REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSENGEMENT SUPERIEUR

Université SAAD DAHLEB-Blida

Faculté des Sciences

Département Informatique

En vu d'obtenir le diplôme de Master 2 en Informatique

Thème:

Mise en place d'un atelier génie logiciel AGL : pour la transformation des diagrammes de comportement UML et la génération de code source Java EE En Utilisant la démarche MDA

Organisme d'Accueil: ***IconSoftware***

Présenté par :

OUMEDDI Meriem
MAMMERI Khadidja

Encadrer par :

Mr. Boussabat Khaled

Promotrice:

Mme. Chikhi imane

Soutenu le :

Devant le Jury composé de :

Président :

Rapporteurs :

Examinateurs :

Promotion: 2012-2013

Remerciements

Quelques lignes ne pourront jamais exprimer la reconnaissance que nous éprouvons envers tous ceux qui, de prés ou de loin, ont contribué, par leurs conseils, leurs encouragements ou leurs amitiés à l'aboutissement de ce travail.

Nos vifs remerciements accompagnés de toute notre gratitude vont tout d'abord à notre Co-promoteur Mr K. BOUSSEBET, pour nous avoir proposé ce sujet, pour les conseils qu'il n'a cessé de nous prodiguer et surtout pour la confiance qu'il nous a accordé pour la réalisation de ce projet.

A Mm M.ADEM BOUSSEBET pour ses conseils et ses encouragements durant toute la période de réalisation de ce projet.

A toute l'équipe de l'entreprise ***Icon Software****.*

A notre promotrice Mm I.Chicki pour ces précieux conseils et de nous avoir dirigé durant notre projet.

Notre reconnaissance va particulièrement à tous nos enseignants et membres du jury pour avoir accepté de juger ce travail et d'honorer notre soutenance par leur présence.

Nous tenons aussi à témoigner notre reconnaissance à tous ceux qui ont participé de prés ou de loin à la réalisation de ce travail,

Khadîdja et Meriem

Dédicaces

Aux témoignages d'affection, d'amour et de grande reconnaissance, aux êtres les plus chers que j'ai dans la vie, qui m'ont toujours été présent dans tous les moments, et qui m'ont soutenu avec tous ce qu'ils ont.

…Mes parents,

A ceux qui ont su me consolider durant les moments les plus difficiles de ma vie

… Mes frères et ma sœur,

A mon fils Louai Abd Alwadoud,

A tous les miens dont mes oncles, mes tentes et à toute ma famille,

A tous mes amis et tous ceux qui me sont chers…

A tous ceux qui veillent les nuits pour faire jaillir la lumière de savoir,

A mon binôme Meriem et sa famille,

Et à toi,

Je dédié le fruit de mes années d'étude

Khadidja

Dédicace

A qui a consacré sa vie à faire mon bonheur, et qui a su guider mes pas d'enfance

Vers ce que je suis devenue aujourd'hui, mon très cher père.

A celle qui m'abreuve d'amour et d'affection intarissable, source de mon Bonheur.

Qui m'a enseigné que l'amour et la patience sont bien la clé du bonheur

Et de la réussite et ma raison d'être, ma très chère mère.

A mes adorables grands parents que Dieu me les garde encore aussi longtemps

A mes adorables frères, et mes douces sœurs,

A mes oncles, tantes, cousins et cousines...et toute ma famille.

A mes chère nièces et neveux

A mon binôme Khadîdja qui, par sa gentillesse et sa patience a énormément

Facilité la naissance du fruit de notre travail,

A toute sa famille en particulier le petit charmant Louai Abd Alwadoud.

A toutes mes ami(e)s, pour les meilleurs moments que j'ai passé avec eux

A tout le groupe des Etudiants Master2 Informatique,

A tout le groupe des employés du département Informatique et surtout

Ma chère sœur Salima.

Je n'oublierai pas pour autant, tous ceux qui sont absents sur cette feuille

Mais toujours présents dans mon cœur

Merci pour tous

Meriem

ملخص:

في هذه المذكرة سنتطرق الى موضوع هام من اجل التوسع في كل من العالم الأكاديمي وعالم صناعة البرمجيات استنادا على هندسة النماذجMDE فإنه يجعل تغييرا كبيرا في تصميم وتنفيذ تطبيقات المؤسسة. و الذى يستند على مجال هندسة البرمجيات MDAو هي التى تسمح بتحويل النماذج لتحقيق حل تقني على أي منصة تطبيق.
هذا العمل هو تطوير ورشة عمل شاملة AGL لمشاريع تطوير البرمجيات وفقا لنهج MDA في مؤسسة IconSoftware.
وتقدم هذه المذكرة بالتفصيل مختلف مراحل التحول من نماذج UML الى java EE.

Résumé Ce travail s'inscrit dans le domaine du Génie Logiciel. Nous nous intéressons plus particulièrement à l'ingénierie dirigée par les modèles (IDM) en adoptant l'architecture dirigée par les modèles (MDA) pour le développement de logiciels. Nous avons mis en place un Atelier de Génie Logiciel (AGL) pour le développement de logiciels selon la démarche MDA au sein de l'entreprise IconSoftware. L'AGL développé permet la génération de code source Java EE à partir d'un certain nombre de transformation des modèles UML. Notre apport, par rapport à l'AGL existant au sein de l'entreprise IconSoftware consiste à développer un AGL qui prend en considération l'ensemble des digrammes UML de structuration ainsi que les diagrammes de comportements.

Mots-Clés : Génie logiciel, Ingénierie dirigée par les modèles IDM, Atelier Génie logiciel (AGL), Architecture dirigée par les modèles MDA, Transformation de modèle UML, Génération de code source, java EE.

Abstract This work is in the field of Software Engineering. We are particularly interested in the model-driven engineering (MDE) adopting the model driven architecture (MDA) for the software development. We have established a Software Engineering Workshop (AGL) for software development according to MDA approach within the company IconSoftware. The developed AGL allows the generation of source code Java EE from a number of transformation of UML models. Our contribution, in relation to the existing AGL within the company IconSoftware , consists in developping an AGL which takes into consideration UML diagrams of structuration as well as the UML digrams of behavior.

Tags: Software Engineering, Model Driven Engineering, Workshop Software Engineering (CASE), models Driven Architecture MDA, Transformation UML model, source code generation, Java EE.

SOMMAIRE

PARTIE III : Mise en œuvre de l'AGL

Table des Figures

Liste des Tables

Introduction Générale

1. Contexte générale

Le terme **Génie Logiciel** (en anglais Software Engineering) désigne l'ensemble des méthodes, techniques et outils concourant à la production d'un logiciel [1]. Autrement dit le génie logiciel est l'ensemble des concepts, des méthodologies et des outils qui permettent de formaliser et même d'automatiser totalement ou partiellement la production de logiciels.

Le Génie Logiciel est un domaine de recherche qui a pour objectif de répondre à un problème qui s'énonçait en deux constatations : d'une part le logiciel n'était pas fiable, d'autre part, il était incroyablement difficile de réaliser dans des délais prévus des logiciels satisfaisant leur cahier des charges.

Parmi les tendances en génie logiciel : la complexité croissante des logiciels, la séparation des préoccupations et la multiplicité des plateformes. Ainsi, le groupe de standardisation **OMG** (**O**bject **M**anagement **G**roup) a défini l'approche **IDM** (L'**I**ngénierie **D**irigée par les **M**odèles, en anglais **M**odel **D**riven**E**ngineering) pour le développement de logiciels.

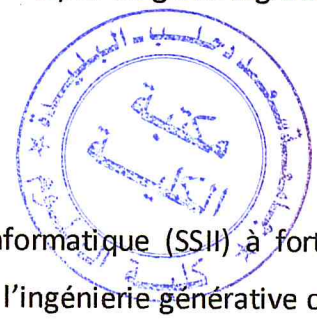
L'approche IDM a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou une partie d'application est engendrée à partir de modèles.

L'intérêt pour l'IDM a été fortement amplifié lorsque l'organisme de standardisation OMG a rendu publique son initiative **MDA** (**M**odel **D**riven **A**rchitecture, en français l'approche dirigée par les modèles). L'approche MDA s'appuie sur le standard **UML** (**U**nified **M**odeling **L**anguage) pour décrire séparément les préoccupations métier du système de celles de l'implémentation de ses fonctionnalistes.

Les activités découlant du génie logiciel sont bien définies : l'analyse des besoins, la conception architecturale, la programmation, la validation et vérification, la gestion de configuration et intégration, le suivi et la maintenance. Afin de faciliter la réalisation de ces

étapes et assurer la cohérence tout en suivant IDM, les AGL (**A**telier de **G**énie **L**ogiciel) sont adoptés comme solution.

Un **AGL** est lui-même un ensemble d'outils intégrés couvrant une partie significative du cycle de vie d'un logiciel et qui permettent de mettre en œuvre les principes du génie logiciel [1].



2. Problématique

IconSoftware est une **S**ociété de **S**ervices en **I**ngénierie **I**nformatique (SSII) à forte valeur ajoutée, en plein croissance. Elle vise à adopter le principe de l'ingénierie générative ou l'approche dirigée par les modèles (IDM) pour le développement de logiciels. En offrant à ses développeurs un AGL selon la démarche MDA afin de faciliter leurs tâches. Cependant un certain nombre de problèmes se sont présentés notamment :

- Le passage d'un modèle de domaine (diagrammes de définition des métiers Cas d'Utilisation, Classe, Activité) à un autre modèle de domaine sur les diagrammes de comportements.
- Le passage d'un modèle de domaine vers un modèle spécifique (diagrammes de classe, séquence) de la plateforme d'exécution sur les diagrammes de comportements.
- La synchronisation entre le modèle et le code source.
- La génération automatique du code source à partir des modèles.
- La sélection de la plateforme d'exécution lors de la génération de code selon les besoins du développeur.

3. Objectif

L'objectif de notre travail consiste à développer un prototype applicatif d'un AGL qui prend en considération les diagrammes UML de comportement, notre objectif est donc :

- Etablir l'inventaire des différents diagrammes de comportement (cas d'utilisation, activité, séquence) à manipuler ainsi que les différentes étapes de transformations et de génération à réaliser.

- Spécifier les profils UML (Ensemble de stéréotypes et contraintes pour un contexte spécifique) pour chaque diagramme UML de comportement afin de faciliter les transformations de modèles et la génération de code source.

Ce prototype doit être conçu et réalisé selon la démarche MDA à base d'Eclipse et les technologies suivantes :

- **UML** pour la modélisation ;
- **Java EE** pour la plateforme de développement.
- **ATL** (**A**tlas **T**ransformation **L**anguage) pour la transformation des modèles.
- **Acceleo** pour la génération de code source.

4. Organisation du mémoire

Ce mémoire est organisé en trois parties :

Dans la **première partie**, nous présentons les concepts de base relatifs à notre travail à savoir le concept d'AGL, l'approche et l'architecture dirigées par les modèles (IDM, MDA), les transformations de modèles et les différentes conventions de la plateforme Java EE.

Dans la **deuxième partie**, nous présentons les motivations qui ont donné naissance au besoin de la mise en place de l'AGL au niveau de l'entreprise IconSoftware. Ceci en décrivant le travail existant (qui se base sur les transformations des diagrammes UML de structuration) et le travail qui nous a été confié (qui se base sur les transformations des diagrammes UML de comportement). Par la suite, nous présentons la solution adoptée afin de répondre à la problématique posée.

Dans la **troisième partie**, nous présentons la conception et le développement du prototype de transformations des diagrammes UML de comportement et la génération de code technique spécifique aux conventions JEE. Cette partie se termine par un exemple d'utilisation du prototype développé.

Le mémoire se termine par une conclusion qui présente un bilan du travail réalisé et expose les perspectives et les travaux futurs pour améliorer et compléter le travail présenté.

Partie I :

Présentation des Concepts de base

Partie I

Chapitre 01 :

Atelier génie logiciel AGL

1. Introduction

Dans ce chapitre, nous présentons la raison de la naissance des ateliers de génie logiciel, ses principaux concepts, ses objectifs, ses types et en terminant par un exemple d'un AGL existant dans le monde de génie logiciel.

2. Génie Logiciel (*software engineering*)

Le génie logiciel est le domaine dédié à l'étude, la conception et la réalisation de programmes informatiques. Ce terme a été défini à l'origine par un groupe de travail en 1968, à Garmisch-Partenkirchen, sous le parrainage de l'OTAN. [03]

Actuellement, le génie logiciel peut se définir comme un ensemble composé de méthodes, de modèles, d'outils d'aide au développement, permettant de produire un logiciel répondant aux besoins exprimés par l'utilisateur. [03]

3. La crise du logiciel *(software crisis)*

La crise du logiciel est apparue à la fin des années 60 et provient d'un décalage entre les progrès matériels d'une part et logiciels d'autre part. [04]

La crise du logiciel peut tout d'abord se percevoir à travers ses symptômes:[03]

- les logiciels réalisés ne correspondent souvent pas aux besoins des utilisateurs
- les logiciels contiennent trop d'erreurs (qualité du logiciel insuffisante)
- les coûts du développement sont rarement prévisibles et sont généralement prohibitifs
- la maintenance des logiciels est une tâche complexe et coûteuse
- les délais de réalisation sont généralement dépassés
- les logiciels sont rarement portables.

L'explosion du domaine informatique a provoqué la naissance d'une horde d'outils de génie logiciel tous destinés à améliorer la production logiciel et augmenter sa productivité, ces outils portent le nom : **atelier de génie logiciel (AGL)**

4. Atelier de génie logiciel (AGL)

AGL signifie Atelier de Génie Logiciel (**CASE**, Computer Aided Software Engineering).

Un AGL est un ensemble d'**outils de Génie Logiciel** qui permet de fabriquer des logiciels. [05]

Un AGL est un ensemble cohérent d'outils informatiques permettant de couvrir le cycle de vie du logiciel. Autrement dit, il formant un environnement d'aide à la conception, au développement et à la mise au point de logiciels d'application spécialisés. [03]

Un AGL intègre des outils adaptés aux différentes phases de la production d'un logiciel et facilite la communication et la coordination entre ces différentes phases. [04]

4.1. Objectifs AGL

L'objectif étant d'aider l'utilisateur dans l'application des méthodes de développement : [06]

- Accélérer la production des systèmes
- Faciliter l'utilisation des techniques de modélisation
- Améliorer la productivité
- Automatiser les tâches manuelles
- Automatiser les contrôles et les vérifications
- Améliorer le suivi.
- Améliorer la qualité : fiabilité, maintenance, évolutivité.

Donc aider au déroulement des activités techniques de l'analyse, la conception, la réalisation et la maintenance

4.2. Les différents types d'AGL On distingue essentiellement deux types d'AGL selon la nature des outils intégrés: [04]

4.2.1. Les environnements de conception (upper-case) ces ateliers s'intéressent plus particulièrement aux phases d'analyse et de conception du processus logiciel. [04]

Ils intègrent généralement des:

- Outils pour l'édition de diagrammes (avec vérification syntaxique),
- Dictionnaires de données,
- Outils pour l'édition de rapports,
- Générateurs de (squelettes de) code, Outils pour le prototypage, ...

Ces ateliers sont généralement basés sur une méthode d'analyse et de conception (UML, Merise, ...).

Exemple d'un environnement de conception: *Modeler UML*

4.2.2. Les environnements de développement (lower-case) ces ateliers s'intéressent plus particulièrement aux phases d'implémentation et de test du processus logiciel. Ils intègrent généralement des : [04]

- Editeurs (éventuellement dirigés par la syntaxe),
- Générateurs d'interfaces homme/machine, des SGBD, des compilateurs, optimiseurs, debugger, ...

Exemple d'un environnement de développement: *Java Developer*

4.3. Exemples d'AGL

Il existe plusieurs AGL dans le monde de génie logiciel, nous citons : ArgoUML, Eclipse, StarUML, Modelio, WinDev...

L'un des exemples des ateliers génie logiciels existants, l'AGL *StarUML*

Star UML De [04]

StarUML permet la création de diagrammes UML

- Il permet la génération de code C#, C++ et JAVA (exemple : génération de code a partir de diagramme de classes)
- Génération de documentation (génération de squelette de fichier Word, génération de java doc)
- Retro-ingénierie de code dans ces langages.

5. conclusion

Dans ce chapitre nous avons présenté le génie logiciel, les principaux concepts des ateliers de génie logiciel, ses objectifs et ses types, et nous avons finalisé ce chapitre par un exemple de l'AGL « Star UML ».

Partie I

Chapitre 02 :

Démarche MDA et ses Standards

1. Introduction

Dans ce chapitre, nous mettons en avant l'apport de l'Ingénierie Dirigées par les Modèles (IDM) dans le nouveau cadre de développement de logiciel qui s'impose aujourd'hui, et de manière plus large, pour le génie logiciel. Plusieurs approches d'IDM ont été proposées, nous nous intéressons à l'approche **MDA** (**M**odel **D**riven **A**rchitecture : Architecture Dirigée par le Modèle) proposée par l'**OMG** (**O**bject **M**anagement **G**roups). Nous décrivons les principes de cette approche, l'architecture MDA et ses avantages. Enfin, nous présentons les standards et les outils de l'approche MDA.

2. Généralité de la démarche MDA

2.1. L'ingénierie dirigée par les modèles IDM

L'IDM offre un cadre méthodologique qui permet d'unifier différentes technologies dans un processus homogène. Ceci permet d'utiliser la technologie la mieux adaptée à chacune des étapes du développement, tout en ayant un processus global de développement qui soit unifié dans un paradigme unique. L'IDM permet cette unification grâce à l'utilisation importante des modèles et des transformations automatiques entre ces modèles. [8]

L'utilisation intensive de modèles permet un développement souple et itératif, grâce aux raffinements et enrichissements par transformations successives de modèles. Par ailleurs, les transformations permettent de passer d'un domaine technologique à un autre. Ainsi, les transformations permettent de choisir le domaine le plus adapté à chaque activité, tout en ayant un cadre méthodologique unique (l'IDM). [8]

L'IDM offre une solution intéressante face au problème de la spécification mouvante, en capitalisant une partie du développement grâce aux modèles. [8]

L'IDM est donc une approche prometteuse qui devrait permettre d'améliorer le développement de logiciel grâce à deux points forts qui sont : [8]

- La capitalisation du savoir-faire de conception grâce à la définition et l'implantation de transformations de modèles.
- La possibilité de choisir la solution technologique la plus adaptée à chaque activité du processus grâce aux ponts entre domaines technologiques.

2.2. Présentation de l'approche MDA

Afin de permettre l'interopérabilité entre les différents systèmes, réduire les coûts de développement et augmenter l'évolutivité, l'OMG propose la démarche **MDA**, où la notion de modèle devient essentielle et centrale [1].

MDA est une démarche de spécification de systèmes informatiques qui sépare la spécification des fonctionnalités d'un système de celle de l'implémentation de ces fonctionnalités sur une plate-forme particulière. Elle se concentre sur les modèles, fournissant un niveau d'abstraction supérieur pendant le développement, et permettant la séparation des modèles indépendants de la plate-forme des modèles spécifiques à une plate-forme cible. [13]

Les modèles étant plus pérennes que les codes, ils permettent ainsi de :

- Conserver les exigences métiers.
- Réutiliser les choix d'architecture et de codage.
- Assurer l'intégrité et la cohérence entre les phases du projet.

L'objectif majeur de MDA est l'élaboration de modèles pérennes, indépendants des détails techniques des plates-formes d'exécution afin de permettre la génération automatique du code des applications.[7]

2.3. Architecture de l'approche MDA

Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, MDA préconise l'élaboration de modèles d'exigences (CIM), d'analyse et de conception (PIM) et de code (PSM). [7]

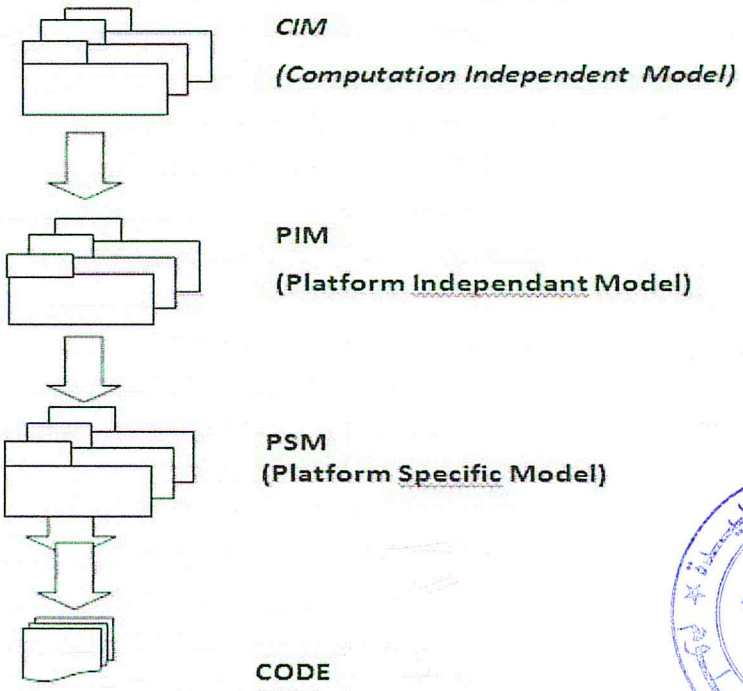


Figure 1 : Aperçu global de l'approche *MDA*. [7]

Les différents modèles du MDA :

1. **Le modèle d'exigences CIM (Computation Independent Model)**

Il est indépendant de tout système informatique. C'est le modèle métier ou le modèle du domaine d'application, il décrit les flux et les actions sur le système mais sans rentrer dans le détail de la structure du système. [7]

2. **Le modèle d'analyse et de conception abstraite PIM (Platform Independent Model)**

MDA considère que les modèles d'analyse et de conception doivent être indépendants de toute plate-forme d'implémentation, qu'elle soit J2EE, .Net, PHP.... et ne contiennent pas d'informations sur les technologies qui seront utilisées pour déployer l'application. C'est un modèle informatique qui représente une vue partielle d'un CIM. [7]

Le PIM représente le logique métier spécifique au système ou le modèle de conception. [9]

Le rôle des modèles d'analyse et de conception est d'être pérennes et de faire le lien entre le modèle d'exigences et le code de l'application. [9]

Le formalisme utilisé pour exprimer un PIM est un diagramme de classes en UML.

3. Le modèle de code ou de conception concrète PSM : (Platform Specific Model)

MDA considère que le code d'une application peut être facilement obtenu à partir de modèles de code. La différence principale entre un modèle de code et un modèle d'analyse ou de conception réside dans le fait que le modèle de code est lié à une plate-forme d'exécution. Dans le vocabulaire MDA, ces modèles de code sont appelés des PSM (Platform Specific Model). [7]

Les modèles de code servent essentiellement à faciliter la génération de code à partir d'un modèle d'analyse et de conception. Ils contiennent toutes les informations nécessaires à l'exploitation d'une plate-forme d'exécution. [9]

2.4. Transformation des modèles

Le MDA identifie plusieurs transformations pendant le cycle de développement. Il est possible de faire des transformations différentes.

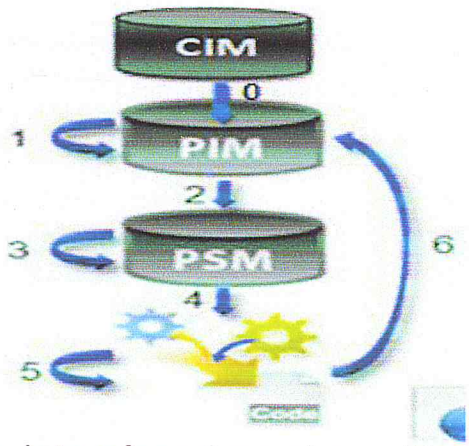


Figure 2 : Operations de transformation sur les modèles de MDA. [14]

1. De PIM vers PIM

Ces transformations sont utilisées pour enrichir, filtrer ou spécialiser les informations des modèles sans rajouter aucune information liée à la plate-forme. [9]

2. De PIM vers PSM

Une fois le PIM suffisamment raffiné pour pouvoir être spécialisé vers une plate-forme donnée, il peut alors être transformé en PSM. Cette opération consiste à ajouter au PIM des informations propres à une plate-forme technique. [9]

3. De PSM vers PSM

Une transformation PIM vers PSM n'est pas toujours suffisante pour permettre la génération de code d'où la nécessité de passer de PSM à PSM.

La transformation PSM à PSM s'effectue lors de phases de déploiement, d'optimisation ou de reconfiguration. [9]

4. De PSM vers code

Le code peut être assimilé par certains à un PSM exécutable, de même que la génération de code n'est pas toujours considérée comme une transformation de modèle. [9]

5. Inter-ingénierie transformation de PSM (ou Code) vers PIM

Cette transformation est utilisée pour revenir à un modèle indépendant de plate-forme (PIM) à partir d'un modèle spécifique de plate-forme (PSM) ou éventuellement du code. Ces transformations sont nécessaires pour permettre l'intégration d'applications existantes dans le processus MDA. [9]

2.5. Les avantages de MDA

Les trois avantages recherchés par MDA sont les suivants :

1. La pérennité des savoir-faire

Le métier des entreprises n'évolue que faiblement par rapport aux technologies qu'elles utilisent pour construire leurs applications informatiques. Fort de ce constat, il est évident que séparer les aspects métier des aspects techniques lors de la construction d'une application est une bonne pratique. Cela permet de rendre les spécifications métier pérennes, indépendamment des spécifications techniques.

L'avantage le plus important qu'offre MDA est précisément celui de la pérennité des spécifications métier. Par nature, son ambition est de faire en sorte que les PIM aient une durée de vie de plus de dix ans. [7]

2. Les gains de productivité

Il est toujours important pour une entreprise d'augmenter sa productivité afin de rester compétitive. Une façon bien connue d'augmenter la productivité est d'automatiser certaines étapes de production. C'est ce que fait MDA en automatisant les transformations de modèles.

Cet avantage est apporté grâce à l'automatisation des transformations des modèles puisque MDA vise une forte intégration des modèles dans le processus de production de l'application puisque les modèles sont directement utilisés pour générer le code de l'application. [7]

3. Prise en compte des plates-formes d'exécution

La prise en compte des plates-formes par MDA se fait en deux endroits : dans les PSM et dans les transformations PIM vers PSM.

Concernant les PSM, Rappelons-nous qu'un PSM est un modèle qui dépend fortement d'une plate-forme. Son métamodèle contient toute l'information relative à la plateforme.

Les transformations PIM vers PSM contiennent toute l'information permettant à un PIM d'utiliser une plate-forme déterminée. [7]

3. Outils et Standards de la démarche MDA

3.1. Architecture à quatre niveaux Méta

Afin d'organiser et de structurer les modèles, l'OMG a défini l'architecture à 4 niveaux :

Nous allons définit les 4 niveaux méta de cette architecture

1. Entités à modéliser : représente le Niveau 0 dans l'architecture de MDA

Les entités à modéliser dans le contexte MDA sont donc essentiellement des applications informatiques. [7] C'est le niveau des données réelles, il est composé des informations que l'on souhaite modéliser (instance de modèle). [12]

2. Les modèles : représentent le Niveau 1 dans l'architecture de MDA

Nous pouvons considérer que les modèles MDA sont des représentations de l'information nécessaire à la production et à l'évolution d'applications informatiques. Comme tout modèle, les modèles MDA doivent être fortement connectés avec la réalité, en l'occurrence avec les applications informatiques. [7]

Lorsque l'on veut décrire les informations d'une application, un modèle est établi. De même, tout modèle est exprimé dans un langage de modélisation. [12]

3. Les méta-modelés : représentent le Niveau 2 dans l'architecture de MDA

Nous avons dit précédemment, les modèles MDA sont des représentations de l'information nécessaire à la production et à l'évolution des applications informatiques et que ces modèles soient pérennes, productifs et qu'ils prennent en compte les plates-formes d'exécution. Ces qualités ne peuvent être obtenues qu'en définissant très précisément la structure des Modèles, ces définitions de structure des modèles se faisaient grâce à des *méta-modèles* est celle de *modèle de modèles*. [7]

Les méta-modèles sont au cœur de MDA. Ils permettent de pérenniser la structure des modèles, autrement dit, tout modèle doit respecter la structure définie par son méta-modèle [10]

Les méta-modèles représentés par des diagrammes de classes (un ensemble de méta-classes, des méta-associations, et méta-opérations) élaborés selon les règles du standard MOF, grâce à ce standard, tous les méta-modèles sont structurés de la même manière. [7]

Une multitude de méta-modèles peut être définie, chacun décrivant les structures d'un formalisme donné. Par exemple, le méta-modèle UML qui est décrit dans le standard UML définit la structure interne des modèles UML, il représente l'ensemble des concepts UML et les relations qui existent entre eux. [12]

4. MétaMétaModéle (Métamodèle des métamodèles) : représente le Niveau 2 dans l'architecture de MDA

Nous pouvons remarquer que la relation qui existe entre le **standard MOF** qui gére les MétaMétaModèle, qui est défini par OMG et les métamodèles est exactement la même que celle qui existe entre un métamodèle et ses modèles instances. [7]

Le *Métamétamodéle* (**MOF**) définissant la structure que doit avoir tout métamodèle ou d'étendre ou de modifier les méta-modèles existants, il est naturel de vouloir le représenter sous forme de diagramme de classes.

L'architecture à quatre niveaux telle que définie par MDA.

Cette architecture pose les bases des relations qui existent entre les quatre niveaux méta :

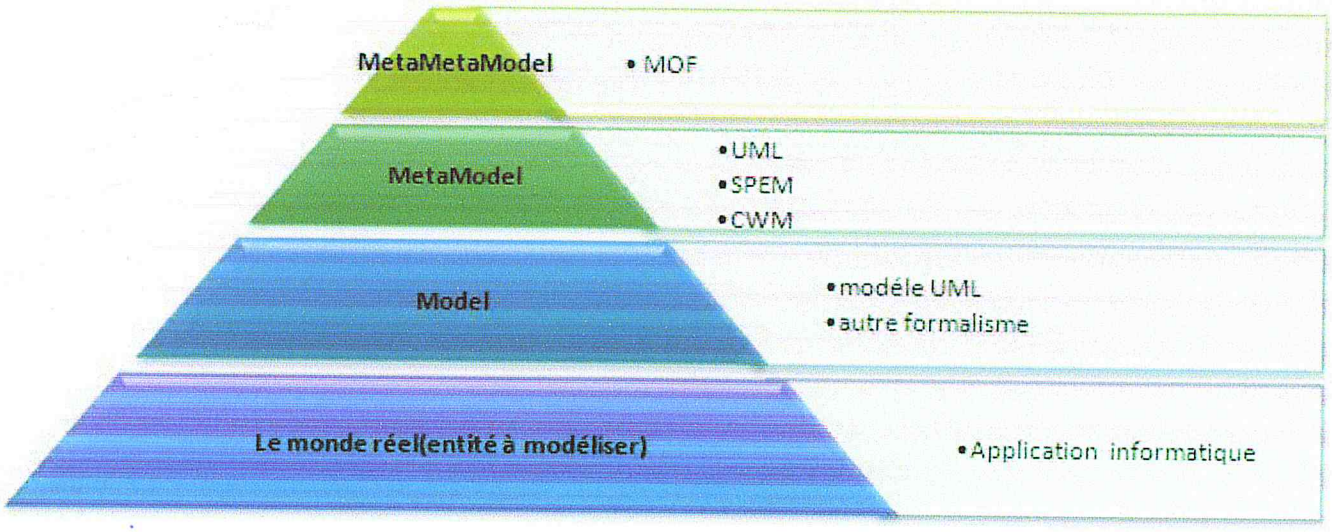


Figure 3: Architecture à quatre niveaux [2]

3.2. Modèles et langages de modélisation

La création de modèles est faite en utilisant un langage de modélisation.

Un langage de modélisation : est un outil de spécification formelle bien définie qui contient les éléments de base pour construire des modèles. De plus, il est conçu dans un domaine limité et avec des buts spécifiques, de sorte que nous pouvons considérer l'existence de plusieurs langages de modélisation, chacun étant adapté à un domaine spécifique (Exp : UML pour la conception). Le langage conçu pour créer des modèles est souvent défini comme un métamodèle.[10] Le langage de modélisation adopté par la démarche MDA est l'UML.

3.2.1. UML (Unified Modeling Language)

UML se définit comme un langage de modélisation graphique et textuel destiné à comprendre et décrire des besoins, spécifier et documenter des systèmes, esquisser des architectures logicielles, concevoir des solutions et communiquer des points de vue. [11]

Il représente une collection des meilleures pratiques d'ingénierie qui ont réussi dans la modélisation de systèmes grands et complexes. [10] Elle comporte:

- Une définition formelle du métamodèle d'UML, qui est le langage abstrait pour spécifier les modèles UML.
- Une notation graphique concrète pour représenter les modèles UML.

- Un format XMI pour échanger des modèles UML.

La première version d'UML a été publiée en 1997 par l'OMG. Depuis, UML est devenue la référence pour la création de modèles pour l'analyse et la conception de logiciels. [11]

UML s'articule autour de treize types de diagrammes, chacun d'eux étant dédié à la représentation des concepts particuliers d'un système logiciel. Ces types de diagrammes sont répartis en deux grands groupes :

3.2.1.1. Six diagrammes structurels (statique) De [11]

1. Diagramme de classes : est le point central dans un développement orienté objet.

En analyse, il a pour objet de décrire la structure des entités manipulées par les utilisateurs.

Il montre les briques de base statiques: classes, associations, interfaces, attributs, opérations, généralisations, etc.

2. Diagramme d'objets : est un graphe instantané, une photo d'un sous-ensemble des objets d'un système à un certain moment du temps.et aussi une instance de diagramme de classe.

C'est probablement le diagramme le moins utilisé d'UML.

Il montre les instances des éléments structurels et leurs liens à l'exécution.

3. Diagramme de packages : est l'organisation logique du modèle et les relations entre packages.

Il permet de structurer les classes d'analyse et de conception, mais aussi les cas d'utilisation.

4. Diagramme de composants : il montre les unités logicielles à partir desquelles on a construit le système informatique, ainsi que leurs dépendances.

Il montre des structures complexes, avec leurs interfaces fournies et requises.

5. Diagramme de déploiement : il montre le déploiement physique des artefacts (éléments concrets tels que fichiers, exécutables, etc.) sur les ressources matérielles.

6. Diagramme de structure composite: Le diagramme de structure composite montre l'organisation interne d'un élément statique complexe sous forme d'un assemblage de parties, de connecteurs et de ports.

3.2.1.2. Sept diagrammes comportementaux (dynamique) De [11]

1. **Diagramme de cas d'utilisation :** est utilisé dans l'activité de spécification des besoins.

Il représente les cas d'utilisation, les acteurs, et les relations entre le cas d'utilisation et les acteurs.

Il montre les interactions fonctionnelles entre les acteurs et le système à l'étude.

2. **Diagramme de séquence :** est un diagramme d'interactions UML.

Il représente des échanges de messages entre éléments, dans le cadre d'un fonctionnement particulier du système.

Les diagrammes de séquence servent d'abord à développer en analyse les scénarios d'utilisation du système.

Les diagrammes de séquence permettent de concevoir les méthodes des classes.

Il montre la séquence verticale des messages passés entre objets au sein d'une interaction.

3. **Diagramme d'activité :** il représente les règles d'enchaînement des actions et décisions au sein d'une activité. Il peut également être utilisé comme alternative au diagramme d'états pour décrire la navigation dans un site web.

4. **Diagramme d'états:** il représente le cycle de vie commun aux objets d'une même classe. Ce diagramme complète la connaissance des classes en analyse et en conception en montrant les différents états et transitions possibles des objets d'une classe à l'exécution.

5. **Diagramme de communication:** est un diagramme d'interactions UML.

Ils représentent des échanges de messages entre éléments, dans le cadre d'un fonctionnement particulier du système.

Les diagrammes communication permettent de concevoir les méthodes des classes

Il montre la communication entre objets dans le plan au sein d'une interaction.

6. **Diagramme de temps :** il fusionne les diagrammes d'états et de séquence pour montrer l'évolution de l'état d'un objet au cours du temps et les messages qui modifient cet état.

7. **Diagramme de vue d'ensemble des interactions :** Il fusionne les diagrammes d'activité et de séquence pour combiner des fragments d'interaction avec des décisions et des flots.

3.2.2. UML et MDA De [7]

- UML permet principalement de construire des modèles d'applications informatiques indépendants des plates-formes d'exécution (phase d'analyse et de conception abstraite)
- UML est donc le métamodèle naturel pour les PIM (Platform Independant Model)
- UML permet aussi de représenter une application dans son environnement afin d'en faire ressortir les exigences (cas d'utilisation)
- UML peut être profilé afin de cibler des plates-formes d'exécution (ex: profil pour CORBA, profil pour EJB)
- UML peut être utilisé pour les PSM (Platform Specific Model)

Il serait donc possible d'appliquer MDA en utilisant uniquement UML

3.2.3. UML2.0 *(Unified Modeling Language)*

Après une longue période de discussions techniques et stratégiques, l'OMG a voté, en juin 2003 à Paris, l'adoption du standard UML2.0 qui fait rentrer UML dans MDA.

UML est actuellement le métamodèle le plus important de l'approche MDA. Sa conception est le fruit de plus de 3 ans de travail collaboratif des meilleurs experts du domaine.

C'est le métamodèle qui définit la structuration des modèles des applications informatiques [7]

- UML2.0 est un métamodèle instance de MOF2.0.
- La sémantique d'UML2.0 est définie à l'aide d'un langage naturel.
- Les diagrammes UML2.0 sont définis à partir du métamodèle
- UML2.0 est le métamodèle dédié à la modélisation d'applications orientées objet.

3.3. Les profils UML

Un profil est un ensemble de techniques et de mécanismes permettant d'adapter UML à un domaine particulier. [7]

Pour permettre l'adaptation d'UML à d'autres domaines et pour préciser la signification de cette adaptation, l'OMG a standardisé le concept de *profil UML.* [7]

Cette adaptation est dynamique, c'est-à-dire qu'elle ne modifie en rien le métamodèle UML et qu'elle peut se faire sur n'importe quel modèle UML. [7]

Les spécificités de chaque plate-forme peuvent être modélisées grâce aux mécanismes d'extension d'UML définis par les profils UML. [10] tel que : les stéréotypes, les tagged Value, les contraintes

Stéréotype : Un stéréotype est une sorte d'étiquette nommée que l'on peut coller sur n'importe quel élément d'un modèle UML. Lorsqu'un stéréotype est collé sur un élément d'un modèle, le nom du stéréotype définit la nouvelle signification de l'élément. [7]

Ils décrivent comment le méta-modèle UML peut être étendu sur un domaine spécifique. [12]

Valeurs marquées(ou Tagged Values): Depuis UML2.0, les tagged-values sont exclusivement utilisées sur des éléments stéréotypés. Cela permet de préciser en détail la façon dont est utilisée la plate-forme. [7]

Une valeur marquée est un couple (nom, valeur).[12]

Contraintes (ou Constraints): les contraintes permettent de restreindre l'utilisation des stéréotypes et des tagged-values. Grâce aux contraintes, il est possible de spécifier des structures obligatoires entre des éléments stéréotypés. [7]

Elles sont écrites soit en un langage naturel soit avec un langage formel tel qu'OCL. [12]

Il est possible d'associer les stéréotypes, les valeurs marquées et les contraintes à tout concept UML (classe, attribut, association, cas d'utilisation). [10]. Ces éléments permettent d'établir une correspondance entre les concepts UML et les concepts du domaine représentés par le profil. [10]

3.4. Langages de méta-modélisation

Un méta-modèle utilise d'autres langages connus comme langages de méta modélisation. Chaque langage de méta-modélisation correspond à un méta-méta-modèle, c'est à dire ce qui précède le méta-modèle. [7]

La relation entre un méta-méta-modèle, un méta-modèle, un modèle et une information est bien définie dans l'architecture à quatre niveaux, [7] présentée précédemment.

