ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

THÈSE PRÉSENTÉE À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DU
DOCTORAT EN GÉNIE
Ph.D.

PAR
Abdelilah KAHLAOUI

MÉTHODE POUR LA DÉFINITION DES LANGAGES DÉDIÉS BASÉE SUR LE MÉTAMODÈLE ISO/IEC 24744

MONTRÉAL, LE 16 JUIN 2011

CETTE THÈSE A ÉTÉ EVALUÉE

PAR UN JURY COMPOSÉ DE :

M. Alain Abran, directeur de thèse
Département du génie logiciel et des technologies de l'information à l'École de technologie supérieure

M. Pierre Bourque, président du jury
Département du génie logiciel et des technologies de l'information à l'École de technologie supérieure

M. Roger Champagne, membre du jury
Département du génie logiciel et des technologies de l'information à l'École de technologie supérieure

M. Chadi El-Zammar, examinateur externe
Ericsson Canada

ELLE A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 9 JUIN 2011

A L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

À la mémoire du professeur Éric Lefebvre

REMERCIEMENTS

Mes remerciements les plus sincères vont au professeur Alain Abran, mon directeur de thèse, d'avoir accepté de diriger ce travail de recherche. Je lui suis très reconnaissant pour sa disponibilité, ses conseils, son soutien indéfectible, sa patience et sa compréhension. Je me considère très chanceux d'avoir été encadré par un directeur aussi généreux sur le plan humain que scientifique.

J'aurais aimé exprimer également mes remerciements les plus profonds au professeur Éric Lefebvre, mon codirecteur de thèse, qui m'a guidé et soutenu avec un grand dévouement pendant toutes les années de travail sur cette thèse. Malheureusement, je ne peux plus exprimer que ma profonde tristesse pour son décès. Je n'oublierai jamais les efforts qu'il a faits pour continuer à diriger cette thèse, même pendant des périodes très difficiles de sa vie.

Je tiens également à remercier les membres du jury : le professeur Pierre Bourque pour avoir accepté la présidence du jury, le professeur Roger Champagne d'avoir accepté d'être un membre de ce jury et Monsieur Chadi El-Zammar d'avoir accepté d'être un membre externe à ce jury.

Finalement, Je tiens à remercier toutes les personnes qui ont contribué de près ou de loin à la réalisation de ce travail.

Merci à tous.

MÉTHODE POUR LA DÉFINITION DES LANGAGES DÉDIÉS BASÉE SUR LE MÉTAMODÈLE ISO/IEC 24744

Abdelilah KAHLAOUI

RÉSUMÉ

Au cours des dernières années, il y a eu un intérêt croissant pour les langages dédiés (*Domain Specific Languages* (DSL)). Cet intérêt est motivé par l'émergence d'approches telles que l'ingénierie dirigée par les modèles, l'architecture dirigée par les modèles, les lignes de produits logiciels (SPL), les usines à logiciels et le développement dirigé par les modèles. Alors qu'au fond l'objectif de ces approches est d'élever le niveau d'abstraction du développement logiciel et d'augmenter le degré d'automatisation en utilisant des modèles de domaine précis et facilement exploitables par les machines, on constate que ces approches manquent de langages capables de produire de tels modèles et qu'elles sont toujours à la recherche de solutions pour mieux soutenir le développement selon ce nouveau paradigme. À cet égard, beaucoup de spécialistes considèrent les langages dédiés comme une solution capable d'aller au delà des modèles limités à la documentation et de produire des modèles précis prêts à être traités automatiquement par la machine.

Les langages dédiés ont démontré un grand potentiel pour augmenter la productivité, améliorer la maintenabilité, élever le niveau d'abstraction, et produire des modèles exécutables. Toutefois, le développement de langages dédiés fiables et intègres est une activité difficile et coûteuse qui demande à la fois une connaissance du domaine et des compétences en développement des langages. Ainsi, l'établissement d'une infrastructure rendant le développement de DSL plus facile et plus accessible constituera une étape importante vers la concrétisation et la consolidation des approches dirigées par les modèles.

Afin de développer cette infrastructure, nous pensons que les efforts doivent être axés sur trois domaines principaux : 1) les processus qui permettent d'offrir une approche disciplinée en matière de développement des DSL, 2) les outils pour soutenir le développement et la maintenance de ces DSL et 3) les standards pour assurer l'unification du développement et l'interopérabilité entre les outils.

Cette thèse est une contribution au domaine des processus. Nous y proposons une méthode de développement de DSL basée sur la norme ISO/IEC 24744 (*Software Engineering-Metamodel for Development Methodologies - SEMDM*). La méthode est générée à partir du métamodèle décrit dans la norme. Elle décrit, entre autres, les activités et les tâches à exécuter lors du développement d'un DSL, les artefacts à manipuler (créer, utiliser ou modifier) et les personnes impliquées. La méthode fournit également, lorsque possible, des techniques et des lignes directrices expliquant comment certains éléments de la méthode peuvent être utilisés.

Mots-clés : langages dédiés, DSL, ingénierie dirigée par les modèles, développement dirigé par les modèles, développement dirigés par les langages, ingénierie des langages.

MÉTHODE POUR LA DÉFINITION DES LANGAGES DÉDIÉS BASÉE SUR LE MÉTAMODÈLE ISO/IEC 24744

Abdelilah KAHLAOUI

ABSTRACT

In the last few years, there has been an increasing interest in Domain Specific Languages (DSL). This interest was triggered by the emergence of approaches such as Model Driven Engineering (MDE), Model Driven Architecture (MDA), software product lines (SPL), software factories and Model Driven Software Development (MDSD). While the goal of these approaches is, basically, to raise the level of abstraction in software development beyond code by using precise, domain-focused models that can be easily processed by computer-based tools, they are all suffering from the lack of languages capable of producing such models and are looking for better solutions to support software development according to this new paradigm. In this regard, a number of specialists believe domain specific languages to be an interesting solution for producing models intended for development purposes rather than just for documentation purposes.

Domain specific languages have been proven efficient in increasing productivity, improving maintainability, raising the level of abstraction, and producing concise models. However, the design of high fidelity domain specific languages with appropriate abstractions and concepts to model the domain at hand is a hard task that requires from developers both domain knowledge and language development skills. Therefore, providing facilities to make DSL development easier is a significant step toward concretizing domain and model driven approaches.

We believe that in order to develop useful facilities for DSL development, efforts need to be focused on three major areas: 1) processes to provide a disciplined approach for DSL development, 2) tools to support language development and maintenance and 3) standards to allow languages and tools to be designed and developed in a unified and interoperable way.

This thesis is a contribution to the process area. It proposes a method for developing DSL based on the ISO/IEC 24744 metamodel (Software Engineering-Metamodel for Development Methodologies - SEMDM). It describes, among others, the activities and tasks to be executed during a DSL development project along with the artifacts to be manipulated (created, used or modified) and the people involved. It also provides, when possible, techniques and guidelines explaining how some method elements can be used.

Keywords : Domain Specific Languages, DSL, Model Driven Engineering, Model Driven Development, Language Driven Development, Language Engineering

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ANTLR	ANother Tool for Language Recognition.
API	Application Programming Interface
ASG	Abstract Syntax Graph
ASP	Active Server Pages
AST	Abstract Syntax Tree
CFG	Context-free grammar
CSS	Cascading Style Sheets
CSV	Comma-separated values
DDL	Database Design Language
DE	Domain Engineering
DI	Diagram Interchange
DML	Domain Modeling Language
DSEL	Domain Specific Embedded Language
DSL	Domain Specific Language
DSM	Domain Specific Modeling
DSML	Domain Specific Modeling Language
DSSA	Domain-Specific Software Architecture
DSVL	Domain Specific Visual Language
DSVML	Domain Specific Visual Modeling Language
DTD	Document Type Definition
EMF	Eclipse Modeling Framework

EMFT	Eclipse Modeling Framework Technology
FAST	Family-Oriented Abstraction, Specification, and Translation
FODA	Feature-Oriented Domain Analysis
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GMP	Graphical Modeling Project
GOPPRR	Graph, Object, Property, Port, Role and Relationship
GPL	General Purpose Language
HTML	HyperText Markup Language
IDE	Integrated Development Editor
IDE	Integrated Development Environment
IDM	Ingénierie dirigée par les modèles
JSP	JavaServer Pages
LPL	Lignes de produits logiciels
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MDT	Model Development Tools
MOF	Meta Object Facility
MSL	Meta Scripting Language
OCL	Object Constraint Language
ODM	Organization Domain Modeling
OMG	Object Management Group

PHP	Personal Home Page (Hypertext Preprocessor)
QVT	Query/View/Transformation
RUP	Rational Unified Process
SBVR	Semantics of Business Vocabulary and Rules
SEI	Software Engineering Institute
SEMDM	Software Engineering Metamodel for Development Methodologies
SPL	Software Product Line
SQL	Structured Query Language
SVG	Scalable Vector Graphics
SWT	Standard Widget Toolkit
sysML	Systems Modeling Language
T4	Text Template Transformation Toolkit
UML	Unified Modeling Language
URL	Uniform Resource Locator
VSIX	Visual Studio Integration Extension
WCL	Web Configuration Language
XHTML	eXtensible HyperText Markup Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XQuery	XML Query
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations
YACC	Yet Another Compiler Compiler

LISTE DES SYMBOLES ET UNITÉS DE MESURE

NomClass/*Kind: notation standard utilisée pour faire référence au patron « *Powertype* » formé par le powertype *NomClassKind* et le type partitionné *NomClass*.

INTRODUCTION

Contexte et enjeux

Bien que les progrès accomplis en génie logiciel soient incontestables, les solutions proposées, jusqu'à date, n'ont pas réussi à maîtriser, d'une manière satisfaisante, la complexité des systèmes informatiques. Les industriels et les chercheurs sont toujours à la recherche de solutions novatrices et viables pour faire face à cette problématique.

Actuellement, parmi les approches de développement logiciel qui s'imposent comme les solutions les plus prometteuses pour répondre aux défis actuels du logiciel, on trouve les approches dirigées par les modèles; notamment, l'ingénierie dirigée par les modèles (IDM), l'architecture dirigée par les modèles (MDA), les usines à logiciel (*Software Factories*), les lignes de produits logiciels, etc. L'idée générale de ces approches repose sur deux principes : **1)** monter en abstraction en systématisant l'utilisation des modèles au long du cycle de développement logiciel et **2)** consolider l'automatisation des activités de développement à travers des mécanismes de transformation de modèles et de génération de code. L'idée semble prometteuse sauf qu'il y a encore des difficultés qui préoccupent les industriels et les chercheurs et qui entravent l'activation et l'adoption de ces approches à une plus grande échelle. La plus importante de ces difficultés est la pénurie de langages de modélisation capables de produire des modèles précis et interprétables par les outils. Les langages de modélisation généralistes, comme UML, ont montré leur incapacité à fournir le niveau de précision demandé par les approches dirigée par les modèles (Greenfield et Short, 2004b).

La pénurie de langages de modélisation viables suscite un grand intérêt pour les langages dédiés (***D**omain **S**pecific **L**anguages* - DSL). Ces langages offrent des caractéristiques intéressantes qui s'alignent bien avec la vision des approches dirigées par les modèles. Ils se caractérisent par leur puissance expressive qui permet d'exprimer les solutions au niveau d'abstraction du domaine du problème traité en utilisant le vocabulaire et les termes utilisés

par les experts du domaine, ce qui permet à ces derniers de comprendre, valider, voire même modifier, les modèles écrits avec ces DSL. Ajoutons à cela les gains apportés par ces langages en termes de productivité et de qualité (Tolvanen et Rossi, 2003; Van Deursen et Klint, 1998).

L'intérêt pour les DSL s'est encore amplifié ces dernières années avec l'engagement des grands éditeurs d'outils de développement comme IBM et Microsoft à construire des outils pour soutenir le développement des DSL.

Les travaux de cette thèse se placent dans la mouvance des langages dédiés et plus particulièrement dans l'axe de leur processus de développement. Dans cette thèse nous proposons une méthode pour la définition des DSL, méthode qui sera basée sur la norme ISO/IEC 24744. Nous aimerions par ce travail rendre plus accessible le processus de définition des DSL et compléter les efforts qui se font au niveau des outils.

Problématique de la recherche

Le développement des langages dédiés est une activité difficile et coûteuse (Greenfield et Short, 2004b; Hudak, 1998) qui demande à la fois une connaissance du domaine et des compétences en développement des langages; peu de développeurs possèdent les deux. De plus, la spécialisation et l'unicité de ces langages implique la nécessité de construire des outils spécialisés et dédiés à ces langages, comme des éditeurs pour éditer les modèles et des outils d'analyse, de compilation et de génération de code. À ces difficultés vient s'ajouter la déficience des outils de métamodélisation et le manque de méthodes et de processus pour piloter le cycle de développement des DSL.

En analysant la problématique du développement des DSL, nous avons constaté qu'elle se manifeste sur trois dimensions : processus, outils et standards. Contrairement aux produits logiciels, les langages dédiés manquent encore de processus permettant de piloter leur cycle de développement, d'outils pour soutenir leur développement et de standards pour garantir la

gouvernance en termes de portabilité des DSL et d'interopérabilité entre les outils. Nous pensons que le traitement de cette problématique devrait se faire en respect de ces trois dimensions, et ce d'une façon équilibrée.

Processus : selon (Bezivin et Heckel, 2004), les langages (langages de programmation, de modélisation, DSL, etc.), comme le logiciel, ont besoin d'une discipline d'ingénierie pour capitaliser leur développement. Cette discipline offrira un support pour la définition, l'implémentation et la validation de ces langages. L'état de l'art montre que le développement des DSL s'est fait à date d'une manière ad hoc et qu'il manque de méthodes capables de piloter le cycle de développement. La qualité des DSL développés est donc, souvent, fonction de la compétence, de la créativité et de l'intuition de leurs développeurs.

Outils : le DSL est par définition un langage spécialisé qui se caractérise par sa propre syntaxe et sa propre sémantique. Cette caractéristique des DSL constitue à la fois leur force et leur faiblesse. D'une part cette spécialisation permet au DSL d'exprimer la solution de la manière la plus adaptée au problème traité. D'autre part, l'utilisation du DSL nécessite la création d'outils spécialisés qui lui sont dédiés afin de pouvoir éditer, analyser, compiler et déboguer ses modèles. Or, le développement de tels outils est une tâche complexe et coûteuse. Le développement d'une infrastructure rendant la construction de ces outils plus accessible et moins coûteuse constituera un grand pas vers la démocratisation des langages dédiés.

Standard : actuellement, l'incompatibilité entre les outils nuit à la portabilité des DSL. Les outils utilisent des métamodèles différents pour la définition des DSL (Bézivin, 2004). Par exemples Eclipse EMF utilise ECORE, XMF Mosaic utilise XMF, MetaEdit+ utilise GOPPRR. Cette diversité fait de la portabilité des DSL et de l'interopérabilité entre les outils un véritable défi. Pour y faire face, il va falloir développer des standards qui serviront de fondation au développement des outils et auxquels les constructeurs d'outils doivent se conformer afin d'assurer la portabilité des DSL et de faciliter l'interopérabilité entre les outils.

Comme déjà mentionné, des efforts considérables se font actuellement au niveau des outils et la majorité des grands éditeurs d'outils de développement offrent maintenant des outils pour soutenir le développement des DSL. On peut argumenter que ces outils n'ont pas encore atteint le niveau de maturité souhaité, mais, comme pour les outils de développement logiciel, nous pensons que les outils de développement des DSL vont prendre du temps avant d'atteindre le niveau de maturité nécessaire. Au niveau des processus, la littérature révèle que très peu d'effort a été consacré aux méthodes de développement des DSL (Mernik, Heering et Sloane, 2005; Thibault, Marlet et Consel, 1999).

Par cette recherche, nous souhaitons contribuer au développement des DSL en l'abordant sous l'axe des processus de développement. Nous croyons qu'une contribution dans cette direction complétera les efforts déployés au niveau des outils et aidera à mieux maitriser l'activité de la définition des DSL.

Objectifs de la recherche

Pour cette recherche nous avons fixé deux objectifs : un objectif principal et un objectif secondaire et exploratoire. L'objectif principal de la recherche consiste à développer une méthode de définition des langages dédiés, basée sur la norme ISO/IEC 24744, qui intègre les phases à suivre, les activités à exécuter, les artefacts à gérer et les acteurs impliqués dans un projet de définition de DSL. Nous croyons que la définition d'une telle méthode aidera à systématiser et à mieux maitriser le développement des DSL et, éventuellement, à promouvoir l'adoption des approches dirigées par les modèles.

L'implémentation des DSL (soit le développement des librairies pour implémenter la sémantique, la construction de compilateurs et de générateurs de code) est considérée hors de la portée de cette recherche.

L'objectif secondaire de la recherche porte sur l'utilisation et la qualité des DSL développés avec la méthode proposée. Dans ce deuxième volet de notre recherche nous explorons le sujet de la qualité des langages dédiés en vue d'identifier un ensemble d'attributs de qualité nous permettant d'évaluer les DSL ainsi définis.

Organisation de la thèse

Cette thèse est composée de neuf chapitres en plus de ce chapitre d'introduction suivis d'une conclusion et de trois annexes.

Le chapitre 1 présente les espaces technologiques utilisant les DSL. L'objectif de ce chapitre est de montrer les contextes d'utilisation des DSL et de comprendre les enjeux et les concepts autour de ces derniers.

Le chapitre 2 présente l'état de l'art dans le domaine des DSL. Nous présentons dans ce chapitre les fondements des DSL, leur processus de développement et les outils utilisés pour leur création.

Le chapitre 3 présente une description sommaire du métamodèle pour les méthodes de développement (SEMDM) défini par la norme ISO/IEC 24744.

Le chapitre 4 présente la méthodologie adoptée pour mener à terme cette recherche. Ce chapitre décrit les principales étapes à exécuter pour atteindre les objectifs fixés pour notre recherche.

Le chapitre 5 présente la version non normalisée de la méthode de définition des DSL qu'on propose dans le cadre de cette thèse.

Le chapitre 6 présente la version normalisée de la méthode en décrivant son processus de génération à partir des éléments définis par la norme ISO/IEC 24744:2007.

Le chapitre 7 présente les attributs de qualité à utiliser en tant que critères d'évaluation lors de la phase d'expérimentation et d'évaluation de la méthode. Le chapitre décrit aussi la technique qui a permis d'identifier ces attributs à partir des facteurs de succès des langages dédiés.

Le chapitre 8 présente l'étude de cas développée pour démontrer le bien fondé de la méthode et pour s'assurer de son applicabilité.

Le chapitre 9 présente nos observations sur la qualité des DSL définis au cours de l'étude de cas. Le chapitre parle aussi de l'apport de ces langages sur l'amélioration de la productivité et de la qualité des produits logiciels développés.

Le chapitre de conclusion dresse le bilan du travail de recherche, présente ses limites et identifie les perspectives de recherche futures.

A la fin de la thèse, trois annexes sont présentées. L'ANNEXE I présente le mappage entre le processus de développement logiciel RUP et le processus de développement des langages dédiés. L'ANNEXE II présente la modélisation de la méthode avec la notation ISO/IEC 24744. L'ANNEXE III présente le contexte général d'utilisation des trois DSL définis dans le cadre de l'étude de cas.

CHAPITRE 1

ESPACES TECHNOLOGIQUES EN RELATION AVEC LES DSL

1.1 Introduction

Les langages dédiés jouent un rôle important dans de nombreuses technologies. L'ingénierie dirigée par les modèles, les usines à logiciel et les lignes de produits logiciels sont des exemples des technologies qui mettent les DSL au cœur de leurs approches de développement. Le but de ce chapitre est d'introduire les espaces technologiques[1] (Kurtev, Bézivin et Aksit, 2002) en relation avec l'espace des langages dédiés afin de comprendre les enjeux et les concepts autour des DSL.

1.2 Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM) est un terme générique qui désigne un paradigme de développement logiciel favorisant l'utilisation systématique des modèles au cours du cycle de développement logiciel. Ce paradigme qui considère les modèles comme les principaux artefacts utilisés pour exprimer les intentions des développeurs vise à monter en abstraction et à automatiser davantage le processus de construction logiciel. Désormais, les modèles ne sont plus considérés du seul point de vue contemplatif où ils sont utilisés à des fins de documentation et de communication mais surtout d'un point de vue productif. Néanmoins, la création de modèles productifs et la transposition de cette nouvelle vision à la pratique restent encore un défi.

---

[1] Un espace technologique correspond à un contexte de travail caractérisé par un corpus de connaissances, un ensemble de concepts, d'outils et de compétences.

Selon Douglas C. Schmidt (Schmidt, 2006), les technologies IDM doivent se doter de : **1)** langages de modélisation dédiés (DSL) qui permettent de spécifier la structure, le comportement et les exigences d'une application dans un domaine donné et **2)** moteurs de transformation qui analysent certains des aspects des modèles, et qui synthétisent différents types d'artefacts à partir de ces modèles (e.g. code source, cas de test, descriptions de déploiement, etc.). Kent (Kent, 2002) ajoute qu'il faut spécifier également le processus utilisé pour coordonner le développement et l'évolution des modèles.

L'état de l'art montre que les technologies dirigées par les modèles manquent encore de techniques efficaces pour la transformation des modèles (Czarnecki et Helsen, 2003; 2006; Robert et Bernhard, 2007). L'OMG propose le standard QVT (OMG, 2008) pour spécifier les transformations de modèles mais l'implémentation de ce standard pose encore quelques problèmes (Sadilek et Wachsmuth, 2008).

1.3 L'architecture MDA

L'architecture dirigée par les modèles (*Model Driven Architecture*) est une marque déposée par le groupe OMG (*Object Management Group*) qui propose une implantation des principes de l'ingénierie des modèles autour des standards OMG (Frankel, 2003). MDA définit un cadre de développement logiciel (Kleppe, Warmer et Bast, 2003) dont le processus de développement consiste en deux activités principales : l'élaboration de modèles représentant le système à différents niveaux d'abstraction avec des langages bien définis et la définition des transformations qui définissent les règles de conversion entre modèles. Les éléments de base de ce Framework sont : les modèles, les langages, les règles de transformation et les outils de transformation (Kleppe, Warmer et Bast, 2003). Quelques définitions reliées à MDA sont maintenant présentées.

Modèle : présente une description abstraite d'un système (ou d'une partie du système) logiciel caractérisé par son niveau d'abstraction (i.e. niveau de spécificité à la plateforme) et son langage d'expression (Kleppe, Warmer et Bast, 2003). Les modèles indépendants de la

plateforme (*Platform Independent Model* – PIM) et les modèles spécifiques à la plateforme (*Platform Specific Model* – PSM) sont les principaux types de modèles manipulés par MDA.

PIM : modèle conceptuel caractérisé par son fort niveau d'abstraction. Un PIM est indépendant de la plateforme et de la technologie déployées pour l'implantation (Kleppe, Warmer et Bast, 2003). Il représente la solution logicielle d'un point de vue fonctionnel sans se préoccuper des détails techniques d'implantation. Théoriquement, le même modèle PIM peut être réutilisé pour générer des solutions pour diverses plateformes.

PSM **:** modèle décrivant une solution spécifique à une plateforme donnée. Comparé au modèle PIM, le PSM tient compte des spécificités et des détails techniques de la plateforme de génération.

Langage : un langage est caractérisé par une forme bien définie (syntaxe) et une sémantique claire et précise facilitant son interprétation par les outils de transformation et de génération de code. MDA n'exige pas l'utilisation d'UML comme langage de modélisation (Fuentes-Fernández et Vallecillo-Moreno, 2004), quoiqu'elle insiste sur l'utilisation de standards de modélisation afin d'assurer l'intégrité, la portabilité et l'interopérabilité.

Définition de transformation : ensemble de règles de transformation qui décrivent comment un modèle écrit dans un langage source peut être transformé en un modèle dans un langage cible en décrivant la manière dont les éléments du premier langage peuvent être transformés en un ou plusieurs éléments dans le langage cible (Kleppe, Warmer et Bast, 2003).

Outils de transformation : servent à effectuer les transformations des modèles selon les règles définies par les définitions de transformation (ex. moteurs de génération, interpréteurs, compilateurs, etc.).

1.4 Ligne de produits logiciels

Les définitions dans la littérature pour une ligne de produits logiciels se rapprochent notablement. En général, une ligne de produits logiciels est définie comme étant un ensemble de systèmes partageant un ensemble de caractéristiques communes répondant aux besoins spécifiques d'un domaine particulier (Clements et Northrop, 2002).

Du point de vue théorique, l'approche des lignes de produits logiciels (LPL) consiste à introduire les principes et les concepts des lignes de produits des autres domaines industriels dans le domaine du génie logiciel. Ainsi, dans une ligne de produits logiciels, les efforts sont orientés vers le développement de familles de produits partageant des caractéristiques communes plutôt que vers le développement de produits séparés. À cet égard, les lignes de produits logiciels reposent fortement sur la réutilisation stratégique et planifiée sans se contenter de la réutilisation contingente utilisée dans le passé et qui a été en grande partie discréditée (Northrop, 2002).

Du point de vue pratique, une ligne de produits logiciels consiste en deux processus interdépendants : le processus de l'ingénierie du domaine qui sert à produire l'ensemble des actifs fondamentaux (*Core Assets*) utilisés dans la construction des membres de la famille, et le processus d'ingénierie d'application qui utilise ces actifs pour produire les membres proprement dits de la famille.

1.4.1 Ingénierie du domaine

L'ingénierie du domaine est une activité dont les origines remontent aux années 70 (1976) (Czarnecki, 2002) avec les travaux de Dijkstra (Dijkstra, 1997) et Parnas (Parnas, 1976). Elle consiste, en résumé, en la collection, l'organisation et le stockage des expériences accumulées lors du développement de systèmes pour un domaine particulier sous forme d'actifs fondamentaux réutilisables (ex. patrons, modèles, composants, DSL, compilateurs, générateurs, etc.).

Dans le processus d'ingénierie du domaine, on trouve trois activités principales : Analyse de domaine, conception de domaine et implémentation du domaine. Quelques définitions reliées à l'ingénierie du domaine sont maintenant présentées.

Analyse de domaine (*Domain Analysis*) : activité qui consiste à identifier et à organiser l'information nécessaire au développement d'une famille de produits logiciels pour un domaine donné (Prieto-Diaz, 1990). En pratique, cette activité consiste en l'analyse des points communs et des variations entre les membres de la famille.

Conception de domaine *(Domain Design)* **:** le but de la conception de domaine est de développer une architecture commune à tous les produits membres de la ligne de produits en déterminant les composants fondamentaux de la famille, leurs interfaces et leurs interconnexions (Czarnecki et Eisenecker, 1999).

Implantation de domaine *(Domain Implementation)* : il s'agit d'implanter l'architecture du domaine en développant les composants et les actifs fondamentaux qui serviront à la construction des membres de la famille (Stropky et Laforme, 1995). Parmi les actifs fondamentaux les plus intéressants, on trouve les DSL (voir Figure 1.1). Ces derniers sont utilisés, généralement, pour la configuration des membres de la famille.

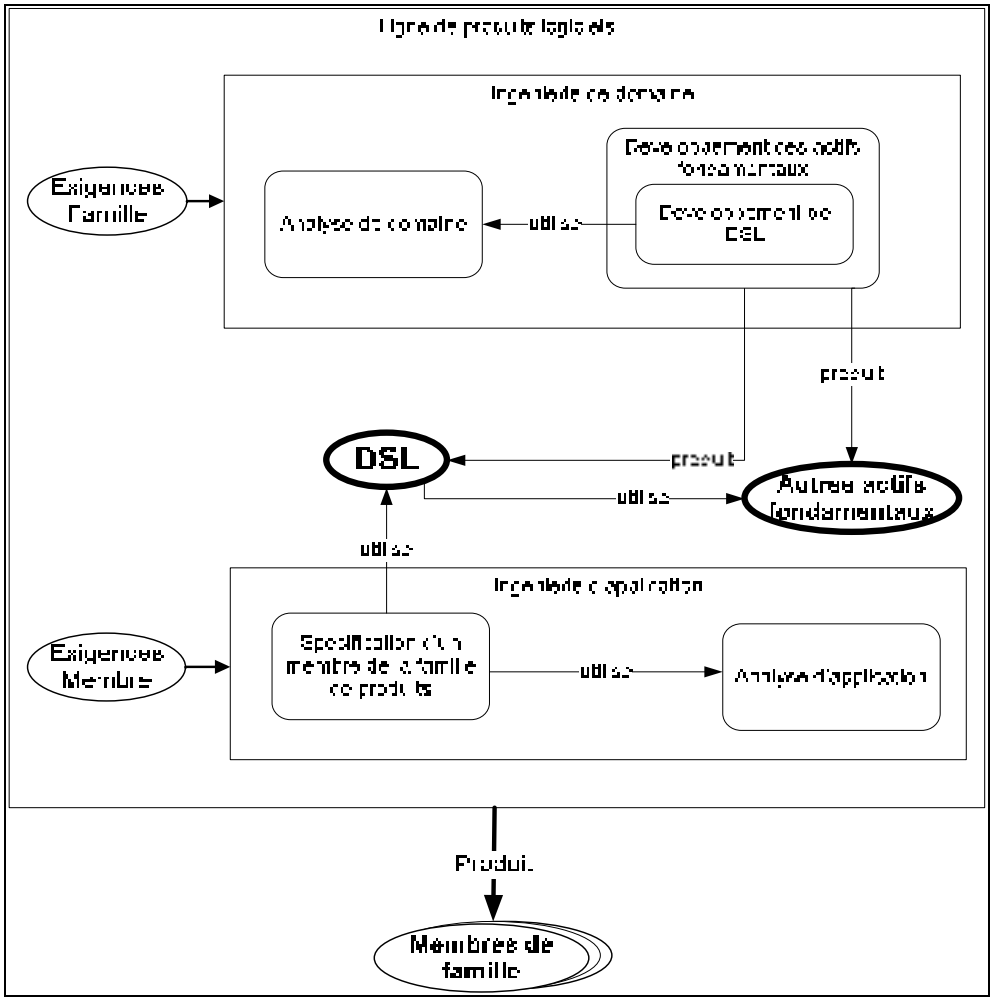

Figure 1.1 Structure générale d'une ligne de produits logiciels.

1.4.2 Ingénierie d'application

Le but de l'ingénierie d'application est de produire les membres de la famille en utilisant les actifs fondamentaux construits dans l'ingénierie de domaine (Ardis et Green, 1998). Le rôle des ingénieurs d'application consiste, entre autres, à spécifier les exigences spécifiques aux différents membres en se servant des langages dédiés comme interfaces de spécifications. Ensuite, les moteurs de génération implémentant les DSL se chargent de l'interprétation de ces spécifications et établissent les correspondances appropriées avec l'architecture de domaine afin de générer une architecture spécifique répondant aux exigences du membre en question. Évidemment, il y aura des situations où l'ingénieur d'application se retrouve avec

des exigences sans correspondance dans l'architecture de domaine. Dans ces cas, la réalisation de l'application nécessite une adaptation, voire une extension de l'architecture de domaine et de ses composants sous-jacents (Jaring et Bosch, 2004).

1.5 Usines à logiciel

Les usines à logiciels (*software factories*) sont une approche proposée par Microsoft pour le développement du logiciel. Une usine à logiciel est une ligne de produits logiciels qui configure les plateformes de développement logiciel extensibles comme « Visual Studio Team System », avec des actifs fondamentaux comme les DSL, les patrons et les frameworks, afin d'optimiser le développement et la maintenance de certains types d'applications (Greenfield et Short, 2004a). Le mécanisme d'extension est réalisé en se basant sur deux concepts : les schémas (*factory schema*) qui définissent et organisent les artefacts utilisés pour construire et maintenir le système, et les modèles (*templates*) qui forment une capacité de production pour la famille de produits en configurant les outils, les processus et les autres actifs fondamentaux (Greenfield et Short, 2004b).

Les usines à logiciels reposent fortement sur les langages dédiés pour soutenir l'automatisation (Greenfield et Short, 2004b). Ces langages sont utilisés pour exprimer différents aspects des membres de la famille à développer avec l'usine. Généralement plusieurs DSL sont nécessaires pour décrire une application. Chaque DSL (ou ensemble de DSL) correspond à un point de vue donné de l'application.

1.6 Développement génératif (*Generative Programming*)

Le développement génératif est un paradigme de développement logiciel qui consiste à générer des familles de systèmes à partir de spécifications écrites dans un ou plusieurs langages dédiés (Czarnecki, 2004). Ce paradigme repose essentiellement sur la conception et le développement de modules logiciels réutilisables (composants logiciels) qui peuvent être combinés pour générer des applications spécialisées répondant à des exigences spécifiques d'un domaine (Eisenecker, 1997).

Selon Czarnecki *et al.* (Czarnecki et al., 2000), la programmation générative utilise les langages dédiés afin d'améliorer la capacité des programmes (modèles) à refléter l'intention des développeurs, et permettre la vérification d'erreurs. Les DSL permettent aux utilisateurs de spécifier le problème en utilisant les concepts du domaine. Ensuite la correspondance entre l'espace du problème et l'espace de la solution est effectuée à l'aide des informations de configuration (*Configuration Knowledge*) (Czarnecki et al., 2000).

1.7 Développement dirigé par les langages

Le développement dirigé par les langages est l'espace technologique le plus intimement lié aux langages dédiés. Il repose essentiellement sur l'adoption d'une approche unifiée pour le développement et l'intégration de langages. L'adoption d'une telle approche permet à ce paradigme d'offrir les langages qui se prêtent le mieux à la résolution du problème. Ces langages se caractérisent par un pouvoir d'expression et un fort niveau d'abstraction proposant des abstractions représentant les concepts du domaine plutôt que les concepts du code.

Le développement dirigé par les langages tente d'apporter une solution à la problématique de la complexité et de la diversité des systèmes logiciels en utilisant une variété de langages dédiés pour adresser les différentes facettes d'un problème. Néanmoins, l'adoption de ce paradigme nécessite la mise en place d'infrastructures qui permettent de soutenir le développement des langages. Malheureusement, en ce moment, les technologies offertes ne se montrent pas encore prêtes pour accompagner efficacement les développeurs de langages dédiés dans la réalisation de leurs tâches.

1.8 UML versus DSL

Comme déjà évoqué, les approches dirigées par les modèles reposent sur l'utilisation de modèles expressifs et sémantiquement précis, contenant suffisamment d'information pour décrire les différents aspects d'un système. Néanmoins, l'exigence d'une sémantique claire et précise pour les modèles soulève des questions sur la capacité des langages de modélisation

dits généralistes, en général, et sur la capacité du langage UML, en particulier, à offrir ce niveau de précision.

Les experts se divisent en deux groupes sur ce sujet : les pros-DSL qui affirment qu'UML ne peut produire des modèles qui répondent aux exigences des approches dirigées par les modèles à cause de son manque de précision et de sa nature générique qui l'empêche d'exprimer les systèmes avec le niveau de précision attendu (Fowler, 2003; Greenfield et Short, 2004b; Karsai et al., 2000; Kelly et Tolvanen, 2008; Tolvanen et Kelly, 2001). En revanche, les pros-UML pensent qu'UML a fait ses preuves dans la plupart des cas et que peu sont les cas où le langage s'est montré moins approprié (Booch, 2004; Watson, 2008). Booch argue que l'adoption d'un DSL n'est judicieuse que dans ces cas où les concepts du domaine sont plus naturellement exprimés dans une syntaxe spécialisée, d'autant plus qu'UML offre des mécanismes d'extension comme les profils qui permettent de donner une forme spécifique à ce langage. Watson (Watson, 2008) précise qu'UML et les DSL ne sont pas nécessairement des opposés incompatibles. Il indique qu'entre les deux réside un continuum de langages de modélisation, tous susceptibles d'être utilisés dans le cadre du développement dirigé par les modèles. Le choix du type de langage utilisé dépend du type d'application et du degré de spécificité du domaine. Voici l'ordre de ces langages dans le continuum du plus générique au plus spécifique :

1. Langage de modélisation unifié UML : plus approprié pour des applications de grande portée ;
2. Langages fortement basés sur UML : langages définis dans le cadre syntaxique et sémantique du langage UML en offrant une personnalisation légère des éléments standards d'UML permettant ainsi de créer un dialecte adapté au domaine. Les DSL basés sur les profils UML sont un exemple typique de ce type de langages. Ces derniers définissent des éléments standards (stéréotypes, valeurs marquées ou contraintes) en personnalisant ceux spécifiés par le métamodèle UML (OMG, 2005a) ;
3. Langages plus ou moins basés UML : sont moins liés à UML mais ils réutilisent une grande partie de sa syntaxe et de sa sémantique. Le langage sysML, un langage de

modélisation spécifique au domaine de l'ingénierie système, est un exemple de ce type de langage ;

4. DSL standards : ne sont pas basés sur UML mais, en revanche, ils partagent avec celui-ci le même métamodèle MOF. SBVR est un exemple de ce type de langages. Ce langage définit un métamodèle, basé sur MOF, pour représenter les règles d'affaires tout en définissant une notation spécialisée pour représenter ces règles ;
5. DSL sur mesure : l'utilisation de ces langages peut être justifiée pour ces cas où les concepts du domaine sont plus naturellement exprimés dans une syntaxe spécialisée. À la différence des DSL standards, les DSL sur mesure ne sont pas contraints à utiliser le MOF. Ils peuvent se définir à l'aide de n'importe quel autre langage de métamodélisation.

1.9 Sommaire

Ces dernières années ont connu l'émergence de nombreuses approches de développement logiciel qui visent à élever le niveau d'abstraction et à masquer la complexité technologique. Le dénominateur commun de ces approches est l'utilisation des modèles comme artefact central dans le processus de développement. Ces modèles ne sont plus considérés du seul point de vue contemplatif où ils sont utilisés à des fins de documentation et de communication mais surtout d'un point de vue productif. Or, la création de modèles productifs requiert des formalismes précis facilitant leur traitement par la machine.

Dans ce chapitre nous avons fait un survol des approches émergentes de développement logiciel et présenté leurs concepts fondamentaux. Nous avons également montré la position des langages dédiés dans chacune de ces approches. Ce survol a révélé le potentiel des DSL et une ferme intention de les introduire dans le cycle de développement logiciel.

Quoique que les avantages présentés par les langages dédiés soient indéniables, la difficulté et le coût associés à leur développement constituent un frein marquant à leur adoption dans l'industrie. À ce titre, l'utilisation du langage de modélisation UML reste envisageable, voire

quelquefois, avantageuse. En fait, il ne faut pas considérer UML et les DSL comme deux opposés incompatibles mais plutôt comme deux solutions complémentaires.

CHAPITRE 2

LANGAGES DÉDIÉS : VUE D'ENSEMBLE

2.1 Introduction

Le concept des langages dédiés (DSL) n'est pas nouveau et leur utilisation a toujours été considérée avantageuse :

> « *We must develop languages that the scientist, the architect, the teacher, and the layman can use without being computer experts. The language for each user must be as natural as possible to her/him. The statistician must talk to his terminal in the language of statistics. The civil engineer must use the language of civil engineering. When a man learns his profession he must learn the problem-oriented languages to go with that profession* » (Martin, 1967).

> « *We must constantly turn to new languages in order to express our ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand* » (Abelson, Sussman et Sussman, 1996).

L'utilisation des DSL ne se limite pas au domaine informatique. Les DSL sont également utilisés dans des domaines comme la finance (Arnold, Van Deursen et Res, 1995), la chimie (Murray-Rust, 1997), la biologie (Hucka et al., 2003), la musique (Boulanger, 2000), etc. Parmi les DSL les plus communément utilisés aujourd'hui dans le cadre informatique, on peut citer le langage de définition et de manipulation de données SQL (*Structured Query Language*) (Chamberlin et Boyce, 1974; ISO, 2008), le langage des expressions régulières

pour la manipulation des chaines de caractères (Friedl, 2006), le langage de formules de Microsoft Office Excel (Microsoft, 2011) ou encore le langage de balisage XML (ISO, 1986).

Un langage dédié est un langage restreint à une classe de problèmes dans un domaine particulier. La restriction du DSL à un domaine d'application particulier permet de mieux réaliser le rapprochement entre les concepts du langage et ceux du domaine. Ce rapprochement présente de nombreux bénéfices en termes d'expressivité et de précision (sémantique) pour le langage. Outre ces bénéfices, les DSL ont démontré également un potentiel intéressant en termes de productivité, de réutilisabilité et de fiabilité (Kelly et Tolvanen, 2008; Kleppe, 2008).

Ce chapitre donne une vue d'ensemble sur le concept de DSL, leur processus de développement et sur les outils disponibles pour leur création.

2.2 Qu'est ce qu'un DSL?

Il est difficile de donner une définition précise de ce qu'est un DSL. La littérature fait état de définitions variées, chacune mettant en évidence un ou plusieurs aspects de ces langages. Ces différences de perceptions affectent même leurs appellations. Ainsi, les DSL sont également appelés « *Little Languages* », « *Micro Languages* », « *Domain Modeling Languages - DML* », « *Domain Specific Modeling Languages* DSML », « *Domain Specific Visual Languages - DSVL* » (Sprinkle et Karsai, 2004), « *Domain Specific Visual Modeling Languages* - DSVML », « *Domain Specific Embedded Languages* - DSEL », etc.[2].

[2] Dans ce document nous utiliserons indifféremment les termes DSL et langage dédié.

Une bonne approche pour introduire le concept de DSL est de le diviser en trois parties : « *Domain* », « *Specific* » et « *Language* » :

Domaine : la difficulté de la définition de DSL est due en grande partie à l'imprécision de la définition de domaine. Le terme « domaine » est tellement surchargé que cela devient impossible de couvrir toutes les définitions données. Chaque communauté a développé sa propre perception du concept de domaine et a formulé sa propre définition. Plus que cela, il est très fréquent de trouver plusieurs définitions au sein de la même communauté.

La communauté des langages dédiés n'a pas encore exposé sa propre perception du concept de domaine. Les définitions qui y sont adoptées consistent en une mixture de définitions venant des différents espaces technologiques liés à l'espace des langages dédiés (e.x. lignes de produits logiciels, réutilisation logicielle, ingénierie de domaine, analyse de domaine, etc.). Cela peut s'expliquer par la nature auxiliaire de l'espace des langages dédiés qui fait que cet espace est souvent cadré par l'espace technologique dans lequel il est utilisé.

Le reste de cette section présente le concept du domaine tel qu'il est perçu par les principaux espaces technologiques liés aux langages dédiés.

Dans le contexte de la réutilisation logicielle un domaine peut être vu comme :

> « *A set of current and future applications which share a set of common capabilities and data* » (Kang et al., 1990).
>
> Ou aussi :
>
> « *Abstractions that group particular sets of systems, or areas of functionality within systems, in ways that allow strategic reuse of assets within the domain scope* » (Simos et al., 1996).

Dans le contexte de l'ingénierie de domaine, un domaine peut être vu comme :

> « *A functional area shared across a group of products* » (Creps et al., 1996).

Dans le contexte des lignes de produits logiciels, un domaine peut être vu comme :

> « *A specialized body of knowledge, an area of expertise, or a collection of related functionality* » (Northrop, 2002).

Dans le contexte de la programmation générative, Czarnecki définit un domaine comme étant :

> « *An area of knowledge* :
> *- Scoped to maximize the satisfaction of the requirements of its stakeholders,*
> *- Including a set of concepts and terminology understood by practitioners in that area, and*
> *- Including knowledge of how to build software systems (or parts of software systems) in that area* » (Czarnecki et Eisenecker, 2000).

Dans le contexte de l'analyse de domaine on constate une tendance à adopter une vision plutôt flexible et adaptable pour le domaine. Cela peut être expliqué par le fait que l'analyse de domaine est utilisée dans divers contextes et par de multiples communautés. Ainsi, selon Kang *et al.* (Kang et al., 1990), le domaine est un concept général qui peut être étendu et appliqué à quasiment toute classe de systèmes. Cette classe constitue ce que l'on appelle le domaine cible (*Target Domain*). Encore à ce niveau on constate une diversité de définitions parmi les différentes méthodes d'analyse de domaine.

La méthode FODA définit un domaine comme étant :

> « *A set of current and future applications which share a set of common capabilities and data* ». (Tracz, 1994).

Dans le contexte de la méthode DSSA (*Domain-Specific Software Architecture*) un domaine est:

> « *A set of "common" problems or functions that applications in that domain can solve/do* » (Kaisler, 2005).

Dans le contexte de la méthode FAST (*Family-Oriented Abstraction, Specification, and Translation*), le domaine correspond à la famille de produits (Weiss, 1998) ; donc à un ensemble de produits caractérisés par un ensemble de points communs et de variations. Selon Weiss (Weiss et Ardis, 1997), il n'y pas de règles qui permettent d'identifier facilement une famille à cause de l'imprécision de ce concept.

> « *We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members* » (Parnas, 1976).

Selon la perspective de la méthode ODM (Simos et al., 1996) :

> « *A domain in the ODM context can be any field of inquiry or realm of discourse that has a common, defining categorical definition and a set of elements that are evaluated to be members of the domain by virtue of sharing those defining features* »

Dans ce contexte de la méthode ODM, une comparaison de la notion de domaine telle qu'elle est utilisée dans différentes disciplines et communautés a été faite (ex. linguistique, intelligence artificielle, technologie orientée objets, réutilisation logicielle, etc.) et deux grandes catégories d'utilisation du terme domaine ont été discernées : domaine en tant que « *Real World* » et domaine en tant qu'ensemble de systèmes (*set of systems*).

1) Domaine en tant que « *Real World* » : cette catégorie de domaine a été adoptée principalement dans les contextes suivants : génie cognitif (*Knowledge Engineering*), intelligence artificielle et modélisation orientée-objet. Dans cette catégorie la connaissance du domaine est élaborée en se basant sur le domaine du problème plutôt que sur le domaine de la solution :

> « *An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area* » (OMG, 1997).

2) Domaine en tant qu'ensemble de systèmes : un domaine de cette catégorie comprend les connaissances utiles à la construction d'une famille de systèmes logiciels. Cette perception est soutenue principalement par la communauté de la réutilisation logicielle où les analystes de domaine sont particulièrement concernés par la création de modèles facilitant la conception de logiciels réutilisables.

Par ailleurs, les domaines en tant qu'ensemble de systèmes sont à leur tour classés en deux sous-catégories : domaines en tant que collections de systèmes et domaines en tant que classes de systèmes. Un domaine en tant que collection de systèmes est décrit en fonction d'une collection de systèmes qui ne prend pas nécessairement en considération la cohérence des caractéristiques de ses systèmes constitutifs. Un domaine en tant que classe de systèmes est décrit selon un ensemble de règles et/ou de critères qui déterminent les systèmes à inclure ou à exclure de la classe.

Lorsque le domaine est défini en tant que « *Real World* », les experts du domaine sont généralement les praticiens alors que s'il est défini en tant qu'« *ensemble de systèmes* », ce sont les ingénieurs qui ont développé les systèmes du domaine qui sont considérés comme les experts de domaine.

Outre la catégorisation « macroscopique » qui classe les domaines selon des domaines en tant que « *Real World* » et des domaines en tant qu'ensemble de systèmes, ODM fait également

la distinction entre les types de domaines suivants : vertical versus horizontal (*Vertical vs. Horizontal*), natif versus novateur (*Native vs. Innovative*), et diffusé versus encapsulé (*Diffused vs. Encapsulated*) (Simos et al., 1996) : les domaines verticaux sont des domaines à grande échelle qui couvrent les systèmes dans leur globalité. Ils correspondent généralement à des domaines d'affaire comme l'éducation, l'assurance, la téléphonie, etc. Les domaines horizontaux, d'autre part, sont habituellement des domaines techniques utilitaires qui contiennent un sous-ensemble des fonctionnalités d'un système. La construction des interfaces usagers, l'accès aux bases de données et la gestion de publipostage sont des exemples de domaines horizontaux.

Un domaine horizontal peut être diffusé ou encapsulé. Un domaine diffusé définit un champ fonctionnel distribué à travers les sous-systèmes du système. Un domaine encapsulé définit un champ fonctionnel dont les fonctionnalités sont regroupées dans un seul composant du système.

Les domaines natifs sont des domaines « familiers » aux praticiens et facilement identifiables par les experts de domaine. Ces domaines correspondent généralement à des champs fonctionnels du niveau d'affaire (e.x. achats, stock, fournitures, etc.). Un domaine novateur, d'autre part, est un domaine défini dans une perspective novatrice visant à créer de nouvelles possibilités et solutions pour le système en question.

Spécificité : la spécificité du domaine est une question de degré. Cette caractéristique détermine le degré de précision avec lequel le DSL définit les besoins de son domaine et sa capacité à satisfaire ces besoins de la manière la plus naturelle possible pour les utilisateurs. Un DSL avec un degré élevé de spécificité offre des constructions et des fonctionnalités qui concordent avec les tâches quotidiennes des experts métier. Ces derniers doivent être à même de comprendre les concepts du langage et de manipuler par eux-mêmes ses modèles/programmes sans avoir besoin de connaissances particulières en programmation.

En revanche, et étant donné que la majorité des domaines changent et évoluent continuellement, le maintien de la spécificité des DSL associés devient un véritable défi. Cela exige du DSL d'avoir une forte capacité d'adaptation et d'évolution afin de pouvoir s'accommoder facilement aux changements et aux transformations du domaine. La difficulté vient du fait que le DSL doit répondre aux changements de son domaine tout en conservant son expressivité et son fort niveau d'abstraction. Il doit également assurer la rétrocompatibilité (*Backward Compatibility*). La nouvelle version du DSL doit continuer son support pour les modèles développés avec les versions précédentes.

Langage : d'une manière très générale, un langage est constitué d'une syntaxe et d'une sémantique. La syntaxe est consacrée à la définition de la forme des expressions valides du langage alors que la sémantique se préoccupe du sens porté par ses expressions. Dans le contexte informatique on distingue deux types de langages : les langages de programmation et les langages de modélisation.

Dans la littérature, il n'y a pas de définition précise et communément admise permettant de faire une distinction claire entre un langage de modélisation et un langage de programmation (Greenfield et Short, 2004b). Cette distinction n'est faite qu'informellement en considérant les utilisations historiques de chacun des deux types de langages. Parmi les aspects qui sont considérés, souvent, comme distinctifs, on trouve : la notation (textuelle vs graphique), le style de spécification (impératif vs déclaratif) et la capacité d'exécution. Il se trouve que la majorité des langages de programmation sont textuels, impératifs et exécutables alors que la plupart des langages de modélisation sont graphiques, déclaratifs, et non exécutables.

Évidemment, ces suppositions ne sont pas bien fondées et peuvent être réfutées facilement avec des exemples. De plus, certains spécialistes pensent que si les tendances actuelles se maintiennent, la distinction entre les langages de programmation et les langages de modélisation deviendra totalement insignifiante (Greenfield et Short, 2004b). Ainsi, un programme peut être considéré comme un type particulier de modèles.

Une définition très générale décrit ces langages selon l'uplet $\{S_A, S_C, S_M, M_{AC}, M_{AS}\}$ (Chen, Sztipanovits et Neema, 2005) représentant respectivement la syntaxe abstraite (S_A), la syntaxe concrète (S_C), le domaine sémantique (S_M), le mappage de la syntaxe abstraite vers la syntaxe concrète ($M_{AC} : S_A \rightarrow S_C$) et le mappage de la syntaxe abstraite vers le domaine sémantique ($M_{AS} : S_A \rightarrow S_M$).

En conclusion et à la lumière des considérations précédentes, nous considérons un DSL comme étant un petit langage de programmation ou de modélisation exécutable caractérisé par :

- un domaine d'application : ce domaine correspond généralement à un domaine horizontal. Un domaine vertical serait bien trop large pour être utilisé dans le contexte d'un DSL. Souvent, dans la pratique, plusieurs DSL sont nécessaires pour implémenter un domaine d'affaire;
- une syntaxe et une sémantique qui est spécifique au domaine;
- une notation expressive et familière aux experts de domaine. La notation peut être textuelle, graphique ou souvent une combinaison des deux.

Tel que déjà mentionné, il est difficile de trouver une définition qui saurait cadrer tous les aspects et caractéristiques d'un DSL. Ainsi, nous proposons une définition qui résume notre perception du terme DSL. Cette définition est basée en grande partie sur la définition proposée dans (Deursen, Klint et Visser 2000) :

> « *Un DSL est un petit langage qui offre, grâce à des notations et des abstractions appropriées, un pouvoir d'expression permettant d'améliorer les performances des parties prenantes dans un domaine particulier* »

Cette définition met l'accent sur un ensemble de caractéristiques que nous considérons fondamentales pour les DSL. Tout d'abord on précise qu'un DSL est un petit langage très restreint dans sa syntaxe et dans sa sémantique. À la différence des langages généralistes, les

DSL ne doivent contenir que les constructions nécessaires et indispensables à la résolution des problèmes du domaine (Wile, 2004).

Un DSL est caractérisé également par son expressivité et son fort niveau d'abstraction. L'élévation du niveau d'abstraction apporte des gains considérables en productivité et en qualité (Mernik, Heering et Sloane, 2005). En effet, en offrant des abstractions et des notations familières aux experts du domaine et en masquant les détails d'implémentation, on rend le processus de développement plus sûr. Ainsi, les experts de domaine peuvent vérifier et valider plus facilement les solutions proposées; ils peuvent même participer à la conception de ces solutions.

La définition met également l'accent sur l'importance de mettre les parties prenantes au centre des préoccupations lors du développement d'un DSL. Comme nous l'avons déjà vu, trop peu de définitions déclarent explicitement les parties prenantes et leurs besoins comme une partie intégrante du concept de domaine (Czarnecki et Eisenecker, 2000). Pour qu'un DSL soit utile pour ses utilisateurs, il doit prendre en considération les rôles organisationnels de ces derniers, leurs expériences et leurs compétences (Wile, 2004).

2.3 Types de DSLs

Fowler classe les DSL en deux catégories (Fowler, 2011) : les DSL enchâssés (*Internal DSL*) et les DSL autonomes (*External DSL*). Un DSL enchâssé, ou DSL embarqué (*embedded DSL*) (Hudak, 1996), est un langage dédié qui se définit à l'intérieur d'un autre langage, généralement un langage généraliste, dit langage hôte. La création d'un DSL enchâssé consiste alors à étendre le langage hôte avec un dialecte dédié au domaine tout en respectant le formalisme et la syntaxe du langage hôte. L'avantage majeur de cette approche est qu'elle permet de profiter de l'infrastructure, des caractéristiques et des outils de développement du langage hôte. Ainsi, le développeur n'a pas à définir une grammaire ni à créer les outils de support pour son DSL. Parmi les langages les plus connus pour le développement de ce genre

de DSL on peut noter : Lisp (McCarthy, 1965; Steele, 1990), Ruby (Hansson, 2011), Haskell (Jones, 2002) et Groovy (Strachan et McWhirter, 2011).

Un DSL autonome, d'autre part, est un langage indépendant du langage principal de l'application. Ainsi, il offre plus de flexibilité en termes de grammaire et de notation en permettant de définir la notation la mieux appropriée pour exprimer les concepts du domaine. Les petits langages d'UNIX (e.x. awk, make, etc.), le langage Ant pour java et les expressions régulières sont des exemples de ce type de DSL. L'inconvénient majeur de ce type de DSL est le coût associé au développement de leurs outils de support, soit les générateurs de code, les compilateurs et interprètes, les éditeurs de modèles, etc. Le développement de ces outils demande des efforts considérables et des compétences particulières qu'on ne trouve, généralement, pas chez un développeur.

Les Tableau 2.1 et Tableau 2.2 présentent les avantages et les inconvénients de chacun des deux types de DSL.

Tableau 2.1 Avantages et inconvénients des DSL enchâssés

Avantages	**Inconvénients**
• Profiter des outils du langage hôte; • Bénéficier de la puissance et des caractéristiques du langage hôte ; • Permettre de mieux se concentrer sur les fonctionnalités d'un haut niveau d'abstraction en laissant le langage hôte gérer les détails de bas niveaux ; • Faciliter l'implémentation : pas besoin d'avoir des connaissances en développement de langages.	• Limités dans leur syntaxe et dans leur structure par celles du langage hôte; • L'efficacité du DSL dépend, largement, des caractéristiques du langage hôte; • Difficulté d'exprimer quelque chose qui n'a pas de correspondance dans le langage hôte.

Tableau 2.2 Avantages et inconvénients des DSL autonomes

Avantages	Inconvénients
• Pas de contraintes préalables : liberté de choisir la forme la plus adéquate pour exprimer les concepts du domaine; • Contrôle total sur le DSL (syntaxe, règles de grammaire, validation des modèles).	• Plus difficile à concevoir et à implémenter; • Manque de support : il faut construire soi-même les outils de support (ex. compilateur/interprète, éditeur, débogueur, etc.); • Les développeurs doivent apprendre un nouveau langage; • Nécessite des connaissances en développement des langages (définition d'une grammaire, analyse grammaticale, etc.).

En plus de la classification établie par Fowler, les DSL peuvent aussi être classifiés selon la notation et le style de spécification adopté :

Notation : les modèles du DSL peuvent être exprimés sous une forme textuelle, graphique[3], ou une combinaison des deux. Ici, il est important, de corriger la fausse idée courante qui associe la notation textuelle aux langages de programmation et la notation graphique aux langages de modélisation (Greenfield et Short, 2004a). Il est totalement permissible, voire dans certain cas plus approprié, de concevoir un DSL de programmation sous format graphique et vice-versa.

Style de spécification : les instructions d'un DSL peuvent être d'une nature impérative ou déclarative (Greenfield et Short, 2004b). Une spécification impérative décrit les instructions à exécuter sans décrire les résultats d'exécution escomptés tandis qu'une spécification déclarative décrit les résultats d'exécution attendus sans décrire la façon dont ils sont obtenus (Liberty et Horovitz, 2008).

[3] Les langages spécifiques au domaine qui utilisent cette notation sont parfois appelés langages visuels spécifiques au domaine (*Domain Specific Visual Languages*) ou DSML (*Domain Specific Modeling Languages*)

2.4 Outils de métamodélisation

Les outils de métamodélisation (*MetaCase Tools*) sont des environnements de développement destinés, entre autres, à la création et à la modélisation de langages de modélisation, et donc de langages dédiés. Une fois que le métamodèle du langage est explicitement défini, ces outils permettent de générer les environnements spécifiques nécessaires à la création et à l'édition des modèles du langage. Ils permettent également de générer les outils de validation et de transformation de modèles, voire même de génération de code.

Cette section présente les outils de métamodélisation actuellement disponibles pour soutenir la définition des DSL.

