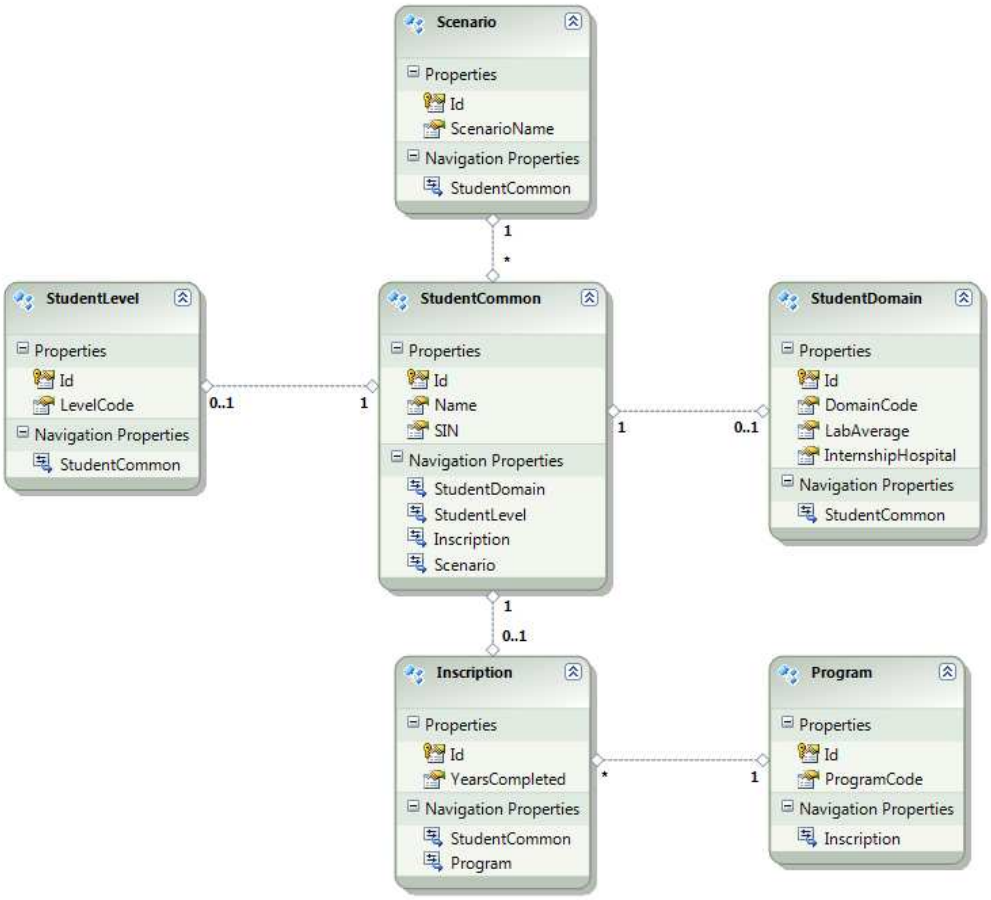
[Log In]

Acceuil | Démo | À propos

Page de la démonstration

Modèle de la base de données

Le modèle objet suivant corresponds aux tables de la base de données.



Le modèle est simple. On a une table d'étudiants (StudentCommon) qui sont regroupés en scénarions tel qu'expliqué précédemment. Chaque étudiant peut être inscrit à au plus 1 programme.

La seule particularité est qu'il y a en fait 3 tables d'étudiants :

- StudentCommon pour les propriétés générales de l'étudiant
- StudentLevel pour la hiérarchie par niveaux
- StudentDomain pour la hiérarchie par domaine

Nous verrons plus loin les mécanismes de synchronisation et l'utilité de cette organisation à première vue redondante.

