UNIVERSITÉ DU QUÉBEC À MONTRÉAL

VERS UNE NOUVELLE APPROCHE DE LA MODERNISATION DES SYSTÈMES LÉGATAIRES À TRAVERS LA MIGRATION VERS UN ENVIRONNEMENT DIRIGÉ PAR LES MODÈLES

THÈSE
PRÉSENTÉE
COMME EXIGENCE PARTIELLE
DU DOCTORAT EN INFORMATIQUE

PAR
GINO CHÉNARD

NOVEMBRE 2013

REMERCIEMENTS

Je tiens à remercier et à témoigner ma reconnaissance à ceux qui ont permis de près ou de loin la réalisation de cette thèse.

Mes remerciements vont d'abord à Ismaïl Khriss qui m'a dirigé de façon soutenue et efficace tout au long de cette thèse.

Je remercie Aziz Salah dont la direction m'a permis de réaliser cette thèse à l'UQAM.

Je remercie les membres du jury pour leurs commentaires éclairés.

Je remercie mes amis montréalais qui m'ont fait apprécier ma ville d'études : Dolorès, Yoke, Stéphane, Marie-Émilie et Jean- François.

Je remercie ma tante Marie-Paule qui a été un soutien important.

Finalement et non les moindres, je tiens à remercier mes parents qui m'ont toujours aidé et soutenu lors de mes études.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ADM	Architecture-Driven Modernization
AOP	Programmation orientée aspect (Aspect Oriented Programming)
AST	Arbre syntaxique abstrait (Abstract Syntax Tree)
ASTM	Métamodèle d'arbre syntaxique abstrait (Abstract Syntax Tree Meta-model)
EJB	Enterprise Java Beans
IDM	Ingénierie dirigée par les modèles
LSI	Indexation sémantique latente (Latent Semantic Indexing)
MDA	Architecture dirigée par les modèles (Model Driven Architecture)
MDE	Ingénierie dirigée par les modèles (Model-Driven Engineering)
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PDM	Modèle de plate-forme (Platform Description Mode)
PIC	Concept indépendant de plate-forme (Platform Independent Concept)
PIM	Modèle indépendant de plate-forme (Platform Independent Model)
PSC	Concept spécifique à une plate-forme (Platform Specific Concept)
PSM	Modèle dépendant de plate-forme (Platform Specific Model)
QVT	Query/View/Transformation
SOA	Architecture orientée services (Service Oriented Architecture)
SQL	Langage SQL (Structured Query Language)
SVD	Décomposition en valeurs singulières (Singular Value Decomposition)
UML	Langage unifié de modélisation (Unified Modeling Language)

RÉSUMÉ

Les organisations sont fortement dépendantes de leurs logiciels dans l'exercice de leurs activités quotidiennes. Malheureusement, les changements répétés qui sont appliqués à ces systèmes rendent leur évolution difficile. Cette évolution peut être rendue nécessaire afin de maintenir le logiciel, de le remplacer ou de le moderniser. Dans le cas de systèmes légataires complexes et mal documentés, la modernisation est la seule solution réalisable afin d'atteindre les objectifs d'évolution, le but de la modernisation étant de faire évoluer un système lorsque les pratiques conventionnelles ne le permettent plus. Mais, il s'agit d'une tâche complexe. Notamment, la prévision des risques et des coûts est difficile.

Afin de faire face aux difficultés de la modernisation, l'OMG a créé l'initiative « Architecture-Driven Modernization » ADM qui propose entre autres de réaliser la modernisation par l'ingénierie dirigée par les modèles (IDM). Dans ce contexte, la modernisation d'un système légataire, non développé dans un environnement IDM, débute par sa migration vers ce type d'environnement. Ce qui pose la problématique de la découverte des modèles nécessaires à l'utilisation d'IDM représentant ce système. Une seconde problématique est que le processus IDM manque de précision au sujet des modèles à employer et de l'application des transformations pour passer d'un à l'autre.

Dans cette thèse, nous présentons une nouvelle approche de modernisation ADM afin d'utiliser l'IDM pour moderniser un système légataire non IDM. Nous y définissons les modèles nécessaires et les transformations à réaliser pour passer d'un à l'autre. La plate-forme d'implémentation y est représentée par deux modèles. Le premier est un profil UML décrivant ses concepts et le second est un ensemble de modèles de transformations paramétrés capturant son code d'infrastructure. Le modèle représentant les éléments du domaine du problème prend la forme d'un diagramme de classes UML. Aussi, nous proposons des algorithmes pour la découverte de ces modèles en analysant le code source du système légataire. Notre approche a été validée sur plusieurs systèmes écrits en Java et a donné de bons résultats pour les systèmes bien structurés avec un bon style de programmation.

INTRODUCTION

MOTIVATION

Les organisations sont fortement dépendantes de logiciels dans leurs activités quotidiennes. Malheureusement, les changements répétés qui sont appliqués à ces systèmes en rendent l'évolution difficile. Cette évolution peut être nécessaire afin de maintenir le logiciel, de le remplacer ou de le moderniser (Weiderman *et al.*, 1997).