2.4.1 Eclipse Modeling Project (EMP)

Le projet *Eclipse Modeling Project* a été créé afin de fédérer l'ensemble des sous-projets Eclipse portant sur la modélisation. L'objectif de ce projet est de créer un ensemble unifié de frameworks, d'outils et de normes pour prendre en charge le développement dirigé par les modèles. La définition des DSL est effectuée en utilisant, principalement, les trois frameworks suivants : EMF (*Eclipse Modeling Framework*), GEF (*Graphical Editing Framework*) et GMF (*Graphical Modeling Framework*).

EMF est un Framework de modélisation qui permet de décrire des modèles et de générer des outils permettant la manipulation de ces modèles. Fondamentalement, EMF consiste en trois composants : un métamodèle pour la définition des modèles (*Ecore*), un simulateur (*runtime)* pour simuler le comportement des modèles définis et un générateur pour générer une implémentation java de ces modèles.

GEF permet de construire des éditeurs afin de pouvoir afficher et éditer, graphiquement, des modèles. Les possibilités d'édition du GEF permettent de construire des éditeurs graphiques pour quasiment tous les modèles. La visualisation graphique est faite avec le Framework

Draw2D, un Framework de dessin 2D basé sur *SWT*, un *toolkit* graphique pour développer des interfaces graphiques en Java.

GMF a été créé pour pallier les difficultés rencontrées lors de l'utilisation de GEF et aussi pour faire le pont entre EMF et GEF. GMF est constitué de deux composantes principales : *GMF Runtime*, une infrastructure d'exécution pour développer des éditeurs graphiques basés sur EMF et GEF; et *GMF Tooling*, un composant générateur pour la génération d'éditeurs graphiques basés sur le *GMF Runtime*.

Le processus de développement d'un DSL en utilisant les outils EMP consiste en :

Définition de la syntaxe abstraite : consiste en la création du modèle de domaine définissant les concepts du DSL (*Domain Classes*), leurs propriétés et leurs relations. Le modèle de domaine est défini par un modèle *Ecore*. Ce dernier peut être créé soit par l'éditeur standard EMF ou par l'éditeur graphique *Ecore Tools*, un composant du projet EMFT (*Eclipse Modeling Framework Technology*) permettant de manipuler graphiquement des modèles *Ecore*.

Spécification de la syntaxe concrète : la définition de la syntaxe concrète consiste à créer les figures. Ces figures définissent comment sont représentés les éléments du modèle de domaine. Pour cela, il faut créer un modèle de définition graphique (*.gmfgraph)* qui va définir les figures associées aux différents éléments définis par le modèle de domaine (*.ecore*).

Description des outils : consiste à créer le modèle d'outils (*tooling definition model*) qui permet de décrire les éléments à utiliser dans l'éditeur graphique (palette; menus, barre d'outils, etc.).

Définition des correspondances : définir la correspondance entre les éléments du modèle de domaine, les éléments du modèle graphique et les éléments du modèle des outils.

2.4.2 MetaEdit+ de Metacase

MetaEdit+ est un environnement dédié à la création et à l'utilisation de langages dédiés. Il est composé de deux outils principaux : *MetaEdit+ Workbench* qui intègre des fonctionnalités permettant la création des DSL et de leurs outils de support (e.x. éditeur de modèles, générateurs, etc.) ; et *MetaEdit+ Modeler* qui offre l'environnement d'utilisation du DSL défini par le *MetaEdit+ Workbench*. Toutes les activités de modélisation (création et édition de modèles, gestion des informations de modélisation, etc.) sont effectuées dans cet environnement.

MetaEdit+ Workbench

Destiné essentiellement aux experts de domaines et aux architectes, cet outil offre un environnement intégré pour la définition et l'implémentation des DSL. L'environnement intègre un ensemble d'outils qui répondent aux besoins les plus fréquents en matière de métamodélisation. On y trouve, entre autres, un éditeur d'objets pour la définition des concepts, un éditeur de propriétés, un éditeur de relations et un éditeur de contraintes. L'environnement offre également un éditeur de symboles qui sert à définir la syntaxe concrète du DSL et un éditeur de générateur pour le développement des générateurs. La définition du langage est stockée sous forme d'un métamodèle dans le référentiel MetaEdit+.

MetaEdit+ Modeler

MetaEdit+ Modeler est l'outil responsable d'offrir un environnement d'utilisation pour les DSL définis avec l'outil *MetaEdit+ Workbench*. Les fonctionnalités offertes par cet environnement comportent, entre autres, les éditeurs graphiques, les explorateurs de modèles et les générateurs. L'environnement permet également de vérifier les modèles du DSL et d'établir des liens entre eux.

Le processus de développement d'un DSL avec *MetaEdit+* consiste en :

Création de la syntaxe abstraite : définir les concepts du DSL, leurs relations et leurs contraintes en utilisant l'outil *MetaEdit+ Workbench*.

Création de la syntaxe concrète : définir la notation représentant les concepts et les relations définis par la syntaxe abstraite. La définition de cette notation est réalisée en utilisant l'éditeur de symbole offert par l'outil *MetaEdit+ Workbench*.

Développement du générateur : construire le (ou les) générateur(s) implémentant le DSL. *MetaEdit+* offre un éditeur de générateurs permettant le développement, le débogage et le test de générateurs. Ces générateurs sont écrits dans un langage dédié qui permet de manipuler les informations des modèles et de générer le code à partir d'une implémentation de référence.

2.4.3 XMF

XMF est un environnement de métamodélisation dédié à la conception des langages. Cet environnement offre un ensemble de langages qui couvrent les principaux aspects de la métamodélisation. La majorité de ces langages ont été créés en étendant des standards reconnus comme MOF, OCL, QVT et EBNF.

2.4.3.1 Éléments clés d'XMF

L'environnement XMF est composé des éléments suivants :

XCore : fournit les concepts fondamentaux utilisés dans XMF. Ces concepts sont définis sous forme d'un ensemble de classes abstraites encapsulant des propriétés génériques pour les types de concepts les plus communs dans les DSL.

XOCL : une extension du langage de contraintes objet OCL qui rajoute un ensemble de primitives d'actions permettant de décrire l'aspect dynamique des DSL. Elle est la base de toutes les expressions impératives susceptibles de changer les états des modèles.

XBNF : basé sur EBNF, ce langage est utilisé pour effectuer l'analyse syntaxique des syntaxes concrètes textuelles. Il peut être utilisé, également, pour produire l'interprétation de celle-ci et en construire la syntaxe abstraite.

XMap : un langage déclaratif qui permet d'exprimer des mappages unidirectionnels. Un mappage unidirectionnel consiste à prendre un ou plusieurs modèles en entrée et à générer un modèle en sortie en utilisant le patron d'appariement (*Pattern Matching*), un mécanisme qui fait la correspondance entre les éléments des modèles sources avec ceux du modèle cible.

XSync : un langage déclaratif qui sert à la définition de la synchronisation des données et des langages en utilisant le mécanisme de mappage bidirectionnel. Le mappage bidirectionnel est utile dans les cas où on désire maintenir une synchronisation entre les éléments et s'assurer que les changements sur un des éléments sont pris en compte dans les autres. Il est particulièrement intéressant lorsqu'on désire maintenir la synchronisation entre les modèles de la syntaxe abstraite et ceux de la syntaxe concrète et entre les modèles utilisateurs et le code généré.

Le processus de création de DSL en utilisant XMF consiste en les activités suivantes :

Création de syntaxe abstraite : l'aspect statique (structurel) de la syntaxe abstraite est défini en utilisant les éléments du métamodèle XCore. L'aspect dynamique décrivant le comportement du DSL est décrit par le langage XOCL. Les règles de grammaire validant l'exactitude des modèles sont exprimées avec le langage OCL.

Création de syntaxe concrète : consiste en deux étapes principales : la première étape est la spécification de la structure syntactique du DSL. Le framework XMF supporte uniquement la

définition de la syntaxe textuelle à l'aide du langage XBNF. Il n'offre malheureusement, pas de langage pour la description des syntaxes concrètes graphiques. La deuxième étape est l'analyse syntaxique définissant comment la syntaxe abstraite est construite à partir de la syntaxe concrète.

Définition de la sémantique : XMF définit un ensemble de primitives d'action qui permettent de capturer la sémantique comportementale du DSL. Un modèle de sémantique XMF consiste en une extension du modèle de la syntaxe abstraite qui ajoute une couche décrivant le sens des concepts du DSL. L'avantage est que la sémantique est définie en utilisant la même approche de métamodélisation utilisée pour la définition de la syntaxe abstraite.

2.4.4 Visual Studio Visualization and Modeling SDK (VSVM)

« *Visual Studio Visualization and Modeling SDK* (VMSDK) » est le successeur des *DSL Tools*, lancés par Microsoft en 2004. C'est un kit de développement logiciel qui est livré maintenant avec *Visual Studio*. Ce kit permet de créer, à partir d'une modélisation des concepts métiers (métamodèle), un environnement de conception personnalisé dans *Visual Studio*.

Le processus de création d'un DSL avec les VMSDK consiste en les étapes suivantes :

Définition de la syntaxe abstraite : consiste à définir le modèle du domaine (*Domain Model*) qui définit les abstractions du DSL (*Domain Classes*) et leurs relations (*Domain Relationships*) et à exprimer les règles de grammaire.
Définition de la notation : définir les formes graphiques (forme, décorateur) constituant la syntaxe concrète du DSL.

Définition des correspondances (*Mapping*) : mettre en relation les concepts du DSL et les éléments de la syntaxe concrète.

Génération de l'environnement de conception (*Graphical Designer*) : générer, à partir de la définition du DSL, un éditeur de modèles personnalisé prenant en considération les concepts et les contraintes définis.

Génération de code : VMSDK permet la génération de code à partir de modèles de texte. La génération de code est effectuée en utilisant la transformation des modèles texte (*Text Template Transformations*). Un modèle texte est un fichier texte contenant des directives et des commandes que le moteur d'analyse exécute afin de générer le code final.

Déploiement du DSL : consiste en la création d'un paquetage Visual Studio .vsix (*Visual Studio Integration Extension*) qui peut être installé sur d'autres machines. Pour ce faire, il suffit de décompresser le dit paquetage dans le dossier des extensions *Visual Studio* pour que le DSL soit considéré comme une extension *Visual Studio* et profite des fonctionnalités du gestionnaire des extensions.

2.5 Standards

Les outils de développement DSL présentés dans la section précédente manipulent soit des technologies propriétaires, soit des versions adaptées de technologies connues comme standards (MOF, QVT, OCL, XMI, etc.). Cette diversité de technologies impacte sur l'interopérabilité des outils et, par conséquent, sur la portabilité des langages. Les standards constituent, à notre avis, les gardiens qui assurent cette interopérabilité et portabilité, et ce en guidant et en restreignant le développement DSL.

Les standards de l'OMG constituent les standards de facto en matière de modélisation et de métamodélisation. Ainsi, plusieurs outils se sont basés sur ces standards pour développer leurs propres fonctions (EMOF, XCore, XOCL, etc.). De plus, le répertoire des standards OMG semble couvrir une bonne partie des besoins exprimés en matière de définition de DSL (métamodélisation - MOF, transformation de modèles - QVT, sérialisation - XMI, etc.).

Voici une description sommaire de ces standards :

MOF (*Meta Object Facility*) : langage de métamodélisation adopté par l'OMG, pour la première fois, en 1997 afin de répondre aux besoins en matière de création et de manipulation de métamodèles. En 2005, la version 1.4.1 de la spécification MOF a été acceptée comme norme ISO - ISO/IEC 19502:2005. La version 2.0 de MOF définit deux versions de complexité différente : EMOF (Essential MOF), un sous-ensemble ne contenant que les concepts essentiels à la définition de métamodèles simples, et CMOF (Complete MOF), une version paquet offrant un ensemble, plus complet, de concepts qui permettent de définir des métamodèles plus complexes (OMG, 2008).

MOF est un langage qui s'auto-définit. Ainsi, on le trouve au sommet de l'architecture de métamodélisation à quatre couches de l'OMG. En revanche, il faut souligner que MOF s'intéresse seulement à la définition de la structure des langages (syntaxe abstraite) et ne prend pas en charge la définition de la sémantique opérationnelle.

QVT (*Query/View/Transformation*) : spécification dédiée à la transformation des modèles. La version 2.0 de cette spécification est composée de deux parties : une partie déclarative adoptant une architecture à deux couches et une partie impérative. La partie déclarative est définie par les langages *Relations* et *Core*. *Relations* permet d'exprimer des spécifications déclaratives pour les relations entre les modèles MOF. Ce langage est caractérisé par sa capacité à traiter des patrons d'appariement complexes (*Pattern Matching*). La syntaxe concrète du langage *Relations* peut être textuelle ou graphique. *Core* est un langage plus simple que *Relations*. Il consiste en une amélioration très minimaliste d'EMOF et des constructions OCL. En raison de cette approche minimaliste, les fonctionnalités manquantes doivent être programmées à la main, ce qui peut rendre les spécifications des transformations verbeuses et complexes (OMG, 2008).

La partie impérative est définie par le langage *Operational*. Ce langage permet d'exprimer des spécifications impératives pour les transformations en adoptant une syntaxe concrète

similaire à celle des langages impératifs. Une transformation entièrement écrite en utilisant *Operational Mapping* est appelée une transformation opérationnelle.

En plus de ce qui a été mentionné, QVT introduit un mécanisme appelé *BlackBox* pour invoquer des fonctions de transformation exprimées dans des langages externes comme XSLT et XQuery.

OCL (*Object Constraint Language*) : langage d'expression de contraintes sur les modèles MOF. Les expressions d'OCL permettent, entre autres, de spécifier des pré- et des post-conditions sur les opérations et les méthodes, de formuler des requêtes sur les objets et de spécifier des invariants sur les classes et les types. L'évaluation de ces expressions ne provoque aucun effet secondaire; quand une expression OCL est évaluée, elle retourne simplement une valeur (OMG, 2006a).

XMI (XML Metadata Interchange) : C'est le nom donné par l'OMG au standard de description des modèles sous forme de documents XML. L'objectif principal de XMI est de faciliter l'échange de métadonnées entre les outils de modélisation et les référentiels de métadonnées basés sur le métalangage MOF (OMG, 2005b). En 2005, la version 2.0.1 de la spécification XMI a été acceptée comme norme ISO (ISO/IEC 19503:2005) (ISO, 2005).

En raison de l'incapacité de XMI à représenter la partie graphique des modèles, l'OMG a défini le standards DI (*Diagram Interchange*), une spécification qui étend le métamodèle UML avec un métamodèle supplémentaire permettant la représentation des informations graphiques. La spécification fournit également un mécanisme de transformation de XMI vers SVG (*Scalable Vector Graphics*) afin d'assurer l'échange avec les outils graphiques (OMG, 2006b).

2.6 Processus de développement d'un DSL

Bien que les DSL aient été utilisés depuis bien longtemps, ils sont toujours dépourvus de processus et de méthodes clairs et précis permettant de piloter leur cycle de développement (Mernik, Heering et Sloane, 2005; Thibault, Marlet et Consel, 1999). Néanmoins, la synthèse de la littérature (Cleaveland, 1988; Consel et Marlet, 1998; Deursen, Klint et Visser, 2000 ; Mernik, Heering et Sloane, 2005; Tolvanen, 2006; Van Deursen et Klint, 1998) montre qu'un cycle de développement de DSL typique implique les quatre activités suivantes : décision, analyse, conception et implémentation.

2.6.1 Décision

L'adhésion dans le développement d'un DSL est, généralement, le résultat d'une décision prise après un examen des différentes options offertes (ex. utilisation d'un langage généraliste ou d'un DSL existant, achat d'un DSL prêt à utiliser, etc.) et une analyse des compromis envisageables. D'habitude, l'idée de développer un DSL ne surgit qu'après avoir développé une connaissance approfondie du domaine et ce en développant un nombre suffisant d'applications pour le domaine en utilisant des langages généralistes. Pour un nouveau domaine où peu de connaissance est disponible, la décision en faveur d'un nouveau DSL fait peu de sens. Cette décision est plus judicieuse quand la connaissance du domaine est suffisamment solide et quand les autres options alternatives proposées ne parviennent pas à combler tous les besoins. Mernik *et al.* (Mernik, Heering et Sloane, 2005) ont identifié un ensemble de patrons qui résument des situations où l'utilisation des DSL a été montrée rentable dans le passé.

2.6.2 Analyse de DSL

L'analyse de DSL consiste à analyser le domaine d'opération du DSL (Deursen, Klint et Visser 2000 ; Mernik, Heering et Sloane, 2005) soit identifier et organiser la connaissance utilisée pour le développement de familles de programmes pour ce domaine (Prieto-Diaz, 1990). Plusieurs méthodes d'analyse de domaine sont disponibles pour guider le processus

d'analyse de domaine quoique Prieto-Diaz affirme que ce processus se fait souvent d'une manière ad hoc en se basant essentiellement sur l'expertise de l'analyste du domaine (Prieto-Diaz, 1990). Généralement, l'analyse de domaine consiste en deux activités principales (America et al., 2001) : délimitation du domaine et modélisation du domaine.

Délimitation du domaine (*Domain scoping*)

Cette activité consiste à délimiter la portée du domaine du DSL et à circonscrire son champ d'opération. Ainsi, dans cette étape, l'analyste du DSL se concentre sur la définition de l'espace des problèmes ciblés par le DSL et sur la spécification des caractéristiques à inclure ou à exclure du DSL (Cleaveland, 1988).

La délimitation du domaine est une tâche à la fois difficile et critique (Schmid, 2002; van der Linden, 2002). Si la portée du domaine est trop large, le développement du DSL devient plus coûteux et le langage tendra à utiliser des concepts de plus en plus génériques, ce qui affectera négativement son expressivité. D'autre part, si le domaine est trop étroit, le DSL devient trop spécifique et son champ d'utilisation devient plus réduit. Afin de bien délimiter la portée du DSL, il faut définir précisément son problème du domaine. Après tout, un DSL est un langage spécialisé conçu pour répondre aux besoins d'un problème particulier dans un domaine donné. La définition claire et précise du problème à résoudre aide à mieux tracer les frontières de la portée du DSL.

Une autre technique consiste à délimiter la portée du DSL en se basant sur le concept de point de vue. Selon Tolvanen, la création d'un DSL commence, souvent, à partir d'un certain point de vue (*viewpoint*) (Tolvanen, 2006). Ce dernier définit la perspective à partir de laquelle le problème et/ou la solution est spécifié ou modélisé. Le développeur définit, d'abord, les points de vue dans le domaine en question, ensuite détermine le (ou les) DSL qui vient les supporter. Les points de vue constituent un moyen efficace pour séparer et organiser les préoccupations des différentes parties prenantes. Les DSL dirigés par les points de vue sont plus susceptibles de répondre aux besoins de leurs utilisateurs.

Dans (Simos, 1995; Simos et al., 1996), on discute plus en détail le concept de domaine et on présente les techniques qui peuvent être utilisées pour sa délimitation.

Modélisation de domaine

La modélisation de domaine consiste à collecter et à organiser la connaissance du domaine dans des modèles. Il n'y a pas de modèles spécifiques à produire pour cette activité (America et al., 2001). Chaque méthode d'analyse de domaine spécifie son propre ensemble de modèles à produire. Néanmoins, deux de ces modèles se révèlent particulièrement importants dans le cas des DSL : le modèle conceptuel, appelé aussi le modèle d'information, et le modèle des points communs et des variations.

- **Modèle conceptuel :** présente la connaissance du domaine en termes de concepts et d'abstractions. L'objectif de la modélisation conceptuelle est d'identifier et d'organiser les concepts à représenter par le DSL. Un modèle conceptuel peut prendre la forme d'un modèle entité-relation, d'un réseau sémantique, ou de toute autre forme susceptible d'exprimer la connaissance du domaine d'une manière claire et précise et, surtout, compréhensible par l'ensemble des parties prenantes.

- **Modèle des points communs et des variations :** l'objectif de l'analyse des points communs et des variations est d'identifier les points communs et les variations à l'intérieur du domaine en séparant les caractéristiques communes à l'ensemble des membres de la famille de programmes de ceux qui ne sont valables que pour certaines variantes. Selon Weis (Weiss, 1998), le résultat de cette analyse peut être exploitée comme une entrée à la conception d'un DSL. Les points communs définissent les parties qui peuvent être implémentés sous la forme d'actifs fondamentaux réutilisables par l'ensemble des membres de la famille de programmes ciblés par le DSL alors que l'étude des variations aide à formaliser la grammaire et la notation du DSL.

Les points de variations peuvent être exploités pour définir les abstractions et pour diriger la décomposition des modules et des composants logiciels implémentant les besoins fonctionnels spécifiques aux différentes variantes. De même, les relations et les dépendances entre ces points sont utilisées comme un point d'entrée à la définition des règles de grammaire du DSL.

L'analyse des variations consiste en deux activités principales : identification des points de variations (fonctionnelles et non-fonctionnelles) et définition des paramètres de variations :

- **Points de variations :** identifient les endroits où les variations peuvent avoir lieu (Ivar, Martin et Patrik, 1997). Chacun de ces endroits correspond à une décision de conception qu'il va falloir prendre lors de la conception des différents membres. Un point de variation est caractérisé par un type spécifiant le type de la variation et un ensemble de relations définissant les dépendances de celle-ci vis-à-vis des autres variations (une discussion plus détaillée sur les type de dépendance peut être trouvée dans (Bühne, Halmans et Pohl, 2003)). Selon Bachmann et Bass, il y a trois types de variations : optionnelle, alternative et ensemble d'alternatives (Bachmann et Bass, 2001) (voir aussi (Muthig et Atkinson, 2002)). Une variation est dite optionnelle si la fonctionnalité sous-jacente est facultative pour le membre de la famille. Elle est alternative si le membre doit implémenter une instance des alternatives offertes par le point de variation. Si le membre doit implémenter au moins une des alternatives, la variation est donc de la forme « ensemble d'alternatives ».
- **Paramètres de variations :** ce sont des variables caractérisées par leurs types et leurs domaines de valeurs. L'objectif ici est de quantifier et paramétrer les variations (Coplien, Hoffman et Weiss, 1998) en déterminant, pour chaque variation, l'ensemble des valeurs admissibles. Ainsi, la configuration d'un membre de la famille consiste en fait à définir un sous-ensemble du domaine des valeurs définies pour la famille.

La modélisation des caractéristiques (*Feature Modeling*) est l'une des techniques les plus utilisées pour modéliser les points communs et les variations (Antkiewicz et Czarnecki,

2004; Czarnecki, Helsen et Eisenecker, 2004; Lee, Kang et Lee, 2002). Elle permet d'identifier, de décrire et de structurer les caractéristiques des membres de la famille en définissant leurs propriétés et leurs interdépendances. Un modèle de caractéristiques consiste en un diagramme de caractéristiques (*Feature Diagram*) complété par des informations supplémentaires (e.g. description générale, description sémantique, contraintes, etc.). Un diagramme de caractéristiques présente les caractéristiques avec des nœuds organisés sous la forme d'un arbre ou d'un graphe. Le nœud racine correspond au concept modélisé alors que les nœuds dérivés présentent les caractéristiques et les sous caractéristiques de ce concept.

2.6.3 Conception

La conception d'un DSL consiste, essentiellement, à définir sa syntaxe abstraite, sa syntaxe concrète et sa sémantique. Les sections suivantes introduisent ces trois activités et présentent les outils et les techniques utilisés pour les réaliser.

2.6.3.1 Syntaxe abstraite

La syntaxe abstraite définit les concepts, les relations et les règles de grammaire caractérisant le DSL. Ces trois aspects forment le corps du vocabulaire et de la grammaire du DSL (Clark, Sammut et Willans, 2008). Les concepts définissent les types d'éléments manipulés par le DSL alors que les relations et les règles de grammaire définissent comment ces éléments peuvent être combinés pour former des modèles valides (Greenfield et Short, 2004b).

La pertinence des concepts définis dépend fortement du niveau d'abstraction choisi. Selon Tolvanen, la partie la plus importante dans le processus de conception d'un DSL est de trouver les bonnes abstractions (Tolvanen, 2006). L'Office québécois de la langue française définit l'abstraction comme étant « une démarche de l'esprit qui consiste, au cours d'un raisonnement, à éliminer les aspects les moins pertinents de la réflexion pour ne considérer que ceux qui sont essentiels » (Office québécois de la langue française, 2002). Afin de ne garder que les concepts qui sont pertinents au DSL, il faut soigneusement déterminer le niveau d'abstraction. Un niveau trop abstrait fait qu'un DSL ne pourra pas fournir suffisamment d'informations dans ses modèles. Un niveau d'abstraction trop détaillé, d'autre

part, peut mener à un DSL compliqué, confus et difficilement compréhensible par les utilisateurs. La question qui s'impose à ce niveau est : comment définir les bonnes abstractions ? Selon, J.Kramer (Kramer, 2007), cela dépend, en fait, de la capacité à effectuer des pensées abstraites. L'auteur va au-delà pour discuter même la possibilité d'enseigner la réflexion abstraite. Mais, d'une manière générale, le niveau et la valeur d'une abstraction est déterminé par la finalité de son utilisation (Kramer, 2007). Plus cette finalité est claire et explicite, plus le processus du filtrage des abstractions devient facile.

Deux approches sont principalement utilisées pour la définition de la syntaxe abstraite (Greenfield et Short, 2004b) : la grammaire non-contextuelle (*Context-free grammars - CFG*), utilisée surtout pour la définition des langages textuels, et la métamodélisation, approche plus moderne utilisée particulièrement pour la définition des langages de modélisation graphiques. Cette idée préconçue sur l'utilisation de ces deux approches n'est pas vraiment justifiée. Greenfield (Greenfield et Short, 2004b) argue qu'il n'y a pas de bonnes raisons qui empêchent l'utilisation de la grammaire non-contextuelle pour la définition de langages graphiques ou l'utilisation de l'approche de métamodélisation pour la définition de langages textuels. (Alanen et Porres, 2004; Xia et Glinz, 2002; 2003) proposent des méthodes pour l'utilisation de la CFG dans la définition de langages graphiques. Pourtant, d'un point de vue pratique, on constate que l'approche de la grammaire non-contextuelle est mieux adaptée à la description des langages textuels puisqu'elle est bien optimisée pour le traitement de texte, alors que l'approche de métamodélisation se prête mieux à la description des langages graphiques de modélisation.

1) Grammaires non-contextuelles

En 1956, le linguiste Noam Chomsky a proposé une hiérarchie pour classifier les types de grammaires formelles utilisées pour la description des langages. Les grammaires non-contextuelles correspondent au deuxième type dans cette hiérarchie (Chomsky, 1956; 1959).

Une grammaire non-contextuelle, appelée aussi grammaire hors-contexte ou encore grammaire algébrique, est une grammaire formelle qui consiste en un ensemble de productions[4] de la forme $A \rightarrow B$ où ***A*** est un symbole non-terminal qui peut être substitué par n'importe quelle chaîne du vocabulaire de la grammaire et ***B*** correspond à des symboles terminaux ou non terminaux. Un symbole non-terminal est un élément composite dont la structure interne peut être détaillée sous forme d'autres symboles (terminaux ou non-terminaux). Il doit, forcément, apparaitre comme la partie gauche d'au moins une production. Or, un symbole terminal est une séquence d'un ou de plusieurs caractères formant un élément irréductible du langage (ISO, 1996).

Il existe plusieurs notations pour représenter une grammaire non-contextuelle. La plus réputée de ces notations est sans doute la notation BNF (*Backus–Naur Form/ Backus Normal Form*). Il y a aussi la norme ISO/IEC 14977 qui définit une notation syntaxique basée sur la notation BNF et sur ses extensions les plus adoptées (ISO, 1996).

BNF

Appelée initialement la forme normale de Backus (*Backus normal form*), d'après son inventeur John Backus qui l'a développée afin de décrire la grammaire du langage ALGOL[5]. Elle est devenue, ultérieurement, la forme de Backus-Naur après que Peter Naur y ait apporté des améliorations (Knuth, 1964). Actuellement, il existe différentes extensions et variantes de cette notation. « BNF élargie » (*Extended BNF - EBNF*), « BNF augmentée » (*Augmented* BNF - ABNF), « BNF réduite » (*Reduced BNF – RBNF*) et les diagrammes syntaxiques (*syntax diagrams*) en sont quelques exemples.

---

[4] Une règle de réécriture spécifiant un symbole de substitution qui peut être effectué récursivement afin de générer de nouvelles séquences de symboles (http://en.wikipedia.org/wiki/Production_(computer_science))

[5] ***ALGO****rithmic* ***L****anguage* est un langage impératif de programmation créé à la fin des années 1950 en vue de décrire algorithmiquement des problèmes de programmation.

ISO/IEC 14977 :1996 - EBNF

La notation syntaxique décrite dans la norme ISO/IEC 14977 est basée, essentiellement, sur la notation EBNF proposée par Niklaus Wirth (ISO, 1996). EBNF se caractérise, principalement, par l'adoption d'une syntaxe proche de celles des expressions régulières. Cette syntaxe permet de faciliter la description des aspects comme la répétition et le regroupement de blocs. Elle rend également la construction de compilateurs beaucoup plus facile puisque les analyseurs syntaxiques sont automatiquement générés avec des compilateurs-compilateurs comme YACC[6] (Garshol, 2010).

2) Métamodélisation

Depuis son utilisation dans la définition du langage UML, la métamodélisation est devenue l'approche favorite pour la définition des langages de modélisation (Greenfield et Short, 2004b). L'adoption de l'approche de métamodélisation pour définir les langages dédiés présente de nombreux avantages. Le plus important de ces avantages est le fait que cette approche permet de définir les DSL de façon unifiée en utilisant le même langage de métamodélisation (*Metamodeling Language*) pour capturer, décrire et manipuler la plupart des aspects du DSL (syntaxe abstraite, syntaxe concrète et sémantique). Ajoutons à cela l'utilisation d'une notation généralement intuitive qui facilite la communication de l'intention du concepteur.

On peut définir la métamodélisation comme étant le processus de définition des métamodèles. Un métamodèle est un modèle (modèle de modèle), écrit avec un métalangage, qui définit un langage qui sert à construire des modèles. Dans cette perspective un modèle est considéré comme une instance du métamodèle qui a servi à la définition de son langage de modélisation. Ce mécanisme d'instanciation de modèles est au cœur de l'approche de métamodélisation et il peut être étendu sur plusieurs niveaux d'abstraction.

---

[6] Outils de génération d'analyseurs syntaxiques pour les langages non-contextuels

Il existe plusieurs frameworks pour la définition des métamodèles. Le plus réputé de ces frameworks est l'architecture de métamodélisation à quatre couches proposée par l'OMG. Cette architecture adopte une approche de métamodélisation dite stricte. Si un modèle M_0 est une instance d'un autre modèle M_1, alors chaque élément de M_0 doit être une instance d'un élément de M_1. Par exemple, le métamodèle UML est considéré comme une instance du méta-métamodèle MOF. Donc; chaque élément UML est une instance d'un élément du méta-métamodèle MOF (OMG, 2009). L'approche recommande, également, que les couches aient des limites strictes qui ne peuvent être traversées que par des relations de type « instance-de ». De plus, une relation « instance-de » n'est permissible qu'entre deux couches immédiatement adjacentes (Atkinson et Kühne, 2002).

Certes cette approche de métamodélisation stricte a permis de simplifier le processus de métamodélisation (Atkinson et Kuhne, 2003). Toutefois on y dénote certaines imperfections (Álvarez, Evans et Sammut, 2001; Atkinson et Kühne, 2002). Parmi les reproches faits à cette approche, retenons celui d'Atkinson *et al.* qui proclament qu'il n'y a pas de description précise du sens exact de la relation « instance-de ». Cela laisse, selon les auteurs, libre cours à diverses interprétations, notamment le langage UML, qu'ils considèrent avoir pris une approche très libérale dans l'interprétation des couches et des relations « instances-de ». Ainsi, ils proposent une définition qui, selon eux, donne une explication précise de ce qu'est la métamodélisation stricte. La description va comme suit : « Dans une architecture de modélisation à N niveaux (M_0, M_1, …, M_{N-1}), chaque élément d'un modèle de niveau M_K doit être une instance d'un seul élément d'un modèle M_{K+1} pour tout $0 <= K <= N-1$, et toute relation autre que la relation « instance de » entre deux éléments X et Y, implique que le niveau (X) = niveau (Y) (Atkinson et Kühne, 2002) ».

Nous pensons que quelle que soit l'approche adoptée pour la définition de la syntaxe abstraite (grammaire hors-contexte ou métamodélisation), le processus demeure à peu près le même (voir (Clark, Sammut et Willans, 2008) pour plus de détails). Il consiste, principalement, en l'indentification et la modélisation des concepts et de leurs relations ainsi qu'en la définition des règles de grammaire à utiliser pour déterminer la validité syntaxique des modèles.

Identification et modélisation des concepts

La première étape dans la définition de la syntaxe abstraite consiste à identifier les concepts à utiliser par le langage dédié. C'est une étape de distillation et de filtration de l'extrant du processus d'abstraction débuté dans la phase d'analyse de domaine. Dans cette étape, on ne prend en considération que les concepts ayant une signification du point de vue du vocabulaire du DSL et on laisse de côté les concepts relevant de la syntaxe concrète ou de la sémantique. Certes, il n'est pas toujours évident de faire cette distinction. Toutefois, l'important est de développer ce reflexe de se demander toujours la pertinence d'un concept. Par exemple, il est clair que les concepts « Étiquette » et « Commentaire » sont purement notationnels et ne sont d'aucune valeur du point de vue de la syntaxe abstraite.

Voici quelques étapes pratiques qui peuvent aider à maîtriser ce processus.

a) Définir le niveau d'abstraction

Travailler au juste niveau d'abstraction est crucial pour le succès du DSL (Tolvanen, 2006) parce qu'il affecte directement son expressivité. Généralement, le niveau d'abstraction est déterminé par la finalité de son utilisation (Kramer, 2007). C'est cette finalité qui aide à mieux gérer la pertinence des abstractions définies. Dans le cas des DSL, le niveau d'abstraction est déterminé en respectant deux contraintes principales : **1)** construire un DSL expressif et d'un fort niveau d'abstraction ; autrement dit, un DSL présentant des concepts proches du problème du domaine et dont les modèles sont facilement compréhensibles par les experts du domaine et **2)** possibilité de générer du code (ou des modèles intermédiaires) à partir des modèles définis.

b) Construire une liste de concepts candidats

Cette activité consiste à préparer une liste contenant les concepts les plus significatifs pour le DSL. L'identification des concepts est considérée comme une activité créative qui dépend, en grande partie, de l'habileté et de l'expertise des analystes. Néanmoins, nous allons mettre en lumière certaines des techniques et des pratiques qui peuvent aider dans ce processus.

Selon Tolvanen (Tolvanen, 2006), l'identification des concepts commence, souvent, à partir d'un certain point de vue. Il propose cinq points de vue comme les plus communément utilisés pour l'identification des concepts :

- **Structure physique du produit :** utile, particulièrement, lorsqu'on modélise des objets physiques. Les concepts décrivent, dans ce cas, la structure physique de l'objet à modéliser. Les DSL basés sur ce type de point de vue se concentrent essentiellement sur l'aspect statique des objets modélisés ;
- **Aspect et convivialité du système (*look-and-feel*) :** les concepts sont identifiés selon le point de vue de ses utilisateurs en se basant sur les interfaces, la navigation et les interactions avec le système ;
- **Espace de variation :** les concepts sont identifiés en se basant sur l'espace de variation de la famille de produits. Les DSL définis de cette façon sont utilisés, essentiellement, pour spécifier la configuration des membres ;
- **Concepts des experts :** les concepts sont identifiés selon le point de vue des experts du domaine. Ils émanent principalement de la terminologie et du vocabulaire utilisés par les experts du domaine afin de monter en abstraction et améliorer l'expressivité du langage ;
- **Output :** les concepts sont identifiés en se basant sur le produit généré. Par exemple, si ce produit est un fichier au format XML, alors on peut utiliser le schéma XML de ce fichier comme source pour l'identification des concepts du DSL.

Après avoir déterminé le point de vue approprié, l'analyste peut recourir aux techniques habituelles d'analyse et de conception orientées objets (Fontoura, Pree et Rumpe, 2000; Larman, 2004) pour identifier les concepts de son DSL. Voici une brève description des techniques que nous jugeons utiles :

- **Réutilisation :** afin d'éviter de réinventer la roue, il est préférable, avant de commencer l'identification des concepts, de vérifier s'il y a déjà une analyse de domaine qui a été faite pour le domaine en question afin de profiter des expériences des autres ;

- **Identification itérative :** cette technique est particulièrement intéressante lorsqu'on ne connaît pas suffisamment le domaine. Elle permet, à travers les différentes itérations, d'apprendre plus sur le problème et sur la solution (dans ce cas le DSL). Le nombre d'itérations dépend, généralement, de la taille et de la complexité du domaine ;
- **Noms et phrases nominale :** cette technique a été proposée dans (Abbott, 1983). Elle consiste à extraire les noms et les phrases nominales à partir des descriptions du domaine. Cependant, il faut noter que ces noms ne constituent qu'une liste de concepts candidats qu'il va falloir affiner afin de ne garder que les concepts pertinents au DSL. Cette activité d'épuration est nécessaire vu l'ambigüité que peut porter le langage naturel ;
- **Identification dirigée par les archétypes :** cette technique consiste à identifier les concepts du domaine en se basant sur les archétypes[7] définis par Coad *et al.* (Coad, de Luca et Lefebvre, 1999). Selon eux, tous les concepts d'un domaine appartiennent à l'un des quatre archétypes suivants : moment-intervalle, rôle, description ou partie. Ces archétypes définissent les attributs, les opérations et les interactions qui sont typiques aux classes qui en font partie (COAD, 1992; Coad, North et Mayfield, 1997). Chacun des archétypes est exprimé avec une couleur différente. Ces couleurs introduisent une technique de conception graphique appelée superposition (*Layering*).

Selon les auteurs, la technique de superposition permet d'exprimer des couches additionnelles d'information et d'offrir une meilleure lisibilité des modèles. Les couleurs utilisées donnent immédiatement des indications sur le contenu du modèle en permettant au lecteur de diviser instantanément le modèle en quatre couches visuelles (Coad, de Luca et Lefebvre, 1999).

Les quatre archétypes sont :

- **Moment-intervalle (Rose) :** représente un événement ou une activité qui se produit, à un moment ou dans un intervalle donné. Par exemple, la location d'une voiture ou l'emprunt

[7] Un archétype est un modèle général représentatif d'un sujet (i.e. sujet général) qui constitue un fonds commun pour la définition de sujets particuliers affiliés à ce sujet.

d'un livre d'une bibliothèque se font sur un intervalle de temps, tandis que la vente d'un article est faite à un moment donné ;

- **Rôle (Jaune) :** représente la participation d'une partie (personne ou organisation), d'un lieu ou d'une chose à un événement ou à une activité ;
- **Description (Bleu) :** représente un concept susceptible d'être utilisé comme une entrée dans un catalogue. La description du concept est fournie par les valeurs de ses attributs, d'où le nom de l'archétype ;
- **Partie, place ou chose (Vert) :** représente un groupe, un lieu, ou une chose qui joue un ou plusieurs rôles.

Dans cette même perspective, Larman a aussi proposé une liste de catégories de classes qui contient, selon lui, un grand nombre de catégories généralement sollicitées par les développeurs (voir (Larman, 2004) pour plus de détails). Des patrons d'analyse, tels ceux proposés par Fowler (Fowler, 1997)), peuvent également être utilisés dans l'identification des concepts.

Définir les règles de grammaire

La deuxième étape dans le processus de définition de la syntaxe abstraite est la définition des règles de grammaire qui déterminent la validité des modèles. Ces règles sont particulièrement utiles quand il s'agit de mettre en œuvre un outil pour vérifier l'exactitude et la validité des modèles du DSL. Dans ce cas il faut exprimer ces règles avec un langage de contraintes comme OCL. Toutefois, dans la pratique, on constate que ces règles sont souvent exprimées informellement (Clark, Sammut et Willans, 2008) avec des langages naturels.

2.6.3.2 Syntaxe concrète

Alors que la syntaxe abstraite se concentre sur la description sous-jacente du vocabulaire et de la grammaire du DSL, la syntaxe concrète, quant à elle, s'occupe de la définition de la notation à utiliser pour représenter les éléments du DSL dans les différents modèles (Greenfield et Short, 2004b). Cette notation peut prendre plusieurs formes. Elle peut être textuelle, graphique ou une combinaison des deux. La syntaxe concrète textuelle est utilisée,

principalement, pour les DSL dits « de programmation » alors que la syntaxe concrète graphique est plutôt réservée aux DSL dits « de modélisation ». Les DSL combinant les deux formes de notations utilisent généralement la notation graphique pour représenter des vues d'un plus haut niveau d'abstraction, et la notation textuelle pour capturer des informations plus détaillées (Clark, Sammut et Willans, 2008).

Syntaxe concrète textuelle

La syntaxe concrète textuelle définit les modèles (programmes) du DSL sous forme de structures textuelles, soit des chaînes de caractères représentants des déclarations de structures de données (e.g. variables, objets, etc.), ou des expressions et des instructions manipulant ces structures et contrôlant leur flux d'exécution. L'avantage principal de la syntaxe textuelle est sa capacité à capturer des expressions complexes. Cependant, au-delà d'un certain nombre de lignes, ces expressions textuelles deviennent difficiles à comprendre et à gérer.

Habituellement, la syntaxe concrète textuelle est spécifiée en utilisant les notations de la famille BNF. L'avantage de cette approche est qu'elle permet de profiter des outils de génération d'analyseurs lexicaux et d'analyseurs de syntaxe. L'analyseur lexical permet l'identification et la séparation des unités lexicales (*tokens*) alors que l'analyseur de syntaxe convertit ces séquences d'unités lexicales en un arbre de syntaxe abstraite (*Abstract Syntax Tree (AST)*).

Syntaxe concrète graphique

Peu de travaux ont été consacrés à la définition rigoureuse de la syntaxe concrète graphique (Greenfield et Short, 2004b). Celle-ci est souvent définie informellement (Fondement et Baar, 2005). La syntaxe concrète graphique décrit les éléments graphiques (symboles) à utiliser pour représenter les modèles du DSL et détermine leurs dispositions géométriques. Elle consiste en une interface utilisateur graphique permettant de manipuler (spécifier, ajouter, retirer, lier, etc.) les concepts du DSL. Chaque élément représentable du langage est associé à un symbole graphique. L'avantage principal d'une syntaxe visuelle est sa capacité à

exprimer l'information sous une forme intuitive et facile à comprendre (Clark, Sammut et Willans, 2008). Cependant, sa faiblesse évidente est qu'elle ne peut exprimer ces informations qu'à un certain niveau d'abstraction. Au-delà de ce niveau, elle devient trop complexe et incompréhensible.

2.6.3.3 Sémantique

Généralement, la syntaxe abstraite ne comporte pas assez d'information pour définir le sens des éléments construits du DSL (*DSL constructs*). Des informations supplémentaires sont nécessaires afin de pouvoir déterminer ce sens. Le but de la sémantique est de capturer et d'organiser l'ensemble de ces informations dans des modèles sémantiques décrivant le sens du DSL.

La compréhension du sens du DSL est indispensable pour la définition de DSL exécutable avec des modèles faciles à analyser et à transformer. Les techniques de transformation et de génération de code dépendent fortement de la clarté et de la précision de la sémantique. Chaque expression syntaxiquement valide dans le DSL doit avoir une signification précise et sans équivoque (Harel et Rumpe, 2000) afin de lever toute ambiguïté concernant la compréhension des modèles et de parer aux risques de contresens et d'usage inadéquat (Clark, Sammut et Willans, 2008).

La sémantique d'un langage en général, et d'un DSL en particulier, consiste en deux parties : le domaine sémantique ($\boldsymbol{S_D}$) et le mappage sémantique ($\boldsymbol{S_M}$) (Harel et Rumpe, 2000). Le domaine sémantique définit l'ensemble des significations possibles qui peuvent être accordées aux différentes expressions autorisées par le DSL. Le mappage sémantique établit le lien entre les expressions définies dans le domaine syntaxique (e.g. ensemble des éléments constituant la syntaxe abstraite et/ou la syntaxe concrète du DSL) et leurs significations correspondantes dans le domaine sémantique (voir Figure 2.1).

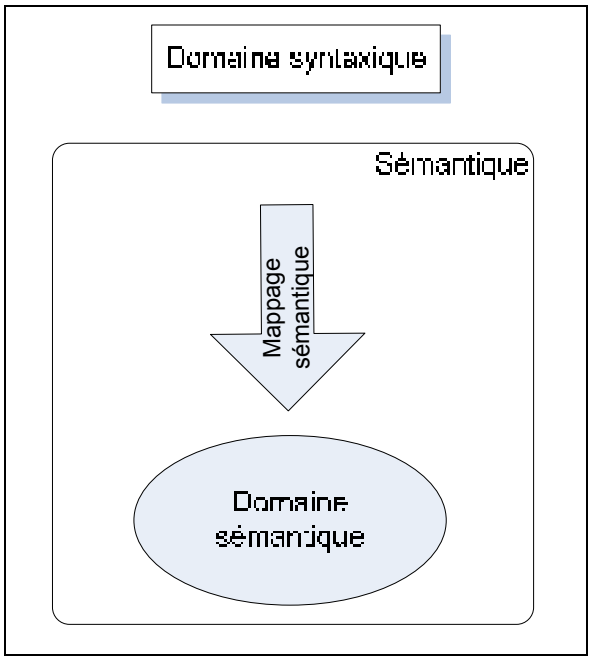

Figure 2.1 Composition de la sémantique.

En pratique, les même approches utilisées pour la définition de la sémantique des langages de programmation (sémantique algébrique, opérationnelle, dénotationnelle, axiomatique, etc.) peuvent être employées pour la définition de la sémantique des DSL. Toutefois, il faut faire la distinction entre les DSL de programmation et les DSL de modélisation. La sémantique de ces derniers est considérée plus complexe et plus exigeante. Par conséquent, elle peut demander des approches différentes et des techniques supplémentaires. Parmi les styles utilisés pour définir la sémantique d'un DSL, on cite :

Sémantique algébrique : consiste en la spécification algébrique des données et des éléments du langage. Ce style de sémantique est, habituellement, utilisé pour la spécification des types abstraits de données (e.g. spécification d'un ensemble de données et des opérations qui peuvent être effectuées sur ces données) (Slonneger et Kurtz, 1995).

Sémantique opérationnelle : décrit le comportement dynamique des éléments du DSL en spécifiant comment un programme est interprété en termes d'étapes d'exécution.

Sémantique dénotationnelle : appelée à l'origine sémantique mathématique (Slonneger et Kurtz, 1995), cette approche décrit le sens d'un langage en construisant des objets mathématiques (dénotations représentées généralement sous forme de valeurs mathématiques comme des nombres et des fonctions (Schmidt, 1986)) décrivant le sens de ses expressions. À la différence de la sémantique opérationnelle, la sémantique dénotationnelle se concentre sur l'effet du programme plutôt que sur ses étapes d'exécution.

Sémantique par traduction : cette approche consiste à traduire les expressions d'un langage « source » en expressions d'un autre langage « cible » dont la sémantique est connue. Dans le cas des DSL, cette approche est utilisée pour traduire les expressions du DSL en expressions d'un autre DSL ou d'un langage généraliste.

2.6.4 Implémentation

L'implémentation des DSL consiste en le développement des outils de transformation des modèles et de génération de code. Mernik *et al.* ont identifié sept patrons pour l'implémentation des DSL et un ensemble de directives aidant à choisir la technique d'implémentation la plus appropriée en fonction des besoins (Mernik, Heering et Sloane, 2005). Dans la littérature, on distingue deux familles de techniques : les techniques pour les DSL autonomes et celles pour les DSL enchâssés (voir section 2.3). Un DSL autonome est généralement implémenté en créant des outils de support : compilateur, interprète, préprocesseur ou générateur de code. L'approche habituellement adoptée pour créer de tels outils est d'utiliser des méta-compilateurs (compilateurs de compilateurs) comme Lex (Lesk, 1975) et Yacc (Johnson, 1974), Flex (Paxson, 1988), Bison (Free Software Foundation, 2009) ou ANTLR (Parr, 2007). Ce sont des outils capables de générer automatiquement du code permettant l'analyse lexicale, l'analyse syntaxique et la gestion des erreurs à partir d'une description formelle (généralement une grammaire) du DSL. Le reste de la construction, à savoir, l'analyse sémantique, est laissé à la charge du concepteur du DSL. Celui-ci doit implémenter les actions selon la sémantique définie pour le DSL.

À la différence des DSL autonomes, l'implémentation des DSL enchâssés ne requiert pas la création de compilateurs pour traduire les expressions du DSL. La syntaxe d'un DSL enchâssé est conforme à la syntaxe de son langage hôte. Dans ce cas, les DSL enchâssés bénéficient des éléments et des outils de support des langages qui les hébergent. Cependant, comme pour les DSL autonomes, l'implémentation de la sémantique est laissée à la charge du concepteur du DSL.

Les approches adoptées pour l'implémentation des DSL enchâssés dépendent fortement des caractéristiques et des fonctionnalités offertes par les langages hôtes. Par exemple le langage C++ est reconnu par son mécanisme de méta-programmation par gabarits, la surcharge d'opérateurs et les macros. Le langage Lisp se caractérise par son système de macros lui permettant des possibilités d'extensions de syntaxe. Le langage Ruby se caractérise par sa flexibilité de syntaxe, son système de macros, son typage dynamique et son mécanisme de fermeture (*Closure*) et de fonction Lambda[8]. Les développeurs sont alors contraints de travailler dans le cadre des possibilités offertes par le langage hôte. Ainsi, le choix du langage généraliste hébergeant le DSL devient une décision déterminante pour le succès du DSL. Voici une liste des caractéristiques et des techniques les plus communément utilisées dans l'implémentation des DSL enchâssés :

Méta-programmation à la compilation (*Compile time Meta-programming*) (Czarnecki et al., 2004; Sheard et Jones, 2002) : la méta-programmation est la technique de programmation qui permet d'écrire des méta-programmes. Un méta-programme est un programme qui peut analyser, générer ou manipuler d'autres programmes qu'il considère comme des données. La technique de méta-programmation est implémentée dans les langages de programmation de différentes façons :

[8] Une fonction lambda est une fonction qui peut être stockée dans une variable et passée en argument à d'autres fonctions ou méthodes. Une fermeture est une fonction lambda ayant des références à des variables libres de son environnement lexical.

- Macros : mécanisme par lequel une partie du code est générée en utilisant des techniques de manipulation et de substitution de texte. L'utilisation de ce mécanisme implique trois étapes : la macro-définition associant un identificateur à un texte de remplacement, la macro-instruction faisant appel à la macro, et la macro-expansion substituant la macro-instruction par le texte de remplacement.

  Le système de macros du langage Lisp est un exemple qui montre la puissance et le potentiel de cette technique. Ce système est utilisé intensivement pour étendre la syntaxe du langage Lisp. Ainsi, il constitue une facilité certaine pour la définition des DSL.

- Template : le principe est de définir un modèle (*Template*) générique représentant la structure générale du code avec des paramètres qui seront substitués lors de l'exécution.

Surcharge d'opérateurs : cette caractéristique désigne la capacité de surcharger les opérateurs prédéfinis d'un langage. Ce mécanisme est largement exploité dans le langage C++ pour modifier la sémantique de ses opérateurs prédéfinis.

Réflexivité : Un langage de programmation est dit réflexif s'il est apte à créer des programmes capables d'examiner et de modifier leur structure (réflexion structurelle) et leur comportement (réflexion comportementale). La réflexion structurelle permet la manipulation du code et des types abstraits d'un programme pendant son exécution. La réflexion comportementale permet à un programme d'obtenir des informations sur son implémentation, et éventuellement de se réorganiser afin de s'adapter à son contexte d'exécution. La capacité de pouvoir intervenir et changer dynamiquement le comportement et/ou la structure des programmes ouvre un éventail de possibilités pour l'implémentation des DSL. En effet, la réflexivité peut être utilisée pour générer du code spécifique et l'intégrer automatiquement au sein du programme exécuté.

Flexibilité de la syntaxe : désigne la capacité d'un langage à étendre sa syntaxe pour s'adapter au domaine du problème. Cette caractéristique permet au compilateur de prendre en charge de nouvelles syntaxes sans modifier aucun de ses composants. Lisp, Smalltalk et

Ruby sont des exemples de langages utilisant cette capacité pour étendre leurs syntaxes et en construire des DSL.

Sucre syntaxique (*Syntactic Sugar*) : capacité d'un langage à "sucrer" sa syntaxe en la rendant plus succincte, plus élégante et plus expressive.

Cette liste de caractéristiques n'est pas exhaustive. En fait, chaque langage de programmation se distingue par son propre ensemble de caractéristiques qui le différencient des autres. Parmi les autres caractéristiques qui se sont montrées également efficaces pour le développement des DSL on trouve les fonctions lambda, les fermetures (*Closures*) et les instructions d'évaluation de chaînes (ex. *eval*, *with* de Javascript). Pour cette raison, nous recommandons une analyse préalable examinant le potentiel et les possibilités du langage hôte qu'on désire utiliser pour héberger le DSL.

2.7 Sommaire et synthèse

Les langages dédiés sont réputés pour leur expressivité et leur fort niveau d'abstraction. Ils sont utilisés dans différents domaines pour permettre aux experts de domaine de concevoir des solutions en utilisant des concepts et des notations qui leur sont familiers. Or, il reste encore des contraintes qui renforcent l'hésitation des organisations envers cette technologie et empêchent son adoption à plus grande échelle.

Dans ce chapitre nous avons discuté du concept des DSL et montré la difficulté de parvenir à une définition précise et consensuelle pour le terme DSL. Les notions de domaine, d'exécutabilité et de spécificité sont les principaux points de divergence entre les définitions proposées. Ajoutons à cela la diversité des types et des propriétés des DSL qui rend difficile la caractérisation de ces langages.

La deuxième contrainte est liée aux outils de développement. Ces outils adoptent des approches propriétaires et peu intuitives, ce qui augmente la pente de la courbe d'apprentissage et rend l'interopérabilité de ces outils un véritable défi. Le développement

des DSL doit être basé sur des méthodes et techniques standardisées afin d'assurer l'interopérabilité et la portabilité des DSL.

La contrainte liée au processus de développement est certainement l'une des contraintes les plus importantes. Le développement des DSL est une tâche difficile qui demande une connaissance du domaine et des compétences en développement des langages. À la différence des langages généralistes qui bénéficient d'un ensemble substantiel de théories et d'expérience, les langages dédiés manquent toujours de méthodes et de processus efficaces pour soutenir leur conception (Selic, 2007).

Dans ce chapitre, une synthèse de la littérature a été réalisée afin d'identifier les activités impliquées dans un processus de développement de DSL. Trois activités ont été considérées comme principales : analyse, conception et implémentation. La Figure 2.2 récapitule ce processus en spécifiant les intrants et les extrants de chacune des activités.

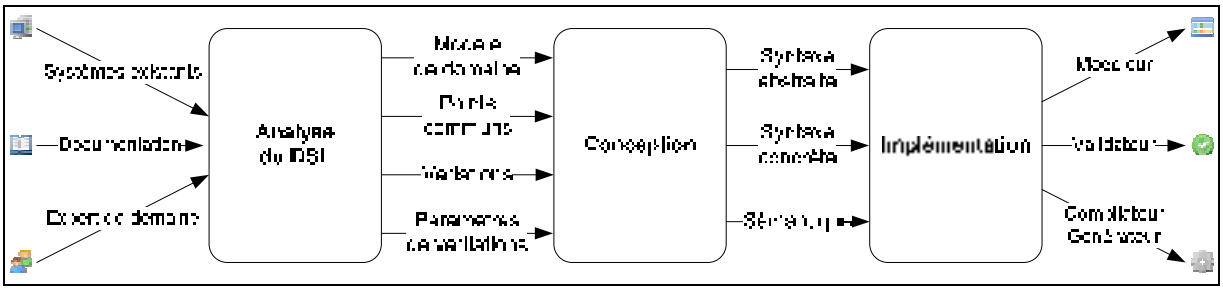

Figure 2.2 Processus de développement des DSL.

CHAPITRE 3

ISO/IEC 24744:2007

3.1 Vue d'ensemble

Les méthodes de développement jouent un rôle crucial dans le développement de systèmes à prépondérance logicielle. Malheureusement, les approches utilisées actuellement pour le développement de ces méthodes ne se montrent pas très concluantes (Gonzalez-Perez et Henderson-Sellers, 2007a). Plus encore, certaines de ces approches comprennent des concepts qui sont mal définis, voire parfois contradictoires. Gonzalez-Perez et Henderson-Sellers (Gonzalez-Perez et Henderson-Sellers, 2007a) affirment que la norme ISO/IEC 24744 est le seul standard disponible capable de spécifier, lors de la modélisation d'une méthode, le processus à suivre, les artéfacts à produire et les personnes impliquées.

La norme ISO définit un métamodèle très générique mais qui permet tout de même de créer des méthodes riches et expressives. Ce métamodèle, dénommé SEMDM (*Software Engineering-Metamodel for Development Methodologies*), a été défini dans une perspective d'être utilisé dans trois contextes différents. Chacun de ces contextes définit un domaine d'expertise bien déterminé qui correspond à un certain niveau d'abstraction dans le processus de développement des méthodes (ISO, 2007b). Ces trois domaines d'expertise sont : **1)** le domaine métamodèle qui comporte le SEMDM et ses éventuelles extensions, **2)** le domaine méthode où les ingénieurs de méthodes définissent leurs méthodes, et **3)** le domaine d'application (*Endeavour Domain*) qui représente le domaine d'utilisation de la méthode. La Figure 3.1 montre les relations entre ces trois niveaux.