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

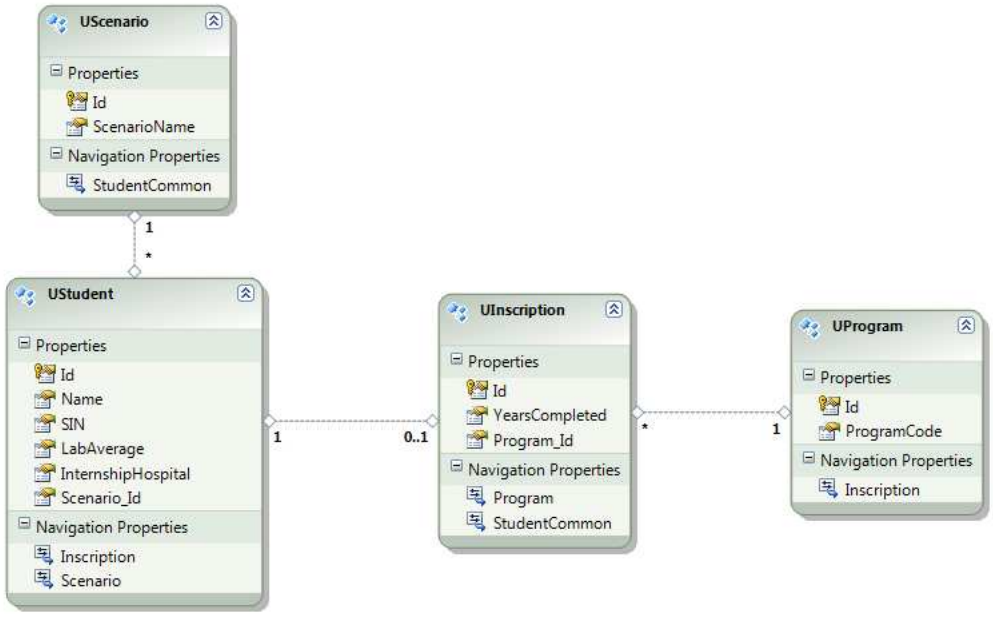
[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

MODÈLE EF DE MISE-À-JOUR

Lors de la création des étudiants, le modèle EF utilisé est le suivant.



Ce modèle ressemble au modèle de la base de données à l'exception des trois tables d'étudiants qui sont regroupées dans une seule classe (UStudent). Les ORM peuvent faire correspondre une classe du modèle objet à plusieurs tables physiques de la base de données. C'est le ORM qui synchronise les tables de la BD. Quand on créé une UStudent, un enregistrement est créé dans chacune des trois tables, rendant l'organisation physique "redondante" transparente pour l'application.

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

CLASSIFICATION DES ÉTUDIANTS

Vous pouvez maintenant procéder à la classification des étudiants

Classifier les étudiants

Avec une nouvelle grille d'étudiants incluant cette fois le code de domaine et le code de niveau.

Nom	Code permanent	Programme	Code niveau	Code domaine
Jacques	SIN100	MasterComputerScience	StudentGraduate	StudentScience
Marie	SIN101	DoctorateOrthophony	StudentGraduate	StudentMedecine
Ginette	SIN102	BachelorMathematics	StudentBachelor	StudentScience
Martin	SIN103	BachelorNutrition	StudentBachelor	StudentMedecine

La classification ne fait que populer le code de niveau (LevelCode) de la table StudentLevel et le code de domaine (DomainCode) de la table StudentDomain. Ce sont les valeurs de ces champs qui permettrons de créer les hiérarchies fonctionnelles d'objet.

L'initialisation de ces champs est le résultat du processus de classification effectué par le reasoner. Nous décrirons plus loin le processus utilisé pour populer ces champs. Assumons pour l'instant que nous pouvons les populer et voyons comment utiliser cette information dans les panneaux suivants.

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

POLYMORPHISME SIMULTANÉ SUR PLUSIEURS HIÉRARCHIES

Montrons maintenant que nous pouvons faire du polymorphisme simultané sur nos deux hiérarchies. Sélectionner un étudiant de la grille.

	Nom	Code permanent	Programme	Code niveau	Code domaine
Select	Jacques	SIN100	MasterComputerScience	StudentGraduate	StudentScience
Select	Marie	SIN101	DoctorateOrthophony	StudentGraduate	StudentMedecine
Select	**Ginette**	**SIN102**	**BachelorMathematics**	**StudentBachelor**	**StudentScience**
Select	Martin	SIN103	BachelorNutrition	StudentBachelor	StudentMedecine

Nous appelons la routine GetHtml() pour l'objet sélectionné sur les deux hiérarchies et montrons le résultat juste en dessous de ce paragraphe. Si vous ne voyez pas de texte en gras, vous n'avez pas sélectionné d'étudiant!