On parle de système légataire (aussi nommés hérité ou patrimonial) dans le cas d'un où il est devenu obsolète sur le plan de l'architecture ou de la plate-forme et dont l'évolutibilité ne permet plus de répondre aux objectifs d'évolution (Alam et Padenga, 2010). Dans ces cas, la modernisation est la seule solution réalisable afin d'atteindre ces objectifs. En effet, le but de la modernisation est de faire évoluer un système lorsque les pratiques conventionnelles ne permettent plus l'atteinte du but recherché (Seacord, Plakosh et Lewis, 2003). Elle cherche donc à modifier ces systèmes afin d'améliorer leur maintenabilité.

Même si la modernisation semble une solution idéale, il s'agit d'une tâche complexe. Notamment, la prévision des risques et des coûts est difficile. Afin de faire face à ce problème, le consortium OMG (« Object Management Group ») a créé l'initiative ADM (« Architecture-Driven Modernization »). L'objectif principal de cette initiative est de fournir un ensemble de normes pour simplifier l'interopérabilité des outils de modernisation (OMG, 2006). ADM propose de réaliser la modernisation en utilisant l'ingénierie dirigée par les modèles (IDM). L'IDM est une approche de développement logiciel basée sur la représentation par des modèles des différents aspects du système à développer et sur l'utilisation de transformations de modèles incluant la génération du code source (Schmidt, 2006).

Dans le contexte d'ADM, la modernisation d'un système légataire développé dans un environnement non IDM débute par sa migration vers ce type d'environnement. Ce qui pose le défi de découvrir les modèles nécessaires à IDM le représentant.

La découverte de modèles à bas niveaux ne représente pas une difficulté majeure. Parmi ces modèles, on retrouve le modèle dépendant de plate-forme ou PSM (« Platform Specific Model »). Ce modèle représente fidèlement l'implémentation courante du système. Il contient à la fois les éléments de la plate-forme d'implémentation et du domaine métier. Mais la découverte de modèles à plus haut niveau d'abstraction est plus difficile à réaliser, que ce soit de façon automatique ou manuelle. Parmi ces modèles, on retrouve le modèle indépendant de plate-forme ou PIM (« Platform Independent Model »). Ce modèle représente le système d'un point de vue indépendant de toute plate-forme d'implémentation. Il contient seulement des éléments du domaine métier du système.

Une autre problématique avec ce type d'approche est que le processus IDM manque de précision. Notamment, la forme que doivent prendre ses modèles et la description et l'application des transformations pour passer d'un modèle à un autre. Ainsi, une difficulté importante est la définition d'un modèle de plate-forme ou PDM (« Platform Description Model »). Ce modèle à haut niveau d'abstraction définie à la fois les concepts d'une plate-forme d'implémentation et comment appliquer ceux-ci à un PIM afin de créer un PSM reflétant ce PIM et ce PDM. Un concept est la formulation d'une exigence système du point de vue du domaine de la plate-forme d'implémentation ou du domaine métier.

OBJECTIFS ET CONTRIBUTIONS

Le but de cette thèse est de proposer une approche de modernisation fondée sur ADM permettant la migration de systèmes légataires développés dans des environnements non IDM vers un environnement IDM.

Afin d'atteindre ce but, nous proposons d'abord un processus de modernisation ADM. Nous y définissons les différents modèles nécessaires et les transformations à réaliser pour passer d'un modèle à un autre. Ce processus permet de passer du code source d'un système légataire à un ensemble de modèles à plus haut niveau d'abstraction permettant d'utiliser l'IDM pour le moderniser.

Dans ce processus, afin de surmonter les problèmes que pose la représentation de la plate-forme d'implémentation et de son application à un modèle métier, nous proposons de la représenter par deux modèles. Le premier modèle représentant la plate-forme d'implémentation est un profil UML décrivant les concepts de la plate-forme sur laquelle le système a été implémenté. Ce premier modèle décrit aussi les contraintes entre les concepts en employant le langage d'expression de contraintes OCL (« Object Constraint Language ») (OMG, 2003). Le second modèle représentant la plate-forme d'implémentation est un ensemble de modèles de transformations paramétrés capturant le code d'infrastructure de la plate-forme d'implémentation. Ces modèles de transformation sont décrits à l'aide du langage QVT (« Query/View/Transformation ») qui est un langage standardisé pour exprimer des transformations de modèle à modèle (OMG, 2008).

Afin de faciliter l'application de ce processus, nous proposons des techniques pour découvrir automatiquement les trois principaux modèles employés à partir du code source de systèmes légataires. La tâche la plus complexe à effectuer pour découvrir ces modèles est l'identification des concepts présents dans le système légataire et distinguer ceux de la plate-forme d'implémentation de ceux du domaine métier. Pour réaliser cette tâche, nous nous basons sur l'hypothèse suivante. Le vocabulaire d'un logiciel peut se décomposer en deux vocabulaires : indépendant (métier) et dépendant de la plate-forme d'implémentation. Le second est constitué des mots-clefs du langage de programmation et de ceux de la plate-forme d'implémentation. Notre hypothèse est que le code lié à la plate-forme et son vocabulaire sont répétitifs ou semi-répétitifs. Avec cette particularité il est possible d'identifier ce vocabulaire et

aussi d'identifier le vocabulaire métier en filtrant du vocabulaire les mots-clefs et ceux de la plate-forme d'implémentation. En exploitant cette hypothèse, nous découvrons trois modèles.