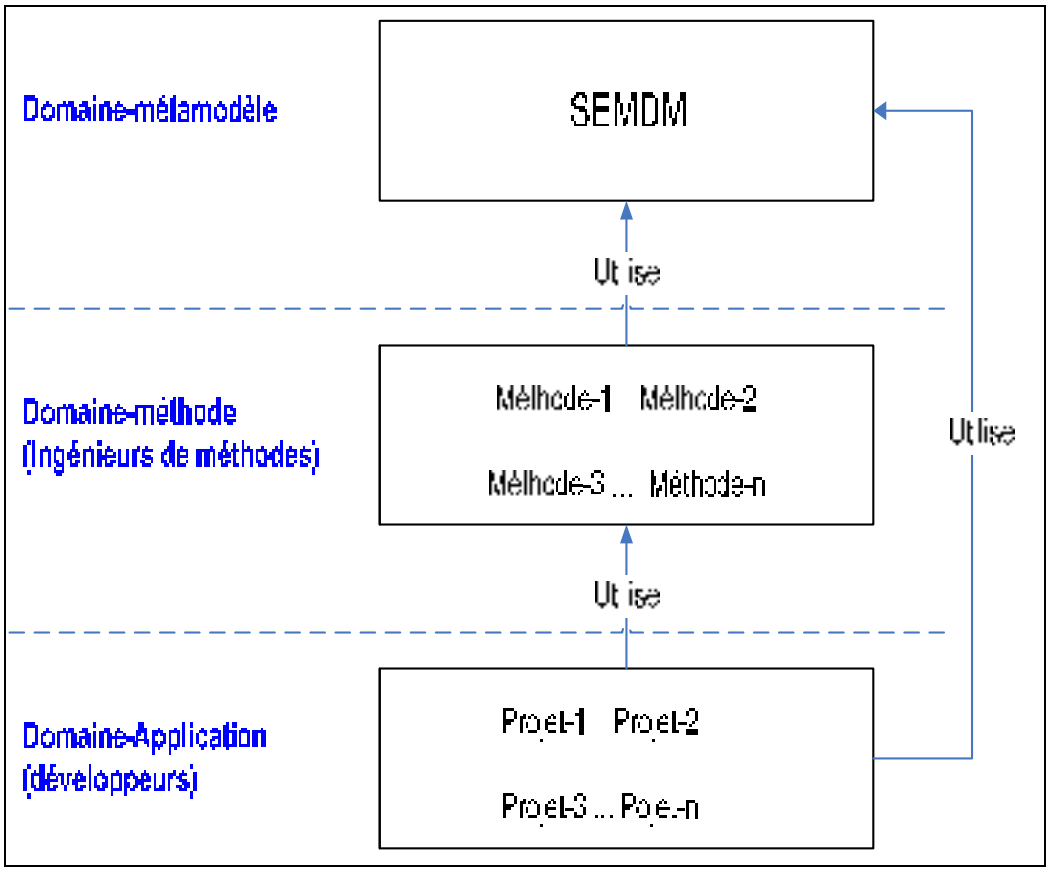

Figure 3.1 Contextes (domaines) définis par la norme ISO/IEC 24744.

Les niveaux sont reliés entre eux par des relations d'utilisation. Ainsi, les ingénieurs de méthodes utilisent, via un mécanisme d'instanciation, les éléments du domaine métamodèle et, pareillement, les développeurs utilisent, dans le contexte de leurs projets, les éléments définis dans les domaines méthode et/ou métamodèle.

En plus de cette architecture à trois niveaux, la norme ISO/IEC 24744 introduit deux concepts clés afin de pallier aux inconvénients des architectures de métamodélisation conventionnelles (Atkinson, 1997; Atkinson et Kühne, 2001), notamment le problème de dualité posée par l'architecture à quatre couches (*Four-layer Architecture*) adoptée par l'OMG. Ce problème se manifeste lorsque les éléments définis dans les couches intermédiaires, agissent à la fois comme des objets (lorsqu'ils sont le résultat d'une

instanciation des éléments de la couche supérieure) et comme des classes (lorsqu'ils font l'objet d'une instanciation). Les deux concepts permettant de contourner ce problème sont le *powertype* et le *clabject*.

Powertype

Un powertype ***P*** d'un autre type ***T***, appelé type partitionné, est un type dont les instances sont des sous-types du type partitionné (ISO, 2007b). C'est la définition donnée à un powertype dans la norme ISO/IEC 24744. Henderson *et al.* (Henderson-Sellers et Gonzalez-Perez, 2005), arguent que cette définition, originairement donnée par Odell (Odell, 1994), a été insuffisante pour régler le problème de dualité posé par les approches de métamodélisation. En effet, les instances du powertype qui sont normalement des objets ne peuvent être en même temps des sous-types (classes) du type partitionné. Henderson *et al.* préfèrent plutôt la définition suivante : « Un powertype ***P*** d'un autre type ***T*** est un type dont les instances sont des facettes objets dans des clabjects dont les facettes classes sont des sous-types du type ***T***» (Henderson-Sellers et Gonzalez-Perez, 2005). Dans cette définition on remarque l'introduction du concept de clabject. C'est ce concept qui permet de parer au problème de dualité des éléments définis dans les niveaux intermédiaires des architectures de métamodélisation multi-niveaux.

Clabject

Le mot « clabject », originairement créé par Atkinson et Kühne (Atkinson et Kühne, 2000), réfère à une entité double qui peut agir, à la fois, comme une classe et comme un objet en présentant deux facettes (Gonzalez-Perez et Henderson-Sellers, 2007b) (voir Figure 3.2) : une facette classe représentant la définition du type et une facette objet contenant les valeurs des attributs et les instances des méthodes de ce type (Atkinson, 1999). Le lien entre ces deux facettes est établi par l'attribut **discriminant** (*discriminator*). Un discriminant est un attribut dont les valeurs affectées à chacune des instances de la classe powertype sont uniques. Ces même valeurs seront utilisées pour nommer les sous-classes du type partitionné correspondant (ISO, 2007b).

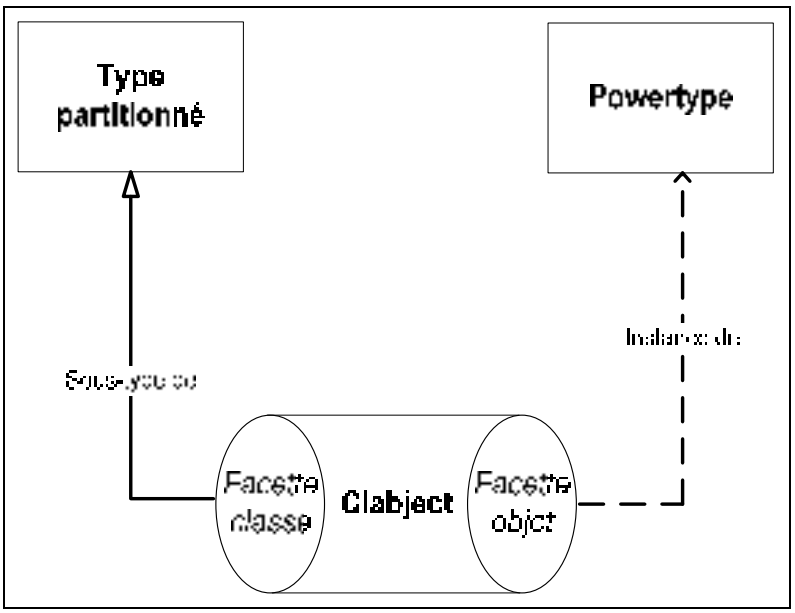

Figure 3.2 Composition d'un clabject.

Patron de powertype

Un patron de powertype consiste en une paire de classes dont l'une représente le powertype et la deuxième le type partitionné (Henderson-Sellers et Gonzalez-Perez, 2005). Cette paire de classes permet de modéliser aussi bien les concepts du niveau méthodologie que ceux du niveau application. La facette objet permet à l'ingénieur de méthode de définir les éléments constitutifs de sa méthode (niveau méthode), tandis que la facette classe sert à l'instanciation des objets du niveau application.

La norme ISO/IEC 24744 utilise deux notations pour représenter les patrons de powertype : une notation textuelle de la forme *Element/*Kind* et une notation graphique (voir Figure 3.3). Textuellement, l'expression *Element/*Kind* réfère à un patron de powertype dont le powertype est une classe appelée *ElementKind* et dont le type partitionné est une classe appelée *Element*. Graphiquement, un patron de powertype est présenté par une ligne

pointillée entre le powertype et le type de partition avec un point sur le côté de la classe powertype.

Figure 3.3 Notation d'un patron de powertype.

Le développement d'une méthode conforme à la norme ISO/IEC 24744 passe principalement par l'instanciation des patrons de powertype offerts par le métamodèle SEMDM. L'instance d'un patron de powertype est un clabject dont la facette objet est une instance de la classe powertype (classe avec le suffixe Kind) et la facette classe est un sous type du type partitionné.

Alors que la plupart des métamodèles destinés au développement des méthodologies sont fortement axés sur les processus (Gonzalez-Perez, 2007; ISO, 2007b), SEMDM se distingue par son intégration uniforme des trois aspects majeurs d'une méthodologie : le processus à suivre, les unités de travail à produire et à utiliser, et les intervenants impliqués (personnes, outils). De plus, SEMDM adopte le concept du niveau de capacité qui permet à une organisation ou à une équipe d'évaluer sa capacité à incorporer les pratiques préconisées par la méthode développée. L'ingénieur de méthode a la possibilité d'établir le niveau de capacité minimal auquel chaque type d'unité de travail peut être effectué. La norme ISO/IEC 24744 adopte une échelle à six niveaux allant de 0 à 5 : incomplet (0), réalisé (1), géré (2), défini (3), maîtrisé (4) et en optimisation (5).

3.2 Structure du SEMDM

SEMDM classe les composants (éléments) d'une méthodologie en trois catégories : les éléments *Endeavour* qui représentent les éléments à créer par le développeur dans le cadre d'un projet, les éléments *Template* qui représentent tout élément créé par l'ingénieur de méthode pendant la construction d'une méthodologie, et les éléments *Resource* qui représentent les éléments de la méthodologie utilisés directement au niveau du projet sans instanciation (ex. : références, directives). Ces trois types d'éléments fournissent toutes les classes nécessaires pour s'y prendre avec les trois aspects des méthodologies : le processus, les produits et les producteurs.

3.2.1 Processus

L'aspect processus est défini dans la norme ISO/IEC 24744 par les powertypes *WorkUnit/*Kind* et *Stage/*Kind*. Le powertype *WorkUnit/*Kind* représente l'aspect "effort" du processus qui se concentre sur le travail à réaliser. La classe *WorkUnitKind* spécifie ce qui doit être fait sans faire référence au cadre temporel dans lequel s'inscrivent les efforts du processus. Elle représente les types d'unités de travail définis par la méthode. Chacun de ces types est caractérisé par un objectif dans le projet et un niveau de capacité.

Le powertype *Stage/*Kind*, d'autre part, permet de spécifier la structure temporelle de la méthodologie en déterminant les blocs de temps (intervalles de temps ou moments instantanés (*point in time*)) gérés dans un projet. La classe *StageKind* représente les types d'étapes (*Stages*) définis par la méthode. Chacun de ces types est caractérisé par son niveau d'abstraction et le résultat produit.

Le lien entre le coté effort et le coté temporel est assuré par une relation qui lie *Process/*Kind* et *StageWithDuration/*Kind*. La relation exprime le fait qu'un processus peut-être exécuté à l'intérieur d'une étape avec durée (voir Figure 3.4).

3.2.2 Producteur

L'aspect Producteur est défini dans la norme ISO/OEC 24744 par le powertype *Producer/*Kind*. Ce dernier est relié au powertype *WorkUnit/*Kind* via le powertype *WorkPerformance/*Kind* afin de mettre en relation les producteurs et les unités de travail qui leur sont assignées (voir Figure 3.4).

La classe *ProducerKind* représente un type spécifique de producteur (personnes, groupes de personnes ou outils) dont la responsabilité est d'exécuter une ou plusieurs unités de travail. Un producteur peut jouer plusieurs rôles et peut être impliqué dans diverses unités de travail.

3.2.3 Produit

L'aspect produit est défini dans la norme par les powertypes *WorkProduct/*Kind*, *ModelUnit/*Kind* et *ModelUnitUsage/*Kind*. Les deux derniers powertypes représentent, spécifiquement, l'aspect modélisation des produits.

La classe *WorkProductKind* représente un type spécifique de produits (documents, matériels ou collections d'informations) caractérisé par la nature de son contenu et l'intention de son utilisation. Un produit fait toujours l'objet d'une ou de plusieurs actions.

La classe *ModelUnitKind* représente un type d'unité de modèle (*ModelUnit*). Un *ModelUnit* est un élément atomique d'un modèle représentant un fragment cohésif d'information dans le sujet modélisé. Chaque *ModelUnitKind* est caractérisé par la nature de l'information qu'il représente et par l'intention de l'utilisation d'une telle représentation. Le powertype *ModelUnitUsage/*Kind* représente l'usage d'une unité de modèle par un modèle donné. Un *ModelUnitUsage/*Kind* appartient toujours à un modèle qui représente son contexte et se réfère à une unité de modèle qui constitue sa cible.

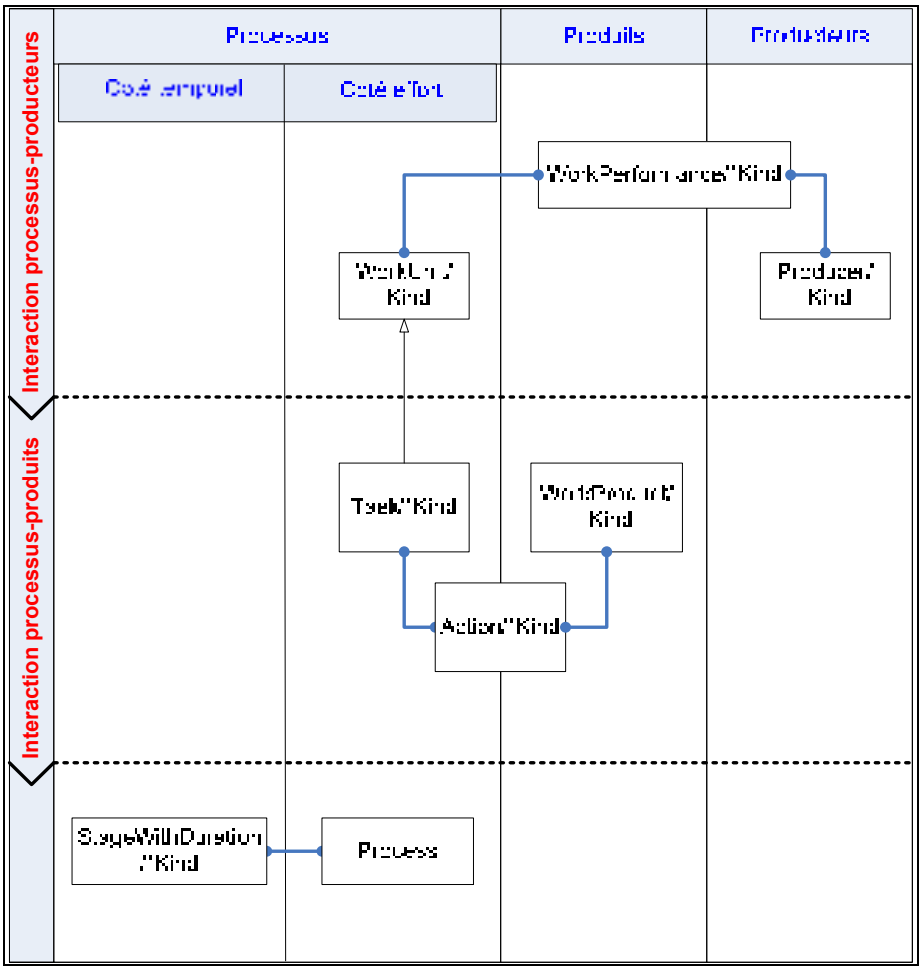

Figure 3.4 Interactions entres les trois aspects du SEMDM.

3.3 Utilisation du SEMDM

Comme déjà mentionné, SEMDM diffère des autres approches de métamodélisation par l'incorporation du concept de powertype. Ceci implique des adaptations dans la façon d'instancier une méthode à partir de ce métamodèle, et plus particulièrement l'instanciation du patron powertype. L'instance d'un patron powertype est un clabject dont la facette objet est une instance de la classe powertype et la facette classe est un sous type du type partitionné. Les classes du métamodèle qui ne font pas partie d'un patron de powertype, comme la classe « *Resource* », sont instanciées selon le mécanisme d'instanciation conventionnel et sont utilisées directement au niveau application (*Endeavour*).

SEMDM est utilisé principalement par les ingénieurs de méthodes. Ces derniers instancient les éléments du SEMDM en créant des clabjects et des objets. Les clabjects sont créés à partir des patrons de powertype, c'est-à-dire à partir des classes dérivées des classes *Template* et *Endeavour*. Les utilisateurs des méthodes créées, d'autre part, utilisent ces méthodes en créant des objets à partir des facettes classes des clabjects. Les facettes objets et les objets *Ressources* sont utilisés en tant que références.

3.4 Extension du SEMDM

La norme ISO/IEC 24744 vise à fournir un métamodèle générique et flexible afin de permettre la création de méthodes riches et expressives. Cette flexibilité est accomplie via le mécanisme d'extension qui consiste à créer des extensions (collections d'éléments du niveau métamodèle) en étendant le métamodèle standard. Ces extensions sont créées en créant des sous types des powertypes, des classes et des types énumérés standards ou en ajoutant des attributs et des associations aux classes existantes.

3.5 Notation pour la norme ISO/IEC 24744

La version actuelle de la norme ISO/IEC 24744 n'inclut pas de notations graphiques pour la représentation des méthodes. L'exemple fourni en « annexe A » de la norme et qui illustre comment créer une méthode conforme au métamodèle SEMDM n'utilise qu'une description textuelle. Pour cette raison, une annexe informative pour la notation est actuellement en développement afin d'offrir une notation graphique susceptible de représenter les différents concepts du métamodèle. Jusqu'à date et à l'exception du diagramme d'*enactment* qui est dans sa phase expérimentale, la version 2007 de cette annexe ne décrit que les concepts du domaine méthode. Ainsi, la notation couvre les trois aspects fondamentaux du SEMDM : processus, produit et producteur ainsi que les relations, les contraintes et les concepts auxiliaires définis par le métamodèle.

Les quatre types de diagrammes définis par l'annexe informative sont :

Diagramme de cycle de vie : représente la structure globale d'une méthode en décrivant son aspect temporel (*temporal aspect*) et son aspect de contenu (*content aspect*). Les éléments qui sont représentés par ce diagramme sont alors *ProcessKind* et *StageKind* (y compris *TimeCycleKind*, *PhaseKind*, *BuildKind* et *MileStoneKind*). La structure temporelle consiste en un ensemble de *StageKind* liés entre eux généralement par des liens génériques afin d'exposer la séquence des étapes du cycle de vie. Les *StageKind* peuvent aussi contenir d'autres *StageKind* afin de représenter les compositions d'éléments (relations d'agrégation entre *StageKind*) ou l'aspect itératif d'un élément. Les processus à exécuter au sein des *StageKind* sont présentés, dans la structure de contenu, en incluant les symboles *ProcessKind* à l'intérieur des symboles *StageKind*.

Diagramme de processus : décrit les détails des processus utilisés dans une méthode en se concentrant sur ce qui doit être fait (le quoi) et qui en est responsable (le qui). Ce diagramme représente, entre autres, les relations entre les types de processus et les liens reliant ces derniers aux types de tâches (*TaskKind*) et aux types de producteurs (*ProducerKind*).

Diagramme d'enactment : représente une mise en application (*Enactment*) spécifique d'une méthode, ou d'une partie de cette méthode. Ce diagramme relie les symboles du diagramme de cycle de vie à un diagramme de Gantt permettant ainsi de visualiser dans le temps les divers éléments *ProcessKind* composant un *StageKind*.

Diagramme d'action : représente les liens entre la partie processus et la partie produit en montrant les interactions entre les types de tâches et les types d'unités de travail de la méthode. Les éléments représentés dans un diagramme d'actions sont les *TaskKind*, les *WorkProductKind* et les *ActionKind.*

3.6 Conclusion du chapitre

Dans ce chapitre nous avons présenté le métamodèle pour les méthodes de développement (SEMDM) défini par la norme ISO/IEC 24744. La particularité de ce métamodèle réside dans son adoption d'une nouvelle approche basée sur le concept de powertype afin de remédier aux lacunes des approches de métamodélisation conventionnelles comme l'approche de métamodélisation stricte adoptée par le groupe OMG. SEMDM est caractérisé également par une sémantique riche permettant de modéliser les interfaces entre l'aspect processus et l'aspect produit d'une méthode. Ajoutons à cela l'adoption du concept de niveau de capacité qui permet aux ingénieurs méthodes d'établir le niveau de capacité minimale auquel chaque type d'unité de travail peut être effectué.

Bien que les avantages présentés par le métamodèle SEMDM soient indéniables, l'utilisation de celui-ci peut, parfois, s'avérer intimidante. La divergence des approches conventionnelles et l'adoption de nouveaux concepts constituent un changement de paradigme qui implique une nouvelle façon de penser et de percevoir le processus de métamodélisation. Ainsi, la courbe d'apprentissage peut être importante. D'autre part, le fait que SEMDM soit spécifié en termes de concepts purement abstraits sans offrir une notation spécialisée permettant de modéliser les méthodes de façon simplifiée, rend le processus de conception et de validation de méthodes plus difficile.

CHAPITRE 4

MÉTHODOLOGIE DE RECHERCHE

4.1 Introduction

Dans le chapitre 1 nous avons recensé les espaces technologiques reliés aux DSL et montré le rôle important que jouent ces langages pour ces technologies. Dans le chapitre 2 nous avons dressé un état d'art du développement des DSL et mis en évidence le besoin d'avoir un processus intégral pour le développement des DSL. Dans le chapitre 3 nous avons introduit la norme ISO/IEC 24744 qui va servir de cadre à la définition de la méthode qu'on propose pour la définition des DSL.

Ce chapitre 4 rappelle les objectifs de la recherche et décrit la méthodologie de recherche adoptée pour mener à terme notre travail de recherche. La description est présentée en deux étapes. D'abord une description générale exposant la méthodologie d'un point de vue global est présentée. Ensuite, une description plus détaillée démontrant les phases et les étapes entreprises pour conduire le travail de recherche est fournie.

4.2 Objectifs de la recherche

La motivation de cette recherche est issue principalement du constat de la difficulté du développement des langages dédiés et de l'absence d'un processus cohérent et clairement défini permettant de piloter le déroulement de ce développement. Le but de ce travail est de proposer une méthode pour la définition des langages dédiés. Nous souhaitons, par cette contribution, participer à rendre le développement des DSL plus intelligible et plus compréhensible.

Comme déjà évoqué dans l'introduction de cette thèse, nous avons fixé deux objectifs pour cette recherche. Un objectif principal et un objectif secondaire et exploratoire. L'objectif

principal de la recherche, pour l'énoncer clairement, consiste à développer une méthode de définition des langages dédiés, basée sur la norme ISO/IEC 24744, qui intègre les phases à suivre, les activités à exécuter, les artefacts à gérer et les acteurs impliqués dans un projet de définition de DSL.

L'objectif secondaire de la recherche porte sur l'utilisation et la qualité des DSL développés avec la méthode proposée. Dans ce deuxième volet de notre recherche nous explorons le sujet de la qualité des langages dédiés en vue d'identifier un ensemble d'attributs de qualité nous permettant d'évaluer les DSL ainsi définis.

Dans cette thèse nous utilisons le terme « définition de DSL » pour décrire le processus de conception qui consiste, essentiellement, en la définition de la syntaxe abstraite, la définition de la syntaxe concrète et la définition de la sémantique. Tous les aspects en relation avec l'implémentation des DSL, soit le développement des librairies pour implémenter la sémantique et la construction de compilateurs et de générateurs de code, sont considérés hors de la portée de cette recherche.

4.3 Démarche globale

D'un point de vue global, la démarche de recherche consiste en un processus d'ingénierie englobant un processus de recherche action[9]. Ainsi, nous avons commencé par observer et comparer les solutions existantes en matière de développement de DSL pour ensuite entamer un processus de recherche action dont le but est de développer une méthode pour la définition des DSL. La suite de cette section décrit les activités de chacun des deux processus (voir Figure 4.1).

---

[9] Pour plus de détail sur les méthodologies de recherche voir :
Adrion, W. Richards. 1993. « Research Methodology in Software Engineering, Summary of the Dagstuhl workshop on future directions in software engineering ». *ACM SIGSOFT Software Engineering Notes,* vol. 18, n° 1, p. 35-48.

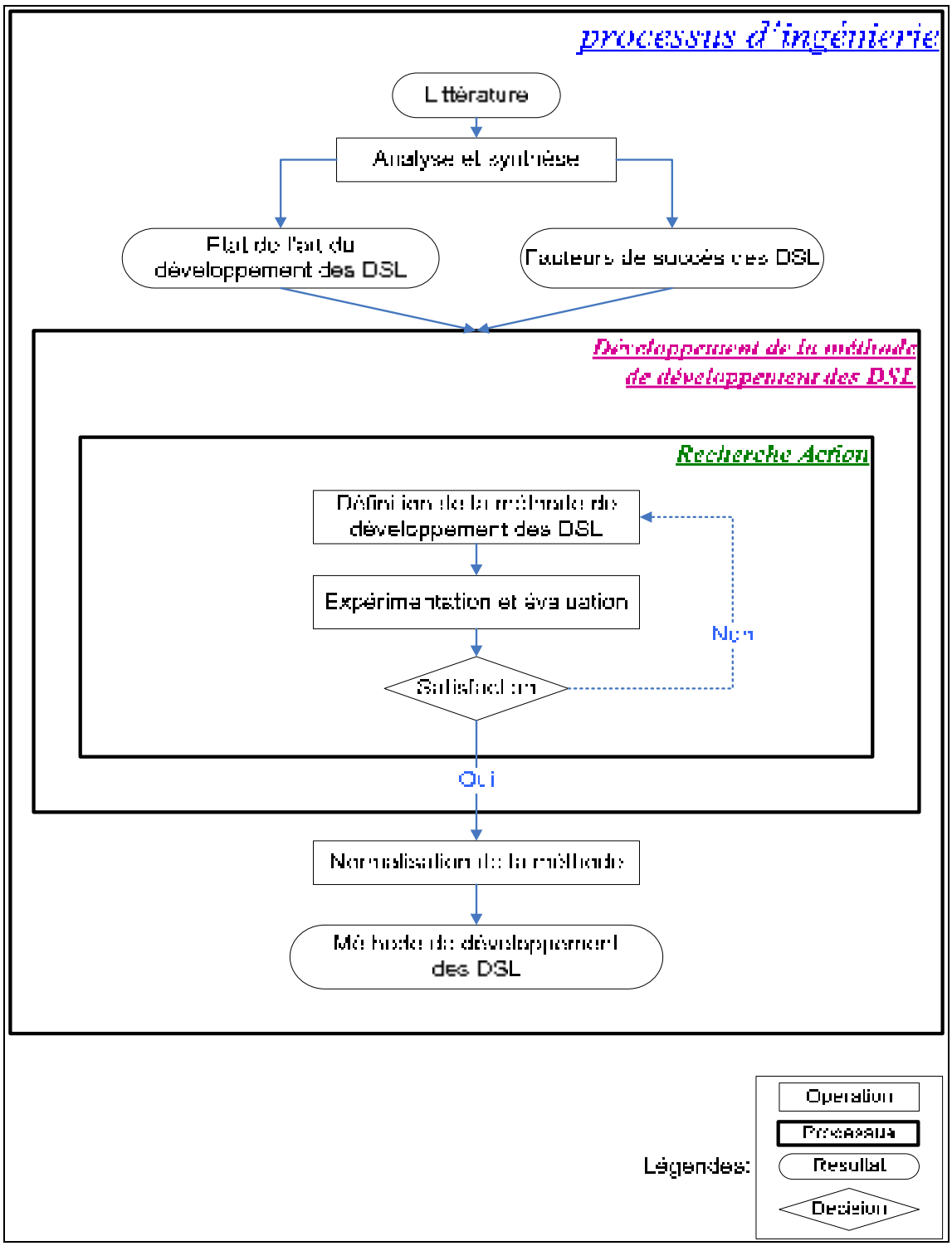

Figure 4.1 Démarche globale du travail de recherche.

4.3.1 Processus d'ingénierie

Analyse et synthèse : passer en revue la littérature, l'analyser en vue de dresser un état de l'art pour le domaine du développement des DSL. La synthèse consiste principalement à expliciter les activités principales constituant un processus de développement de DSL et à identifier les facteurs de succès des DSL qui vont servir de critères d'évaluation lors de la phase d'expérimentation de la méthode (voir section 4.4.3).

Développement de la méthode : développer une méthode pratique qui décrit la démarche à suivre lors de la définition d'un DSL : notamment les phases du cycle de développement, les activités à entreprendre et les artefacts à produire. Cette étape consiste en un processus de recherche action (voir section 4.3.2) où l'on propose, expérimente et évalue itérativement la méthode qu'on propose.

4.3.2 Processus de recherche-action

Le processus de recherche-action consiste en une approche qualitative qui combine un volet théorique et un volet pratique. Le volet théorique offre le cadre théorique de référence utilisé pour le développement de la méthode alors que le volet pratique expérimente et évalue l'applicabilité de la méthode issue de ce cadre théorique. Les activités de ce processus sont :

- **Définition de la méthode de définition de DSL :** cette activité consiste, dans la première itération du processus, à proposer une première version de la méthode et, dans les itérations subséquentes, à améliorer et à raffiner la méthode proposée ;
- **Expérimentation et évaluation :** préparer une étude de cas pour vérifier l'efficacité de la méthode proposée et évaluer son applicabilité ;
- **Satisfaction :** c'est une activité de décision. Si l'utilisation de la méthode donne satisfaction, on passe à l'étape de transposition de la méthode au métamodèle ISO/IEC 24744; sinon on commence une nouvelle itération d'amélioration et de mise à niveau. Le niveau de satisfaction est évalué par rapport aux critères d'évaluation qui seront énoncés dans le chapitre 7. Les critères retenus pour l'évaluation sont essentiellement ceux reliés aux aspects fonctionnalités et utilisation, soit la convenance, la satisfaction, la facilité

d'exploitation et la facilité de compréhension. Toutefois, à cause de l'absence de méthodes de mesure permettant de mesurer ces critères de façon objective, nous nous somme basés surtout sur notre appréciation qualitative de ces critères.

4.3.3 Normalisation de la méthode

Cette activité sert à redéfinir la méthode conformément au métamodèle défini par la norme ISO/IEC 24744 (***S**oftware **E**ngineering **M**etamodel for **D**evelopment **M**ethodologies* – SEMDM) (voir chapitre 3). Nous pensons que le développement de notre méthode selon ce métamodèle permettra d'assurer un certain niveau d'intégrité et de cohérence en termes de concepts et de terminologie utilisés pour la définition de la méthode.

4.4 Démarche détaillée

D'un point de vue détaillé, la méthodologie de recherche comporte trois phases principales : une phase informationnelle qui sert à dresser l'état de l'art en matière de développement des DSL et à en faire une synthèse, une phase propositionnelle où l'on propose une méthode pour la définition des DSL et une phase d'évaluation qui sert à évaluer la méthode proposée et à vérifier son applicabilité. Les sections suivantes décrivent en détail ces phases et leurs activités sous-jacentes (voir Figure 4.2).

4.4.1 Phase informationnelle

Comme évoqué précédemment, le but de cette phase est de passer en revue la littérature traitant le développement des DSL afin d'identifier les processus, les méthodes, les techniques et les bonnes pratiques utilisés dans ce domaine.

- **Revue de la littérature :** Passer en revue la littérature traitant le développement des DSL et les espaces technologiques qui lui sont reliés (e.g. ingénierie des langages, modélisation spécifique au domaine, développement dirigé par les modèles, ligne de produits logiciels, analyse de domaine, etc.). La revue de littérature couvre également les

expériences industrielles, les outils de développement de DSL, et les travaux de recherche dans ce domaine. Le but est de dresser un état de l'art des pratiques suivies dans le développement des DSL et d'identifier les facteurs de succès qui serviront à l'évaluation de la méthode ;

- **Analyse et synthèse :** organiser les pratiques utilisées pour la définition des DSL en phases, activités et artéfacts.

Cette phase informationnelle s'étale sur tout le cycle de vie du travail de recherche. Son but est d'alimenter en permanence le travail de recherche en lui apportant l'information et la connaissance nécessaires.

4.4.2 Phase propositionnelle

Le but de cette phase est d'élaborer la première version de la méthode qu'on propose comme solution à la problématique de la définition des DSL. D'après la revue de la littérature, il a été constaté que le processus de développement des DSL était proche, dans ses phases et ses activités, de celui du développement du logiciel. Ainsi, nous avons décidé de commencer l'élaboration de notre méthode en nous basant sur un processus de développement logiciel, soit le processus unifié de Rational (*Rational Unified Process* - *RUP*). La démarche consiste donc, essentiellement, à adapter le processus RUP à la définition des DSL en y incorporant les éléments ressortis de l'analyse de la littérature. Cette démarche nous permet de tirer profit de la maturité du processus logiciel et ainsi garantir un certain niveau de cohérence de la méthode et, en même temps, de vérifier s'il y a des activités du processus logiciel qui sont pertinentes au développement des DSL et qui, jusqu'alors, n'étaient pas introduites au cycle du développement des DSL.

Cette phase est donc composée de deux activités principales :

- **Adaptation du processus RUP :** adapter le processus RUP au développement des DSL. Le processus d'adaptation consiste essentiellement à sélectionner, en tenant compte des contraintes et des spécificités des DSL, les phases, les activités et les artéfacts qui sont pertinents à la définition des DSL. La sélection tient aussi compte du fait qu'on vise à

élaborer une méthode agile focalisant surtout sur les activités essentielles à la définition des DSL ;

- **Compilation et organisation des activités :** une fois que le cadre et la structure générale de la méthode sont établis, le but de cette activité est d'arranger les activités, les techniques et les artefacts ressortis de la revue de littérature dans le processus.

À ce stade on a une première version « non-normalisée » de la méthode (version brute non conforme à la norme ISO/IEC 24744). C'est cette version qui va former le cadre théorique du processus de recherche-action. Les modifications à apporter sur la méthode, en réponse à son évaluation (voir section 4.4.3), se feront toujours sur la version non normalisée.

4.4.3 Phase d'expérimentation et d'évaluation

Une fois le développement de la première version de la méthode complété, nous entamons un processus de recherche-action, un processus itératif dont le critère d'arrêt est la satisfaction de l'efficacité et de l'applicabilité de la méthode. Ce processus consiste essentiellement en : **1)** revue et analyse de la méthode, **2)** expérimentation, **3)** évaluation et **4)** mise à niveau et amélioration.

- **Revue et analyse :** passer en revue la méthode proposée, analyser ses points forts et ses faiblesses ;
- **Expérimentation :** examiner l'efficacité et l'applicabilité de la méthode à travers des exemples de test. Le but est de s'assurer de l'applicabilité de la méthode pour définir les différents types de DSL (DSL de programmation, DSL de modélisation, etc.) ;
- **Évaluation** : évaluer les résultats obtenus lors de l'expérimentation de la méthode, en se basant sur le facteurs de succès identifiés dans la phase informationnelle, afin de justifier son efficacité et arrêter son processus de développement ou, le cas échéant, identifier ses problèmes d'applicabilité et recommencer une nouvelle itération ;
- **Mise à niveau et amélioration :** si l'évaluation de la méthode ne donne pas satisfaction, on procède à une activité de mise à niveau de la méthode. Le but est de produire une

version améliorée remédiant aux défauts, insuffisances et imperfections détectés lors de la phase d'évaluation.

4.4.4 Finalisation

Normalisation de la méthode : si l'évaluation de la méthode donne satisfaction, on procède à la normalisation de la méthode afin de produire la version finale. Par normalisation, nous désignons le processus permettant la définition de la méthode selon le métamodèle défini par la norme ISO/IEC 24744.

Clôture : clôturer le travail de recherche en résumant ses points forts et ses faiblesses et en annonçant les travaux futurs qui peuvent le compléter ainsi que les autres opportunités de recherche dans ce champ disciplinaire.

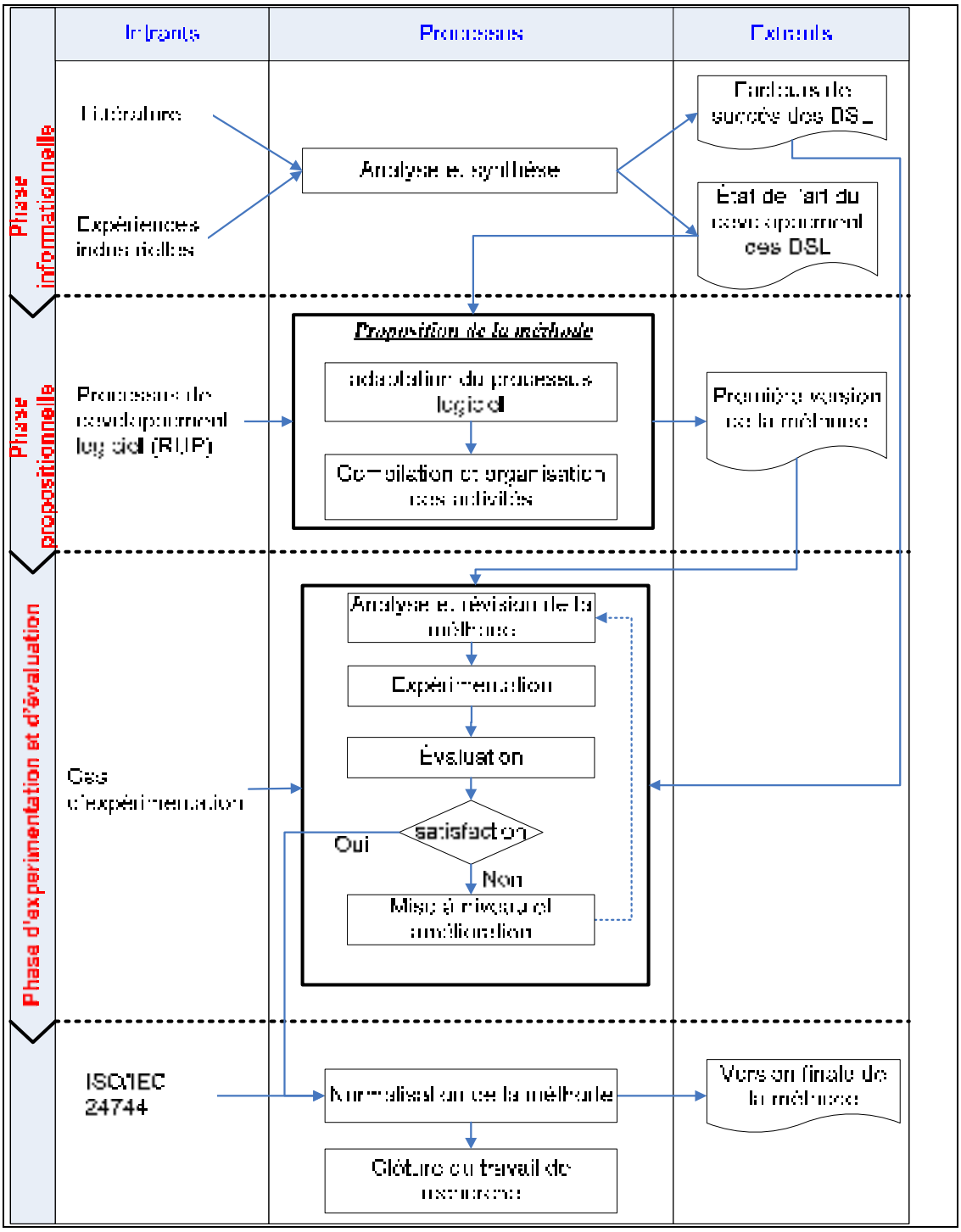

Figure 4.2 Démarche détaillée du travail de recherche.

CHAPITRE 5

MÉTHODE DE DÉFINITION DES DSL : DESCRIPTION GÉNÉRALE

5.1 Introduction

Ce chapitre présente la méthode de définition des DSL qu'on propose en sa version brute (non normalisée). Le résultat de la compilation des pratiques de développement des DSL, tirées de la littérature, est organisé et structuré pour former un processus cohérent décrivant les phases principales, les activités et les artefacts à produire dans un cycle de définition de DSL.

La structuration de la méthode est grandement inspirée du processus de développement logiciel RUP (*Rational Unified Process*). Outre son utilisation comme framework de structuration, RUP est utilisé également dans ce travail pour dresser un parallèle entre les pratiques du développement logiciel et celles utilisées actuellement dans le développement des DSL. Par cette démarche, nous souhaitons utiliser les pratiques du développement logiciel comme une référence afin d'examiner s'il n'y aurait pas des pratiques qui jusqu'alors, n'étaient pas introduites dans le développement des DSL.

À l'instar de l'architecture du processus RUP, la méthode proposée adopte une architecture à deux dimensions : une dimension dynamique exprimée en termes de phases, d'itérations et de jalons et une dimension statique décrite en termes d'unités de travail, d'artefacts et de rôles (voir Figure 5.1).

Comme illustré à la Figure 5.1, la méthode proposée consiste en trois phases principales : **1)** phase d'initialisation, **2)** phase d'élaboration et **3)** phase de construction. Le Tableau 5.1 montre la correspondance entre les phases RUP et les phases de la méthode de définition des DSL (voir ANNEXE I pour l'ensemble des correspondances).

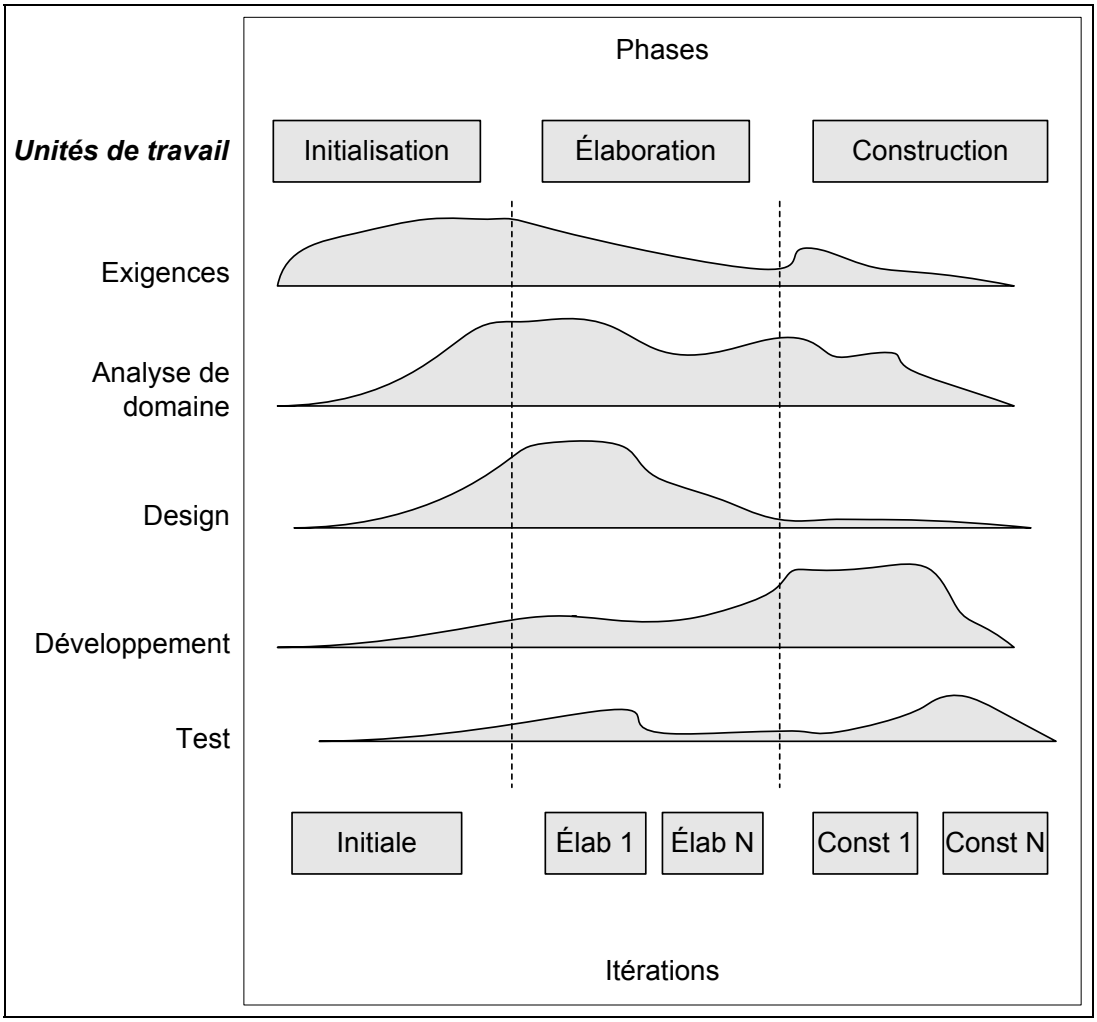

Figure 5.1 Vue d'ensemble de d'architecture de la méthode.

Les sections suivantes présentent une description détaillée incluant le but, les objectifs, les critères d'évaluation, les activités à exécuter et les artefacts à produire pour chacune des phases.

Tableau 5.1 Mappage entre les phases RUP et les phases du développement d'un DSL

Phases RUP	Phases DSL
Initialisation	Initialisation
Élaboration	Élaboration
Construction	Construction
Transition	

5.2 Phase d'initialisation

5.2.1 But

Le but de la phase d'initialisation est d'évaluer la valeur et la faisabilité du DSL et de parvenir à un accord entre toutes les parties prenantes sur les objectifs à atteindre à la fin du développement du DSL. Cette phase est importante, surtout lorsqu'on envisage le développement d'un nouveau DSL. Au départ, il est souvent difficile de prévoir l'utilité du DSL ou de justifier que son développement pourrait être rentable (Mernik, Heering et Sloane, 2005). Néanmoins, ces questions doivent être appréhendées avant de se lancer dans le développement. Une analyse préliminaire peut être utilisée afin de d'aider à appréhender certains des dits questionnements.

5.2.2 Objectifs principaux

Les objectifs principaux de la phase d'initialisation comprennent :

- Définition du domaine du DSL (portée) : définir une vision opérationnelle pour le DSL ; c'est-à-dire définir ce que le DSL doit (ou ne doit pas) faire ;
- Détermination des caractéristiques principales (*Features*) du DSL et de ses critères d'acceptation : déterminer les fonctionnalités que le DSL doit offrir à ses utilisateurs et définir les critères permettant de valider sa conformité aux exigences spécifiées et de s'assurer qu'il répond bien aux attentes ;

- Estimation du coût global et du temps nécessaire au développent du DSL : évaluer le coût du projet afin de déterminer si ça vaut la peine de s'y investir ;
- Estimation des risques potentiels : une analyse préalable des risques auxquels le projet peut être exposé est essentielle afin de formuler les stratégies les plus appropriées pour y faire face ;
- Préparation d'un environnement pour soutenir le développement du DSL : préparer les ressources et les outils nécessaires au développement du DSL.

5.2.3 Jalon

Le jalon de la phase d'initialisation consiste en les objectifs du cycle de vie du projet de développement du DSL. À ce stade, on examine ces objectifs et on décide soit de poursuivre le projet ou de l'annuler.

5.2.4 Critères d'évaluation

- Consentement des parties prenantes sur la portée du DSL ;
- Consentement des parties prenantes sur l'estimation du coûts/temps du développement du DSL ;
- Consentement sur les exigences que doit satisfaire le DSL : s'assurer que les parties prenantes affectées par le DSL ont une compréhension commune des exigences auxquelles le DSL doit répondre ;
- Les risques les plus critiques ont été identifiés et une stratégie de mitigation a été prévue pour chacun.

5.2.5 Activités

La phase d'initialisation comporte des activités de base qui sont recommandées pour tous les projets et d'autres qui sont optionnelles.

Activités de base

- Définir le domaine (*Scoping*) : Définir le contexte, les exigences principales et les contraintes du DSL. Le résultat de cette activité doit permettre d'identifier les critères d'acceptation du langage ;
- Définir la vision du DSL : identifier les problèmes à résoudre par le DSL, ses parties prenantes (e.g. utilisateurs potentiels, évaluateurs et validateurs, clients si le DSL est destiné à la commercialisation, etc.) et ses principales caractéristiques.

Activités optionnelles

- Préparer le plan d'affaire : évaluer le coût et la rentabilité du DSL ainsi que les alternatives pour gérer les risques associés ;
- Préparer l'environnement de soutien : mettre en place les ressources, les compétences et les outils nécessaires à la réussite du projet.

5.2.6 Artefacts

5.2.6.1 Artefacts principaux

Le Tableau 5.2 présente une description sommaire des principaux artefacts qu'on recommande de produire pendant la phase d'initialisation.

Tableau 5.2 Artéfacts de base de la phase d'initialisation

Nom	Objectif	Responsable
Vision	Documenter les exigences, les caractéristiques et les contraintes principales du DSL	Analyste/Expert de domaine
Glossaire	Définir les termes importants utilisés dans le domaine	Expert de domaine
Exigences	Spécifier les besoins et les exigences que le DSL doit satisfaire	Analyste/Expert de domaine
Modèle du domaine	Définir, généralement en se basant sur le document des terminologies, les concepts clefs du DSL et les relations qui les relient.	Analyste/Expert de domaine

5.2.6.2 Artefacts optionnels

Le Tableau 5.3 présente une description sommaire des artefacts optionnels qu'on peut produire pendant la phase d'initialisation. La décision des artefacts à produire dépend, entre autres, du type de DSL, de sa taille et de son niveau de complexité.

Tableau 5.3 Artéfacts optionnels de la phase d'initialisation

Nom	**Objectif**	**Responsable**
Liste des risques	Identifier la liste des éventuels risques	Analyste
Cas d'affaire	Élaborer un cas d'affaire	Analyste

5.2.7 Recommandations

- Considérer l'adoption de DSL déjà existants. C'est moins coûteux et cela nécessite beaucoup moins d'expérience que d'en élaborer un nouveau, quoiqu'il soit difficile de trouver des DSL bien documentés ;
- Adopter une vision à moyen et à long terme et éviter les considérations à court terme ;
- Prévoir une équipe disposant d'une connaissance du domaine et d'une expertise dans le développement de langages afin de prendre les meilleures décisions.

5.3 Phase d'élaboration

5.3.1 But

Établir une architecture répondant aux exigences les plus importantes du DSL.

5.3.2 Objectifs

- S'assurer que les exigences sont suffisamment stables pour prévoir le coût et le temps nécessaires au développement du DSL ;
- Établir une architecture et démontrer que celle-ci répondra aux exigences ;
- Établir une procédure de test permettant de valider le DSL ;
- Mettre en place un environnement de support pour soutenir le développement du DSL.

5.3.3 Activités de base

- Identifier les abstractions du domaine ;
- Définir l'architecture du DSL ;
- Préparer les procédures de test du DSL ;
- Raffiner, suite à l'acquisition de nouvelles informations et/ou à une meilleure compréhension des exigences, la vision du DSL.

5.3.4 Activités facultatives

- Créer un plan détaillé d'itérations de la phase de construction ;
- Mettre en place un environnement favorable au développement du DSL. Ceci comprend le processus de développement, les outils d'automatisation, les experts du domaine et toute ressource capable d'apporter une valeur ajoutée à la construction du DSL.

5.3.5 Jalon

Le jalon de la phase d'élaboration est l'architecture du DSL. Le projet peut être interrompu ou repensé si on ne parvient pas à avoir une architecture stable.

5.3.6 Critères d'évaluation

- La vision, les exigences et l'architecture sont stables ;
- Les méthodes de test et d'évaluation sont fiables ;
- Les plans d'itération de la phase de construction sont à un niveau de détail permettant le lancement du développement du DSL.

5.3.7 Artefacts

5.3.7.1 Artefacts principaux

Le Tableau 5.4 présente une description sommaire des principaux artefacts qu'on recommande de produire pendant la phase d'élaboration.

Tableau 5.4 Artéfacts de base de la phase d'élaboration

Nom	**Objectif**	**Responsable**
Concepts du domaine	Identifier les abstractions du domaine et comment elles s'interagissent	Analyste/Expert de domaine
Document d'architecture	Décrire l'architecture du métamodèle du DSL, soit la structure et l'organisation de ses éléments	Concepteur
Vision	Raffiner, suite à l'acquisition de nouvelles informations et/ou à une meilleure compréhension des exigences, la vision du DSL	Analyste/Expert de domaine

5.3.7.2 Artefacts optionnels

Le Tableau 5.5 énumère les artefacts optionnels qu'on peut produire pendant la phase d'élaboration. La décision des artefacts à produire dépend, entre autres, du type de DSL, de sa taille et de son niveau de complexité.

Tableau 5.5 Artéfacts optionnels de la phase d'élaboration

Nom	**Objectif**	**Responsable**
Liste des risques	Revue et mise à jour	Analyste
Cas d'affaire	Mettre à jour le cas d'affaire si l'examen architectural révèle des soucis qui peuvent mettre en cause la validité du développement du DSL.	Analyste

5.4 Phase de construction

5.4.1 But

Développer une version stable et utilisable du DSL.

5.4.2 Objectifs

- Compléter l'analyse, la conception, le développement et le test de toutes les fonctionnalités requises ;
- Mise en production de nouvelles versions (alpha, bêta, etc.) du DSL.

5.4.3 Activités de base

- Définir la syntaxe abstraite du DSL ;
- Définir sa syntaxe concrète ;
- Définir sa sémantique;
- Faire des tests.

5.4.4 Activités facultatives

- Évaluer les éléments du DSL par rapport aux critères d'acceptation et la vision ;
- Développer un manuel d’utilisation pour les utilisateurs du DSL.

5.4.5 Jalon

Le jalon de la phase de construction est une nouvelle mise à jour (*Release*) du DSL.

5.4.6 Critères d'évaluation

La version livrée est stable et prête à être déployée.

5.4.7 Artefacts

5.4.7.1 Artefacts principaux

Le Tableau 5.6 liste les principaux artefacts qu'on recommande de produire pendant la phase de construction.

Tableau 5.6 Artéfacts de base de la phase de construction

Nom	**Objectif**	**Responsable**
Concepts du domaine	Raffiner et mettre à jour	Analyste/Expert de domaine
Syntaxe abstraite (Métamodèle)	Définir les concepts, les relations et les règles de grammaire caractérisant le DSL.	Développeur
Syntaxe concrète (Notation)	Définir la syntaxe concrète du DSL	Développeur
Sémantique	Décrire la sémantique des éléments du DSL	Développeur

5.4.7.2 Artefacts optionnels

Le Tableau 5.7 liste les artefacts optionnels qu'on peut produire pendant la phase de construction. La décision des artefacts à produire dépend, entre autres, du type de DSL, de sa taille et de son niveau de complexité.

Tableau 5.7 Artéfacts optionnels de la phase de construction

Nom	**Objectif**	**Responsable**
Guide d'utilisation	Aider les utilisateurs à bien utiliser le DSL	Développeur

CHAPITRE 6

MÉTHODE DE DÉFINITION DES DSL : NORMALISATION

6.1 Introduction

Ce chapitre présente la méthode de définition des DSL décrite dans le chapitre précédent en utilisant le métamodèle de la norme ISO/IEC 24744:2007 (voir chapitre 3). Le développement de la méthode avec un tel métamodèle permet à celle-ci de bénéficier d'un cadre de définition de méthodes intégral et cohérent. Le métamodèle représente les éléments principaux à la définition d'une méthode, notamment, le processus à suivre, les artefacts à produire, les ressources et les outils impliqués, etc. De plus, le métamodèle a été aussi une occasion pour nous de réviser et d'améliorer les éléments déjà définis dans le chapitre 5.

Dans ce chapitre, nous allons porter le chapeau d'un ingénieur de méthodes dont le rôle est de générer une méthode à partir du métamodèle ISO/IEC 24744:2007. Cette génération consiste essentiellement à décrire la structure et la sémantique des éléments composant la méthode. Ces éléments relèvent essentiellement du domaine-méthode du métamodèle (voir section 3.1).

La génération de la méthode est effectuée en instanciant les classes définies par le métamodèle. L'instanciation de ces classes prend l'une des trois formes suivantes :

- Instanciation conventionnelle pour les classes qui ne font pas partie d'une relation de type powertype;
- Instanciation du patron powertype pour les classes impliquées dans des relations de type powertype (voir section 3.1);
- Extension des types énumérés définis par le métamodèle.

La suite de ce chapitre présente la génération des aspects suivants de la méthode :

- Processus : exprime ce qu'on doit faire, quand et comment le faire ;
- Produits : définit les artefacts à produire et à utiliser dans les projets adoptant la méthode ;
- Producteurs : détermine les groupes et les personnes responsables de l'exécution des activités du processus et de la production des artefacts.

6.2 Processus

Cette section définit l'aspect processus de la méthode, et ce en instanciant les powertypes *Stage/*Kind* et *WorkUnit/*Kind*.

6.2.1 Stage/*Kind

Le powertype *Stage/*Kind* représente un bloc de temps (*managed time frame*) dans un projet de définition de DSL en spécifiant la structure temporelle globale de la méthode. Ce powertype est spécialisé par les powertypes : *TimeCycle/*Kind*, *Phase/*Kind*, *Build/*Kind* et *Milestone/*Kind*.

TimeCycle/*Kind

L'instanciation du powertype *TimeCycle/*Kind* permet de représenter le cycle de développement au bout duquel le DSL sera livré (voir Tableau 6.1 et Figure 6.1)

Tableau 6.1 Instance du powertype *TimeCycle/*Kind*

Nom	Sémantique
Cycle-DEV	Cycle de temps du processus de développement du DSL

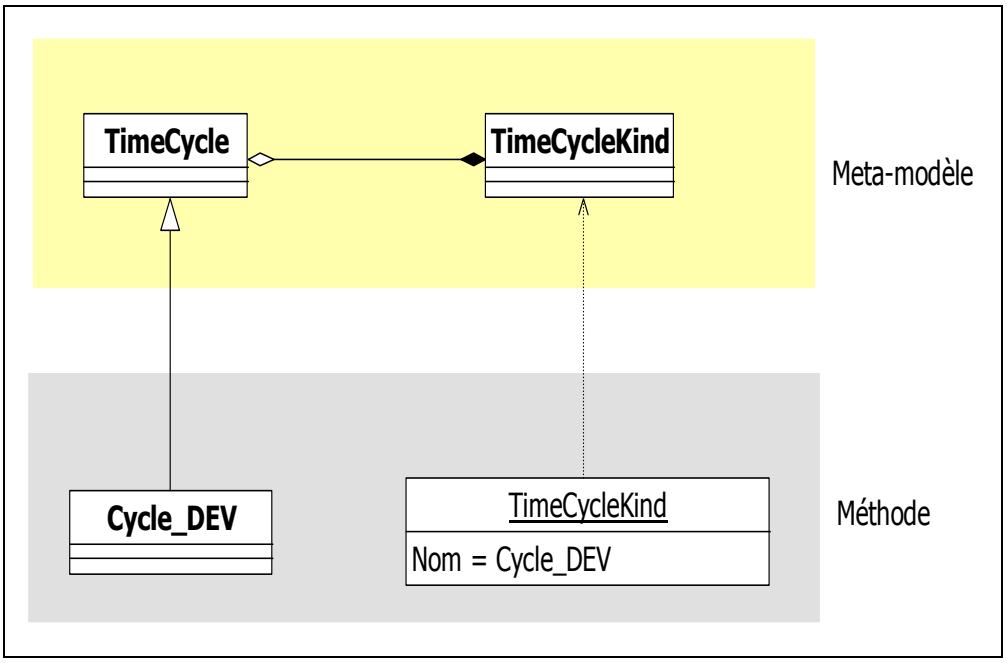

Figure 6.1 Instanciation du powertype *TimeCycle/*Kind.*

Phase/*Kind

Les phases à suivre durant le cycle de développement Cycle-DEV sont définies en instanciant le powertype *Phase/*Kind*. Les trois phases définies par la méthode sont : la phase d'initialisation, la phase d'élaboration et la phase de construction (voir Figure 6.2).