Ce paragraphe est produit par la méthode GetHtml() de la classe StudentBachelor

Le nom de l'étudiant est Ginette

Ce paragraphe est produit par la méthode GetHtml() de la classe StudentScience

La sélection d'une ligne de la grille déclenche l'exécution de la routine suivante

```
protected void PolyGrid_SelectedIndexChanged(object sender, EventArgs e)
{
    using (HierarchiesEntities context = new HierarchiesEntities())
    {
        string scenName = this.Session["scenario"].ToString();
        string selectedSIN = PolyGrid.SelectedRow.Cells[2].Text.Trim();
        StudentBase sBase = (from s in context.StudentBaseSet
                             where s.SIN == selectedSIN
                             && s.Scenario.ScenarioName == scenName
                             select s).FirstOrDefault();
        this.PolyLiteral.Text = sBase.StudentLevel.GetHtml() + sBase.StudentDomain.GetHtml();
    }
}
```

Les méthodes GetHtml() des différentes classes ont l'allure suivante (ici pour StudentGraduate) :

```
public override string GetHtml()
{
            return "<p>Ce paragraphe est produit par la méthode GetHtml()" +
                "  de la classe StudentGraduate</p>"
            +"<p>Le nom de l'étudiant est " + this.StudentCommon.Name + "</p>";
}
```

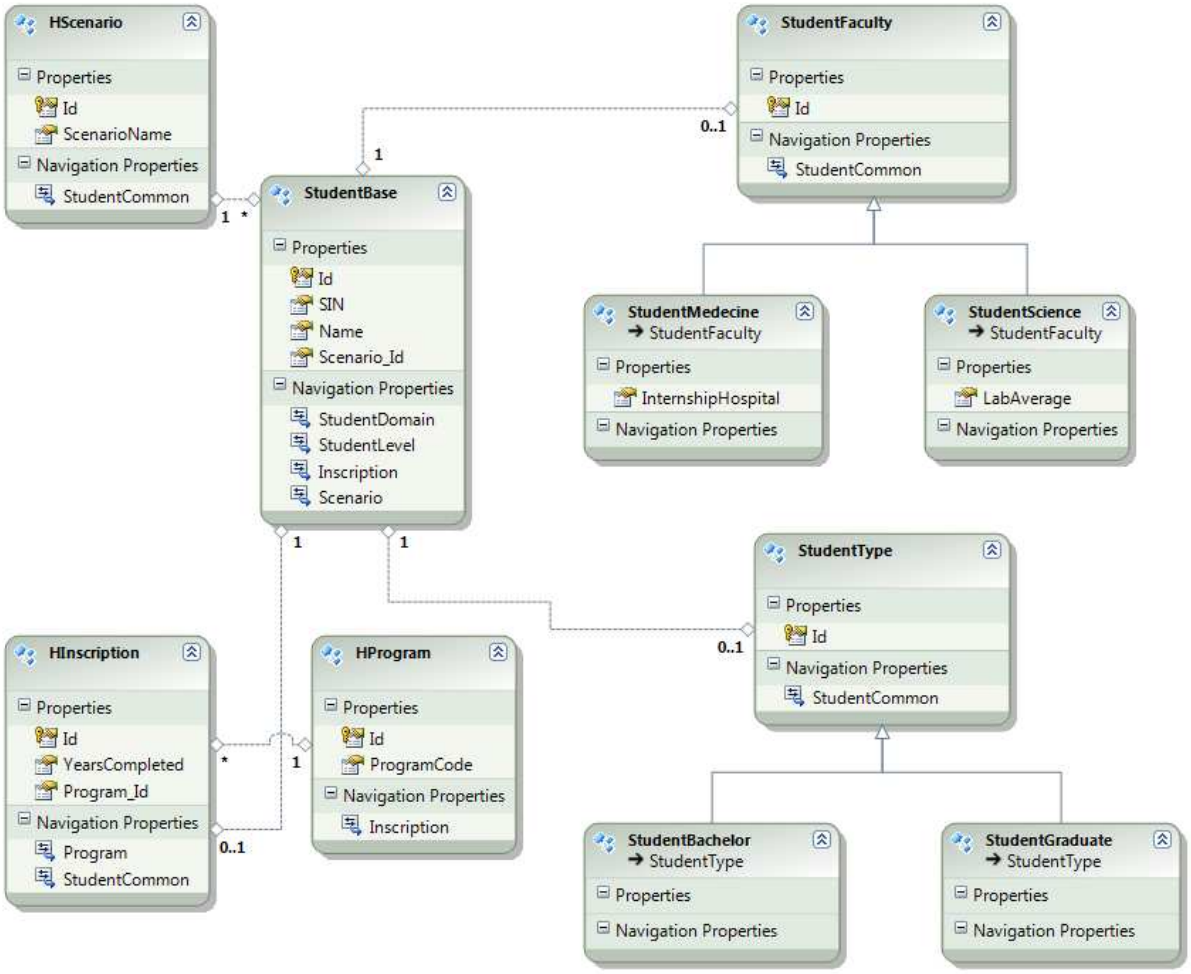
Sans entrer dans les détails, il est clair qu'aucun de ces traitements ne teste le programme auquel est inscrit l'étudiant sélectionné, ni même le type des objets ou les codes de niveau ou de domaine. Il s'agit donc bien de code polymorphe appliqué séparemment aux deux hiérarchies. La clé est dans le modèle EF **HierarchiesEntities** que nous expliquerons au panneau suivant. Le but ici est de voir que le problème mentionné au début de la démonstration est effectivement résolu, même si le comportement polymorphe présenté est simpliste. La même recette peut être appliquée à des hiérarchies et des comportements arbitrairement complexes.