3.4.1. MOF (*Meta Object Facility)*

MOF fait partie des standards définis par l'OMG. Le MOF et l'UML constituent le cœur de la technologie MDA. [7]

Le MOF spécifie la structure et la syntaxe de tous les méta-modèles comme UML, CWM et SPEM. [10]

À ce jour, les versions stables et normalisées sont la version 2.0 de MOF et la version 2.0 d'UML. [10]

3.4.2. MOF2.0

Le MOF (Meta-Object Facility) a été adopté en 1997 par l'OMG. La spécification de MOF définit un langage abstrait pour la spécification, la construction, et la gestion de méta-modèles générique. [10]

- MOF2.0 demeure le métamétamodèle au niveau le plus abstrait (M3) dans l'architecture à 4 niveaux de MDA. [7]
- Le standard MOF définit le langage de définition des métamodèles (UML) [7]
- MOF peut être défini à l'aide d'un diagramme de classe. Ce diagramme est le métamétamodèle [7]
- MOF est aussi utilisé pour définir un format d'inter-change pour modèles du niveau2, il s'agit du format standard XMI. [2]
- En ce qui concerne l'architecture MDA, MOF permet la définition des langages de modélisation. [2]

D'un point de vue conceptuel, MOF2.0 est toujours considéré comme étant l'unique métamétamodèle. [7]

4. Conclusion

Au cours de ce premier chapitre nous avons illustré l'apport de l'ingénierie dirigée par LES modèles pour le développement de logiciels à travers les points forts de l'IDM qui ont été identifiéS au début du chapitre.

Nous avons présenté la démarche MDA ainsi ces outils et standards tel qu'ils sont définis par l'OMG.

Partie I

Chapitre 03 :

Transformations des modèles

1. Introduction

Au cours de ce chapitre nous introduisons le concept et le mécanisme des transformations de modèles vers modèles (M2M : model to model), et le mécanisme de génération de code source (M2T : model to text).

Ensuite nous présentons les langages de transformation des modèles et de génération utilisés. Pour les transformations M2M nous utilisons le langage ATL qui s'appuie sur le langage de contraintes OCL. pour la génération du code source nous utilisons le générateur Acceleo.

2. Transformations de modèles

Les transformations sont au cœur de l'approche MDA. Elles permettent d'obtenir différentes vues d'un modèle ou de le raffiner, de plus elles permettent de passer d'un langage vers un autre. [23]

Une transformation de modèle se fait par mapping entre un modèle initial et un modèle cible. Chaque modèle doit être décrit par un méta-modèle qui recense les caractéristiques de ce modèle. Le mapping est alors définit comme une traduction entre le méta-modèle initial et le métamodèle cible (Figure 04). [22]

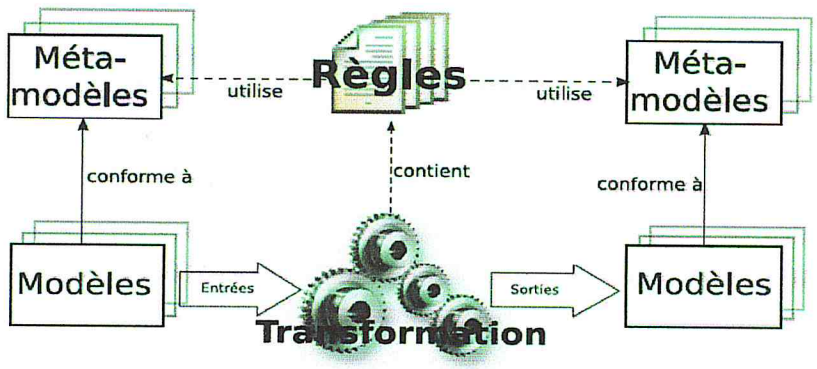


Figure 04 : Architecture générale de la transformation des modèles. [23]

Une transformation de modèle peut être [20] :

- Selon le méta-modèle source et cible endogène ou exogène :
  - **Exogène** si le métamodèle source est différent du méta-modèle cible.
  - **Endogène** si les modèles source et cible ont le même méta-modèle.
- Selon les niveaux d'abstraction des modèles sources vis-à-vis des modèles cibles, horizontale ou verticale :
  - **Horizontale** si les modèles sources et cibles ont le même niveau d'abstraction.
  - **Verticale** si les modèles sources et cibles sont de niveaux d'abstraction différents.

3. Types de transformations de modèles

Les transformations de modèles se décomposent en deux catégories : transformation d'un modèle vers un autre modèle (notée M2M) et transformation d'un modèle vers du texte (notée M2T). [07]

3.1. Transformation d'un modèle vers un modèle

La figure ci-dessous représente la transformation de modèles. Le **modèle MC**, conforme au **méta-modèle MMC,** est obtenu par l'application de la **transformation MMS à MMC** au **modèle MS**, qui est aussi conforme au **méta-modèle MMS**.

De plus, en suivant le principe *tout est modèle*, la transformation elle-même est un modèle dont le méta-modèle est **MMT**. On dit alors que **MMS à MMC** est un **modèle de transformation**. Le langage de transformation (ses concepts et leurs relations) est donc capturé par ce méta-modèle. Les trois méta-modèles MMS, MMC et MMT sont conformes au méta-méta-modèle qui est conforme à lui-même. [12]

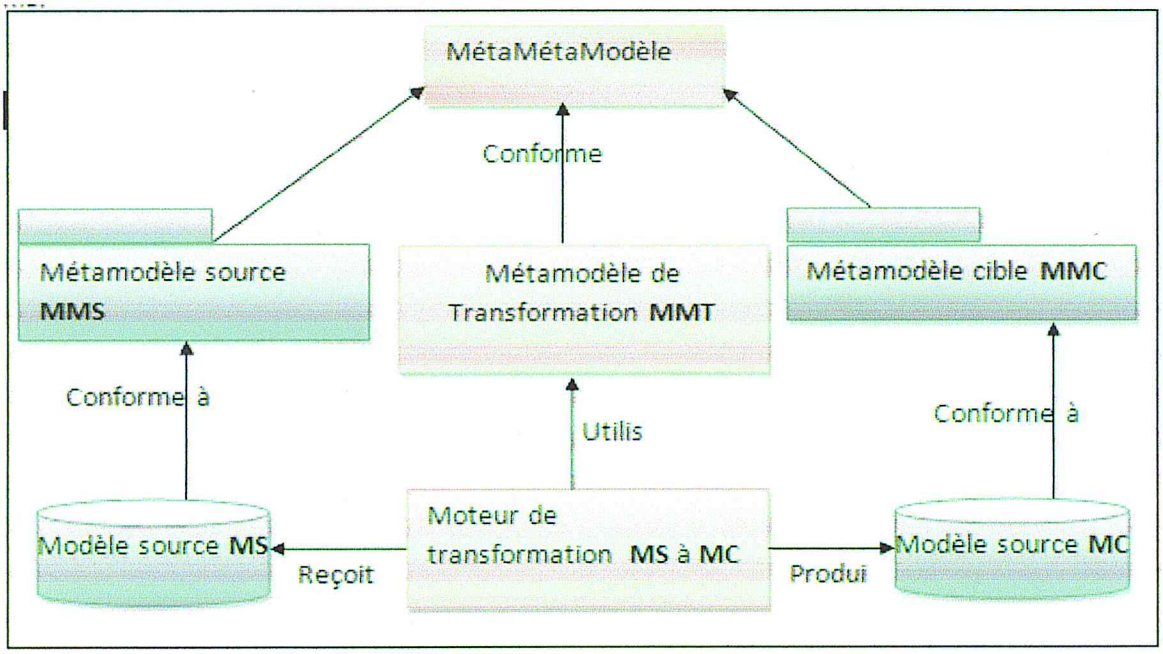


Figure 05 : le mécanisme général de la transformation d'un modèle à un modèle [12]

3.2. Transformation d'un modèle vers un texte

Un besoin naturel est apparu après la réalisation d'une transformation M2M, celui de pouvoir simplement générer un modèle productif à partir d'un modèle abstrait; autrement dit, il s'agit de générer du code à partir d'un modèle et d'effectuer des transformations de type **modèle à texte** [02].

La transformation d'un modèle vers du texte est une opération couramment réalisée. Elle permet, une fois le modèle saisi, de l'utiliser directement après l'avoir transformé sous un format textuel. Plusieurs types de sortie textuelle sont possibles. A partir d'un modèle, il est possible de générer du code exécutable [12].

4. Langages de transformation

4.1. Le Standard QVT

Le langage **QVT** (**Q**uerry **V**iew **T**ransformation) a été normalisé par le groupe OMG dans le cadre de l'approche MDA, pour la transformation et de manipulation de modèles [21] :

- **Query :** Sélectionner des éléments sur un modèle en utilisant pour cela le langage **OCL** avec une syntaxe simplifiée.
- **View (ou Vue) :** C'est un modèle à part, avec éventuellement un méta-modèle restreint spécifique à cette vue.
- **Transformation** : Permet de transformer un modèle en un autre.

4.2. Le langage ATL

Dans notre travail, pour la transformation de type modèle vers modèle nous avons utilisé le langage de transformation ATL. Il a été développé par le laboratoire INRIA de Nantes dans le cadre de son projet ATLAS [16]. Le projet ATL fait partie du projet Eclipse, Il est développé sur sa branche **EMF** (**E**clipse **M**odeling **F**ramework) de la plateforme Eclipse [15].

Une transformation ATL peut transformer un ensemble de modèles sources en un ensemble de modèles cibles, à la condition que les méta-modèles sources et cibles lui soient connus. [12]

4.2.1. Description d'un module ATL

Un module ATL définit une transformation d'un modèle à un modèle. C'est le type d'unités ATL qui permet de spécifier la façon de produire un ensemble de modèles cibles à partir d'un ensemble de modèles sources [12]. Un module ATL est composé des éléments suivants [17] : Déclaration du module: **Header section ;** Import de librairies: **Import section ;** Opérations: **Helpers** et Règles de transformation: **Rules**

4.2.1.1. Header section ou section d'entête

Cette section est utilisée pour déclarer : le nom du module, le méta-modèle d'entrée et de sortie ainsi que les bibliothèques nécessaires [15].

```
Module nom_module;
Create output_models from input_models;
```

Figure 06 : Entête d'un module ATL. [18]

4.2.1.2. Section d'import

Elle permet de déclarer quelles librairies ATL doivent être importées dans un module ATL. [18]

```
Uses nom_fichier_librairie;
```

Figure 07: Déclaration d'une librairie ATL dans un module ATL. [18]

4.2.1.3. Helpers

Les fonctions ATL sont appelées **Helpers** d'après le standard OCL sur lequel il se base. Un Helper est définit par les éléments suivants : le nom, le type du contexte, le type de la valeur de retour, une expression OCL et un ensemble de paramètres [18].

```
Helper context contexte_helper def : nom_helper (paramètre: type_parmètre):
                                     Type_résulat =code_helper;
```

Figure 08 : Définitions de helpers (avec *contexte* et *paramètre*) dans un module ATL. [18]

4.2.1.4. Règles ou rules

Un programme de transformation écrit en ATL est composé d'un ensemble de règles. Ces règles sont au cœur de ces transformations car :

- Elles décrivent comment les éléments du modèle source sont reconnus et parcourus.
- Elles sont utilisées pour créer et initialiser les éléments du modèle cible. Sachant que chaque éléments d'entré est basé sur un méta-modèle d'entré et chaque éléments de sortie est basé sur un méta-modèle de sortie. [16]

4.3. Le générateur Acceleo

Notre travail porte également sur les transformations de type modèle vers texte. Pour cela, nous avons adopté le générateur de code Acceleo. Il a été développé par les fondateurs de la société Obeo. Acceleo est une technologie Open Source [24].

Acceleo est caratctérisé par: [25]

- Son intégration complète à l'environnement Eclipse
- La gestion de la synchronisation entre le code et le modèle.
- La facilité de mise au point et de maintenabilité des templates.
- La colorisation syntaxique, la complétion, la détection d'erreurs.

4.3.1. Principe de fonctionnement d'Acceleo

Le principe de fonctionnement d'Acceleo est comme suit : Acceleo utilise un template pour générer du code à partir d'un modèle. Afin d'exécuter la génération (template), il faut créer un fichier de type « .chain » en spécifiant dans cette chaine les templates, les modèles d'entrée et leur méta-modèle source, le répertoire cible des fichiers générés ainsi que le fichier log « journal des erreurs de génération». [24]

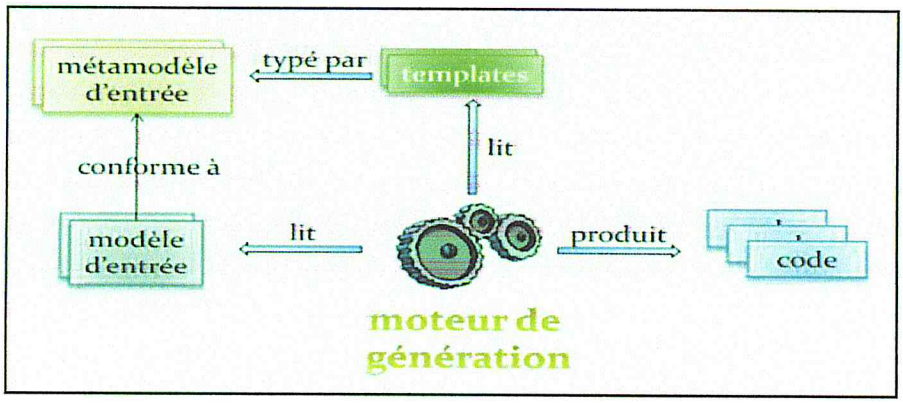


Figure 09 : Acceleo - principe général [26]

4.3.2. Concepts de base

a. Syntaxe: La syntaxe d'Acceleo est conçue pour garder une visibilité optimiale sur la cible technique à générer. [24]

b. Scripts : Tous les scripts commencent par une déclaration <%script ... %>. Elle définit l'élément du méta-modèle sur lequel le script va s'appliquer et le nom du script. [24]

Exemple d'un scriprte Acœleo : <% script type="Class" name="generate" file="<% fileputh %>' ' %>

c. Texte variable: Un template de génération est un patron qui se doit de délimiter de façon claire le texte statique généré et les éléments variables. Le texte variable commence par <% et fini par %>. [24]

```
13  package <%nGet("11").getModel().name%>.service.api;
14  import javax.ejb.Local;
15
16  @Local
17  public interface <%name.toU1Case()%> {
18
```

Figure 10 : Exemple d'un texte variable

d. Code utilisateur: Le code utilisateur est le texte qui sera conservé d'une génération à l'autre. Tout le code cible écrit entre les balises identifiées par <%startUserCode%> et<%endUserCode%> est protégé et n'est pas écrasé à la génération suivante ce qui permet de personnaliser le code généré. [24]

e. Services : Les services permettent d'étendre les modules de génération pour réaliser des opérations plus complexes. Ils permettent de manipuler simplement les objets du méta modèle, ou des types standards du JDK ou d'Eclipse. [24]

f. Imports : La section "*import*" se trouve toujours en début de fichier. Elle permet de préciser les dépendances des templates avec d'autres templates et les services. [24]

La première ligne représente toujours la déclaration du méta-modèle sur lequel s'applique le template. Les imports peuvent être des templates dont les scripts seront accessibles et/ou des services Java. [24]

```
1  <%
2  metamodel http://www.eclipse.org/uml2/3.0.0/UML
3  import com.iconsoft.apps.gencodesource.service.ServiceM2T
4  import differentsTemplate.apiService
5  import differentsTemplate.apiDAO
6  %>
```

Figure 11 : exemple de la section import

5. Le Langage de contraintes OCL

OCL (**O**bject **C**onstraint **L**anguage) est un langage proposé par l'OMG pour la description de contraintes sur des modèles UML. OCL est également un langage de requête qui permet de calculer une expression sur un modèle en s'appuyant sur sa syntaxe (son méta-modèle). Une expression exprimée une fois, pourra être évaluée sur tout modèle conforme au méta-modèle correspondant. [19]

OCL est défini comme un langage sans effet secondaires (side-effect free en anglais), c'est-à-dire qu'il garantit que l'évaluation d'une requête OCL sur un modèle quelconque ne modifiera en rien le modèle en question. [16]

Dans une transformation ATL, le langage OCL est utilisé comme langage de navigation pour les modèles. [20]

6. conclusion

Dans ce chapitre, nous avons présenté le concept de transformation des modèles et les différents types de transformation de modèles. Nous avons présenté également les langages utilisés que nous allons utiliser au sein de notre travail à savoir : ATL pour les transformations (M2M) et Acceleo pour la génération du code source (transformation M2T).

Partie I

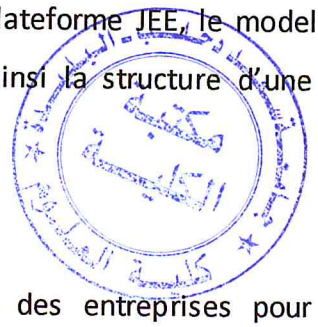
Chapitre 04 :

Plateforme Java EE (JEE)

1. Introduction

L'AGL à développer génère des applications web à partir de modèles UML PIM tout en passant par des modèle UML PSM conforme à la plateforme Java Enterprise Edition (**Java EE ou JEE**).

Nous présentons donc dans ce chapitre la spécification de la plateforme JEE, le model MVC (Model View Controler) pour mieux structurer l'application, ainsi la structure d'une application web selon les conventions de l'entreprise IconSoftware



2. Présentation générale de JEE

JEE (Java Enterprise Edition) est né de besoins grandissant des entreprises pour développer des applications complexes distribuées et ouvertes sur l'Internet. C'est une plateforme pour la technique Java de **Sun Microsystems** permettant de faciliter le développement d'applications d'entreprise en fournissant un ensemble de composants sous forme d'APIs (EJB, JSP, JDBC...) [30].

JEE est un ensemble de spécifications coordonnées et pratiques qui permettent un ensemble de solutions pour le développement, le déploiement et la gestion des applications multi-tiers centralisées sur un serveur [31].

JEE regroupe à la fois un ensemble de bibliothèques (Servlets/EJB/ JSP) et une plate-forme logicielle. Elle fournit de nombreuses solutions au niveau de la gestion des problèmes récurrents dans ce domaine : sécurité (API JAAS), prise en charge des transactions (API JTA et gestion déclarative dans les EJB). JDBC fait aussi partie de ces technologies [31].

2.1. Architecture de la plateforme Java EE

La plateforme JEE est constituée de trois couches : la couche Persistance, la couche présentation et la couche métier.

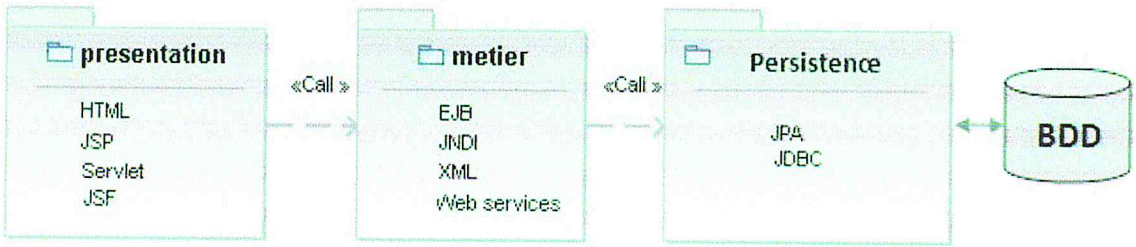


Figure 12 : Architecture à 3 couches de JEE [30]

2.1.1. La couche Persistance

La couche de persistance contient les données sauvegardées physiquement sur disque, c'est-à-dire la base de données. En Java, le protocole d'accès aux données est JDBC [34].

Elle regroupe le stockage et les mécanismes d'accès aux données de façon à ce qu'elles soient utilisables par l'application au niveau traitement.

Utiliser des objets codés en Java (ou dans tout autre langage objet) pour représenter des données stockées dans une base relationnelle à l'aide du Framework JPA ; dans le besoin de transformer ces objets afin de les stocker ou de les lire. [35]

Cette couche permet : [35]

- De simplifier la couche métier qui utilise les traitements (ou services) de cette couche.
- De faciliter le remplacement de la base de données utilisée (on peut remplacer le type de donnée ou le SGBD)
- De masquer les traitements réalisés pour mapper les objets dans la base de données et vice versa.

La couche métier utilise les classes de la couche persistance qui encapsulent ces traitements. Ainsi la couche métier manipule des objets pour les accès à la base de données. [35]

2.1.2. La couche métier (service ou traitement)

La plupart des actions qu'effectue un utilisateur dans un logiciel à travers la couche présentation ont pour but de consulter ou modifier l'état d'un ou plusieurs objets de la couche persistance. La couche métier est développée pour gérer les entités beans de la couche persistance en utilisant les EJB, et utilise l'interface Entity Manager pour accéder au classe JPA. Ainsi, que son langage de requêtes JPQL pour accéder aux données. [31]

2.1.3. La couche Présentation (User Interface UI)

La couche de présentation est la partie visible de l'application. Elle permet à un utilisateur d'interagir avec le système. Elle relie les requêtes de l'utilisateur à destination de la couche métier, et en retour lui présente les résultats renvoyés par les traitements. On parle alors d'interface homme-machine (IHM). Aucun traitement n'est implémenté dans cette couche [34].

3. Le Design Pattern MVC

Dans le domaine de l'analyse et de la conception orientée-objet, un design pattern ou motif de conception est une manière de construire la structure d'une classe. Ainsi, il décrit une solution générale à un problème qui revient souvent. [32]

Le Model-View-Controler (MVC) est un modèle de conception logicielle très répandu et fort utile. Créé dans les années 80 par **Xerox PARC** pour **Smalltalk-80** dans le but de mieux structurer une application avec une interface graphique. Il est fortement recommandé dans l'univers JEE. [31]

MVC étant un modèle de conception, il est donc indépendant de tout langage de programmation [31].

MVC permet de séparer le modèle, les vue et le contrôleur. On peut appliquer cette architecture à la mise en œuvre d'une application Web.

3.1. Architecture MVC

Chaque composant d'une architecture MVC a un rôle bien défini :

- **Le Modèle :** Il gère le comportement et les données du domaine métier et répond aux demandes d'informations de l'utilisateur. [34]
- **La Vue** : Est un composant graphique. Elle correspond à l'IHM. Elle affiche les données du modèle et interagit avec l'utilisateur. Elle représente la présentation visuelle de l'application [34].
- **Le Contrôleur :** Il est chargé d'intercepter les requêtes de l'utilisateur, d'appeler le modèle puis de rediriger vers la vue adéquate [34].

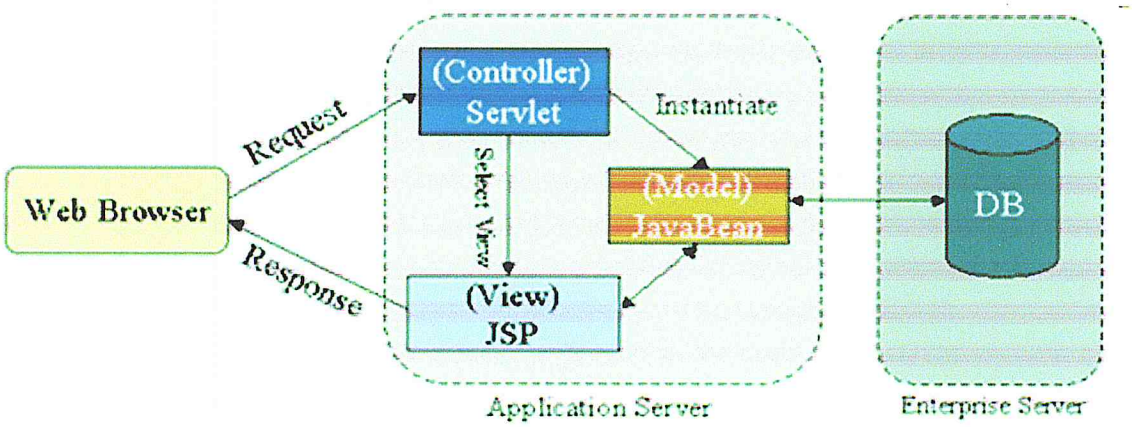


Figure 13: L'architecture MVC [33]

3.2. Techniques d'implémentation du MVC en JAVA

Dans notre cas nous avons utilisé MVC orienté composant.

3.2.1. MVC orienté requête (MVC version N°01)

Dans ce modèle, chaque requête est traitée par un contrôleur sous la forme d'une servlet. Celle-ci traite la requête, fait appel aux éléments du modèle si nécessaire et redirige la requête vers une JSP qui se charge de créer la réponse à l'utilisateur. [33]

3.2.2. MVC orienté composant (MVC version N°02)

Le MVC orienté composant est une amélioration du MVC orienté requête. Dans une telle architecture, il n'existe plus qu'un seul et unique contrôleur réceptionnant toutes les requêtes clientes. Le contrôleur unique devient le point d'entrée exclusif de l'application. [33]

Ainsi, dans ce modèle :

- Le **modèle** : gérer par les ManegedBeans.
- La **vue** : des pages JSF qui fournissent au client la réponse à ses requêtes.
- Le **contrôleur** : une FacesServlet qui gère les transactions. Il exécute le code nécessaire à une requête, et en coordonne la réponse.

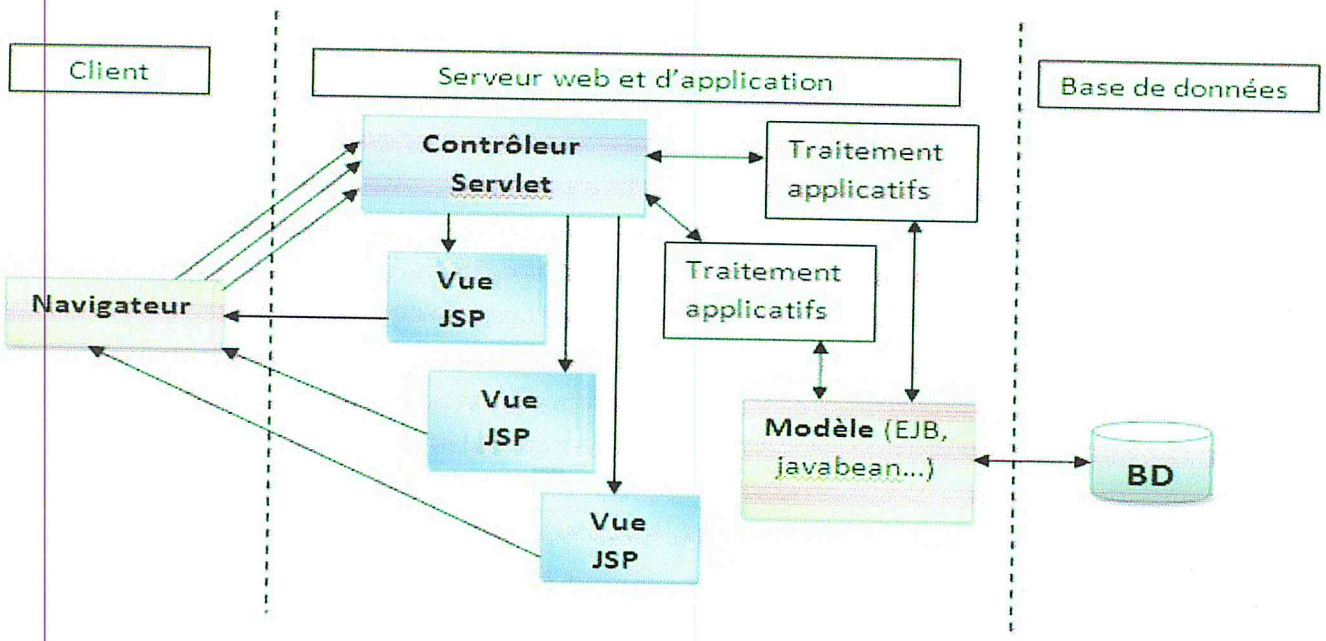


Figure 14 : Schéma du MVC version N°2 [33]

4. Composants JEE

4.1. Les Framework on va présenter l'ensemble des Framework utilisés pour développer une application JEE sous l'environnement Eclipse :

4.1.1. Java Persistence API (JPA) Elle se base sur la programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java. Toutes les classes, interface et annotations de cette API se trouvent dans le paquetage. javax persistence. [34]

JPA repose notamment sur:

- L'utilisation des Annotations (@Entity, @Table, @Column, @id....).
- Un gestionnaire de persistance (EntityManager) qui assure la gestion des entités persistantes.
- La configuration via des fichiers xml.

4.1.2. Entreprise Java Beans (EJB) EJB est une classe qui contient la logique métier du composant JPA et doit implémenter une interface (possédant les services fournis correspondant à ce composant). Le client de l'EJB n'utilise pas directement cette classe, mais doit passer par cette interface [37].

4.1.3. Java Server Faces (JSF) JSF est un Framework de développement d'applications Web. Il permet de développer des interfaces client riches tout en respectant le paradigme MVC [36].

4.1.4. Java DataBase Connectivity (JDBC) JDBC est l'API standard offerte aux développeurs pour accéder à des bases de données relationnelles. [35]

Elle se charge de trois étapes indispensables à l'accès des données: [34]

- La création d'une connexion à la base.
- L'envoi d'instructions SQL.
- L'exploitation des résultats provenant de la base.

5. Structure d'une application Web JEE

Une application web est un archive qui se présente sous la forme d'un fichier WAR (*.war), qui est l'acronyme de **W**eb **A**pplication **R**esource.

Le fichier WAR structure et organise les éléments de l'application web :

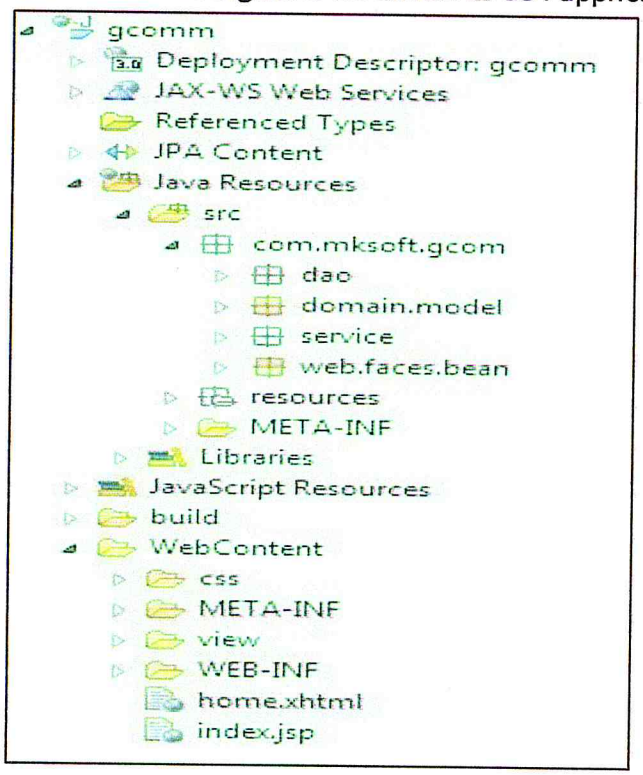


Figure 15: Structure interne d'un projet d'une application JEE sous Eclipse

Le premier niveau contient le package src qui structure et organise l'ensemble des sous package de l'application JEE

Le package principale : com.iconsoft.apps.nom de projet

❖ **Les sous packages**

- **Domaine** : contient l'ensemble des classes JPA qui correspond aux différentes entités de la Base de données
- **DAO** : contient un ensemble d'interfaces et de classes qui gèrent les différentes méthodes de base (créer, modifier, supprimer et afficher) pour chaque classe du package Domaine.

- **Service** : un ensemble des interfaces et des classes service qui contiennent les différentes méthodes et traitements métiers pour chaque entité JPA et qui utilise la classe DAO correspondante.
- **Web.faces.bean :** contient l'ensemble de classes (*ManagedBean*) qui gèrent l'ensemble des interfaces graphique qui se trouvent dans le répertoire *Web Content.*

Le second niveau et constitué par le répertoire WebContent qui contient différentes fichiers tels que : le fichier META-INF,

Le fichier CSS : contient le code CSS qui organise l'ensemble des interfaces graphique de l'application.

Le répertoire WEB-INF qui contient des divers éléments tel que : le sous-répertoire « lib » contient tous les Framework et les bibliothèques utiliser par l'application tels que jdbc, Primefaces...

Le répertoire View qui contient les différentes interfaces graphiques (JSF, XHTML..) de l'application JEE.

6. Conclusion

Dans ce chapitre nous avons présenté la plateforme JEE.

Nous avons illustré le modèle MVC, les deux versions : MVC1 orienté requête et MVC2 orienté composant, les composants JEE et nous avons terminé par la structure d'une application JEE.

Partie II :

Besoins de l'entreprise IconSoftware et Solution Proposée

Partie II

Chapitre 01 :

Besoins de l'entreprise IconSoftware

1. Introduction

Dans ce chapitre, nous présentons les besoins de l'entreprise **IconSoftware**, pour le développement d'un atelier génie logiciel (AGL) de réalisation d'applications web selon la plateforme JEE en adoptons la démarche MDA. Nous présentons en premier, un travail réalisé au sein de l'entreprise qui consistait en une partie de cet AGL. Nous présentons par la suite, l'apport de notre travail par rapport au travail existant ayant pour objectif de le compléter.

2. Besoin de l'entreprise (Mise en place d'AGL)

Afin de faciliter la création d'application de grande taille et pour réduire le cout de développement en termes de temps, l'entreprise **IconSoftware** souhaite de mettre en place à ces développeurs un AGL complet de réalisation de projets logiciel selon la démarche **MDA** concentré sur les diagrammes UML (**Figure 16**).

L'objectif est d'accélérer et d'automatiser le développement d'applications en réalisant un outil permettant d'effectuer les différentes transformations sur les diagrammes UML de structuration et de comportement :

- **Les transformations de diagrammes UML de structuration** consistent dans la transformation du diagramme de classe de domaine PIM vers le diagramme de classe technique PSM. Ce dernier permet de générer le code source des méthodes de base *(CRUD)* de l'application web.
- **Les transformations de diagrammes UML de comportement** consistent dans la transformation du diagramme de domaine cas d'utilisation vers le diagramme de classe PIM. Cette transformation a pour objectif d'intégrer le travail existant au sein de notre travail. Les transformations des diagrammes de comportement portent également sur le diagramme d'activité de domaine PIM. A partir de dernier ainsi que des classes PSM générées précédemment, il sera possible de générer le diagramme de séquence PSM. Ce dernier représente le modèle source de la génération de l'application web selon la plateforme JEE.

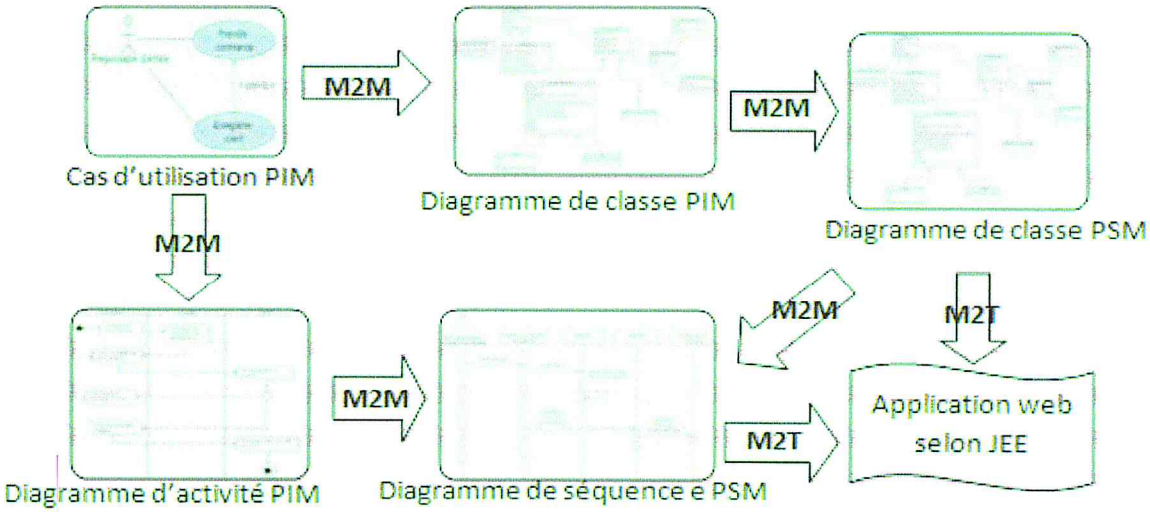


Figure 16 : architecture future de l'Atelier génie logiciel de l'entreprise

3. Présentation du travail existant au sein de l'entreprise IconSoftware

Afin de répondre aux besoins de l'entreprise IconSoftware cités précédemment, un projet a été lancé et réalisé par les deux ingénieurs R. Boumendjas et H. Bekkouche. Le projet consistait à mettre en place un outil permettant d'effectuer les **transformations** de modèles sur **les diagrammes UML de structuration** (diagramme de classe) ainsi que la **génération** de code technique de l'application web selon la plateforme JEE. Ceci en s'appuyant sur des solutions Open Source tel que le langage de transformation **ATL** et le langage de génération de code source **Acceleo**.

Pour la mise en place de cet outil, les deux ingénieurs ont développé des profils UML tout en spécifiant les règles de transformations et de génération à effectuer sur les diagrammes de structuration (diagramme de classe). Dans ce qui suit, nous présentons une architecture décrivant les différentes phases du projet réalisé : transformations de modèles et génération de code.