Le Tableau 6.2 énumère les trois phases qui composent le cycle de développement d'un DSL.

Tableau 6.2 Instances du powertype *Phase/*Kind*

Nom (Code)	**Contexte**	**Sémantique**
Phase d'initialisation (PHSE-01)	Cycle-DEV	Phase dont le but est d'évaluer la valeur et la faisabilité du DSL et de parvenir à un accord entre toutes les parties prenantes sur les objectifs à atteindre à la fin du développement du DSL
Phase d'élaboration (PHSE-02)	Cycle-DEV	Phase dont le but est de préparer le terrain pour la phase de construction. Elle consiste à établir une connaissance du domaine, à définir une vision du DSL et à concevoir une architecture pour le métamodèle du DSL
Phase de construction (PHSE-03)	Cycle-DEV	Phase de réalisation du DSL. Durant cette phase sont définies la syntaxe abstraite, la syntaxe concrète et la sémantique du DSL

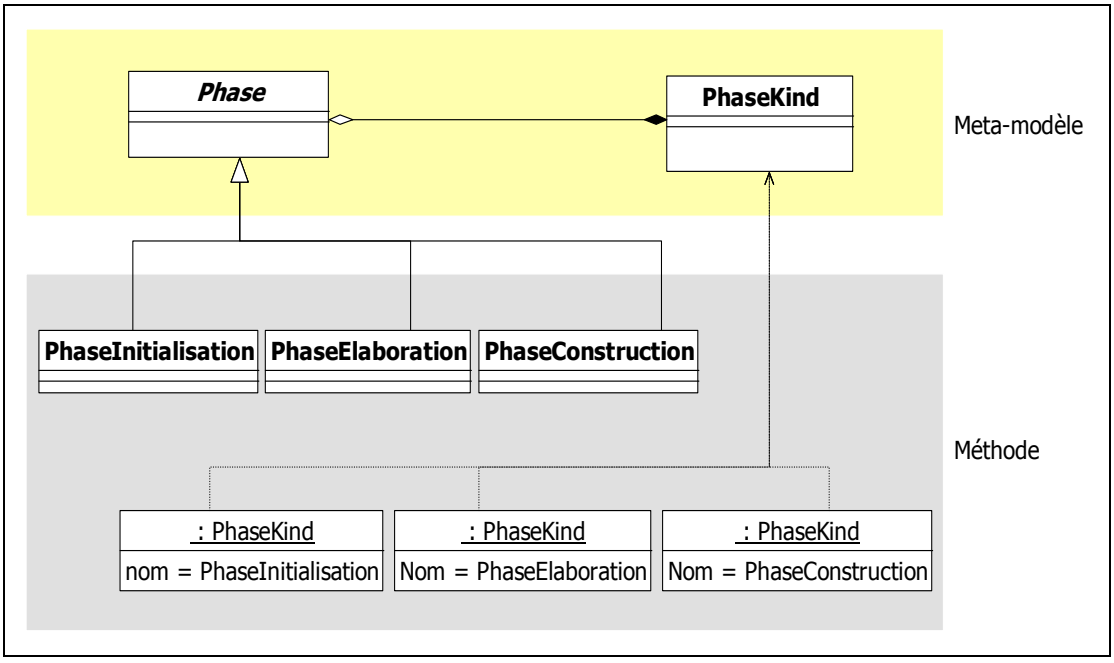

Figure 6.2 Instanciation du powertype Phase/*Kind.

Build/*Kind

Comme la construction du DSL peut se faire d'une façon incrémentale, alors la définition d'un concept qui représente les incréments est nécessaire. Dans le SEMDM, ce concept est défini par le powertype *Build/*Kind*. Ce dernier représente un intervalle de temps (*StageWithDuration*) dont l'objectif principal est la livraison d'une version incrémentée d'une série d'unités de travail déjà existantes (voir Figure 6.3).

Le Tableau 6.3 liste les trois types de constructions utilisés pour gérer la livraison des incréments des unités de travail.

Tableau 6.3 Instances du powertype *Build/*Kind*

Nom (Code)	**Contexte**	**Sémantique**
Build-Initialisation (BUILD-01)	PHSE-01	Étape dont le but est de livrer une évaluation de la valeur et de la faisabilité du DSL ainsi qu'une spécification des objectifs à atteindre à la fin de son développement
Build-Élaboration (BUILD-02)	PHSE-02	Étape dont le but est de livrer une architecture stable pour le DSL et une définition des abstractions principales du domaine
Build-Construction (BUILD-03)	PHSE-03	Étape dont le but est de livrer une version utilisable du DSL

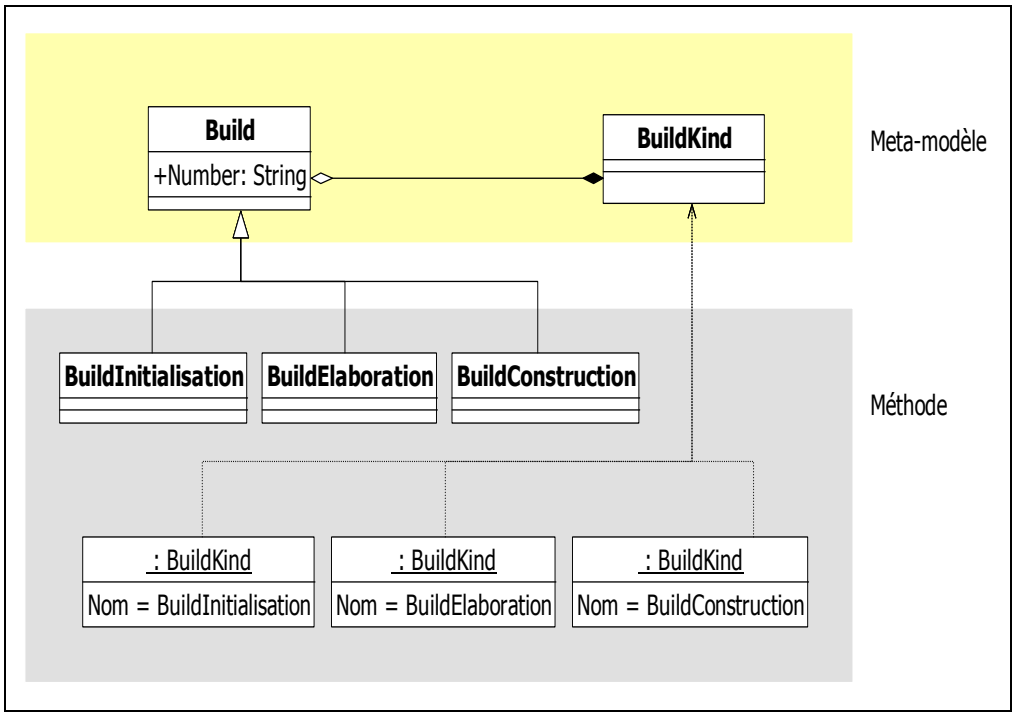

Figure 6.3 Instanciation du powertype Build/*Kind.

Milestone/*Kind

Le powertype *Milestone/*Kind* est utilisé pour marquer des moments ou des événements importants dans le cycle de développement d'un DSL. Les instances de ce powertype sont particulièrement importantes pour identifier les moments de transition entres les différentes phases (voir Figure 6.4).

Le Tableau 6.4 liste les jalons (*Milestone*) principaux dans un projet de définition de DSL.

Tableau 6.4 Instances du powertype *Milestone/*Kind*

Nom	**Description**
Objectifs-DSL (MS-01)	Marque la fin de l'évaluation de la valeur et de la faisabilité du DSL
Architecture-DSL (MS-02)	Marque la fin de l'établissement d'une architecture pour le DSL
Release-DSL (MS-03)	Marque l'achèvement du développement d'une version répondant aux exigences du client. À ce stade, on décide si les objectifs ont été atteints, et si on doit commencer un autre cycle de développement

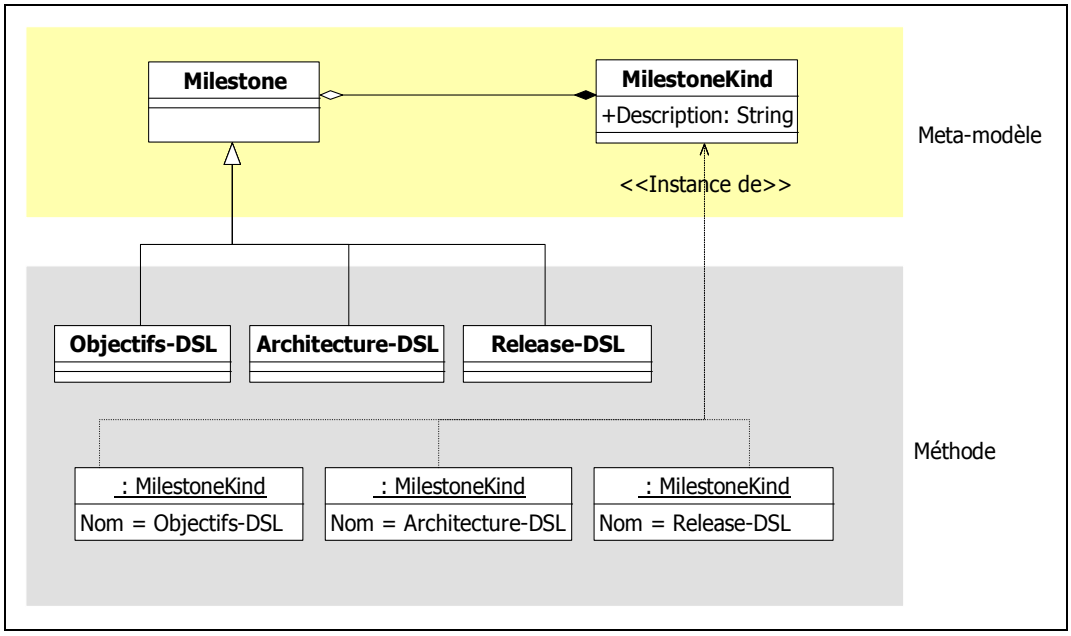

Figure 6.4 Instanciation du powertype *Milestone/*Kind.*

6.2.2 WorkUnit/*Kind

Les éléments créés jusqu'ici représentent la structure temporelle de la méthodologie. Dans cette section, nous présentons les éléments représentant les activités à effectuer durant ces intervalles de temps.

Une unité de travail est caractérisée par son but et son niveau de capacité minimale. C'est cette dernière caractéristique qui détermine le niveau minimal auquel il devient judicieux d'exécuter l'unité de travail.

Process/*Kind

Le powertype *Process/*Kind* représente une activité grande en taille, qui s'exécute à l'intérieur d'un domaine d'expertise particulier (voir Figure 6.5). Le Tableau 6.5 énumère les types d'activités définies par la méthode afin de couvrir les domaines d'expertise nécessaires à la définition des DSL.

Tableau 6.5 Instances du powertype *Process/*Kind*

Nom	**Contexte**	**But**	**NMC[10]**
Exigences (WU-01)	PHSE-01	Formuler les exigences du DSL	1
Analyse de domaine[11] (WU-02)	PHSE-01	Collecter et organiser la connaissance du domaine. À la fin de cette tâche, on devrait avoir au moins ces éléments définis : portée du domaine, terminologie, liste de concepts candidats, points communs et variations	1
Design (WU-03)	PHSE-02	Transformer les exigences en une architecture qui définit les éléments architecturalement pertinents au développement du DSL	1
Développement (WU-04)	PHSE-03	Développer les éléments du DSL	1
Test (WU-06)	PHSE-03	Tester le DSL et vérifier que toutes les exigences ont été satisfaites.	2

[10] Niveau minimum de capacité auquel un type d'unité de travail peut être effectué (voir *Minimum Capability Level* dans (ISO, 2007b)).
[11] Les ingénieurs de domaine peuvent passer outre cette activité s'ils connaissent suffisamment le domaine d'application des DSL.

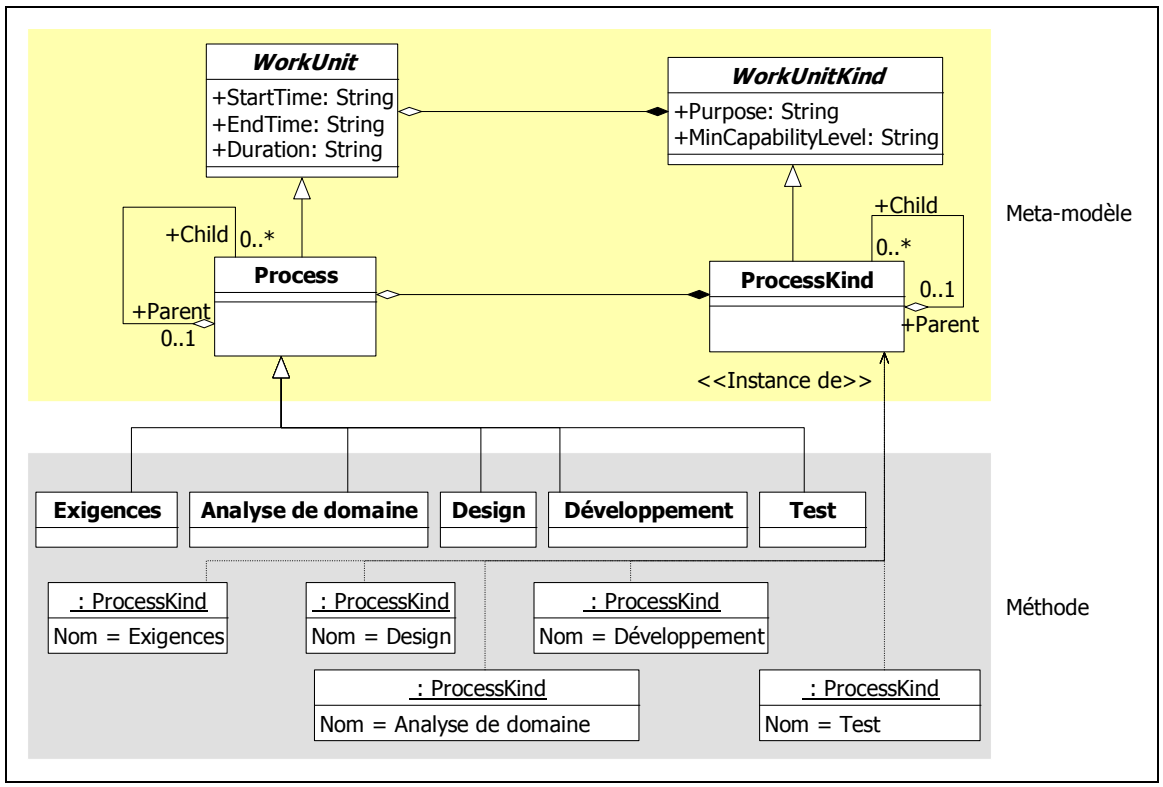

Figure 6.5 Instanciation du powertype *Process/*Kind.*

Task/*Kind

Le powertype *Task/*Kind* représente des petites unités de travail qui servent à implémenter le *Process/*Kind*. La relation d'agrégation entre les unités de travail (*WorkUnit/*Kind*) et les tâches (*Task/*Kind*) permettent de définir récursivement des unités de travail jusqu'à atteinte du niveau de détail désiré (voir Figure 6.6 et Figure 6.7).

Le Tableau 6.6 énumère les *TaskKind* définies par la méthode afin d'implémenter les activités du *ProcessKind*.

Tableau 6.6 Instances du powertype *Task/*Kind*

Nom (Code)	**Contexte**	**But**	**NMC**
Comprendre les besoins (TSK-01)	WU-01	Discuter et comprendre les besoins des utilisateurs	1
Développer la vision (TSK-02)	WU-01	Définir les limites du DSL, décrire ses caractéristiques principales et identifier ses parties prenantes.	1
Préparer l'environnement de soutien (TSK-03)	WU-02	Préparer et mettre en place un environnement de soutien pour assurer le succès du DSL.	1
Définir la portée du domaine (TSK-04)	WU-02	Déterminer la portée du domaine	1
Définir la terminologie (TSK-05)	WU-02	Définir le vocabulaire utilisé dans le domaine en identifiant les termes et les expressions les plus communément utilisés par les experts du domaine.	1
Définir les abstractions (TSK-06)	WU-02	Définir les concepts du domaine	1
Définir le système de production (TSK-07)	WU-02	Définir le système de production à adopter pour le développement de la famille (i.e. architecture de la ligne de produits logiciels).	1
Établir une architecture (TSK-08)	WU-03	Définir une architecture définissant l'ensemble des éléments architecturalement pertinents et organiser ces derniers en fonction de leurs domaines conceptuels	1
Définir la syntaxe abstraite (TSK-09)	WU-04	Définir les concepts du DSL, leurs relations et les règles de grammaire qui déterminent comment ils peuvent être combinés pour créer des modèles valides.	1
Définir la syntaxe concrète (TSK-10)	WU-04	Définir la notation à utiliser dans les modèles du DSL	1
Définir la sémantique (TSK-11)	WU-04	Définir la sémantique du DSL	1

Nom (Code)	Contexte	But	NMC
Identifier les Concepts du DSL (TSK-12)	TSK-09	Identifier les concepts du DSL, leurs caractéristiques et leurs relations.	1
Définir les règles de grammaire (TSK-13)	TSK-09	Définir les règles de grammaire qui déterminent les combinaisons valides des différents concepts du DSL.	1
Préparer les scénarios de test (TSK-14)	WU-05	Préparer des scénarios de test afin de vérifier l'exactitude du DSL et de s'assurer qu'il répond bien aux attentes de ses utilisateurs.	2
Exécuter les scénarios de test (TSK-15)	WU-05	Réaliser les scenarios de test	2

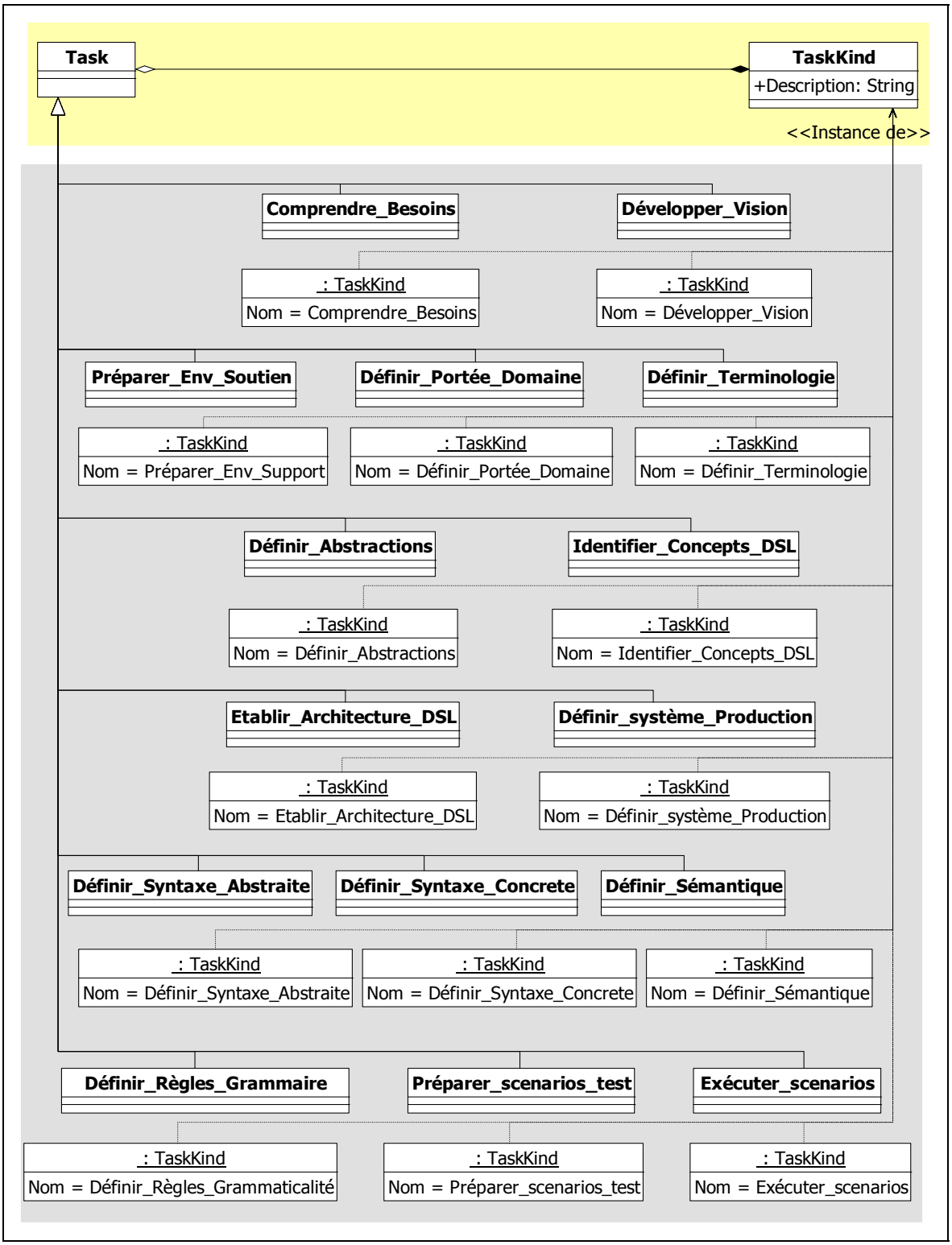

Figure 6.6 Instances du powertype *Task/*Kind.*

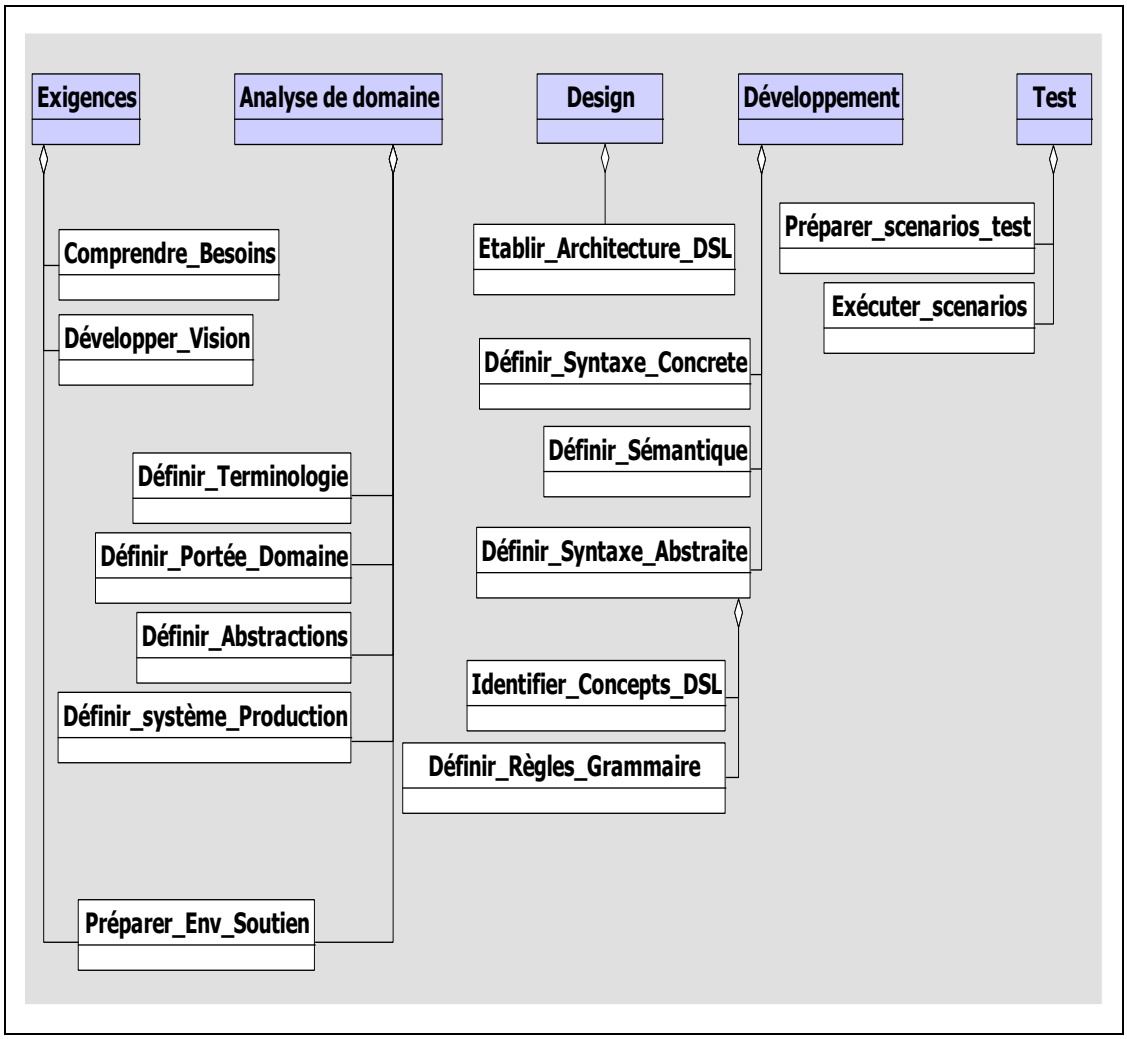

Figure 6.7 Modèle structurel des tâches.

Certaines des classes *TaskKind* définies par la méthode nécessitent des attributs supplémentaires afin de compléter leurs descriptions. Le Tableau 6.7 énumère ces attributs qui viennent s'ajouter à ceux qui sont hérités de la classe *TaskKind* standard.

Tableau 6.7 Attributs supplémentaires aux classes *TaskKind*

Classe	**Nom Attribut**	**Type**	**Sémantique**
Analyse de domaine	méthode	String	Méthode d'analyse de domaine utilisée pour réaliser l'analyse de domaine (ex. FAST, FODA, etc.)
Définir la syntaxe abstraite	langage	String	Langage utilisé pour définir la syntaxe abstraite du DSL (MOF, EMF, UML, etc.)
Définir la syntaxe concrète	langage	String	Langage utilisé pour définir la syntaxe concrète du DSL (BNF, EBNF, métamodélisation, etc.)
Définir les règles de grammaire	langage	String	Langage utilisé pour définir les règles de grammaire du DSL (e.x. OCL, langage naturel, etc.)
Établir l'architecture	Langage	String	Langage utilisé pour modéliser l'architecture

Technique/*Kind

Le powertype *Technique/*Kind* représente une unité de travail, petite en taille, qui se concentre sur la manière d'implémenter une tâche (*Task/*Kind*). Une technique est caractérisée, dans SEMDM, par son but dans le projet (voir Figure 6.8).

Le Tableau 6.8 énumère les techniques qui peuvent être utilisées pour accomplir les tâches définies par la méthode.

Tableau 6.8 Instances du powertype *Technique/*Kind*

Nom	**Utilisé par *Task/*Kind***	**But**	**NMC**
Atelier des exigences (TECH-01)	TSK-01	Comprendre et discuter des attentes des parties prenantes. Le groupe de discussion devrait comprendre des représentants des utilisateurs et des membres de l'équipe chargé du développement du DSL.	1
Dictionnaire du domaine (TECH-02)	TSK-05	Élaborer un dictionnaire expliquant les termes et les expressions utilisés dans le domaine. Ces derniers sont, généralement, identifiés en examinant le jargon utilisé par les experts du domaine, les objets d'affaires, les systèmes existants, etc.	1
Décomposition conceptuelle dirigée par les points de vue (TECH-03)	TSK-12	Identifier les concepts du domaine en se concentrant sur un point de vue à la fois. Cette technique consiste à : - Définir le point de vue qui définit la perspective à partir de laquelle les concepts DSL vont être définis. (e.x. structure physique du produit, interface du système, espace de variation, etc.) - Ensuite, construire une liste de concepts candidats par point de vue défini. (voir (Larman, 2004; Tolvanen, 2006) pour plus de détail). - Raffiner la liste des concepts en établissant des scénarios d'utilisation de ces concepts.	1
Validation par l'exemple (TECH-04)	TSK-13	Cette technique consiste en trois étapes : 1) Préparer des exemples de modèles valides et de modèles invalides afin d'aider à définir les règles de grammaire. 2) Établir une liste des règles de grammaire 3) Examiner les implications des différentes combinaisons de ces règles afin d'identifier celles qui sont possibles et celles qui peuvent causer des conflits.	1

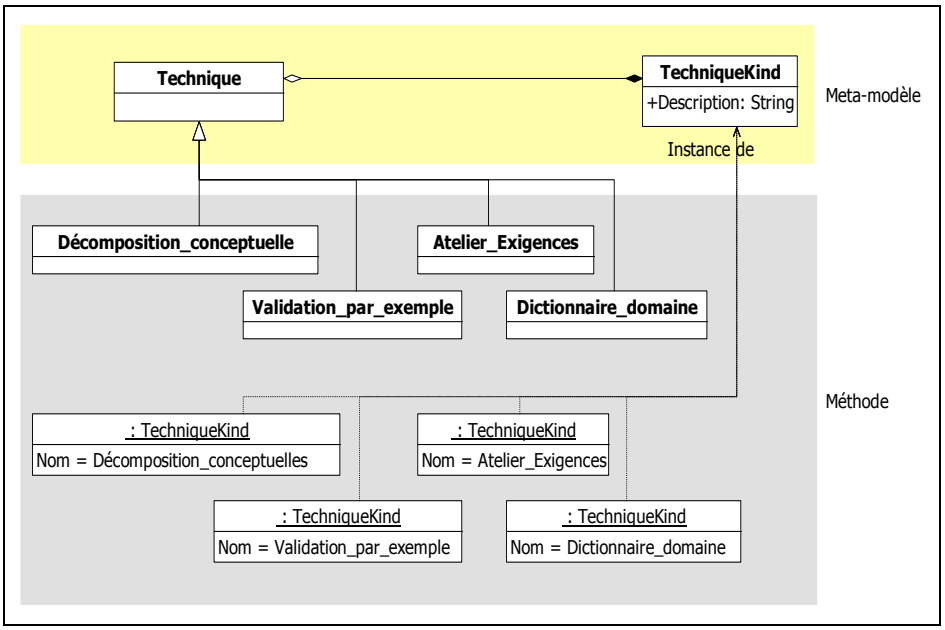

Figure 6.8 Instances du powertype Technique/*Kind.

TaskTechniqueMapping/*Kind

Ce powertype représente le fait qu'une certaine technique est utilisée pour accomplir une certaine tâche (voir Figure 6.9). Le Tableau 6.9 énumère les *TaskTechniqueMappingKind* associant les *TaskKind* au *techniqueKind* qui peuvent être utilisées pour les accomplir.

Tableau 6.9 Instances du powertype *TaskTechniqueMapping /*Kind*

TaskTechniqueMapping/*Kind	Technique/*Kind	Task/*Kind	Recommandation
MAP-01	Atelier des exigences	Comprendre les besoins	Recommandé
MAP-02	Dictionnaire du domaine	Définir la terminologie	Recommandé
MAP-03	Décomposition conceptuelle	Identifier les Concepts du DSL	Recommandé
MAP-04	Validation par l'exemple	Définir les règles de grammaire	Recommandé

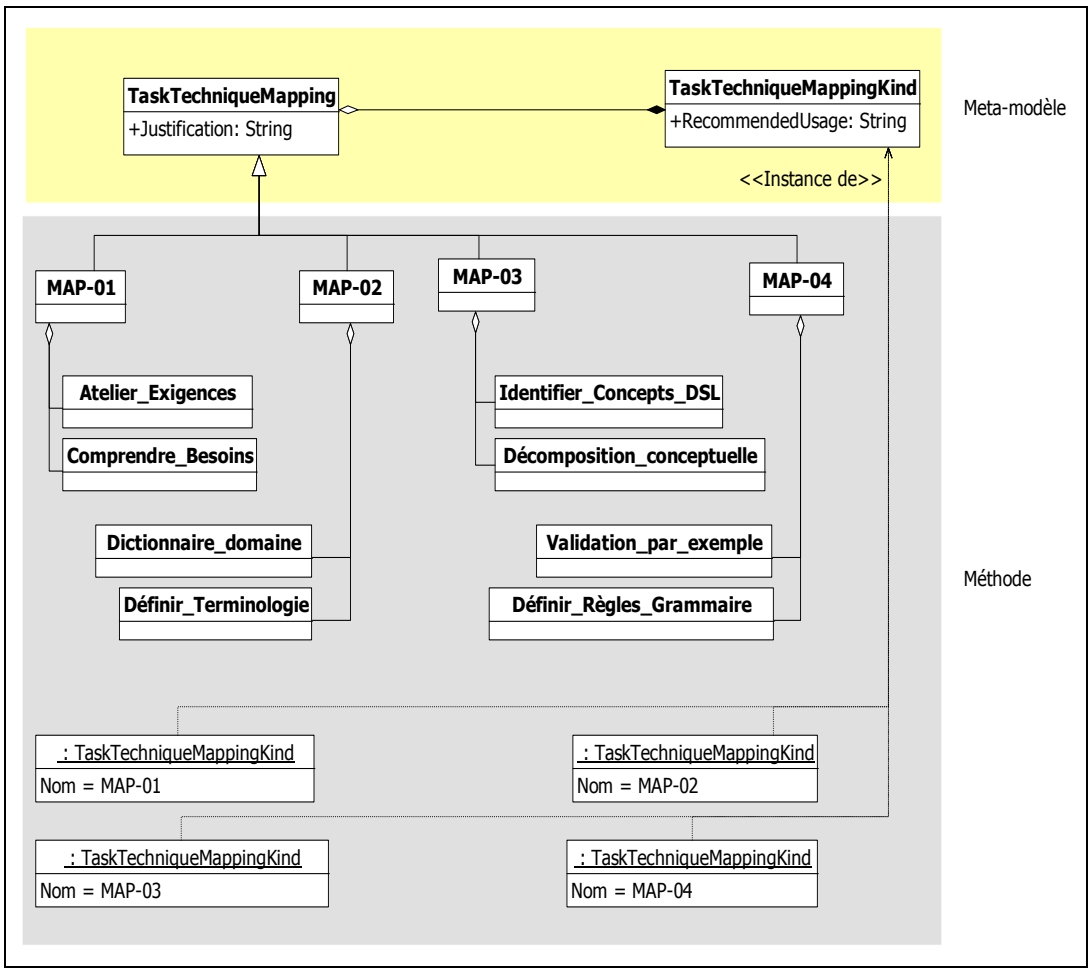

Figure 6.9 Instances du powertype *TaskTechniqueMapping /*Kind.*

6.3 Producteurs

Cette section présente l'aspect producteur qui définit les entités (i.e. personnes, groupes, rôles, etc.) responsables de l'exécution des unités de travail (*WorkUnit/*Kind*) lors du développement d'un DSL. Cet aspect est représenté dans SEMDM par le powertype *Producer/*Kind*. Le lien entre les unités de travail et les producteurs qui en sont responsables est établi par le powertype *WorkPerformance/*Kind*.

La suite de cette section décrit les powertypes spécialisant le powertype *Producer/*Kind* ; soit, *Role/*Kind*, *Team/*Kind* et *Tool/*Kind*.

Role/*Kind

Ce powertype représente un ensemble de responsabilités qu'un producteur peut assumer dans un projet de définition de DSL. Un rôle est joué par un seul producteur et un producteur peut jouer un ou plusieurs rôles (voir Figure 6.10).

Le Tableau 6.10 énumère les types de rôles (*RoleKind*) définies par la méthode.

Tableau 6.10 Instances du powertype *Role/*Kind*

Nom	**Responsabilité**
Analyste de domaine	Un analyste de domaine est responsable de l'Analyse de domaine et de la collecte des exigences
Expert de domaine	Un expert du domaine agit en tant que référence en termes de connaissance du domaine. Il est également responsable de l'évaluation et de la validation du DSL
Concepteur	Un concepteur est responsable de la conception du DSL
Développeur	Un développeur est responsable de la définition du DSL
Testeur	Un testeur est responsable de l'exécution des activités de test du DSL

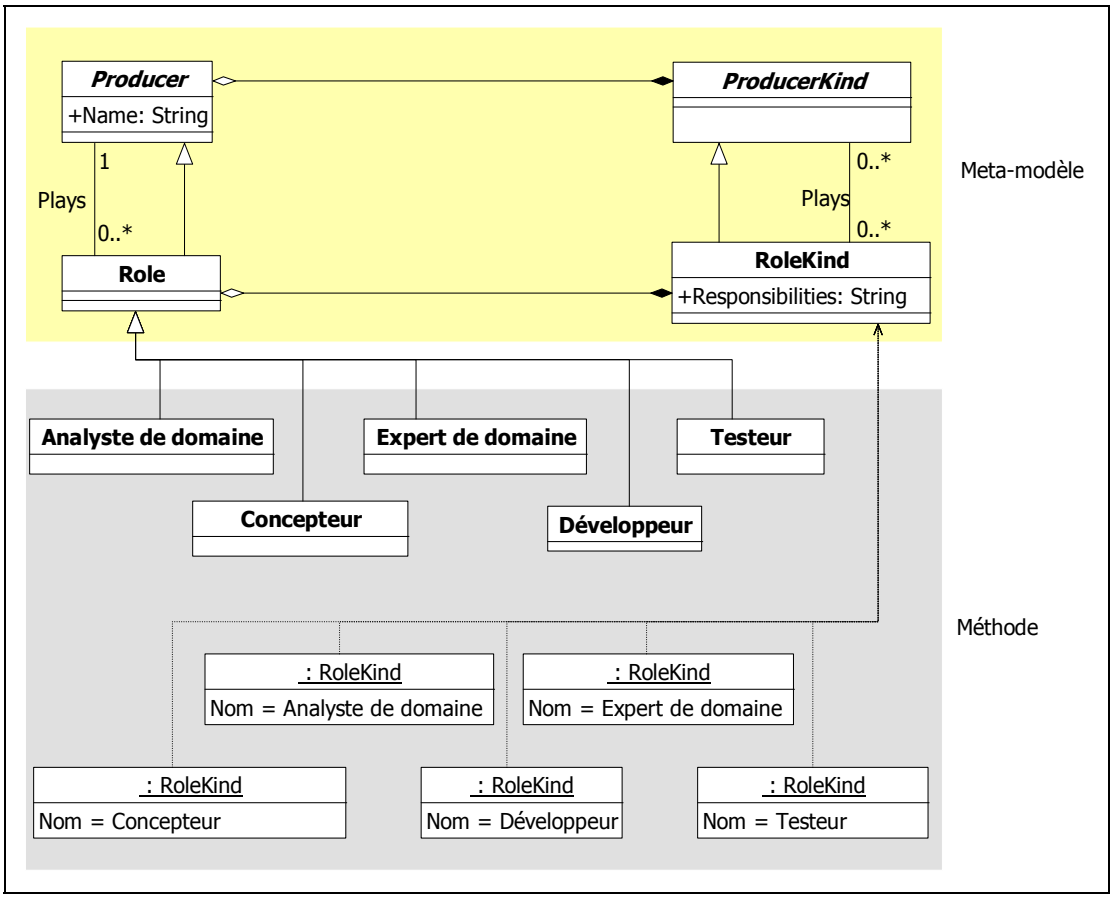

Figure 6.10 Instances du powertype *Role /*Kind.*

Team/*Kind

Le powertype *Team/*Kind* représente un groupe de producteurs qui se concentrent collectivement sur des unités de travail communes (voir Figure 6.11).

Le Tableau 6.11 énumère les types de groupes (*TeamKind*) qu'on peut être amené à former dans un projet de définition de DSL.

Tableau 6.11 Instances du powertype *Team/*Kind*

Team/*Kind	**Composé de Producer/*Kind**	**Description**
Analystes	Analyste de domaine Expert de domaine	Groupe responsable de l'Analyse de domaine et de la collecte des exigences
Développeurs	Concepteur Développeur	Groupe responsable de la conception et de la définition du DSL
Testeurs	Testeur Expert de domaine	Groupe responsable de l'évaluation et du test du DSL.

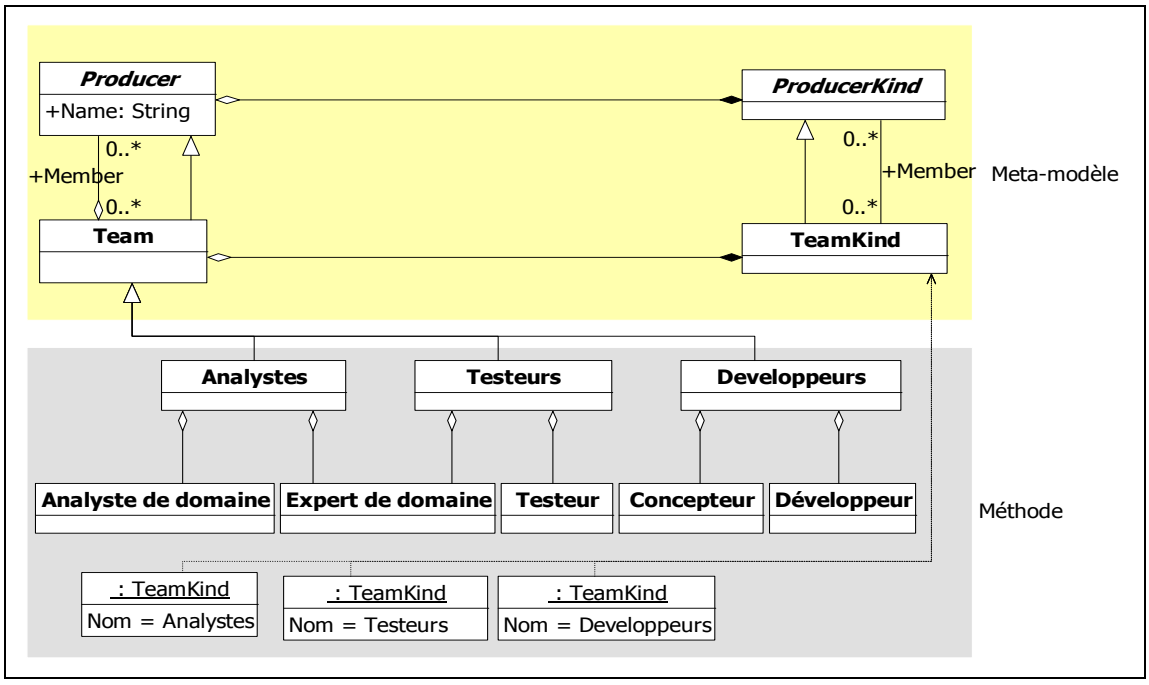

Figure 6.11 Instances du powertype *Team/*Kind.*

Tool/*Kind

Le powertype *Tool/*Kind* représente les outils et les mécanismes aidant les autres types de producteurs à accomplir leurs tâches d'une manière automatisée (voir Figure 6.12).

Le Tableau 6.12 énumère les types d'outils qui peuvent être employés pour automatiser les tâches des analystes, des développeurs et des testeurs.

Tableau 6.12 Instances du powertype *Tool/*Kind*

Nom	**Assigné au WorkUnit/*Kind**	**Responsabilité**	**Exemples**
Meta-outil	WU-04	Environnement intégré regroupant un ensemble d'outils pour la définition des DSL et pour la génération des outils spécialisés qui leur sont dédiés	Eclipse EMF, MetaEdit+, DSL Tools
Meta-Compilateur	WU-04	Outil de création d'analyseurs lexicaux et d'analyseurs de syntaxe	YACC, Lex, ANTLR

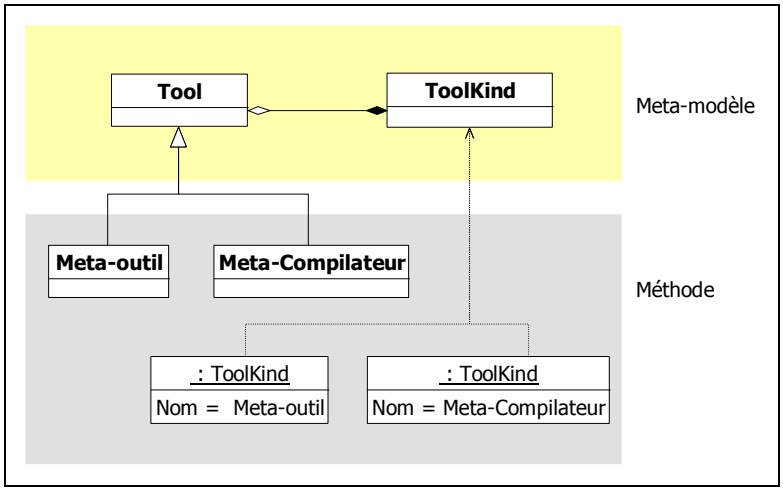

Figure 6.12 Instances du powertype *Tool/*Kind.*

WorkPerformance/*Kind

Le powertype *Workperformance/*kind* permet d'établir le lien entre les unités de travail (*WorkUnit/*Kind*) et les producteurs (*Producer/*Kind*) responsables de leur production (voir Figure 6.13 et Figure 6.14).

Le Tableau 6.13 énumère les *WorkPerformanceKind* définies par la méthode. Chaque *WorkPerformanceKind* est caractérisée par un niveau de recommandation qui détermine le degré d'obligation de l'attribution de la responsabilité. La Figure 6.13 montre une représentation graphique de l'instanciation du powertype WorkPerformance/*Kind.

Tableau 6.13 Instances du powertype *WorkPerformance/*Kind*

WorkPerformance/*Kind	**ProducerKind**	**Assigné au WorkUnit/* Kind**	**Recommandation**
WP-Analystes_WU01	Analystes	WU-01	Recommandé
WP-Analystes_WU02	Analystes	WU-02	Recommandé
WP-Expert_WU01	Expert de domaine	WU-01	Optionnel
WP-Expert_WU02	Expert de domaine	WU-02	Optionnel
WP-Développeurs_WU03	Développeurs	WU-03	Recommandé
WP-Développeurs_WU04	Développeurs	WU-04	Recommandé
WP-Testeurs_WU05	Testeurs	WU-05	Recommandé
WP- Expert_WU05	Expert de domaine	WU-05	Recommandé
WP- Développeurs_WU05	Développeurs	WU-05	Non Recommandé

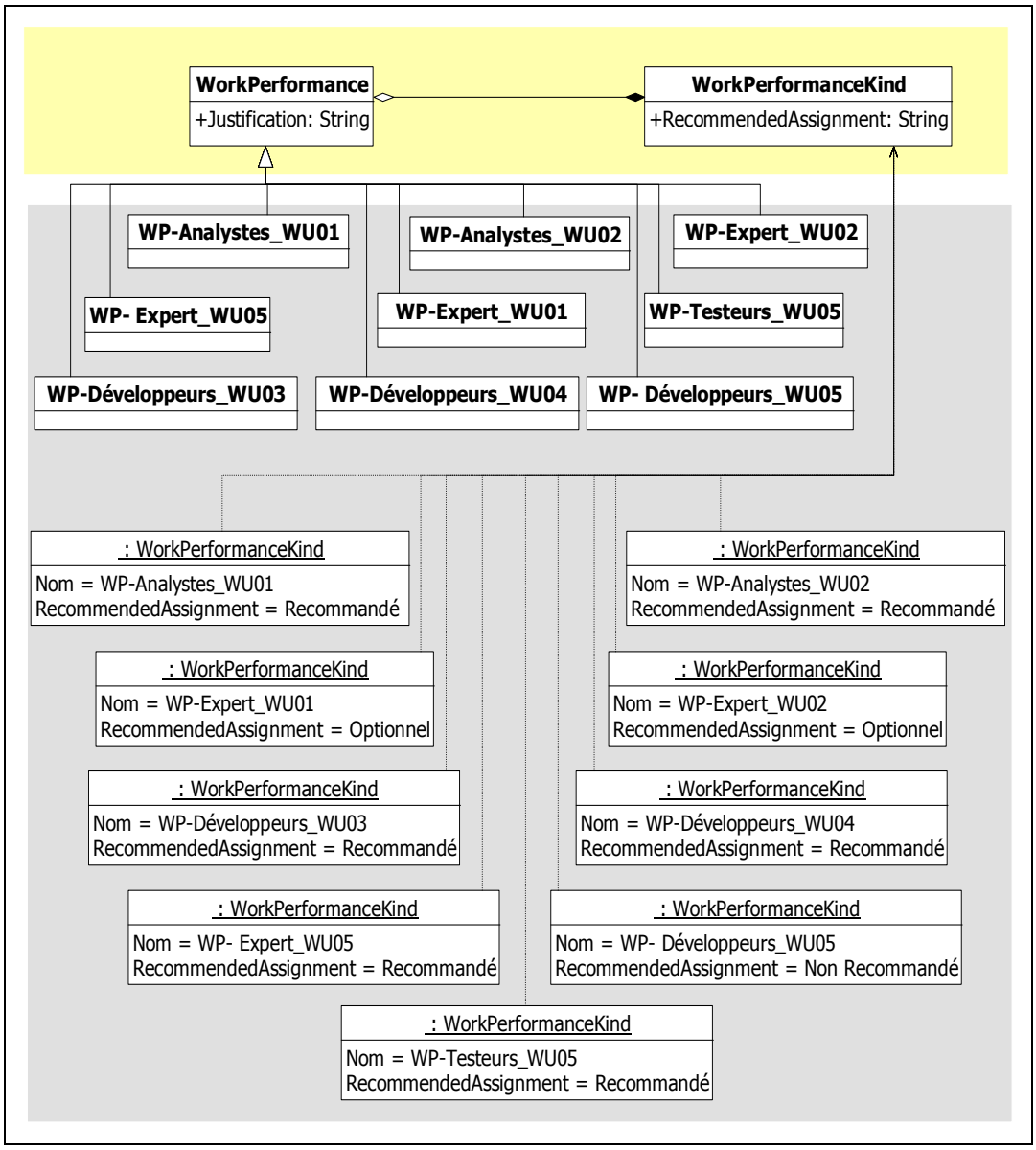

Figure 6.13 Instances du powertype *WorkPerformance/*Kind.*

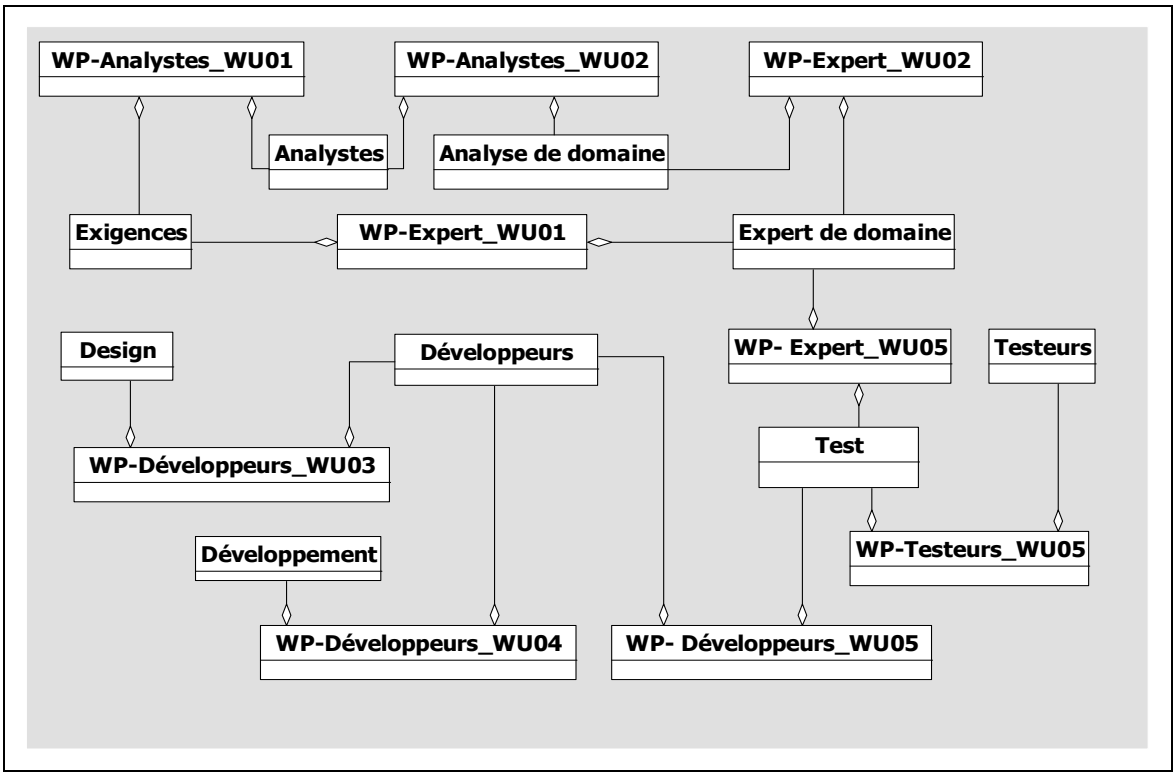

Figure 6.14 Association *WorkPerformance*, *Producer* et *WorkUnit.*

6.4 Produits

Comme la définition des DSL implique l'élaboration de modèles (ex. : modèle d'architecture, modèle de syntaxe abstraite et modèle de syntaxe concrète) et d'artefacts (ex. : document d'analyse et document de conception), il est donc nécessaire d'introduire les éléments représentant l'aspect produit dans la méthode. Cet aspect est représenté dans la norme par le powertype *Product/*Kind* et ses powertypes dérivés : *CompositeWorkProduct/*Kind*, *SoftwareItem/*Kind*, *HardwareItem/*Kind*, *Document/*Kind* et *Model/*Kind*, ainsi que par les classes *Ressource* qui s'y rattachent (e.g. *Language*, *Notation*, et *Outcome*).

Model/*Kind

Le powertype *Model/*Kind* est utilisé dans la méthode pour donner une représentation abstraite de la syntaxe abstraite, de la syntaxe concrète et de l'architecture du DSL. Une classe *ModelKind* est caractérisée par son but et son niveau d'abstraction. La Figure 6.15 montre une représentation graphique de l'instanciation du powertype *Model/*Kind.*

Le Tableau 6.14 énumère les types de modèles définis par la méthode.

Tableau 6.14 Instances du powertype *Model/*Kind*

Nom	**Description**
Modèle de domaine	Modèle décrivant les concepts du domaine pertinents pour le DSL
Modèle de syntaxe abstraite	Modèle de syntaxe abstraite exprimé en CFG ou en utilisant une approche de métamodélisation
Modèle de syntaxe concrète	Modèle de syntaxe concrète exprimé en CFG ou en utilisant une approche de métamodélisation
Modèle d'architecture	Modèle d'architecture du DSL exprimé en UML

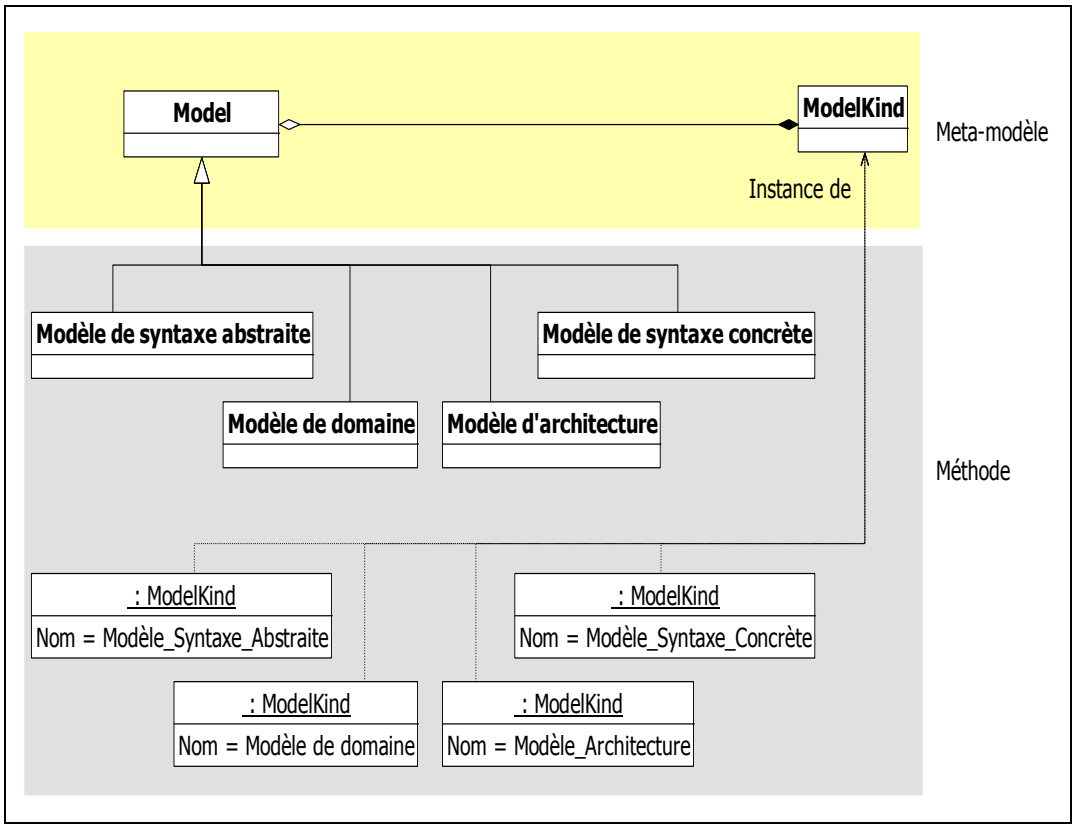

Figure 6.15 Instances du powertype *Model/*Kind.*

Langage

Les instances de la classe *Language* permettent de spécifier les langages utilisés par les différents types de modèles (*ModelKind*). Un langage est défini dans la norme comme étant une structure de *ModelUnitKind* qui met l'accent sur une perspective de modélisation donnée. *ModelKind* et *Language* sont directement liées par une association qui permet de spécifier quel langage est utilisé par quel modèle. La Figure 6.16 montre une représentation graphique de l'instanciation de la classe *Language*.

Le Tableau 6.15 énumère les langages définis par la méthode.