Démo - Logiques et applications d'entreprise

Page de la démonstration

Modèle EF hiéarchique

Le modèle objet utilisé pour obtenir le polymorphisme est différent de celui utilisé précédemment pour la création des étudiants.



Les ORM doivent faire correspondre des modèles objet à des tables relationnelles. Le principal problème provient du fait que le concept d'héritage n'existe pas dans le modèle relationel. Plusieurs stratégie sont possibles dont la stratégie TPH - Table Per Hierarchy, qui consiste à utiliser une seule table pour conserver les objets de toutes les classes d'une hiérarchie.

La stratégie TPH est utilisée pour faire correspondre la hiérarchie des niveaux (StudentType, StudentBachelor et StudentGraduate) à la table StudentLelvel et la hiérarchie des domaines (StudentFaculty, StudentMedecine, StudentScience) à la table StudentDomain. La classe StudentBase quand à elle corresponds à la table StudentCommon.

Comme plusieurs classes correpondent à la même table, le ORM doit déterminer dans quelle classe créer l'objet lorsque celui-ci est lu de la table. On utilise des conditions dans le mapping du ORM. Le plus souvent une valeur spécifique d'un champ détermine l'appartenantce à une classe. C'est ainsi que de code de domaine et le code de niveau sont utilisés pour instancier les objets dans les bonnes classes à la lecture.

À titre d'information, ceci est le mapping de la table StudentLevel dans le cas du modèle des hiérarchies.

```
<EntitySetMapping Name="StudentLevel">
<EntityTypeMapping TypeName="IsTypeOf(HierarchiesModel.StudentType)">
    <MappingFragment StoreEntitySet="StudentLevel">
    <ScalarProperty Name="Id" ColumnName="Id" />
    </MappingFragment>
</EntityTypeMapping>
<EntityTypeMapping TypeName="IsTypeOf(HierarchiesModel.StudentBachelor)">
    <MappingFragment StoreEntitySet="StudentLevel">
    <ScalarProperty Name="Id" ColumnName="Id" />
    <Condition ColumnName="LevelCode" Value="StudentBachelor" />
    </MappingFragment>
</EntityTypeMapping>
<EntityTypeMapping TypeName="IsTypeOf(HierarchiesModel.StudentGraduate)">
    <MappingFragment StoreEntitySet="StudentLevel">
    <ScalarProperty Name="Id" ColumnName="Id" />
    <Condition ColumnName="LevelCode" Value="StudentGraduate" />
    </MappingFragment>
</EntityTypeMapping>
</EntitySetMapping>
```

Des outils existent pour paramétrer facilement de tels mappings.