D'abord, nous proposons une approche automatique permettant la découverte du profil UML de la plate-forme d'implémentation d'un système légataire. Ce travail a été présenté dans (Chénard, Khriss et Salah, 2010).

Ensuite, nous proposons une autre approche automatique qui est la découverte du modèle représentant les éléments du domaine du problème contenu dans le système légataire. Ce second modèle, sous forme de diagrammes de classes UML, donne une vue du système d'un point de vue indépendant de toute plate-forme d'implémentation. Cette approche est le prolongement du mémoire (Chénard, 2007) et a été présentée dans (Chénard, Khriss et Salah, 2007 ; Khriss et Chénard, 2008).

Finalement, nous proposons une approche automatique pour la découverte du troisième modèle qui contient un ensemble de modèles de transformations paramétrés capturant le code d'infrastructure de la plate-forme d'implémentation. Cette approche a été présentée dans (Chénard, Khriss et Salah, 2012 ; Khriss, Chénard et Salah, 2008).

ORGANISATION DE LA THÈSE

Cette thèse est organisée comme suit.

Au chapitre I, nous présentons la problématique à laquelle nous désirons nous attaquer, celle de la maintenance des systèmes légataires. Nous y présentons la pratique de la modernisation et en particulier celle de la modernisation avec l'IDM.

Au chapitre II, nous présentons plus en détail l'IDM et particulièrement l'approche MDA qui en est une forme courante. Nous présentons MDA par ses principaux modèles employés, les transformations et les normes qui y sont reliés. Ce chapitre se termine par une présentation de la modernisation dirigée par une architecture de

modèles ou ADM qui est basée sur MDA. Nous décrivons ADM en présentant les normes, les scénarios et son processus pour terminer par une discussion sur ses avantages.

Au chapitre III, nous faisons un état de l'art entourant les sujets et techniques développés dans cette thèse. Nous débutons le chapitre par notre cadre de classification de la rétro-ingénierie, avant de discuter de sujets qui sont reliés à notre approche soit : l'analyse des identificateurs, l'indexation sémantique latente, la détection de clones, l'analyse de flux de données, l'identification de concepts et les approches de migration.

Au chapitre IV, nous donnons un aperçu global de notre approche de modernisation. Nous présentons une vue globale de l'approche avant d'en préciser les différentes parties et de montrer un système jouet qui sera utilisé aux chapitres suivants pour présenter en détail les éléments de notre approche. Ensuite, ce chapitre fournit un ensemble de définitions utiles pour comprendre le reste de cette thèse. Le chapitre se termine par une présentation de nos processus de découvertes de modèles.

Au chapitre V, nous décrivons en détail notre phase de découverte du profil UML de la plate-forme d'implémentation en employant des règles et des exemples. Le chapitre se termine par la présentation de notre processus de validation, des résultats et des discussions sur ces résultats de validation.

Les chapitres VI et VII sont structurés comme le chapitre V, mais présentent respectivement la phase de découverte du modèle indépendant de plate-forme et la phase de découverte des modèles de transformation.

Nous terminons cette thèse par une conclusion générale où nous faisons une synthèse de nos recherches et décrivons ce qui sera à faire afin de poursuivre nos travaux.

Nous avons aussi inclus trois appendices. Le premier présente un système que nous employons pour aider à l'explication de notre approche. Le second appendice décrit les différents éléments de notre approche sous forme de pseudo-code. Le dernier

appendice fournit des indications au sujet de l'utilisation de trois techniques de rétro-ingénierie : l'indexation sémantique latente, la détection de clones et l'analyse de flux de données.

CHAPITRE I

PROBLÉMATIQUE

Le cycle de vie d'un logiciel ne se termine pas lorsqu'il est remis au client. Il doit généralement ensuite être modifié, et ce, souvent à répétition. Ces modifications sont notamment rendues nécessaires par la nécessité de corriger des problèmes, de l'adapter à des changements d'environnement ou pour adapter ses fonctionnalités.

Dans ce chapitre, nous voyons la problématique des systèmes légataires. Nous proposons ensuite la modernisation de logiciels comme solution afin d'y faire face. Nous terminons le chapitre en discutant de l'utilisation de l'ingénierie dirigée par les modèles dans le but d'accomplir cette modernisation.

1.1 MAINTENANCE ET SYSTÈME LÉGATAIRE

La maintenance est l'étape du cycle de vie d'un logiciel où celui-ci est modifié après sa livraison dans le but de corriger ou de prévenir des problèmes, d'améliorer certaines de ses caractéristiques ou de l'adapter aux changements de son environnement. Quatre types de maintenance se distinguent. D'abord, la maintenance correctrice qui a pour but de corriger des erreurs de conception ou de programmation. Ensuite, la maintenance adaptative qui a pour but l'adaptation à un nouvel environnement. Il y a aussi la maintenance évolutive qui a pour but de répondre à de nouvelles exigences. Finalement, il y a la maintenance préventive qui cherche à empêcher les problèmes avant leur manifestation (ISO/IEC, 1999).

Malheureusement, les changements répétés causés par la maintenance produisent souvent un code source dont l'évolution est problématique. C'est ce qui caractérise la problématique posée par les systèmes légataires (« legacy ») (Bisbal *et al.*, 1997). Ces

systèmes présentent de nombreuses difficultés de maintenance, dont la fragilité, le manque de flexibilité, la difficulté d'interfaçage (aussi appelée connectivité) et la non-extensibilité (Bisbal *et al.*, 1999).