Tableau 6.15 Instances du powertype *Language*

Langage	**Utilisé par**	**Description**	**Exemples**
Métalangage	- Modèle de syntaxe abstraite - Modèle de syntaxe concrète	Langage de métamodélisation offrant des facilités pour la modélisation de la syntaxe abstraite, la syntaxe concrète et la sémantique.	MOF UML, Ecore
Langage CFG	- Modèle de syntaxe abstraite - Modèle de syntaxe concrète	Langage utilisé pour la définition de la grammaire non-contextuelle (*CFG*). Ce type de langages est utilisé, surtout, pour la définition des DSL textuels	EBNF, BNF,
Langage de modélisation	Modèle d'architecture	Langage de modélisation objet utilisé pour décrire l'architecture du DSL	UML
Langage de contraintes	-Modèle de syntaxe abstraite - Modèle de syntaxe concrète - Modèle d'architecture	Langage utilisé pour définir les règles de grammaire validant l'exactitude des modèles du DSL	OCL

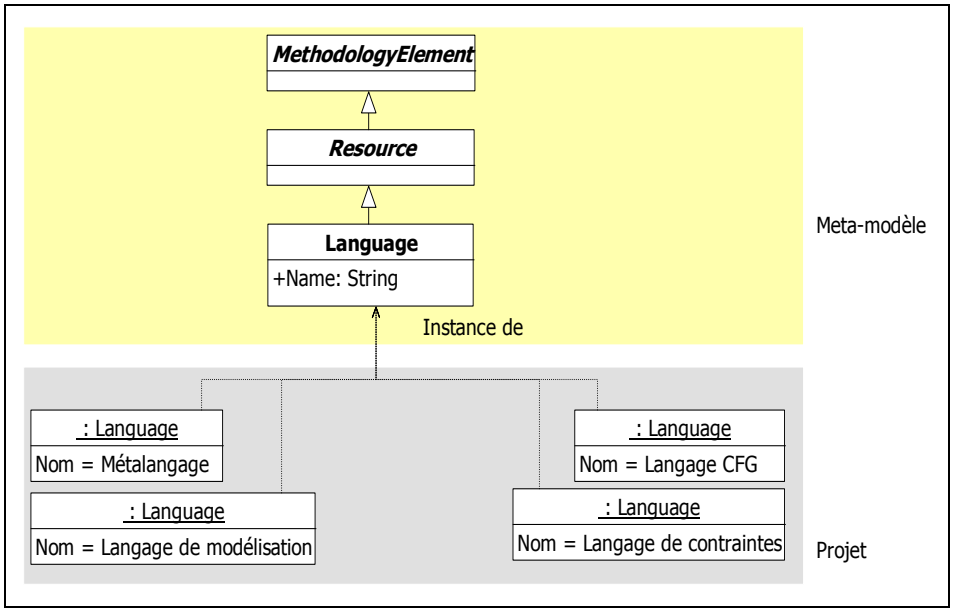

Figure 6.16 Instances de la classe *Language.*

ModelUnit/*Kind

Le powertype *ModelUnit/*Kind* représente les unités de modèles composants un modèle. Une unité de modèle est un composant atomique focalisant sur la représentation d'une partie cohésive d'information à propos du sujet modélisé. Une unité de modèle peut apparaître dans plusieurs modèles. À ce titre, *ModelUnit/*Kind* maintient une relation avec le powertype *ModelUnitUsage/*Kind* pour montrer comment chaque type d'unité de modèle (*ModelUnitUsageKind*) est utilisé dans chacun des types de modèles (*ModelKind*), et une relation avec la classe *Language* afin de déterminer les langages intégrant les différentes *ModelUnitKind.* La Figure 6.17 montre une représentation graphique de l'instanciation du powertype *ModelUnit/*Kind.*

Le Tableau 6.16 énumère les types d'unités de modèles composants les différents types d'unités de modèles définis par la méthode.

Tableau 6.16 Instances du powertype *ModelUnit/*Kind*

Nom	**Context**	**Description**
Séquence (Sequence)	EBNF BNF	Une liste ordonnée de zéro ou plusieurs éléments
Sous-séquence (*Subsequence*)	EBNF BNF	Une séquence dans une séquence
Symbole non-terminal (*non-terminal symbol*)	EBNF BNF	Une partie syntaxique du langage à définir
Méta-identifiant (*meta-identifier*)	EBNF BNF	Nom d'un symbole non-terminal
Symbole de début (*Start symbol*)	EBNF BNF	Un symbole non-terminal défini par une ou plusieurs règles de syntaxe mais qui ne participe à aucune autre règle de syntaxe
Phrase (*Sentence*)	EBNF BNF	Une séquence de symboles qui représente le symbole de début.
Symbole terminal (*Terminal symbol*)	EBNF BNF	Une séquence composée d'un ou de plusieurs caractères formant un élément irréductible du DSL

Nom	Context	Description
Package	UML MOF	Un paquetage
Class	UML MOF	Une classe
Association	UML MOF	Une association entre deux classes

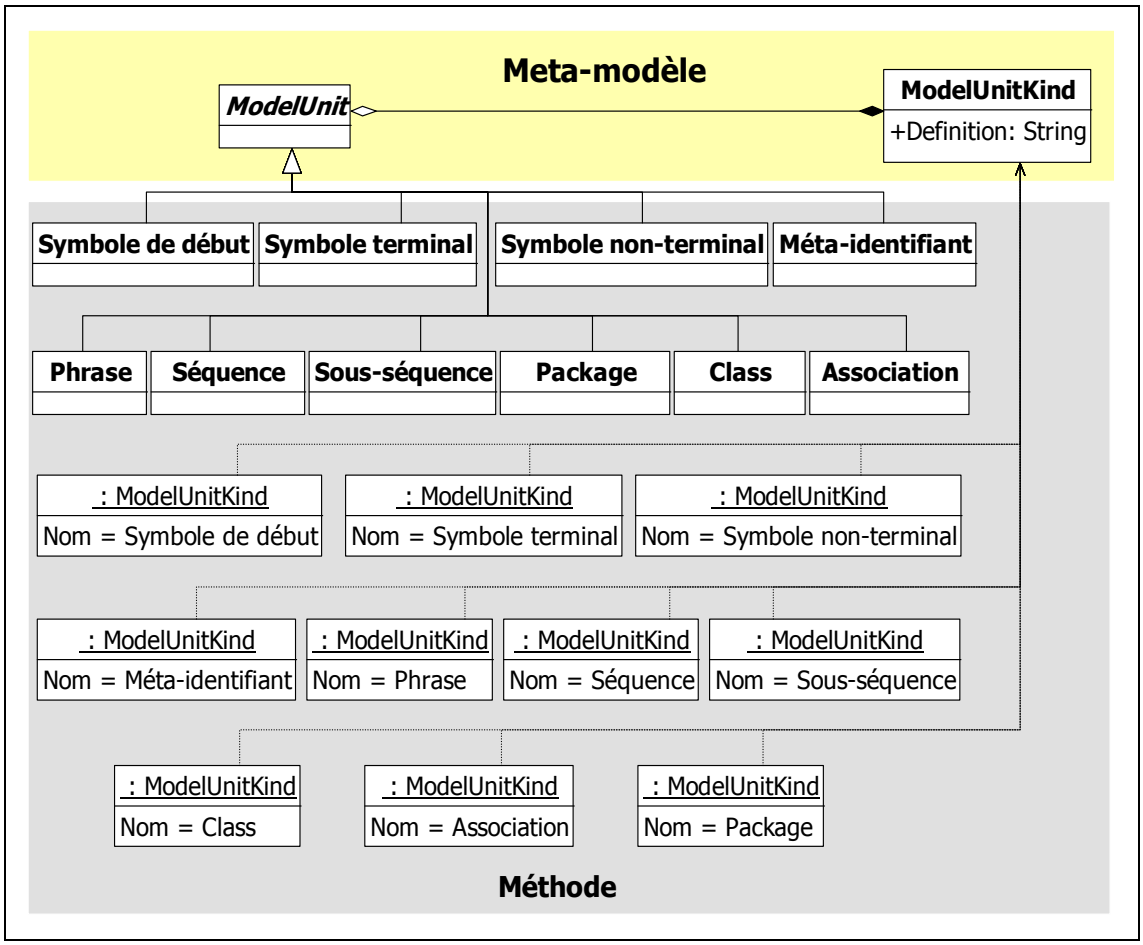

Figure 6.17 Instances du powertype *ModelUnit/*Kind.*

ModelUnitUsage/*Kind

Le powertype *ModelUnitUsage/*Kind* présente une utilisation spécifique d'une unité de modèle dans un modèle. L'utilisation d'un *ModelUnitKind* dans un *ModelKind* est spécifiée en utilisant un *ModelUnitUsageKind* (voir Figure 6.18 et Figure 6.19).

Le Tableau 6.17 énumère les types *ModelUnitUsageKind* définis par la méthode.

Tableau 6.17 Instances du powertype *ModelUnitUsage/*Kind*

ModelUnitUsage /*Kind	**Contexte (Model/*Kind)**	**ModelUnit/*Kind cible**	**NMinU[12]**	**NMaxU[13]**
MUU-01	Syntaxe abstraite	Séquence	1	N
MUU-02	Syntaxe concrète	Séquence	1	N
MUU-03	Syntaxe abstraite	Sous-séquence	1	N
MUU-04	Syntaxe concrète	Sous-séquence	1	N
MUU-05	Syntaxe abstraite	Symbole non-terminal	1	N
MUU-06	Syntaxe concrète	Symbole non-terminal	1	N
MUU-07	Syntaxe abstraite	Méta-identifiant	0	N
MUU-08	Syntaxe concrète	Méta-identifiant	0	N
MUU-09	Syntaxe abstraite	Symbole de début	1	1
MUU-10	Syntaxe concrète	Symbole de début	1	1
MUU-11	Syntaxe abstraite	Phrase	1	N
MUU-12	Syntaxe concrète	Phrase	1	N
MUU-13	Syntaxe abstraite	Symbole terminal	1	N
MUU-14	Syntaxe concrète	Symbole terminal	1	N
MUU-15	Modèle de domaine	Package	0	N
MUU-16	Modèle d'architecture	Package	0	N
MUU-17	Modèle de domaine	Class	0	N

[12] Nombre Minimum d'utilisation
[13] Nombre Maximum d'utilisation

ModelUnitUsage /*Kind	**Contexte (Model/*Kind)**	**ModelUnit/*Kind cible**	**NMinU[12]**	**NMaxU[13]**
MUU-18	Modèle d'architecture	Class	0	N
MUU-19	Modèle de domaine	Association	0	N
MUU-20	Modèle d'architecture	Association	0	N

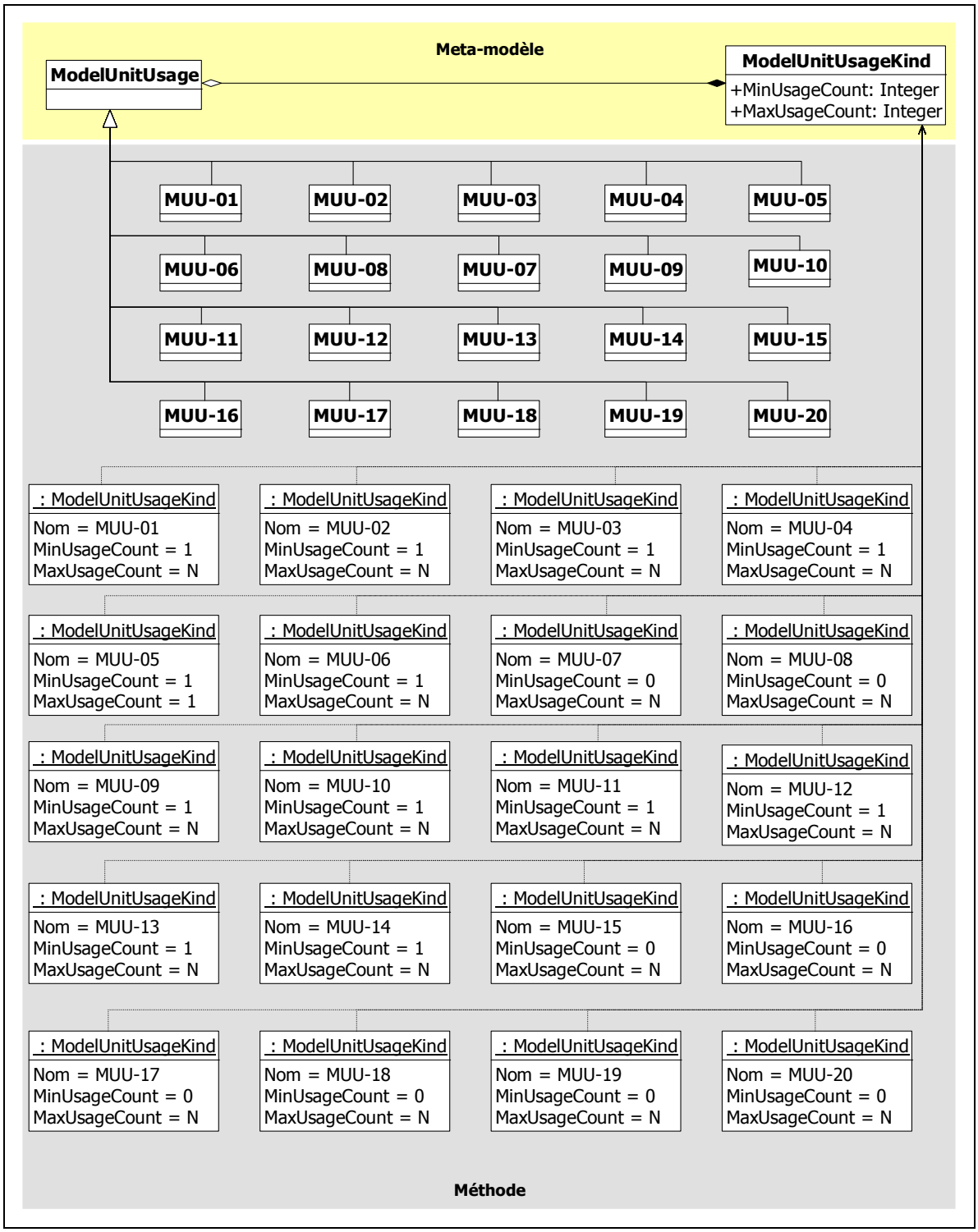

Figure 6.18 Instances du powertype *ModelUnitUsage/*Kind.*

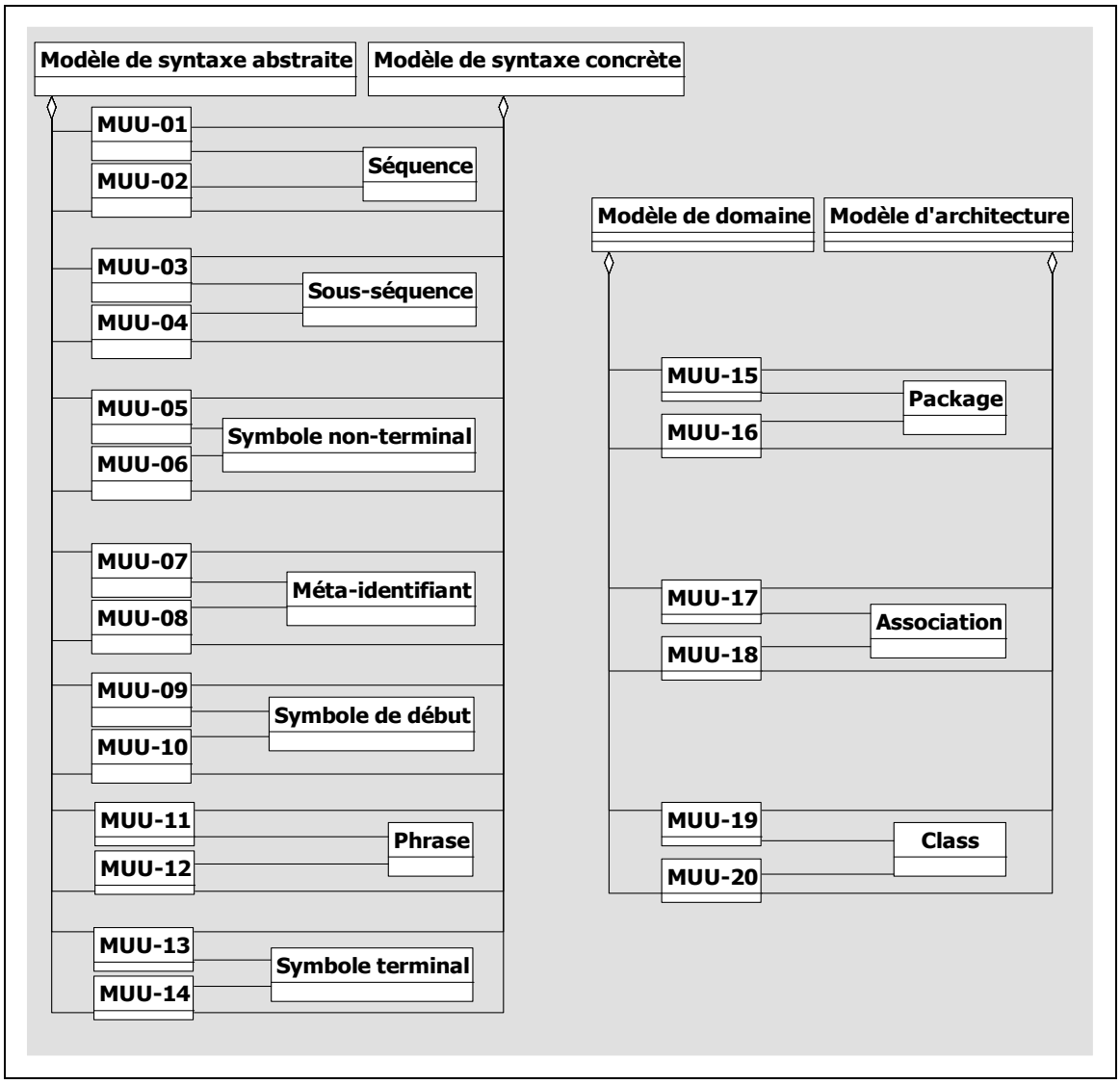

Figure 6.19 Association *ModelUnitUsage*, *ModelUnit* et *Model.*

CompositeWorkProduct/*Kind

Le powertype *CompositeWorkProduct/*Kind* représente le fait qu'un artefact (*WorkProduct*) est composé d'autres artefacts. La Figure 6.20 et Figure 6.21 montrent une représentation graphique de l'instanciation du powertype *CompositeWorkProduct/*Kind*.

Le Tableau 6.18 énumère les *CompositeWorkProductKind* définis par la méthode.

Tableau 6.18 Instances du powertype *CompositeWorkProduct/*Kind*

CompositeWorkProduct /*Kind	**Composé de WorkProduct/*Kind**	**Description**
Analyse du DSL	- Glossaire - Modèle de domaine - Points communs et variations - Système de production - Vision du DSL	Décrit l'Analyse de domaine ; soit, la portée, la terminologie, les concepts, les points communs et les variations ainsi que la vision du DSL.
Conception	- Modèle d'architecture - Modèle de syntaxe abstraite - Modèle de syntaxe concrète	Document de conception du DSL

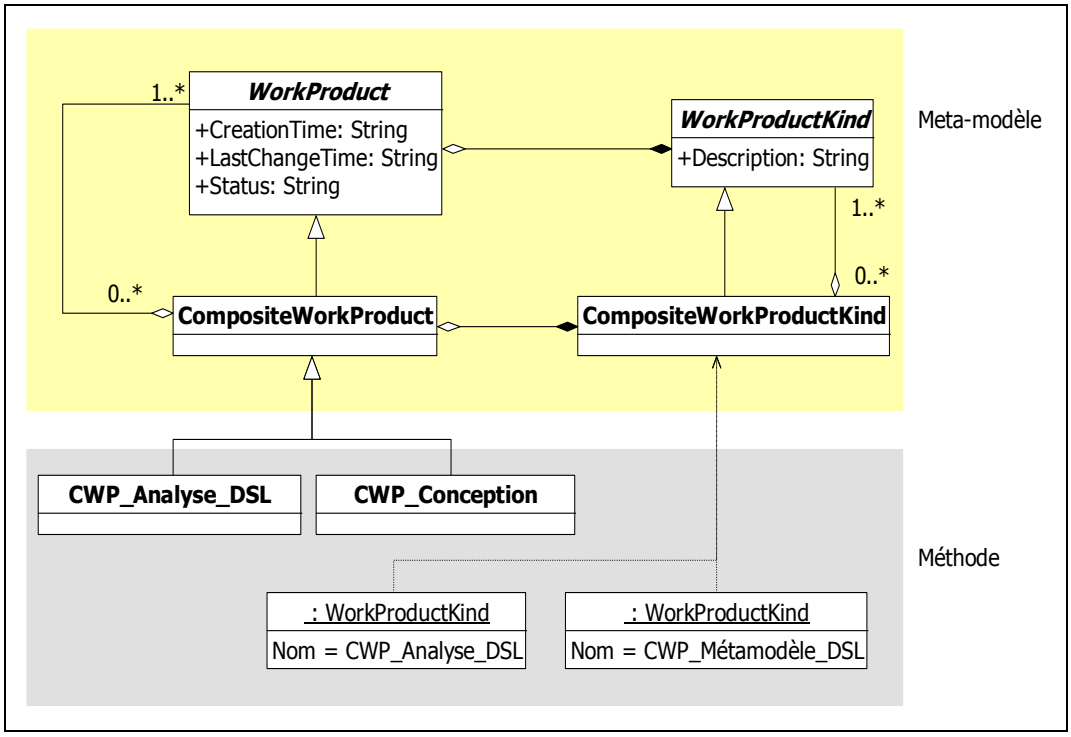

Figure 6.20 Instances du powertype *CompositeWorkProduct/*Kind.*

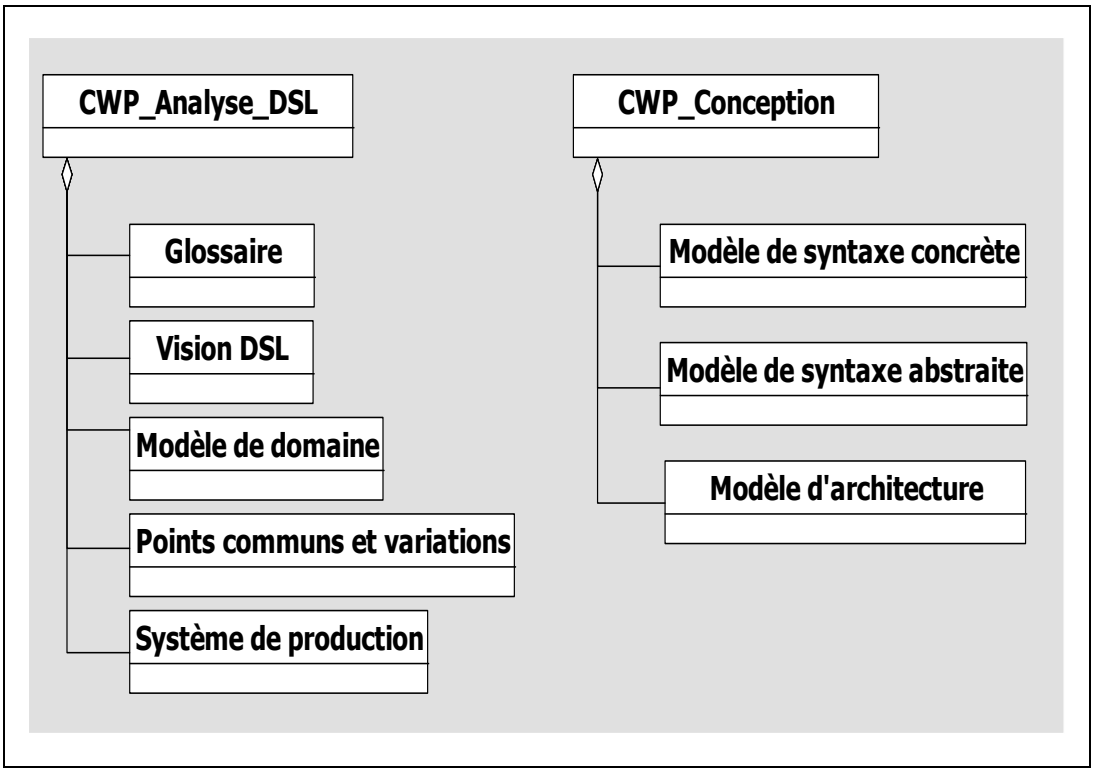

Figure 6.21 Association *CompositeWorkProduct* et *WorkProduct.*

Document/*Kind

Le powertype *Document/*Kind* est utilisé pour représenter les documents gérés durant un projet de définition de DSL. La Figure 6.22 montre une représentation graphique de l'instanciation du powertype *Document/*Kind*.

Le Tableau 6.19 énumère les *DocumentKind* définis par la méthode.

Tableau 6.19 Instances du powertype *Document/*Kind*

Nom	**Sujet représenté (WorkProduct/*Kind)**	**Description**
Document d'analyse de domaine	Analyse du DSL	Organiser et structurer la connaissance du domaine du DSL
Document de vision	Analyse du DSL	Définir la vision du DSL en termes d'exigences et de caractéristiques principales. Ce document constitue la base contractuelle pour des besoins techniques plus détaillées.
Document d'architecture	Modèle d'architecture	Définir l'architecture du DSL
Document de syntaxe abstraite	Modèle de syntaxe abstraite	Décrire la syntaxe abstraite du DSL
Document de syntaxe concrète	Modèle de syntaxe concrète	Décrire la syntaxe concrète du DSL
Document de test	Scénarios de test	Documenter le but et les objectifs du test du DSL ainsi que les scenarios et les résultats de ce test.

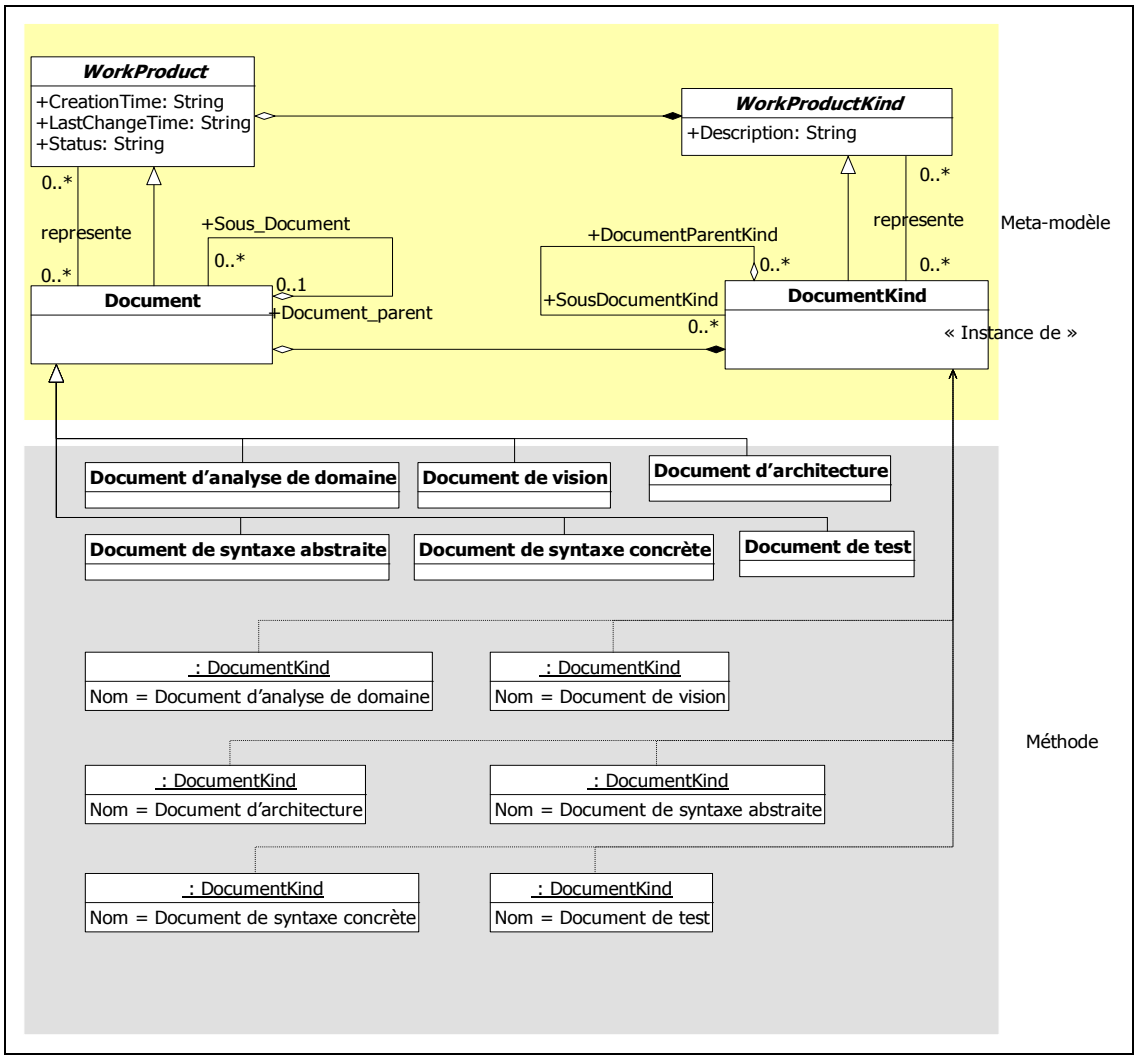

Figure 6.22 Instances du powertype *Document/*Kind.*

Outcome

La classe Outcome est utilisée pour marquer le succès de certaines unités de travail. Elle représente un résultat observable témoignant de la bonne exécution d'une unité de travail. La Figure 6.23 montre une représentation graphique de l'instanciation de la classe *Outcome.*

Le Tableau 6.20 énumère les *Outcome* définis par la méthode.

Tableau 6.20 Instances de la classe *Outcome*

Nom	**Est un résultat de**	**Description**	**NMC**
O_Exigences	WU-01	Les exigences du DSL sont identifiées, décrites et validées par les commanditaires du DSL.	1
O_Architecture	WU-03	L'architecture du DSL a été mise en place	1
O_Syntaxe_abstraite	WU-04	La syntaxe abstraite du DSL est définie	1
O_Syntaxe_concrète	WU-04	La syntaxe concrète du DSL est définie	1
O_Sémantique	WU-04	La sémantique du DSL est définie	1
O_Test	WU-05	Le DSL est testé et validé	1

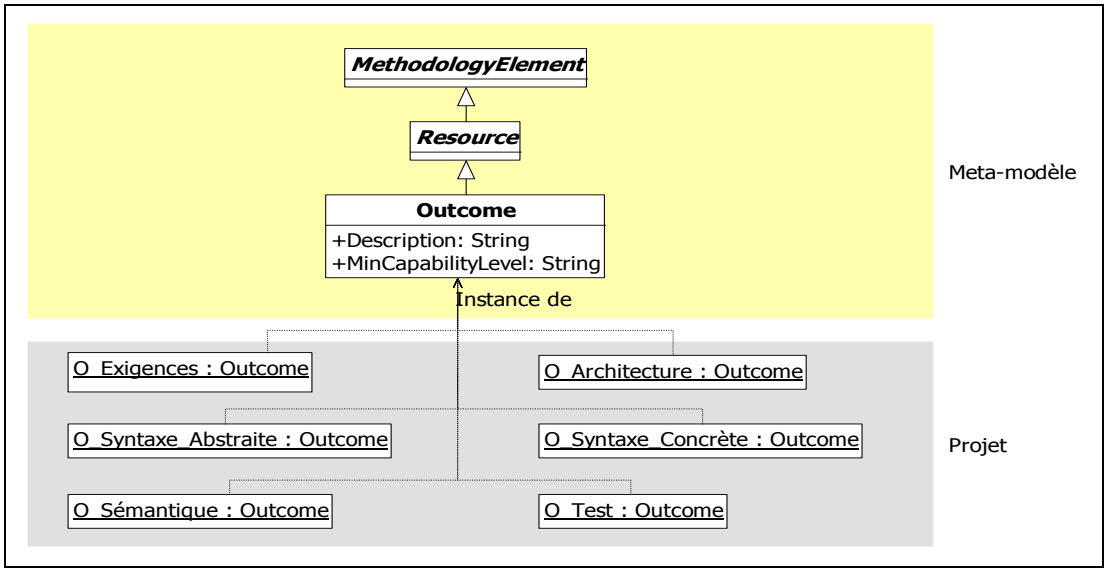

Figure 6.23 Instances de la classe *Outcome.*

6.5 Lien entre l'aspect Processus et l'aspect Produit

Pour montrer les interactions entre l'aspect Processus et l'aspect Produit d'une méthode, SEMDM utilise le powertype *Action/*Kind*. Les instances de ce powertype représentent le fait que certaines tâches agissent sur certaines unités de travail.

Action/*Kind

Une action est un événement d'utilisation effectué par une tâche sur un produit du travail. L'attribut *Type* de la classe *ActionKind* indique le type de l'action exercée sur le *WorkProductKind* (création, modification lecture, etc.). La Figure 6.24 montre une représentation graphique de l'instanciation du powertype *Action/*Kind*.

Le Tableau 6.21 liste les types d'actions définis par la méthode.

Tableau 6.21 Instances du powertype *Action/*Kind*

Nom	**Contexte Task/*Kind**	**Agit sur WorkProduct/*Kind**	**Type**
ACTION–01	Développer la vision	Document de vision	C
ACTION–02	Définir la portée du domaine	Document d'analyse	C
ACTION–03	Définir la terminologie	Document d'analyse	M
ACTION–04	Définir les abstractions	Document d'analyse	M
ACTION–05	Définir le système de production	Document d'architecture	C
ACTION–06	Établir une architecture	Document d'architecture	M
ACTION-07	Définir la syntaxe abstraite	Document de syntaxe abstraite	C
ACTION-08	Définir la syntaxe concrète	Document de syntaxe concrète	C
ACTION-09	Préparer les scénarios de test	Document de test	C
ACTION-10	Exécuter les scénarios de test	Document de test	L/M

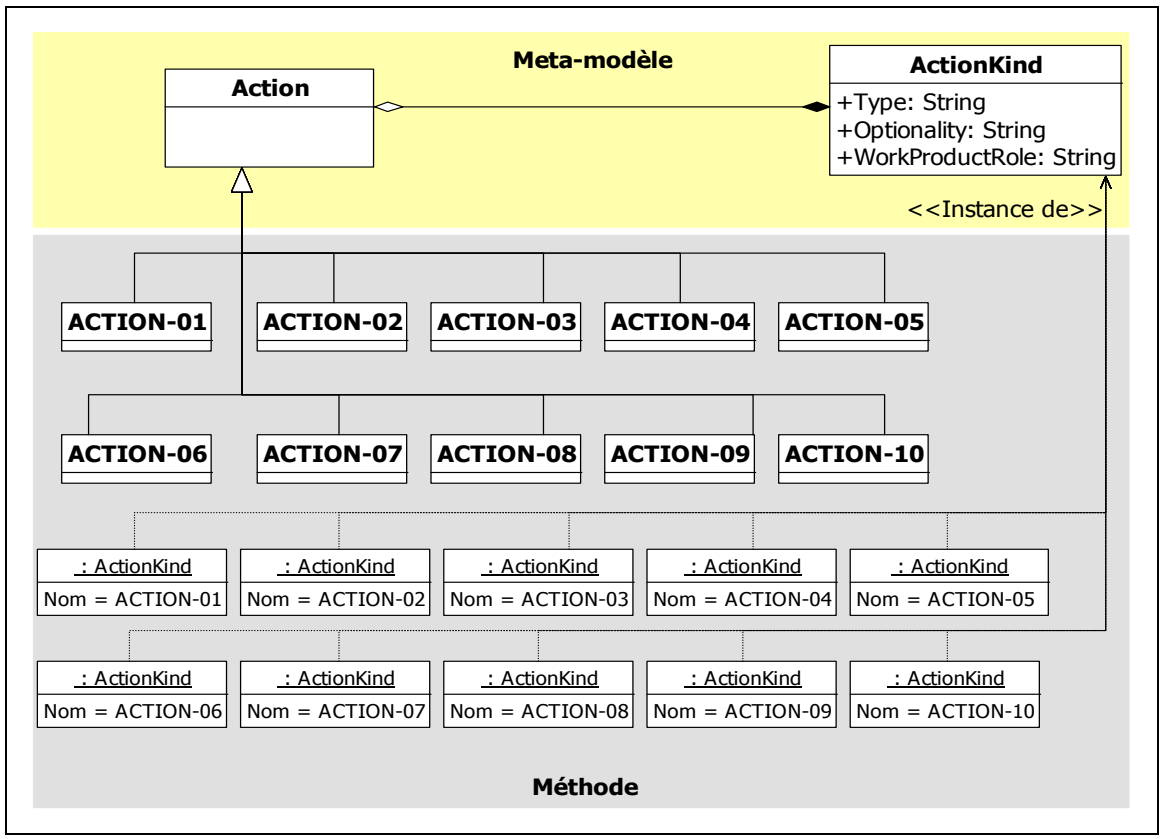

Figure 6.24 Instances du powertype *Action/*Kind.*

CHAPITRE 7

FACTEURS DE SUCCÈS ET ATTRIBUTS DE QUALITÉ DES DSL

À ce stade nous avons terminé la phase propositionnelle en présentant la méthode qu'on propose pour la définition des DSL et nous sommes prêts à entamer la phase d'expérimentation et d'évaluation dont l'objectif est de s'assurer de l'efficacité et de l'applicabilité de la méthode proposée. L'évaluation de la méthode peut se faire à deux niveaux : une évaluation de la constitution, c'est-à-dire de la structuration et de l'organisation de la méthode, et une évaluation de l'extrant produit par cette méthode (*Outcome*), soit les DSL résultants.

L'évaluation de la constitution de la méthode fait peu de sens dans notre cas vu que celle-ci a été créée conformément à la norme ISO/IEC 24744. En revanche, une évaluation du degré de conformité de la méthode à la norme serait plus pertinente. Or, comme déjà présenté dans le chapitre 6, la méthode proposée est complètement conforme à la norme de fond en comble.

Le Tableau 7.1 montre les éléments du SEMDM utilisés pour définir la méthode. À noter que la norme ISO/IEC 24744 n'a défini aucun élément comme étant obligatoire pour la définition d'une méthode. C'est au concepteur de décider quels sont les éléments qui sont pertinents pour la définition de sa méthode.

Tableau 7.1 Éléments du SEMDM utilisés dans la méthode

Élément SEMDM	**Implémentation dans la méthode**
Action/*Kind	✓
Build/*Kind	✓
CompositeWorkProduct/*Kind	✓
Conglomerate	
Constraint	
Document/*Kind	✓
Guideline	
HardwareItem/*Kind	
InstantaneousStage/*Kind	✓
Language	✓
Milestone/*Kind	✓
Model/*Kind	✓
ModelUnit/*Kind	✓
ModelUnitUsage/*Kind	✓
Notation	
Outcome	✓
Person	
PhaseKind	✓
PostCondition	
PreCondition	
Process/*Kind	✓
Producer/*Kind	✓
RoleKind	✓
SoftwareItem/*Kind	✓
Source	✓
StageKind	✓
StageWithDuration/*Kind	✓
TaskKind	✓
TaskTechniqueMapping/*Kind	✓
Team/*Kind	✓
Technique/*Kind	✓
TimeCycle/*Kind	✓
Tool/*Kind	✓
WorkPerformance/*Kind	✓
WorkProduct/*Kind	✓
WorkUnit/*Kind	✓

La deuxième forme d'évaluation consiste à évaluer la qualité des DSL produits en appliquant la méthode. Malheureusement, jusqu'à date, il n'y a pas de normes ni de modèles reconnus qui définissent les attributs de qualité pour les langages de programmation et de modélisation et encore moins pour les DSL.

Ce chapitre présente le résultat d'une étude que nous avons menée afin d'élaborer une liste d'attributs de qualité pour les DSL. L'approche consiste à étudier les facteurs de succès des DSL et à proposer une technique pour en tirer les attributs de qualité. Ce sont ces attributs qui serviront de critères d'évaluation lors de la phase d'expérimentation de la méthode.

7.1 Facteurs de succès des DSL

Portée du domaine : la portée du domaine est d'une grande importance et doit être soigneusement déterminée. Elle constitue un facteur déterminant pour le succès du DSL. Si la portée est trop large, le DSL sera moins précis et moins expressif et si elle est trop étroite, le DSL sera trop spécifique et ses opportunités d'utilisation seront très limitées.

Connaissance du domaine : le développement de DSL requiert une connaissance approfondie du domaine. Une bonne connaissance du domaine permet d'identifier correctement les concepts du domaine et d'exprimer convenablement ses règles et ses contraintes.

Niveau d'abstraction : travailler au juste niveau d'abstraction permet de définir des abstractions pertinentes et d'un fort niveau d'abstraction et, par conséquent, d'offrir un DSL avec des concepts plus familiers aux utilisateurs et aux experts de domaine.

Notation : la notation est souvent prise à la légère par les concepteurs des DSL alors qu'elle constitue l'interface principale d'interaction avec les utilisateurs. Une notation justifiée par une analyse détaillée des besoins, des attentes et des habitudes de ces utilisateurs est plus susceptible d'être acceptée et utilisée. Le concepteur du DSL doit utiliser, autant que possible, les notations et les représentations utilisées par les experts du domaine.

Infrastructure technique : consiste en les méthodes d'analyse de domaine et les approches de conception et d'implémentation de DSL.

- L'utilisation des méthodes d'analyse de domaine permet de systématiser le processus d'analyse de domaine, soit la collecte, l'organisation et la modélisation de la connaissance du domaine (section 2.6.2). Toutefois, il faut choisir la méthode la plus adéquate au développement de DSL, c'est-à-dire une méthode dont les artefacts sont plus facilement exploités dans la conception et l'implémentation du DSL. Nous pensons que les méthodes qui se basent sur l'analyse des points communs et des variations sont plus facilement exploitables dans l'analyse de DSL, et ce en raison du fait que les DSL sont utilisés, généralement, soit comme des interfaces d'accès aux fonctionnalités des actifs fondamentaux de la famille, soit comme des outils de configuration permettant de spécifier les paramètres de variations pour les membres de la famille (Coplien, Hoffman et Weiss, 1998) ;
- L'infrastructure de conception consiste en les approches, les techniques et les patrons de conception utilisés pour la définition des DSL. Cette infrastructure devrait offrir les éléments nécessaires pour définir les aspects principaux du DSL, soit la syntaxe abstraite, la syntaxe concrète et la sémantique.

Voici quelques caractéristiques à considérer lors de la mise en place d'une telle infrastructure :

- Unification : caractérise la possibilité à définir les DSL d'une manière unifiée qui leur permet de coexister sans conflit. Comme cela, les DSL peuvent être gérés et manipulés de manière uniforme, ce qui aide à surmonter le problème de la diversité des langages ;
- Simplicité : caractérise la capacité de l'infrastructure à offrir les éléments permettant d'accomplir les tâches de définition du DSL d'une manière simple et intuitive ;
- Clarté : caractérise la capacité à rendre la compréhension de la définition du DSL plus facile ;
- Capacité fonctionnelle : caractérise la capacité de l'infrastructure à offrir des fonctions qui répondent aux besoins des développeurs de DSL.

L'infrastructure d'implémentation consiste en les approches, les techniques et les patrons utilisés pour l'implémentation des DSL. Cette infrastructure est de loin la plus importante et la plus décisive pour le succès d'un DSL. Elle détermine la capacité de générer des applications à partir des modèles du DSL. Un DSL dont les modèles ne sont pas exécutables est de peu de valeur.

Outils de support: couvrent les outils d'analyse de domaine, les outils de conception et les outils d'implémentation. Les outils d'analyse de domaine supportent, entre autres, la collecte, l'organisation et la structuration de la connaissance du domaine. Les outils de conception supportent principalement la création et le stockage des définitions des DSL (syntaxe abstraite, syntaxe concrète et règles de grammaire). Les outils d'implémentation servent à la création des interfaces d'utilisation du DSL (éditeurs de modélisation, outils de vérification/validation des modèles, etc.) et des outils de génération de code (compilateurs, transformateurs de modèles, librairies, etc.).

Expertise en développement de langage : le développement de DSL est un processus délicat qui demande des connaissances solides en développement de langages. Le manque de connaissances en ce domaine peut affecter négativement la qualité d'un DSL. Idéalement, le développement de DSL devrait se réaliser par une équipe d'ingénierie de domaine qui possède une connaissance dans le domaine et une expertise en développement de langages. Ces deux compétences augmentent considérablement les chances d'aboutir à un DSL utile et pratique pour les utilisateurs (Mernik, Heering et Sloane, 2005).

Soutien des parties prenantes : l'approbation du DSL par les différentes parties prenantes et en particulier les experts de domaines est essentielle pour son adoption. Les développeurs de DSL doivent faire en sorte qu'ils construisent des DSL qui répondent bien aux besoins de leurs utilisateurs.

Processus de développement : Contrairement aux langages généralistes, le développement des DSL se fait toujours d'une manière ad hoc. Afin de mieux maitriser cette activité, il va falloir rassembler l'ensemble des activités et des pratiques et les intégrer dans un processus

cohérent décrivant explicitement les activités à réaliser, les artefacts à produire et les intervenants impliqués.

7.2 Classification des facteurs de succès

Dans cette section, nous classons les facteurs de succès définis selon un système de classement similaire à celui proposé par Wile (Wile, 2004) (voir Tableau 7.2). Le but de cette classification est de déterminer la nature des facteurs de succès afin de faciliter leur gestion. Par exemple, si on cherche à agir sur un facteur de nature organisationnelle, alors il faut évaluer les aspects de l'organisation touchant ce facteur. Voici une description sommaire des catégories définies :

- **Organisationnel :** caractérise les facteurs liés à la culture organisationnelle, c'est-à-dire à la mission de l'organisation, ses normes, ses pratiques, etc. ;
- **Personnel :** caractérise les facteurs liés aux capacités des ressources humaines, soit l'expertise, les compétences, les expériences, etc. ;
- **Social :** caractérise les facteurs liés aux relations et aux comportements sociaux ;
- **Technique :** caractérise les facteurs liés à des questions techniques, tels que les outils et les technologies.

Tableau 7.2 Classification des facteurs de succès des DSL

Facteur de succès	Technique	Organisationnel	Social	Personnel
Portée de domaine		X		
Connaissance du domaine				X
Niveau d'abstraction	X			
Notation	X			
Infrastructure technique	X			
Outils de support	X			
Expertise en développement des langages		X		X
Soutien des parties prenantes			X	
Processus de développement		X		

7.3 Des facteurs de succès aux attributs de qualité

Afin de transformer les facteurs de succès en critères d'évaluation pour les DSL, nous proposons une technique à six étapes. Cette technique a la caractéristique d'être générique et non spécifique au domaine des DSL. La Figure 7.1 montre les étapes à suivre pour déterminer des critères d'évaluation à partir des facteurs de succès.

1. **Identification des aspects :** l'identification des aspects clefs du sujet (dans notre cas le sujet est un DSL). Pour les DSL nous avons identifié trois aspects : la syntaxe abstraite, la syntaxe concrète et la sémantique;
2. **Identification des impacts :** la présence ou l'absence d'un facteur de succès a un impact direct ou indirect sur le sujet. L'impact peut être positif ou négatif et affecte généralement un ou plusieurs aspects du sujet. Il est important de donner une description détaillée de l'impact. Plus cette description est complète et détaillée, plus il devient facile d'identifier les attributs de qualité;
3. **Identification des valeurs ajoutées / perdues :** généralement un impact positif est caractérisé par une ou plusieurs valeurs ajoutées. De même, un impact négatif cause la perte d'une ou de plusieurs valeurs. Le but ici est d'énumérer les valeurs en rapport avec chaque impact. Souvent cette opération est effectuée lors de la description détaillée des impacts;
4. **Identification des aspects touchés :** identifier les aspects touchés par chaque impact. Le but est de déterminer les aspects en rapport avec les valeurs de l'impact;
5. **Élaboration de la liste des critères :** pour chaque aspect touché, on détermine les critères permettant d'évaluer l'impact en se basant sur les valeurs correspondantes. Il est préférable d'avoir un modèle d'évaluation comme référence. Dans notre cas nous avons utilisé le modèle de la qualité des produits logiciels défini dans la norme ISO / IEC 9126 (ISO, 2001);
6. **Épuration de la liste des critères :** la liste des critères établie dans l'étape précédente peut contenir des doublons, des critères difficiles à mesurer, etc. Le but de cette étape est de peaufiner cette liste en ne gardant que les critères pertinents pour l'évaluation du sujet.

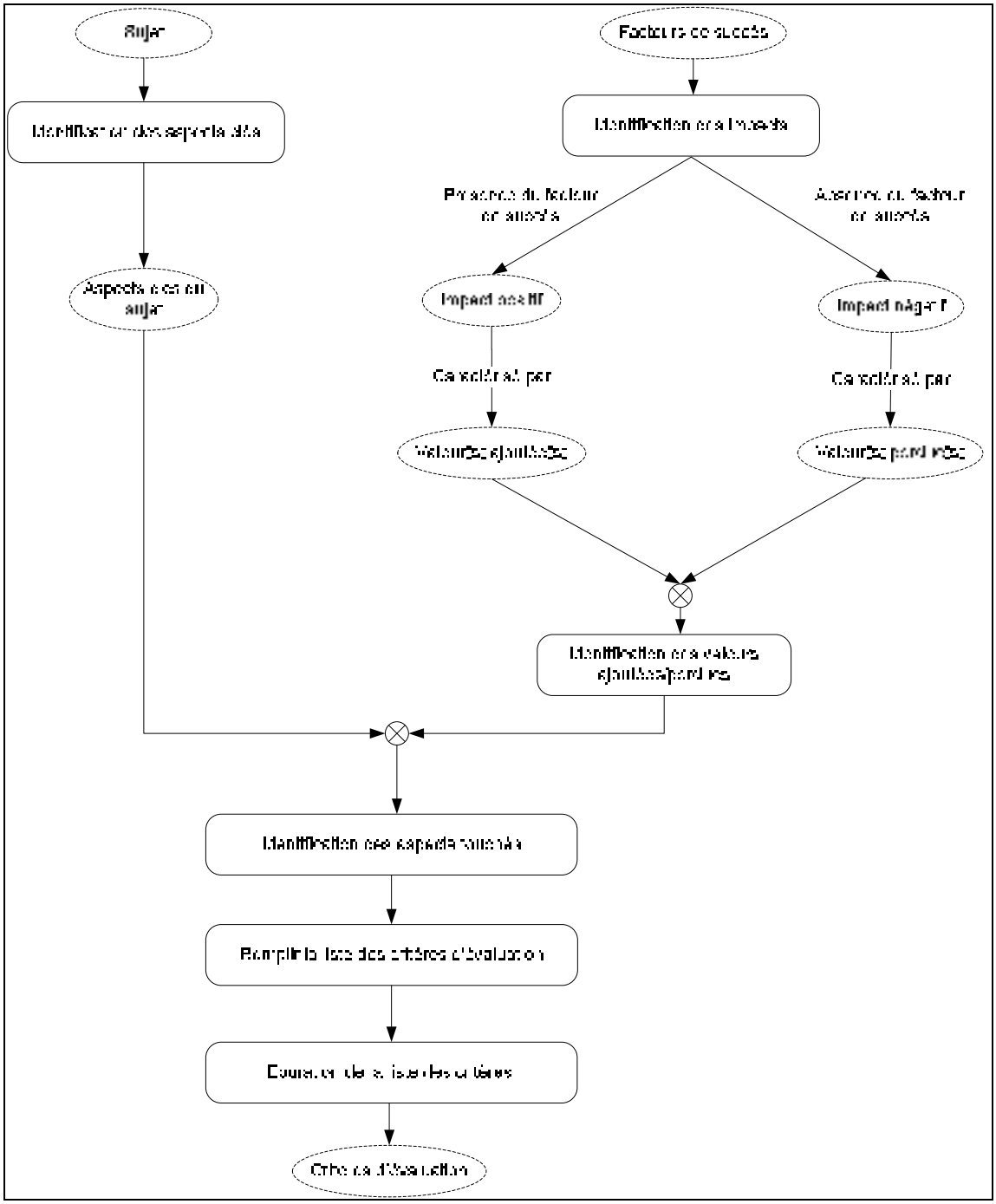

Figure 7.1 Étapes de transformation des facteurs de succès en attributs de qualité.

7.4 Application de la technique aux DSL

7.4.1 Portée du domaine

Impacts positifs

- Domaine soigneusement défini : la portée du domaine correspond exactement aux besoins exprimés. Ainsi, le domaine ne contient que les concepts pertinents aux préoccupations des utilisateurs.

Impacts négatifs

- Domaine trop large : le développement et la maintenance du DSL deviennent plus difficiles et plus coûteux. Un DSL conçu pour un tel domaine a tendance à être plus générique et plus difficile à apprendre, à maintenir et à tester ;
- Domaine trop étroit : dans ce cas, le DSL devient trop spécifique et ses opportunités d'utilisation sont réduites.

Aspects touchés

- Syntaxe abstraite : les concepts et leur niveau d'abstraction dépendent, en partie, de l'étendue de la portée du domaine. Si la portée est trop large on tendra à utiliser des concepts plus génériques afin de pouvoir répondre aux besoins divers couverts par cette portée.

Critères d'évaluation

- Expressivité : capacité du DSL à offrir des concepts expressifs et d'un fort niveau d'abstraction ;
- Facilité d'exploitation : attribut du DSL portant sur l'effort fourni par l'utilisateur afin de l'exploiter et de contrôler son exploitation ;
- Facilité d'apprentissage : attribut du DSL portant sur l'effort fourni par l'utilisateur afin de l'apprendre.

- Facilité de compréhension : attribut du DSL portant sur l'effort fourni par l'utilisateur afin de comprendre comment il peut être utilisé pour réaliser les différentes tâches de programmation/modélisation ;
- Facilité de test : attribut du DSL portant sur l'effort nécessaire pour le valider après une modification ;
- Convenance (*Suitability*) : capacité du DSL à fournir un ensemble approprié de fonctions répondant aux besoins des utilisateurs.

7.4.2 Connaissance du domaine

Impacts positifs

- Comprendre et parler le langage des experts de domaine, ce qui permet de renforcer et de pérenniser la collaboration entre les développeurs de DSL et les experts de domaine ;
- Compréhension approfondie des besoins, ce qui permet de réduire les incertitudes et de construire des modèles de domaine plus stables ;
- Les modèles de domaine deviennent une source d'informations pour le développement de DSL ;
- Concepts et notations adaptés au domaine.

Impacts négatifs

- Communication difficile entre les développeurs et les experts de domaine;
- DSL basé sur des perceptions plutôt que sur une connaissance objective et des éléments factuels;
- Une connaissance superficielle du domaine risque d'affecter le bon déroulement du développement du DSL en termes de temps et d'efforts.

Aspects touchés

- Syntaxe abstraite : une bonne connaissance de domaine permet de définir une syntaxe abstraite représentant les concepts qui sont d'intérêt pour les experts de domaines. En revanche, une connaissance insuffisante du domaine risque d'engendrer des concepts inadaptés ;

- Syntaxe concrète : une bonne connaissance du langage et des notations utilisées par les experts de domaine permet d'adopter une notation pour le DSL qui leur est familière. Par contre, la méconnaissance de ces aspects peut aboutir à une notation inadéquate.

Critères d'évaluation

- Expressivité : capacité du DSL à offrir des concepts expressifs et d'un fort niveau d'abstraction ;
- Convenance (*Suitability*) : capacité du DSL à fournir un ensemble approprié de fonctions répondant aux besoins des utilisateurs.

7.4.3 Niveau d'abstraction

Impacts positifs

- Concepts proches du domaine du problème, ce qui permet de concevoir les solutions à un haut niveau d'abstraction ;
- Notation familière aux utilisateurs : les DSL définis à un haut niveau d'abstraction tendent à proposer une notation plus naturelle pour le domaine ;
- Séparation des préoccupations : séparer entre le quoi et le comment. Les experts de domaine spécifient les caractéristiques de l'application sans trop se préoccuper des détails techniques ;
- Détection des erreurs tôt dans la phase de conception : les DSL avec un fort niveau d'abstraction manipulent des modèles expressifs et, surtout, compréhensibles par les experts de domaine. Ces derniers peuvent comprendre, valider et modifier ces modèles tôt dans la phase de spécification (conception).

Impacts négatifs

- Si le niveau d'abstraction est trop bas alors les concepts seront plus proches du domaine d'implémentation que du domaine du problème. En revanche, un niveau d'abstraction trop abstrait risque de rendre l'implémentation du DSL plus difficile, voire impossible ;

- Un grand écart entre le niveau d'abstraction du DSL et celui du domaine du problème oblige le concepteur de la solution à passer par une activité mentale de mise en correspondance entre les concepts du DSL et ceux du domaine du problème.

Aspects touchés

- Syntaxe abstraite : la syntaxe abstraite du DSL dépend, en partie, du niveau d'abstraction dans lequel on travaille. Un fort niveau d'abstraction génère une syntaxe abstraite forte en abstraction basée sur les concepts du domaine plutôt que sur les concepts d'implémentation.

Critère d'évaluation

- Expressivité : la capacité du DSL à offrir des concepts expressifs et d'un fort niveau d'abstraction.

7.4.4 Notation

Impacts positifs

- Facilité d'utilisation ;
- Approbation des utilisateurs : un DSL adoptant une notation intuitive, conviviale et attrayante est plus susceptible de fidéliser ses utilisateurs et de recevoir leur approbation.

Impacts négatifs

- Interface subtile et ambigüe ;
- Résistance des utilisateurs : une notation subtile et complexe peut engendrer la résistance, voire le rejet du DSL par ses utilisateurs.

Aspects touchés

- Syntaxe concrète

Critères d'évaluation

- Pouvoir d'attraction : attribut portant sur la capacité du DSL à être attractif pour ses utilisateurs ;
- Facilité d'exploitation : attribut du DSL portant sur l'effort fourni par l'utilisateur afin de l'exploiter et de contrôler son exploitation.

7.4.5 Infrastructure technique

Impacts positifs

- Connaissance de domaine bien structurée : le patrimoine du domaine est organisé en modèles de domaine représentant, entre autre, la définition du domaine (portée du domaine), sa terminologie, la description de ses concepts, les points communs et les variations ;
- Systématisation du processus de développement des DSL en adoptant des pratiques éprouvées et démontrées ;
- Unification du développement : les projets de développement logiciel deviennent de plus en plus complexes et nécessitent souvent plusieurs DSL pour répondre aux différentes préoccupations. L'existence d'une infrastructure technique commune permet de développer les DSL d'une manière unifiée et interopérable (Clark, Sammut et Willans, 2008) ;
- Faciliter la communication entre les parties prenantes : l'unification des pratiques de développement, surtout au niveau de la modélisation, facilite la communication entre les développeurs, les experts de domaine et les utilisateurs ;
- Meilleures opportunités de réutilisation : les définitions des DSL existants peuvent être réutilisées pour la définition de nouveaux DSL. Par exemple, si on adopte une approche de métamodélisation pour la définition des DSL, alors on peut définir de nouveaux DSL en utilisant les mécanismes de spécialisation et de référencement offerts par cette approche, c'est-à-dire en spécialisant ou en faisant référence à un ou plusieurs DSL existants ;

- Amélioration de la productivité : l'automatisation est le plus efficace des moyens technologiques utilisés pour stimuler la productivité (Selic, 2003). L'infrastructure d'implémentation constitue la pierre angulaire de l'automatisation de la génération d'application à partir des modèles du DSL.