Précédent Suivant

Démo - Logiques et applications d'entreprise

[Log In]

Acceuil | Démo | À propos

Page de la démonstration

Cycle d'utilisation des logiques de description

On a vu jusqu'à maintenant qu'une utilisation astucieuse d'un ORM peut permettre d'utiliser simulanément plusieurs hiérarchies fonctionnelles, en autant que l'on puisse populer les champs permettant au ORM d'instancier les objets dans les bonnes classes à la lecture.

Le mémoire de maîtrise auquel cette démonstration est attachée décrit comme réaliser ce travail de classification à l'aide d'un reasoner de logiques de description grace au "cycle d'utilisation des logiques de description". Il s'agit d'un processus en cinq (5) étapes :

1. Convertir les données existantes en OWL
2. Définir les hiérarchies cibles en OWL
3. Combiner les ontologies
4. Effectuer la classification sur l'ensemble à l'aide d'un reasoner
5. Retourner les résultats dans la base de données

Toutes ces étapes sont implantées dans l'opération de classification des étudiants que nous avons effectué précédemment.

Les prochains panneaux présenteront certains livrables intermédiaires de ce processus pour donner une meilleure idée de l'utilisation des logiques de description.

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

1- EXTRAIRE ASSERTIONS ET CONVERSION EN OWL

L'extraction utilise le modèle objet de MAJ déja utilisé pour la création des étudiants. Pour chaque étudiant du scénario demandé on produit deux axiomes: un pour déclarer l'existence de l'individu qui correspond à l'objet et un pour indiquer le programme auquel est inscrit l'étudiant. Cliquer le bouton suivant pour obtenir les axiomes créés pour votre scénario.

Extraction des axiomes

```
<?xml version="1.0" encoding="utf-16"?>
<list>
     <owl:Declaration xmlns:owl="http://www.w3.org/2002/07/owl#">
          <owl:NamedIndividual abbreviatedIRI="sc:SIN100" />
     </owl:Declaration>
     <owl:ObjectPropertyAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
          <owl:ObjectProperty abbreviatedIRI="sc:isRegistered" />
          <owl:NamedIndividual abbreviatedIRI="sc:SIN100" />
          <owl:NamedIndividual abbreviatedIRI="sc:MasterComputerScience" />
     </owl:ObjectPropertyAssertion>
     <owl:Declaration xmlns:owl="http://www.w3.org/2002/07/owl#">
          <owl:NamedIndividual abbreviatedIRI="sc:SIN101" />
     </owl:Declaration>
     <owl:ObjectPropertyAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
          <owl:ObjectProperty abbreviatedIRI="sc:isRegistered" />
          <owl:NamedIndividual abbreviatedIRI="sc:SIN101" />
          <owl:NamedIndividual abbreviatedIRI="sc:DoctorateOrthophony" />
     </owl:ObjectPropertyAssertion>
     <owl:Declaration xmlns:owl="http://www.w3.org/2002/07/owl#">
          <owl:NamedIndividual abbreviatedIRI="sc:SIN102" />
     </owl:Declaration>
     <owl:ObjectPropertyAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
          <owl:ObjectProperty abbreviatedIRI="sc:isRegistered" />
          <owl:NamedIndividual abbreviatedIRI="sc:SIN102" />
          <owl:NamedIndividual abbreviatedIRI="sc:BachelorMathematics" />
     </owl:ObjectPropertyAssertion>
     <owl:Declaration xmlns:owl="http://www.w3.org/2002/07/owl#">
          <owl:NamedIndividual abbreviatedIRI="sc:SIN103" />
     </owl:Declaration>
     <owl:ObjectPropertyAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
          <owl:ObjectProperty abbreviatedIRI="sc:isRegistered" />
          <owl:NamedIndividual abbreviatedIRI="sc:SIN103" />
          <owl:NamedIndividual abbreviatedIRI="sc:BachelorNutrition" />
     </owl:ObjectPropertyAssertion>
</list>
```

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

2- DÉFINIR LES HIÉRARCHIES CIBLES

Le reasoner a besoin que l'on définisse les hiérarchies de façon formelle pour pouvoir classifier les objets extraits de la base de données. Cette démonstration supporte la syntaxe de Manchester du langage OWL. Cliquez pour voir la définition de nos hiérarchies d'étudiants.