La modernisation est une pratique ayant pour but de résoudre cette problématique (Seacord, Plakosh et Lewis, 2003). Elle peut prendre plusieurs formes, dont le redéveloppement, le rhabillage et la migration (aussi appelée transformation) (Bisbal *et al.*, 1999). Même si la modernisation apparait comme une solution idéale, elle représente une tâche compliquée. Notamment, la prévision des risques et des coûts est difficile à réaliser. La section suivante traite en détail de cette pratique.

1.2 LA MODERNISATION

Comme nous l'expliquons à la section précédente, la modernisation est une pratique ayant pour but de résoudre les problématiques de maintenance des systèmes légataires. Elle cherche à modifier ces systèmes afin d'améliorer leur maintenabilité, c'est-à-dire, faciliter leur maintenance. Une fois ce but atteint, le système présentera notamment un meilleur retour sur les investissements qu'il a nécessités, des coûts de maintenance et d'évolution réduits, un cycle de vie prolongé et un interfaçage et une intégration simplifiés (Kajko-Mattsson, Ta et Wilczek, 2007).

Nous présentons dans cette section la pratique de la modernisation des logiciels légataires. En particulier, la phase de compréhension qui est la première de toute approche de modernisation. Ensuite, nous discutons des objectifs que la modernisation peut espérer atteindre. Finalement, nous montrons quelles techniques de modernisation sont possibles.

1.2.1 Phase de compréhension

La modernisation requiert d'abord la compréhension du système légataire avant de pouvoir le moderniser (ADM Domain Task Force, 2003). Les techniques de rétro-ingénierie permettent de faciliter la tâche de compréhension (Chia-Chu et Bayrak,

2006). La rétro-ingénierie est, selon Chikofsky (Chikofsky, 1990), le processus d'analyse d'un logiciel ayant pour objectif d'identifier les composantes et les relations entre les composantes de ce système et d'en créer une représentation à un plus haut niveau d'abstraction. Nous présentons cette pratique plus en détail au début du chapitre III.

Le niveau de connaissances requis au sujet d'un système afin d'en réaliser la modernisation est déterminé par le type de celle-ci. En effet, il existe deux types de modernisation : boîte noire et boîte blanche (Weiderman *et al.*, 1997). Une modernisation de type boîte blanche exige une connaissance de la structure interne du système; alors qu'une modernisation de type boîte noire nécessite seulement de connaître ses interfaces externes (Comella-Dorda *et al.*, 2000a).

1.2.2 Objectifs de modernisation

Lorsque le système légataire est bien compris et cerné, il est important de déterminer quelle partie de celui-ci, il est souhaitable de moderniser. Nous pouvons distinguer trois cibles principales de modernisation : l'interface utilisateur, les données et les fonctionnalités (Comella-Dorda *et al.*, 2000b).

La modernisation de l'interface utilisateur peut prendre la forme du rhabillage (« wrapping ») d'une interface texte avec une autre qui sera graphique (Bodhuin, Guardabascio et Tortorella, 2003 ; De Lucia *et al.*, 2006). La modernisation des données, quant à elle, a pour but de modifier le modèle de données ou son accès, comme avec l'adoption de passerelles (Corrêa, Moacir et José, 2005 ; Law, Ip et Wei, 1998). Enfin, la modernisation de la logique d'un système peut prendre la forme du portage d'un système écrit dans un langage non orienté objet vers un langage orienté objet (Cimitile *et al.*, 1999 ; Jacobson et Lindstr, 1991). Il peut aussi prendre la forme de la transformation de son architecture en une architecture orientée composantes (Kim et Kim, 2004 ; Li *et al.*, 2000) ou simplement de la traduction de son code source dans un autre langage de programmation (Seacord, Plakosh et Lewis, 2003).

1.2.3 Techniques de modernisation

Une fois que la compréhension du système légataire est réalisée et que l'objectif de modernisation est fixé, il faut déterminer quel type de modernisation devra être appliqué au système légataire. Trois techniques principales de modernisation se distinguent : redéveloppement, rhabillage et migration (Bisbal *et al.*, 1999).

Le redéveloppement implique de réécrire à partir de zéro le système légataire. Cette approche présente de grands risques d'échec et des coûts élevés (Bisbal *et al.*, 1999). De plus, certaines entreprises remplacent ainsi un système légataire par un autre système qui le deviendra lui aussi rapidement à son tour (Seacord, Plakosh et Lewis, 2003).

Le rhabillage implique l'élaboration de composantes logicielles appelées « wrapper » permettant au système légataire d'être accessible par d'autres systèmes de l'entreprise. Il s'agit d'une solution temporaire n'améliorant pas les possibilités d'évolution. De plus, les coûts de maintenance demeurent élevés (Bisbal *et al.*, 1999).