Impacts négatifs

- Limiter les opportunités de coexistence : développer les DSL de manière improvisée sans s'appuyer sur une infrastructure qui supporte efficacement le développement des DSL peut rendre l'interopérabilité et la coexistence de plusieurs DSL très difficile ;
- Limiter les opportunités de réutilisation : le développement improvisé de DSL peut affecter négativement la capacité du DSL à être utilisé pour la définition d'autres DSL ;
- Limiter les capacités d'exécution : les DSL ont été prouvés efficaces pour augmenter la productivité du développement logiciel. Cette qualité dépend de la capacité du DSL à générer des applications à partir de ses modèles. C'est le but de l'infrastructure d'implémentation qui offre les actifs fondamentaux capables de générer de telles applications ;

Aspects touchés

- Syntaxe abstraite : les éléments de l'infrastructure technique qui affectent la syntaxe abstraite sont : l'infrastructure de conception et celle de l'analyse de domaine. La première sert à modéliser les concepts du DSL et leurs relations (syntaxe abstraite) tandis que les méthodes d'analyse de domaine servent à extraire les concepts pertinents au domaine ;
- Syntaxe concrète : l'infrastructure de conception sert également à la modélisation de la syntaxe concrète et à la définition des correspondances (*mapping*) entre celle-ci et la syntaxe abstraite ;
- Sémantique : l'infrastructure de conception contient aussi les éléments permettant la description de la sémantique du DSL.

Critères d'évaluation

- Facilité de modification : attribut du DSL portant sur l'effort nécessaire à l'implémentation d'une modification ou à la correction d'une anomalie ;
- Facilité de réutilisation : attribut du DSL portant sur l'effort nécessaire pour son utilisation dans la définition de nouveaux DSL ;
- Capacité d'exécution : attribut du DSL portant sur sa capacité à générer des applications à partir de ses modèles ;
- Réutilisabilité : capacité du DSL à être réutilisé, dans le même domaine, sans modification ou avec des modifications mineures.

7.4.6 Outils de support

Impacts positifs

- Démocratisation du développement des DSL : les outils constituent un grand pas vers la démocratisation du développement des DSL. Ces outils offrent des facilités pour la définition et l'édition des DSL, incluant des fonctionnalités de métamodélisation et de stockage de modèles, des frameworks pour la génération de générateurs de code, des assistants pour la définition d'éditeurs graphiques, etc. ;
- Amélioration de la productivité : les outils de support permettent d'augmenter la productivité à deux niveaux. D'abord au moment de la définition des DSL, en offrant les facilités déjà mentionnées dans le point précédent. Ensuite, au moment de la génération des applications. Ainsi, une grande partie des applications peut être générée à partir de spécifications exprimées dans les modèles du DSL ;
- Amélioration de la qualité : la qualité se trouve améliorée également à deux niveaux : au niveau de la définition et au niveau applicatif. Au niveau de la définition, les DSL développés en utilisant des outils éprouvés seront plus efficaces, plus fiables et de meilleure qualité. Au niveau applicatif, les applications générées bénéficieront de la qualité, des standards et des bonnes pratiques implémentées au niveau des générateurs. Ceci signifie moins de bogues et de reprises, et donc une meilleure qualité et un meilleur respect des délais pour les projets.

Impacts négatifs

- Développement laborieux et onéreux : sans l'utilisation des outils, le développement des DSL demeure un processus complexe et laborieux ;
- Limitation des possibilités d'automatisation : en l'absence de générateurs d'applications, il serait impossible de générer des applications à partir des modèles du DSL. En conséquence, les modèles ne serviront qu'à des fins de documentation et ne peuvent être utilisés comme des artefacts de programmation.

Aspects touchés

- Syntaxe concrète : les outils de support offrent des éditeurs de symboles permettant la définition de la syntaxe concrète du DSL. Ces éditeurs supportent la création des symboles représentant les types d'objets (concepts) et les types de relations. Ces outils peuvent également offrir des facilités pour la génération/développement d'éditeurs graphiques supportant la syntaxe concrète définie ;
- Sémantique : les outils de supports offrent des mécanismes pour définir le mappage entre les concepts du DSL et les éléments du domaine de sémantique.

Critères d'évaluation

- Productivité : attribut portant sur la capacité du DSL à permettre aux utilisateurs de produire des modèles avec efficacité ;
- Satisfaction : attribut du DSL portant sur la capacité du DSL à satisfaire ses utilisateurs ;
- Facilité de test : attribut portant sur l'effort nécessaire pour valider le DSL après une modification ;
- Facilité d'exploitation : attribut portant sur l'effort fourni par l'utilisateur afin d'exploiter le DSL.

7.4.7 Expertise en développement de langages

Impacts positifs

- Meilleur contrôle de l'activité de développement de DSL ;

- Approche structurée et disciplinée ;
- Conception concise et homogène.

Impacts négatifs

- Approche hasardeuse et aventureuse.

Aspects touchés

- Syntaxe abstraite : les compétences en développement de langages rendent plus facile l'identification des abstractions du DSL (concepts et relations) qui sont à la fois intéressantes aux utilisateurs et pertinentes à la génération de code ;
- Syntaxe concrète : choisir le type de notation à adopter pour le DSL (ex. textuelle, graphique ou combinaison des deux.) est une décision importante à prendre lors de la conception de celui-ci. Le développeur doit posséder les compétences nécessaires lui permettant de choisir la notation la mieux adaptée pour le DSL en question. Les compétences en développement de langages aident à atteindre l'homogénéité entre les concepts, leurs types de relations et leur sémantique d'une part et la notation qui les représente d'autre part.

Critères d'évaluation

- Facilité d'exploitation : attribut du DSL portant sur l'effort fourni par l'utilisateur afin de l'exploiter et de contrôler son exploitation ;
- Facilité d'apprentissage : attribut du DSL portant sur l'effort fourni par l'utilisateur afin de l'apprendre ;
- Facilité de compréhension : attribut du DSL portant sur l'effort fourni par l'utilisateur afin de comprendre comment il peut être utilisé pour réaliser les différentes tâches de modélisation.

7.4.8 Soutien des parties prenantes

Impacts positifs

- Adoption du DSL : le soutien des parties prenantes, et surtout des experts de domaine et des utilisateurs du DSL, est indispensable pour l'implantation et l'adoption du DSL ;
- Collaboration : participation et implication des parties prenantes dans le processus de développement et d'évolution du DSL.

Impacts négatifs

- Résistance et rejet : un DSL ne parvenant pas à attirer l'attention et à susciter l'intérêt de ses utilisateurs est plus susceptible d'être rejeté.

Aspects touchés

- Tous

Critère d'évaluation

- Pouvoir d'attraction : attribut portant sur la capacité du DSL à être attractif pour l'utilisateur.

7.4.9 Processus de développement

Impacts positifs

- Meilleur contrôle : un processus permet de parer à l'improvisation et de bien maîtriser le cycle de développement des DSL ;
- Meilleure anticipation : les projets de développement de DSL seront plus faciles à gérer et à anticiper ;
- L'extrant du processus est plus prévisible ;
- Possibilité de planifier un processus d'amélioration pour le processus de développement des DSL ;

- Donner plus de confiance aux clients ;
- Améliorer la collaboration entre les parties prenantes impliquées dans le développement des DSL.

Impacts négatifs

- Improvisation : en l'absence d'une approche disciplinée pour développer les DSL c'est généralement l'improvisation qui prend le dessus ;
- Output imprévisible.

Aspects touchés

- Tous.

Critères d'évaluation

- Facilité de modification : attribut du DSL portant sur l'effort nécessaire à l'implémentation d'une modification ou à la correction d'une anomalie ;
- Facilité de test : attribut portant sur l'effort nécessaire pour valider le DSL après une modification.

7.5 Sommaire et synthèse

En raison de l'absence d'un modèle de qualité pour les DSL, nous avons développé une technique qui permet d'identifier des attributs de qualité à partir de facteurs de succès. Cette technique a été appliquée aux DSL pour identifier les attributs de qualité des DSL à partir de leurs facteurs de succès. Le Tableau 7.3 synthétise les résultats de cette mise en application.

Les attributs de qualité identifiés seront utilisés, dans le chapitre 9, pour évaluer les DSL définis en utilisant la méthode de définition des DSL décrite dans le chapitre 6.

Tableau 7.3 Facteurs de succès des DSL et leurs attributs de qualité

Facteur de succès	**Description**	**Aspects DSL touchés**	**Attributs de qualité**
Portée du domaine	Avoir une portée de domaine soigneusement définie.	Syntaxe abstraite	- Expressivité - Facilité d'exploitation - Facilité d'apprentissage - Facilité de compréhension - Facilité de test - Convenance
Connaissance du domaine	Avoir une bonne connaissance du domaine d'opération du DSL.	- Syntaxe abstraite - Syntaxe concrète	- Expressivité - Convenance
Niveau d'abstraction	Travailler au juste niveau d'abstraction.	Syntaxe abstraite	Expressivité
Notation	Utiliser des notations appropriées.	Syntaxe concrète	- Pouvoir d'attraction - Facilité d'exploitation
Infrastructure technique	Avoir une infrastructure technique qui facilite le développement des DSL.	Tous	- Facilité de modification - Facilité de réutilisation - Capacité d'exécution - Réutilisabilité
Outils de support	Utiliser des outils de supports qui facilitent la définition et l'édition des DSL.	- Syntaxe concrète - Sémantique	- Productivité - Satisfaction - Facilité de test - Facilité d'exploitation
Expertise en développement de langages	Avoir des compétences en développement de langages.	- Syntaxe abstraite - Syntaxe concrète	- Facilité d'exploitation - Facilité d'apprentissage - Facilité de compréhension
Soutien des parties prenantes	Avoir le soutien des parties prenantes et en particulier des experts de domaines.	Tous	Pouvoir d'attraction
Processus de développement	Avoir un processus bien défini pour le développement des DSL	Tous	- Facilité de modification - Facilité de test

Le Tableau 7.4 énumère les attributs de qualité identifiés.

Tableau 7.4 Attributs de qualité des DSL

Attribut de qualité	Définition
Expressivité	Capacité du DSL à offrir des concepts expressifs et d'un fort niveau d'abstraction.
Convenance	Capacité du DSL à fournir un ensemble approprié de fonctions répondant aux besoins des utilisateurs.
Facilité d'exploitation	Attribut du DSL portant sur l'effort fourni par l'utilisateur afin de l'exploiter et de contrôler son exploitation.
Facilité de modification	Attribut du DSL portant sur l'effort nécessaire à l'implémentation d'une modification ou à la correction d'une anomalie.
Réutilisabilité	Capacité du DSL à être réutilisé, dans le même domaine, sans modification ou avec des modifications mineures.
Facilité de réutilisation	Attribut du DSL portant sur l'effort nécessaire pour son utilisation dans la définition de nouveaux DSL.
Satisfaction	Attribut du DSL portant sur la capacité du DSL à satisfaire ses utilisateurs.
Pouvoir d'attraction	Attribut portant sur la capacité du DSL à être attractif pour ses utilisateurs.
Productivité	Attribut portant sur la capacité du DSL à permettre aux utilisateurs de produire des modèles avec efficacité.
Capacité d'exécution	Attribut du DSL portant sur sa capacité à générer des applications à partir de ses modèles.
Facilité de compréhension	Attribut du DSL portant sur l'effort fourni par l'utilisateur afin de comprendre comment le DSL peut être utilisé pour réaliser les différentes tâches de programmation ou de modélisation.
Facilité d'apprentissage	Attribut du DSL portant sur l'effort fourni par l'utilisateur afin de l'apprendre.
Facilité de test	Attribut du DSL portant sur l'effort nécessaire pour le valider après une modification.

CHAPITRE 8

ÉTUDE DE CAS

Afin de s'assurer de l'applicabilité de la méthode qu'on propose pour la définition des DSL, nous avons conduit une expérimentation en adoptant la méthode d'étude de cas, une méthode de recherche qualitative particulièrement utile pour tester l'applicabilité de modèles théoriques dans la pratique (Eisenhardt, 1989). Il s'agit dans notre cas de tester l'utilisation de la méthode proposée dans le chapitre 6 dans la définition de langages dédiés pour un domaine de l'industrie logicielle.

Le présent chapitre présente le cas étudié et illustre l'utilisation de la méthode pour développer des langages dédiés pour le domaine étudié.

8.1 Choix du cas étudié

Le cas étudié dans ce chapitre a été choisi selon deux critères principaux. Le premier critère concerne l'utilité du cas. À cet égard, nous avons veillé à ce que le cas soit réaliste et utile dans la pratique. Ainsi, il était important pour nous de choisir un cas qui reflète les besoins réels et pratiques d'un domaine de l'industrie du logiciel. Le deuxième critère concerne la capacité du cas à présenter des situations variées permettant de tester la méthode dans différents contextes. En effet, l'objectif est de pouvoir tester l'applicabilité et l'efficacité de la méthode pour définir différents types de DSL, soit des DSL de programmation, des DSL de modélisation et des DSL de spécification.

Le cas étudié dans cette thèse est celui de la conception et développement de sites Web. En plus de sa conformité aux critères cités ci-haut, le domaine du Web présente l'avantage d'être un domaine commun dont les concepts sont facilement assimilés par la plupart des lecteurs.

8.2 Énoncé du domaine

Le domaine d'expérimentation choisi pour cette étude de cas, pour l'énoncer clairement, est celui des sites Web dynamiques multilingues reposant sur des bases de données et exposant deux interfaces d'interaction : une interface guichet (*front office*) accessible au public et une interface arrière-guichet (*back office*) destinée aux administrateurs autorisés à effectuer des tâches d'administration et de gestion de contenu sur ces sites.

Le domaine d'étude est défini, dans ce travail, en tant qu'ensemble de systèmes (voir section 2.2) : soit l'ensemble des sites Web partageant les caractéristiques citées ci-haut. En pratique, cela revient à définir une famille de sites Web qui va caractériser ce domaine. Ainsi, en premier lieu, une description générale de la famille est présentée. Ensuite, une analyse de domaine est effectuée afin d'identifier les besoins en termes de DSL. Enfin, la méthode de définition des DSL est utilisée pour créer les DSL nécessaires.

À noter que les trois DSL définis dans le cadre de cette étude de cas sont déjà implémentés et utilisés dans la pratique. Ainsi, dans cette étude de cas, nous allons utiliser notre méthode pour faire une redéfinition de ces DSL.

8.3 Exigences générales de la famille de produits

Le but est de construire une ligne de produits, basée sur les DSL, capable de produire la famille des sites dynamiques mentionnée précédemment. Les membres de cette famille seront des sites dynamiques ayant tous l'architecture générale déjà mentionnée mais avec des variations au niveau des caractéristiques et des fonctionnalités offertes par chacun. Voici une brève description des principales exigences à respecter pour la ligne de produits :

La ligne de produits doit permettre la génération des sites Web en respectant les contraintes suivantes :

- Générer un site selon les paramètres de configuration spécifiés par le concepteur du site (e.x. thème, langues, types de menu, etc.) et, surtout, sans connaitre à l'avance la structure de la base de données ;

- Générer la base de données soutenant le site ;
- Offrir la possibilité de choisir les modules (optionnels) à inclure dans chaque site (ex. magasinage en ligne, panier d'achat virtuel, forum, pavé publicitaire, diffusion par courrier électronique, statistiques, etc.) ;
- Générer une version pleinement fonctionnelle pour la partie arrière-guichet ;
- Générer une version de base prête à être personnalisée, selon les spécifications du client, pour la partie guichet ;
- Permettre une gestion personnalisée des pages Web à générer pour les différentes tables de la base de données. Le concepteur du site Web peut choisir une ou plusieurs pages parmi les pages suivantes : création, modification, fiche, liste, recherche rapide et recherche détaillée ;
- Permettre une gestion personnalisée du contenu à mettre dans chacune de ces pages. Ainsi, le concepteur du site doit être en mesure de spécifier l'information à inclure dans chaque page.

8.4 Analyse de domaine

Cette section présente l'Analyse de domaine qui fait le sujet de notre étude de cas. L'analyse consiste essentiellement à analyser la famille des sites Web dynamiques en examinant les points communs et les variations apparentes entre l'ensemble des membres de cette famille.

L'analyse du domaine est réalisée en suivant le processus FAST (*Family-Oriented Abstraction, Specification, and Translation*) (Weiss et Lai, 1999) qui semble convenir à nos besoins. Ce processus permet de définir des familles de produits et de développer des environnements permettant la production des membres de ces familles. Les activités du processus FAST consistent essentiellement en l'identification des abstractions, en la création du (ou des) DSL à utiliser pour décrire ces abstractions, et finalement en la transformation de ces descriptions en produits logiciels. FAST se caractérise également par l'intégration de l'analyse des points communs et des variations dans son processus (Weiss, 1998). Cette analyse est particulièrement utile pour éclairer la portée de la famille et pour décrire comment les membres de cette famille peuvent varier. À l'issue de cette activité on devrait

avoir, entre autres, la terminologie, les points communs, et les variations (Weiss et Ardis, 1997).

Le reste de cette section présente l'Analyse de domaine des sites Web dynamiques, soit la description de la terminologie utilisée dans ce domaine, la modélisation des abstractions, l'architecture de la ligne de produits et l'analyse des points communs et des variations.

8.4.1 Terminologie du domaine

L'importance de la terminologie vient du fait qu'elle permet de rendre la communication entre les parties prenantes plus facile et plus précise et qu'elle constitue une bonne source d'abstractions (Weiss, 1998). Le Tableau 8.1 énumère les termes les plus fréquemment utilisés dans le domaine.

Tableau 8.1 Liste des termes les plus fréquemment utilisés dans le domaine

Terme	**Sémantique**
Administrateur	Utilisateur ayant accès à toutes les fonctionnalités du système sans aucune restriction
Administration visuelle	Fonctionnalité permettant de gérer le contenu du site à partir de l'interface guichet
Alertes	Alertes envoyées aux utilisateurs
Back-office	Interface arrière-guichet servant à la gestion et à l'administration du site
Base de données	Base de données du site
Champ	Champ d'une table de la base de données
Cron	Un script utilisé pour exécuter des tâches pré-planifiées
Données brutes	Données telle qu'elles sont stockées dans la base de données
Données formatées	Données tirées de la base de données puis formatées selon leurs types (lien HTML, Adresse email, image, etc.)
Droit d'accès	Droit d'accès d'un utilisateur aux ressources du site
Environnement local	Environnement de développement d'un site Web
Environnement de production	Environnement opérationnel normal du site
Environnement de validation	Environnement de test ayant la même configuration que celui de la production
Export	Export des données de la base de données dans d'autres formats (e.x. CSV, XML, Excel. etc.)

Terme	Sémantique
Fiche	Page Web descriptive non éditable d'une table
Procédure de maintenance	Programme utilitaire ajouté pour répondre à un besoin spécifique d'un membre donné de la famille
Front-office	Interface guichet accessible au public
Identifiant	Surnom permettant d'identifier un utilisateur sur le site
Identification	Processus par lequel le système reconnaît un utilisateur valide
Import	Import de fichiers CSV, XML ou Excel dans une table de la base de données
Liaison	Liaison entre deux champs appartenant à deux tables différentes
Log	Journal des connexions
Schéma	Définition de la base de données du site
Module	Ensemble de scripts caractérisé par des responsabilités bien ciblées et fortement liées
Page liste	Page Web affichant un nombre prédéfini d'enregistrements sous forme d'une liste
Paramétrage	Configuration des paramètres du site
Paramètres	Un attribut qui sert à la configuration du site
Dossier de partage	Dossier contenant des ressources partagées par deux ou plusieurs tables (fichiers, images, vidéos, etc.)
Profil	un ensemble de droits qui caractérise une catégorie d'utilisateurs
Recherche détaillée	Fonctionnalité permettant de faire des recherches avancées sur une table de la base de données
Recherche rapide	Fonctionnalité permettant de faire des recherches simples sur une table de la base de données
Statistiques	Ensemble d'informations sur certaines activités effectuées sur le site (comportement des utilisateurs, pages les plus visitées, etc.)
Table contenu	Table gérant le contenu du site
Table système	Table utilitaire contenant des données spécifiques nécessaires à la gestion du site (e.x. statistiques, droits des utilisateurs, menu, journal, contenu divers, etc.)
Thème	Thème du site
Contenu divers	Contenu dynamique du site qui n'est pas géré par les tables contenus
"*Customization*"	Activité de personnalisation de la partie guichet selon les spécifications du client

8.4.2 Analyse des points communs et des variations

Les points communs désignent les exigences, les caractéristiques et les hypothèses qui sont vraies pour tous les membres de la famille. Ces points communs sont très utiles pour le

développement des actifs fondamentaux réutilisables par tous les membres. D'habitude, les points communs et les variations sont présentés avec des modèles de caractéristiques (*Feature Models*) (Lee, Kang et Lee, 2002). Ces modèles utilisent des diagrammes schématisant la représentation des relations structurelles et conceptuelles entre les variations. Or, pour des raisons de simplicité nous allons nous contenter d'une description textuelle des points communs et des variations.

Le Tableau 8.2 énumère les points communs entre les membres de la famille.

Tableau 8.2 Points communs entre les membres de la famille

Point commun	**Description**
Installation	Création ou mise à jour de la base de données
Gestion des utilisateurs et des droits d'accès	Création, modification ou suppression des utilisateurs et des droits d'accès
Journal des connexions	Journal des connexions des utilisateurs
Paramètres du site	Gestion dynamiques des paramètres généraux du site
Paramètre des tables	Gestion dynamique des paramètres des tables
Procédure de maintenance	Système de support pour les procédures de maintenance
Informations sur les tables	Vue de la structure de la base de données
Rechercher / remplacer	Recherche et remplace un mot ou une phrase dans la base de données
Administration visuelle	Gestion du contenu du site à partir de l'interface guichet

Variations

Alors que l'analyse des points communs est utilisée pour identifier les éléments réutilisables à travers la ligne de produits, l'analyse des variations est plutôt utilisée pour modéliser les particularités des membres de la famille. Le succès de cette activité dépend grandement de la capacité à percevoir et à anticiper les membres qui seront gérés par la famille (Weiss, 1998). Cette capacité vient généralement après avoir développé un nombre assez important de membres en utilisant les approches conventionnelles de développement logiciel.

Dans le cadre du présent cas d'étude, on distingue trois types de variations : optionnelle, alternative et libre. Un point de variation optionnel correspond à une fonctionnalité

facultative pour les sites (certains sites peuvent l'avoir, d'autres non). Un point de variation alternatif correspond à une opportunité de choisir 0, 1, 0..1, 0..N ou 1..N[14] options parmi un ensemble d'alternatives possibles. Finalement, un point de variation libre correspond à une variation dont le paramètre de variation peut prendre n'importe quelle valeur alphanumérique.

Paramètres de variations

Les paramètres de variations servent à quantifier les points de variations (Harsu, 2002). Le Tableau 8.3 énumère les paramètres de variations pour les membres de la famille.

Tableau 8.3 Paramètres de variations pour les membres de la famille

Paramètres	**Type**	**Valeur**	**Dépendance**
Alertes	Optionnel	Oui-Non	
Boutique	Optionnel	Oui-Non	
Captcha	Optionnel	Oui-Non	
Compte	Optionnel	Oui-Non	Identification
Contributions	Optionnel	Oui-Non	Compte
Export	Optionnel	Oui-Non	
Identification	Optionnel	Oui-Non	Compte
Import	Optionnel	Oui-Non	
Panier	Optionnel	Oui-Non	Boutique
Publicité	Optionnel	Oui-Non	
Rapport publicité	Optionnel	Oui-Non	Publicité
Sondage	Optionnel	Oui-Non	
Thème	Alternatif (0..1)	Classique, Professionnel	
Editeur HTML	Alternatif (1)	FCKeditor, HTML Area	
Langues	Alternatif (1..N)	Fr, En, Es, De, It, etc.	
Enregistrements par page-liste	Libre	multiples de 10 Min : 10 Max : 100	

[14] Notation utilisée pour représenter la multiplicité (cardinalité) dans la modélisation des données, N étant le nombre d'alternatives offertes.

Paramètres	Type	Valeur	Dépendance
Email Administrateur	Libre	Adresse courriel administrateur	
Nom BD	Libre	Nom de la base de données	
IP serveur BD	Libre	Adresse IP du serveur de base données	
Mot de passe Install	Libre	Mot de passe d'installation	
Table Visualisable sur l'interface guichet	Optionnel	Oui-Non	
Table Modifiable sur l'interface guichet	Optionnel	Oui-Non	
Table présente dans le menu	Optionnel	Oui-Non	
Listes de champs signifiants	Alternatif (0..N)	Champs signifiants à utiliser pour les titres, pour le tri, etc.	
Type Champ	Alternatif (1)	Type de données du champ	
Champ Obligatoire	Optionnel	Oui-Non	
Champ présent dans la page création	Optionnel	Oui-Non	
Champ présent dans la page modification	Optionnel	Oui-Non	
Champ présent dans la page recherche	Optionnel	Oui-Non	
Champ présent dans la page liste	Optionnel	Oui-Non	
Champ présent dans la page fiche	Optionnel	Oui-Non	

8.4.3 Abstractions du domaine

Les abstractions du domaine jouent un rôle central dans le processus FAST. Elles sont utilisées dans la conception des DSL, dans la conception des familles de produits et dans la création des actifs fondamentaux (Harsu, 2002). Les abstractions définies dans cette section sont destinées principalement à la conception des DSL. Autrement dit, nous n'avons retenu que les abstractions présentant un intérêt pour des DSL (consulter la section 2.6.3.1 pour des lignes directrices aidant à identifier des abstractions utiles).

La Figure 8.1 montre une représentation graphique du modèle d'abstractions.

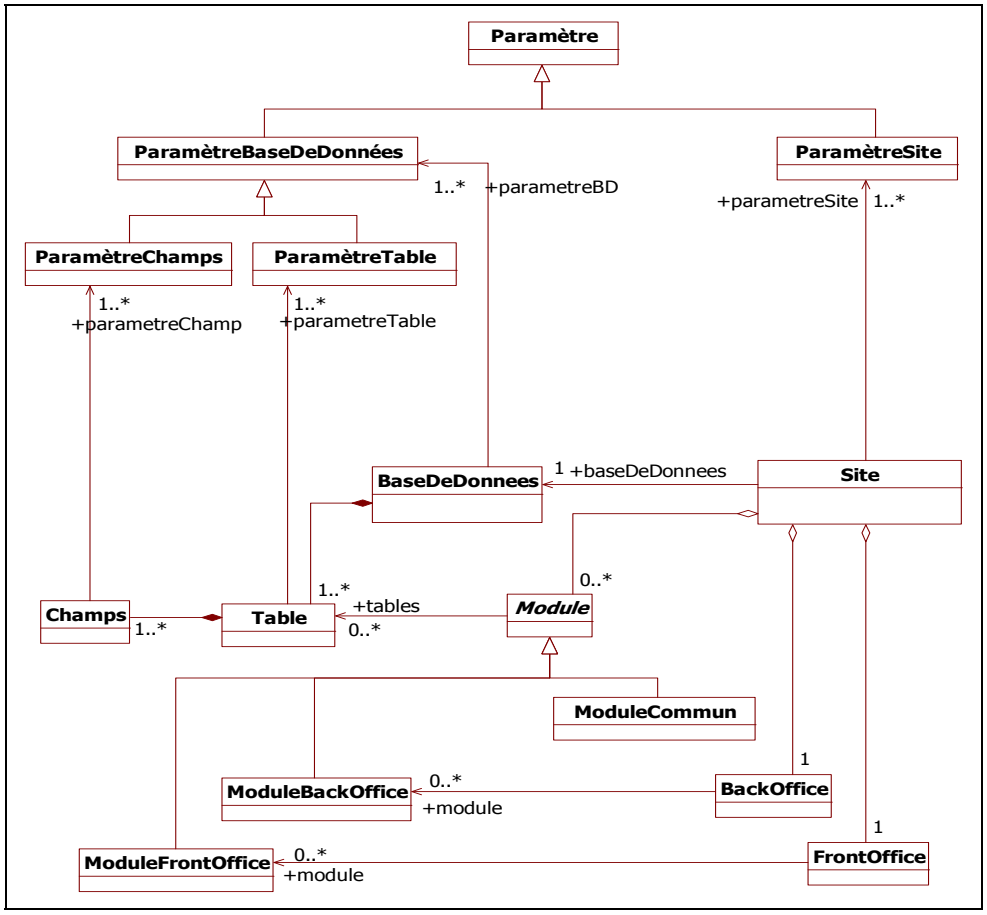

Figure 8.1 Modèle des abstractions de la ligne de produits.

Voici une description sommaire des abstractions définies :

Paramètre : classe qui représente une directive de configuration permettant de spécifier une variante.

Paramètre-Site : sous-classe de la classe *Paramètre* qui représente les directives de configurations relatives à l'aspect général du site.

Paramètre-Base-Données : sous-classe de la classe *Paramètre* qui représente une directive de configuration de la base de données.

Paramètre-Table : sous-classe de la classe *Paramètre-Base-Données* qui représente une directive de configuration d'une table de la base de données.

Paramètre-Champs : sous-classe de la classe *Paramètre-Base-Données* qui représente une directive de configuration d'un champ de table dans la base de données.

Base-Données : classe qui représente une collection de données, typiquement une base de données relationnelle.

Site : classe qui représente un site Web.

Table : classe qui représente une collection d'unités de données partageant des caractéristiques communes.

Champ : classe qui représente une unité élémentaire de données dans une classe *Table*.

FrontOffice : classe qui représente l'interface guichet d'un site.

BackOffice : classe qui représente l'interface arrière-guichet d'un site.

Module : classe qui représente un composant offrant un ensemble cohésif de fonctionnalités.

Module-Commun : classe qui représente un module commun au front-office et au back-office.

Module-BackOffice : classe qui représente un module de l'interface arrière-guichet.

Module-FrontOffice : classe qui représente un module de l'interface guichet.

8.4.4 Architecture de la ligne de produits

Le but de cette étape est de concevoir une architecture générale pour la ligne de produits qui montre l'ensemble des outils (langages, générateurs et compilateurs) à mettre en place pour produire les différents membres de la famille (voir Figure 8.2).

À ce niveau se posent deux questions importantes : 1) comment sont déterminés les DSL nécessaires pour combler les besoins de la famille et 2) comment sont établies les exigences de ces DSL à partir des exigences de cette famille ?

Quoique le développement des langages dédiés soit considéré comme une activité importante dans le processus FAST, ce dernier ne fournit pas de démarche détaillée pour l'identification ni pour la définition de ces langages. Dans la littérature, nous n'avons pas trouvé de méthode systématique permettant d'identifier de manière automatique les DSL nécessaires à l'implémentation d'une ligne de produits logiciels. Cela peut s'expliquer par le fait que l'identification et le développement de DSL vient tard dans le processus d'ingénierie d'une famille de produits logiciels. D'habitude, l'élaboration d'une famille de produits commence par le développement d'un nombre suffisants de membres. Durant ces développements sont planifiées des activités de ré-usinage de code, de construction de composants logiciels et de développement de bibliothèques de programmes. Le développement des DSL vient couronner les efforts de ces activités d'ingénierie de domaine et améliorer davantage les capacités de développement, et ce en offrant une façon plus naturelle, aux utilisateurs et aux experts de domaine, de développer des applications.

Nous avons identifié deux types de DSL. Nous les avons nommés « DSL primaires » et « DSL dérivés ».

DSL primaires : généralement, ces DSL sont utilisés soit comme des interfaces d'accès aux fonctionnalités des actifs fondamentaux, soit comme des langages de configuration permettant de spécifier les particularités des différents membres de la famille (Coplien, Hoffman et Weiss, 1998). Ce type de DSL est identifié généralement à partir des caractéristiques et des exigences de la famille, des interfaces des composants, des modules et des bibliothèques de programmes.

DSL dérivés : parfois la solution choisie pour implémenter la ligne de produits impose le développement de nouveaux DSL dits utilitaires (*Helpful DSL*). Le langage MSL (voir section 8.7) est un exemple de ce type DSL.

Tel qu'illustré dans la Figure 8.2, le processus de développement dans cette ligne de produits consiste en trois activités principales : le paramétrage du site, la modélisation de la base de données et le développement des modèles (*templates*). Chacune des ces activités est réalisée en utilisant un DSL qui lui est propre. Il n'y a pas de contraintes particulières sur l'ordre d'exécution de ces activités ; elles peuvent être exécutées en parallèle ou en séquence.

Paramétrage du site

La configuration des paramètres du site est réalisée à l'aide du langage dédié WCL (***W**ebsite **C**onfiguration **L**anguage*). Ce dernier sert à définir, entre autres, les paramètres généraux du site, les paramètres d'accès à la base de données, les modules à inclure, les thèmes à appliquer et les langes du site Web. Le paramétrage est sauvegardé dans un fichier de configuration au format INI pour un usage ultérieur par le générateur[15].

Modélisation de la base de données

La modélisation de la base de données est réalisée à l'aide du langage DDL (***D**atabase **D**esign **L**anguage*). Ce dernier sert essentiellement à modéliser la structure de la base de données et à spécifier les caractéristiques des tables et des champs correspondants. Les informations de la définition de la base de données sont sauvegardées dans le même fichier de configuration.

Développement des modèles

Le processus de développement utilise la technique des modèles (*Templates*) comme mécanisme de réutilisation de code. Ces modèles sont écrits dans le langage dédié MSL (***M**eta **S**cripting **L**anguage*) qui est conçu pour être intégré à n'importe quel autre langage ou script de programmation (ex. HTML, PHP, ASP, JSP, etc.). Les variables définies dans les

[15] Cela nécessite le développement d'un générateur capable d'interpréter les modèles WCL et générer un fichier de configuration au format INI.

modèles sont substituées, au moment de la génération, par les variantes effectives du site en construction[16].

La suite du chapitre se concentre sur la description détaillée et le cycle de définition (redéfinition) des trois DSL déjà cités. La description des caractéristiques des générateurs et de leur mode de fonctionnement dépasse la cadre de cette thèse.

[16] Cela nécessite le développement d'un interpréteur capable d'aller lire dans le fichier de configuration les valeurs qui y ont été placées et de transformer le code générique MSL en code spécifique (HTML, PHP, JSP, etc.).

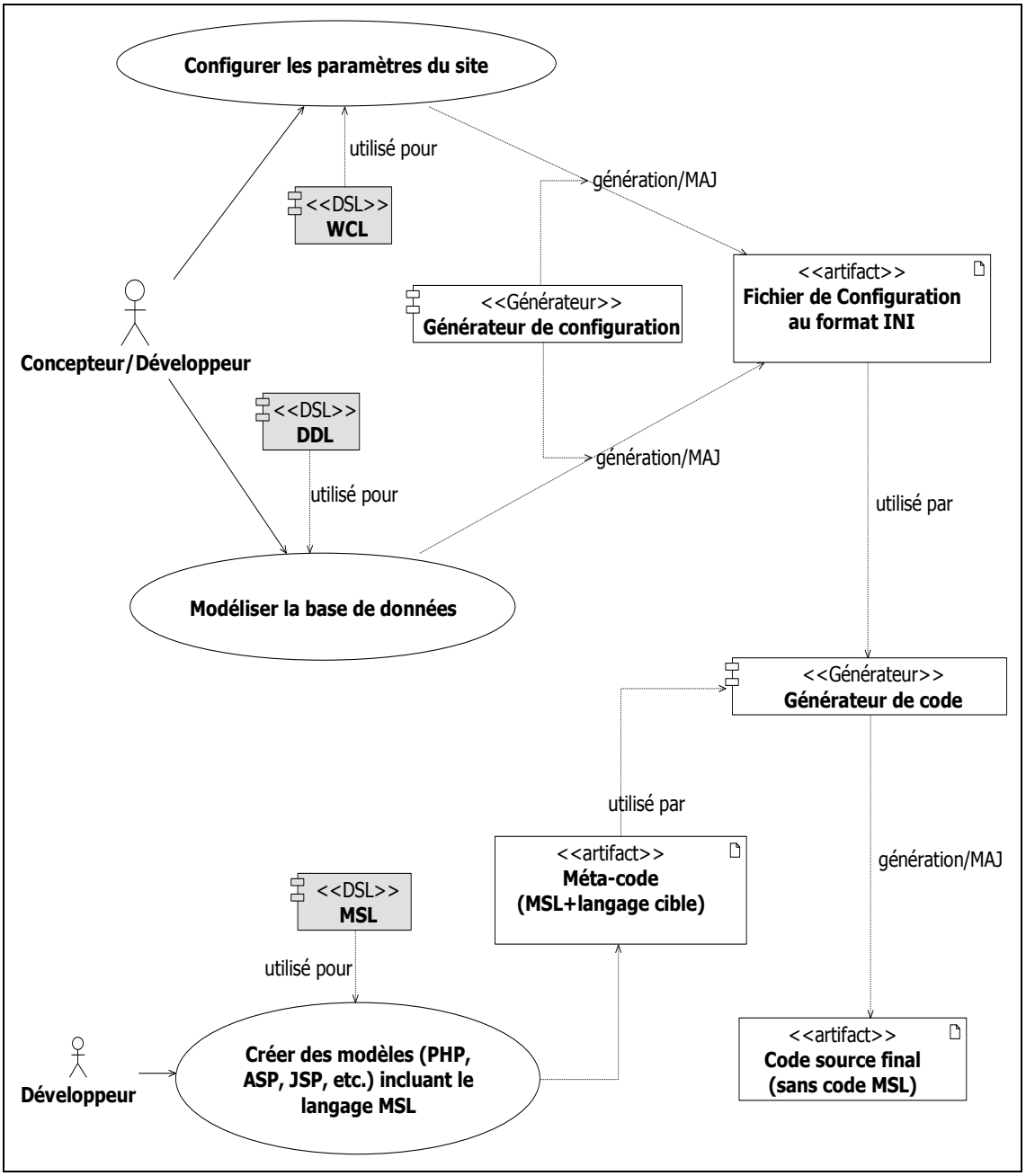

Figure 8.2 Processus de production des sites dynamiques.

8.5 Langage WCL

8.5.1 Besoins

Spécifier le paramétrage général du site, notamment les langues du site, les paramètres d'accès à la base de données, les modules à inclure ainsi que les thèmes à appliquer.

8.5.2 Vision

8.5.2.1 But

Offrir une interface permettant de définir les valeurs des différents paramètres de configuration d'un site.

8.5.2.2 Spécifications des caractéristiques

Le Tableau 8.4 résume les caractéristiques principales du langage WCL.

Tableau 8.4 Caractéristiques du langage WCL

Code	Contexte	Description
C1		Paramétrer le site Web
C1.1	C1	Spécifier les modules à inclure ou à exclure du site
C1.2	C1	Spécifier les paramètres généraux du site, notamment le thème, les langues, l'adresse du serveur web, les emails, etc.
C1.3	C1	Spécifier les paramètres d'accès à la base de données

8.5.2.3 Utilisateurs

Les utilisateurs potentiels du langage WCL sont les concepteurs et les développeurs de sites Web.

8.5.3 Syntaxe abstraite

8.5.3.1 Identification des concepts

Compte tenu des spécifications précitées pour le langage WCL, quatre concepts candidats peuvent être considérés pertinents à la configuration du site : Section, paramètre, entrée et valeurs.

Paramètre : représente une paire (entrée, valeur) qui définit une variation relative à la configuration d'un site.

Paramètre Site : représente un paramètre général du site.

Paramètre de base de données : représente un paramètre relatif à la configuration de la base de données.

Module : représente un module à inclure ou à exclure du site Web.

Entrée : représente le nom du paramètre de configuration.

Valeur : représente la valeur du paramètre de configuration.

8.5.3.2 Architecture

Le métamodèle du langage WCL est relativement simple. Ainsi, d'un point de vue architectural, ce langage consiste en un seul paquetage contenant tous les éléments du modèle de la syntaxe abstraite (voir Figure 8.3).

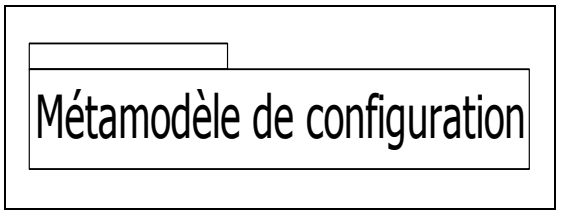

Figure 8.3 Architecture générale du langage WCL.

8.5.3.3 Métamodèle

Le métamodèle est composé des éléments suivants :

Élément : classe abstraite qui représente une superclasse commune à tous les éléments des différents modèles du langage WCL. Chaque *Élément* du langage est défini par les attributs suivants :

Nom (String) : nom de l'élément ;

Libellé (String) : libellé de l'élément ;

Description (String) : description de l'élément.

Paramètre : classe qui représente un paramètre de variation. Chaque *Paramètre* est défini par :

Entrée (String) : nom de la variation ;

Valeur (String) : valeur de la variation.

Section : représente un regroupement logique de paramètres de variation.

La Figure 3.4 montre le modèle de la syntaxe abstraite du langage WCL.

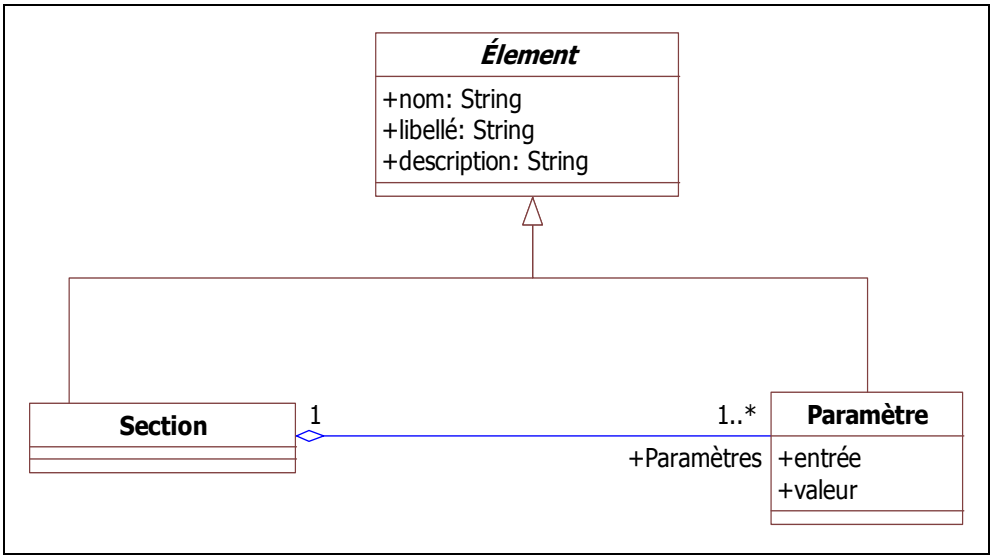

Figure 8.4 Modèle de la syntaxe abstraite du langage WCL.

8.5.3.4 Règles de grammaire

Un modèle de configuration est considéré valide s'il respecte les règles de grammaire suivantes :

- unicité des sections : deux sections ne peuvent avoir le même nom dans le même modèle ;
- deux entrées ne peuvent avoir le même nom dans la même section.

8.5.4 Syntaxe concrète

Comme déjà mentionné, le paramétrage fait avec le langage WCL est sauvegardé dans un fichier de configuration au format INI. Donc, la syntaxe concrète que nous avons adoptée pour le langage WCL est celle d'un fichier INI.

L'éditeur de ce langage consiste en deux formulaires qui permettent aux utilisateurs de saisir les informations relatives aux paramètres de variations. Le premier formulaire permet de définir les différentes sections (voir Figure 8.5). Le deuxième offre une interface de saisie pour spécifier les valeurs des paramètres (voir Figure 8.6).

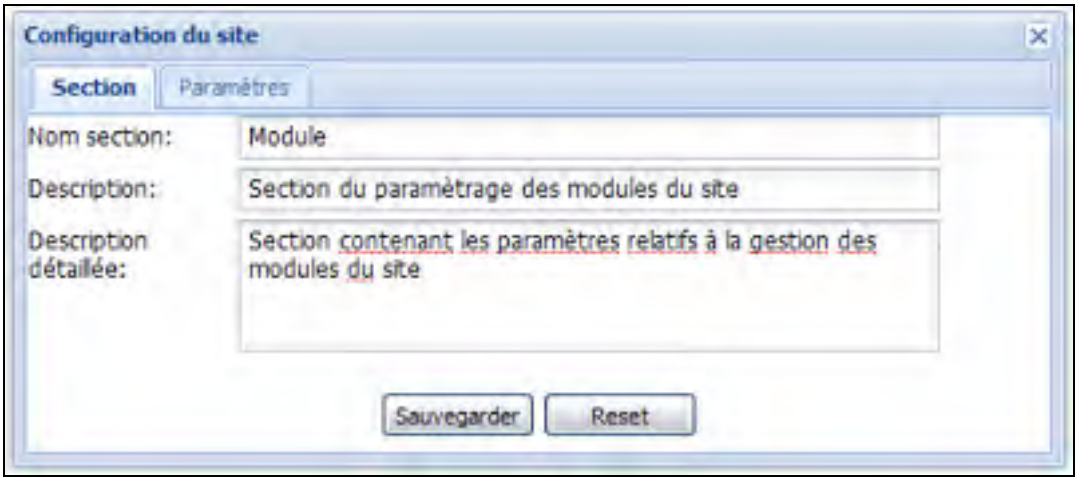

Figure 8.5 Interface de gestion des sections.

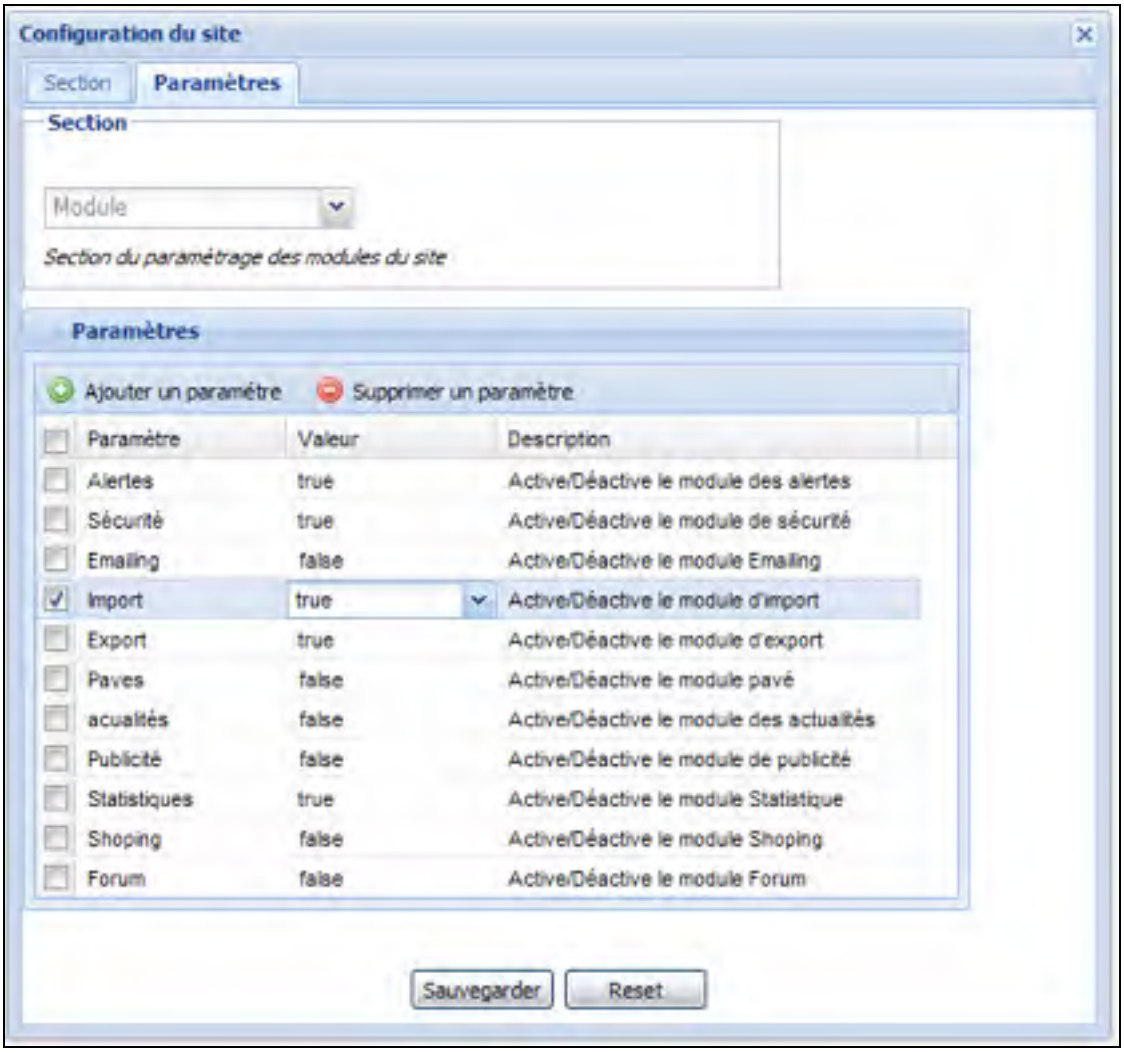

Figure 8.6 Interface de gestion des paramètres.

8.5.5 Sémantique

Le domaine sémantique ($\boldsymbol{S_D}$) du langage *WCL* consiste en l'ensemble des éléments constituant un fichier INI, soit : section (*S*), paramètre (*P*), valeur (*V*) et commentaire (*C*).

$$\boldsymbol{S_D} = \{S, P, V, C\} \tag{8.1}$$

Le mappage sémantique établissant le lien entre les expressions définies dans le domaine syntaxique du langage *WCL* et les éléments correspondants dans le domaine sémantique se présente comme suit :

$$Map_{<section>} : \text{Section} \rightarrow S$$
$$Map_{<paramètre>} : \text{Paramètre} \rightarrow P$$
$$Map_{<valeur>} : \text{Valeur} \rightarrow V$$
$$Map_{<description>} : \text{Description} \rightarrow C$$

8.6 Langage DDL

DDL est un langage dédié à la conception des bases de données relationnelles. En plus de la conception habituelle des bases de données qui consiste en la définition des tables, des champs et des relations, DDL permet de spécifier également les caractéristiques Web, notamment, la présence de ces tables et de leurs champs sur le site Web.

8.6.1 Besoins

Être capable de créer des modèles de données orientées Web.

8.6.2 Vision

8.6.2.1 But

Créer un langage permettant la modélisation de la structure d'une base de données ainsi que la spécification de ses caractéristiques Web.

8.6.2.2 Spécifications des caractéristique

Le Tableau 8.5 résume les caractéristiques principales du langage DDL

Tableau 8.5 Caractéristiques du langage DDL

Code	Contexte	Description
C1		Définir les tables de la base de données
C1.1	C1	Définition générale de table : nom, libellé et description
C1.2	C1	Définition des paramètres Web de la table : • Visualisation sur le site ; • Critère de tri par défaut: utilisé lors d'une recherche ; • Modification à partir du front-office ; • Présence dans le menu du site.
C1.3	C1	Définition des champs des tables
C1.3.1	C1.3	Définition générale d'un champ : • Nom : Nom du champ ; • Type de donnée : peut être primitive ou évolué (e.x. document, email, image, lien, flash, vidéo, etc.) ; • Taille du champ : l'interprétation de cette valeur est établie en fonction du type de données du champ ; • Clef : définit si le champ est la clé de la table ; • Obligatoire : définit si le champ est obligatoire ; • Valeur par défaut : spécifie la valeur par défaut du champ.
C1.3.2	C1.3	Définition des paramètres Web du champ : • Présence du champ dans les pages du site, notamment, les pages de modification-création, les fiches, les listes et les pages de recherche ; • Liste de choix : spécifie si la valeur du champ est à sélectionner parmi une ou plusieurs options ; • Type de la Liste de choix : liste déroulante, butons radios, case à cocher, etc.
C2		Définition des liaisons entre les tables
C2.1	C2	Définition générale de la liaison • Champs de la liaison : établir une relation entre la clé primaire de la table source et un autre champ jouant le rôle d'une clé étrangère dans la table destination ; • Action en cas de modification ou de suppression.
C2.2	C2	Définition des paramètres Web de la liaison • Afficher les liens sur les fiches ; • Afficher les enregistrements fils.

8.6.2.3 Utilisateurs

Les utilisateurs potentiels du langage DDL sont les concepteurs de bases de données et les concepteurs/développeurs Web.

8.6.3 Syntaxe abstraite

8.6.3.1 Identification des concepts

Compte tenu des spécifications stipulées dans la section 8.6.2.2, les concepts suivants peuvent être considérés pertinents à la modélisation des bases de données pour les sites Web :

Modèle : représente un modèle de données, soit, un ensemble d'éléments (tables, attributs, relations, etc.) décrivant la structure et les caractéristiques Web des données manipulées par un site.

Base de données : représente une collection structurée de données.

Table : représente un ensemble organisé de données en utilisant un format prédéfini de lignes (enregistrements) et de colonnes (attributs/champs).

Enregistrement : représente une rangée dans une table.

Attribut : représente une unité d'information dans un enregistrement. L'ensemble des données représentées par cette unité ont le même type de données.

Clé étrangère : représente un attribut utilisé dans une table pour référencier un attribut dans une autre table.

Clé primaire : représente un attribut permettant d'identifier de manière unique un enregistrement dans une table.

Type de donnée : représente le type de données qui peut être stocké dans un attribut

Relation : représente un lien entre deux tables dont l'une possède une clé étrangère qui fait référence à la clé primaire de l'autre.

Contrainte : représente une règle ou une restriction sur un élément du modèle.

8.6.3.2 Architecture

Le métamodèle décrivant la syntaxe abstraite du langage DDL est subdivisé en cinq sous-métamodèles (voir Figure 8.7). Les concepts décrits par ces métamodèles sont : *Modèle*, *Table*, *Attribut*, *Type de données* et *Relation*.

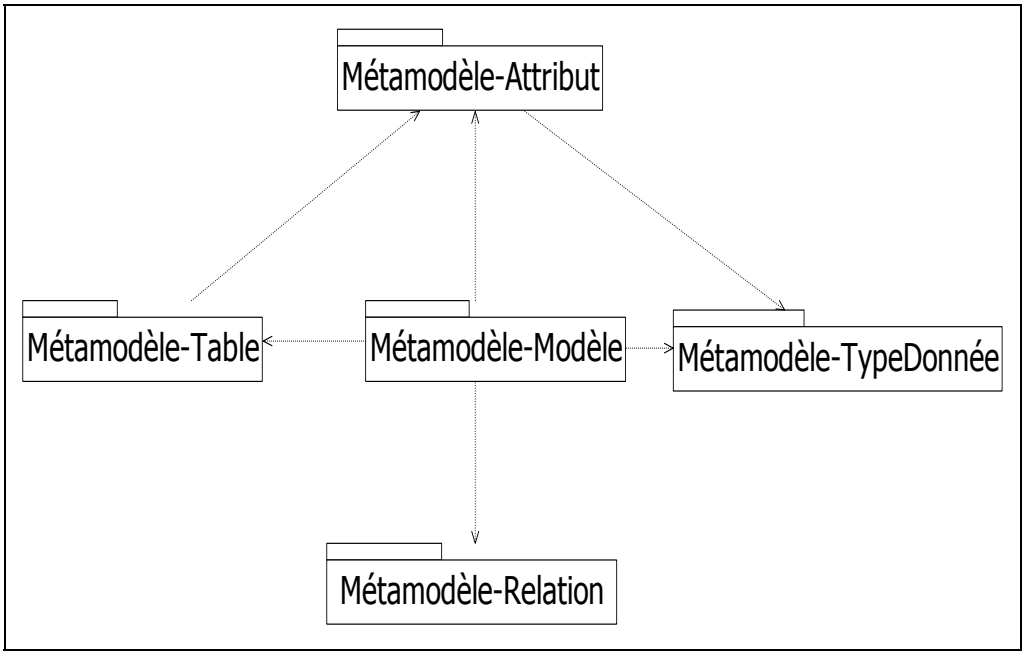

Figure 8.7 Architecture générale du langage DDL.

8.6.3.3 Métamodèle « *Modèle* »

Ce métamodèle définit les éléments composant un modèle DDL (voir Figure 8.8) :

Élément : classe abstraite qui représente une superclasse commune à tous les éléments d'un modèle. Les attributs définis par cette classe sont :

- *nom (String) :* représente le nom de l'élément ;
- *libellé (String) :* représente le libellé de l'élément ;
- *description (String) :* représente la description de l'élément.

Modèle : représente un ensemble d'éléments (tables, attributs, relations, etc.) décrivant la structure et les caractéristiques Web des données manipulées par un site.

Table : représente un ensemble organisé de données.

Attribut : représente une unité d'information dans une instance de la classe *Table*.

Type de donnée : classe abstraite qui représente une superclasse commune aux classes représentant les types de données caractérisant les instances de la classe *Attribut*.

Contrainte : classe abstraite qui représente une superclasse commune aux contraintes s'appliquant aux éléments d'un modèle. Cette classe définit un seul attribut :

- *expression (String) :* représente la spécification de la contrainte.

Modèle⇔Élément [1..*] [17] **:** spécifie les éléments qui composent un modèle.

Élément ⇔Contrainte [0..*] : spécifie les contraintes d'un élément

[17] Les associations entre les métaclasses sont représentées sous la forme : élément1[cardinalité1]⇔élément2[cardinalité2]

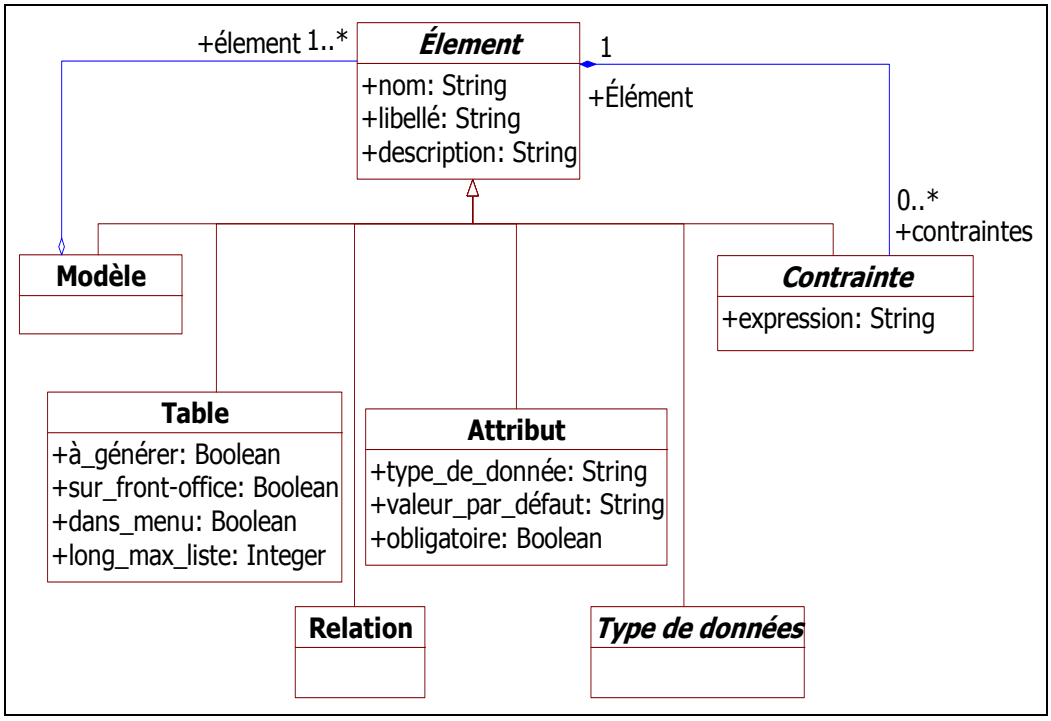

Figure 8.8 Métamodèle *Modèle* du langage DDL.