Définition des hiérarchies

```
Prefix: sc: <http://www.dogico.com/onto/StudentsClassification#>
Prefix: dgc: <http://ontologies.dogico.com/2012/01/owlservices#>

Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: dc: <http://purl.org/dc/elements/1.1/>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: xml: <http://www.w3.org/XML/1998/namespace>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Prefix: owl: <http://www.w3.org/2002/07/owl#>


Ontology: <http://www.dogico.com/ontologies/2011/12/StudentsClassification>

ObjectProperty: sc:isRegistered

Class: sc:Programs

Class: sc:DoctorateProgram
    SubClassOf:
        sc:Programs

Class: sc:MasterProgram
    SubClassOf:
        sc:Programs

Class: sc:BachelorProgram
    SubClassOf:
        sc:Programs

Class: sc:MedecineProgram
    SubClassOf:
        sc:Programs

Class: sc:ScienceProgram
    SubClassOf:
        sc:Programs

Class: dgc:Hierarchy

Class: sc:StudentFaculty
        SubClassOf:
                dgc:Hierarchy

Class: sc:StudentScience
        SubClassOf:
                sc:StudentFaculty
```

```
    EquivalentTo:
        sc:isRegistered some sc:ScienceProgram

Class: sc:StudentMedecine
        SubClassOf:
                sc:StudentFaculty
    EquivalentTo:
        sc:isRegistered some sc:MedecineProgram

Class: sc:StudentType
        SubClassOf:
                dgc:Hierarchy

Class: sc:StudentGraduate
        SubClassOf:
                sc:StudentType
    EquivalentTo:
        (sc:isRegistered some sc:DoctorateProgram)
         or (sc:isRegistered some sc:MasterProgram)

Class: sc:StudentBachelor
        SubClassOf:
                sc:StudentType
        EquivalentTo:
                sc:isRegistered some sc:BachelorProgram

Individual: sc:MasterComputerScience
    Types:
        sc:ScienceProgram,
        sc:MasterProgram

Individual: sc:DoctorateOrthophony
    Types:
        sc:DoctorateProgram,
        sc:MedecineProgram

Individual: sc:BachelorMathematics
    Types:
        sc:ScienceProgram,
                sc:BachelorProgram

Individual: sc:BachelorNutrition
    Types:
        sc:MedecineProgram,
                sc:BachelorProgram
```

Nous avons développé un traducteur de la syntaxe de Manchester vers la sérilaisation XML du langage OWL car aucun outil de ce type n'existe en C# / .Net. Si désiré, vous pouvez voir le résultat de cette traduction en cliquant le bouton suivant :

Traduction Xml des définitions

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

3- COMBINAISON DES AXIOMES

À l'étape 3 on doit combiner les axiomes extraits de la base de données avec la définition des hiérarchies.

En OWL on peut simplement concaténer les axiomes des deux ontologies pour obtenir le résultat souhaité.

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

4- CLASSIFIER À L'AIDE DU REASONER

La prochaine étape consiste à envoyer l'ontologie combinée au reasoner et à lui demander de classifier les objets.

Le reasoner utilisé est Pellet. Il s'agit d'un programme Java implantant un API (Application Programmin Interface) en Java (le Owl-Api). Comme nous développons en C# / .Net nous avons utilisé un autre API, OwlLink, pour accéder au reasoner. OwlLink inclu un processus Java qui répond à des requêtes HTTP formattées selon un language Xml particulier et qui transforme ces requêtes selon le Owl-Api compris par Pellet. Nous avons développé une composante cliente, en C# / .Net, qui interagit avec le serveur OwlLink à travers des requêtes HTTP duement formattées.

OwlLink suit une structure Tell / Ask. On fournit des axiomes (Tell) et on pose ensuite des questions (Ask). On obtient des réponses à nos questions comme des listes de classes ou d'objets selon la question posée.

Notre dialogue avec le reasoner prend la forme suivante :

1. Demande pour créer une base de connaissance
2. Transmission des axiomes combinés produits à l'étape précédente (Tell)
3. Demande (Ask) de la liste des sous-classes de la classe dgc:hierarchy qui nous donne une classe représentant chacune des hiérarchies (ex. sc:StudentType).
4. Pour chaque hiérarchie, on demande la liste des sous-classes qui correspondent aux feuilles de la hiérarchies (ex. sc:StudentGraduate).
5. Pour chaque feuille de chaque hiérarchie, on demande au reasoner d'énumérer les individus membres de cette classe (ex. sc:SIN101).
6. Demande de libération de la base de connaissance.