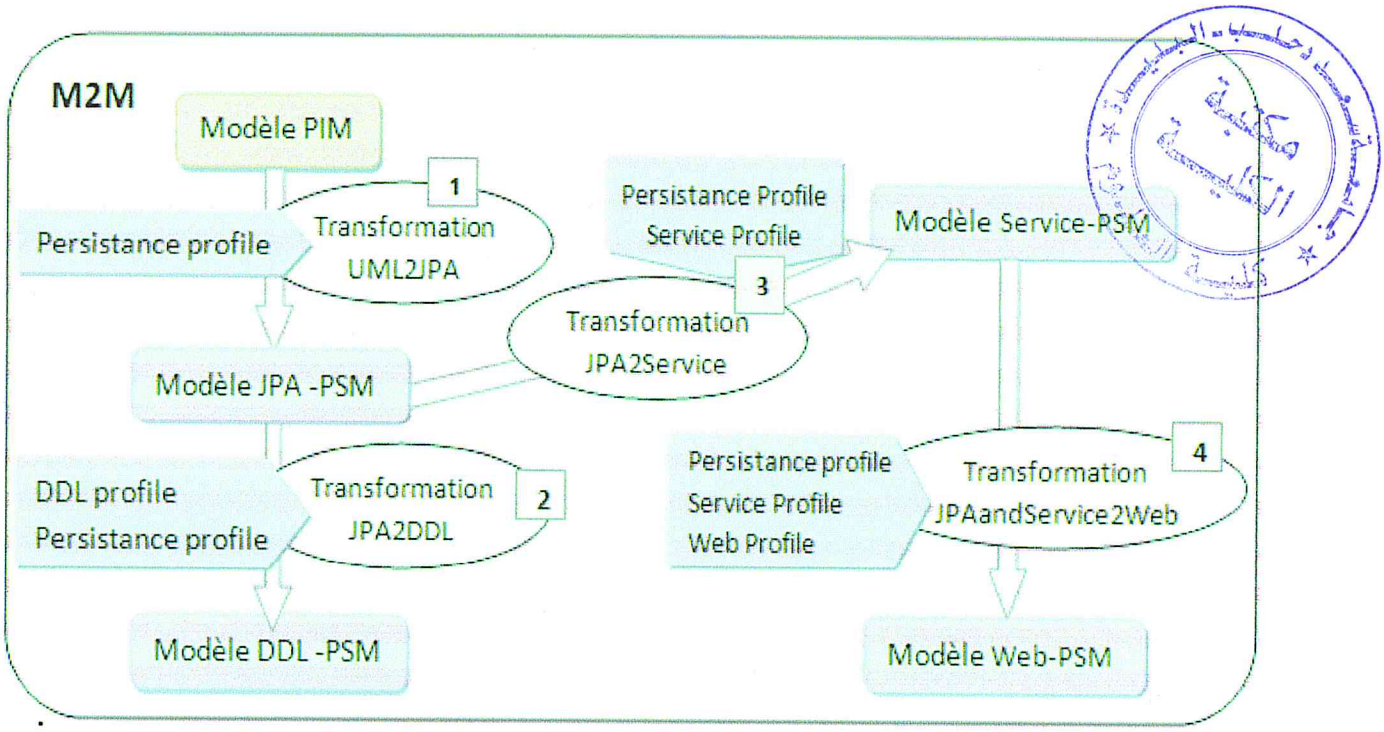


Figure 17: les transformations M2M dans le cadre du projet existant [02]

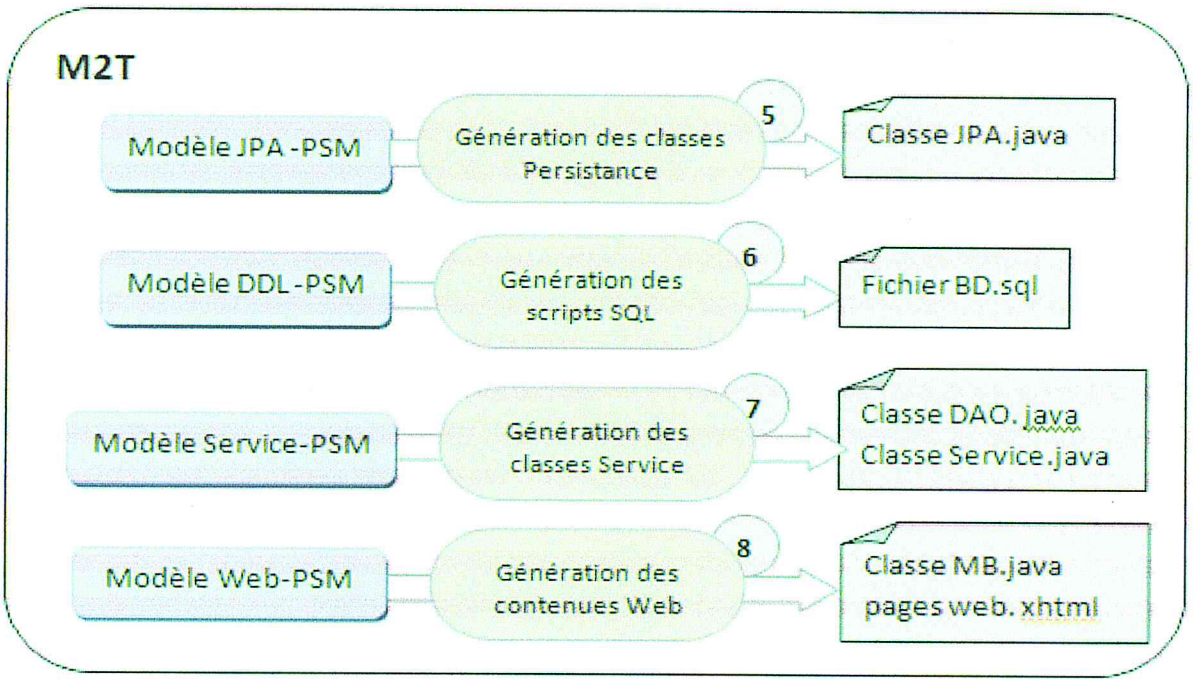


Figure 18: les générations de code source M2T dans le cadre du projet existant [02]

La Figure 17 présente les 4 transformations de modèles UML vers modèles UML à partir d'un diagramme de classe UML de domaine PIM pour arriver aux différents diagrammes de classe PSM selon les conventions de la plateforme JEE (PIM vers PSM) : [02]

- La première transformation nommée **UML2JPA** (N°1) est appliquée sur le modèle PIM (diagramme de classe) en utilisant le profil UML nommé « **Persistance profile** » pour générer le modèle JPA PSM.
- Le modèle JPA PSM généré représente les classes du modèle PIM sous forme de classes Java. Ces dernières représentent les classes de la couche persistance dans l'architecture de la plateforme JEE.
- Le modèle JPA PSM résultant de la première transformation doit subir trois transformations de modèle.
- La première transformation nommée **JPA2DDL** (N°2) se base sur deux profil UML : le profil UML de JPA « **persistance profile** » et le profil UML de DDL « **DDL profile** ». Elle permet d'obtenir le modèle DDL PSM. Ce dernier est un diagramme de classe qui représente le schéma de la base de données relationnelle de l'application générée.
- La deuxième transformation que doit subir le modèle JPA PSM est nommée **JPA2Service** (N°3), en utilisant le profile UML de JPA «**Persistance Profile**» et le profile UML «**Service Profile**». Elle permet de générer un diagramme de classe PSM représentant les deux packages dao et service. Ces deux packages contiennent des classes java.
  - Les classes du package **dao** assurent l'accès à la base de donnée.
  - Les classes du package **service** assurent les services de base (CRUD) sur une entité de l'application.
- La dernière transformation est nommée **JPAandService2Web** (N°4). Elle possède deux modèles sources : les diagrammes PSM JPA et Service. Cette transformation s'appuie sur un certain nombre de profils UML : le profil UML «**persistance profil**», «**Service profil**» et «**Web Profil**». Ces profils sont appliqués sur les modèles source afin de générer le diagramme de classe web. Ce dernier inclut les classes nommées Managed-Bean et les pages web de l'application générée.

La **Figure 18** représente les 4 générations (M2T) du code technique selon les conventions de la plateforme JEE : [02]

- La première génération (N°5) a comme modèle source le diagramme de classe JPA PSM et génère le code technique des classes persistance *« classe.java »*.
- La deuxième génération (N° 6) a comme modèle source le diagramme de classe DDL PSM et génère le script SQL de la base de données, *« fichier.sql »*.
- La troisième génération (N°7) a comme modèle source le diagramme de classe Service PSM. En sortie, elle génère deux packages **dao** et **service**. Chaque package contient des classes java *« classeDao.java / classeService.java»*.a
- La quatrième génération (N°8) a comme modèle source le diagramme de classe Web PSM. En sortie, elle génère les classes java nommées managed-Bean *« classeMB.java »* ainsi que les pages JSF et le fichier XML de configuration de l'application.

4. Travail demandé

Le travail qui nous a été confié a pour objectif de compléter le projet réalisé au sein de l'entreprise IconSoftware présenté ci-dessus. Ceci en intégrant également les diagrammes UML de comportement. Ainsi, l'AGL que nous allons développer sera à base de MDA en prenant en considération les digrammes UML de structuration et de comportement.

Notre travail consiste à spécifier :

- Les différents diagrammes UML de comportement à manipuler. Dans le cadre de notre travail, nous avons manipulé les modèles de comportement suivants : le diagramme des cas d'utilisation, le diagramme d'activité et le diagramme de séquence.
- Les différents profils à développer afin de faciliter les différentes transformations et génération de code.
- Les règles de passage d'un diagramme de comportement de domaine à un autre digramme de comportement de domaine à savoir :
  - Les règles de passage du diagramme des cas d'utilisation au diagramme d'activité.
  - Les règles de passage du diagramme des cas d'utilisation au diagramme de classe.

- Les règles de passage d'un modèle de domaine à un autre modèle spécifique à la plateforme JEE : les règles de passage du diagramme d'activité au diagramme de séquence.
- Les règles de passage d'un modèle PSM au code technique conforme à la plateforme JEE : les règles de passage du diagramme de séquence spécifique à la plateforme JEE au code source spécifique à la même plateforme.

5. Conclusion

Dans ce chapitre, nous avons présenté les besoins les besoins de l'entreprise **IconSoftware**, pour le développement d'un AGL de réalisation d'applications web en adoptons la démarche MDA. L'AGL a pour objectif d'aider les développeurs de l'entreprise à réaliser des applications web selon la plateforme JEE. Nous avons en premier présenté les travaux réalisé au sein de l'entreprise afin répondre à ces besoins. Par la suite, nous avons décrit le travail que nous devons réaliser afin de compléter les travaux existant et présenter notre apport. Dans le chapitre suivant, nous présentons une description détaillée de notre travail.

Partie II

Chapitre 02 :

Solution Proposée :

Développement d'un AGL basé sur les diagrammes UML de comportement

1. Introduction

Dans ce chapitre, nous présentons en premier à l'aide d'un ensemble de diagrammes de classes les différents méta-modèles correspondant aux modèles sources et cibles des transformations intégrées au sein de l'AGL. Deux principales catégories de transformation sont distinguées : Transformation de modèles PIM vers modèles PIM et Transformation de modèles PIM vers modèles PSM. Par la suite, nous décrivons l'ensemble des stéréotypes et profils UML que nous avons développé afin de faciliter les transformations et la génération de code selon la plateforme JEE. Enfin, nous présentons une description détaillée des transformations en décrivant les différentes correspondances entre modèles source et cible et une architecture globale de l'AGL développé.

La solution proposée s'applique sur les diagrammes de comportement (diagramme de cas d'utilisation, diagramme d'activité et le diagramme de séquence) et le diagramme de classe.

2. Les méta-modèles utilisés

Les différents méta-modèles correspondant aux modèles sources et cibles des transformations intégrées au sein de l'AGL sont présentés sous forme de diagrammes de classes. Chaque concept fondamental d'un méta-modèle est représenté à l'aide d'une classe (méta-classe). Chaque relation entre concepts dans un méta-modèle est décrite à l'aide d'une association au niveau du diagramme de classes.

2.1. Le méta-modèle du diagramme de cas d'utilisation

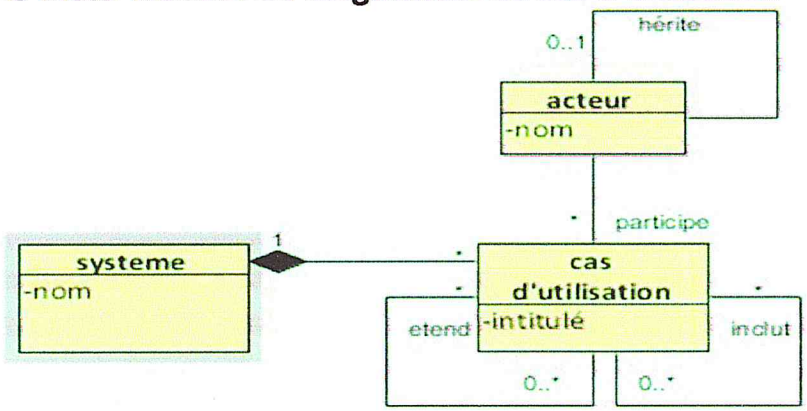


Figure 19: Représentation du méta-modèle du diagramme de cas d'utilisation [7]

Le méta-modèle du digramme de cas d'utilisation décrit les principaux concepts d'un diagramme de cas d'utilisation à savoir **Acteur, Système** et **Cas d'utilisation** ainsi que les **relations** entre eux **:** [7]

- Un **acteur** a un nom et est relié aux cas d'utilisation. Il peut **hériter** d'un autre acteur.
- Un **cas d'utilisation** a un intitulé et peut **étendre** ou **inclure** un autre cas d'utilisation.
- Le **système** possède lui aussi un nom. Il inclut tous les cas d'utilisation.

2.2. Le méta-modèle du diagramme d'activité

Un diagramme d'activité est utilisé pour afficher une séquence des activités. La description d'un cas d'utilisation par un diagramme d'activité correspond à sa traduction algorithmique. Une activité est l'exécution d'une partie du cas d'utilisation [29]. Dans ce qui suit, nous décrivons l'ensemble des concepts du méta modèle relatif au digramme d'activité présenté dans la **figure 20** [25]:

- **Activity :** Elle englobe toutes les actions, les flux de contrôle et d'autres éléments qui composent l'activity.

- **Structured Activity Node** : Cet élément (activité) représente une partie structurée de Activity. Il peut avoir des Control Edges qui lui sont connectés.

- **Activity edges :** Dans les diagrammes d'activité, un Activity edges est un lien orienté entre deux nœuds d'activité. Lorsqu'une action spécifique d'une activité est terminée, l'Activity edges continue l'écoulement du flux à l'action suivante dans la séquence des activités. Il existe deux types d'Activity edges à utiliser dans la modélisation :

  - **Control flow :** représente le contrôle à effectuer d'un nœud à un autre.
  - **Object flow :** représente le flux de données ou d'objet d'un nœud à un autre.

- **Object nodes:** Dans les diagrammes d'activité, un Object Node est une classe d'activité abstraite qui permet de définir le flux de l'objet dans une activité. Un Object Node indique que l'instance d'un classificateur peut être disponible à un moment particulier à l'activité.

- **Initial Node:** Représente le début d'un ensemble d'actions ou d'activités.

- **Final-Activity Node:** Il est utilisé pour arrêter tous les flux d'une activité.

- **Opaque Action :** Représente une action ou activité externe par rapport au diagramme d'activité courent.

- **Opaque expression :** Représente une expression ou un contrôle qui ne doit être interpréter.

- Les différents **nœuds de contrôles :**

  - **Decision Node:** Il est utilisé pour représenter un test avec une condition pour assurer que Control flow ou Object flow suit un seul chemin.

  - **Merge Node:** Il est utilisé pour regrouper les différents chemins de décision crées par les Decision Node.

  - **Fork Node:**Il est utilisé pour partager les comportements des activités en un ensemble de flux parallèles.

  - **Join Node:** Il est utilisé pour regrouper un ensemble de chemins parallèles des activités.

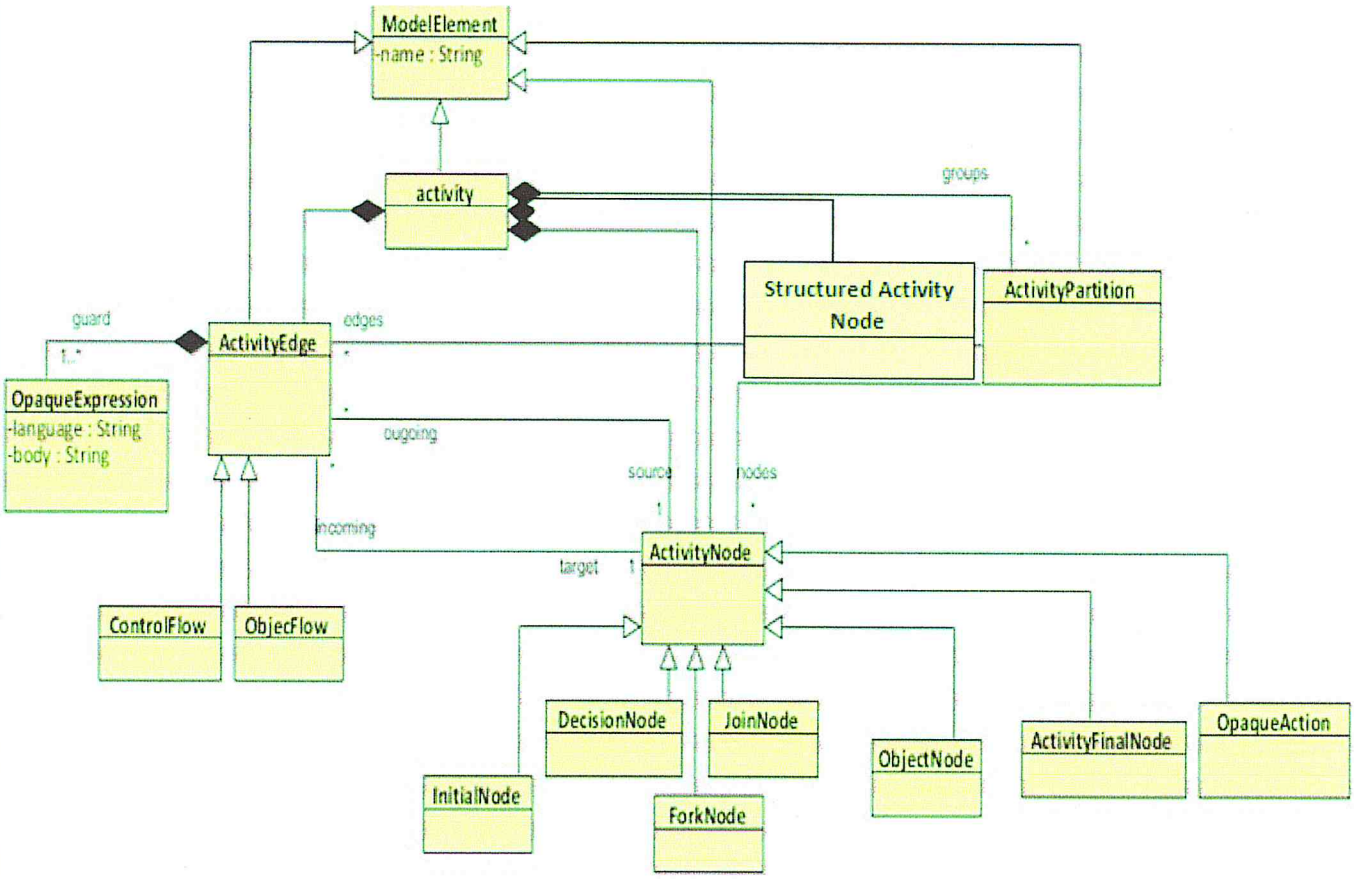


Figure 20 : Représentation du métamodèle de diagramme d'activité UML2 [29]

2.3. Le Méta-modèle du diagramme de séquence

OMG fournie au diagramme séquence UML 2 un métamodèle, qui affiche graphiquement la syntaxe abstraite en termes de diagramme de classes. Il montre les constructions syntaxiques essentielles d'un diagramme de séquence, afin de faciliter les efforts des utilisateurs pour comprendre la notation. [27]

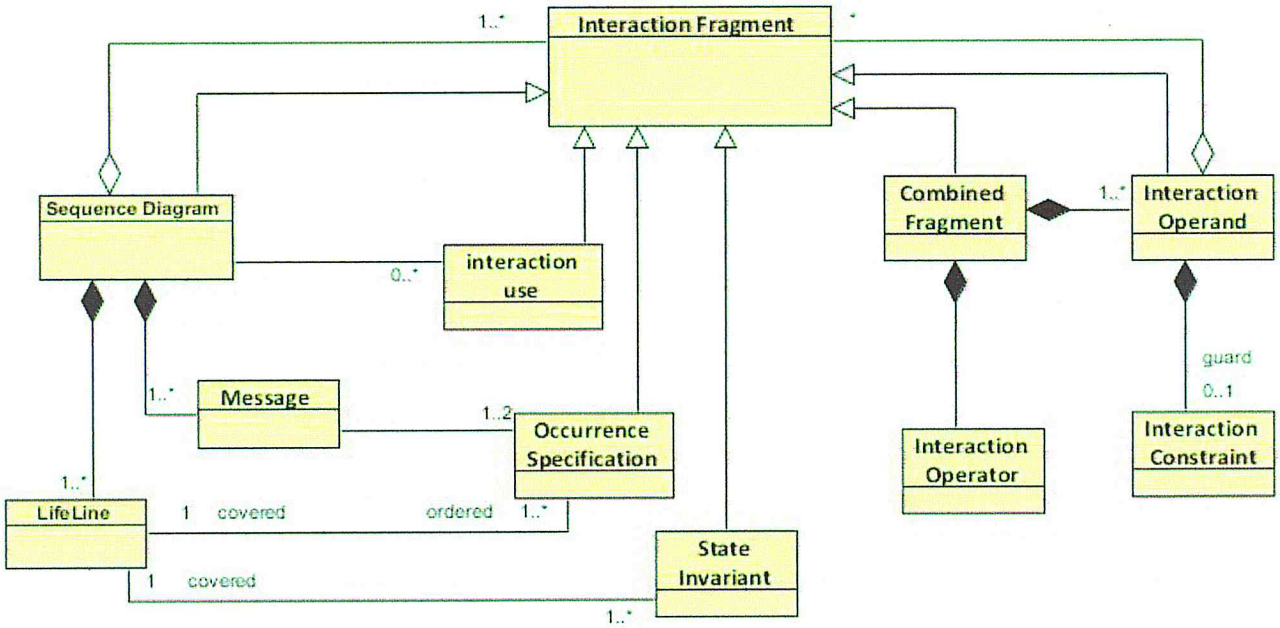


Figure 21 : Représentation du métamodèle de diagramme de séquence UML2 [01]

L'ensemble des éléments de ce méta modèle sont décrits ci-dessous :

- **InteractionFragment** : est une classe abstraite pour l'interaction, CombinedFragment, InteractionOperand, InteractionUse et le maintien, et aussi pour OccurrenceSpecification, et StateInvariant.

- **Sequence Diagram :** est une sorte d'InteractionFragment. qui se concentre sur l'échange d'un ou plusieurs messages entre une ou plusieurs Lignes de vie. Un diagramme de séquence peut avoir un ou plusieurs InteractionFragments, qui peut être un InteractionUse, un OccurrenceSpecification, un StateInvariant, un CombinedFragment, et un InteractionOperand.

- **LifeLine :** Les lignes verticales en pointillés, appelées *Lignes de vie*, représentent l'individu participants sur une période de temps dans l'interaction, chaque ligne de vie doit avoir un nom. Ces individus (lignes de vies) sont communiqués par des messages.

- **Message:** Les lignes horizontales entre Lifelines sont appelés *messages*. Un message est envoyé à partir de l'appel (source) Survie à la prétendue(Cible) Lifeline. Un message peut être un signal ou une opération appelée.

- **OccurrenceSpecification** : les deux extrémités d'un message, c'est le point d'intersection entre le point de départ et le point d'arrivé d'un message sur la ligne de vie.

- **InteractionUse :** se réfère à une interaction, elle permet au diagramme de séquence de désigner un autre diagramme de séquence. Il copie le contenu de la référence. L'execution d'une InteractionUse a l'effet est la même que l'exécution du diagramme de séquence référencée.

- **StateInvariant** : est une contrainte d'exécution sur l'un des participants de l'interaction (ligne de vie). La contrainte est évaluée immédiatement avant l'exécution de la prochaine OccurrenceSpecification sur la ligne de vie. Si la contrainte est vraie, la trace est une trace valide, sinon, la trace est invalide.

- **CombinedFragment :** une expression d'InteractionFragment. Il est définie par : InteractionOperator et une ou plusieurs InteractionOperands.

- **InteractionOperator** : est une énumération désignant les différents types d'opérateurs de CombinedFragments. L'InteractionOperand définit le type d'opérateur d'un CombinedFragment. Les valeurs littérales de cette énumération sont les suivants: Alt, opt, par, loop, critical, neg, assert, strict, seq, ignore et consider.

- **InteractionOperand** : est contenue dans un CombinedFragment. Un InteractionOperand représente l'un des opérandes de l'expression proposée par le CombinedFragment englobant.

- **InteractionConstraints** : est utilisée pour faire des gardes sur chaque InteractionsOperands d'un CombinedFragment.

2.4. Le Méta-modèle du diagramme de classes et d'un profil UML [02]

La figure 22 illustre la partie du méta-modèle UML2.3 utilisé dans le cadre du projet existant et contenant les méta-classes relatives aux profils. Ces méta-classes correspondent de fait à la définition d'un profil. Un modèle instance de ces méta-classes correspond à la définition d'un nouveau profil et non à l'application d'un profil existant à un modèle UML. Dans ce méta-modèle :

- La méta-classe **Profile** hérite de la méta-classe Package. Cela signifie que les définitions de nouveaux profils sont maintenant considérées comme étant modèles UML et plus précisément comme des *packages*.
- La méta-classe **Stereotype** hérite de son côté de la méta-classe *Class*. Cela signifie que les définitions de stéréotypes sont considérées comme étant des classes UML. Le nom de la classe correspond au nom du stéréotype.
- Étant donné qu'un stéréotype est une classe, il peut contenir des propriétés. On considère alors qu'un stéréotype contient des **attributs**. Les éléments stéréotypés peuvent attacher des valeurs à ces propriétés.

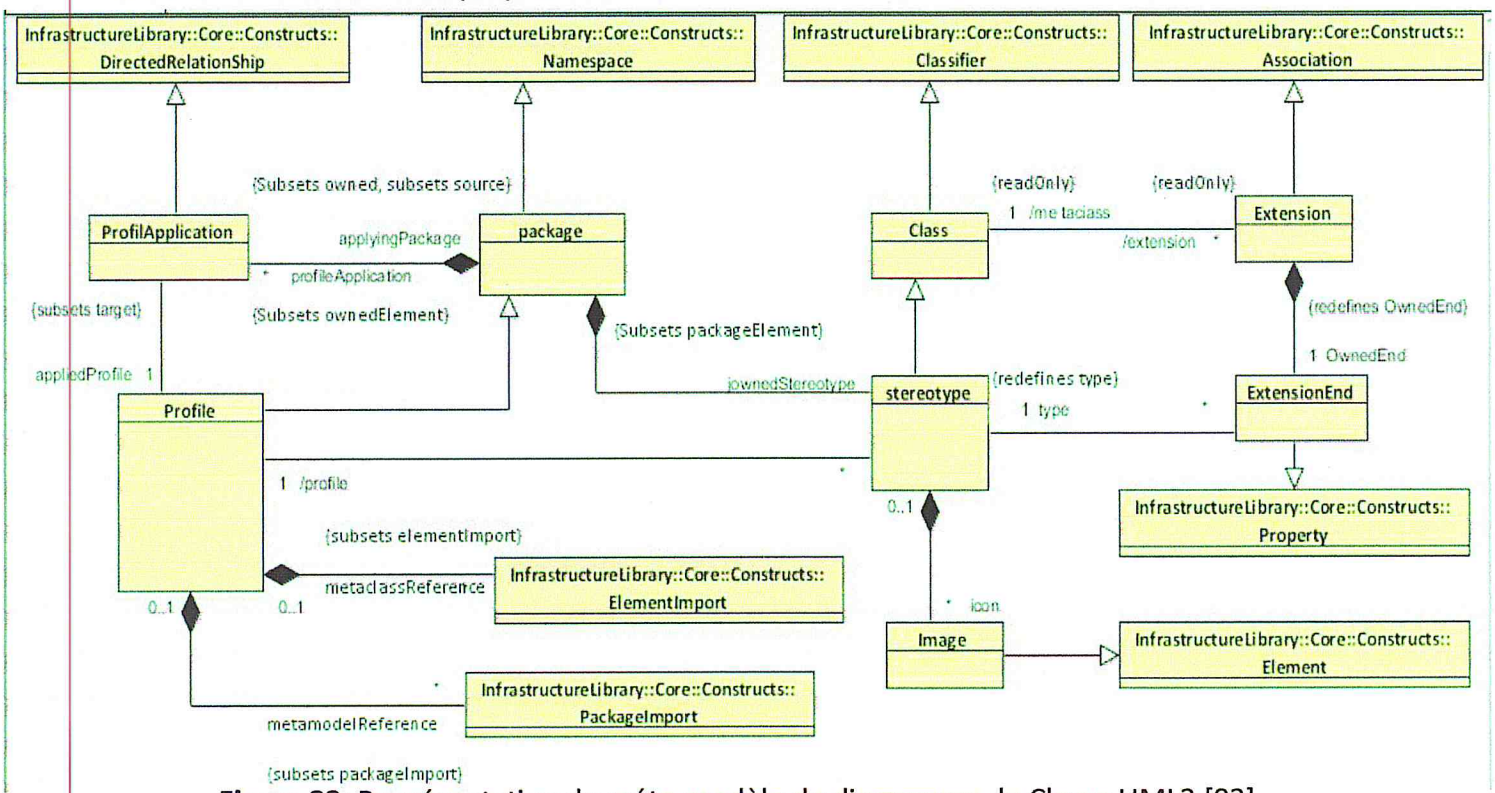


Figure 22: Représentation du méta-modèle de diagramme de Classe UML2 [02]

3. Les Profils UML développés

Nous avons développé un certain nombre de profils UML afin de faciliter les différentes transformations de modèles (PIM vers PIM et PIM vers PSM) ainsi que la génération de code du modèle PSM selon les conventions JEE.

3.1. Profil_UseCase_Classe

C'est le profil UML utilisé pour faciliter la transformation du modèle UML PIM (diagramme cas d'utilisation) vers le modèle UML PIM (diagramme de classe).

Il est Possible d'appliquer sur les use case du diagramme cas d'utilisation un stéréotype nommé «***Business Model***» pour spécifier le *business Object* (la classe service) qui sera générée à partir de ce use case comme illustre la figure ci-dessous :

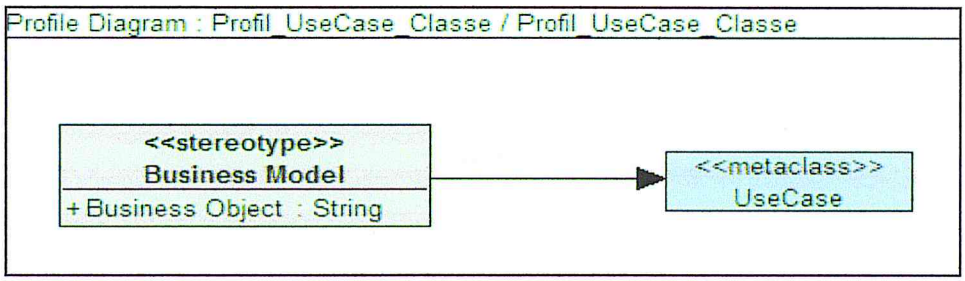


Figure 23: Profil_UseCase_Classe

3.2. Profil_UseCase_Activity

C'est le profil UML utilisé pour faciliter la transformation du modèle UML PIM (diagramme de cas d'utilisation) vers le modèle UML PIM (diagramme d'activité).

Il est Possible d'indiquer sur les use case du diagramme de cas d'utilisation :

- L'ordre d'exécution des use case par l'application de l'un des stéréotypes « **ordre1** », « **ordre2** », « **ordre3** » et « **ordre4** ».
- Le use case global par l'application du stéréotype « **à revoir** » à fin de faciliter le raffinement du diagramme d'activité généré.
- Le use case final par l'application du stéréotype « **niveau fin** » afin de générer le nœud final du diagramme d'activité.

Les relations **include** entre cas d'utilisation peuvent être stéréotypées par l'application du stéréotype « **useSource** » afin de récupérer le use case source de cette relation. Les relations acteurs du système et cas d'utilisation (associations) peuvent être stéréotypées par l'application du stéréotype « **membreEnd** » afin de référencer l'acteur source et use case cible de cette association (voir figure 24).

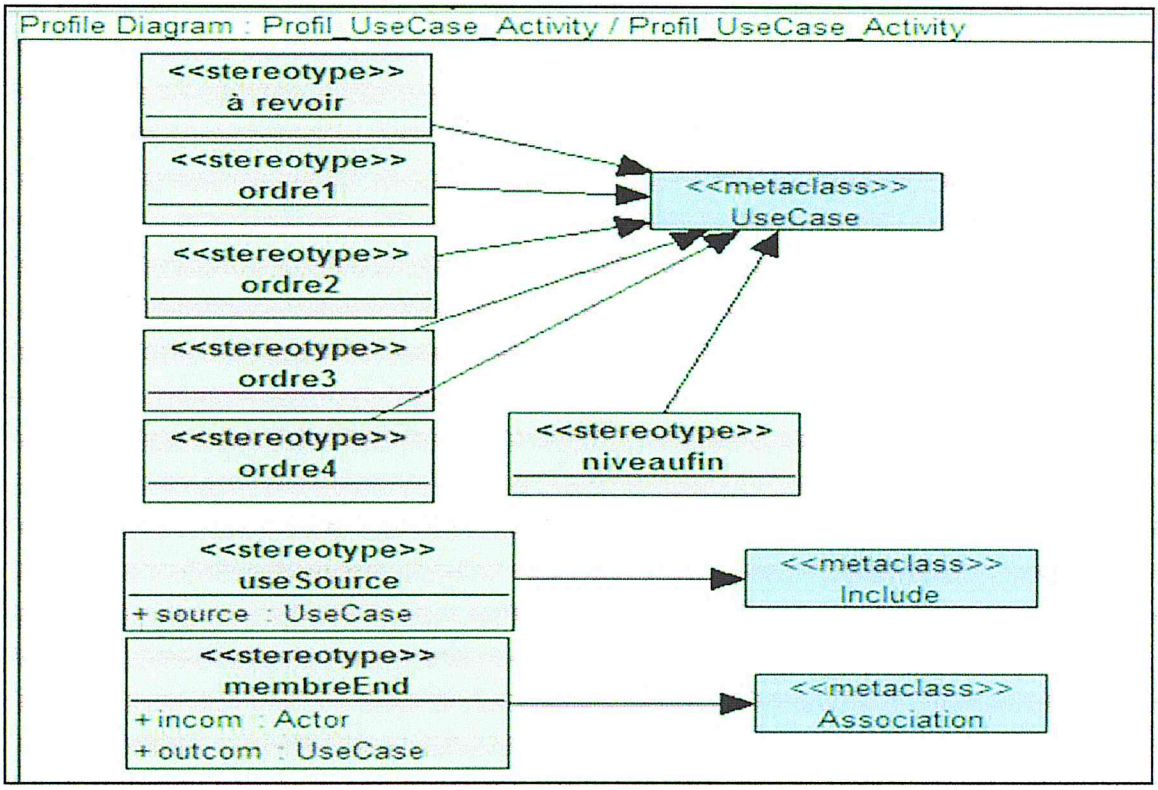


Figure 24: Profil_UseCase_Activity

3.3. Profil_Activity_Sequence

C'est le profil UML utilisé pour faciliter la transformation du modèle UML PIM (diagramme d'activité) vers le modèle UML PSM (diagramme de séquence).

Ce profil permet d'indiquer sur les activités du diagramme d'activité, les différentes méthodes et classes service liées à cette activité à l'aide du stéréotype **Business Model** ainsi que d'indiquer le service à créer par le stéréotype **ActivityModel** (voir figure 25).

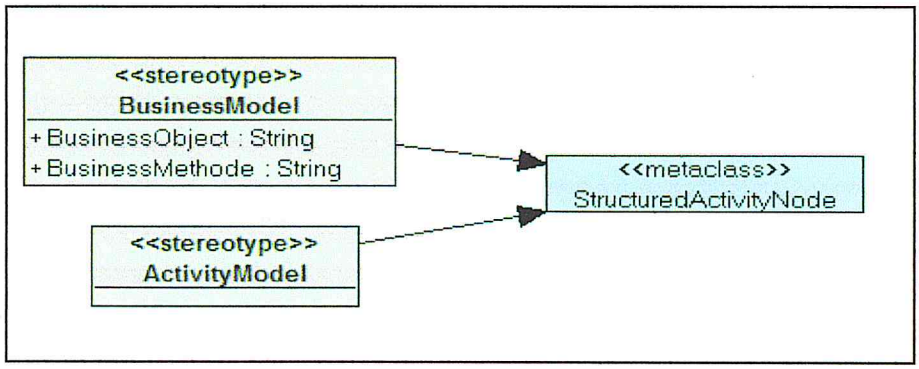


Figure 25: Profil_Activity_Sequence

3.4. Profile Sequence2Code

C'est le profil UML utilisé pour faciliter la génération de code source à partir du modèle UML PSM (diagramme de séquence).Il permet d'indiquer sur :

- Une ligne de vie la couche (Service ou web selon JEE) qui la comprend. Ceci en appliquant l'un des stéréotypes **ManagedBean, Service** ou **dao.**
- les messages, le type de retour de la méthode (pour les messages de type 'Synchronous call') et le type de la variable à créer (pour les messages de type 'reply'. Ceci en appliquant stéréotypes **TypeObject** ou **PrimitiveType.**

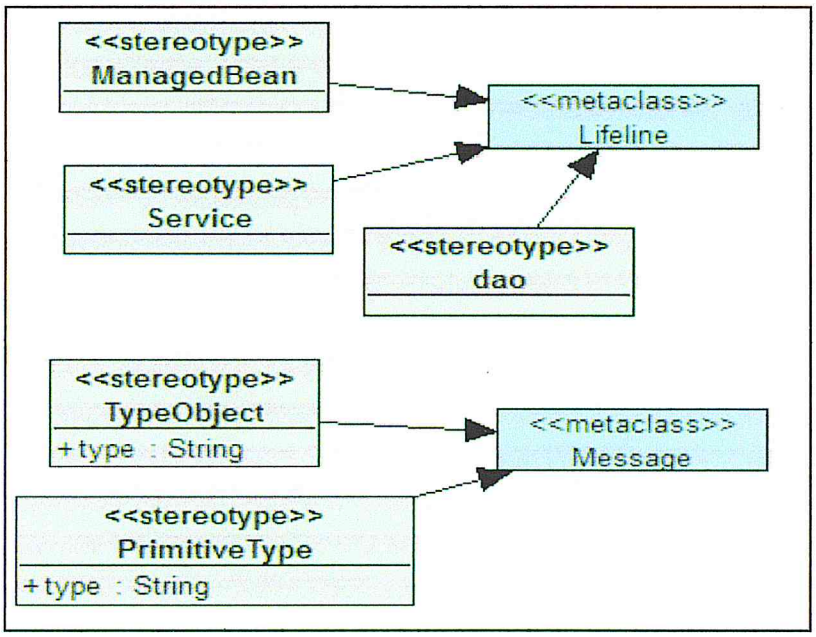


Figure 26: Profil_Sequence_code

4. Correspondances entre les méta-modèles utilisés

Une fois les différents méta-modèles relatifs à l'ensemble des modèles sources et cibles ont été décrits, nous allons présenter dans ce qui suit, les différentes correspondances entre les différents éléments de ces méta-modèles. Ces correspondances rentrent dans le cadre de deux principales catégories de transformation :

- Transformation de modèles PIM vers modèles PIM ;
- Transformation de modèles PIM vers modèles PSM.

4.1. Transformation de modèles PIM vers modèles PIM

Cette catégorie de transformation inclut deux transformations : la transformation d'un diagramme cas d'utilisation vers un digramme de classe ainsi que la transformation d'un diagramme cas d'utilisation vers un diagramme d'activité.

4.1.1. Transformation d'un diagramme cas d'utilisation vers un diagramme de classes

Cette transformation à comme modèle source le diagramme de cas d'utilisation dont le méta-modèle a été présenté précédemment (voir figure 19). En sortie, elle génère un diagramme de classe dont le méta-modèle est présenté dans la figure 22. Afin de faciliter cette transformation, nous utilisons le profil UML **Profil_UseCase_Class** (voir figure 23). Ce dernier inclut un ensemble de stéréotypes à appliquer sur les différents éléments du modèle source « diagramme de cas d'utilisation ».

Ainsi, chaque Use Case stéréotypé par *«Business Model»* correspond à une classe qui porte le nom sauvegardé dans la propriété «Business Object» du stéréotype.

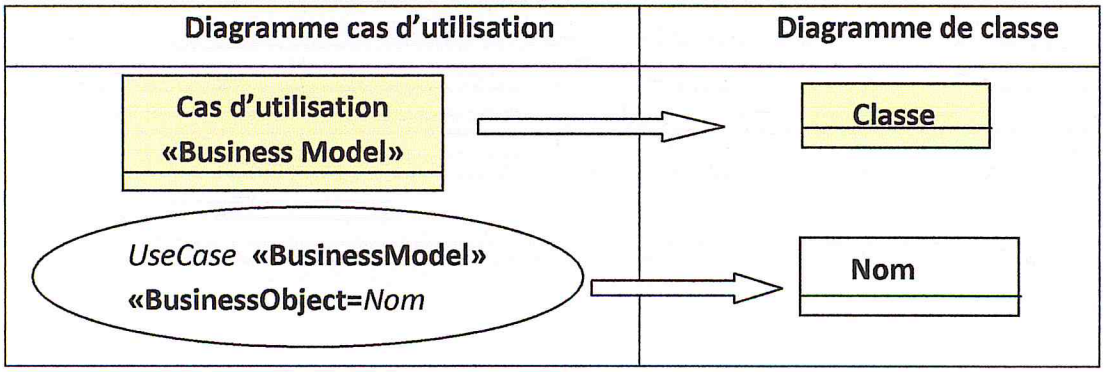


Tableau 01: Les correspondances de transformation du diagramme cas d'utilisation vers le diagramme de classe

Nous avons comme résultat de la transformation d'un diagramme cas d'utilisation vers un diagramme de classe, un diagramme de classe de domaine basique qui doit être raffiner par l'Analyste par l'ajout des propriétés et des relation entre classe, ce diagramme raffiné sera comme modèle source de AGL de structuration existant et qui nous donne comme résultat après l'exécution des transformation de modèles un diagramme de classe PSM spécifique à la plateforme JEE, que nous allons utiliser dans notre travail pour réaliser la transformation d'un diagramme d'activité vers un diagramme de séquence.

4.1.2. Transformation d'un diagramme cas d'utilisation vers un diagramme d'activité

Cette transformation à comme modèle source le diagramme de cas d'utilisation dont le méta-modèle est présenté dans la figure 19. Nous avons manipulé les méta-classes : système, acteur et cas d'utilisation, et les méta-associations : la relation Include entre deux cas d'utilisation et l'association (la relation entre un acteur et un cas d'utilisation). Le modèle cible de cette transformation est un diagramme d'activité dont le méta-modèle est présenté dans la figure 20. Nous avons manipulé les métaclasses : activity, initialNode, StructuredActivityNode, ActivityFinalNode, ControlFlow.

Cette transformation repose sur le profil UML **Profil_UseCase_Activity** (figure 24). Selon ses besoins, l'Analyste peut appliquer les différents stéréotypes de ce profil sur le diagramme source « cas d'utilisation ».