La migration a pour but d'adapter le système légataire à une nouvelle plate-forme tout en conservant ses fonctionnalités et ses données sans avoir à le redévelopper. Elle vise à conserver autant que possible le système existant. Une fois la migration effectuée, le système aura acquis des qualités qui faciliteront sa maintenance. Malheureusement, le processus de migration est complexe et la prévision des risques et des coûts demeure un exercice difficile (Bisbal *et al.*, 1999)

1.3 MODERNISATION D'UN SYSTÈME LÉGATAIRE À TRAVERS LA MIGRATION VERS UN ENVIRONNEMENT IDM

Une voie intéressante afin de pratiquer la modernisation est d'employer les approches d'ingénierie dirigée par les modèles (IDM) ou MDE (« Model-Driven Engineering »). L'IDM est une approche de développement de logiciels basée sur la représentation par des modèles des différents aspects du système à développer et sur l'utilisation

d'outils pour faire des transformations de modèles incluant la génération du code source. (Schmidt, 2006). Nous expliquons plus en détail l'IDM au chapitre II.

1.3.1 Modernisation avec IDM

Afin qu'une approche IDM de modernisation soit réalisable, il faut d'abord obtenir les modèles représentant le système à moderniser. Dans ce contexte, moderniser un système consiste à le faire migrer d'un environnement non IDM à un environnement IDM.

À moins que le système soit déjà parfaitement documenté dans des modèles employables par l'IDM, il faut obtenir ces modèles, ce qui n'est pas trivial (Mansurov et Campara, 2005). Nous pouvons toujours imaginer analyser manuellement le système pour les extraire, mais dans ce cas nous serons plus face à une pratique de réingénierie et nous serons aussi face aux problèmes reliés au redéveloppement. Il est donc souhaitable de trouver une approche automatique (ou tout au moins semi-automatique) pour découvrir ces modèles. Nous savons que dans tous les cas, le code source d'un logiciel et ses données en sont la représentation la plus fiable. Il est donc logique que ce soit ces derniers qui doivent être analysés afin de découvrir un modèle fiable. L'analyse d'un code source afin d'en tirer de l'information, implique la pratique de la rétro-ingénierie. Malheureusement, à notre connaissance, il n'existe que peu d'approches, comme nous allons le voir dans le chapitre III, qui exploitent sérieusement les techniques de rétro-ingénierie dans le but de faire une migration réelle d'un système légataire vers un environnement IDM.

1.3.2 Architecture-Driven Modernization (ADM)

Afin de proposer une approche de modernisation d'ingénierie dirigée par les modèles, l'OMG a créé l'Architecture-Driven Modernization (ADM) (OMG, 2006) qui est basé sur leur approche MDA. ADM propose des normes et des scénarios. Les normes encadrent les métamodèles et des pratiques comme l'analyse, les métriques, la visualisation et les transformations. Plusieurs scénarios de modernisation sont

proposés afin de guider l'implémentation de ces normes (OMG, 2006). Nous traitons plus en détail d'ADM au chapitre II.

1.4 RÉINGÉNIERIE VERSUS MIGRATION

Chikofsky décrit la réingénierie de logiciels comme l'examen et la modification d'un système afin de la reconstituer sous une nouvelle forme. Ce qui à première vue peut paraître similaire à la migration. Mais comme le note Bisbal et al., la réingénierie se distingue par le sort réservé au système légataire. La migration cherche à conserver autant que possible le système légataire en le transformant. Alors que la réingénierie mène au redéveloppement complet du système.

1.5 SYNTHÈSE

Nous avons expliqué dans ce chapitre que :

- le code source d'un système se dégrade avec le temps et la maintenance en causant ce qu'on appelle les problèmes du code légataire;
- la pratique de la modernisation apporte des solutions au problème du code légataire, mais que son implémentation demeure difficile;
- ADM aide à la réalisation de la migration, mais que la découverte des modèles dont elle a besoin demeure problématique;
- la rétro-ingénierie apporte une solution à la découverte des modèles nécessaires à ADM, mais que des approches efficaces restent à développer dans ce contexte.

Nous présentons dans le chapitre suivant l'ingénierie dirigée par les modèles.

CHAPITRE II

L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES

L'ingénierie dirigée par les modèles (IDM) ou MDE (« Model Driven Engineering ») est une approche de développement où les principaux éléments sont des modèles représentant à différents niveaux d'abstraction des vues de différents aspects du logiciel. L'objectif de cette approche est de générer en tout ou en partie le code du système à partir de ces modèles. Ainsi, le processus de développement est basé sur des transformations successives de modèles menant au code source.

Cette approche présente trois principaux avantages. Tout d'abord, elle permet l'automatisation d'une partie du processus de développement grâce aux transformations. En outre, ces transformations peuvent être réutilisées sur les mêmes types de modèles. Ensuite, grâce à l'obligation des modèles à se conformer à des métamodèles, des tests de conformité peuvent être réalisés.

Dans ce chapitre, nous approfondissons ce nouveau paradigme, principalement en étudiant l'approche IDM proposée par l'OMG : MDA (Kleppe, Warmer et Bast, 2003). Nous étudions aussi l'approche IDM que propose l'OMG dans le cadre de la modernisation : ADM (OMG, 2006).

2.1 L'ARCHITECTURE DIRIGÉE PAR LES MODÈLES (MDA)

L'OMG propose l'architecture dirigée par les modèles ou MDA (« Model Driven Architecture ») qui peut être vue comme une variante particulière de l'IDM (Bézivin *et al.*, 2004). Elle est basée sur plusieurs autres normes de l'OMG et sur un ensemble de modèles à différents niveaux d'abstraction qui sont transformés d'un niveau à l'autre.