Comme on peut poser plusieurs questions dans une même demande on peut effectuer tout le processus en trois échanges avec le serveur OwlLink.

On prends ensuite toutes ces réponses pour former la structure des groupes d'axiomes déduits par le reasoner. Cliquer le bouton suivant pour obtenir le format de réponse.

Extraction des axiomes

```
<?xml version="1.0" encoding="utf-16"?>
<groups>
     <group>
          <owl:Class abbreviatedIRI="sc:StudentType" xmlns:owl="http://www.w3.org/2002/07/owl#" />
          <axioms>
               <owl:ClassAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
                    <owl:Class abbreviatedIRI="sc:StudentBachelor" />
                    <owl:NamedIndividual abbreviatedIRI="sc:SIN102" />
               </owl:ClassAssertion>
               <owl:ClassAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
                    <owl:Class abbreviatedIRI="sc:StudentBachelor" />
                    <owl:NamedIndividual abbreviatedIRI="sc:SIN103" />
               </owl:ClassAssertion>
               <owl:ClassAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
                    <owl:Class abbreviatedIRI="sc:StudentGraduate" />
                    <owl:NamedIndividual abbreviatedIRI="sc:SIN100" />
               </owl:ClassAssertion>
               <owl:ClassAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
                    <owl:Class abbreviatedIRI="sc:StudentGraduate" />
                    <owl:NamedIndividual abbreviatedIRI="sc:SIN101" />
               </owl:ClassAssertion>
          </axioms>
     </group>
     <group>
          <owl:Class abbreviatedIRI="sc:StudentFaculty" xmlns:owl="http://www.w3.org/2002/07/owl#" />
          <axioms>
               <owl:ClassAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
                    <owl:Class abbreviatedIRI="sc:StudentScience" />
                    <owl:NamedIndividual abbreviatedIRI="sc:SIN102" />
               </owl:ClassAssertion>
```

```
                    <owl:ClassAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
                         <owl:Class abbreviatedIRI="sc:StudentScience" />
                         <owl:NamedIndividual abbreviatedIRI="sc:SIN100" />
                    </owl:ClassAssertion>
                    <owl:ClassAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
                         <owl:Class abbreviatedIRI="sc:StudentMedecine" />
                         <owl:NamedIndividual abbreviatedIRI="sc:SIN103" />
                    </owl:ClassAssertion>
                    <owl:ClassAssertion xmlns:owl="http://www.w3.org/2002/07/owl#">
                         <owl:Class abbreviatedIRI="sc:StudentMedecine" />
                         <owl:NamedIndividual abbreviatedIRI="sc:SIN101" />
                    </owl:ClassAssertion>
               </axioms>
          </group>
     </groups>
```

Les groupes correspondent aux différents hiérarchies.

Précédent Suivant

DÉMO - LOGIQUES ET APPLICATIONS D'ENTREPRISE

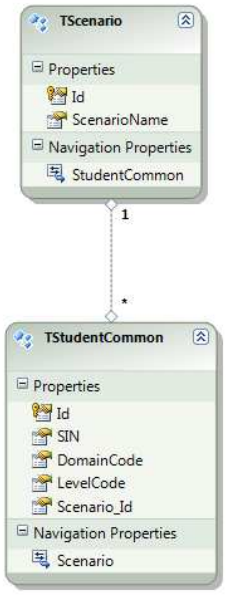
[Log In]

Acceuil | Démo | À propos

PAGE DE LA DÉMONSTRATION

5- RETOURNER LES RÉSULTATS À LA BASE DE DONNÉES

Pour mettre à jour la base de données à partir des axiomes déduits on utilise un troisième modèle de classes EF :



La classe TStudentCommon correspond aux trois tables physiques d'étudiants et ne permets de mettre à jour que le code de domaine et le code de niveau.

La mise è jour des champs code de domaine et code de niveau complète l'opération de classification.

Précédent Suivant