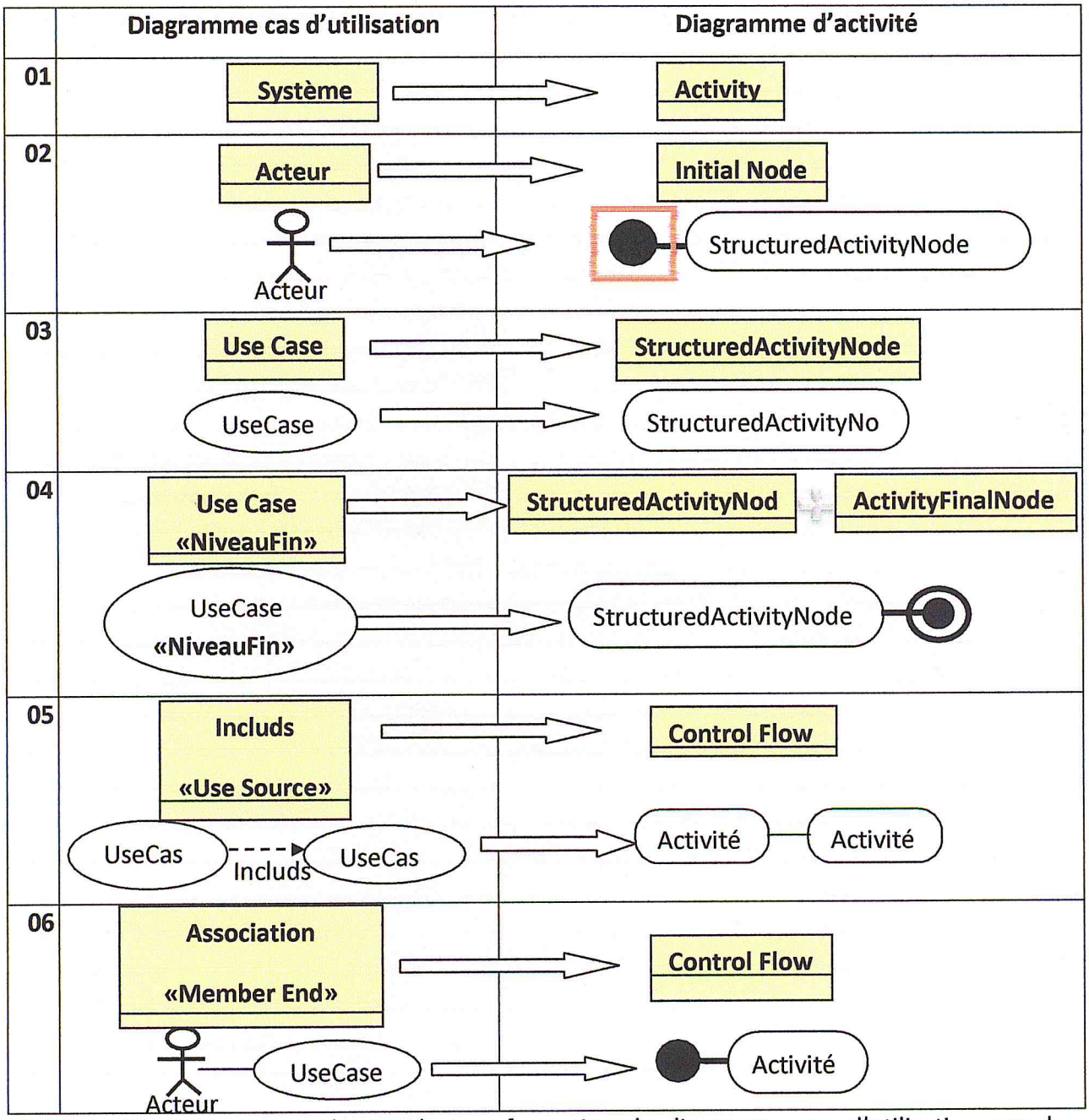


Tableau 02: Les correspondances de transformation du diagramme cas d'utilisation vers le diagramme d'activité

- **01 :** le *système* qui comporte tout les éléments du diagramme de cas d'utilisation devient *Activity* qui représente l'élément père pour le diagramme d'activité et qui contient tous ses éléments.
- **02 :** l'*acteur* du diagramme cas d'utilisation devient le *nœud initial* du diagramme d'activité résultant.
- **03 :** un cas d'utilisation devient une activité (StructuredActivityNode) au niveau du diagramme d'activité.
- **04 :** un cas d'utilisation stéréotypé *«Niveau Fin»* devient la dernière *activité* et le *nœud final* du diagramme d'activité généré.
- **05 :** la relation *Includs* entre deux cas d'utilisation devient une relation *ControlFlow* entre deux activités (StructuredActivityNode), ordonnées par le stéréotype *«Use Source»*.
- **06 :** la relation (association) entre un acteur et un cas d'utilisation est transformé en une relation *ControlFlow* entre le nœud initial et une activité de diagramme d'activité à l'aide du stéréotype «Member End».

4.2. Transformation de modèles PIM vers modèles PSM

Cette catégorie de transformation à pour but de générer un diagramme de séquence qui va être utilisé comme modèle source dans l'étape de génération de code source.

4.2.1. Transformation d'un diagramme d'activité vers un diagramme de Séquence

Cette transformation génère un digramme de séquence qui est conforme au méta-modèle de la figure 21. Pour réaliser cette transformation nous avons manipulé les méta-classes : Interaction Fragment, Lifeline, BehaviorExecutionSpecification, message, OccurenceSpecification. Le diagramme de séquence est généré à partir du diagramme d'activité produit précédemment.

Cette transformation repose sur le profil UML **Profil_ Activity_Sequence** (Figure 25). Selon ses besoins, l'Analyste peut appliquer les différents stéréotypes de ce profil sur le diagramme d'activité source.

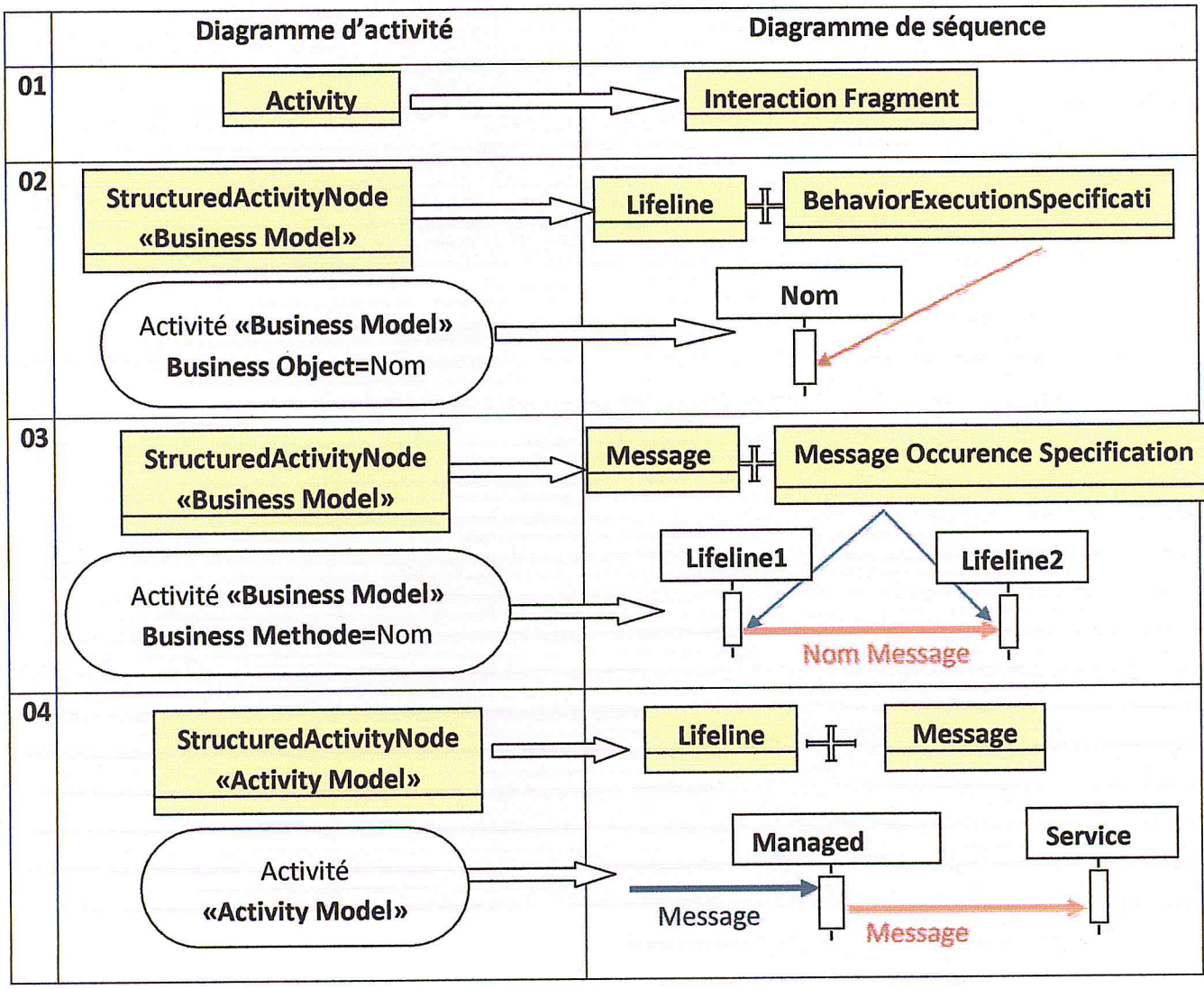


Tableau 03: Les correspondances de transformation du diagramme d'activité vers le diagramme de séquence

En utilisant le diagramme de classe PSM généré de l'AGL de structuration existant et en appliquant le stéréotype **«Business Model»** (de *Profil_UseCase_Activity*) sur une activité, nous pouvons transformer cette dernière en une lifeline « ligne de vie » qui porte le nom sauvegardé dans la propriété **Business Object** du stéréotype.

4.3. La génération de code source JEE a partir du Modèle PSM (diagramme de séquence)

Notre AGL génère du code source Java EE à partir du diagramme de séquence produit précédemment qui est conforme au méta-modèle présenté dans la **figure 21**

Pour faciliter la génération de code source JEE, nous avons développé le profil UML **Profil_Sequence_code (Figure 26)**. Selon ses besoins, le développeur applique les différents stéréotypes de ce profil sur le diagramme d'activité source.

Dans le tableau ci-dessous, nous présentons les différentes correspondances entre les éléments du diagramme de séquence source (méta-classes de son méta-modèle) et des concepts de la plateforme JEE.

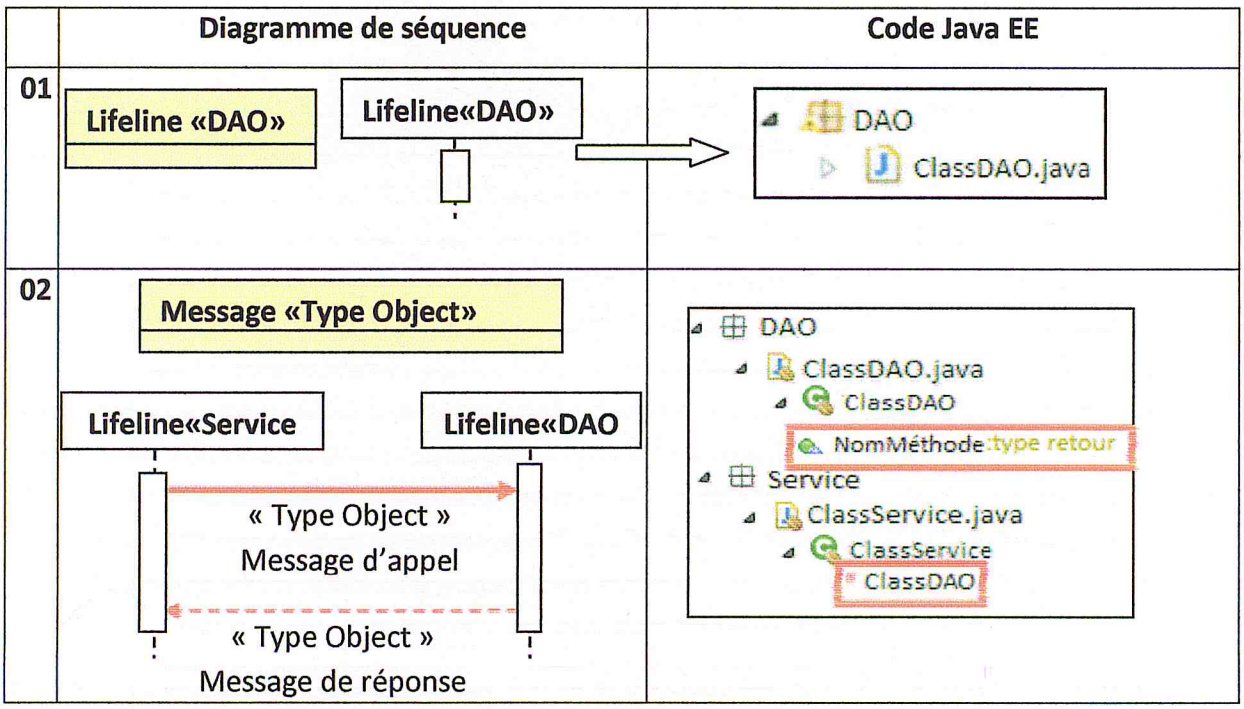


Tableau 04 : les correspondances de génération de code source JEE à partir d'un diagramme de séquence PSM

- **01 :** Chaque ligne de vie du diagramme de séquence source correspond a un Package et une Classe java selon le stéréotype appliqué sur cette ligne de vie : DAO, ManagedBean ou Service.
- **02 :** A partir d'un message du digramme de séquence source portant le stéréotype **Type Object** ou **Primitive Type**, il est possible de récupérer le type de retour de la méthode (pour les messages d'appel) et le type de la variable à créer (pour les messages de réponse).

➢ Le message d'appel représente:

- Une définition de méthode (la méthode *NomMethode* dans l'exemple) dans la classe qui représente la ligne de vie qui reçoit le message (*ClassDAO* dans l'exemple) ;
- Une instanciation de la ligne de vie qui reçoit le message d'appel (ClassDAO dans l'exemple) dans la ligne de vie qui fait l'appel de la méthode (ClassService dans l'exemple).

➢ Le message de réponse représente :

- Nous sommes mis en accord sur une convention de nommage pour ce type de message : *NomVariable=NomMethode*.
- Chaque message de réponse représente une déclaration de la variable spécifié dans le message qui reçoit le retour de l'appel de méthode.

5. Architecture de notre AGL base sur les diagrammes UML de structuration et de comportement

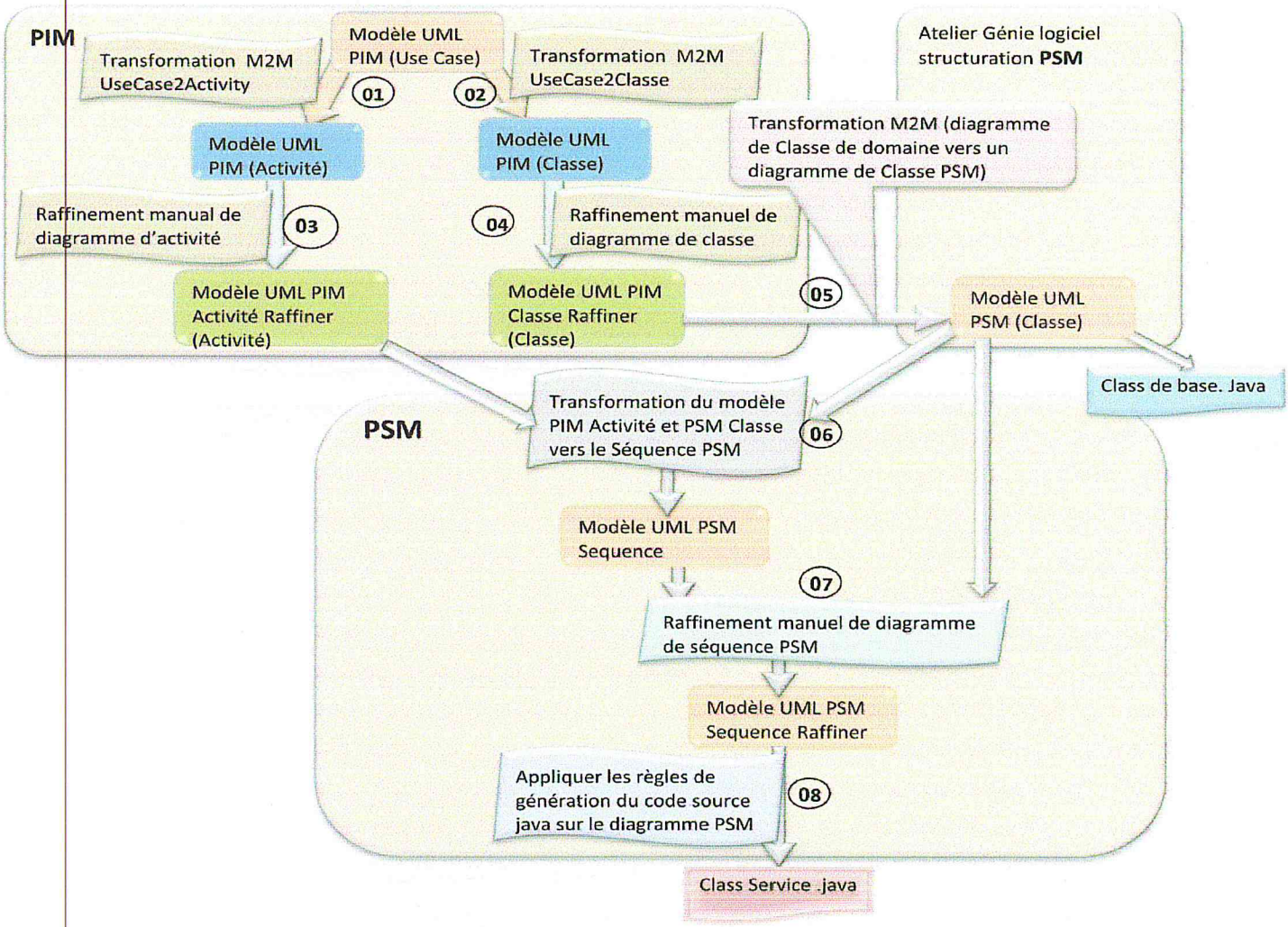


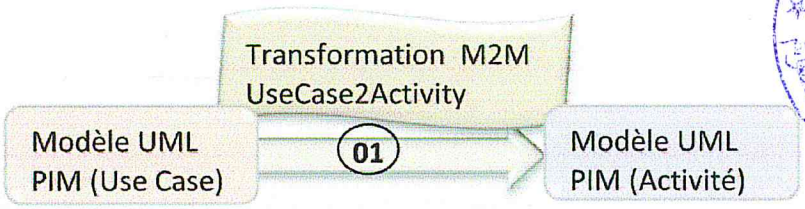
Figure 27 : Les étapes de transformation et de génération de l'AGL des diagrammes de comportements

La Figure ci-dessus présente les étapes de transformations principales que doit subir un modèle UML PIM pour arriver à un modèle UML PSM puis au code source Java.

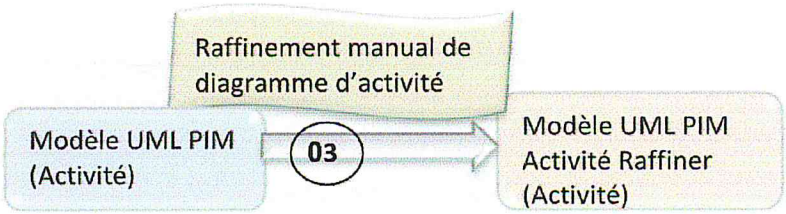
- Le model PIM source est un diagramme UML de cas d'utilisation (use Case) qui fournit la première vue dynamique sur l'application. Ce diagramme doit être conforme au Meta-modèle UML (figure 19).

- Un profil UML est appliqué sur ce modèle. Le modèle stéréotypé est toujours un modèle PIM indépendant de toute plateforme.
- le modèle PIM source (diagramme de cas d'utilisation), doit subir les deux premières transformations M2M sont appliquées en se basant sur deux profils UML :

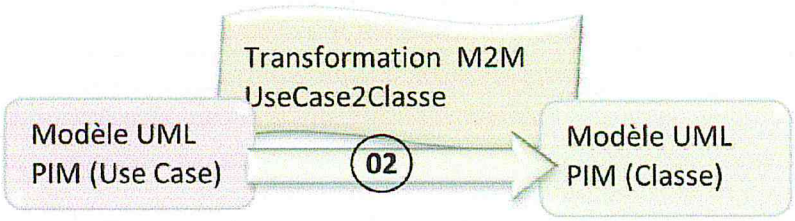
- La première transformation est appliquée pour générer le premier modèle PIM cible (diagramme d'activité). Elle est nommée UseCase2Activity (numéro 01 dans la figure) et se base sur le profil UML nommé «*Profil-UseCase_Activity*». Ce dernier est appliqué sur le diagramme use case source de cette transformation.



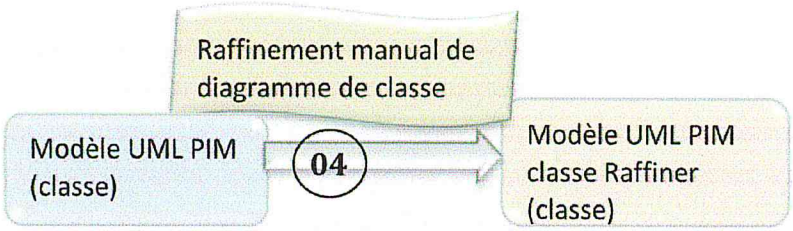
- Le modèle PIM de domaine (diagramme d'activité) résultat de cette transformation doit subir des raffinements pour le compléter (numéro 03 dans la figure), afin de l'utiliser comme source dans une transformation de PIM vers PSM.



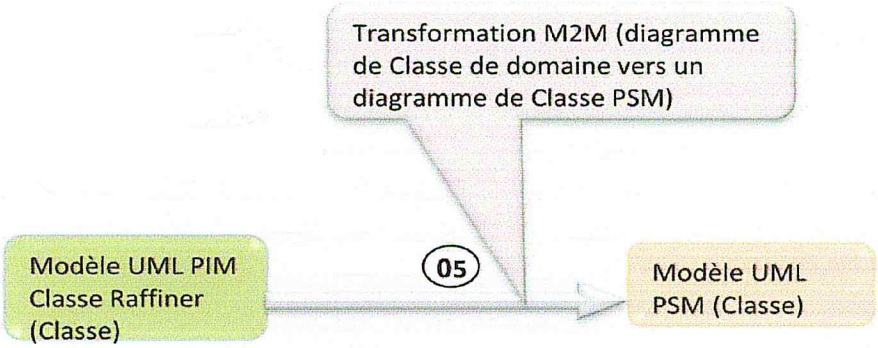
- La deuxième transformation est appliquée pour avoir le deuxième modèle PIM cible (diagramme de Classe de domaine). Elle se base sur le profil UML «*Profil_UseCase_Classe*» et est réalisée en exécutant le scripte de transformation nommé UseCase2Class (numéro 02 dans la figure).



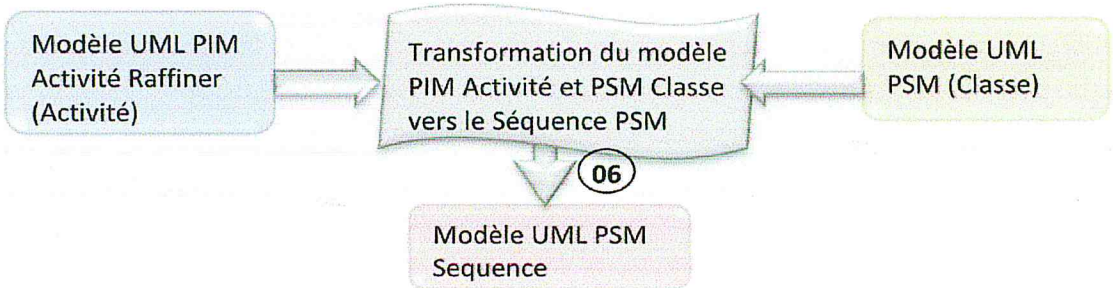
- le modèle PIM de domaine (diagramme de Classe) résultat de cette transformation doit subir des raffinements pour le raffiner et le compléter (numéro 04 dans la figure).



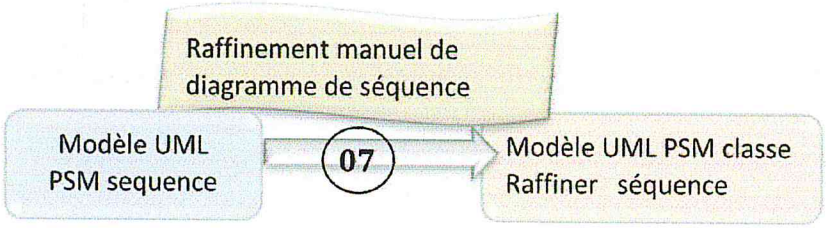
- Le diagramme de classe raffiné est utilisé comme modèle source pour la transformation M2M (numéro 05 dans la figure) de l'*AGL des diagrammes de structurations* au niveau de l'entreprise **IconSoftware.** Le modèle résultat de cette transformation est un diagramme de classe PSM spécifique à la plateforme JEE et au design pattern MVC.



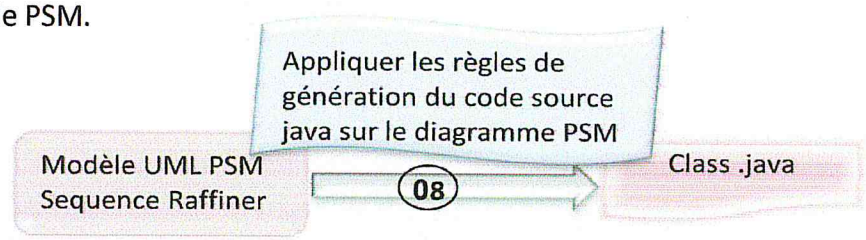
- Pour avoir le modèle PSM (diagramme de séquence), on doit appliquer la troisième transformation M2M de notre atelier (numéro 06 dans la figure). Cette transformation utilise comme source le modèle PIM UML (diagramme d'activité raffiné) et le diagramme de classe PSM (résultat de l'intégration de l'AGL des diagrammes de structuration). Sur le modèle activité raffiné est appliqué le Profile UML «*Profil_Activity_Sequence*».



- Le diagramme de séquence généré a besoin d'un ensemble de raffinement (plus l'application des profils PSM définis par l'Architect) pour le compléter (numéro 07 dans la figure).



- Le diagramme de séquence raffiné est utilisé comme source dans l'étape de la génération du code source Java (numéro 08 dans la figure), après avoir appliqué les règles de génération sur ce modèle PSM.



- On va avoir comme résultat de cette étape un package *DAO* qui contient les *classe.java* qui assure les services de base.
- un package *Service* contient les *classe.Java* service (métier) qui permet d'enrichir l'aspect métier du code source de l'application web générer
- un package *Web_facesBean* contient l'ensemble des *classe.java* qui sont : les Managed-Bean qui représente la classe Java intermédiaire entre les classe service.java et les pages Web de l'application web JEE.

6. Conclusion

Dans ce chapitre, nous avons présenté l'ensemble ; des méta-modèles correspondant aux modèles UML sources et cibles des différentes transformations et génération, les deux catégorie de transformations de modèles PIM vers PIM et PIM vers PSM, en suite les profils développés afin de faciliter les transformations et génération de code source selon la plateforme JEE. Et nous avons terminé par une présentation détaillée des différentes correspondances entre les modèles sources et cibles des différentes transformations.

Partie III :

Mise en œuvre de l'AGL

Partie III

Chapitre 01 :

Conception de l'Atelier génie logiciel

1. Introduction

Dans le chapitre précédent, nous avons présenté la solution proposée nécessaires à la mise en place d'un AGL complet pour la génération d'application web JEE en appliquant la démarche MDA. Dans ce chapitre nous présentons la conception de notre atelier de génie logiciel pour compléter le travail existant afin de générer une application web Java EE.

2. Processus de développement

Un processus définit une séquence d'étapes, partiellement ordonnées, qui concourent à l'obtention d'un système logiciel ou à l'évolution d'un système existant.[38]

Le Processus Unifié **UP (Unified Process)** répond bien à ces exigences. C'est un processus de développement moderne, itératif, efficace sur des projets informatiques de toutes tailles. [39]

Il couvre l'ensemble des activités, depuis la conception du projet jusqu'à la livraison de la solution. Intégrant une organisation de projet et suit une méthodologie utilisant UML.[38]

L'objectif du Processus unifié UP est de guider les développeurs vers l'implémentation et le déploiement efficaces de systèmes répondant aux besoins des clients. Cette efficacité mesure en termes de coûts, de qualité et de délai de fabrication. [39]

On a opté de suivre le processus UP (UP pour Unified Process) qui est un processus de développement qui possède les caractéristiques suivantes : [39]

- **Itératif et incrémental**

Une **itération** prend en compte un certain nombre de cas d'utilisation et traite en priorité les risques majeurs.

Un **incrément** correspond à l'avancement dans les différents stades de développement.

- **Pilotés par les cas d'utilisation** L'objectif d'un système logiciel est de rendre service à ses utilisateurs. Pour réussir la mise au point d'un système, il importe, par conséquent, de bien comprendre les désirs et les besoins de ses futurs utilisateurs.

- **Centré sur l'architecture** le démarrage du processus, on aura une vue sur L'architecture logicielle qui correspond aux aspects statiques et dynamiques les plus significatifs du système

3. Diagramme des Cas d'Utilisation

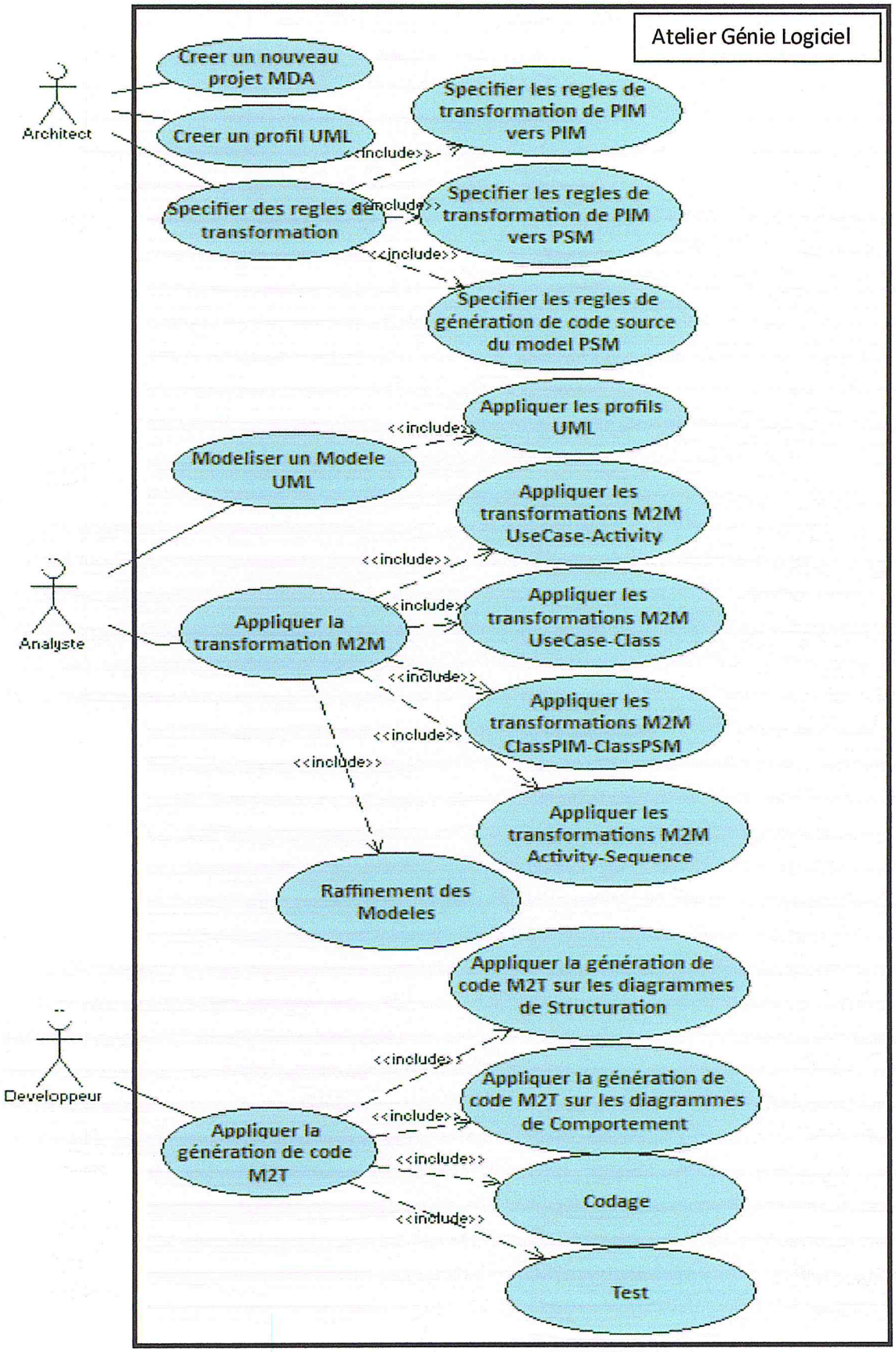
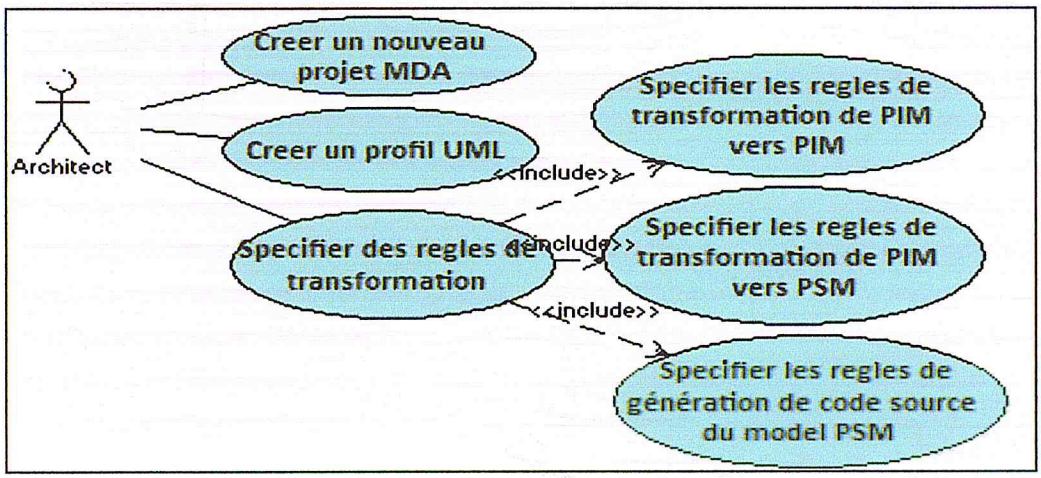


Figure 28 : Diagramme des cas d'utilisation

3.1. Description des cas d'utilisation

3.1.1. Description des fonctionnalités de l'Architect



Acteur Principal	Architect
Fonctionnalités	▪ Créer un nouveau projet ▪ Créer un nouveau profil ▪ Spécifier les règles de transformation PIM-PIM, PIM-PSM et PSM-Code
Résumé	Créer un nouveau projet organisé selon la démarche MDA contenant un package pour les Modèles_PIM_Source, un package pour les modèles_PIM_Cible, un package pour les Modèles_PSM_Cible, un package pour les profils UML développés, un package pour les transformations M2M et un package pour la génération de code M2T.
Résultat	Projet MDA
Description	▪ Création du projet et ces sous packages ▪ Création des nouveaux profils ▪ Spécification des règles de transformation d'un diagramme des cas d'utilisation PIM vers un diagramme de Classe PIM ▪ Spécification des règles de transformation d'un diagramme des cas d'utilisation PIM vers un diagramme d'Activité PIM ▪ Spécification des règles de transformation d'un diagramme d'activité PIM vers un diagramme de séquence PSM ▪ Spécification des règles de génération d'un diagramme de sequence PSM vers code technique

Table 05 : Description du cas d'utilisation « Créer un nouveau projet MDA »

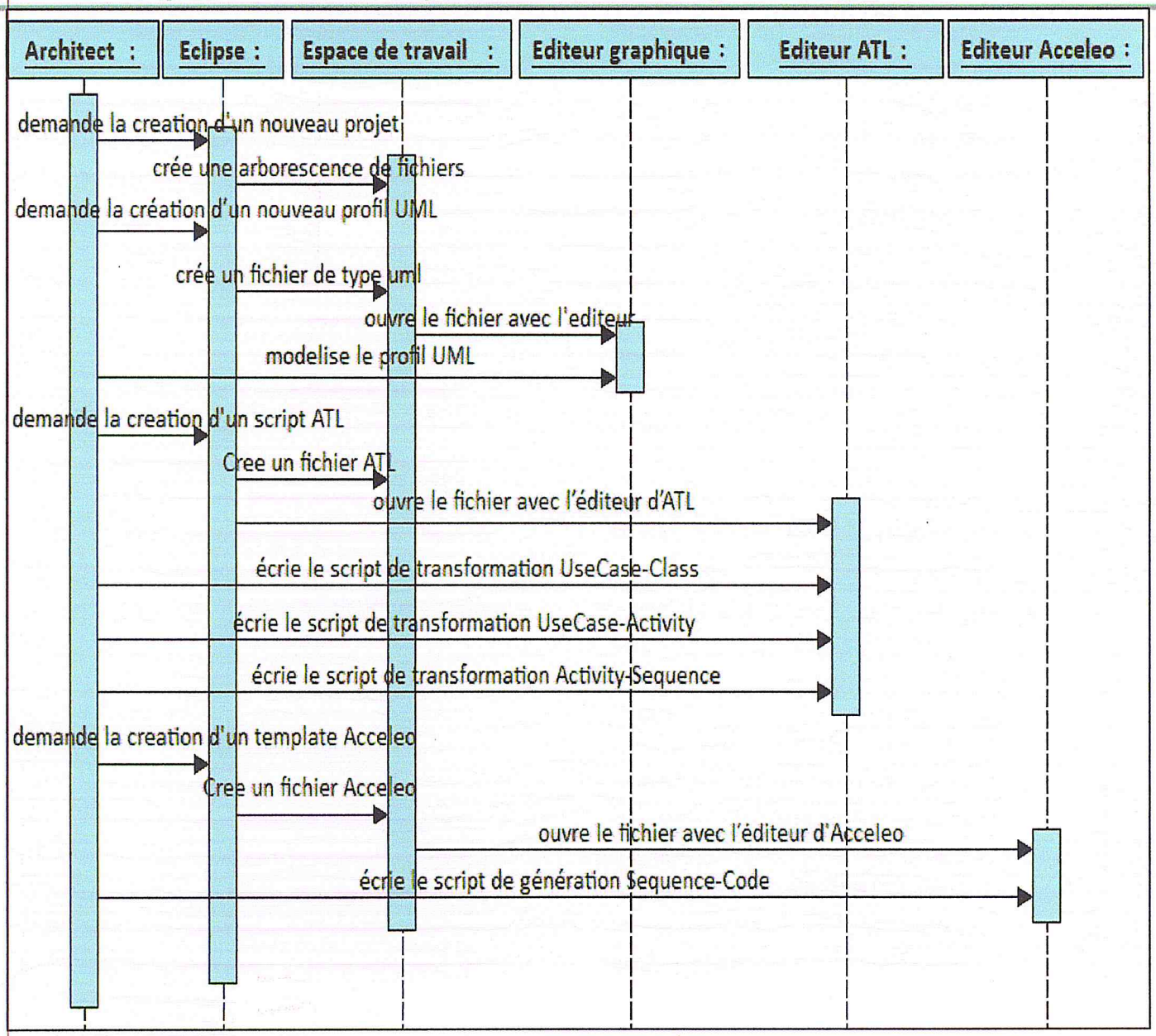
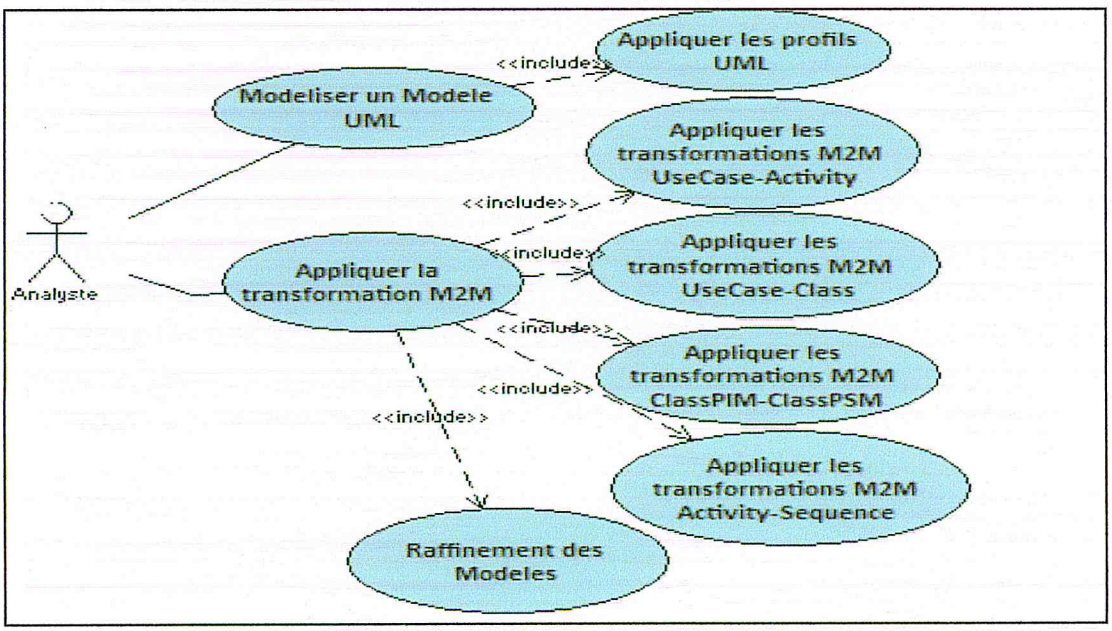