2.1.1 Les modèles MDA

L'approche MDA est donc basée sur plusieurs modèles, dont voici les principaux :

2.1.1.1 Le modèle métier (CIM)

Un premier modèle MDA est le modèle métier ou CIM (« Computation Independent Model »). Ce modèle s'adresse principalement aux spécialistes du domaine métier du système et le décrit à l'aide d'un vocabulaire familier au domaine plutôt que d'utiliser le vocabulaire informatique. Le CIM se concentre sur les exigences du système ainsi que sur son environnement, mais ne donne pas de détails sur sa structure.

2.1.1.2 Le modèle indépendant de plate-forme (PIM)

Le modèle indépendant de plate-forme ou PIM (« Platform Independent Model ») décrit le système d'un point de vue indépendant d'une plate-forme. Une vue du PIM peut représenter les classes du système ainsi que leurs interrelations. Les classes ne doivent pas refléter la plate-forme de laquelle le PIM est indépendant. Le concept d'indépendance d'une plate-forme est donc relatif et il ne devient clair que si nous précisons de quelle plate-forme le PIM est indépendant.

Dans cette thèse, le PIM d'un système est un modèle le représentant de façon indépendante de sa plate-forme d'implémentation. Rappelons qu'une plate-forme d'implémentation est « un ensemble de sous-systèmes et de technologies qui fournit un ensemble cohérent de fonctionnalités à travers des interfaces et l'utilisation de patrons » (Miller et Mukerji, 2003). Dans ce cas, les classes du PIM représenteront principalement les classes métier du système ainsi que les relations entre elles.

2.1.1.3 Le modèle dépendant de plate-forme (PSM)

Le modèle dépendant d'une plate-forme ou PSM (« Platform Specific Model ») représente le PIM d'un point de vue spécifique à une plate-forme. Un PIM peut être

employé afin d'implémenter le système sur plus d'une plate-forme, donc, il peut donner lieu à plusieurs PSMs.

2.1.1.4 Le modèle de description de plate-forme (PDM)

Le modèle de description plate-forme[1] ou PDM (« Platform Description Model ») représente la définition d'une plate-forme. Selon OMG, un PDM « fournit un ensemble de concepts techniques, représentant les différents types de composantes d'une plate-forme et les services fournis par cette plate-forme. Il fournit également, pour une utilisation dans des PSMs, les concepts représentant les différents types d'éléments à être utilisés en précisant l'utilisation de la plate-forme par une application (Miller et Mukerji, 2003). » Par exemple, le modèle de la plate-forme EJB fournit des concepts tels que `Bean` et `Home`.

L'OMG recommande l'utilisation de profils UML pour exprimer un PDM. Les profils UML fournissent un mécanisme d'extension générique (stéréotypes, définitions de balises et de contraintes) pour la personnalisation de modèles UML dans des domaines et plates-formes particuliers. Les profils UML sont exprimés sous forme de diagrammes de classes UML. Comme l'ont notés Jouault et al, deux difficultés sont associées à la définition d'un PDM : l'absence d'une définition claire de ce qu'est un concept de plate-forme et comment composer un PIM avec un PDM afin d'obtenir un PSM (Jouault, Bézivin et Barbero, 2009). Cette seconde difficulté vient du fait que l'utilisation de PDMs exprimés uniquement dans un profil UML n'est pas suffisante pour effectuer les transformations.

2.1.2 Les transformations

Comme MDA est une approche IDM, elle emploie des transformations de modèles à modèles. Ces transformations définissent comment certains éléments contenus dans un modèle doivent être transformés pour s'intégrer à un autre modèle. Un exemple

[1] OMG utilise le terme modèle de plate-forme (« platform model »).

d'une telle transformation est la transformation d'un PIM afin d'obtenir un PSM. Une transformation peut être réalisée manuellement, avec un profil ou avec des modèles de transformations. Ces transformations peuvent être plus ou moins automatisées.

Les transformations peuvent être verticales, c'est-à-dire que le modèle en entrée est à un niveau différent d'abstraction du modèle en sortie. S'il est à un plus grand niveau d'abstraction, il s'agit d'une « abstraction » et dans le cas inverse, il s'agit de « raffinement ». Les transformations peuvent aussi être horizontales, c'est-à-dire que les modèles en entrée et en sortie sont au même niveau d'abstraction. On peut imaginer de telles transformations dans le cas de réusinage ou d'optimisation.

Dans les outils MDA, on peut retrouver des transformations de style « boîte noire ». Ce sont des transformations prédéfinies, généralement pour une technologie particulière. Par exemple, elles peuvent servir à créer des squelettes de classes Java implémentant l'architecture J2EE pour les classes d'un PIM. Ces transformations sont faciles à utiliser, mais peu souples. Par contre, l'on retrouve aussi des transformations de style « boîte blanche ». Ce sont des transformations que l'utilisateur peut définir(ou modifier). Par exemple, au lieu d'être limité à une plate-forme précise, l'usager pourra en définir une variante ou une nouvelle. Il s'agit d'une solution plus souple, mais plus complexe à mettre en œuvre.