8.6.3.4 Métamodèle *Table*

Ce métamodèle définit les éléments utilisés pour la définition des tables (voir Figure 8.9). Il est constitué des éléments et des associations suivants :

Table : représente un ensemble d'entrées partageant la même structure de données et les mêmes caractéristiques Web. Les attributs définis par cette classe sont :

- *à_générer (Booléen)* : spécifie si la table doit être générée ;
- *sur_front-office (Booléen)* : spécifie si la table est visualisable sur le front-office ;
- *dans_menu (Booléen)* : spécifie si la table apparaît sur le menu principal du site ;
- *long_max_liste (Entier)* : spécifie le nombre maximum d'enregistrements affichables sur un page liste.

Contrainte Table : représente une contrainte qui s'applique aux instances de classe *Table.*

Table [1]⇔Attribut [1..*] : spécifie les attributs d'une table.

Table [1]⇔Clé [1] : spécifie la clé d'une table.

Relation [0..*]⇔Table [2] : spécifie les relations de la table.

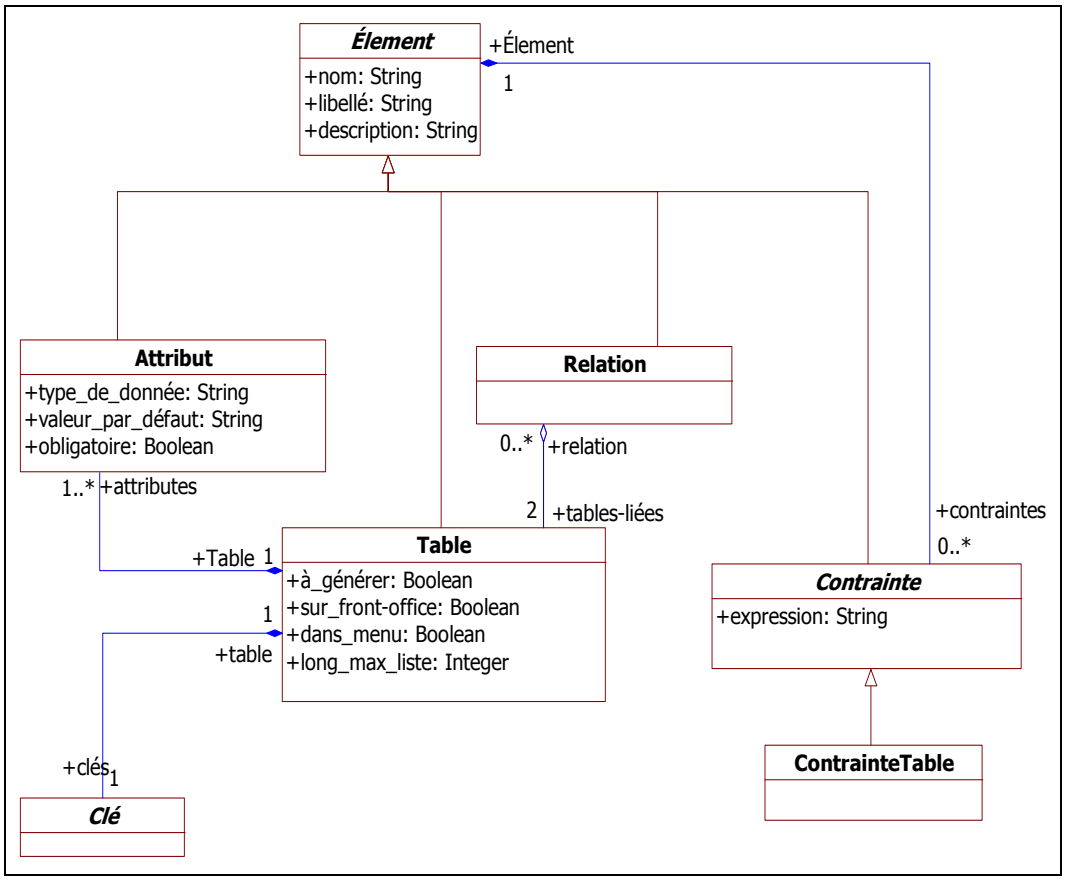

Figure 8.9 Métamodèle *Table* du langage DDL.

8.6.3.5 Métamodèle Attribut

Ce métamodèle définit les éléments utilisés dans la définition des attributs (voir Figure 8.10). Il est constitué des éléments et des associations suivants :

Attribut : représente une unité d'information dans une instance de la classe *Table*. L'ensemble des données représentées par cette unité ont le même type de données. Les attributs définis par cette classe sont :

- *valeur par défaut (String)* : spécifie la valeur par défaut de l'attribut ;
- *obligatoire (booléen)* : indique si cet attribut est requis. Un attribut obligatoire doit avoir une valeur pour chaque entrée dans une instance *Table*.

Clé : classe abstraite qui représente une superclasse commune aux attributs clés.

Clé_primaire: représente un attribut qui identifie d'une façon unique une instance de la classe *Table.*

Clé_étrangère: représente une clé étrangère.

Contrainte_attribut: représente une contrainte commune à toutes les instances de classe *Attribut***.**

Table [1]⇔Attribut [1..*] : spécifie la table contenant l'attribut.

Attribut⇔Clé [1] : spécifie si l'attribut est une clé.

Attribut⇔type de données [1] : spécifie le type de données de l'attribut.

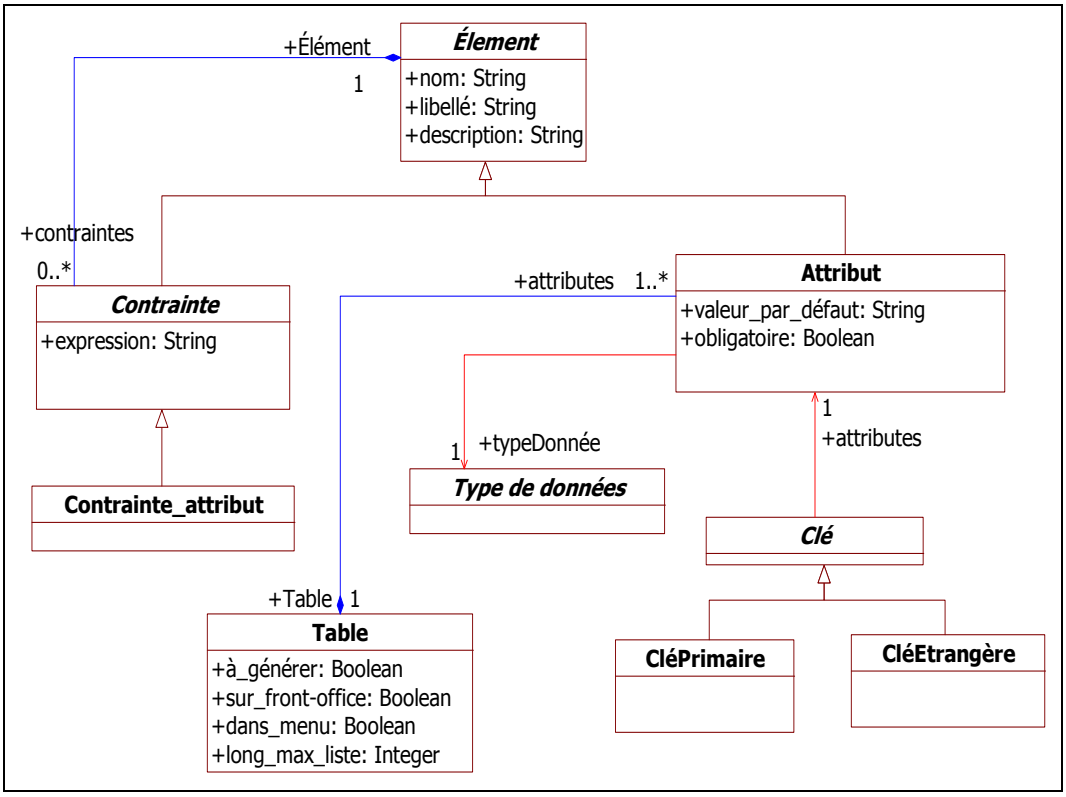

Figure 8.10 Métamodèle *Attribut* du langage DDL.

8.6.3.6 Métamodèle Relation

Ce métamodèle définit les éléments utilisés dans la définition des relations entre les tables (voir Figure 8.11). Il est constitué des éléments et des associations suivants :

Relation : représente un lien entre deux instances de la classe *Table*.

ContrainteRelation : représente une contrainte commune à toutes les instances de classe *Relation*.

Relation [0..*]⇔Table [2] : spécifie les deux tables participant à la relation.

Relation [1]⇔ContrainteRelation [1..*] : spécifie les contraintes d'une relation.

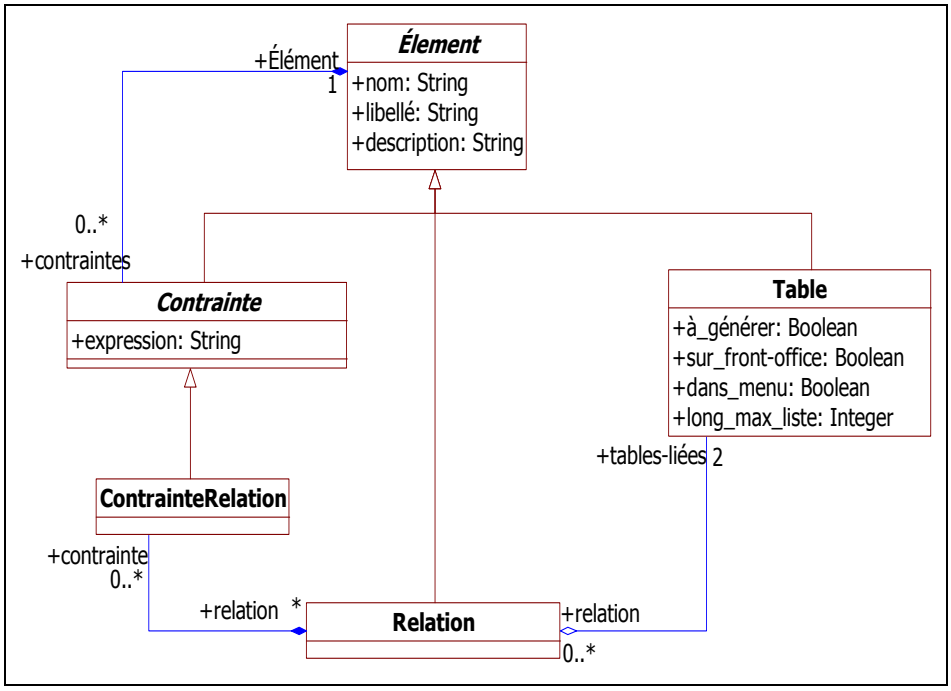

Figure 8.11 Métamodèle *Relation* du langage DDL.

8.6.3.7 Métamodèle « Type de donnée »

Ce métamodèle définit les éléments définissant les types de données utilisés dans le domaine du Web (voir Figure 8.12). Il est constitué des éléments et des associations suivants :

Type de donnée : classe abstraite qui représente une superclasse commune aux classes représentant les types de données caractérisant les instances de la classe *Attribut*.

TypeDonnéePrimitif : représente un type de données primitif.

TypeDonnéesWeb : représente un type de données orienté Web qui peut être utilisé comme un type de données pour les instances de la classe *Attribut*. Cette classe définit un seul attribut :

- *typeDeBase* (*String*) : type de base du type dérivé.

Document : représente un type de données dont les valeurs sont des références à des documents (ex. Pdf, Word, Excel, etc.).

Email : représente un type de données dont les valeurs sont des emails.

Lien : représente un type de données dont les valeurs sont des URL.

Flash : représente un type de données dont les valeurs sont des références à des fichiers Flash.

Video : représente un type de données dont les valeurs son des références à des documents Vidéo.

Image : représente un type de données dont les valeurs sont des images.

Commentaire_HTML : représente un type de données consacré au stockage de textes formatés en HTML.

Contrainte_TypeDonnéeDomaine : représente une contrainte commune à toutes les instances de classe *TypeDonnéesWeb*.

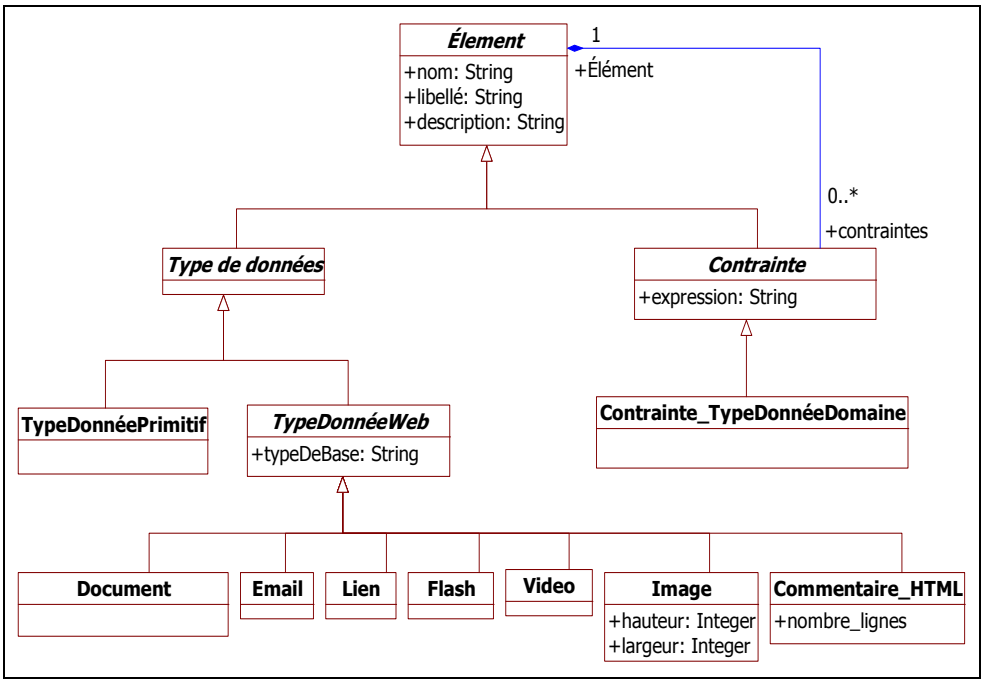

Figure 8.12 Métamodèle Type de données du langage DDL.

8.6.3.8 Règles de grammaire

Un modèle de données est considéré valide s'il respecte les règles de grammaire suivantes :

- Deux tables ne peuvent avoir le même nom dans un même modèle ;
- Une table contient au moins un attribut ;
- Deux attributs ne peuvent avoir le même nom dans la même table.

8.6.4 Syntaxe concrète

Le paramétrage fait avec le langage DDL est sauvegardé dans un fichier de configuration au format INI. Donc, la syntaxe concrète que nous avons adoptée pour ce langage est celle d'un fichier INI.

L'éditeur du langage DDL consiste en trois formulaires qui permettent aux utilisateurs de saisir les informations relatives à la définition des bases de données et de leurs caractéristiques Web. Le premier formulaire permet de définir les tables de la base de données (voir Figure 8.13). Le deuxième formulaire offre une interface de saisie pour la définition des champs d'une table (voir Figure 8.14). Finalement, le troisième formulaire permet de spécifier les relations entre les tables de la base de données (voir Figure 8.15).

Figure 8.13 Formulaire de définition d'une table.

Figure 8.14 Formulaire de définition des champs d'une table.

Figure 8.15 Formulaire de définition des relations entre les tables.

8.6.5 Sémantique

Le domaine sémantique ($\boldsymbol{S_D}$) du langage DDL consiste en l'ensemble des éléments constituant un fichier INI, soit : section (*S*), paramètre (*P*), valeur (*V*) et commentaire (*C*).

$$\boldsymbol{S_D} = \{S, P, V, C\} \tag{8.2}$$

Le mappage sémantique établissant le lien entre les expressions définies dans le domaine syntaxique du langage DDL et les éléments correspondants dans le domaine sémantique se présente comme suit :

$\forall M \in MetaClasses$ où $MetaClasses = \{Table, Attribut, Relation\}$

Le mappage sémantique est :

$$Map_{<M>} : M \rightarrow S$$

Et pour chaque attribut *AT* de *M :*

$$Map_{<AT>} : AT \rightarrow (P, V)$$

8.7 Langage MSL

8.7.1 Besoins

Construire un langage de programmation capable de générer du code personnalisé en fonction des paramètres de configuration. Ce langage est destiné à être encapsulé dans du HTML, PHP, ASP, JSP ou tout autre langage de programmation Web.

8.7.2 Vision

8.7.2.1 But

Offrir un langage de programmation capable d'aller lire dans le fichier de configuration les valeurs qui y ont été placées lors de la configuration du site (voir section 8.5) et de la modélisation de sa base de données (voir section 8.6), et d'écrire ces valeurs dans les pages générées.

8.7.2.2 Spécifications des caractéristiques

Le Tableau 8.6 liste les caractéristiques du langage MSL.

Tableau 8.6 Caractéristiques du langage MSL

Code	Contexte	Description
C1		Offrir des expressions permettant d'accéder à des valeurs
C1.1	C1	Offrir des expressions d'accès aux valeurs des variables
C1.2	C1	Offrir des expressions d'accès aux valeurs du fichier de configuration
C2		Offrir les instructions nécessaires à l'écriture de valeurs et au contrôle du flux d'exécution
C2.1	C2	Offrir une instruction permettant l'écriture de valeurs dans le fichier généré. Cette instruction peut se trouver en milieu de ligne comme elle peut se trouver plusieurs fois dans la même ligne.
C2.2	C2	Offrir une instruction permettant d'itérer sur une liste de valeurs et de répéter un bloc de code autant de fois qu'il y a d'éléments dans la liste
C2.4	C2	Offrir une instruction de branchement conditionnel permettant de générer, ou non, un bloc de code en fonction d'une condition

8.7.2.3 Utilisateurs

Les utilisateurs potentiels du langage MSL sont les développeurs Web.

8.7.3 Syntaxe abstraite

8.7.3.1 Identification des concepts

En se basant sur la spécification donnée dans la section 8.7.2.2, les concepts candidats suivants peuvent être immédiatement identifiés.

Boucle : ensemble d'instructions qui s'exécutent de façon répétitive jusqu'à ce que la condition d'arrêt soit vérifiée.

Branchement conditionnel : branchement qui n'est exécuté que si une condition ou un ensemble de conditions sont préalablement remplies.

Condition : expression logique dont le résultat est une valeur booléenne (Vrai ou Faux).

Déclaration : instruction qui sert à définir une variable.

Expression: expression qui s'évalue pour produire des valeurs MSL. À la différence des expressions des langages de programmation habituels, les expressions MSL ne permettent pratiquement aucun calcul. Elles sont surtout utilisées pour accéder aux valeurs des variables et au contenu du fichier de configuration.

Instruction : ligne de code qui sert à contrôler le flux d'exécution ou à mettre à jour des valeurs.

Liste : une suite de valeurs séparées par des virgules ou des points-virgules.

Paramètre : une entrée dans le fichier de configuration.

Valeur : un entier, une chaine de caractères ou une liste de chaines de caractères.

Variable : zone mémoire qui va contenir des données.

8.7.3.2 Architecture

Le métamodèle du langage MSL est subdivisé en trois sous-métamodèles représentant les trois concepts : valeur, expression et instruction (voir Figure 8.16).

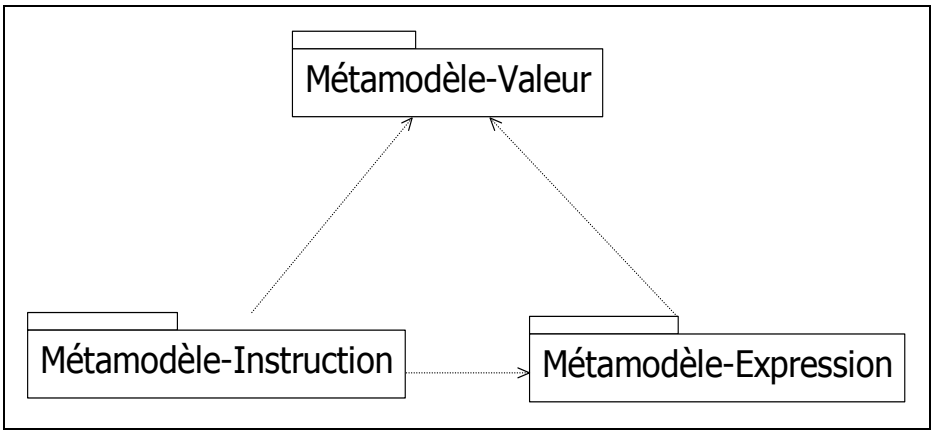

Figure 8.16 Architecture générale du langage MSL.

8.7.3.3 Métamodèle Valeur

Ce métamodèle définit les classes utilisées pour définir les types de valeurs supportées par le langage MSL (voir Figure 8.17). Il consiste en les éléments et les relations suivants :

Valeur : classe abstraite qui représente une superclasse commune à tous les types de valeurs pris en charge par le langage MSL. Cette classe définit un seul attribut, valeur, qui représente le contenu de la valeur.

Valeur-primitive : classe abstraite qui représente une superclasse commune à tous les types de valeurs primitifs.

Chaine-de-caractères : classe qui représente une séquence finie et ordonnées de caractères.

Nombre-entier : classe qui représente des nombres entiers positifs.

Liste : classe qui représente une suite de valeurs primitives.

Liste [1] ⇔ Valeur-primitive [1..*] : spécifie la suite de valeurs dans une liste.

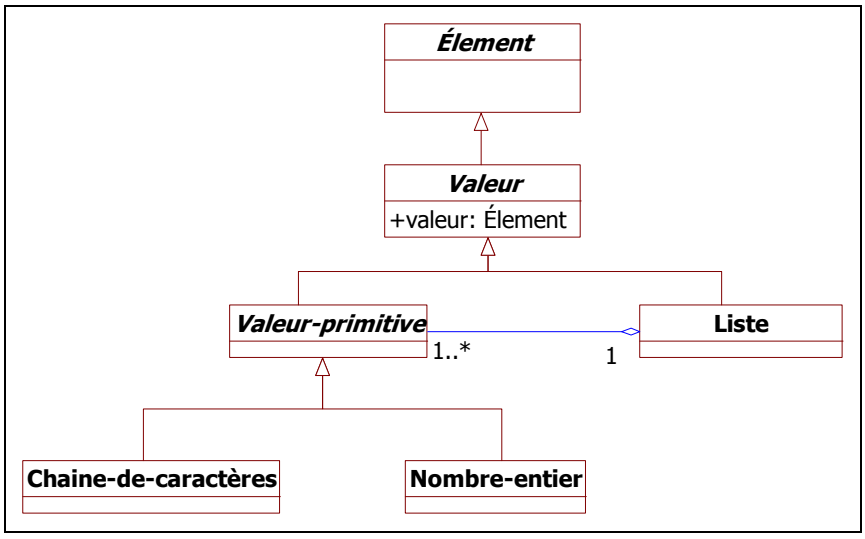

Figure 8.17 Métamodèle Valeur du langage MSL.

8.7.3.4 Métamodèle Expression

Ce métamodèle définit les classes utilisées pour définir les expressions du langage MSL (voir Figure 8.18). Il consiste en les éléments et les relations suivants :

Expression : classe abstraite qui représente une superclasse commune à toutes les expressions qui s'évaluent pour produire une valeur admissible par le langage MSL.

Expression-binaire : représente une expression composée de deux sous-expressions séparées par un operateur. Les attributs définis par cette classe sont :

- *opérande-gauche (Expression)* : première sous-expression ;
- *opérande-droite (Expression)* : deuxième sous-expression ;
- *operateur (String)* : action à appliquer sur les deux sous-expressions.

Constante : représente des nombres ou des chaînes de caractères. Cette classe définit un seul attribut :

- *valeur (String)* : valeur de la constante.

Variable : classe qui représente des références à des valeurs MSL. Cette classe définit un seul attribut :

- *nom (String)* : nom de la variable.

Contenu-de-configuration : représente des noms qui font référence à des valeurs dans le fichier de configuration. Cette classe définit deux attributs :

- *section (String)* : nom de la section dans le fichier de configuration ;
- *paramètre (String)* : nom du paramètre.

Expression-binaire [1] ⇔ Expression [2] : spécifie les expressions composant une expression binaire.

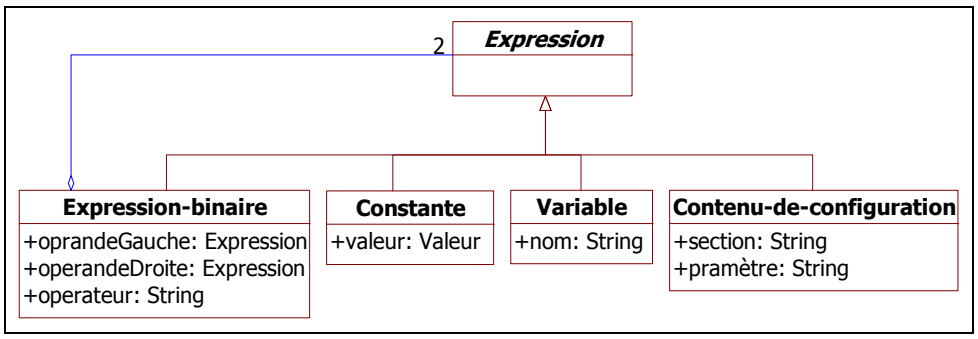

Figure 8.18 Métamodèle Expression du langage MSL.

8.7.3.5 Métamodèle Instruction

Ce métamodèle définit les classes utilisées pour définir les instructions du langage MSL (voir Figure 8.19). Il consiste en les éléments et les relations suivants :

Instruction : classe abstraite qui représente une superclasse commune à toutes les instructions exprimées par le langage MSL.

Déclaration : classe abstraite qui représente une superclasse commune à toutes les instructions destinées à la définition de variables.

Déclaration-variable : représente une instruction de déclaration de variables. Cette classe définit deux attributs :

- *nom* : nom de la variable ;
- *valeur* : valeur d'initialisation de la variable.

Boucle : classe abstraite qui représente une superclasse commune à toutes les classes définissant un ensemble d'instructions qui s'exécutent de façon répétitive jusqu'à ce qu'une condition d'arrêt soit vraie.

Boucle-PourChaque : représente une instruction qui sert à parcourir les éléments d'une liste. Cette classe définit trois attributs :

- *variable* : variable où sont stockées les valeurs de la liste ;

- *liste-parcourue* : référence à une liste de valeurs ;
- *bloc-code* : bloc de code à exécuter à chaque itération.

Condition : classe abstraite qui représente une superclasse commune à toutes les classes définissant un ensemble d'instructions qui s'exécutent si une certaine condition est vraie.

Condition-if: représente une instruction qui permet d'exécuter un bloc d'instructions si une certaine condition est vraie. Cette classe définit trois attributs :

- *expression-test* : Expression représentant la condition d'exécution ;
- *partie-If* : bloc à exécuter si l'expression *expression-test* est évaluée à *Vrai* ;
- *partie-else* : bloc à exécuter si l'expression *expression-test* est évaluée à *Faux.*

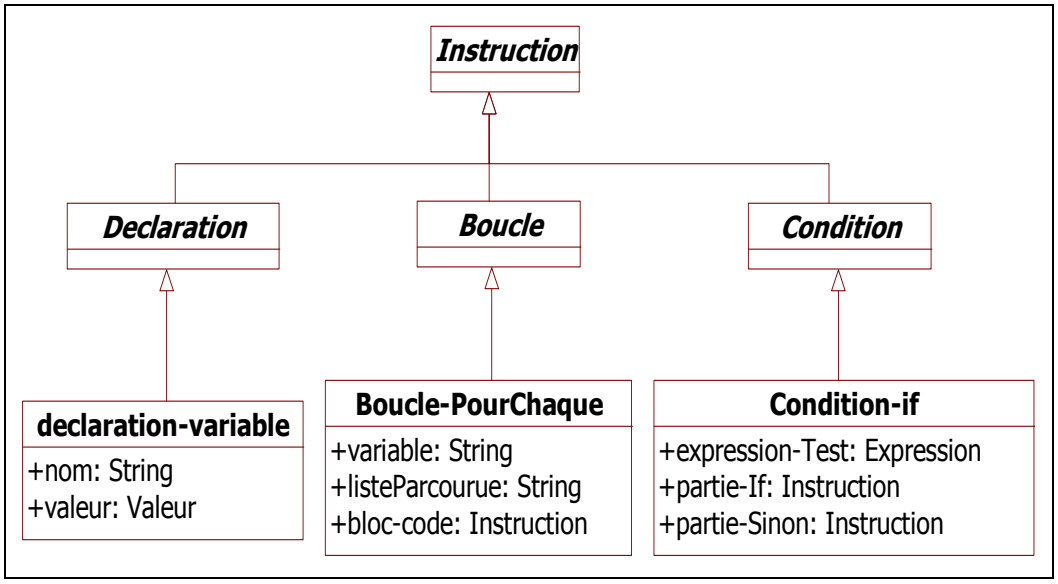

Figure 8.19 Métamodèle Instruction du langage MSL.

8.7.3.6 Règles de grammaire

Un code MSL est considéré valide s'il respecte les règles de grammaire suivantes :

- Tout bloc *Condition* doit avoir un début et une fin. Ne pas terminer une *Condition* est interdit ;

- Toute *Boucle* doit avoir un début et une fin. Ne pas terminer une *Boucle* est interdit ;
- Si un bloc *Condition* commence dans une *Boucle* ou dans un autre bloc *Condition*, alors la fin de ce bloc doit s'y trouver aussi ;
- Si une *Boucle* commence dans un bloc *Condition* ou dans une autre *Boucle*, alors la fin de ce bloc doit s'y trouver aussi.

8.7.4 Syntaxe concrète

La syntaxe concrète du langage MSL est définie en utilisant le langage EBNF (voir Figure 8.20 et Figure 8.21). La suite de cette section présente une description des constructions principales du langage.

Code MSL

Un code MSL est contenu entre les balises <%MSL et %MSL>. Le générateur n'interprète que le code compris entre ces deux balises.

```
// Définition BNF
Code_MSL = "<%MSL" Instruction "%MSL>".

// Code MSL
<%MSL Instruction %MSL>
```

Instruction d'écriture de valeur

L'instruction d'écriture de valeur permet d'écrire dans la page générée la valeur d'une expression. Cette instruction peut se trouver en milieu de ligne. On peut la trouver plusieurs fois dans une ligne.

```
// Définition BNF
Expression = ExpressionSimple [ Relation ExpressionSimple ].
Relation = "=" | "<" | ">".
ExpressionSimple = Nombre | Chaine_de_caracteres | Variable | Valeurconfiguration.
```

```
// Code MSL
<%MSL $Variable %MSL>
<%MSL $Section|Entree %MSL>
<div class="<%MSL $Variable %MSL>"> ... </div>
```

Instruction Var

L'instruction *VAR* permet de donner une valeur à une variable interne du générateur.

```
// Définition BNF
Declaration = "VAR" { DeclarationVariable }.
DeclarationVariable = Variable ["=" Expression].

// Code MSL
<%MSL VAR $NomDeVariable = Expression %MSL>
```

Bloc SI ... SINONSI ... SINON ... FINSI

Ce bloc d'instructions permet de générer ou non le texte situé entre les balises en fonction d'une condition. Les instructions <%MSF SINONSI *Condition* %MSF> et <%MSF SINON %MSF> sont facultatives.

```
// Définition BNF
Instruction_Si = "Si" ["NON"] Expression.
Instruction_SiNonSi = "SINONSI" ["NON"] Expression .
Instruction_SiNon = "SINON".
Instruction_FinSi = "FINSI".

// Code MSL
<%MSL SI [NON] Expression = Expression %MSL>
...
<%MSL SINONSI [NON] Expression = Expression %MSL>
...
<%MSL SINON %MSL>
...
<%MSL FINSI %MSL>
```

Bloc POUR_CHAQUE ... FIN_POUR_CHAQUE

Cette instruction permet de répéter le texte situé entre les deux balises POUR_CHAQUE et FIN_POUR_CHAQUE autant de fois qu'il y a d'éléments dans l'expression *ExpressionListe,* une expression quelconque donnant pour résultat une suite de valeurs séparées par des virgules ou des points-virgules. Pour chaque itération, la valeur correspondante dans la liste est affectée à la variable de la boucle.

```
// Définition BNF
Instruction_Boucle = "POUR_CHAQUE" Variable "DANS" ExpressionListe .
ExpressionListe = Valeurconfiguration|Variable.
Instruction_FinBoucle = "FIN_POUR_CHAQUE" .

// Code MSL
<%MSL POUR_CHAQUE NomDeVariable DANS ExpressionListe %MSL>
...
<%MSL FIN_POUR_CHAQUE %MSL>
```

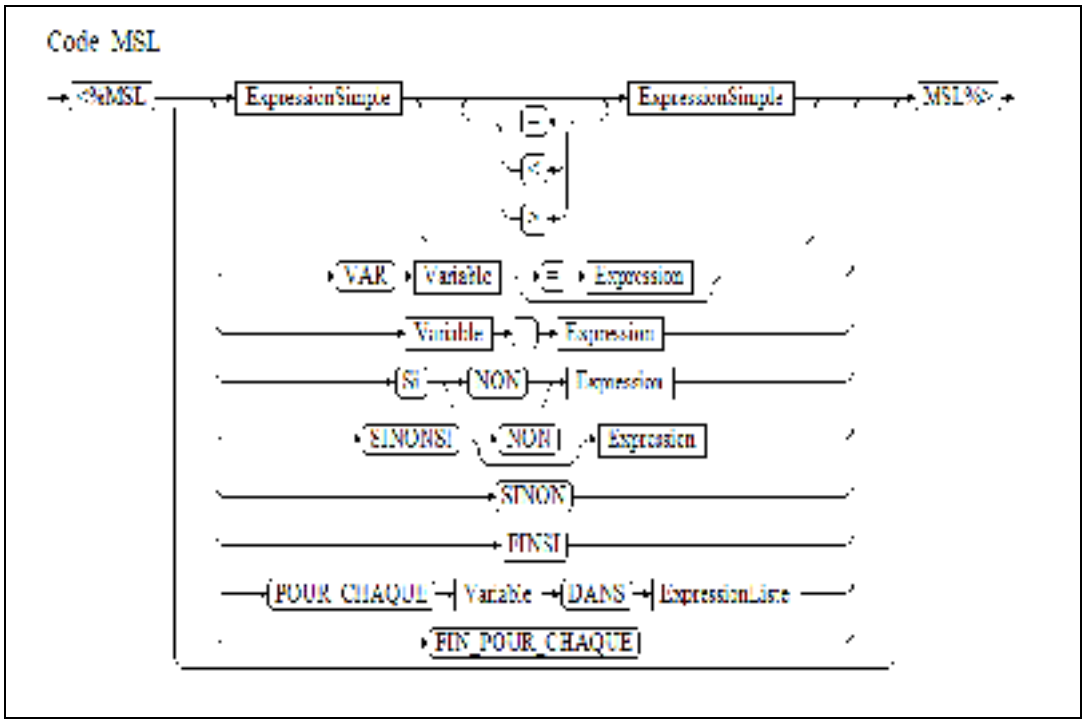

Figure 8.20 Grammaire du langage MSL.

```
// Définition BNF
Identifiant = letter { letter | digit |"_"}.
Variable = "$" Identifiant.
Nombre = digit { digit }.
Chaine_de_caracteres = "'" { character } "'" | '"' { character } '"'.

Code_MSL = "<%MSL" Instruction "%MSL>".
Instruction = [ Expression | Declaration | Affectation | Instruction_Si |
Instruction_SiNonSi | Instruction_SiNon | Instruction_FinSi | Instruction_Boucle |
Instruction_FinBoucle] .
Expression = ExpressionSimple [ Relation ExpressionSimple ].
Relation = "=" | "<" | ">".
ExpressionSimple = Nombre | Chaine_de_caracteres | Variable | Valeurconfiguration.
Valeurconfiguration = NomSection "|" NomEntree.
NomSection = Identifiant.
NomEntree = Identifiant.

Declaration = "VAR" { DeclarationVariable }.
DeclarationVariable = Variable ["=" Expression].

Affectation = Variable "=" Expression.

Instruction_Si = "Si" ["NON"] Expression.
Instruction_SiNonSi = "SINONSI" Expression .
Instruction_SiNon = "SINON".
Instruction_FinSi = "FINSI".

Instruction_Boucle = "POUR_CHAQUE" Variable "DANS" ExpressionListe .
ExpressionListe = Valeurconfiguration|Variable.
Instruction_FinBoucle = "FIN_POUR_CHAQUE" .
```

Figure 8.21 Grammaire du langage MSL en BNF.

8.7.5 Sémantique

Les instructions du langage *MSL* sont assez simples et leur sémantique est assez explicite. Toutefois, et à titre d'exemple, nous présentons la description de la sémantique des deux instructions POUR_CHAQUE et SI en utilisant des diagrammes d'activités (voir Figure 8.22 et Figure 8.23).

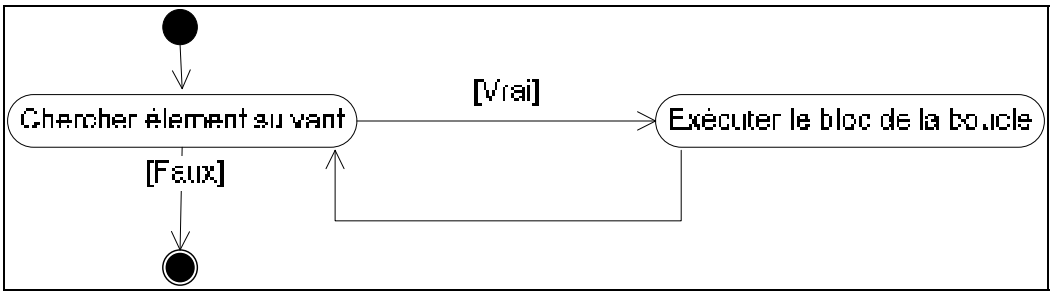

Figure 8.22 Diagramme d'activités pour l'instruction POUR_CHAQUE.

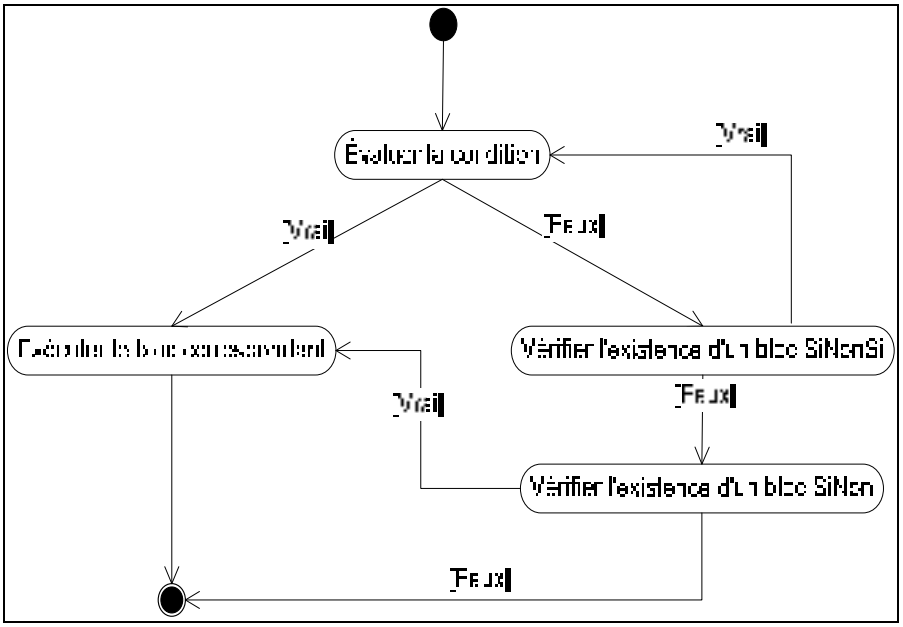

Figure 8.23 Diagramme d'activités pour l'instruction Si.

8.8 Sommaire et synthèse

Dans ce chapitre, nous avons montré, à travers une étude de cas, comment la méthode proposée dans le chapitre 6 peut être utilisée pour définir des langages dédiés. Au cours de cette étude de cas, nous avons procédé à la redéfinition de trois types de DSL en appliquant la dite méthode : un DSL de spécification (WCL), un DSL de modélisation (DDL) et un DSL de programmation (MSL).

Nos observations sur l'application de la méthode se résument comme suit :

- **Applicabilité :** la méthode s'est montrée simple et pratique et ce pour les trois types de DSL définis. Les éléments de la méthode (e.g. phases, activités, artefacts, techniques, etc.) se sont avérés pertinents et utiles pour la redéfinition de ces trois langages ;
- **Complétude :** la définition des langages WCL, DDL et MSL a montré que la méthode couvre bien les aspects principaux de la définition d'un langage dédié, notamment, la syntaxe abstraite, la syntaxe concrète et la sémantique. En effet, tous les éléments de la méthode ont été définis de façon à rendre la définition de ces aspects facile et intelligible ;
- **Adéquation et flexibilité :** la méthode a été appliquée pour la redéfinition de trois types de DSL et ce sans aucune adaptation particulière. Ceci nous amène à présumer que la méthode peut être utilisée pour définir différents types de DSL. Certes, des expérimentations supplémentaires sont nécessaires pour examiner de façon plus exacte l'adéquation de la méthode au développement des différents types de DSL.

Les leçons apprises de cette étude de cas se résument comme suit :

1. **Souvent plusieurs DSL sont nécessaires pour développer une application complète :** il est rare de trouver un seul DSL capable de résoudre tous les problèmes d'un domaine. Ainsi le développement d'un DSL pour un domaine donné doit prendre en considération le contexte d'utilisation afin faciliter sa coexistence et son interopérabilité avec les autres DSL de ce domaine ;
2. **Ce qui fait un bon DSL est une question de perspective :** nombreuses sont les caractéristiques d'un bon DSL dans la littérature. Néanmoins, en pratique, il est très

difficile, voire impossible, de développer un DSL qui possède toutes ces caractéristiques. Les caractéristiques considérées par les développeurs sont pour la plupart dictées par les besoins, les contraintes et les propriétés particulières caractérisant chaque DSL. Par exemple, dans notre cas, la définition des langages WCL, DDL et MSL était dirigé principalement par la simplicité et la facilité d'implémentation. Des caractéristiques telles que la facilité de réutilisation et le pouvoir d'attraction étaient considérées moins importantes ;

3. **Gardez les DSL aussi simples que possible :** se limiter à ce qui est absolument indispensable afin de réduire au maximum la complexité des DSL et ainsi faciliter leur développement et leur maintenance. Les utilisateurs auront toujours des propositions d'amélioration et des souhaits d'évolution. La réalisation d'une simple évolution peut être très coûteuse ;
4. **Définissez des DSL faciles à implémenter :** les meilleures définitions de DSL ne sont pas toujours faciles à implémenter. Par exemple, dans le cas du langage DDL, il aurait été plus souple vis-à-vis des utilisateurs d'utiliser une syntaxe concrète basée sur des diagrammes entité-association sauf que l'implémentation des outils de support aurait été nettement plus difficile. Ainsi, nous avons opté pour une notation à base de formulaires, ce qui est nettement plus facile à implémenter.

CHAPITRE 9

OBSERVATIONS SUR L'UTILISATION DES LANGAGES WCL, DDL et MSL

9.1 Introduction

Ce chapitre présente une discussion de l'utilisation des langages dédiés WCL, DDL et MSL, décrits dans le chapitre précédent, dans le développement de sites Web commerciaux.

Nous commençons par parler des types de sites Web développés en utilisant lesdits DSL. Ensuite, nous présentons nos observations sur la qualité de ces DSL en utilisant les attributs de qualité identifiés dans le chapitre 7. Enfin, nous concluons par une discussion de l'apport de ces DSL sur l'amélioration de la productivité et de la qualité des sites Web développés.

9.2 Projets développés avec les DSL WCL, DDL et MSL

Les projets réalisés avec les langages WCL, DDL et MSL sont divers et variés (voir ANNEXE III pour plus de détail sur le contexte de développement de ces projets). On y trouve des intranets, des extranets, des sites e-business, des vitrines, des catalogues, des blogues, et autres. Les modèles génériques prévoient un ensemble de modules répondant à la plupart des besoins suscités par ces types de sites Web.

Certains de ces projets avaient des besoins spécifiques qui n'ont pas été pris en compte par les modèles standards. Pour ces projets, une analyse est effectuée afin d'évaluer la pertinence de ces besoins pour la famille. Si ces besoins ont été jugés pertinents pour la famille alors ils sont insérés dans les modèles standards. Sinon ils sont implémentés en tant que personnalisation spécifique au membre en question.

9.3 Observations sur la qualité

La qualité des langages dédiés WCL, DDL et MSL est évaluée en utilisant les attributs de qualité identifiés dans le chapitre 7. Les scores ont été estimés en se basant sur des discussions menés avec huit développeurs Web qui ont utilisé ces DSL pendant une période allant de un à cinq ans. Chacun de ces développeurs a participé au développement d'au moins cinq projets basés sur ces DSL (voir ANNEXE III pour plus de détail sur ces développeurs).

Nous avons adopté une échelle à six niveaux allant de 0 à 5 : faible (0), moyen (1), assez bien (2), bien (3), très bien (4) et excellent (5).

La suite de cette section résume les observations faites par les huit développeurs sur la qualité des trois DSL. En se basant sur ces observations, nous, les chercheurs, avons établi (subjectivement) des scores pour les différents attributs de qualité.

9.3.1 Langage WCL

Le Tableau 9.1 résume les observations sur la qualité du langage WCL.

Tableau 9.1 Observation sur la qualité du langage dédié WCL

Attribut de qualité	**Observation**	**Score**
Expressivité	Les concepts offerts par le langage WCL sont expressifs et familiers aux concepteurs/développeurs de sites Web.	4
Convenance	Les fonctions offertes par le langage sont appropriées et répondent bien aux besoins en termes de configuration de sites Web.	4
Facilité d'exploitation	L'exploitation du DSL est simple. Elle consiste en l'édition des valeurs des paramètres de configuration.	4
Facilité d'apprentissage	Le langage est facile à apprendre. Il suffit de connaître la signification des paramètres de configuration et leurs valeurs correspondantes.	4

Attribut de qualité	**Observation**	**Score**
Facilité de modification	L'ajout de nouveaux paramètres de configuration ne nécessite aucune modification au niveau du générateur. Les paramètres ajoutés dans le formulaire sont automatiquement pris en charge par le générateur.	4
Facilité de test	Pour tester le langage il suffit de vérifier si les paramètres de configuration et leurs valeurs sont bien générés dans le fichier de configuration.	4
Facilité de réutilisation	Les langages WCL et DDL utilisent le même générateur, ce qui nuit à l'utilisation du langage WCL comme base pour le développement de nouveaux langages.	1
Réutilisabilité	Le langage peut être facilement réutilisé pour la configuration d'autres propriétés et ce sans aucune modification.	3
Capacité d'exécution	Le langage est doté d'un générateur capable de générer un fichier de configuration à partir d'un modèle de configuration.	4
Productivité	La production d'un modèle de configuration est rapide une fois qu'on connait les spécificités et les besoins du site Web.	4

9.3.2 Langage DDL

Le Tableau 9.2 résume les observations sur la qualité du langage DDL.

Tableau 9.2 Observations sur la qualité du langage dédié DDL

Attribut de qualité	**Observation**	**Score**
Expressivité	Les concepts offerts par le langage sont généralement expressifs. Cependant certaines expressions utilisées dans les modèles ne sont pas assez claires.	3
Convenance	Les fonctions offertes par le langage sont appropriées et répondent aux besoins de modélisation d'une base de données.	4

Attribut de qualité	**Observation**	**Score**
Facilité d'exploitation	L'exploitation du langage DDL est facile. Son éditeur offre une interface graphique facile à utiliser avec des fonctionnalités comme la copie, la suppression, l'import de tables, etc.	4
Facilité d'apprentissage	Les concepts du langage sont faciles à comprendre, toutefois la compréhension de la sémantique de tous les éléments demande du temps et surtout de la pratique.	3
Facilité de modification	La modification du langage implique des modifications au niveau de l'éditeur de modèles et du générateur. Par exemple, l'ajout d'une nouvelle caractéristique Web pour une table implique son ajout au niveau du modèle *Table* afin de permettre son édition et l'adaptation du générateur afin qu'il puisse la générer dans le fichier de configuration.	3
Facilité de test	Le test du langage consiste à vérifier la correspondance entre les données des modèles et la configuration générée. Cela implique la connaissance des correspondances entre les éléments du langage et les sections/paramètres du fichier INI de configuration.	3
Facilité de réutilisation	Les langages DDL et WCL utilisent le même générateur, ce qui rend l'utilisation du langage DDL comme base pour le développement de nouveaux langages difficile.	1
Réutilisabilité	Il est difficile de réutiliser le langage DDL pour des domaines autres que le celui du Web à cause des propriétés Web intégrées dans ce langage.	3
Capacité d'exécution	Le langage est doté d'un générateur capable de générer un fichier de configuration à partir des modèles définis.	4
Productivité	La production d'un modèle de données avec le langage DDL est rapide. Les fonctionnalités offertes par l'éditeur facilitent considérablement la création et la modification de modèles de données.	4

9.3.3 Langage *MSL*

Le Tableau 9.3 résume les observations sur la qualité du langage *MSL*.

Tableau 9.3 Observations sur la qualité du langage dédié MSL

Attribut de qualité	**Observation**	**Score**
Expressivité	Le langage MSL ressemble, dans ses concepts, aux langages PHP et JSP. À ce titre, il opère au même niveau d'abstraction qu'un langage de programmation généraliste.	3
Convenance	Le langage offre un ensemble restreint d'instructions qui répondent uniquement aux besoins les plus essentiels en termes d'accès et de manipulation de données de configuration. Ceci oblige parfois les programmeurs à chercher des solutions de contournement afin d'arriver aux résultats désirés.	3
Facilité d'exploitation	Le langage MSL ressemble dans son exploitation aux langages PHP et JSP. Or, le fait que chacune de ses instructions doit être insérée entres des balise <%MSL et %MSL> rend le langage verbeux et peu flexible.	3
Facilité d'apprentissage	Bien que le langage MSL ne soit constitué que de quelques instructions, il a été observé que le temps moyen nécessaire pour s'y familiariser est de trois semaines.	3
Facilité De modification	La modification du langage implique des modifications au niveau du compilateur, ce qui nécessite des compétences en développement de langages.	2
Facilité de test	Tester le langage revient à tester son compilateur et son générateur de code. Cela demande une connaissance des méthodes et des techniques de test utilisées pour tester les compilateurs et les générateurs de code (voir (Kossatchev et Posypkin, 2005) pour plus de détail).	2
Facilité de réutilisation	L'utilisation du langage MSL comme base de développement pour de nouveaux langages nécessite des modifications importantes au niveau du compilateur.	1
Réutilisabilité	Le langage MSL a été conçu pour être encapsulé dans le code source de quasiment n'importe quel langage de programmation et ce sans aucune modification au niveau du compilateur / générateur.	4
Capacité d'exécution	Le langage est doté d'un générateur de code capable de générer du code source à partir des modèles génériques (*templates*).	4

Attribut de qualité	Observation	Score
Productivité	Il n'y pas un éditeur spécial pour le langage MSL. Les *templates* sont édités généralement dans l'éditeur du langage hôte ou dans n'importe quel éditeur de texte (ex. Eclipse, Bloc-notes, etc.). Les utilisateurs du langage MSL sont donc privés de fonctionnalités comme l'auto-complétion, la coloration syntaxique, le pliage et dépliage du code, etc. De plus le compilateur ne fournit aucune indication sur les erreurs rencontrées. S'il trouve une instruction invalide, le compilateur ignore la ligne entière et passe à la ligne suivante, ce qui rend la détection d'erreurs difficile surtout pour les novices dans ce langage.	2

9.4 Apport des DSL sur le développement des applications

Les huit développeurs considèrent que la mise en place d'un processus de développement basé sur les DSL a permis de réaliser des gains considérables en termes de coût de développement, de productivité et de qualité. Les bénéfices réalisés (mais évalués subjectivement et qualitativement) incluent :

Réduction du coût/temps de développement : les DSL ont permis de systématiser la réutilisation d'une grande partie des actifs fondamentaux. Cette systématisation a permis de réduire, entre autres, les besoins, tant quantitatifs que qualitatifs, en ressources nécessaires au développement de sites Web. En effet, le développement d'un site Web avec cette approche ne nécessite habituellement que trois à quatre intervenants, soit un ou deux développeurs (concepteurs) Web, un intégrateur XHTML/CSS et un développeur Flash. Les tâches de ces intervenants se limitent habituellement à la personnalisation de l'interface guichet (*front-office*), ce qui réduit considérablement le temps de développement.

Amélioration de la productivité : les DSL ainsi que leurs outils associés ont été conçus pour automatiser au maximum les tâches de programmation répétitives et pour améliorer la productivité des équipes de développement. Effectivement, avec ces outils, les équipes de

développement sont en mesure de créer plus rapidement et plus efficacement des sites Web et ce grâce à la génération automatique de code et à la minimisation du nombre de tâches à réaliser pendant le cycle de développement d'un site Web.

Simplification du développement et de la maintenance : l'utilisation des DSL a permis de raccourcir considérablement le cycle de développement. Ainsi, le développement d'un nouveau site Web ne nécessite généralement pas plus de trois activités principales : **1)** la configuration du site avec le langage WCL, **2)** la conception de la base de données avec le langage DDL et **3)** la personnalisation du site pour répondre aux besoins spécifiques du client.

La configuration du site et la conception de la base de données sont faciles et rapides à réaliser grâce aux éditeurs conçus spécialement à cette fin. Ces éditeurs offrent aux développeurs un environnement de conception de haut niveau d'abstraction qui masque beaucoup des détails techniques d'implémentation. À partir de cette conception, les générateurs sont en mesure de générer un site Web fonctionnel constitué d'un back-office totalement opérationnel et d'une structure de base fonctionnelle pour le front-office. Ceci réduit considérablement le taux de programmation manuelle et simplifie ainsi le développement. De cette façon, l'équipe de développement peut concentrer ses efforts sur la personnalisation du site afin de répondre aux mieux aux besoins spécifiques du client.

Cette approche de développement facilite également la maintenance. Comme tous les sites générés adoptent la même architecture et la même structure implémentées au niveau des modèles génériques, un développeur ne trouve pas de difficulté à maintenir un site développé par un autre développeur. Plus que cela, il a été constaté que même les nouveaux développeurs ne trouvent pas de difficulté à maintenir les sites existants après une formation sur les modèles génériques.

CONCLUSION

Ces dernières années, les langages dédiés ont suscité beaucoup d'intérêt et ont fait l'objet d'une attention toute particulière de la part de la communauté du développement logiciel. Leur pouvoir à exprimer les solutions à un niveau d'abstraction élevé et leur capacité à améliorer la productivité et la qualité des produits logiciels sont les principales motivations derrière cet intérêt. Néanmoins, le développement de ces langages est encore considéré comme une activité difficile et coûteuse, ce qui constitue souvent un frein à leur adoption dans un cadre industriel.

La problématique de développement des langages dédiés est abordée en partie par certains éditeurs d'outils de développement comme IBM et Microsoft. Ces derniers ont reconnu la nécessité de fournir des technologies et des outils pour soutenir et rendre plus accessible le développement des DSL, ce qui a donné naissance à des outils comme les *DSL Tools* et *EMP*. Néanmoins, le traitement de l'axe des outils ne peut, à lui seul, résoudre cette problématique. La maitrise de cette problématique nécessite de prendre en considération aussi les axes processus et standardisation.

Le travail de recherche décrit dans cette thèse est une contribution à l'axe des processus. Le but principal était d'élaborer une méthode pour la définition des langages dédiés qui décrit, entre autres, les phases et les activités à entreprendre, les artefacts à manipuler et les techniques à utiliser lors du développement d'un DSL.

Le travail de recherche a été réalisé en trois phases : une phase de revue de littérature, suivie de la phase d'élaboration de la méthode, et enfin, la phase d'expérimentation permettant d'évaluer la méthode proposée.

Bilan du travail

Dans la première phase de notre recherche (phase informationnelle), une revue de la littérature a été menée afin de comprendre les fondamentaux des langages dédiés ainsi que les contextes et les enjeux liés à leur création et à leur utilisation. Notre revue de littérature a porté donc sur deux domaines : 1) les fondamentaux des DSL et 2) les approches et technologies utilisant les DSL.

La revue de littérature sur les approches et technologies utilisant les DSL a fait ressortir que de nombreuses technologies (ex. lignes de produits logiciels, usines à logiciel, développement dirigé par les langages) préconisent l'utilisation des langages dédiés pour améliorer la productivité et élever le niveau d'abstraction du développement logiciel. Ces langages présentent de nombreux avantages par rapport aux langages généralistes comme UML, mais ils sont loin d'être la panacée pour résoudre tous les problèmes du développement logiciel. Ainsi, l'utilisation du langage de modélisation UML reste envisageable voire, parfois, plus avantageuse.

De la revue de littérature sur les fondements des DSL, il en ressort qu'il n'existe aucune méthode reconnue pour le développement des DSL. Ce développement se fait toujours de manière ad-hoc en utilisant des pratiques et des techniques très variées. L'absence d'une méthode clairement définie pour le développement des DSL constitue un des facteurs importants de la problématique de développement des DSL. Au cours de cette revue de littérature nous avons identifié d'autres facteurs de cette problématique, notamment, l'ambigüité du concept du DSL et de celui du domaine, les compétences exigées en développement de langages et l'insuffisance des outils.