Figure 29 : diagramme de séquence des différentes fonctionnalités de l'Architect

3.2. Description des fonctionnalités de l'Analyste



Acteur principal	Analyste
Fonctionnalités	▪ Modéliser le modèle UML ▪ Application des profils UML ▪ Application des transformations PIM-PIM et PIM-PSM ▪ Raffinement des modèles résultants
Résumé	Modéliser le modèle UML diagramme des cas d'utilisation spécifié à un domaine particulier, appliquer sur ce dernier les profils UML et les deux transformations UseCase-Class et UseCase-Acivity, raffiner le diagramme de classe et appliquer sur ce dernier la transformation ClassPIM-ClassPSM **qui correspond au travail existant au sein de l'entreprise**, raffiner le modèle d'activité ,appliquer sur ce dernier le profil UML et la transformation Activitty-Sequence, raffiner le diagramme de sequence et appliquer sur ce dernier le profil Sequence-Code
Résultat	Le modèle de sequence PSM spécifique à la plateforme
Description	▪ Modéliser le diagramme UML des cas d'utlisation ▪ Appliquer le profil UML UseCase-Class sur le diagramme des cas d'utilisation ▪ Appliquer la transformation UseCase-Class ▪ Raffiner le diagramme de classe résultant ▪ Appliquer la transformation ClassPIM-ClassPSM ▪ Appliquer le profil UseCase-Activity sur le diagramme des cas d'utilisation ▪ Raffiner le diagramme d'activité PIM ▪ Appliquer le profil Activity-Sequence sur le diagramme d'activité raffiné ▪ Appliquer la transformation Activity-Sequence ▪ Raffiner le diagramme de sequence résultant ▪ Appliquer le profil Sequence-Code sur le diagramme de sequence raffiné

Table 06 : Description des différentes fonctionnalités de l'Analyste

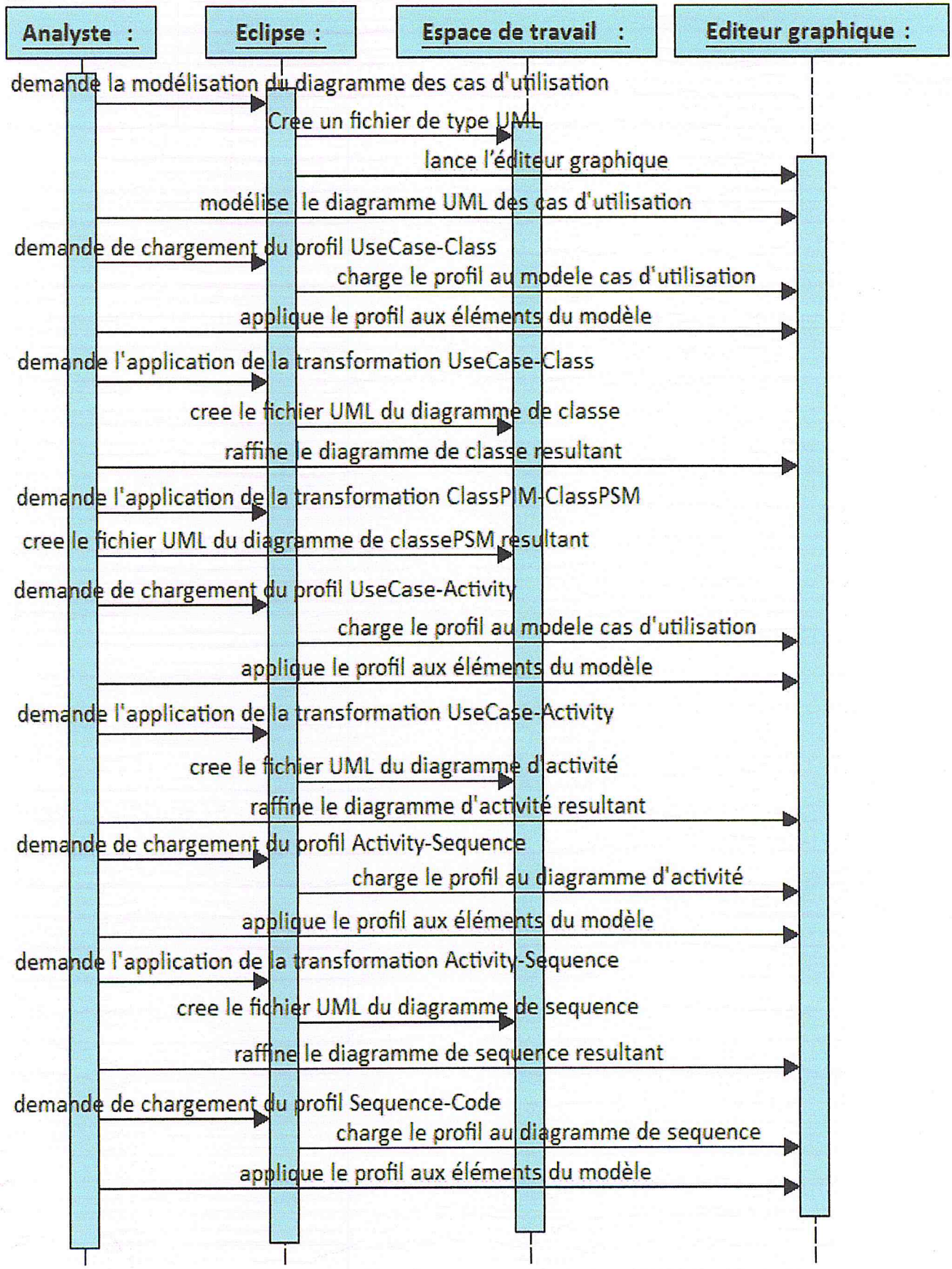
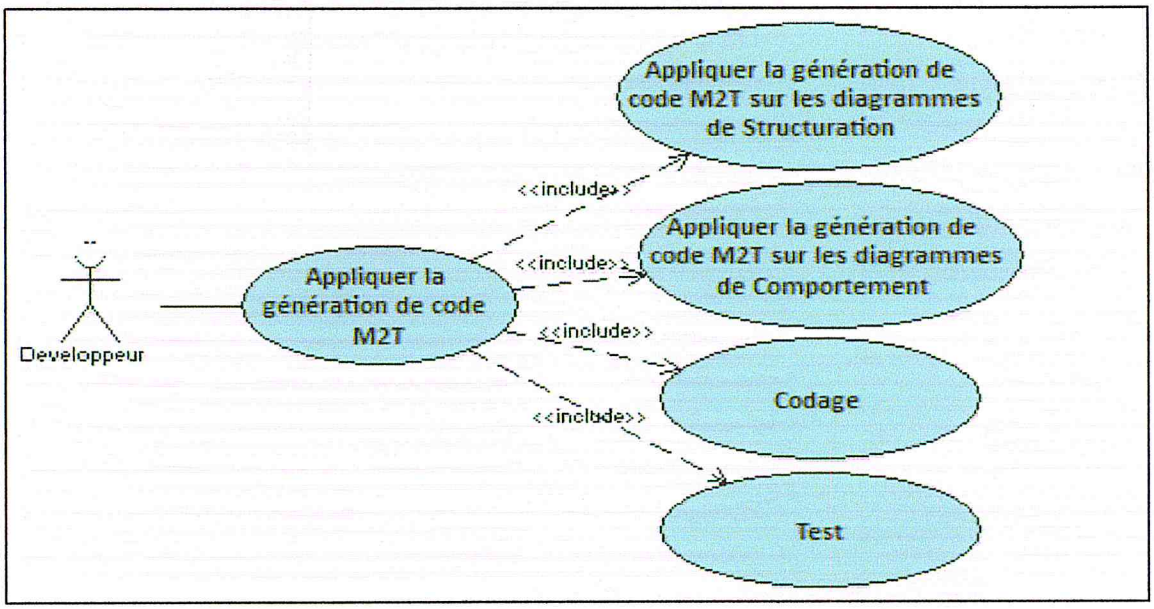


Figure 30: diagramme de séquence des différentes fonctionnalités de l'Analyste

3.3. Description des fonctionnalités du développeur



Acteur principal	Développeur
Fonctionnalités	▪ Appliquer la génération de code sur les diagrammes de structuration ▪ Appliquer la génération de code sur les diagrammes de comportement ▪ Codage ▪ Test
Résumé	Appliquer les règles de génération de code sur le diagramme de classe PSM qui **correspond au travail existant au sein de l'entreprise** et appliquer les regles de generation de code sur le diagramme de sequence PSM
Résultat	Code source JAVA de l'application web selon les conventions JEE
Description	Exécuter la chain de génération de code sur le diagramme de classe PSM pour générer du code java des différentes entités, méthodes de base CRUD pour chaque entités et les pages JSF correspondantes à ces méthodes, l'exécution de la chain de génération de code sur le diagramme de sequence PSM pour générer du code java des différents service métier , compléter le code généré et effectuer des tests

Table 07: Description des différentes fonctionnalités du développeur

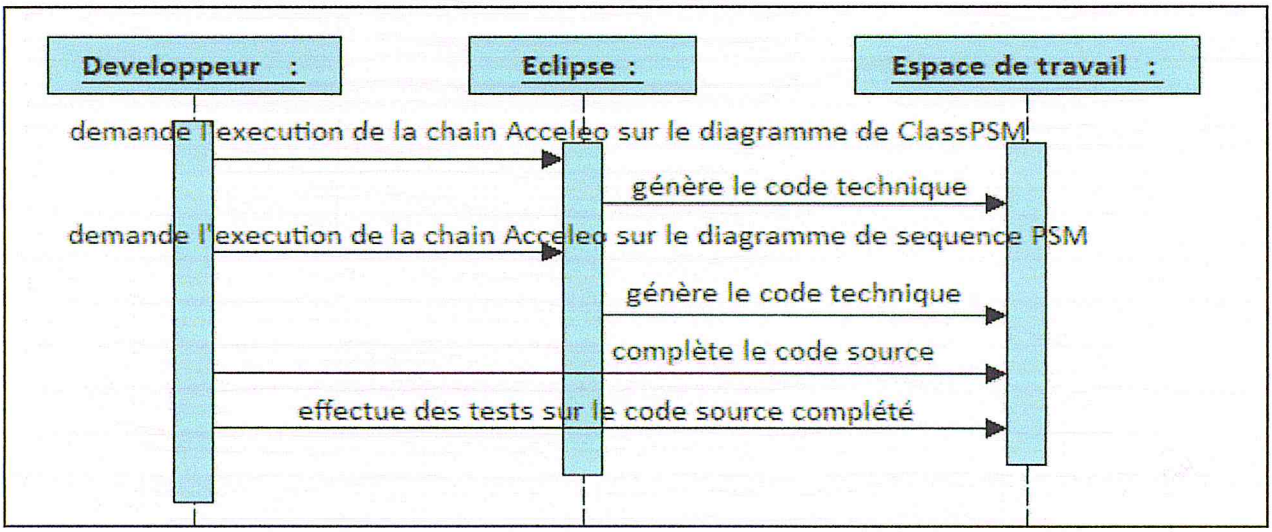


Figure 31 : diagramme de séquence des différentes fonctionnalités du développeur

4. Diagramme d'Activité

Les étapes de la démarche MDA que nous allons utiliser sont représentées par le diagramme d'Activités suivant:

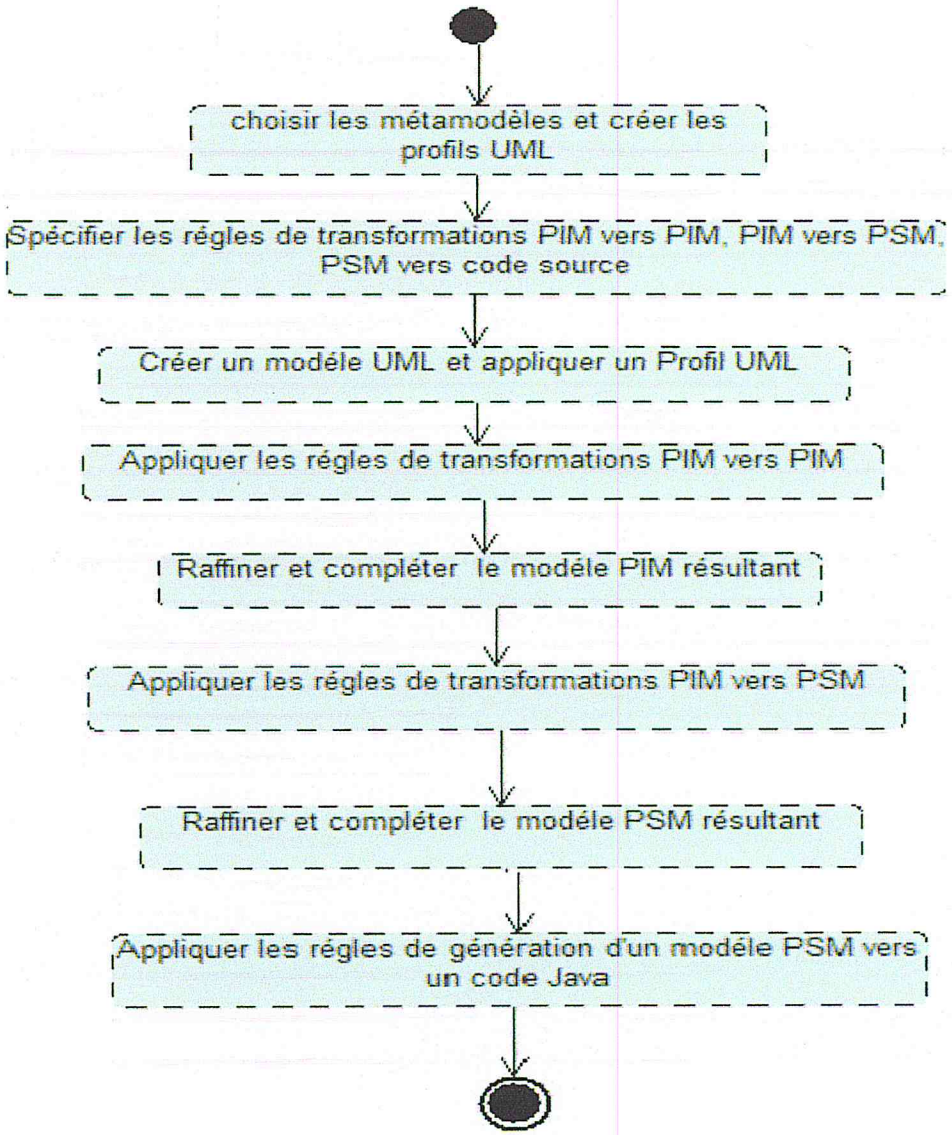


Figure 32: Diagramme d'activité de l'atelier génie logiciel MDA

5. Conclusion

Dans ce chapitre nous avons spécifié les fonctionnalités que doit assurer l'atelier de génie logiciel qui se base sur l'architecture MDA.

Nous allons présenter en détail dans le chapitre suivant les règles de transformations de modèles UML et les scripts nécessaires pour générer le code source d'une application web Java EE.

Partie III

Chapitre 02 : Implémentation de l'AGL

1. Introduction

Dans ce chapitre nous représentons l'implémentation des scripts de transformation d'un modèle UML vers un modèle UML (M2M) de type PIM vers PIM et PIM vers PSM, ainsi les scripts de génération de code source d'une application web selon les conventions de la plateforme JEE à partir d'un modèle UML PSM (M2T), en reposant sur la démarche MDA.

2. Outils d'implémentation

Pour l'outil de développement nous avons opté pour Eclipse et Topcased.

- **Eclipse IDE** est un environnement de développement intégré libre, permet de mettre en œuvre n'importe quel langage de programmation. [40] *(plus de détail sur Eclipse voir l'annexe)*

L'avantage majeur de cet IDE vient du fait de son architecture développée autour de plug-in, et c'est notre cas dans cet atelier génie logiciel, on a intégré des plugins pour qu'il soit compatible avec le processus de transformation des modèles et de génération de code source de MDA.

- **Topcased** est un projet atelier logiciel open source intégré à l'IDE Eclipse. [41]

Il est utilisé dans le développement de systèmes d'informations, ou la réalisation des diagrammes UML donc il convient bien pour notre AGL comme un modeleur. *(Plus de détail sur Topcased voir l'Annexe)*

Pour que notre AGL soit complet et supporte le processus MDA il reste à lui intégrer les outils de transformation des modèles et génération de code source **ATL** et **Acceleo** et ça grâce au mécanisme des plug-ins d'Eclipse qui permet l'extensibilité de la plateforme de développement et l'ajout des fonctionnalités qui ne sont pas fournies avec le standard.

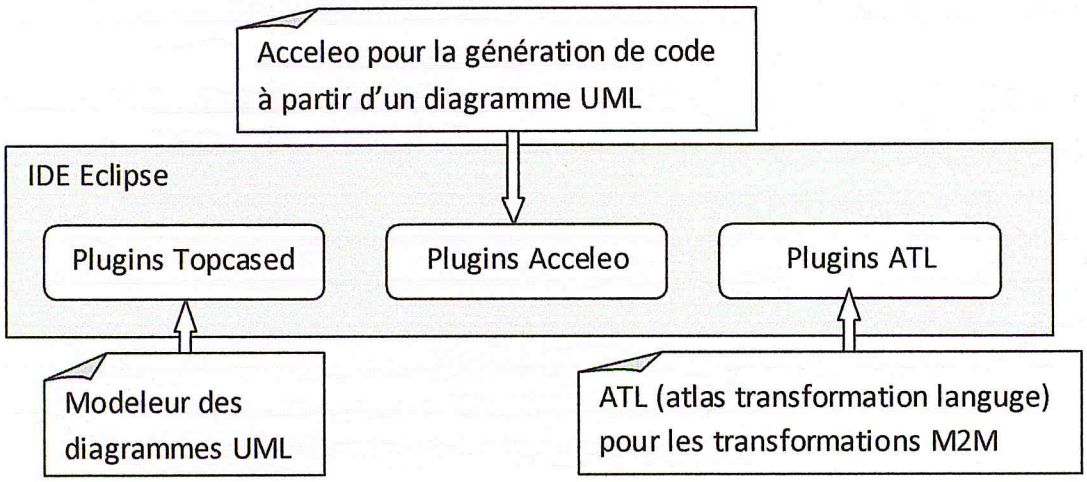


Figure 33: IDE Eclipse avec les plugins ajouter pour AGL développé

3. Les Scripts de Transformation Modèle vers Modèle (M2M)

3.1. Transformation des modèles PIM vers modèle PIM

La première transformation à faire de PIM vers PIM est de type M2M (modèle vers modèle), il y a deux transformations de ce type :

3.1.1. Transformation d'un diagramme cas d'utilisation vers un diagramme de classe

La première transformation à réaliser de type M2M (model to model) par le script « *UseCase2Classe.atl* » qui doit avoir le modèle PIM source en *UML2* (diagramme de cas d'utilisation) pour générer le modèle PIM cible en *UML2* (diagramme de classe), en appliquant le profil UML « *Profil_UseCase_Classe* » **[Partie 02_Chapitre02_ Figure23]** sur le modèle source.

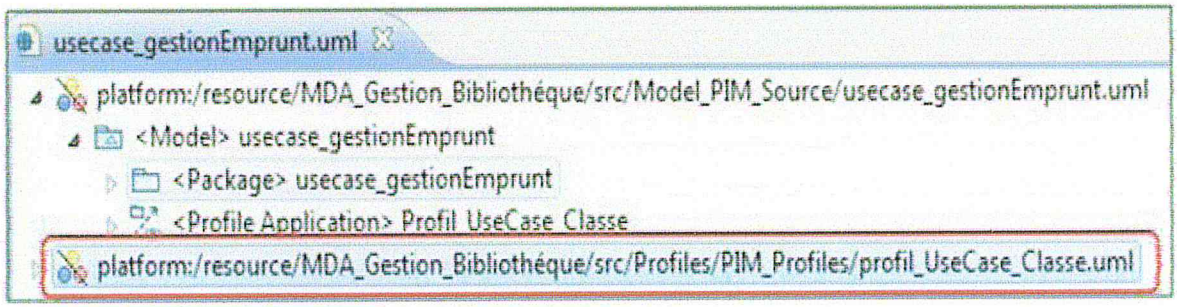


Figure 34: le profile UML « *Profil_UseCase_Classe* » *est appliqué sur le modèle use case source*

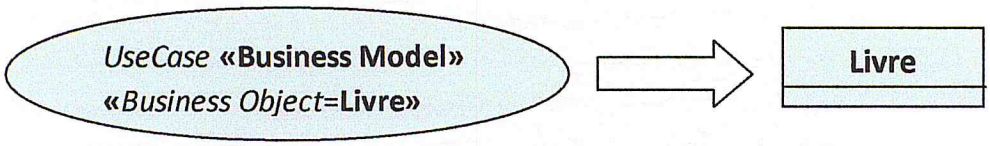
Dans la figure ci-dessous Le premier champ définie l'URI d'espace de noms de méta-modèle c'est un lieu abstrait de méta-modèle aide l'éditeur ATL a trouvé les mots clés des éléments UML des contraintes OCL pour faciliter l'Edition des transformations.

UseCase2Classe_PIM.atl

```
-- @nsURI UML2=http://www.eclipse.org/uml2/2.0.0/UML
module UseCase2Classe;
create OUT:UML2 from IN:UML2, Profil_UseCase_Classe:UML2;
```

Figure 35: Entête de script de transformation «UseCase2Classe.atl»

Un use case du diagramme cas d'utilisation source de la règle de transformation « *UseCase2Classe* » est transformé en une classe du diagramme de classe cible de cette transformation **[Partie 02_Chapitre02_ Tableau01]**, la classe résultat porte le nom sauvegardé dans la propriété «**Business Object**» du stéréotype « ***Business Model*** » du profile UML *« Profil_UseCase_Classe».*




```
rule UseCase2Class {
    from
        s : UML2!"uml::UseCase"  (thisModule.inElements->includes(s)
       and s.isStereotypeApplied(UML2!Stereotype.allInstances()->select(st | st.name='Business Model').first()))
    to  act: UML2!"uml::Class"(
    name <- s.getValue(UML2!Stereotype.allInstances()->select(st | st.name='Business Model').first(), 'Business Object'),
    visibility <- s.visibility,
    package <- thisModule.package
        )
}
```

Figure 36: la règle de transformation ATL « useCase2Class » d'un use Case vers une classe

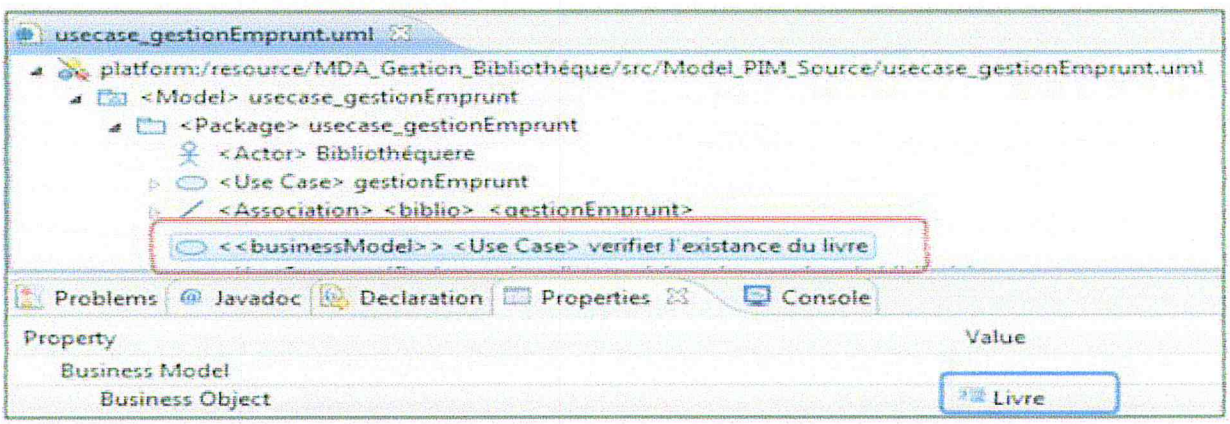


Figure 37 : Exemple d'un use case stéréotypé par «*Business Model*» pour le transformer par la règle « UseCase2Class » en une classe nommée par «*Business Object*»

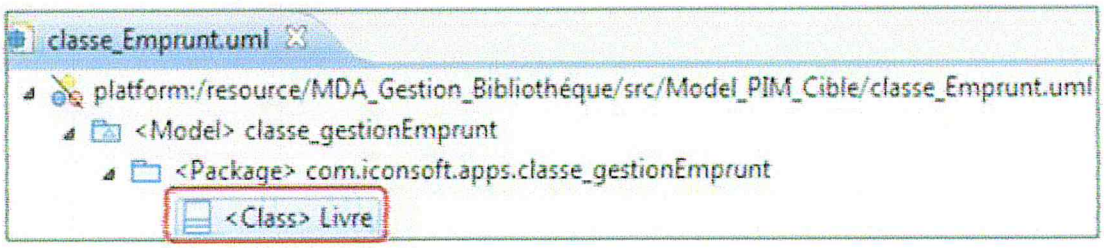


Figure38: la classe résultat d'exécution de la règle de transformation «useCase2Class»

3.1.2. Transformation d'un diagramme cas d'utilisation vers un diagramme d'activité

La deuxième transformation à réaliser de type M2M (model to model) par le script « *UseCase2Activity.atl* » qui doit avoir le modèle PIM source en *UML2* (diagramme de cas d'utilisation) pour générer le modèle PIM cible en *UML2* (diagramme d'activité), en appliquant le profil UML « *Profil_UseCase_Activity* » **[Partie 2_Chapitre02_ Figure23]** sur le modèle source.

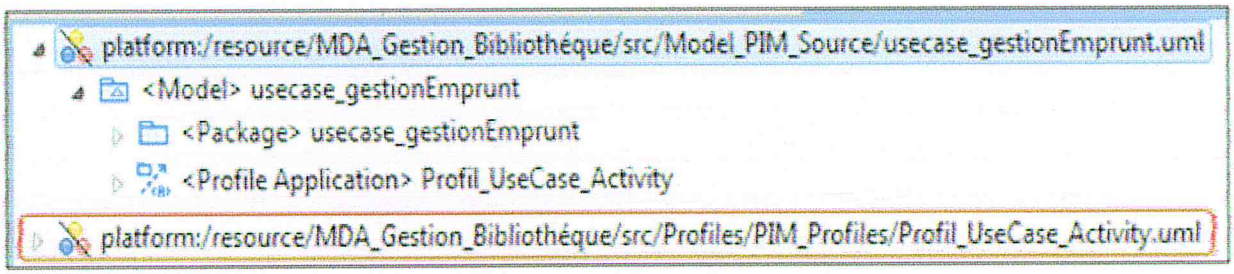
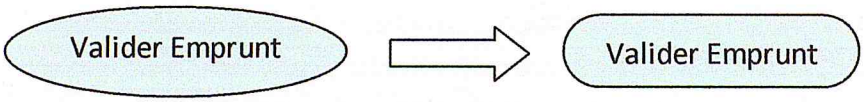


Figure 39: le profile UML *«Profil_UseCase_Activity»* est appliqué sur le modèle use case source

Figure 40: Entête de script de transformation «*UseCase2Activity.atl*»

Pour cette transformation nous montrons l'implémentation de quelques éléments du diagramme cas d'utilisation source vers les éléments correspondants dans le diagramme d'activité cible. **[Partie 02_chapitre02_ tableau 02]**

1. Une transformation est faite à partir d'un use case vers une activité nommée <StructuredActivityNode>.




```
rule activite {
    from
        s : UML2!"uml::UseCase"
    to
        act: UML2!"uml::StructuredActivityNode"(
          name <- s.name,
          visibility <- s.visibility,
          activity <- thisModule.Activity
        )
}
```

Figure 41: la règle de transformation «*activite*» d'un use case vers une activité en ATL

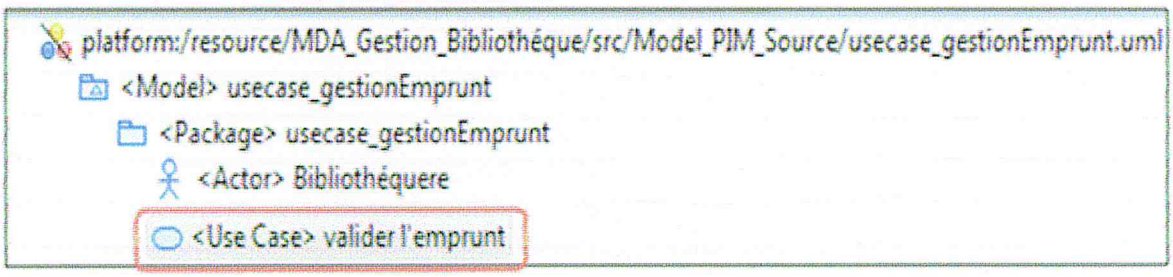


Figure 42: Exemple d'un use case qui sera transformer par la règle «*activite*» en une activité <Structured Activity Node>

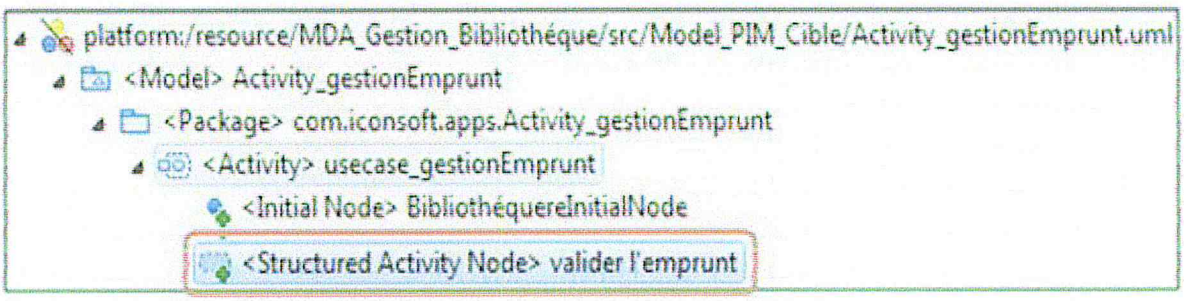
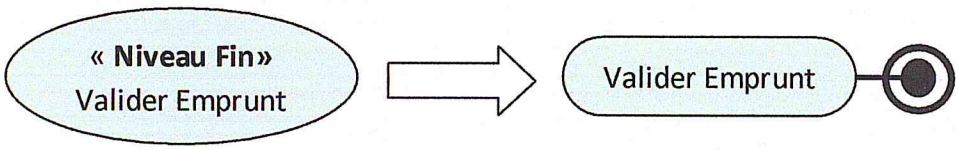


Figure 43: l'*activité* <Structured Activity Node> résultat d'exécution de la règle de transformation «*activite*»

2. Une transformation est faite à partir d'un use case stéréotypé par «**Niveau Fin**» vers une activité <Structured Activity Node> et le nœud final du diagramme d'activité, qui est nommé <Activity Final Node>




```
rule FinalNode {
    from
        s : UML2!"uml::UseCase" (thisModule.inElements->includes(s)
             and s.isStereotypeApplied(UML2!Stereotype.allInstances()->select(st | st.name='niveaufin').first()))
    to
        t : UML2!"uml::ActivityFinalNode"(
            name <- s.name+'FinalNode',
            visibility <- s.visibility,

            activity <- thisModule.Activity
        ),
            act: UML2!"uml::StructuredActivityNode"(
            name <- s.name,
```

Figure 44: la règle de transformation «*FinalNode*» d'un use case stéréotypé par «**NiveauFin**» vers une activité et le nœud final du diagramme d'activité cible

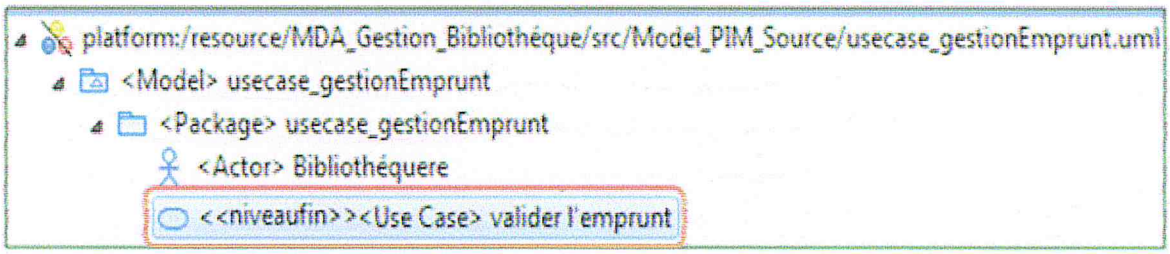


Figure 45: le use case stéréotypé par « *Niveau Fin* » pour le transformer par la règle «*FinalNode*» en une activité et le nœud final du diagramme d'activité cible

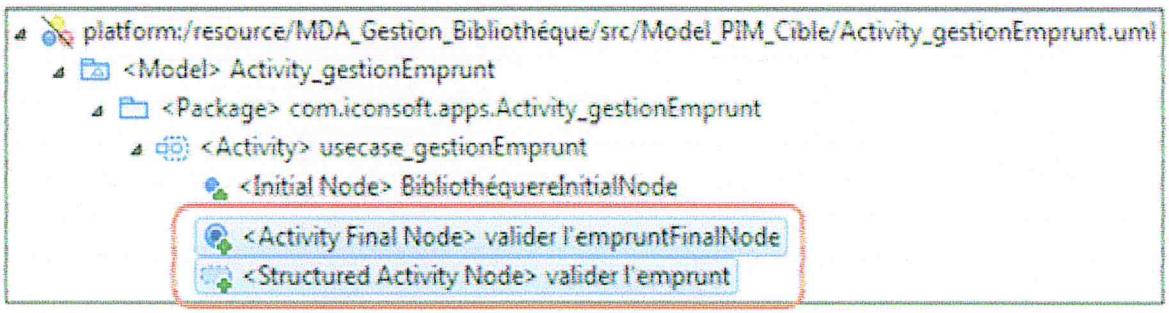


Figure 46: l'activité et le nœud Final résultat d'exécution de la règle de transformation « *FinalNode*»

3.2. Transformation des modèle PIM vers modèle PSM

La seconde transformation à réaliser est de type M2M (model to model) d'un modèle PIM source UML (diagramme d'activité) vers un modèle PSM selon la plateforme JEE (diagramme de séquence) par le script «*Activity2Sequence.atl* ».

Cette transformation prend en entrée le diagramme d'activité raffiné plus le profil «*Profil_Acivity_Sequence*» **[partie 02_chapitre02_ figure25].**

Cette transformation à pour but de générer un diagramme de séquence qui va être utilisé comme source dans l'étape suivante de génération de code source.

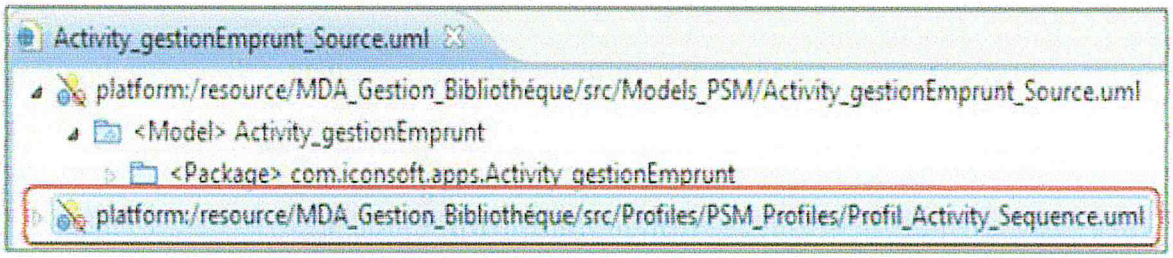


Figure 47: le profile UML *« Profil _Activity_Sequence »* est appliqué sur le modèle activité source

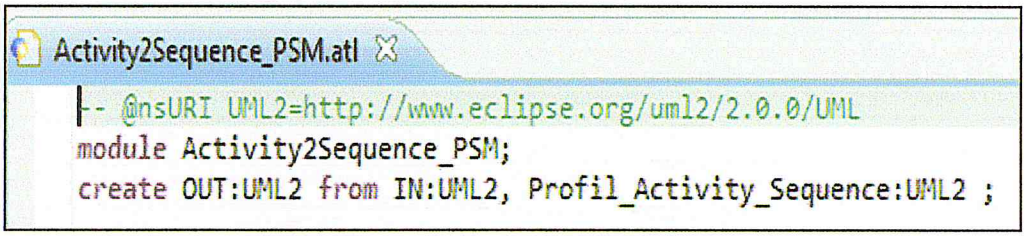
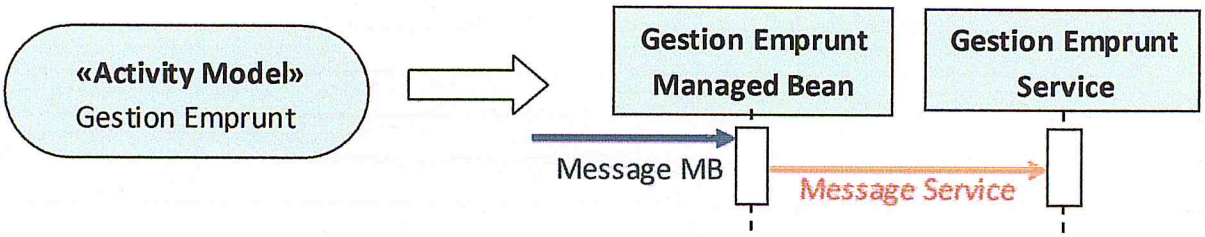


Figure 48: Entête de script de transformation «Activity2Sequence.atl»

Pour cette transformation nous montrons l'implémentation de quelques éléments du diagramme d'activité source vers les éléments correspondants dans le diagramme de séquence cible. **[Partie 02_chapitre02_ tableau 03].**

1. Cette transformation a comme élément source une activité stéréotypée par **«Activity Model»** et elle génère deux lignes de vie une représente le **Service** à créer et l'autre représente le **ManagedBean** qui gère la page web (.xhtml) du service, chacune avec son activation «BehaviorExecutionSpecification» du digramme de séquence.

Et pour chaque ligne de vie son message d'appel, et chacun avec son «*Message Occurence Specification*».