2.1.3 Les normes reliées à MDA

MDA emploie un ensemble de normes de l'OMG afin d'atteindre ses objectifs. D'abord, avec MDA, les modèles sont généralement représentés en UML. Ensuite, MDA emploie OCL qui est un langage permettant d'exprimer des contraintes sur les modèles UML. Les expressions écrites en OCL permettent d'ajouter des informations aux modèles. Souvent, ces informations ne peuvent pas être exprimées dans un diagramme.

La norme MOF est centrale dans MDA (Miller et Mukerji, 2003). En effet, elle définit un langage permettant d'exprimer des métamodèles. UML, par exemple à

l'instar d'autres langages, est défini à l'aide de MOF. MDA préconise d'utiliser QVT comme langage pour définir les transformations entre les différents modèles (OMG, 2008). QVT dépend d'OCL et MOF. Ainsi, il emploie les caractéristiques d'OCL qu'il étend pour la programmation impérative. De même, les modèles à transformer par QVT doivent être définis avec MOF.

Comme son nom l'indique (« Query/View/Transformation »), QVT possède trois parties : les requêtes sur des modèles, la création de vues ou de modèles et la définition de transformations entre ces modèles (OMG, 2008). Une requête sélectionne des éléments spécifiques d'un modèle, une vue est un modèle dérivé d'un autre modèle et une transformation est la spécification d'un mécanisme de conversions des éléments d'un modèle dans les éléments d'un autre modèle. Les deux modèles sont soit définis par un même métamodèle ou par des métamodèles différents.

2.2 MODERNISATION DIRIGÉE PAR UNE ARCHITECTURE DE MODÈLES (ADM)

Afin de faire face à la problématique de la modernisation, l'OMG a lancé une initiative nommée « Architecture-Driven Modernization » (ADM) (OMG, 2006). Celle-ci cherche à faire bénéficier des avantages de MDA lors de la modernisation de systèmes légataires. Dans un contexte non IDM, la modernisation d'un système demande donc de migrer le système légataire d'un environnement non IDM vers un environnement IDM.

ADM poursuit trois buts principaux : la revitalisation des systèmes existants, l'amélioration de leur agilité et la promotion de normes et de pratiques menant à une modernisation réussie. Afin d'atteindre ces objectifs, ADM propose un ensemble de normes et de scénarios de modernisation.

2.2.1 Les normes

Les normes préconisées par ADM incluent le métamodèle de découverte de connaissances ou KDM (« Knowledge Discovery Meta-Model »). Ce modèle a pour but de faciliter la compréhension du système analysé à un haut niveau d'abstraction. Il peut afficher l'information sur le comportement, la structure des données de ce système. Il sert également de format d'échange de données entre les outils ayant à manipuler les modèles afin de faciliter leur interopérabilité. KDM est basé sur un autre métamodèle : le métamodèle d'arbre syntaxique abstrait ou ASTM (« Abstract Syntax Tree Meta-model »). Celui-ci donne de l'information à un bas niveau d'abstraction du système analysé. Les normes ciblent aussi, entre autres, l'analyse, les métriques, la visualisation, le réusinage (« refactoring »), le mappage (« mapping ») et les transformations (OMG, 2006).

2.2.2 Les scénarios

Même en maîtrisant toutes les normes d'ADM, il peut être difficile de déterminer comment les mettre en œuvre lors de la modernisation (Koskinen *et al.*, 2005). Afin d'aider l'utilisation de ces normes, l'ADM propose plusieurs scénarios reliés à la modernisation de systèmes logiciels. Ces scénarios aident à envisager des applications potentielles d'ADM. Ils aident également à déterminer les tâches, les coûts et à contrôler le projet de modernisation. Ils visent entre autres les activités de modernisation suivantes : amélioration, traduction du code d'un langage à un autre, intégration, consolidation et transformations MDA.

2.2.3 Le processus de modernisation d'un système existant avec ADM

Le processus de modernisation d'un système légataire avec d'ADM est telle qu'exprimé dans la figure 2.1, la migration d'un système dans un environnement non IDM vers un environnement IDM. Ce processus comprend deux étapes. La première étape utilise la rétro-ingénierie pour découvrir les modèles représentant les systèmes légataires. La seconde étape du processus de modernisation utilise l'ingénierie en

avant («forward engineering») afin d'obtenir le code source du nouveau système à l'aide de transformations. Ainsi, MDA est une approche descendante (« top-down ») et ADM est ascendante (« bottom-up ») (via la rétro-ingénierie) et ensuite descendante (avec MDA).

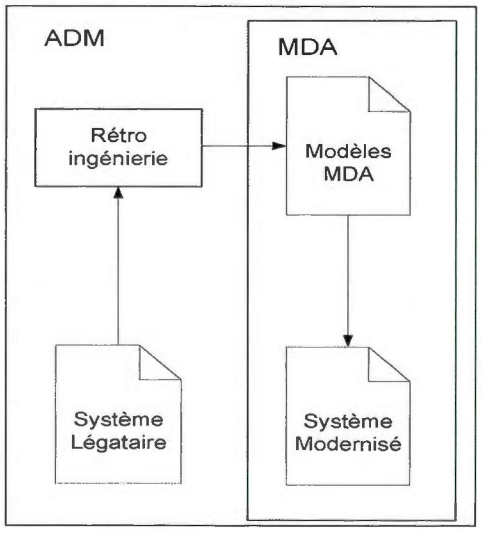

Figure 2.1 Le processus de modernisation d'un système existant avec ADM

2.2.4 Avantages d'ADM

Le paradigme ADM présente plusieurs avantages dans le cadre de la modernisation d'un système légataire. D'abord, il emploie des modèles à un haut niveau d'abstraction qui favorise la compréhension des systèmes. La séparation de la description d'un système des détails de son implémentation fournit une portabilité nécessaire à la migration. Une fois, la migration réalisée, les modèles à haut niveau et la génération automatique (ou semi-automatique) d'un système opérationnel permettront l'atteinte des objectifs de modernisation. Ils faciliteront la maintenance en permettant de modifier directement les modèles. L'interopérabilité est également un aspect principal du paradigme.