En résumé, cette phase d'exploration nous a permis de comprendre les enjeux et les concepts autour des DSL et d'approfondir la problématique de développement des langages dédiés. Dans cette même phase, nous avons réalisé une synthèse de la littérature traitant le développement des DSL afin d'identifier les pratiques et les techniques généralement impliquées dans un processus de définition de DSL. Ce sont ces pratiques et ces techniques

qui nous ont servi d'intrants pour la phase d'élaboration de notre méthode de définition des DSL.

La deuxième phase de recherche (phase propositionnelle) a été consacrée à l'élaboration de la première version de la méthode qu'on propose pour la définition des langages dédiés. Nous avons élaboré cette version en utilisant le processus unifié de Rational (RUP) comme cadre de structuration. Ainsi, le processus de définition des langages dédiés a été décomposé en trois phases : initiation, élaboration et construction. Chacune de ces phases a été décrite en termes de son but, ses objectifs, ses activités de base, ses activités facultatives, ses critères d'évaluation et ses artefacts.

La troisième phase de la recherche (phase d'évaluation et d'expérimentation) a porté sur l'amélioration et le perfectionnement de la méthode. Cette phase a été réalisée dans le cadre d'un processus de recherche-action dont le volet théorique a porté sur l'élaboration du cadre théorique de la méthode et le volet pratique s'est concentré sur l'expérimentation et l'évaluation de l'applicabilité de cette méthode.

L'adoption d'un processus de recherche-action nous a permis d'envisager de manière réaliste et pragmatique le développement de notre méthode. Chaque itération de ce processus a consisté en quatre activités : analyse, expérimentation, évaluation et mise à niveau. Plusieurs itérations ont été nécessaires avant d'aboutir à la version que nous avons jugée satisfaisante. Après la fin du processus de recherche-action, nous avons procédé à la normalisation de la méthode avec la norme ISO/IEC 24744. Cette normalisation nous a permis de proposer une méthode conforme à la norme ISO/IEC 24744, ce qui, à notre avis, assure un certain niveau d'intégrité et de cohérence à notre méthode.

Enfin, afin de démontrer le bien fondé de la méthode proposée et de vérifier son applicabilité dans la pratique, une étude de cas réaliste a été réalisée pour le domaine du Web. Ce cas a fait ressortir trois DSL : un DSL de configuration (WCL) pour la spécification des paramètres d'un site Web, un DSL de modélisation (DDL) pour la définition de la base de données et de ses caractéristiques Web et un DSL de programmation (MSL) pour implémenter le mécanisme de méta-programmation permettant de faciliter la génération de

code. Les trois DSL ont été définis en appliquant la méthode proposée. Cette dernière s'est montrée simple et pratique pour définir les trois types de DSL. Les éléments de la méthode (phases, activités, artefacts, etc.) se sont avérés utiles et suffisants pour couvrir les aspects principaux de définition d'un langage dédié, soit, la syntaxe abstraite, la syntaxe concrète et la sémantique.

En conclusion, nous considérons que l'objectif de notre recherche a été accompli. L'objectif principal consistait à développer une méthode pour la définition des langages dédiés, basée sur la norme ISO/IEC 24744. L'objectif secondaire de la recherche portait sur l'utilisation et la qualité des DSL développés avec cette méthode. Néanmoins, certaines limites liées à ce projet de recherche méritent d'être soulignées.

Limites de la recherche

Les limites majeures de cette recherche se résument comme suit :

- La méthode proposée se limite au processus de définition des langages dédiés (phases d'analyse et de conception) sans aborder la phase d'implémentation ;
- La méthode a été élaborée du point de vue des développeurs, c'est-à-dire qu'elle s'est limitée aux activités essentielles à la définition des DSL sans considérer les aspects gestion de projets, déploiement, etc. ;
- Insuffisance du nombre de cas d'expérimentation : l'évaluation de la méthode a été réalisée en utilisant trois DSL relevant tous du même domaine. Bien que cette expérimentation nous ait permis de montrer l'applicabilité et la flexibilité de la méthode, elle ne peut être considérée comme suffisante pour juger avec certitude de son efficacité. D'autres expérimentations seront nécessaires afin de pouvoir affirmer avec confiance que la méthode proposée est bien capable de définir les différents types de DSL. Malheureusement, Il a été difficile pour nous de trouver des partenaires industriels familiarisés avec le développement des DSL et qui sont prêts à expérimenter notre méthode.

En dépit des limites notées dans ce projet de recherche, nombres de contributions ont été apportées par cette recherche.

Contributions de la recherche

La contribution principale de cette recherche est la méthode qu'on propose pour la définition des langages dédiés. Cette méthode est destinée à combler un vide existant en termes de méthodes de développement de ces langages. L'élaboration d'une telle méthode répond donc à un besoin réel et clairement exprimé par de nombreux spécialistes dans ce domaine.

À court terme, la méthode proposée contribuera à surmonter la difficulté du développement des DSL et à rendre celui-ci plus intelligible et plus compréhensible. Les développeurs de DSL y trouveront un cadre cohérent et clairement défini pour la définition de leurs DSL.

À plus long terme, le travail pourrait contribuer à l'instauration d'une discipline d'ingénierie pour le développement des langages dédiés. Ce travail devrait compléter les efforts déployés au niveau des outils dans le but de préparer une infrastructure solide pour soutenir la pratique d'une éventuelle ingénierie de langages dédiés. Une telle discipline aidera à dissiper le scepticisme des industriels autour des approches dirigées par les langages et ainsi promouvoir leur adoption dans le cadre industriel.

Les contributions de ce travail de recherche se résument comme suit :

- Compilation et organisation des pratiques utilisées dans le développement des langages dédiés dans un processus intégral et cohérent facilitant la définition des DSL ;
- Proposition d'une technique permettant d'identifier des attributs de qualité à partir de facteurs de succès ;
- Identification d'une liste d'attributs de qualité pour les langages dédiés. Ces critères seront utiles aussi bien pour les développeurs désirant assurer un niveau de qualité pour leur DSL que pour les acquéreurs souhaitant évaluer et choisir entre plusieurs DSL ;
- Mise en correspondance des pratiques de définition des langages dédiés et des concepts de la norme ISO/IEC 24744.

L'originalité de ce travail de recherche est double. D'une part, nous proposons une méthode pour la définition des DSL qui est proche dans sa constitution des méthodes de développement logiciel. D'autre part, la méthode est définie conformément à un standard en matière de conception de méthodes, soit la norme ISO/IEC 24744.

Publication

A. Kahlaoui, A. Abran, E. Lefebvre (2008) « DSML Success Factors and Their Assessment Criteria », METRICS News. Vol.13: No.1. p.p 43-51 February.

Perspectives d'avenir

Ce travail de recherche ouvre la voie vers différentes perspectives et peut être poursuivi dans plusieurs directions. Parmi ces directions, nous suggérons celle du perfectionnement de la méthode. Ensuite, il y a la direction de l'expérimentation et de l'évaluation de la méthode, et enfin la direction de la qualité des DSL.

Perspectives sur le perfectionnement de la méthode

La méthode proposée dans cette thèse s'est concentrée surtout sur l'analyse et la conception des langages dédiés, ce qui laisse des opportunités de recherche importantes en matière d'amélioration et d'enrichissement de cette méthode. Parmi ces opportunités nous pouvons noter :

- Enrichissement de la méthode en y incorporant les pratiques relatives à l'implémentation des langages dédiés ainsi que celles traitant d'autres aspects comme le déploiement et la gestion de configuration. Il y a aussi la possibilité de compléter et d'affiner la description des éléments déjà définis ;
- Étude approfondie de la relation entre la conception et l'implémentation des DSL afin de compléter la méthode avec des techniques permettant de combler le fossé entre ces deux activités. Jusqu'à date, le passage de la conception à l'implémentation n'est pas assez direct. Nous pensons qu'une recherche dans cette direction aidera à simplifier ce passage.

Perspectives sur l'évaluation de la méthode

Comme déjà mentionné dans les limites de la recherche, des expérimentations supplémentaires sont nécessaires pour bien évaluer la pertinence de notre méthode. Dans cette direction nous suggérons de mener avec des partenaires industriels des études expérimentales mettant en œuvre cette méthode. Il va sans dire que ces expérimentations doivent couvrir les différents types de DSL. Elles doivent également être menées de façon à garantir l'objectivité et l'impartialité des résultats.

Perspectives sur la qualité des DSL

Les résultats de notre travail sur les attributs de qualité des DSL pourraient servir de base à d'autres projets de recherche visant la qualité des langages dédiés. Plusieurs voies de recherche sont envisageables sur ce sujet :

- Étudier la possibilité d'élaborer un cadre intégré pour l'évaluation des langages dédiés en se basant sur la liste des attributs de qualité identifiés dans cette thèse ;
- Étudier la possibilité d'élaborer un modèle de qualité pour les langages dédiés tel que celui établi pour les produits logiciels dans la norme ISO/IEC 9126. Un tel modèle permettra d'évaluer la qualité des DSL selon différents points de vue (interne, externe et en utilisation).

Enfin, nous espérons que cette recherche aura contribué à rendre le développement des langages dédiés plus accessible et plus intelligible et qu'elle servira de base à d'autres recherches dans ce champ disciplinaire.

ANNEXE I

MAPPAGE ENTRE LE PROCESSUS DE DÉVELOPPEMENT LOGICIEL RUP ET LE PROCESSUS DE DÉVELOPPEMENT DES LANGAGES DÉDIÉS

Concepts

Tableau-A I-1 Mappage des concepts

RUP	DSL
Milestone	Milestone
Phase	PhaseKind
Discipline	ProcessKind
Activity, Workflow detail	WorkUnitKind
Discipline, activity or technique	WorkUnitKind (or Process?)
Process (Life cycle)	TimeCycleKind
Artifacts	WorkProductKind
Output, deliverable, Output	Outcome
Workflow	N/A
Technique	TechniqueKind
Guideline	Guideline
Release	N/A
baseline	N/A
Build	N/A
Iteration	BuildKind
Role	RoleKind
Step	TaskKind

Phases

Tableau-A I-2 Mappage des phases

RUP	DSL
Initialisation	Initialisation
Élaboration	Élaboration
Construction	Construction
Transition	

- **Phase d'initialisation**

Tableau-A I-3 Mappage des objectifs

RUP	DSL
Définition du périmètre du projet	Définition du domaine du DSL (portée)
Détermination des exigences principales du projet et de ses critères d'acceptation	Détermination des caractéristiques principales du DSL et de ses critères d'acceptation
Estimation du coût global du projet	Estimation du coût global et du temps nécessaire au développent du DSL
Estimation des risques potentiels	Estimation des risques potentiels
Préparation d'un environnement de soutien pour le projet	Préparation d'un environnement de soutien pour le développement du DSL

Tableau-A I-4 Mappage des activités

RUP	DSL
Définir la portée du logiciel	Définir le domaine (Scoping)
Définir la vision du logiciel	Définir la vision du DSL
Préparer le plan d'affaire	Préparer le plan d'affaire
Préparer l'environnement de soutien	Préparer l'environnement de soutien

Tableau-A I-5 Mappage des artefacts

RUP	DSL
Vision	Vision
Glossaire	Terminologies
Exigences	Exigences
Modèle du domaine	Modèle du domaine
Liste des risques	Liste des risques
Cas d'affaire	Cas d'affaire

Tableau-A I-6 Mappage des jalons

RUP	DSL
Objectifs du logiciel	Objectifs du DSL

Tableau-A I-7 Mappage des critères d'évaluation

RUP	DSL
Consentement des parties prenantes sur la portée du logiciel et sur l'estimation du coût/temps	Consentement des parties prenantes sur la portée du DSL et sur l'estimation du coût/temps
Consentement sur les exigences que doit satisfaire le logiciel	Consentement sur les exigences que doit satisfaire le DSL
Les risques les plus critiques ont été identifiés et une stratégie de mitigation a été prévue pour chacun	Les risques les plus critiques ont été identifiés et une stratégie de mitigation a été prévue pour chacun

- **Phase d'élaboration**

Tableau-A I-8 Mappage des objectifs

RUP	DSL
S'assurer que les exigences sont suffisamment stables pour prévoir le coût et le temps nécessaires au développement du logiciel	S'assurer que les exigences sont suffisamment stables pour prévoir le coût et le temps nécessaires au développement du DSL
Établir une architecture de référence et démontrer que celle-ci répondra aux exigences	Établir une architecture et démontrer que celle-ci répondra aux exigences
Établir une procédure de test permettant de valider le logiciel	Établir une procédure de test permettant de valider le DSL
Mettre en place un environnement de support pour soutenir le développement du logiciel	Mettre en place un environnement de support pour soutenir le développement du DSL

Tableau-A I-9 Mappage des activités

RUP	DSL
N/A	Identifier les abstractions du domaine
Définir l'architecture du logiciel	Définir l'architecture du DSL
Préparer les procédures de test du logiciel	Préparer les procédures de test du DSL
Raffiner la vision du logiciel	Raffiner la vision du DSL
Créer un plan d'itération détaillé de la phase de construction	Créer un plan d'itération détaillé de la phase de construction
Mettre en place un environnement favorable au développement du logiciel	Mettre en place un environnement favorable au développement du DSL

Tableau-A I-10 Mappage des jalons

RUP	DSL
Architecture de référence du logiciel	Architecture du DSL

Tableau-A I-11 Mappage des artefacts

RUP	DSL
N/A	Concepts du domaine
Document d'architecture	Document d'architecture
Vision	Vision
Liste des risques	Liste des risques
Cas d'affaire	Cas d'affaire

Tableau-A I-12 Mappage des critères d'évaluation

RUP	DSL
La vision, les exigences et l'architecture sont stables	La vision, les exigences et l'architecture sont stables
Les méthodes de test et d'évaluation sont fiables	Les méthodes de test et d'évaluation sont fiables
Les plans d'itération de la phase de construction sont à un niveau de détail permettant le lancement du développement du logiciel	Les plans d'itération de la phase de construction sont à un niveau de détail permettant le lancement du développement du DSL

- **Phase de construction**

Tableau-A I-13 Mappage des objectifs

RUP	DSL
Compléter l'analyse, la conception, le développement et le test de toutes les fonctionnalités requises	Compléter l'analyse, la conception, le développement et le test de toutes les fonctionnalités requises
Mise en production de nouvelles versions (alpha, bêta, etc.) du logiciel	Mise en production de nouvelles versions (alpha, bêta, etc.) du DSL

Tableau-A I-14 Mappage des activités

RUP	DSL
N/A	Définir la syntaxe abstraite du DSL
N/A	Définir sa syntaxe concrète
N/A	Définir sa sémantique
Faires des tests	Faires des tests
Évaluer le produit logiciel du logiciel par rapport aux critères d'acceptation et la vision	Évaluer les éléments du DSL par rapport aux critères d'acceptation et la vision
Développer un manuel d'utilisation pour les utilisateurs du logiciel	Développer un manuel d'utilisation pour les utilisateurs du DSL

Tableau-A I-15 Mappage des jalons

RUP	**DSL**
Nouvelle mise à jour (Release) du logiciel	Nouvelle mise à jour (Release) du DSL

Tableau-A I-16 Mappage des artefacts

RUP	**DSL**
N/A	Syntaxe abstraite
N/A	Syntaxe concrète
N/A	Sémantique
Guide d'utilisation	Guide d'utilisation

Tableau-A I-17 Mappage des critères d'évaluation

RUP	**DSL**
La version livrée du logiciel est stable et prête à être déployée	La version livrée du DSL est stable et prête à être déployée

ANNEXE II

MODÉLISATION DE LA MÉTHODE AVEC LA NOTATION ISO/IEC 24744

Le langage de définition de méthode spécifié par la norme ISO/IEC 24744 utilise une approche semi-formelle textuelle sans offrir une notation graphique. Un complément a été proposé (ISO, 2007a) afin de doter la norme d'une notation graphique qui permet de simplifier son utilisation.

La notation vise particulièrement à représenter les concepts du domaine méthode comme les étapes (*Stages*), les unités de travail, les producteurs et les contraintes. À cette fin, elle définit des symboles graphiques et des diagrammes que l'ingénieur de méthode peut utiliser pour la modélisation de sa méthode.

Ce chapitre présente la mise en application de cette notation pour la modélisation de la méthode de définition de DSL décrite dans le chapitre 6. Les diagrammes présentés sont : le diagramme de cycle de vie, le diagramme de processus et le diagramme d'action (voir section 3.5 pour plus de détail).

Diagramme de cycle de vie

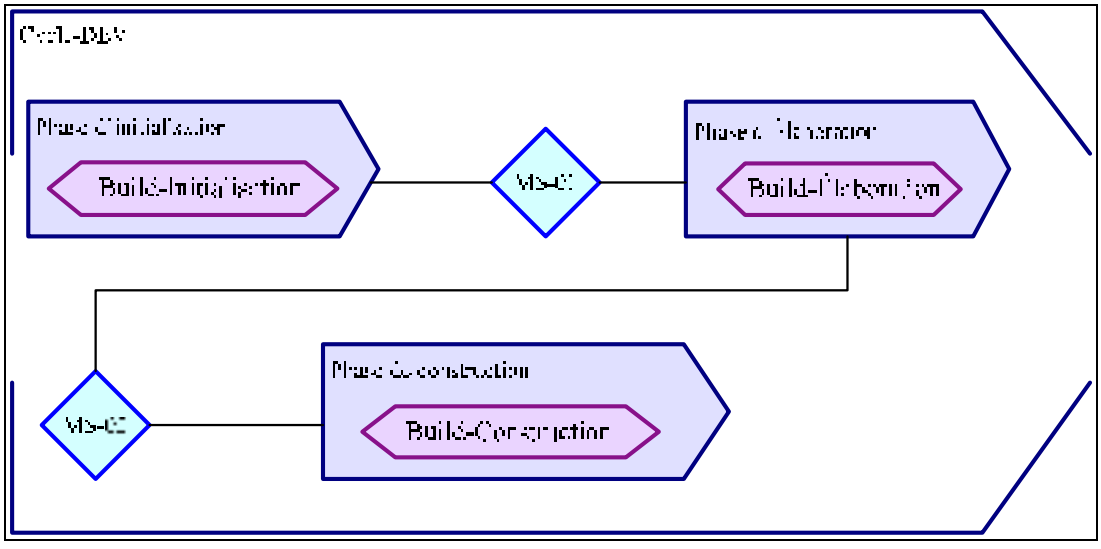

Figure-A II-1 Diagramme de cycle de vie.

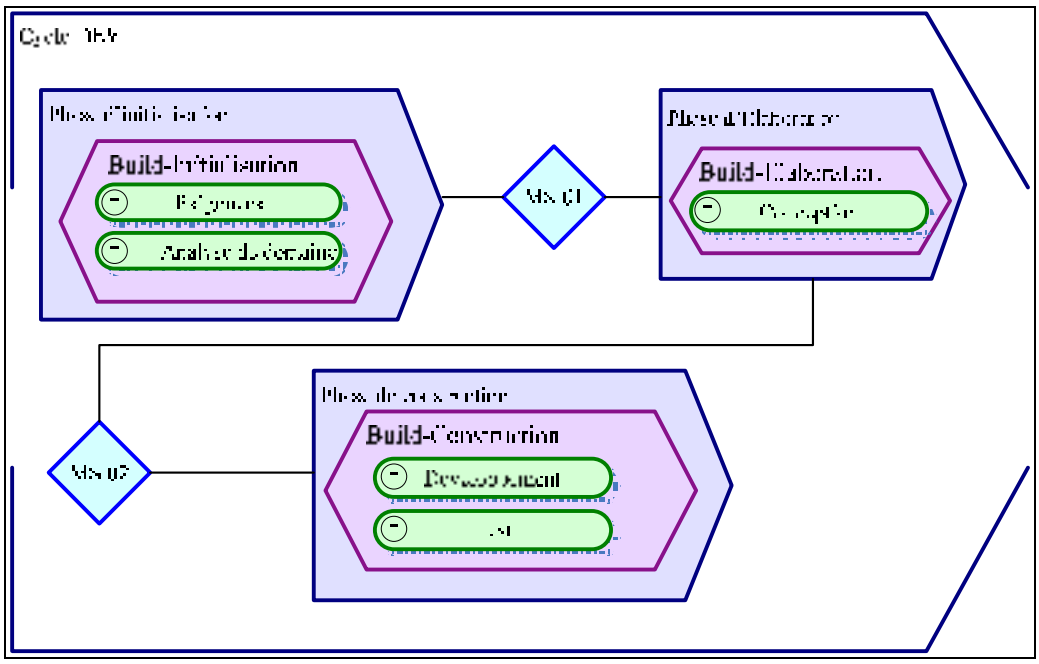

Figure-A II-2 Diagramme de cycle de vie détaillé.

Diagramme de processus

Un diagramme de processus représente les détails d'un type de processus (*ProcessKind*) ou d'une collection de types de processus.

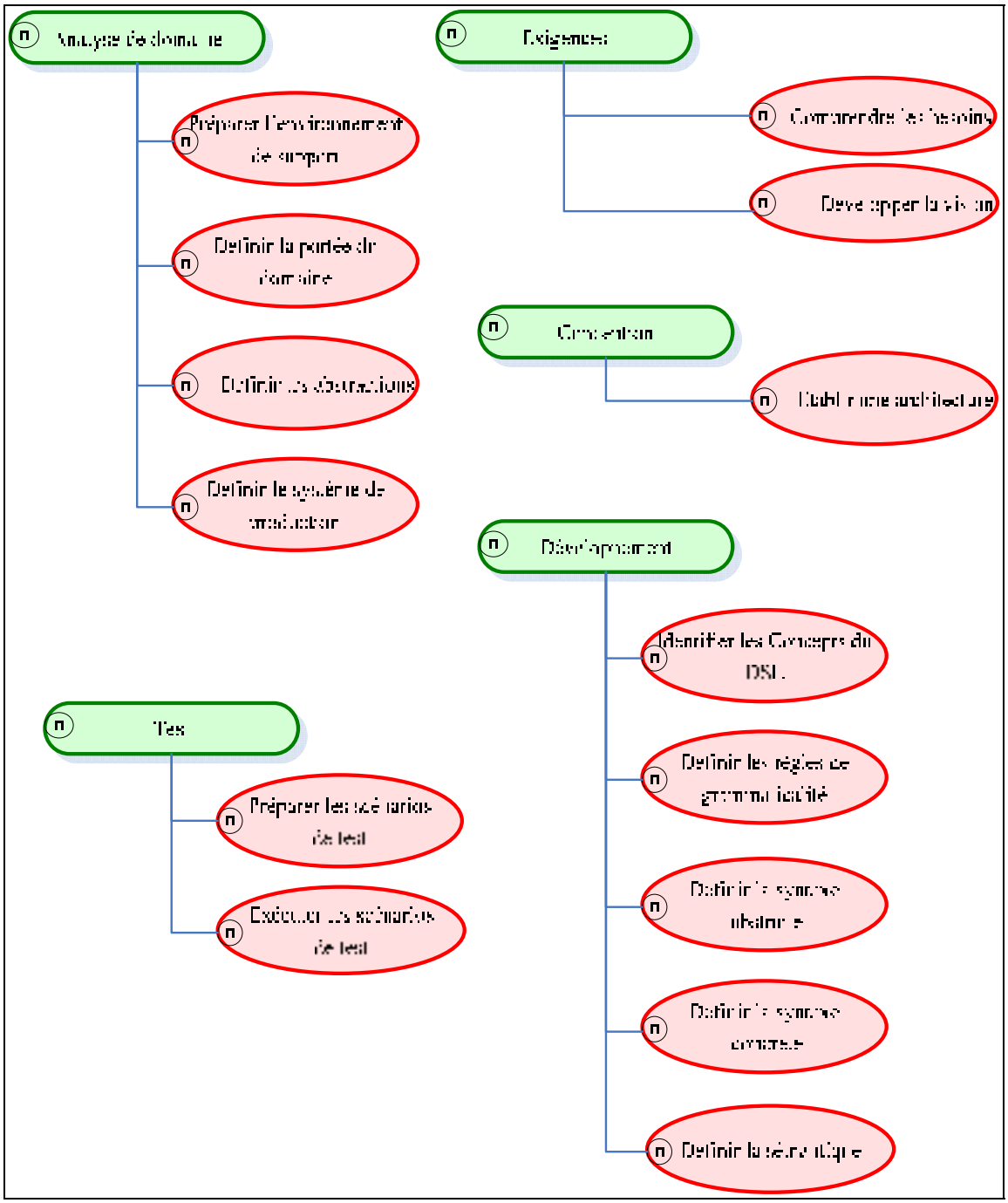

Figure-A II-3 Diagramme de processus.

Diagramme d'action

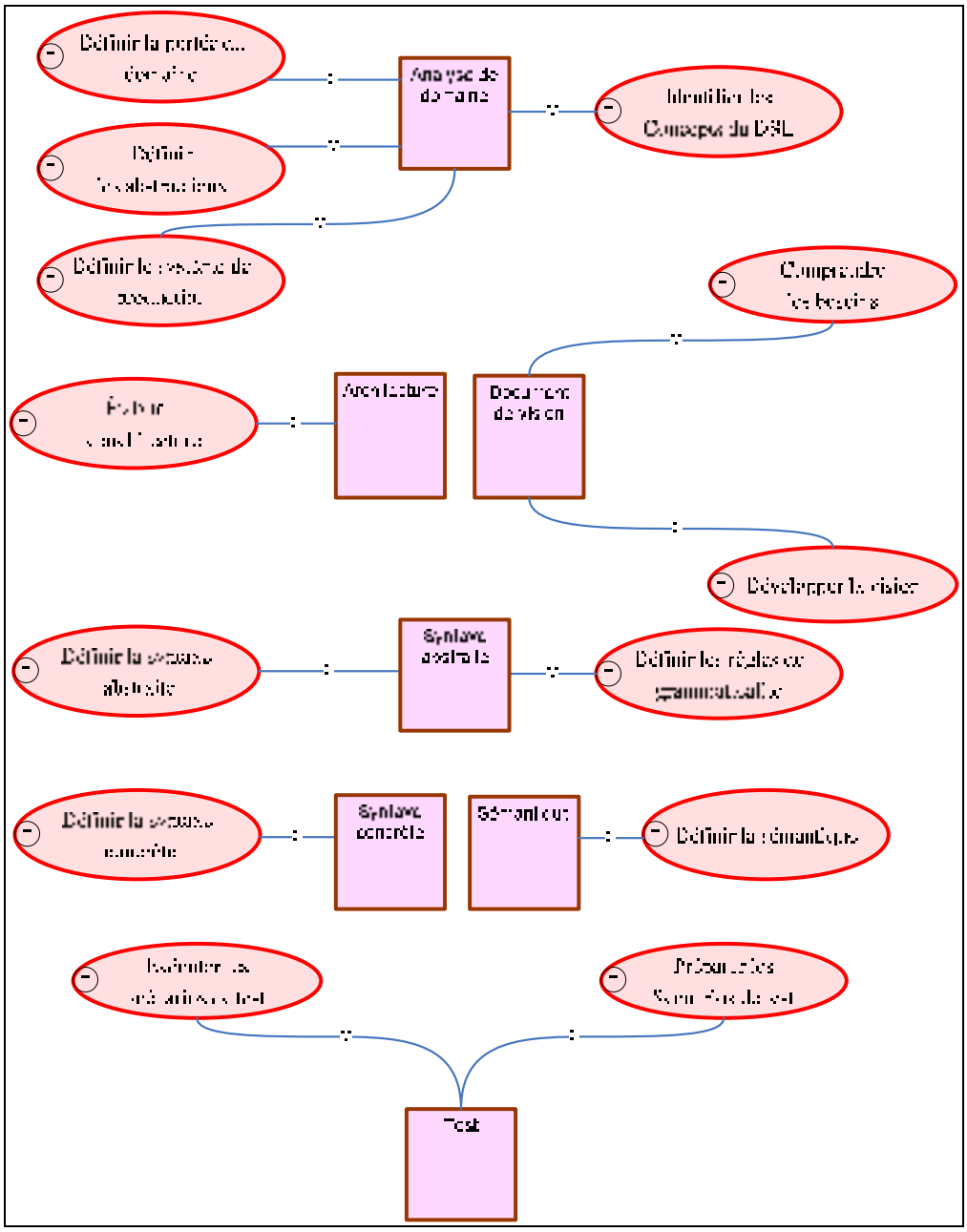

Figure-A II-4 Digramme d'action.

ANNEXE III

CONTEXTE D'UTILISATION DES LANGAGES WCL, DDL et MSL

Cette annexe présente le contexte général d'utilisation des trois DSL que nous avons redéfinis en utilisant la méthode qu'on propose pour la définition des langages dédiés, soit le DSL de spécification WCL, le DSL de modélisation DDL et le DSL de programmation MSL.

Le Tableau-A III-1 décrit le contexte général d'utilisation de ces trois langages.

Tableau-A III-1 Contexte général de l'utilisation des langages WCL, DDL et MSL

Contexte	**Description**
Contexte d'utilisation des langages WCL, DDL et MSL	Les DSL sont utilisés dans le cadre d'une ligne de produits logiciels développée par une entreprise afin de générer des sites Web dynamiques à partir de spécifications de réalisation.
Types de DSL	Les trois DSL sont des DSL propriétaires développés dans l'entreprise.
Taille de l'entreprise	L'entreprise compte près de 40 personnes
Équipe de développement	L'équipe de développement est composée de huit développeurs : - un chef d'équipe pour superviser l'équipe de développement; - un ingénieur recherche et développement pour maintenir les DSL et les modèles génériques de génération de code; - quatre ingénieurs conception et développement pour l'implémentation des personnalisations sur les sites Web; - deux développeurs pour l'intégration HTML/CSS/Flash.
Compétences de l'équipe	- Compétences en développement Web - Compréhension du système de production (ligne de produits) adopté par l'entreprise
Formation des développeurs	La durée moyenne de la formation d'un nouveau développeur sur la ligne de produits est de trois semaines.
Type de projets	Plusieurs types de sites Web sont réalisés avec cette ligne de produits logiciels. On y trouve des intranets, des extranets, des sites e-business, des vitrines, des catalogues et des blogues.
Taille des projets	La taille d'un projet est fonction de la taille de sa base de données et des modules qui y sont inclus. L'effort total du développement d'un projet est fonction du type et du nombre de personnalisations à apporter au site Web. Cet effort varie généralement entre 60 et 360 jours-personnes.

Nombre de développeurs par projet	Le développement d'un site Web implique généralement : - un à trois développeurs; - un intégrateur HTML/CSS; - un développeur Flash. Un développeur peut travailler simultanément sur plusieurs projets.
Code généré (en %)	Le code généré est estimé approximativement à : - 95% à 100% pour la partie back-office - 50% à 80% pour la partie front-office

Le Tableau-A III-2 présente une description des profils des développeurs utilisant les DSL.

Tableau-A III-2 développeurs utilisant les DSL pour le développement des sites web

	Fonction	**Nombre de projets réalisés**	**Années d'expérience**	**Ville (pays)**
Développeur 1	CE	20 à 30	4	Casablanca (Maroc)
Développeur 2	IRD	5 à 10	4	Casablanca (Maroc)
Développeur 3	ICD	15 à 20	5	Casablanca (Maroc)
Développeur 4	ICD	10 à 15	4	Casablanca (Maroc)
Développeur 5	ICD	5 à 10	4	Casablanca (Maroc)
Développeur 6	ICD	5 à 10	2	Casablanca (Maroc)
Développeur 7	DEV	5 à 10	3	Casablanca (Maroc)
Développeur 8	DEV	5 à 10	2	Casablanca (Maroc)

- **CE** Chef d'équipe ;
- **IRD** Ingénieur Recherche et développement ;
- **ICD** Ingénieur Conception et développement ;
- **DEV** Développeur.

BIBLIOGRAPHIE

Abbott, RJ. 1983. « Program design by informal English descriptions ». *Communications of the ACM,* vol. 26, n° 11, pp. 882-894.

Abelson, H., Sussman, GJ. et Sussman, J. 1996. *Structure and interpretation of computer programs*, 2nd edition. Cambridge, MA, USA: MIT Press, 657 p.

Adrion, W. Richards. 1993. « Research Methodology in Software Engineering, Summary of the Dagstuhl workshop on future directions in software engineering ». *ACM SIGSOFT Software Engineering Notes,* vol. 18, n° 1, pp. 35-48.

Alanen, M., et Porres, I. 2004. *A Relation Between Context-free Grammars and Meta Object Facility Metamodels*. Technical Report 606. Turku Centre for Computer Science.

America, P., Thiel, S., Ferber, S. et Mergel, M. 2001. *Introduction to Domain Analysis*. ESAPS project (Engineering Software Architectures, Processes, and Platforms for System Families), 25 p.

Antkiewicz, M., et Czarnecki, K. 2004. « FeaturePlugin: feature modeling plug-in for Eclipse ». In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange* (Vancouver, British Columbia, Canada, October 24-24). pp. 67-72.

Ardis, Mark A., et Green, Janel A. 1998. « Successful introduction of domain engineering into software development ». *Bell Labs Technical Journal,* vol. 3, n° 3, pp. 10-20.

Arnold, BRT., Van Deursen, A. et Res, M. 1995. « An algebraic specification of a language for describing financial products ». In *EEE Workshop on Formal Methods Application in Software Engineering* (April 1995). pp. 6–13.

Atkinson, C. 1999. « Supporting and applying the UML conceptual framework ». *LECTURE NOTES IN COMPUTER SCIENCE,* vol. 1618, pp. 21-36.

Atkinson, C, et Kühne, T. 2000. « Meta-level independent modelling ». In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming* (Sophia Antipolis and Cannes, France, June 12-16). pp. 12-16.

Atkinson, C, et Kühne, T. 2002. « Rearchitecting the UML infrastructure ». *ACM Transactions on Modeling and Computer Simulation (TOMACS),* vol. 12, n° 4, pp. 290-321.

Atkinson, C. 1997. « Meta-modelling for distributed object environments ». In *Enterprise Distributed Object Computing Workshop [1997]. EDOC '97. Proceedings. First International* (Gold Coast, Qld. , Australia, 24-26 Oct). pp. 90-101.

Atkinson, C., et Kuhne, T. 2003. « Model-driven development: a metamodeling foundation». *Software, IEEE,* vol. 20, n° 5, pp. 36-41.

Atkinson, Colin, et Kühne, Thomas. 2001. « The Essence of Multilevel Metamodeling ». In *«UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. pp. 19-33. <http://dx.doi.org/10.1007/3-540-45441-1_3>.

Bachmann, F, et Bass, L. 2001. « Managing variability in software architectures ». *ACM SIGSOFT Software Engineering Notes,* vol. 26, n° 3, pp. 126-132.

Bezivin, J., et Heckel, R. 2004. « Language Engineering for Model-driven Software Development ». In *Dagstuhl Seminar 04101* (Dagsthul, Germany, February 29th to April 5th). pp. 1-8.

Bézivin, Jean. 2004. « In Search of a Basic Principle for Model Driven Engineering ». *The European Journal for the Informatics Professional,* vol. 5, n° 2, pp. 21-24.

Booch, Grady. 2004. « Microsoft and Domain Specific Languages ». En ligne. <http://www.ibm.com/developerworks/forums/thread.jspa?threadID=67637>. Consulté le 8 Février 2011.

Boulanger, RC. 2000. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. Cambridge, MA, USA: MIT press, 740 p.

Bühne, S., Halmans, G. et Pohl, K. 2003. « Modelling dependencies between variation points in use case diagrams ». In *Proceedings of the 9th Workshop in Requirements Engineering - Foundations for Software Quality* (Klagenfurt, Austria,). pp. 59–69.

Chamberlin, DD., et Boyce, RF. 1974. « SEQUEL: A structured English query language ». In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control* (Ann Arbor, Michigan, May 01-03). pp. 249-264. New York, NY, USA: ACM.

Chen, K., Sztipanovits, J. et Neema, S. 2005. « Toward a semantic anchoring infrastructure for domain-specific modeling languages ». In *Proceedings of the 5th ACM international conference on Embedded software* (Jersey City, NJ, USA, September 19 - 22). pp. 35-43. In *ACM*. New York, NY, USA.

Chomsky, N. 1956. « Three models for the description of language ». *IRE Transactions on information theory,* vol. 2, n° 3, pp. 113-124.

Chomsky, N. 1959. « On certain formal properties of grammars ». *Information and control,* vol. 2, n° 2, pp. 137-167.

Clark, T., Sammut, P. et Willans, J. 2008. *Applied Metamodelling A Foundation For Language Driven Development*, Second Edition. CETEVA, 241 p.

Cleaveland, J.C. 1988. « Building application generators ». *Software, IEEE,* vol. 5, n° 4, pp. 25-33.

Clements, Paul C., et Northrop, Linda. 2002. *Software product lines practices and patterns*. Coll. « SEI series in software engineering ». Boston: Addison-Wesley, 563 p.

COAD, P. 1992. « Object-oriented patterns ». *Communications of the ACM,* vol. 35, n° 9, pp. 152-159.

Coad, P., de Luca, J. et Lefebvre, E. 1999. *Java Modeling Color with Uml: Enterprise Components and Process*. Prentice Hall PTR Upper Saddle River, NJ, USA, 221 p.

Coad, Peter, North, David et Mayfield, Mark. 1997. *Object models (2nd ed.): strategies, patterns, and applications*. Upper Saddle River, NJ: Prentice Hall, 515 p.

Coplien, J., Hoffman, D. et Weiss, DM. 1998. « Commonality and variability in software engineering ». *Software, IEEE,* vol. 15, pp. 37-45.

Creps, D., Tracz, W., Payton, T. et Webb, M. 1996. *Domain Engineering Handbook*. Tech. rep., Lockheed Martin Software and Systems Resource Center, 1996.

Czarnecki, K. 2004. « Overview of generative software development ». In *Proceedings of Unconventional Programming Paradigms (UPP) 2004* (Mont Saint-Michel, France, 15-17 September). Vol. 3566 of LNCS, pp. 313-328. Springer-Verlag.

Czarnecki, K. 2002. « Domain Engineering ». *Encyclopedia of Software Engineering, 2nd Edition*. pp. 433–444.

Czarnecki, K., Eisenecker, U., Gluck, R., Vandevoorde, D. et Veldhuizen, T. 2000. « Generative Programming and Active Libraries ». *Selected Papers from the International Seminar on Generic Programming*. pp. 25-39.

Czarnecki, K., et Eisenecker, Ulrich. 2000. *Generative programming : methods, tools, and applications*. Boston: Addison Wesley, 832 p.

Czarnecki, K., et Eisenecker, Ulrich W. 1999 «Components and generative programming (invited paper) ». In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on*

Foundations of software engineering (Toulouse, France, September 06-10). pp. 2-19 Springer-Verlag, London, UK.

Czarnecki, K., et Helsen, S. 2003. « Classification of Model Transformation Approaches ». In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture* (Anaheim, CA, USA, October 2003).

Czarnecki, K., et Helsen, S. 2006. « Feature-based survey of model transformation approaches ». *IBM Systems Journal,* vol. 45, n° 3, pp. 621-645.

Czarnecki, K., Helsen, S. et Eisenecker, U. 2004. « Staged Configuration Using Feature Models ». *SOFTWARE PRODUCT LINES, Lecture Notes in Computer Science,* vol. 3154, pp. 266-283.

Czarnecki, K., O'DONNELL, JT., Striegnitz, J. et Taha, W. 2004. « DSL implementation in MetaOCaml, Template Haskell, and C++ ». *DOMAIN-SPECIFIC PROGRAM GENERATION, Lecture Notes in Computer Science,* vol. 3016, pp. 51-72.

Deursen, Arie van, Klint, P. et Visser , J. 2000 «Domain-specific languages: an annotated bibliography ». *ACM SIGPLAN Notices* vol. 35 n° 6 pp. 26-36

Dijkstra, Edsger Wybe. 1997. *A discipline of programming*. Prentice Hall PTR Upper Saddle River, NJ, USA 240 p.

Eisenecker, UW. 1997. « Generative Programming (GP) with C++ ». In *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages* (March 19-21, 1997). pp. 351-365. Springer-Verlag London, UK.

Eisenhardt, KM. 1989. « Building theories from case study research ». *Academy of management review,* vol. 14, n° 4, pp. 532-550.

Fondement, F., et Baar, T. 2005. « Making metamodels aware of concrete syntax ». *Lecture Notes in Computer Science*. Vol. 3748, pp. 190-204. In Springer. Berlin Heidelberg.

Fontoura, M., Pree, W. et Rumpe, B. 2000. *The Uml Profile for Framework Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 240 p.

Fowler, M. 2003. « UmlMode ». In *Martin Fowler*. En ligne. <http://martinfowler.com/bliki/UmlMode.html>. Consulté le 31 Janvier 2011.

Fowler, Martin. 1997. *Analysis patterns reusable object models*. Coll. « The Addison-Wesley object technology series ». Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 357 p.

Fowler, Martin. 2011. « Domain Specific Languages ». In *Martin Fowler*. En ligne. <http://martinfowler.com/bliki/DomainSpecificLanguage.html>. Consulté le 31 Janvier 2011.

Frankel, DS. 2003. *Model driven architecture: Applying MDA to Enterprise Computing*. Indianapolis, Indiana, USA: Wiley New York, 352 p.

Free Software Foundation, Inc. 2009. « Bison - GNU parser generator ». In *The GNU Operating System*. En ligne. <http://www.gnu.org/software/bison/>. Consulté le 10 Janvier 2011.

Friedl, J. 2006. *Mastering regular expressions*, Third Edition. Sebastopol, CA, USA: O'Reilly Media, Inc., 516 p.

Fuentes-Fernández, L., et Vallecillo-Moreno, A. 2004. « An Introduction to UML Profiles ». *The European Journal for the Informatics Professional,* vol. 5, n° 2, pp. 5-13.

Garshol, LM. 2010. « BNF and EBNF: What are they and how do they work? ». En ligne. <http://www.garshol.priv.no/download/text/bnf.html>. Consulté le 10 Janvier 2011.

Gonzalez-Perez, C, et Henderson-Sellers, B. 2007a. « Modelling software development methodologies: A conceptual foundation ». *Journal of Systems and Software,* vol. 80, n° 11, pp. 1778-1796.

Gonzalez-Perez, C. 2007. « Supporting Situational Method Engineering with ISO/IEC 24744 and the Work Product Pool Approach ». Vol. 244, pp. 7-18. In Ralyté, J., Brinkkemper, S., and Henderson-Sellers, B. (Eds.): Situational Method Engineering -- Fundamentals and Experiences. Springer, Boston.

Gonzalez-Perez, C., et Henderson-Sellers, B. 2007b. « Modelling software development methodologies: A conceptual foundation ». *The Journal of Systems & Software,* vol. 80, n° 11, pp. 1778-1796.

Greenfield, Jack, et Short, Keith. 2004a. « Software factories Industrialized Software Development ». <http://www.softwarefactories.com/index.html>. Consulté le 10 Novembre 2008.

Greenfield, Jack, et Short, Keith. 2004b. *Software factories: assembling applications with patterns, models, frameworks, and tools*, 1st Edition. Indianapolis, Indiana: Wiley Publishing Inc., 500 p.

Hansson, DH. 2011. *Ruby on Rails*. En ligne. <http://www.rubyonrails.org/>. Consulté le 4 Février 2011.

Harel, D., et Rumpe, B. 2000. *Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff*. Technical Report MCSOO-16. Israel, Israel: Weizmann Science Press of Israel, 28 p.

Harsu, M. 2002. *FAST product-line architecture process*. Report 29. Tampere University of Technology: Institute of Software Systems, 45 p.

Henderson-Sellers, Brian, et Gonzalez-Perez, Cesar. 2005. « The rationale of powertype-based metamodelling to underpin software development methodologies ». In *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling* (Newcastle, New South Wales, Australia, January 01, 2005). Vol. 43, pp. 7-16. Darlinghurst, Australia, Australia: Australian Computer Society, Inc.

Hucka, M., et al. 2003. « The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models ». *Bioinformatics,* vol. 19, n^{o} 4, pp. 524-531.

Hudak, P. 1996. « Building domain-specific embedded languages ». *ACM Computing Surveys (CSUR) - Special issue: position statements on strategic directions in computing research*. Vol. 28, n^{o} 4es. In ACM. New York, NY, USA.

Hudak, P. 1998. « Modular domain specific languages and tools ». In *Proceedings of the 5th International Conference on Software Reuse* (Los Alamitos, CA, June 02-05). pp. 134-142. In *EEE Computer Society*. Washington, DC, USA.

ISO. 1986. *Standard Generalized Markup Language (SGML)*. Standard, ISO 8879:1986 International Organization for Standardization, 155 p.

ISO. 1996. *Syntactic metalanguage -- Extended BNF*. Standard, ISO/IEC 14977:1996. International Organization for Standardization, 12 p.

ISO. 2001. *Software engineering -- Product quality -- Part 1: Quality model*. Standard, ISO/IEC 9126-1:2001. International Organization for Standardization, 25 p.

ISO. 2005. *XML Metadata Interchange (XMI)*. ISO/IEC 19503:2005. International Organization for Standardization, 115 p.

ISO. 2007a. *NWI Proposal - Amendment to ISO/IEC 24744 .Software Engineering - Metamodel for Development Methodologies: addition of an informative annex for notation*. International Organization for Standardization, 22 p.

ISO. 2007b. *Software Engineering -- Metamodel for Development Methodologies*. Standard, ISO/IEC 24744:2007. International Organization for Standardization, 78 p.

ISO. 2008. *Database languages -- SQL -- Part 2: Foundation (SQL/Foundation)*. ISO/IEC 9075:2008. International Organization for Standardization, 1317 p.

Ivar, J., Martin, G. et Patrik, J. 1997. *Software reuse: Architecture, process and organization for business success*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 512 p.

Jaring, M., et Bosch, J. 2004. « Architecting product diversification - formalizing variability dependencies in software product family engineering ». In *Proceedings. Fourth International Conference on Quality Software (QSIC)* (Braunschweig, Germany, September 08-10, 2004). pp. 154-161. In *IEEE Computer Society*. Washington, DC, USA.

Johnson, SC. 1974. « YACC-yet another compiler-compiler ». *UNIX Programmer's Manual* (Bell Laboratories, Murray Hill, NJ).

Jones, SP. 2002. « Haskell 98 language and libraries: the revised report ». En ligne. <http://www.haskell.org/onlinereport/>. Consulté le 3 Février 2011.

Kaisler, Stephen H. 2005. *Software paradigms*, 1st Edition. Hobocen, New Jersey: John Wiley & Sons, 440 p.

Kang, K. , Cohen, S. , Hess, J. , Novak, W. et Peterson, A. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. En ligne. Tech. rep. CMU/SEI-90-TR-21. Carnegie Mellon University: Software Engineering Institute. <http://www.sei.cmu.edu/publications/documents/90.reports/90.tr.021.html>. Consulté le 2 Février 2011.

Karsai, G., Nordstrom, G., Ledeczi, A. et Sztipanovits, J. 2000. « Specifying graphical modeling systems using constraint-based meta models ». In *IEEE Symposium on Computer Aided Control System Design* (Anchorage, AK , USA, Sep 25-27). pp. 89-94.

Kelly, S., et Tolvanen, JP. 2008. *Domain-Specific Modeling*. Hobocen, New Jersey: John Wiley & sons, Inc., 448 p.

Kent, Stuart. 2002. « Model Driven Engineering ». In *Proceedings of the Third International Conference on Integrated Formal Methods* (Turku, Finland, May 15-18, 2002). pp. 286-298. London, UK: Springer-Verlag.

Kleppe, A. 2008. *Software language engineering: creating domain-specific languages using metamodels*, 1st Edition. Addison-Wesley Professional, 240 p.

Kleppe, Anneke G., Warmer, Jos B. et Bast, Wim. 2003. *MDA explained: the model driven architecture : practice and promise*. Coll. « Addison-Wesley object technology series ». Boston: Addison-Wesley, 170 p.

Knuth, DE. 1964. « backus normal form vs. Backus Naur form ». *Communications of the ACM*. Vol. 7, n° 12, pp. 735-736. In ACM. New York, NY, USA.

Kossatchev, AS., et Posypkin, MA. 2005. « Survey of compiler testing methods ». *Programming and Computer Software*. Vol. 31, n° 1, pp. 10-19. In Plenum Press. New York, NY, USA.

Kramer, J. 2007. « Is abstraction the key to computing? ». *Communications of the ACM*. Vol. 50, n° 4, pp. 36-42. In ACM. New York, NY, USA.

Kurtev, I., Bézivin, J. et Aksit, M. 2002. « Technological Spaces: an Initial Appraisal ». In *Proceedings of the confederated international conferences CoopIS, DOA, and ODBASE* (Irvine, CA, USA, 2002).

Larman, C. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd Edition. Upper Saddle River, NJ, USA: Prentice Hall, 736 p.

Lee, Kwanwoo, Kang, Kyo et Lee, Jaejoon. 2002. « Concepts and Guidelines of Feature Modeling for Product Line Software Engineering ». In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*. pp. 62-77. London, UK: Springer-Verlag.

Lesk, ME. 1975. *Lex: A lexical analyzer generator*. Computing Science Technical Report 39. Bell Laboratories Murray Hill, NJ, 11 p.

Liberty, J., et Horovitz, A. 2008. *Programming. NET 3.5*. O'Reilly Media, Inc., 476 p.

Martin, J. 1967. *Design of Real-Time Computer Systems*, 1st Edition. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 629 p.

McCarthy, J. 1965. *LISP 1.5 programmer's manual*. Cambridge, MA, USA: The MIT Press, 112 p.

Mernik, Marjan, Heering, Jan et Sloane, Anthony M. 2005. « When and how to develop domain-specific languages ». *ACM Computing Surveys (CSUR)*. Vol. 37, n° 4, pp. 316-344.

Microsoft. 2011. *Microsoft Excel 2010*. En ligne. <http://office.microsoft.com/en-ca/excel/>. Consulté le 31 Janvier 2011.

Murray-Rust, P. 1997. « Chemical markup language ». *World Wide Web Journal - Special issue on XML: principles, tools, and techniques*. Vol. 2, n° 4, pp. 135-147.

Muthig, D., et Atkinson, C. 2002. « Model-driven product line architectures ». In *Proceedings of the Second International Conference on Software Product Lines* (San Diego, CA, August, 19-22). Vol. 2379, pp. 110-129. London, UK: Springer-Verlag.

Northrop, Linda M. 2002. « SEI's software product line tenets ». In *IEEE Software*. Vol. 19, n° 4, pp. 32-40. Los Alamitos, CA, USA: IEEE Computer Society Press.

Odell, J. 1994. « Power types ». *Journal of Object-Oriented Programming*. Vol. 7, n° 2 (May 1994), pp. 8-12.

Office québécois de la langue française. 2002. *Grand dictionnaire terminologique*. En ligne. <http://www.oqlf.gouv.qc.ca/>. Consulté le 31 Janvier 2011.

OMG. 1997. *UML 1.0 : UML Semantics Appendix M1 : UML Glossary*. Rational Software Corporation.

OMG. 2005a. « Catalog of UML Profile Specifications ». En ligne. OMG. <http://www.omg.org/technology/documents/profile_catalog.htm>. Consulté le 31 Janvier 2011.

OMG. 2005b. *XML Metadata Interchange V 2.1*. En ligne. formal/2005-09-01. OMG, 120 p. <http://www.omg.org/spec/XMI/2.1/PDF/>. Consulté le 31 Janvier 2011.

OMG. 2006a. *Object Constraint Language V2.0*. En ligne. formal/2006-05-01. 232 p. <http://www.omg.org/spec/OCL/2.0/PDF/>. Consulté le 30 Janvier 2011.

OMG. 2006b. *Unified Modeling Language: Diagram Interchange v 2.0*. En ligne. formal/2006-04-04. OMG, 36 p. <http://www.omg.org/spec/UMLDI/1.0/PDF>. Consulté le 31 Janvier 2010.

OMG. 2008. *Query/View/Transformation, v1.0* En ligne. formal/08-04-03. OMG, 240 p. <http://www.omg.org/spec/QVT/1.0/PDF/>. Consulté le 31 Janvier 2011.

OMG. 2009. *UML 2.2 Infrastructure Specification*. En ligne. formal/2009-02-04, 226 p. <http://www.omg.org/cgi-bin/doc?ptc/2003-09-15>. Consulté le 31 Janvier 2010.

Parnas, David. 1976. « On the Design and Development of Program Families' ». *IEEE Transactions on Software Engineering,* vol. 2, n° 1 (March 1976), pp. 1-9.

Parr, T. 2007. *The definitive ANTLR reference: building domain-specific languages*. Coll. « The pragmatic programmers ». Pragmatic Bookshelf Raleigh, NC, 369 p.

Paxson, V. 1988. « Flex–fast lexical analyzer generator ». *Free Software Foundation*.

Prieto-Diaz, R. 1990. « Domain analysis: an introduction ». *SIGSOFT Software Engineering Notes*. Vol. 15, n° 2, pp. 47-54. In ACM. New York, NY, USA.

Robert, F., et Bernhard, R. 2007. « Model-driven Development of Complex Software: A Research Roadmap ». In *2007 Future of Software Engineering* (Minneapolis, MN, May 23-25, 2007). pp. 37-54. IEEE Computer Society. In *IEEE Computer Society*. Washington, DC, USA.

Sadilek, DA., et Wachsmuth, G. 2008. « Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages ». In *Proceedings of Model-Drivern Architecture-Foundations and Applications: 4th European Conference, Ecmda-Fa 2008* (Berlin, Germany, June 9-13, 2008). Springer.

Schmid, Klaus. 2002. « A comprehensive product line scoping approach and its validation ». In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida, USA, May 19-25). pp. 593-603. ACM.

Schmidt, D. C. 2006. « Guest Editor's Introduction: Model-Driven Engineering ». *Computer*. Vol. 39, n° 2, pp. 25-31. In IEEE Computer Society Press.

Schmidt, David A. 1986. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, 331 p.

Selic, B. 2003. « The pragmatics of model-driven development ». *IEEE software,* vol. 20, n° 5, pp. 19-25.

Selic, B. 2007. « A Systematic Approach to Domain-Specific Language Design Using UML ». In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*. pp. 2-9.

Sheard, T., et Jones, SP. 2002. « Template metaprogramming for Haskell, ACM SIGPLAN Haskell Workshop 02 (Manuel MT Chakravarty, ed.) ». In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (Pittsburgh, Pennsylvania , October 03). pp. 1-16. ACM Press.

Simos, Mark. 1995. « Organization domain modeling (ODM): formalizing the core domain modeling life cycle ». In *Proceedings of the 1995 Symposium on Software reusability* (Seattle, Washington, United States, April 29-30). pp. 196-205.

Simos, Mark, Creps, Dick, Klingler, Carol, Levine, Larry et Allemang, Dean. 1996. *Organization Domain Modeling (ODM) Guidebook, Version 2.0*. Software Technology For Adaptable, Reliable Systems (STARS).

Slonneger, K., et Kurtz, BL. 1995. *Formal syntax and semantics of programming languages*. Addison-Wesley, 550 p.

Sprinkle, Jonathan, et Karsai, G.Gabor. 2004. « A domain-specific visual language for domain model evolution ». *Journal of Visual Languages & Computing,* vol. 15, n° 3-4 (2004/0), pp. 291-307.

Steele, GL. 1990. *Common LISP: the language*, 2nd Edition. Digital Press, 1029 p.

Strachan, James, et McWhirter, Bob. 2011. *Groovy : An agile dynamic language for the Java Platform*. En ligne. <http://groovy.codehaus.org/>. Consulté le 30 Janvier-2011.

Stropky, Maria E., et Laforme, Deborah. 1995 «An automated mechanism for effectively applying domain engineering in reuse activities ». In *Proceedings of the conference on TRI-Ada '95: Ada's role in global markets: solutions for a changing complex world* (Anaheim, California, United States). pp. 332-340 ACM Press.

Thibault, S.A., Marlet, R. et Consel, C. 1999. « Domain-specific languages: from design to implementation application to video device drivers generation ». *Software Engineering, IEEE Transactions on,* vol. 25, n° 3, pp. 363-377.

Tolvanen, JP. 2006. « Domain-Specific Modeling: How to Start Defining Your Own Language ». *DevX. com, article accessible sur le site de l'entreprise à: http://www.devx.com/enterprise/Article/30550,* vol. 1, n° 09, pp. 07.

Tolvanen, JP., et Kelly, S. 2001. « Modelling languages for product families: a method engineering approach ». In *Proceedings of OOPSLA Workshop on Domain-specific Visual Languages* (Tampa, FL, USA, 2001). pp. 135–140.

Tolvanen, JP., et Rossi, M. 2003. « Metaedit+: defining and using domain-specific modeling languages and code generators ». In *OOPSLA '03 Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Anaheim, CA, USA, October 26-30). pp. 92-93. ACM New York, NY, USA.

Tracz, W. 1994. « Domain-specific software architecture (DSSA) frequently asked questions (FAQ) ». *ACM SIGSOFT Software Engineering Notes,* vol. 19, n° 2, pp. 52-56.

van der Linden, F. 2002. « Software product families in Europe: the Esaps & Cafe projects ». *Software, IEEE,* vol. 19, n° 4, pp. 41-49.

Van Deursen, A., et Klint, P. 1998. « Little languages: little maintenance? ». *Journal of Software Maintenance,* vol. 10, n° 2, pp. 75-92.

Watson, Andrew. 2008. *UML vs. DSLs: A false dichotomy*. En ligne. OMG, 3 p. <http://www.omg.org/docs/omg/08-09-03.pdf>.

Weiss, DM. 1998. « Commonality Analysis: A Systematic Process for Defining Families ». In *Development and Evolution of Software Architectures for Product Families*. pp. 214-222. <http://dx.doi.org/10.1007/3-540-68383-6_30>.

Weiss, DM., et Ardis, MA. 1997. « Defining families: The commonality analysis ». In *Proceedings of the 19th international conference on Software engineering* (Boston, Massachusetts, USA, May 17-23). pp. 649-650.

Weiss, DM., et Lai, Chi Tau Robert. 1999. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 426 p.

Wile, D. 2004. « Lessons learned from real DSL experiments ». *Sci. Comput. Program.* Vol. 51, n^{o} 3, pp. 265-290.

Xia, Y., et Glinz, M. 2002. *A Syntax Definition Method for Visual Specification Languages*. En ligne. IFI/UNIZH Technical Report 01-2002. University of Zurich. <http://www.ifi.unizh.ch/req/ftp/syntax/syntax.pdf>. Consulté le 31 Janvier 2011.

Xia, Y., et Glinz, M. 2003. « Rigorous EBNF-based definition for a graphic modeling language ». In *Software Engineering Conference, 2003. Tenth Asia-Pacific*. pp. 186-196.