- Exemple de la ligne de vie « **Service** » de la règle de transformation « *ActivityModel2LifeLine* » qui gère cette ensemble de transformation (même traitements pour **ManagedBean**) :

```
ActivityModel2LifeLine {
from
    s : UML2!"uml::StructuredActivityNode" (thisModule.inElements->includes(s)
            and  s.isStereotypeApplied(UML2!Stereotype.allInstances()->select(st | st.name='ActivityModel').first()))
to
    lfS : UML2!"uml::Lifeline" (
        name <- s.name+'Service',
        visibility <- s.visibility,
        interaction <- thisModule.Interaction
    ),
```

Figure49: la règle de transformation «*ActivityModel2LifeLine*» d'une activité stéréotypée par «***Activity Model***» vers une ligne de vie «*Service*»

```
beh2:UML2!"uml::BehaviorExecutionSpecification"(
name <-  lfS.name+'Behavior',
visibility <- s.visibility,
covered <- lfS,
enclosingInteraction <- thisModule.Interaction
),
```

Figure 50: la règle de transformation : d'une activité stéréotypée par «***Activity Model***» vers l'activation «*BehaviorExecutionSpecification*» de la ligne de vie «*Service*»

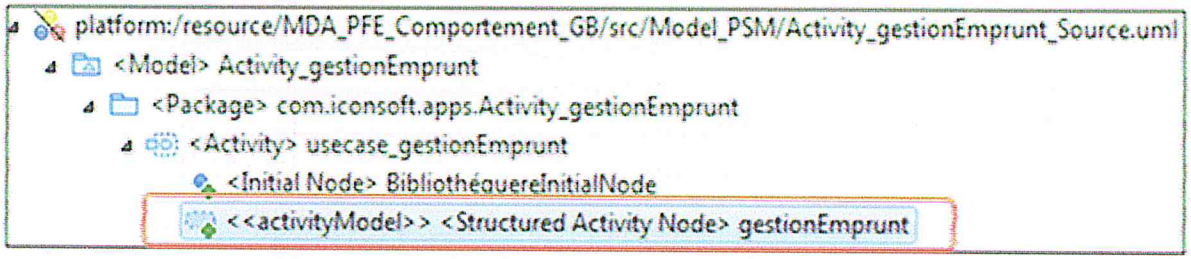


Figure 51 : l'activité stéréotypée «ActivityModel» pour la transformer en une ligne de vie et son *BehaviorExecutionSpecification* du diagramme de séquence cible

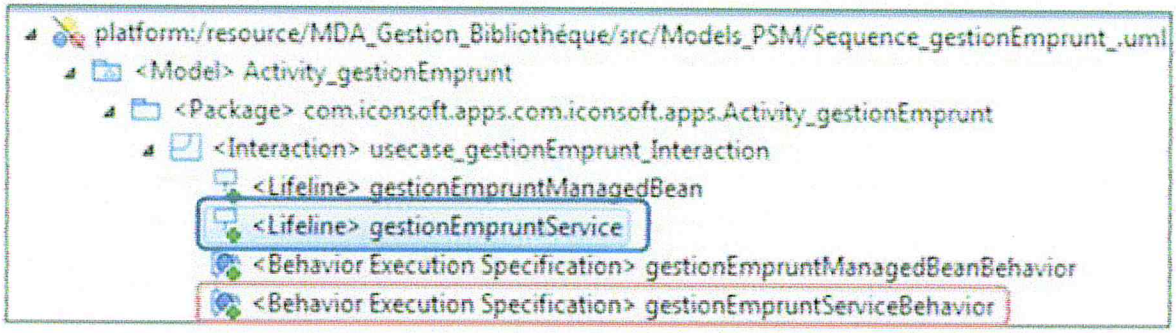


Figure 52: résultat d'exécution : LifeLine avec son *BehaviorExecutionSpecification*

- **le message d'appel de la ligne de vie**

```
--Message entre de la ligne de vie ManagedBean et service
msgS:UML2!"uml::Message"(
    name <- s.name+'Service',
    visibility <- s.visibility,
    sendEvent <- mocS1,
    receiveEvent <- mocS2,
    interaction <- thisModule.Interaction
),
```

Figure 53: la règle de transformation : d'une activité stéréotypée par «***Activity Model***» vers le message d'appel de la ligne de vie «*Service*»

```
mocS1:UML2!"uml::MessageOccurrenceSpecification"(
        name <- s.name+'Service_MocSend',   --on considère
        visibility <- s.visibility,
        message <-msgS ,
        covered <-lfMB,
        enclosingInteraction <- thisModule.Interaction
        ),
mocS2:UML2!"uml::MessageOccurrenceSpecification"(
        name <- s.name+'Service_MocReceive',   --on consid
        visibility <- s.visibility,
        message <-msgS ,
        covered <-lfS,
        enclosingInteraction <- thisModule.Interaction
        )
```

Figure 54 : la règle de transformation : d'une activité stéréotypée par «***Activity Model***» vers les «*Message Occurrence Specification*» de message d'appel de la ligne de vie «*Service*»

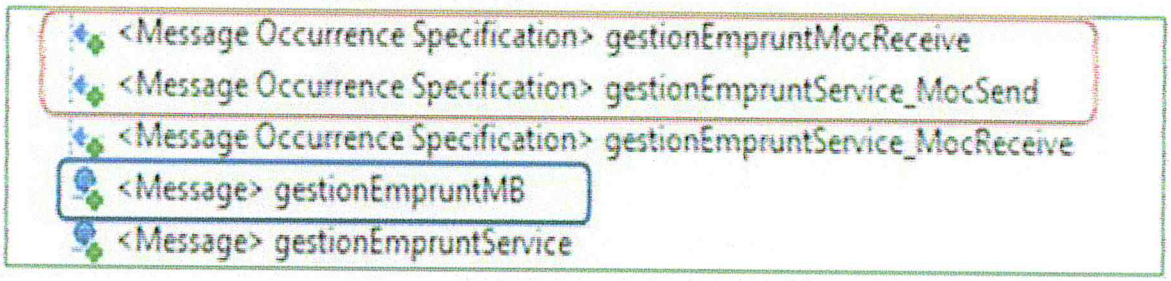
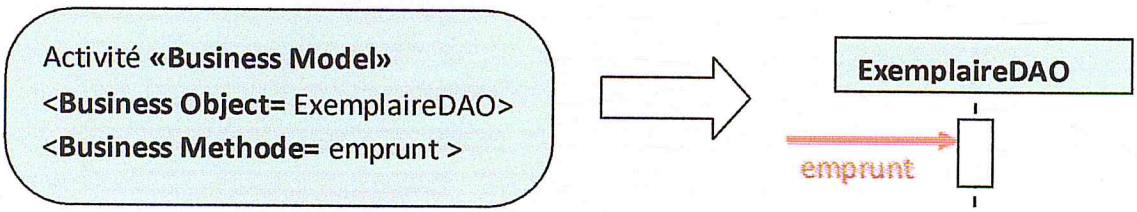


Figure 55 : résultat d'exécution : Messages «service» avec ses *Message Occurrence Specification*

2. Chaque activité stéréotypée par **Business Model** correspond à une ligne de vie avec son activation «BehaviorExecutionSpecification», qui porte le nom sauvegardé dans la propriété **Business Object** du stéréotype.

Et son message d'appel qui porte la signature sauvegardée dans la propriété **Business Méthode** du stéréotype, avec ses « *Message Occurence Specification* » d'appel et de réponse.



- Exemple de la ligne de vie « **ExemplaireDAO** » générée à partir de la règle de transformation « *BusinessModel2LifeLine* ».

```
rule BusinessModel2LifeLine {
    from
        s : UML2!"uml::StructuredActivityNode" (thisModule.inElements->includes(s)
                    | and  s.isStereotypeApplied(UML2!Stereotype.allInstances()->select(st | st.name='BusinessModel').first()))
    to
        lfM : UML2!"uml::Lifeline" (
            name <- s.getValue(UML2!Stereotype.allInstances()->select(st | st.name='BusinessModel').first(), 'Business Object'),
            visibility <- s.visibility,
            interaction <- thisModule.Interaction
```

Figure 56 : la règle de transformation «*BusinessModel2LifeLine*» d'une activité stéréotypée par «***Business Model***» vers une ligne de vie nommée par «***Business Object***»

```
beh:UML2!"uml::BehaviorExecutionSpecification"(
name <-  s.getValue(UML2!Stereotype.allInstances()->select(st | st.name='BusinessModel').first(), 'Business Object')+'Behavior',
visibility <- s.visibility,
covered <- lfM,
enclosingInteraction <- thisModule.Interaction
),
```

Figure 57 : la règle de transformation qui gère l'activation «*BehaviorExecutionSpecification*» de la ligne de vie nommée par «***Business Object***»

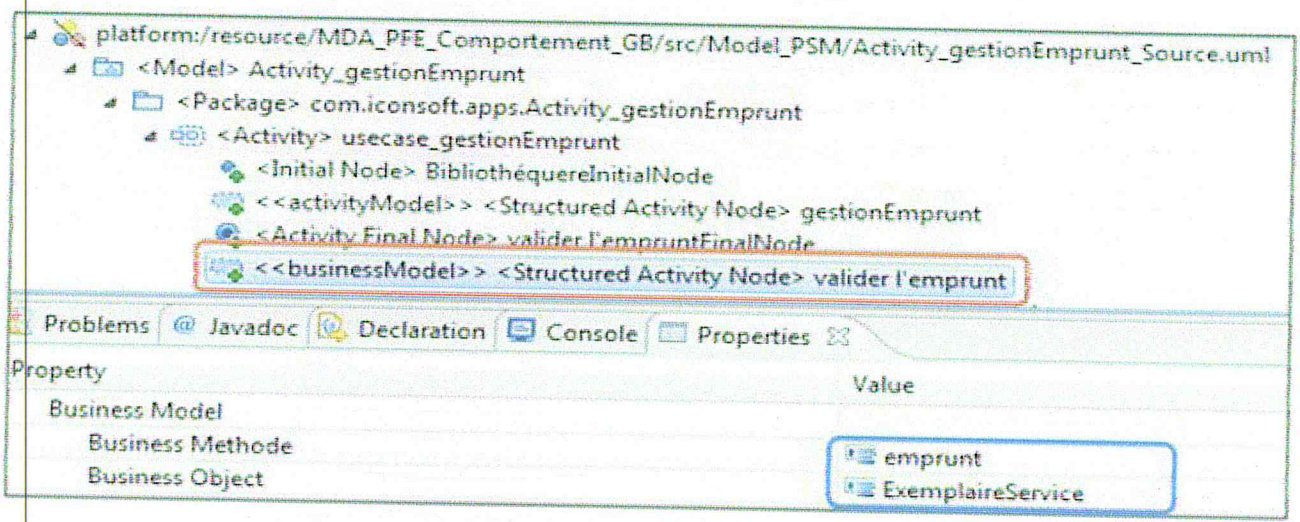


Figure 58: l'activité stéréotypée «***BusinessModel***» pour la transformer en une ligne de vie et son *BehaviorExecutionSpecification* du diagramme de séquence cible

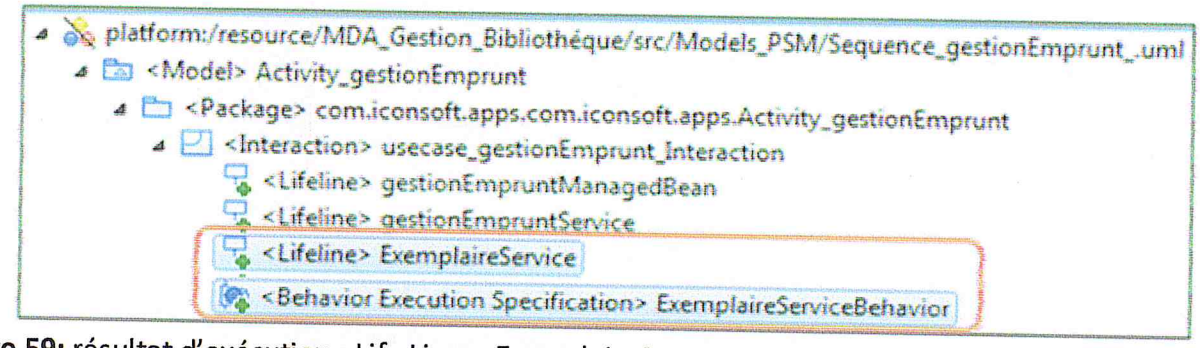


Figure 59: résultat d'exécution : Life Line «ExemplaireService» avec son *BehaviorExecutionSpecification*

- **le message d'appel de la ligne de vie**

```
msg:UML2!"uml::Message"(
    name <- s.getValue(UML2!Stereotype.allInstances()->select(st | st.name='BusinessModel').first(), 'Business Methode'),
    visibility <- s.visibility,
    receiveEvent <- moc2,
    sendEvent <- moc1,
    messageSort <- msg.messageSort.debug(),
    interaction <- thisModule.Interaction
)
```

Figure 60: règle de transformation d'un Message «*emprunt*» d'appel de la ligne de vie

```
moc1:UML2!"uml::MessageOccurrenceSpecification"(
            name <- s.getValue(UML2!Stereotype.allInstances()->select(st | st.name='BusinessModel').first(),
                                                                    'Business Methode')+'MocSend',
            visibility <- s.visibility,
            message <-msg ,
            covered <-thisModule.service,
            ),
moc2:UML2!"uml::MessageOccurrenceSpecification"(
        name <- s.getValue(UML2!Stereotype.allInstances()->select(st | st.name='BusinessModel').first(),
                                                                    'Business Methode')+'MocReceive',
        visibility <- s.visibility,
        message <- msg,
        covered <-lfM,
```

Figure 61: règle de transformation des *MessageOccurenceSpecification send* et *receive* un Message d'appel «emprunt»

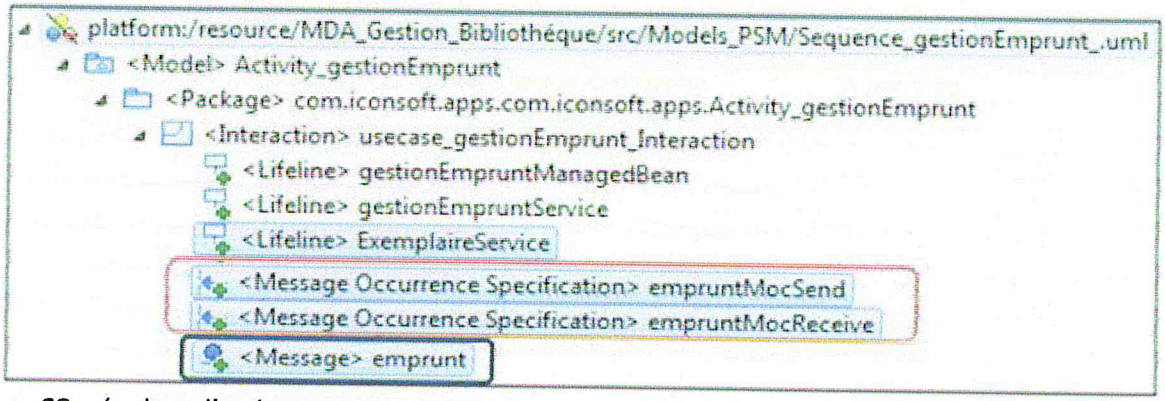


Figure 62: résultat d'exécution : Message «*emprunt*» avec ses *MessageOccurrenceSpecification*

4. Les scripts de génération de code source a partir du Modèle PSM

Aprés avoir réalisé les transformations de PIM vers PIM et de PIM vers PSM, il est temps de générer le code source à partir du modèle PSM (diagramme de séquence), une génération dans Acceleo est representée par un fichier de type **chain**, ce dernier peut avoir un ou plusieurs templates de génération représentant un ou plusieurs fichiers à générer.

Pour réaliser une génération avec Acceleo, on doit spécifier deux fichiers le premier c'est le template « .mt » qui décrit les règles de génération et le deuxième «. **chain** » qui est utilisé par le moteur d'Acceleo comme source d'information pour réaliser la génération. Ce fichier spécifie le méta-modèle, le modèle source, les fichiers templates à appliquer, le répertoire cible des fichiers générés et un fichier log pour le journal de génération.

Au cours de notre travail nous avons développé des services qui n'existent pas dans les bibliothèques intégrées dans Acceleo à fin de faciliter l'écriture des scripts de génération.

La figure ci-dessous illustre les services que nous avons utilisés :

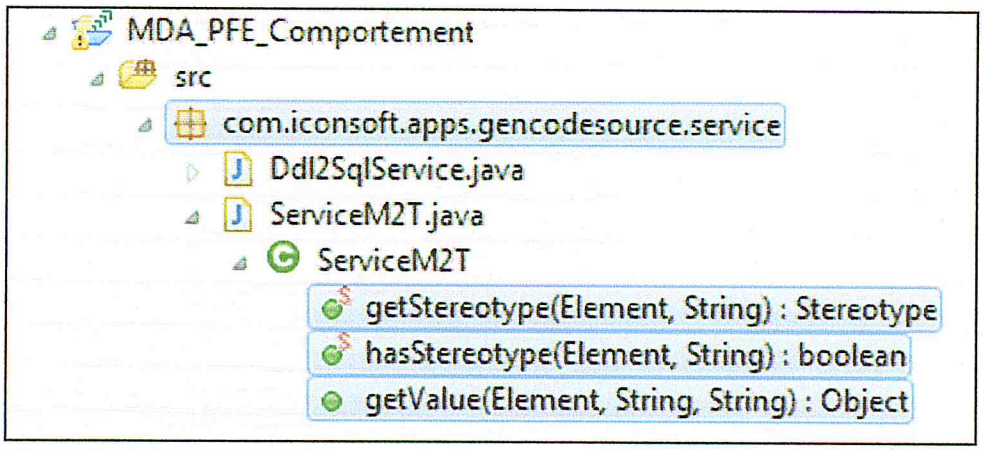


Figure 63 : les services développés pour la génération de code M2T

- Le service **getStereoype (Element, String)** : retourne les stéréotypes appliqués sur l'Elément en paramètre.
- Le service **hasStereotype (Element, String)** : vérifie si le stéréotype en paramètre (String) est appliqué sur l'élément Element.
- Le service **getValue (Element, String, String) :** retourne les valeurs (String) des propriétés du stéréotype (String) appliqué à l'élément (Element) en paramètre.

Cette génération a pour but de générer du code technique conforme aux conventions de la plateforme JEE à partir du diagramme UML de séquence PSM et après l'application du profil « *Profil_sequence_code* » sur le diagramme UML de séquence.

La figure ci-dessous illustre le diagramme de séquence après l'application du profil « *Profil_sequence_code* ».

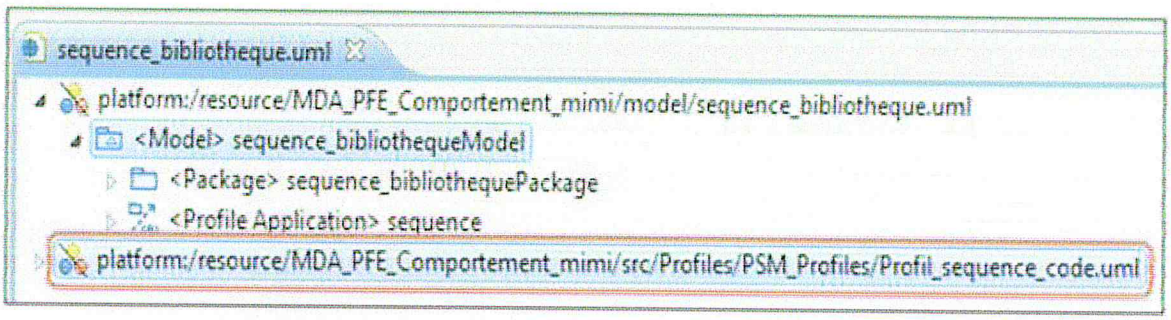


Figure 64 : le profil « *Profil_sequence_code* » appliqué à un diagramme de séquence

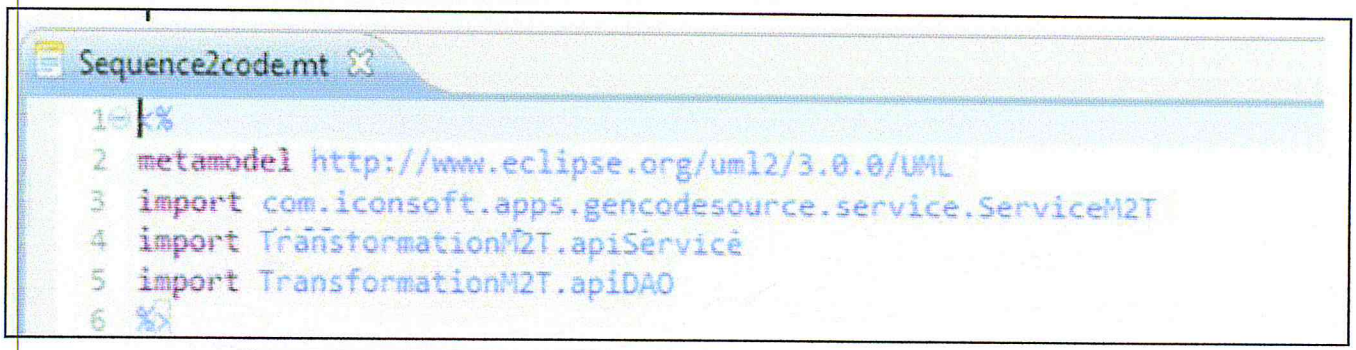
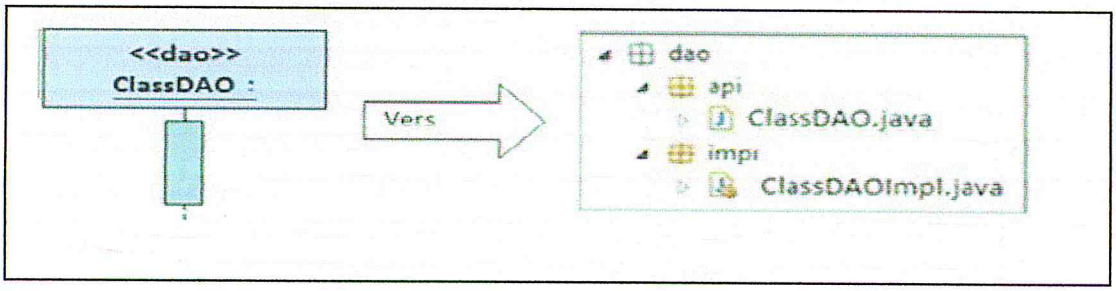
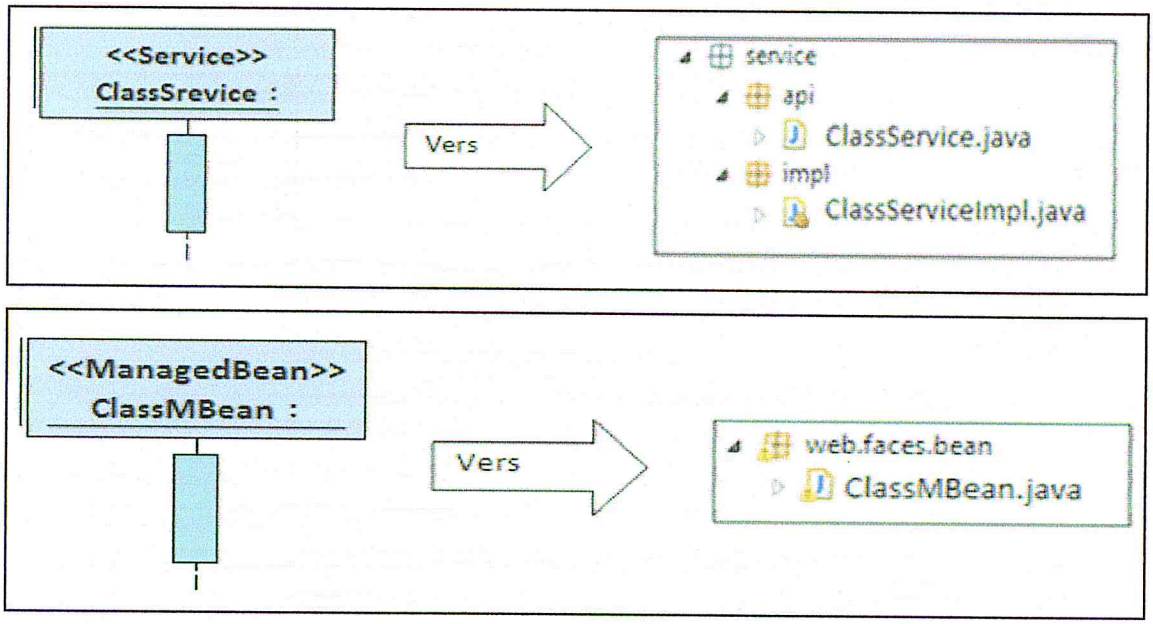


Figure 65 : entête du script de génération de code M2T avec Acceleo

- Dans la **[Partie02_chapitre2_Tableau 04]** nous avons illustré les différentes correspondances entre un diagramme de séquence et le code source associé, donc dans ce qui suit nous allons vous présenter les règles de génération de quelques correspondances.

  - Chaque ligne de vie correspond a un Package et une Classe java selon le stéréotype appliqué sur cette ligne de vie (DAO, ManagedBean, Service) :






```
<%script type="uml.Lifeline" name="fullFilePath" post="trim()"%>
<%nPut("lifeline")%>
<%for (getAppliedStereotypes()){%>
<%nPut("stereotype")%>
<%if (nGet("stereotype").name.equalsIgnoreCase("DAO")){%>
<%nGet("lifeline").getModel().name.replaceAll("\.","/" )%>/dao/Impl/<%nGet("lifeline").name.toU1Case()%>Impl.java
<%}%>
```

Figure 66: script de génération du package **dao** et ses sous packages **api** et **impl**

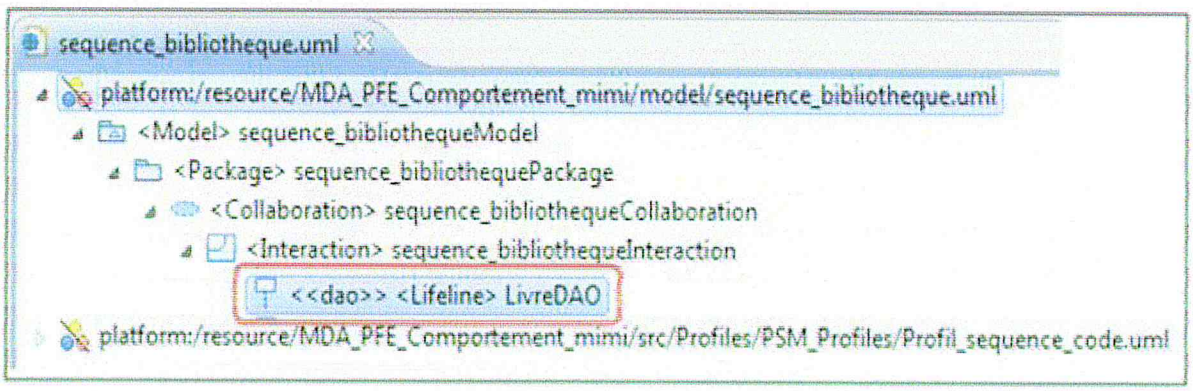


Figure 67: Exemple d'une ligne de vie source stéréotypée par «*DAO*»

LivreDAO.java

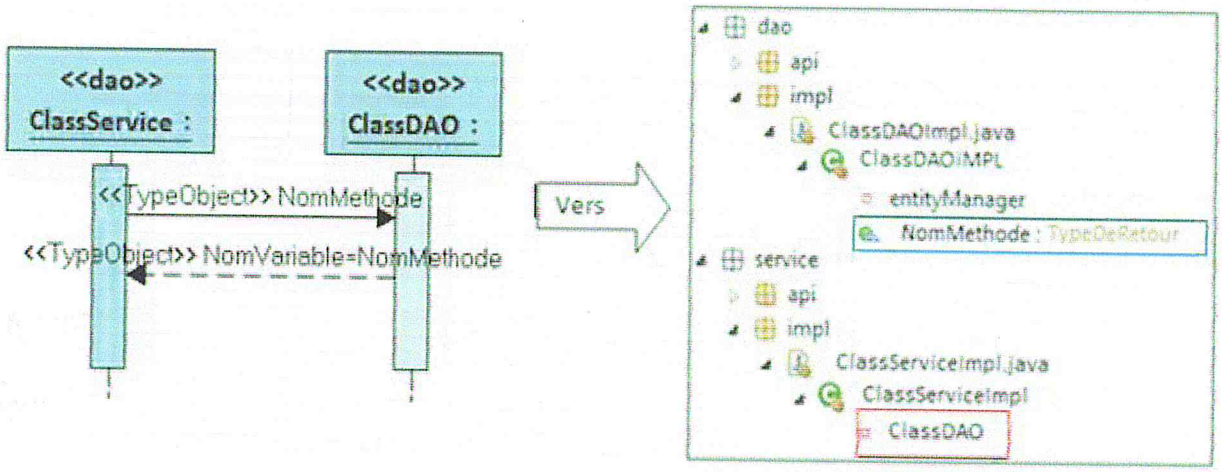
```
1  package sequence_bibliothequeModel.dao.api;
2
3  import javax.ejb.Local;
4
5  @Local
6  public interface LivreDAO {
7
8      // creer la Methode= findById
9      public Livre findById();
10
11     // creer la Methode= findById
12     public Livre findById();
13
14 }
15
```

Figure 68: code source de l'interface *LivreDAO* généré correspondant à ligne de vie « *LivreDAO* »

LivreDAOImpl.java

```
1  package sequence_bibliothequeModel.dao.Impl;
2
3  import javax.ejb.Stateless;
9  @Stateless
10 public class LivreDAOImpl implements LivreDAO {
11     @PersistenceContext
12     private EntityManager entityManager;
13
14     // creer la Methode= findById
15     public Livre findById() {
16
17         // return de la Methode= findById
18         return " ";
19     }
20
21 }
```

Figure 69: code source de la classe *LivreDAOImpl* qui implémente l'interface *LivreDAO*



- On a indiqué dans **[Partie02_chapitre 02_tableau 04]** qu'il est possible d'appliquer sur les messages le stéréotype **TypeObject** ou **PrimitiveType** pour récupérer le type de retour de la méthode (pour les messages de type 'Synchronous call') et le type de la variable à créer (pour les messages de type 'reply').

- Pour les messages de type « **Synchronous call** » : Chaque message de type « Synchronous call» représente :

- Une instanciation de la ligne de vie qui le reçoit (ClassDao dans l'exemple) dans la ligne de vie qui fait l'appel de la méthode (ClassService dans l'exemple).

- Une définition de méthode (la méthode NomMethode dans l'exemple) dans la ligne de vie qui reçoit l'appel de méthode (ClassDAO dans l'exemple)

- Pour les messages de type « **Reply** » :

- Nous sommes mis en accord sur une convention de nommage pour ce type de message : NomVariable=NomMethode

- Chaque message de type « Reply » représente une déclaration de la variable spécifié dans le message qui reçoit le retour de l'appel de méthode.

Sequence2code.mt

```
98  <%for (nGet("occurenceReceive").message){%>
99  <%nPut("msg")%>
100 <%if (nGet("msg").messageSort=="reply"){%>
101 //Creer la Variable= <%nGet("msg").name.split("=").nGet(0).toL1Case()%>
102 <%for (getAppliedStereotypes()){%>
103 <%nPut("str")%>
104 <%if (nGet("str").name.equalsIgnoreCase("TypeObject")){%>
105 <%nGet("msg").getValue(name,"type").toU1Case()%> <%nGet("msg").name.split("=").nGet(0).t
106 <%}else{%>
107 <%nGet("msg").getValue(name,"type").toU1Case()%> <%nGet("msg").name.split("=").nGet(0).t
108 <%nGet("msg").name.split("=").nGet(0).toL1Case()%>=<%for (owner.eAllContents("Lifeline")
109 <%}else{%>//creer la Methode= <%nGet("msg").name%>
110  <%visibility%> <%if (hasStereotype("TypeObject")|| hasStereotype("PrimitiveType")){%><%
```

Figure 70: script de génération d'une méthode ou déclaration d'une variable à partir d'un message du diagramme de séquence

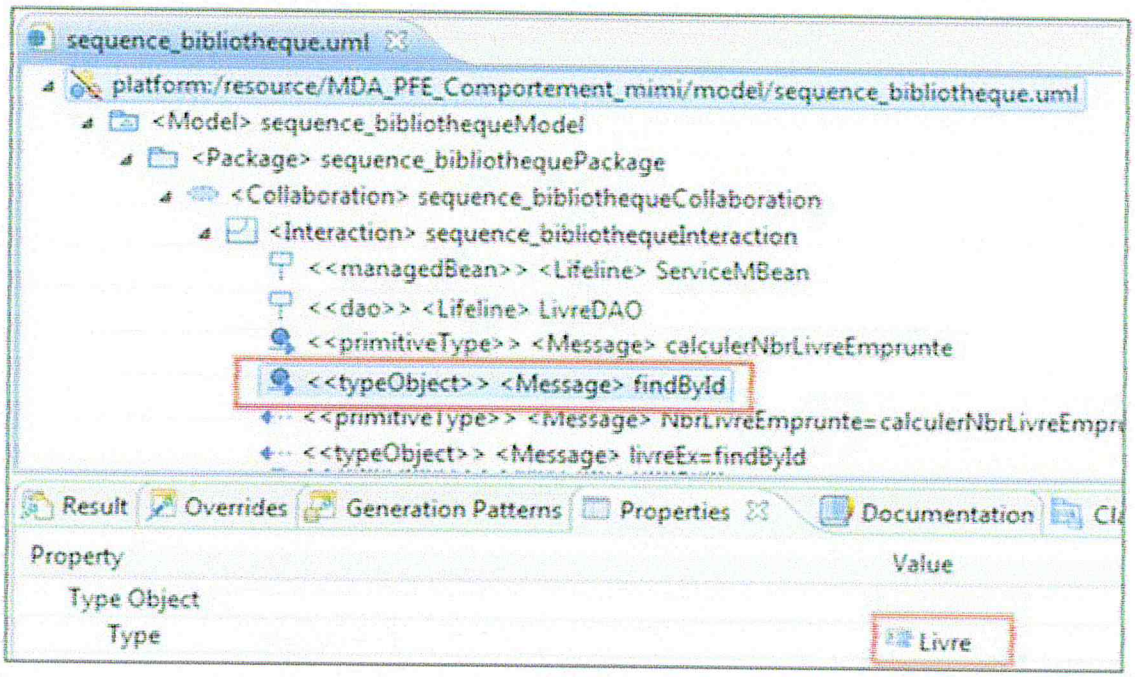


Figure 71 : exemple du message d'appel *findById* stéréotypé par « *typeObject* »

LivreDAOImpl.java

```
1  package sequence_bibliothequeModel.dao.Impl;
2
3  import javax.ejb.Stateless;
8
9  @Stateless
10 public class LivreDAOImpl implements LivreDAO {
11     @PersistenceContext
12     private EntityManager entityManager;
13
14     // creer la Methode= findById
15     public Livre findById() {
16
17         // return de la Methode= findById
18         return " ";
19     }
20
21 }
```

Figure 72: code source de la classe *LivreDAOImpl* qui implémente l'interface *LivreDAO* compenant la définition de la methode *find*

5. Conclusion

Nous avons vu dans ce chapitre la représentation et l'implémentation des modèles UML PIM et PSM puis les scripts nécessaires pour construire un prototype de transformation des modèles UML de comportement et la génération du code source d'une application web JEE en reposant sur la démarche MDA.

Partie III

Chapitre 03 :

Test de l'AGL

1. Introduction

L'objectif de ce chapitre est de faire un test sur les scripts de transformation des modèles UML (M2M) et aussi ceux de la génération du code source (M2T) afin d'avoir un code source java d'une application web généré automatiquement à partir des différentes transformations des modèles UML comportement.

Nous nous sommes intéressées au service de la **Gestion des emprunts** des livres dans une bibliothèque.

2. Présentation du projet MDA sous Eclipse

Le projet MDA est un projet eclipse construit des classes Services Acceleo et fichiers jar nécessaire pour faciliter la transformation Avec Acceleo, et un ensemble de Package et de fichiers pour organiser le travail (Figure 73):

- Models_PIM_Source: on doit trouver dans ce répertoire le fichier UML qui représente le modèle PIM source de l'application.
- Model_PIM_Cible: on doit trouver dans ce répertoire le fichier uml qui représente les modèle PIM résultat de la transformation M2M_PIM de l'application.
- Models_PSM: les modèles de résultat de transformation PIM vers PSM doivent se trouver dans ce répertoire.
- Profiles: c'est le Package ou se trouve deux sous packages représentant:
  - PIM_Profiles: tous les profiles qui représente le méta-modèle des modèles PIM.
  - PSM_Profiles: tous les profiles qui représente le méta-modèle des modèles PSM.
- Transformation_M2M: dans ce répertoire est enregistré tous les scriptes de transformation modèle vers modèle d'ATL.
- Transformation_M2T: dans ce répertoire est enregistré tous les scriptes de génération modèle vers texte, on trouve les Templates et les chaines de transformation Acceleo.

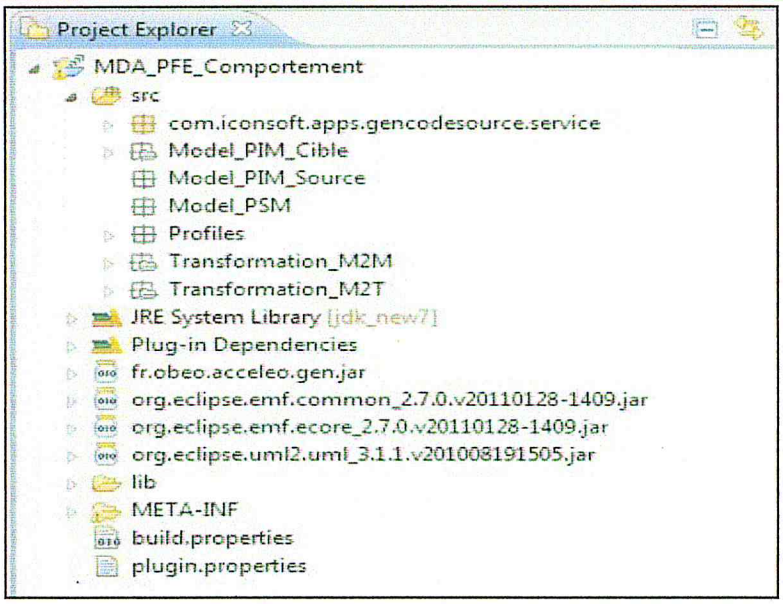


Figure 73 : Structure du projet MDA

Notre système d'étude est la gestion Bibliothèque, nous présentons par la suite le diagramme Use case globale de cette gestion.

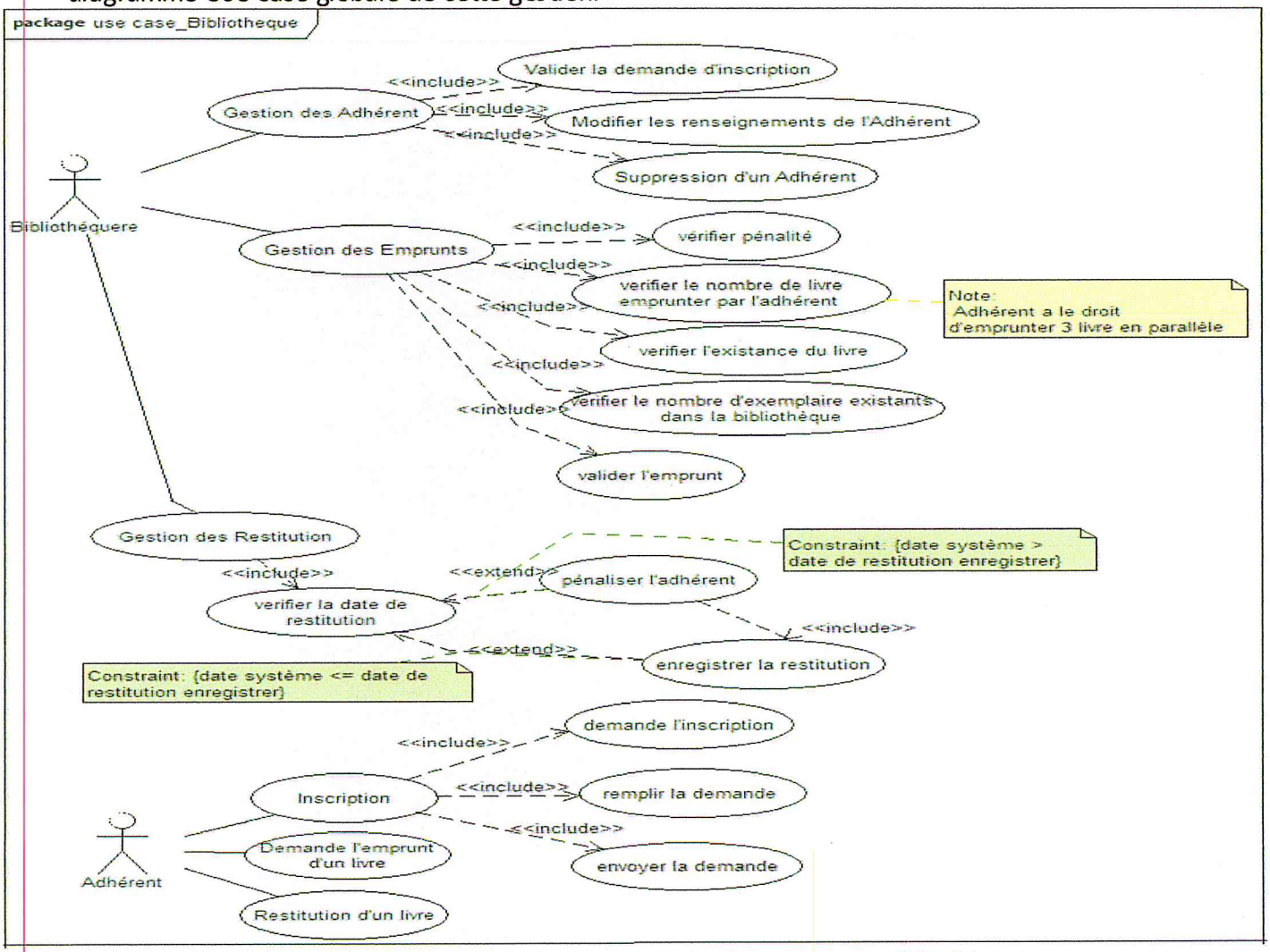


Figure 74: Modèle PIM UML Use Case globale de la Gestion Bibliothèque

La première étape dans le projet MDA est de créer le modèle UML PIM source du système étudié, et cette création se fait avec l'éditeur des modèles UML « TOPCASED d'eclipse», par les étapes suivantes (**Figure 75**) :

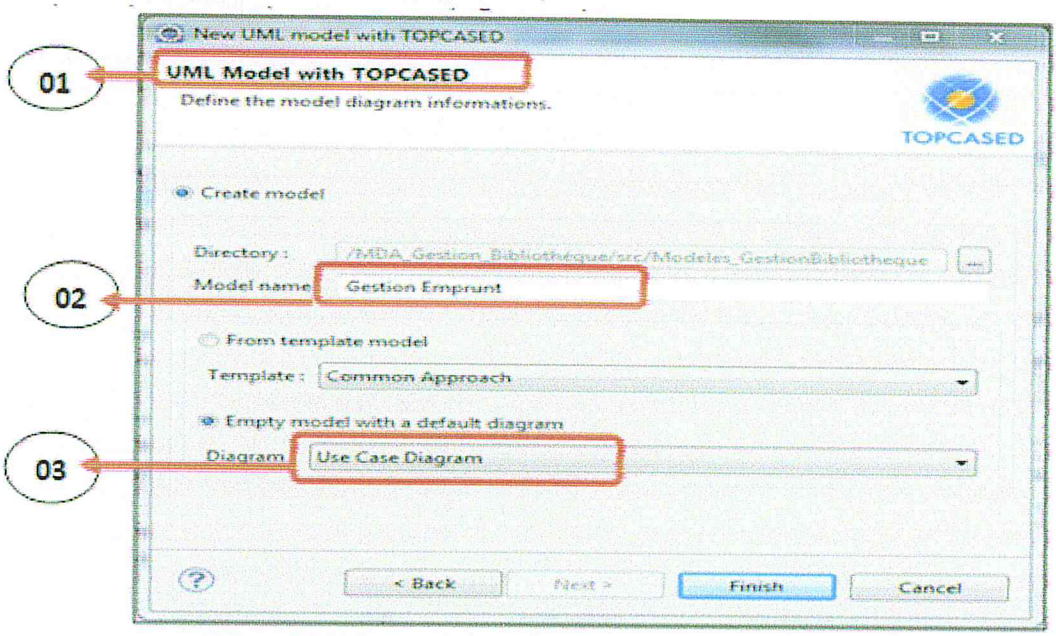


Figure 75: nom et type de diagramme UML PIM source

➢ **Explication de la figure**

1. Sur la liste des «*New Wizard* » en sélectionnant un nouveau projet TOPCASED «*UML Model with TOPCASED*».
2. En donnant le nom du diagramme à modéliser «*Gestion Bibliothèque* ».
3. En sélectionnant le type de diagramme à modéliser.

Le modèle PIM UML de la Gestion d'Emprunt éditée avec l'éditeur TOPCASED est présenté dans la figure suivante.

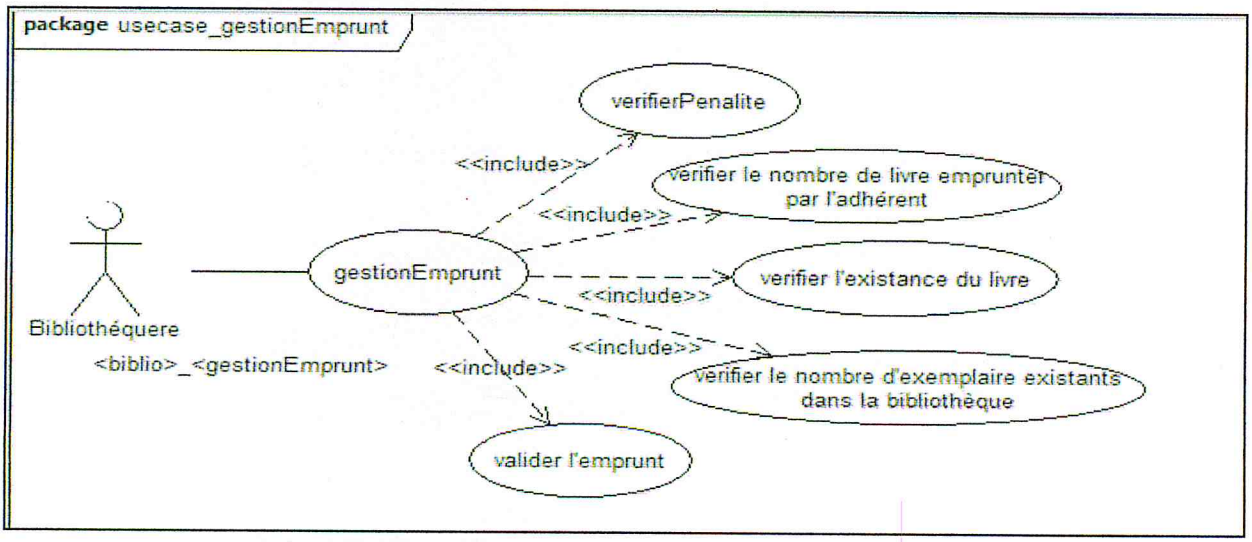


Figure 76: Modèle PIM UML Use Case du Gestion des emprunts des livres

L'éditeur Topcased permet de générer un format d'arborescence (XML) pour chaque diagramme UML modélisé, nous présentons ci-dessous le format d'arborescence du diagramme cas d'utilisation PIM du Gestion Emprunt :

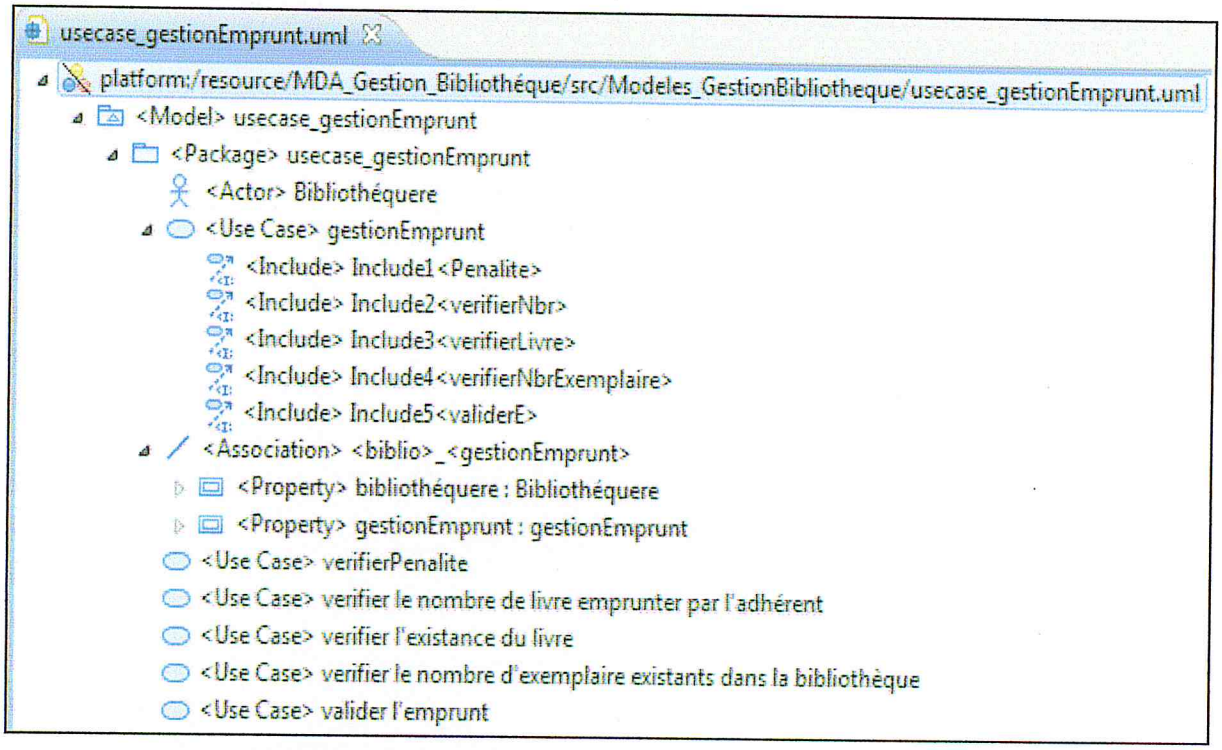


Figure 77: l'arborescence du modèle PIM UML Use Case du Gestion Emprunt

- Dans ce chapitre on va appliquer les transformations de modèle UML vers modèle UML pour arriver à générer le code source de l'application Web selon la plateforme JEE de la Gestion des emprunts des livres dans la bibliothèque.
- Il y a deux grandes étapes de transformation de : modèle UML PIM vers modèle UML PIM et modèle UML PIM vers modèle UML PSM et aussi modèle UML PSM vers texte (génération de code source).

1. Transformations de PIM vers PIM avec ATL

➢ **UseCase2Activity_PIM**: transformer le modèle PIM use case vers le modèle PIM Activité par l'application du profil UML *Profil_UseCase_Activity*.

➢ **UseCase2Classe_PIM**: transformer le modèle PIM use case vers le modèle PIM Classe par l'application du profil UML Profil_UseCase_Classe.

➢ **Activity2Sequence_PSM**: transformation du modèle PIM Activité vers le modèle PSM Séquence par l'application du profil UML *Profil_ Activity_sequence*.

2. Transformations de PSM vers Code source : à partir de modèle PSM Séquence et avec l'application du profil UML *Profil_Sequence_code* sur ce modèle PSM, on va générer le code source Java de l'application Web.

3. Transformation de M2M avec ATL

Après la validation de modèle PIM Source (diagramme use case), en commence à lui appliquer les transformations avec ATL pour générer les modèles PIM Cible.

3.1. Transformation PIM vers PIM

❖ **UseCase2Activity** : La première transformation est faite par l'application du profil UML «*Profil_UseCase_Activity*» sur le modèle PIM source (diagramme cas d'utilisation), et en exécutant le scripte de transformation «*UseCase2Activity_PIM.atl*», nous donne le modèle PIM Cible (diagramme d'activité).

a. préparation du modèle source : La figure suivante montre la manière d'appliquer un profil UML sur un diagramme UML édité sur Topcased sous Eclipse :

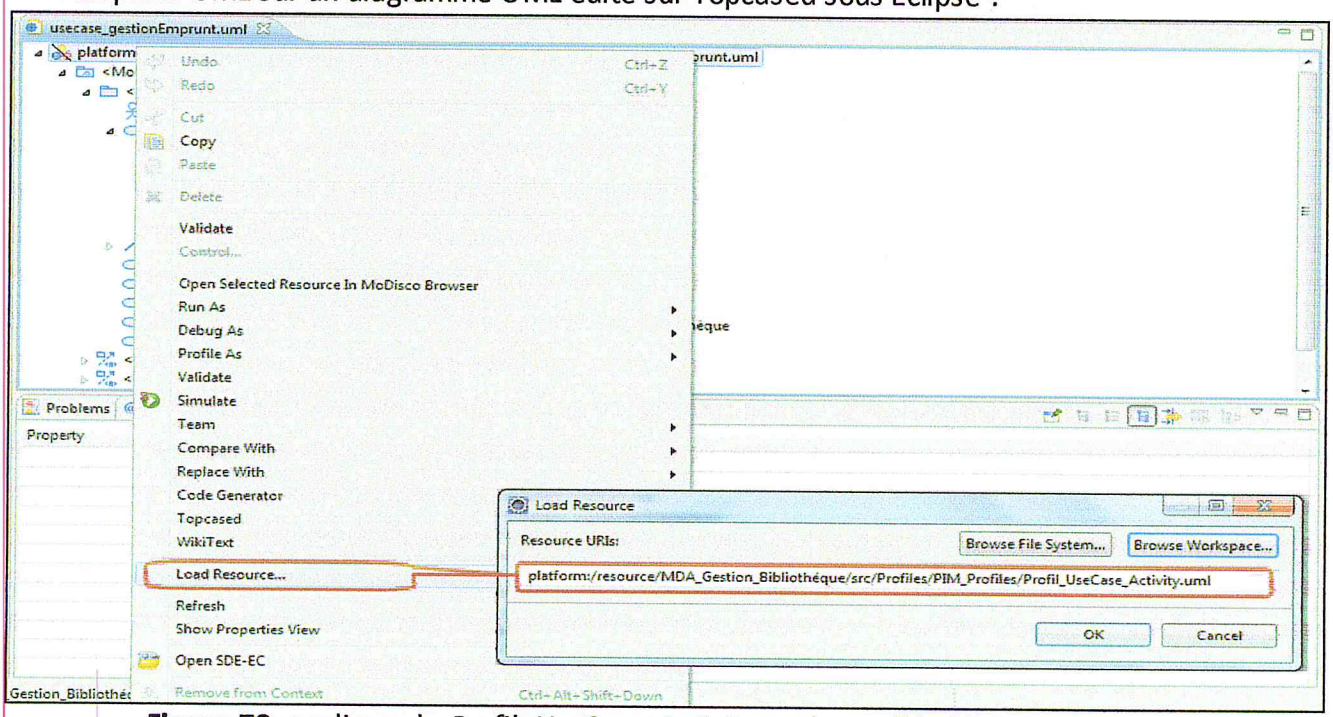


Figure 78: appliquer le *Profil_UseCase_Activity* sur le modèle PIM Use Case source

Nous présentons ci-dessous le diagramme cas d'utilisation PIM du *Gestion Emprunt* après l'application du profil UML *Profil_UseCase_Activity* et tous ses stéréotypes nécessaires pour la transformation d'un modèle PIM cas d'utilisation vers le modèle activité :

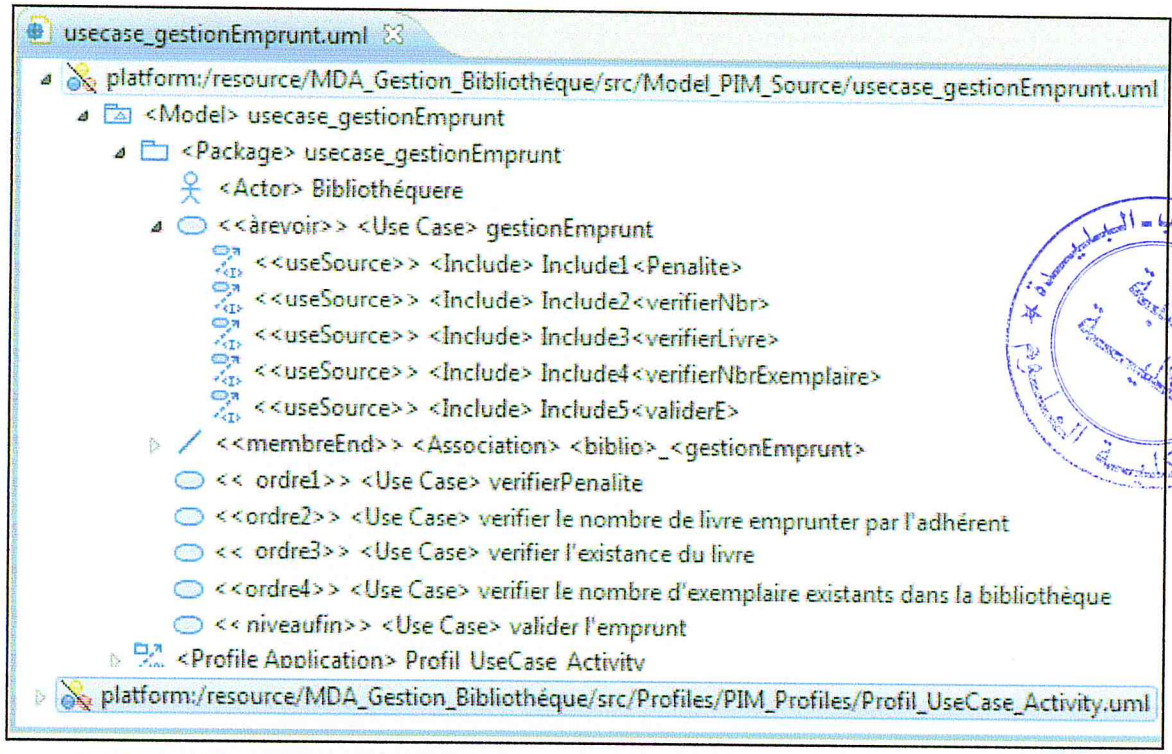


Figure 79: le modèle PIM Use Case après l'application du *Profil_UseCase_Activity* et ses stéréotypes

b. Exécution de scripte de transformation

- La première étape pour exécuter un scripte de transformation «*UseCase2Activity_PIM*.atl» sous Eclipse : un clic droit sur ce fichier puis « Run As » en suite « ATL transformation » comme montre la figure ci-après, puis Eclipse ouvre une boite de dialogue «Run configuration» pour configure l'exécution de la transformation.

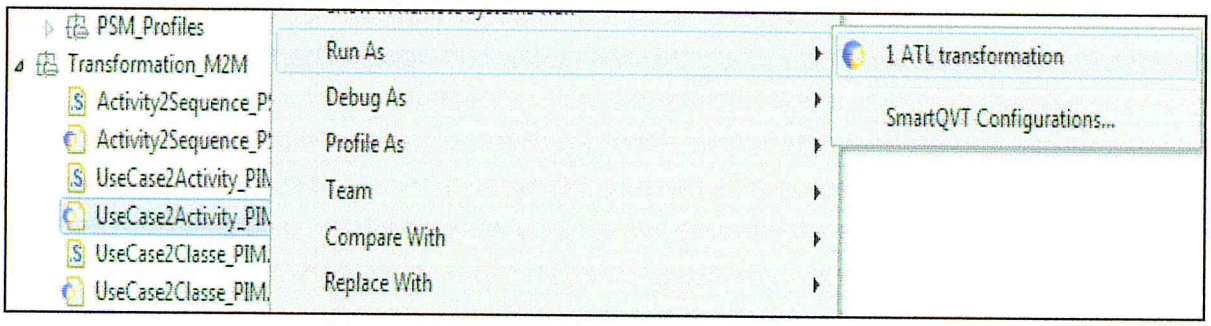


Figure 80 : Exécuter la transformation de modèle «*.atl* »

- La deuxième étape pour exécuter un scripte de transformation «*UseCase2Activity_PIM*.atl» est la configuration d'une transformation ATL est devisée en 4 parties :

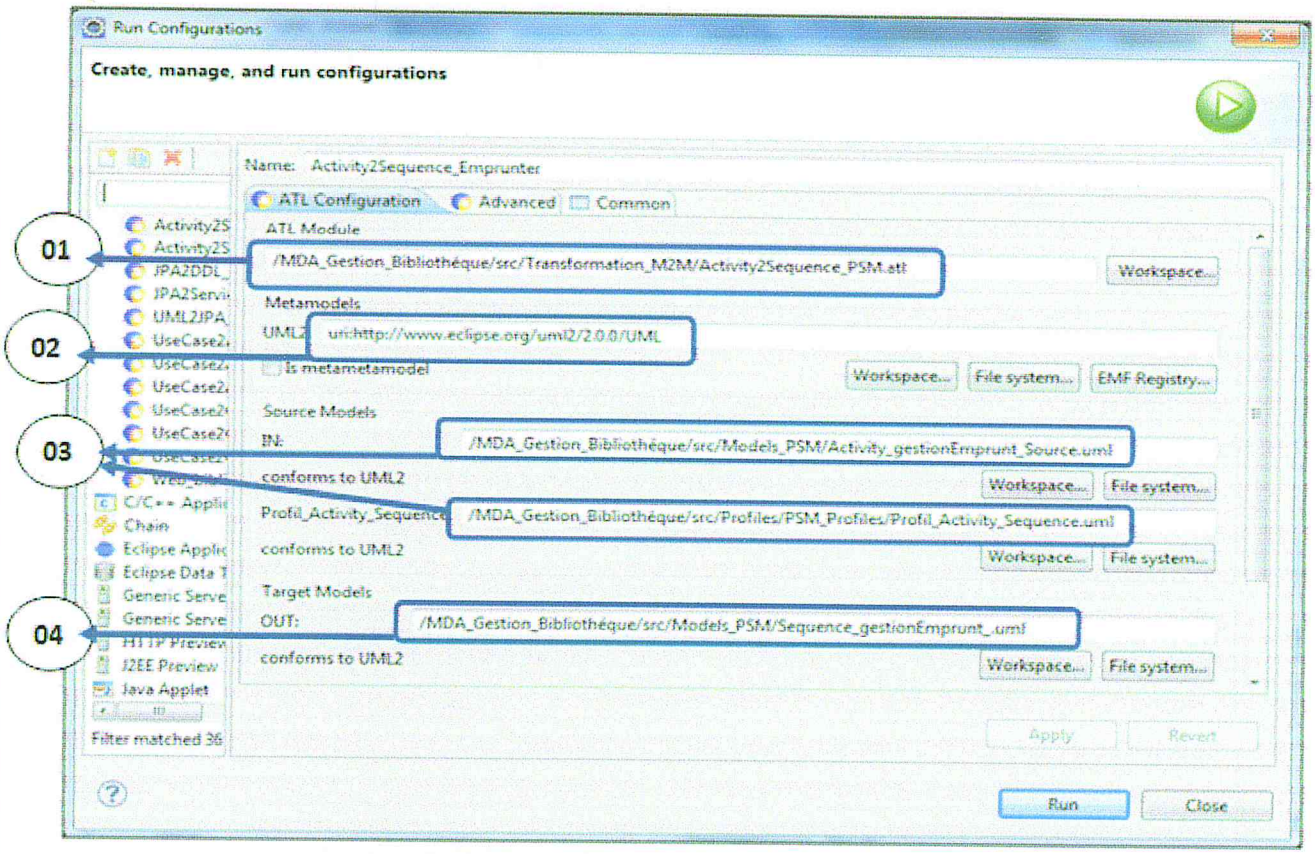


Figure 81: La boite de dialogue "Run Configuration" de scripte *UseCase2Activity_PIM*

➢ **Explication de la figure**

1. **ATL Module:** le chemin relatif du scripte «*UseCase2Activity_PIM.atl* » dans le projet MDA.

2. **Metamodels**: liste des URI ou chemins des méta-modèles utilisés dans la transformation, ici nous n'avons qu'une seule URI Celle d'UML2.0 (stable pour toutes les transformations).

3. **Source Modeles**: on trouve dans cette partie la liste des modèles source de cette transformation, on a deux modèles: le modèle PIM (use case *gestion emprunt*) et le profile *Profil_UseCase_Activity*.

4. **OUT** : dans ce champ on doit spécifier le répertoire et le nom du modèle cible (diagramme d'activité) généré après cette transformation.

- la Figure suivante représente le résultat de transformation est: le modèle activité PIM Cible nommé «Activity_gestionEmprunt.uml» ouvert par l'éditeur TOPCASED.

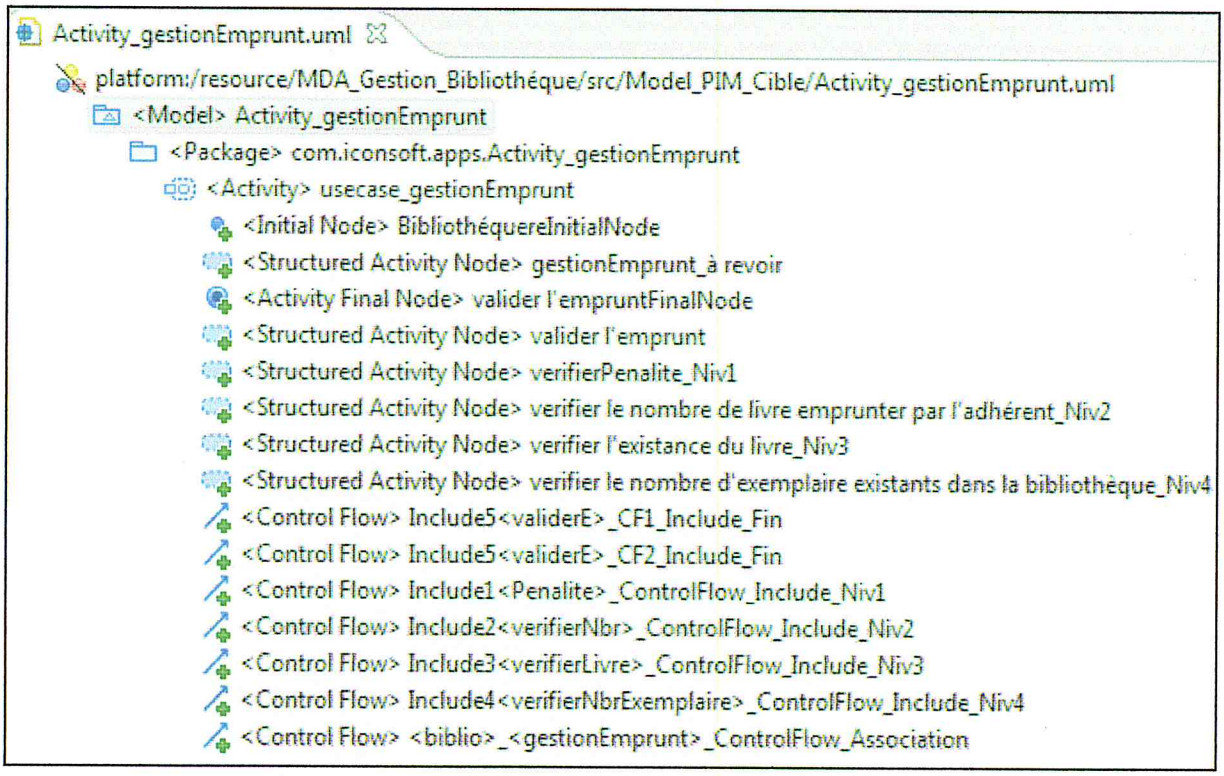


Figure 82: le modèle PIM '*Activity_gestionEmprunt*' résultat de transformation *UseCase2Activity_PIM*

❖ **UseCase2Class :** La deuxième transformation ATL appliqué sur le modèle PIM cas d'utilisation de '*Gestion d'emprunt*', par l'application du profil «*Profil_UseCase_Classe*», et en exécutant le scripte «*UseCase2Classe_PIM.atl*», elle nous donne un modèle PIM diagramme de classe de domaine basique.

a. préparation de modèle source

Nous présentons ci-dessous le diagramme cas d'utilisation PIM du *Gestion Emprunt* après l'application du profil UML *Profil_UseCase_Class* et tous ses stéréotypes nécessaires pour la transformation d'un modèle PIM cas d'utilisation vers le modèle diagramme de classe PIM :

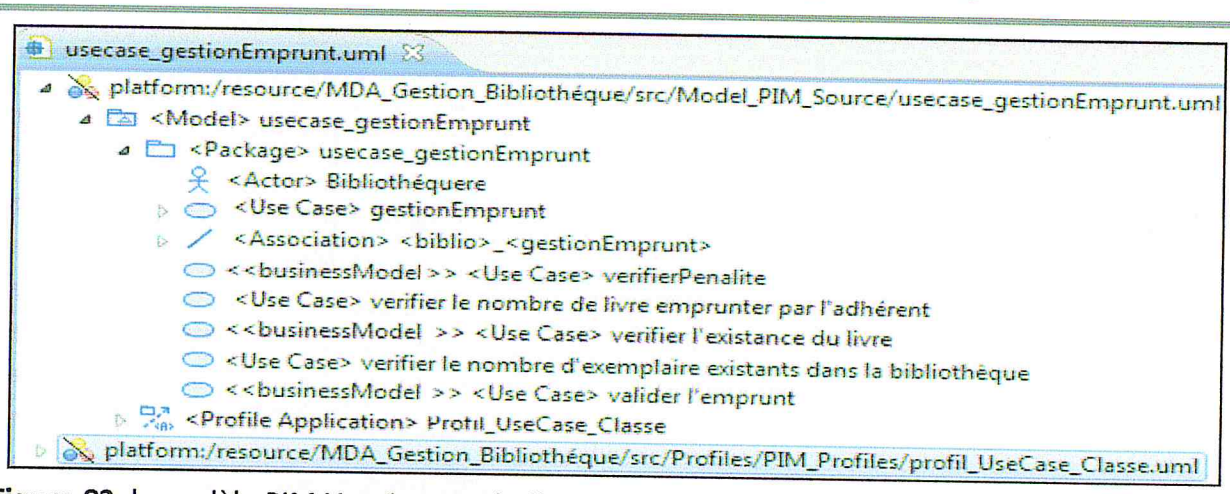


Figure 83: le modèle PIM Use Case après l'application du *Profil_UseCase_Class* et ses stéréotypes

b. exécution de scripte de transformation

- pour exécuter un scripte de transformation «*UseCase2Class_PIM.atl* » est la configuration d'une transformation ATL est devisée en 3 parties :

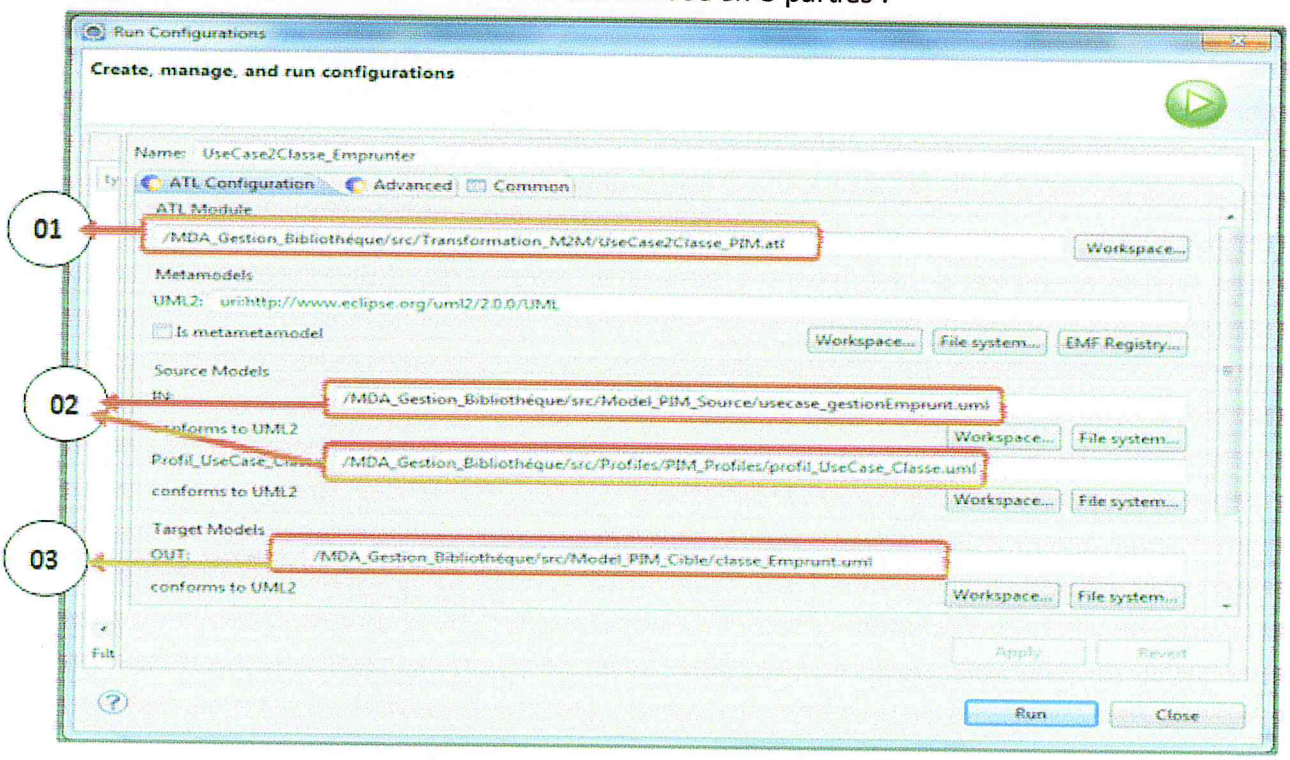


Figure 84 : La boite de dialogue "Run Configuration" de scripte *UseCase2Class_PIM*

➢ **Explication de la figure**

1. **ATL Module:** le chemin relatif du scripte «*UseCase2Class_PIM.atl* »dans le projet MDA.

2. Source Modeles: on trouve dans cette partie la liste des modèles source de cette transformation, on a deux modèles : le modèle PIM (use case *gestion emprunt*) et le profile *Profil_UseCase_Class*.

3. OUT : dans ce champ on doit spécifier le répertoire et le nom du modèle cible (diagramme de classe) généré après cette transformation.

le résultat de cette transformation est présenté dans la figure ci-dessous :

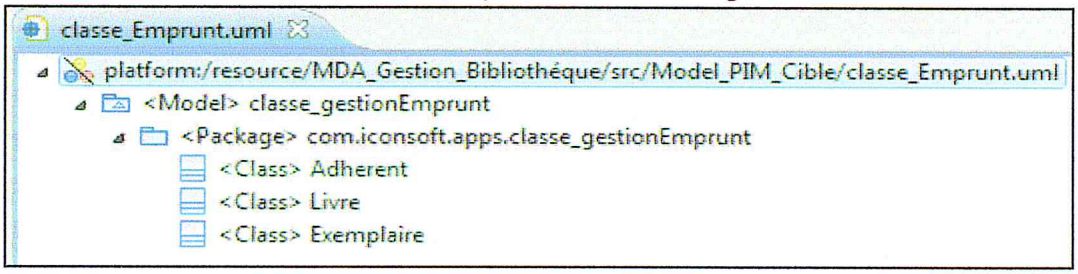


Figure 85: le modèle PIM *'Class_Gestion emprunt'* résultat de transformation *UseCase2Classe_PIM*

C. exécution de l'AGL de diagrammes de structuration

Après le raffinement de modèle de domaine (diagramme de classe) générer précédemment, on va l'utilisé comme modèle source pour les transformations de l'AGL (de diagramme de structuration) existant au niveau de l'entreprise **IconSoftware**, afin de générer le diagramme de classe PSM correspondant (diagramme de classe dépond de la plateforme JEE), pour l'utiliser dans la transformation : Activity2sequence_PSM.

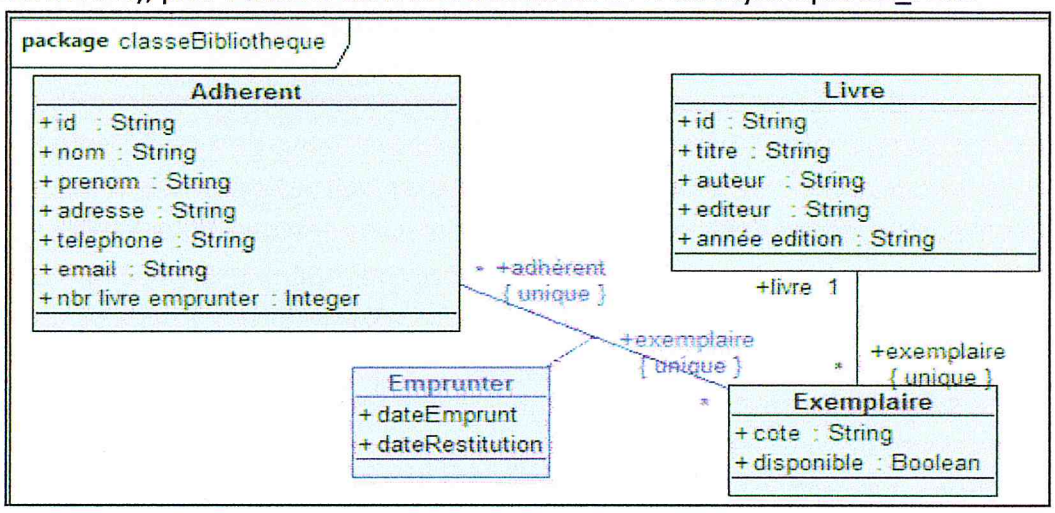


Figure 86: le modèle *'Class_Gestion Emprunt'* PIM Raffiner

Après l'exécution des transformations dans l'AGL existant, nous avons comme résultat un diagramme de classe PSM 'pour les classes *Service* et *DAO*', représenté dans la figure ci-dessous :

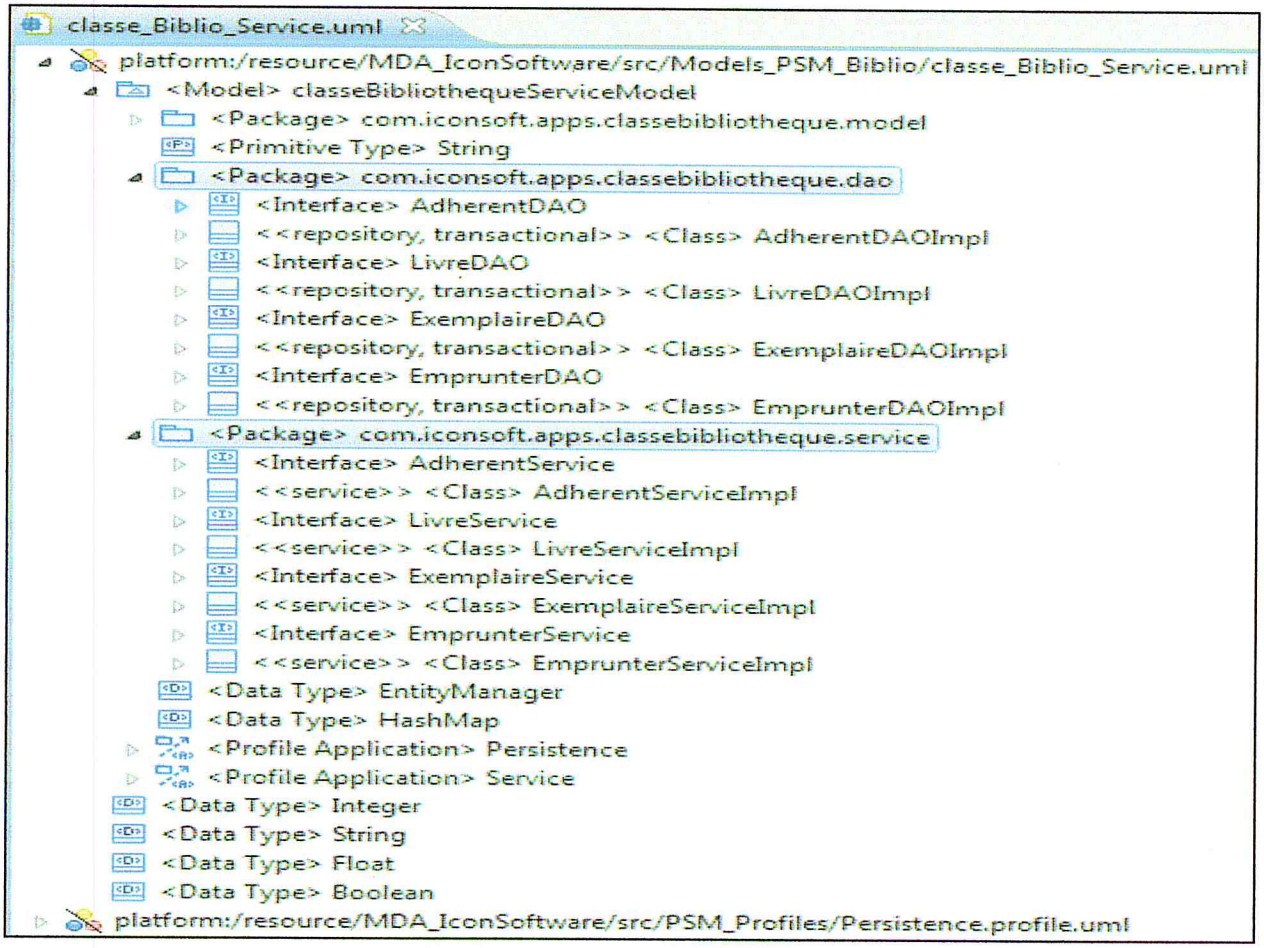


Figure 87: le modèle '*Class_Biblio_Service*' PSM Service et DAO (Data Access Object)

3.2. Transformation PIM vers PSM

❖ **Activity2Sequence :** La troisième transformation ATL appliqué sur le modèle PIM UML diagramme d'activité de '*gestion des emprunts*' résultat de la première transformation (Figure 82), par l'application du profil «*Profil_Activity_Sequence*» et en exécutant le scripte «*Activity2Sequence_PSM.atl*», elle nous donne un modèle PSM diagramme de séquence.

a. **préparation de modèle source :** Nous présentons ci-dessous le diagramme d'activité PIM de la *gestion d'emprunt* après l'application du profil UML *Profil_Activity_Sequence* et tous ses stéréotypes nécessaires pour la transformation d'un modèle PIM activité vers le modèle PSM diagramme de séquence:

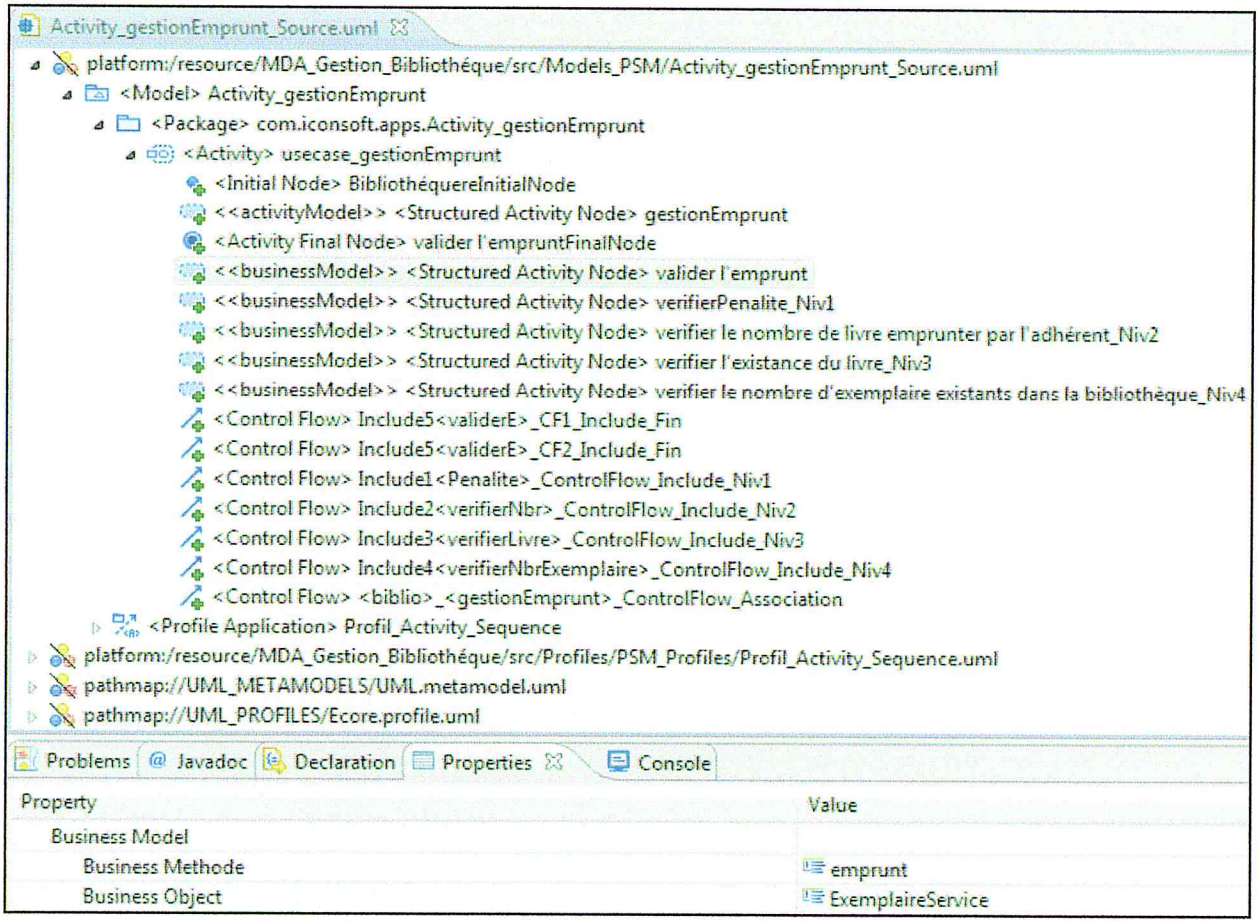


Figure 88: le modèle PIM Activité après l'application du *Profil_Activity_Sequence* et ses stéréotypes

b. **exécution de scripte de transformation**

- pour exécuter un scripte de transformation «*Activity2Sequence_PSM.atl*» est la configuration d'une transformation ATL est devisée en 3 parties :

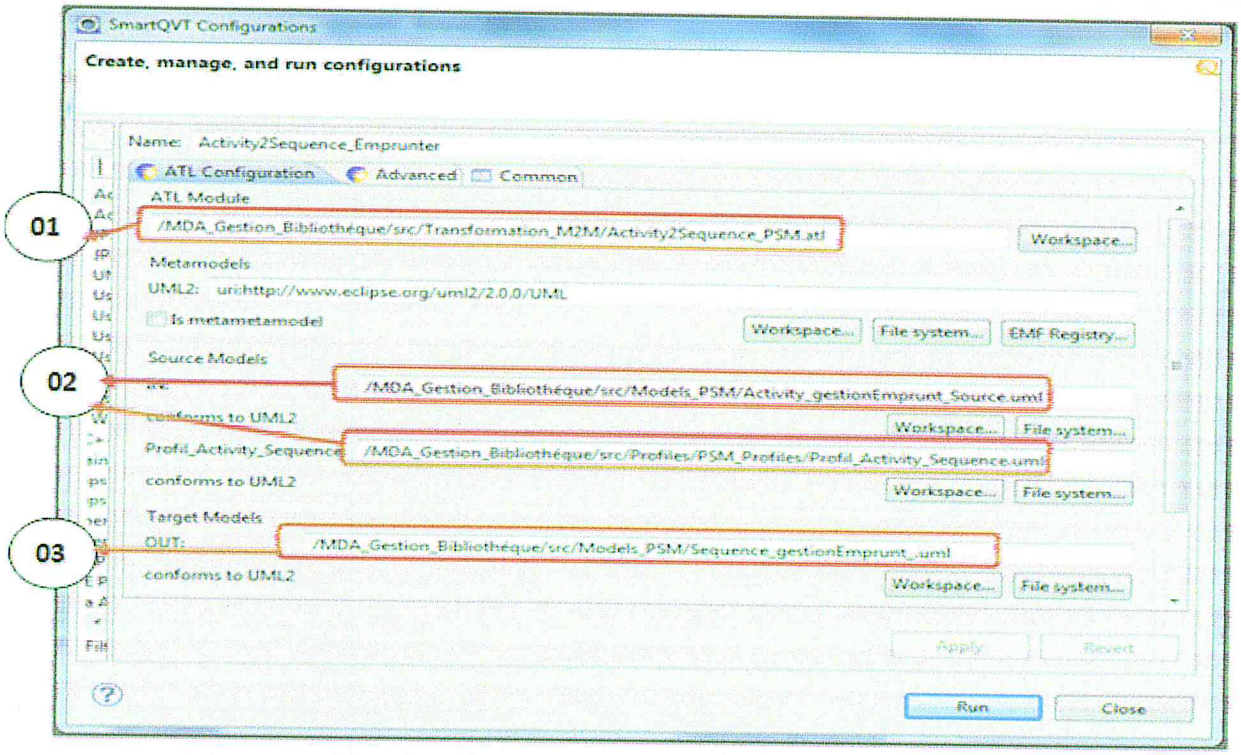


Figure 89: La boite de dialogue "Run Configuration" de scripte Activity2Sequence_*PSM*

➢ **Explication de la figure**

1. **ATL Module:** le chemin relatif du scripte «*Activity2Sequence_PSM.atl*» dans le projet MDA.

2. **Source Modeles**: on trouve dans cette partie la liste des modèles source de cette transformation, on a deux modèles : le modèle PIM (activité *gestion emprunt*) et le profile *Profil_Activity_Sequence*.

3. **OUT** : dans ce champ on doit spécifier le répertoire et le nom du modèle cible (diagramme de Séquence) généré après cette transformation.

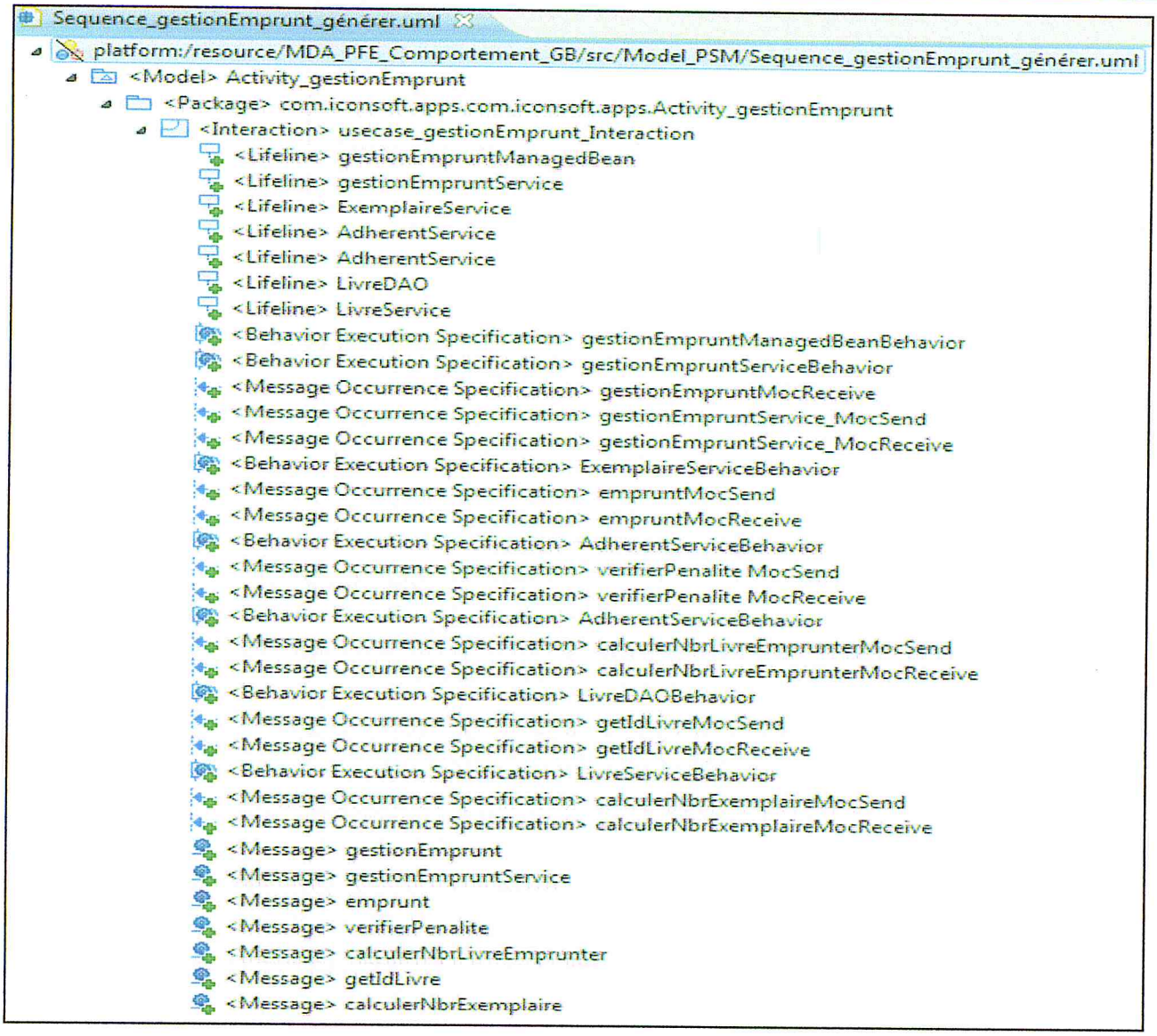


Figure 90: le modèle PSM '*Séquence_Gestion emprunt*' résultat de transformation *Activity2Sequence_PSM*

4. Génération de code source java avec Acceleo

Après avoir réalisé les transformations de modèle PIM vers PIM et de PIM vers PSM, il est temps de générer le code source de l'application à partir du modèle PSM UML, diagramme de séquence de '*gestion des emprunts*' résultat de la troisième transformation (Figure 90), par l'application du profil «*Profil_Sequence_Code*» et en exécutant les scriptes de génération (templates), pour arriver au code source de l'application web 'gestion des emprunts des livres de la bibliothèque' selon la plateforme JEE.

a. préparation de modèle source

Sur le diagramme de séquence générer précédemment (Figure 90), nous devrons appliquer des raffinements tel que : l'ajout des messages de type ***reply***, l'ajout des lignes de vies

Puis nous devrons appliquer le profil UML « *Profil_Sequence_Code* » et ses stéréotypes nécessaire pour utiliser ce diagramme de séquence comme modèle source dans la génération de code source JEE.

Dans les figures ci-dessous nous présentons quelques exemples des éléments de diagramme de séquence après les raffinements et l'application des stéréotypes:

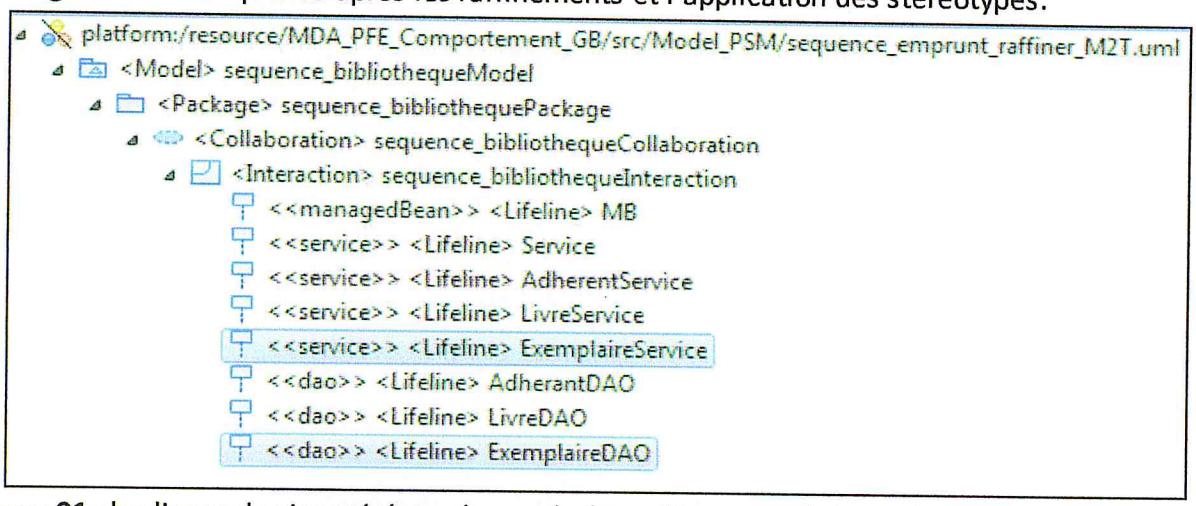


Figure 91 : les lignes de vies stéréotypées après les raffinements (ajouts des nouvelles lignes de vies)

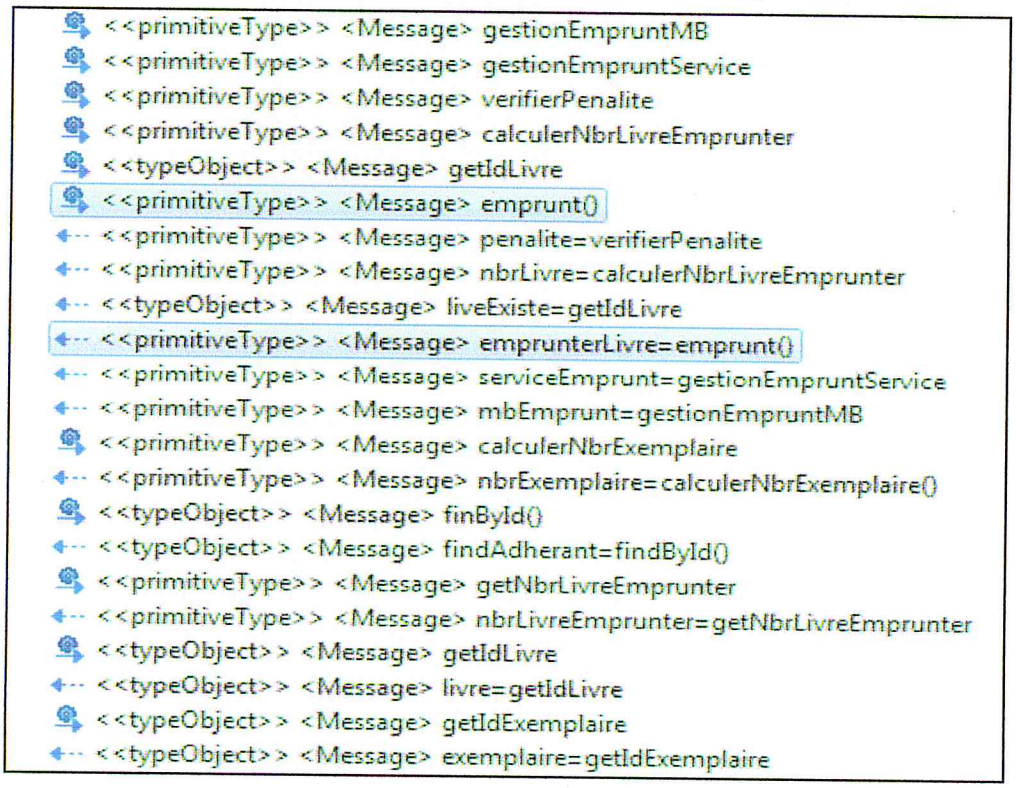


Figure 92 : les messages stéréotypés après le raffinement (ajout des messages de réponse)

b. Exécution du scripte de génération de code source JEE

Après la préparation du modèle diagramme de séquence PSM source de la génération, et avant la préparation de la chaine d'exécution «.chain», nous devrons créer un nouveau projet Java sous Eclipse avec le nom « **Gestion Bibliothèque** », qui sera le projet cible de la génération du code source de l'application web JEE «gestion des emprunts ».

Pour préparer la chaine d'exécution nous devrons sélectionner les éléments suivants :

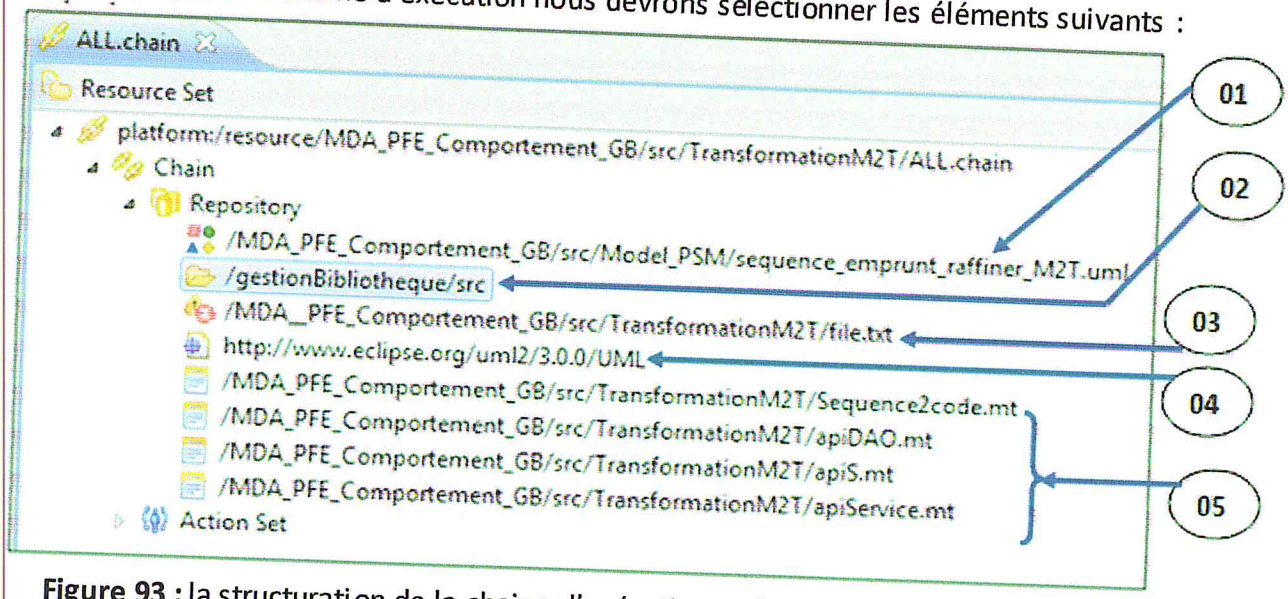


Figure 93 : la structuration de la chaine d'exécution «.chain» de la génération du code JEE

➢ **Explication de la figure**

1. le chemin de modèle source de la génération (diagramme de séquence PSM)
2. le chemin du projet Java cible de cette génération
3. le chemin du fichier journal des erreurs de la génération
4. le chemin du métamodèle du modèle source de cette génération
5. les chemins des différentes scriptes (Template) de génération

Pour lancer la génération, un clic droit sur le fichier «All.chain » puis « Launch », pour exécuter toutes les Template de génération

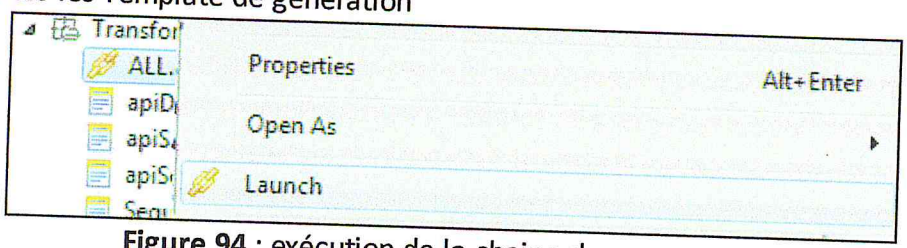


Figure 94 : exécution de la chaine de

Le résultat de cette génération est un projet java présenté par la figure ci-dessous :

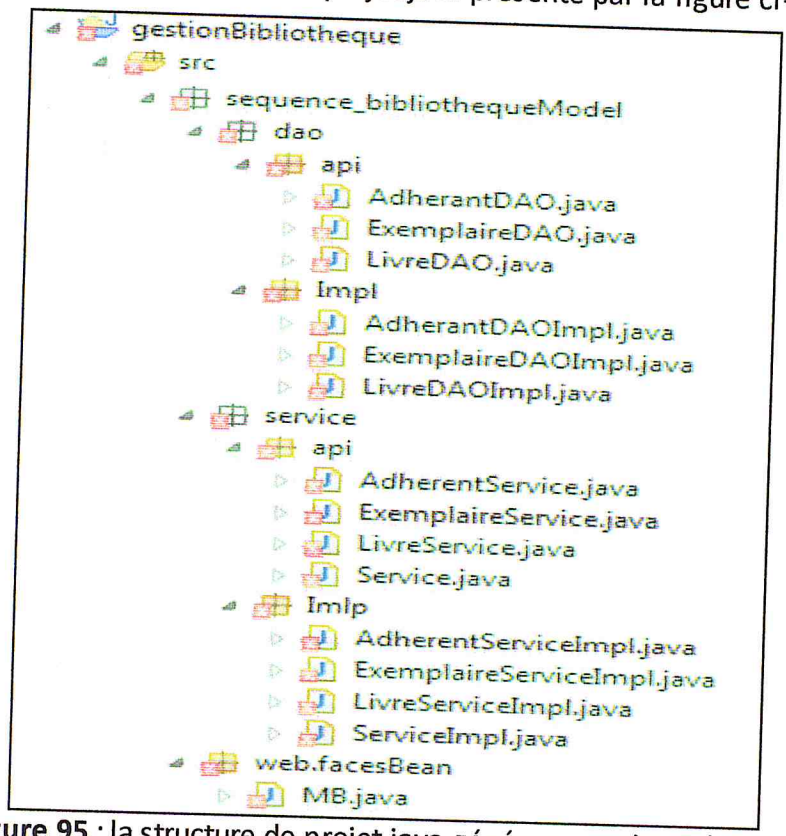


Figure 95 : la structure de projet java générer «gestion Bibliothèque»

5. Conclusion

Dans ce chapitre, nous avons présenté les différentes phases de transformation et de génération à réaliser afin de procéder à la génération du code source de l'application web selon la plateforme JEE conforme au service de la **gestion des emprunts** des livres dans une bibliothèque.

En commençant par l'ensemble de transformations M2M sur les diagrammes UML de comportement à savoir la transformation d'un modèle PIM vers un modèle PIM (cas d'utilisation vers classe et cas d'utilisation vers activité) et la transformation d'un modèle PIM vers un modèle PSM (activité vers séquence) pour aboutir à la génération de code source M2T à partir d'un modèle PSM (séquence vers code source).

Conclusion Générale et Perspective

Conclusion générale

Dans ce mémoire, nous avons développé un **AGL** selon la démarche **MDA** au sein de l'entreprise **IconSoftware**. L'AGL développé permet :

- le passage d'un modèle de domaine à un autre modèle de domaine (digramme de cas d'utilisation vers diagramme de classe, diagramme cas d'utilisation vers diagramme d'activité) ;
- le passage d'un modèle de domaine à un modèle spécifique à la plateforme JEE (diagramme d'activité vers digramme de séquence) ;
- la génération de code technique selon les conventions de la plateforme JEE (séquence vers code source) ;

Le travail réalisé avait pour objectif de compléter un travail existant au sein de l'entreprise **IconSoftware** basé sur les transformations des diagrammes UML de structuration [2].

Pour mieux exposer notre travail, Nous avons présenté en premier un état de l'art sur les principaux concepts sur lesquels porte notre travail à savoir : le concept d'**AGL**, l'**IDM** et l'approche **MDA** pour le développement de logiciels ainsi que les concepts relatifs à la plateforme de développement **JEE** et aux transformations de modèles.

Nous avons ensuite identifié les besoins de l'entreprise d'accueil IconSoftware à savoir le développement d'un AGL prenant en considération les **deux catégories de digrammes UML (de structuration et de comportement)**. En effet, un AGL a été développé au sein de l'entreprise se basant uniquement sur les digrammes de structuration réalisé par les deux ingénieurs R. Boumendjas et H. Bekkouche [2].

Par la suite, nous avons présenté la démarche adoptée pour la mise en œuvre de notre AGL. Nous avons décrit les différents **modèles UML manipulés** et les **méta-modèles correspondants** à savoir le diagramme des cas d'utilisation, le diagramme d'activité et le diagramme de séquence. Une description détaillée des différentes transformations et génération établies au sein de l'AGL (la transformation **UseCase2Classe**, **UseCase2Activity**, **Activity2Sequence** et **Sequence2code**) en décrivant les différentes correspondances entre les éléments des méta-modèles manipulés.

Nous avons développé un certain nombre de profils UML permettant de faciliter les différentes transformations et génération de code technique :

- Le profil **Profil_UseCase_Classe** pour le passage d'un diagramme de cas d'utilisation à un diagramme de classe ;
- Le profil **Profil_UseCase_Activity** pour le passage d'un diagramme de cas d'utilisation à un diagramme d'activité ;
- Le profil **Profil _Activity_Sequence** pour le passage d'un diagramme d'activité à un diagramme de séquence ;
- Le profil **Profil_sequence_code** pour le passage d'un diagramme de séquence au code technique.

Enfin, pour la conception de notre AGL nous avons adopté le **processus UP**. Pour l'implémentation, nous avons utilisé l'environnement de développement **Eclipse**. Ceci en intégrant un certain nombre de **plugins** pour éditer les différents modèles et méta-modèles manipulés, spécifier les profils UML et mettre en œuvre (spécification et exécution) les différentes transformations de modèles et la génération du code JEE.

Deux majeures catégories de transformations de modèles ont été distinguées :

- Transformation de modèles Indépendants d'une plateforme vers le même type de modèle (**PIM vers PIM**). Cette catégorie de transformation inclut deux transformations : la transformation d'un **diagramme des cas d'utilisation** vers le **diagramme de classes** et la transformation d'un **diagramme des cas d'utilisation** vers un **diagramme d'Activité**.
- Transformation de modèles indépendants d'une plateforme vers des modèles spécifiques à une plateforme (**PIM vers PSM**), qui est dans notre cas il s'agit de la **plateforme JEE**. Cette catégorie inclut la transformation du **diagramme d'activité (PIM)** vers le **diagramme de séquence (PSM)**.

Nous avons spécifié les différentes transformations de modèles (PIM vers PIM et PIM vers PSM) à l'aide du **langage ATL**. Le code source a été généré à partir d'un modèle PSM à l'aide du **générateur Acceleo** selon les conventions de la plateforme JEE.

Afin d'illustrer et valider notre travail, nous avons pris le cas d'un **système d'information de gestion de bibliothèque**. Nous nous sommes intéressés plus particulièrement à la **gestion des emprunts**.

Perspectives

En perspective, le travail présenté dans ce mémoire peut être complété en intégrant les éléments des méta-modèles non pris en considération lors des transformations et la génération de code source. En effet :

- Lors de la **transformation d'un diagramme des cas d'utilisation**, nous n'avons pas manipulé tous les éléments du méta-modèle auquel ce digramme est conforme. Par exemple, la relation Extends entre deux cas d'utilisation. Nous proposons de faire correspondre cet élément à l'élément DecisionNode du méta-modèle correspond au digramme d'activité.
- Lors de la **transformation d'un diagramme d'activité**, nous n'avons pas pris en considération un certain nombre d'éléments du méta-modèle correspondant à savoir : Activity Edge (Object Flow), Activity partition, ActivityNode (DecisionNode, joinNode, ForkNode, ObjectNode, OpaqueAction).
- Des éléments du méta-modèle relatif au diagramme de séquence n'ont pas été pris en compte lors de la **transformation d'un digramme de séquence** : Interaction Use, state invariant, combined Fragment, Interaction Operand, Interaction Constraint, Interaction Operator.

Bibliographie

[01] : ANDRE Sylvain MDA (Model Driven Architecture) principes et états de l'art. Conservatoire National des Arts et Métiers – novembre 2004

[02] : Boumenjas Med Riad, Bekkouche Hocine Mise en place d'un atelier génie logiciel « AGL »par la démarche de développement MDA en 2010,2011

[03] : Un atelier de genie logiciel dedié a l'estimation du cout logiciel *par* houria laouti et soumia meflahi _Université Des Sciences Et De La Technologie D'Oran Mohamed Boudiaf - 2008

[04] : Ateliers de Génie Logiciel Christine Solnon 1996/97

[05] : Ateliers de Génie Logiciel, Notes destinées aux étudiants---2011 *Amélie Cordier, Marie Lefevre*

[06] : Ateliers de Génie Logiciel NFE121 : Licence professionnelle analyste-concepteur des systèmes d'information et de décision -Virginie Thion, CNAM Paris

[07] : Blanc Xavier MDA en action, Groupe Eyrolles,Paris, 2005.

[08] : Jean-Marc Jézéquel1_ Sébastien Gérard2, Chokri Mraidha2, et Benoît Baudry2 Approche unificatrice par les modèles

[09] : Sylvain ANDRE MDA (Model Driven Architecture) principes et états de l'art. Soutenu le 05 novembre2004

[10] : Mounir GRARI. Principes et états de l'art de l'approche MDA et applications pour des plates- formesPHP orienté 3-tiers. UNIVERSITÉ MOHAMMED PREMIER, 2007.

[11] : Pascal Roques le cahier de programmeur UML2 –modéliser une application web- Groupe Eyrolles,4iéme edition en 2007

[12] : Hadjar DAOUADJI, Imane CHIKHI Mémoire Génération d'une ontologie OWL à partir du méta-modèle SPEM,USDB, 2009.

[13] : MDA La technologie de l'architecture pilotée par le modèle (Model-Driven Architecture) http://www.modeliosoft.com/technologies/mda.html

[14] : BUSINESS FIRST Agile BPM Applications http://www.w4.eu/model-driven.htm

[15] : Freddy Allilaire &Tarik Idrissi _ ADT Eclipse development tools for ATL Université de Nantes_Faculté de Sciences et Techniques LINA (Laboratoire d'Informatique de NantesAtlantique) Nantes Cedex 3, France

[16] : Antoine Wiedemann en juin 2006

Mémoire : Approche MDA pour la transformation d'un diagramme de classes conforme UML 2.en un schéma relationnel conforme CWM et normalisé_L.

[71] : Jean-Philippe Babau Ingénierie Dirigée par les Modèles _ Transformation de modèles Laboratoire Lab-STICC

[18] : ATLAS group_ LINA & INRIA de Nantes ATL: Atlas Transformation Language _ATL User Manual - version 0.7 –2006

[19] : Benoît Combemale, Xavier Crégut, Marc Pantel, Présentation d'OCL 2. Un langage de requête pour exprimer la sémantique statique des modèles. Université de Toulouse, octobre 2007.

[20] : Jean Bézivin, Erwan Breton, Grégoire Dupé, Patrick Valduriez,
The ATL Transformation-based Model Management Framework, RESEARCH REPORT, No 03.08, Nantes, Septembre 2003.

[21] : Eric Cariou Ingénierie des Modèles Transformation de modèles _Université de Pau et des Pays de l'Adour UFR Sciences Pau _2007_ Eric.Cariou@univ-pau.fr

[22] : Pierre Parrend, Mars 2005 Introduction pratique au Développement orienté Modèle,

[23] : Farah FOURATI Une approche IDM de transformation exogène de Wright vers Ada _ L'École Nationale d'Ingénieurs de Sfax Juin 2010

[24] : http://www.obeo.fr/pages/acceleo/fr

[25] :Obeo Acceleo 2.6 : Guide utilisateur. 2008.

[26] : Cedric Dumoulin Introduction à l'Ingenierie Dirigée par les Modeles (figure du modele au code)

[27] : OMG Unified Modeling Language TM (OMG UML) Superstructure_Version *2.2* formal/2009-02-02

[28] : OMG Unified Modeling Language 2.2 specification

[29] : Graph and Model Transformation Tools for Model Migration
http://www4.in.tum.de/~schaetz/papers/model_migration_journal.pdf

[30] : mistra http://www.mistra.fr/ 10/03/2013

[31] : Jérôme Molière Le cahier des programmeurs J2EE **EYROLLE** [Jérôme Molière en décembre 2004]

[32] : http://www.nalis.fr/accueil.html/MVC, 15/03/2013

[33] : Jean-Michel DOUDOUX Développons en Java chapitre 86 : Les Frameworks

[34] : Antonio Goncalves Les cahiers des programmeurs J2EE 5–Eyrolles- mai 2007

[35] : Jean-Michel DOUDOUX Développons en Java (chapitre 41)

[36] : www.developpez/faqjsf.com dernière mise à jour 22 décembre 2009, Date de consultation 21/03/2013

[37] : les Cahiers du Programmeur SWING **EYROLLES**

[38] : Pascal Roques UML2 « Les Cahiers des Programmeurs » modéliser une application web **EYROLLES** 4ieme Edition

[39] : Méthodologies de développement de logiciels de gestion Chapitre 6 Le Processus unifiéde développement logiciel Partie I Les concepts
réalisée par P.-A. Sunier Professeur à la HE-Arcde Neuchâtel _http://lgl.isnetne.ch

[40] :Eclipse http://www.Eclipse.org/.

[41] :Topcased http://www.topcased.org/.

Annexe

Annexe

1. Object Management Group *OMG* est un organisme international (américain) à but non lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes. [28]

L'OMG est notamment à la base des standards UML (*Unified Modeling Language*), MOF (*Meta-Object Facility*), CORBA (*Common Object Request Broker Architecture*) et IDL (*Interface Definition Language*).[12]

L'OMG est aussi à l'origine de la recommandation MDA (*Model Driven Architecture*) ou ingénierie dirigée par les modèles, avec en particulier le langage standardisé de transformation de modèles QVT (*Query/View/Transformation*).[07]

2. Groupe Obeo est un fournisseur leader d'outils ALM et aide les entreprises à industrialiser le cycle de vie de leurs systèmes informatiques, Spécialisée dans l'approche MDA. Il propose à ses clients des solutions pragmatiques pour industrialiser chacune des étapes du cycle de vie des applications: la génération de code, la migration, la refactorisation, ...[24]

Il est basé sur la plate-forme Eclipse, l'héritage des outils de soutien, Obeo construit avec des technologies différentes. Ils peuvent également intégrer facilement et rapidement de nouvelles langues et de leurs évolutions. [24]

Impliqué dans la communauté Open Source, Obeo a développé le cœur du projet Acceleo (générateur de code basé sur EMF). La société est également un Eclipse Foundation stratégique membres: Obeo est le principal responsable de l'EMF Compare projet (EMF comparaison des modèles) et industrialisation de la boîte à outils ATL. Obeo propose conseil et de formation pour aider les entreprises à améliorer l'utilisation de leurs outils. [24]

3. **L'Institut national de recherche en informatique et en automatique** (**Inria**) est un institut de recherche français en mathématiques et informatique. Il a le statut d'établissement public à caractère scientifique et technologique, créé le 3 janvier 1967 suite au lancement du Plan Calcul. [18]

Son ambition est de mettre en réseau les compétences et talents de l'ensemble du dispositif de recherche français et international, dans ses domaines de compétence. [18]

LINA est partenaire du centre de recherche ***INRIA*** Rennes Bretagne Atlantique. Le centre compte 28 équipes-projets dont les deux équipes-projets ATLAS commune au CNRS, l'Ecole des Mînes de Nantes et l'Université de Nantes et accueillies au sein du LINA. [18]

4. **Eclipse IDE** est un environnement de développement intégré libre (le terme *Eclipse* désigne également le projet correspondant, lancé par IBM) extensible, universel et polyvalent, permettant potentiellement de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. [40]

Figurant parmi les grandes réussites de l'Open Source, Eclipse est devenu un standard du marché des logiciels de développement, intégré par de grands éditeurs logiciels et sociétés de services. [40]

La spécificité d'IDE Eclipse vient du fait de son architecture totalement développée autour de la notion de plug-in, toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plug-in. [40]

5. **Topcased** est un projet atelier logiciel open source intégré à l'IDE Eclipse. Le projet a débuté en 2004. [41]

Topcased s'inscrit dans le processus MDE (Model Driven Engineering) qui consiste à baser le développement d'outils à partir de modèles. Il contient nativement de nombreux éditeurs graphiques (UML2, SYSML, AADL, SAM, ECORE) mais offre aussi la possibilité de générer son propre éditeur de méta-modèle. [41]

Topcased est modeleur pour les diagrammes UML qui permet la création et la manipulation des diagrammes UML en mode graphique et aussi avec le format XML. [41]

Bien que TOPCASED peut très bien être utilisé dans des contextes différents, comme le développement de systèmes d'informations, ou la réalisation des diagrammes UML et ce qui est important c'est que ces diagrammes sont conforme aux UML2.0 et compatible avec le framwork EMF d'Eclipse. [41]