2.3 SYNTHÈSE

Nous avons montré dans ce chapitre que :

- l'ADM fournit une approche de modernisation qui aide à répondre aux problématiques de la modernisation en lui faisant bénéficier des avantages de l'ingénierie dirigée par les modèles;
- il reste à trouver des pratiques pour la mettre en œuvre notamment avec des techniques de rétro-ingénierie permettant d'extraire des modèles représentant le code légataire.

Nous présentons dans le chapitre suivant l'état de l'art des pratiques qui lui sont reliées et qui sont reliées à l'approche que nous présentons dans cette thèse.

CHAPITRE III

ÉTAT DE L'ART

Dans les chapitres précédents, nous démontrons l'intérêt de la migration afin d'affronter la problématique posée par l'évolution d'un système légataire. Nous montrons aussi l'importance qu'ont les approches de rétro-ingénierie dans son accomplissement.

Dans ce chapitre, nous présentons d'abord la pratique de la rétro-ingénierie via notre cadre de classification dans lequel nous intégrons les pratiques de modernisation. Nous consacrons ensuite le reste de ce chapitre à différentes approches de rétro-ingénierie pouvant favoriser la migration.

3.1 CADRE DE CLASSIFICATION DE LA RÉTRO-INGÉNIERIE

Comme la rétro-ingénierie représente une part importante de l'approche présentée dans cette thèse, nous allons d'abord nous attarder sur cette pratique. Afin de démontrer la diversité des pratiques de rétro-ingénierie, nous utilisons le cadre de classification présenté à la figure 3.1 (ce cadre est une extension du travail fait dans (Chénard, 2007) afin d'y intégrer la modernisation et ses trois sous-objectifs). Celui-ci permet de constater que la rétro-ingénierie est une pratique très vaste. Nous l'avons divisé en quatre grandes dimensions. La première dimension représente les données utilisées comme source d'information primaire. Ces données peuvent prendre plusieurs formes comme : le code source, les bases de connaissances, les traces d'exécution, l'exécutable, la documentation ou les connaissances des différentes personnes intervenantes dans le cycle de vie du logiciel. Il est souvent considéré que la donnée la plus fiable et la plus disponible est le code source.

La deuxième dimension représente le fait que les données ont souvent à être manipulées afin d'être utiles. Ces manipulations sont ici catégorisées selon deux critères. Le type de traitement employé indique l'emploi ou non de méthodes formelles. Les approches informelles sont fondées soit sur la recherche d'éléments dont on suppose la présence (approche descendante ou « top-down »), soit sur l'analyse des données afin de déterminer ce qui est présent (approche ascendante ou « bottom-up »). Les approches formelles cherchent à effectuer une simple traduction ou en plus une abstraction via une transformation.

Les types de traitements vus au point précédent peuvent être implémentés selon différentes techniques. Par exemple : association de concepts, traitement du langage naturel, ou analyse de flux de contrôle ou de données. Chacune de ces techniques nécessite l'un ou les deux types d'analyse : statique et dynamique. L'analyse statique implique en général l'analyse des informations obtenues directement en examinant le code source du système visé. L'analyse dynamique quant à elle, examine lors d'exécutions le comportement du système. Le plus souvent, ce sont les traces d'exécutions qui sont analysées. Notons que l'analyse dynamique est confrontée entre autres à la taille élevée des traces d'exécution, le problème d'échantillonnage et la représentativité des données. C'est pourquoi toute approche permettant de réduire cette taille est la bienvenue.

La troisième dimension montre qu'afin d'être traitées, les données brutes doivent tenir en mémoire et de préférence sous une forme facile à traiter. Parmi les langages intermédiaires utilisés pour représenter ces informations, nous retrouvons ceux qui conservent une information brute non retravaillée comme les arbres syntaxiques abstraits. Il y a aussi des représentations d'informations plus ciblées et parfois déjà retravaillées comme un graphe de flux de contrôle ou un graphe des appels. L'information sur les logiciels peut être aussi emmagasinée sous forme architecturale comme avec un diagramme de composantes UML.

La quatrième dimension indique que le but de la rétro-ingénierie est en général de fournir une abstraction d'un système à un niveau plus élevé d'abstraction, mais aussi parfois d'effectuer une transformation du logiciel comme lors de la modernisation. L'information affichée à l'utilisateur par un outil de rétro-ingénierie peut prendre plusieurs formes incluant l'architecture, les clones, les patrons de conception, les exigences et le tranchage (« slicing »). L'information générée lors d'un processus de rétro-ingénierie peut être affichée soit sous forme graphique, soit sous forme textuelle.

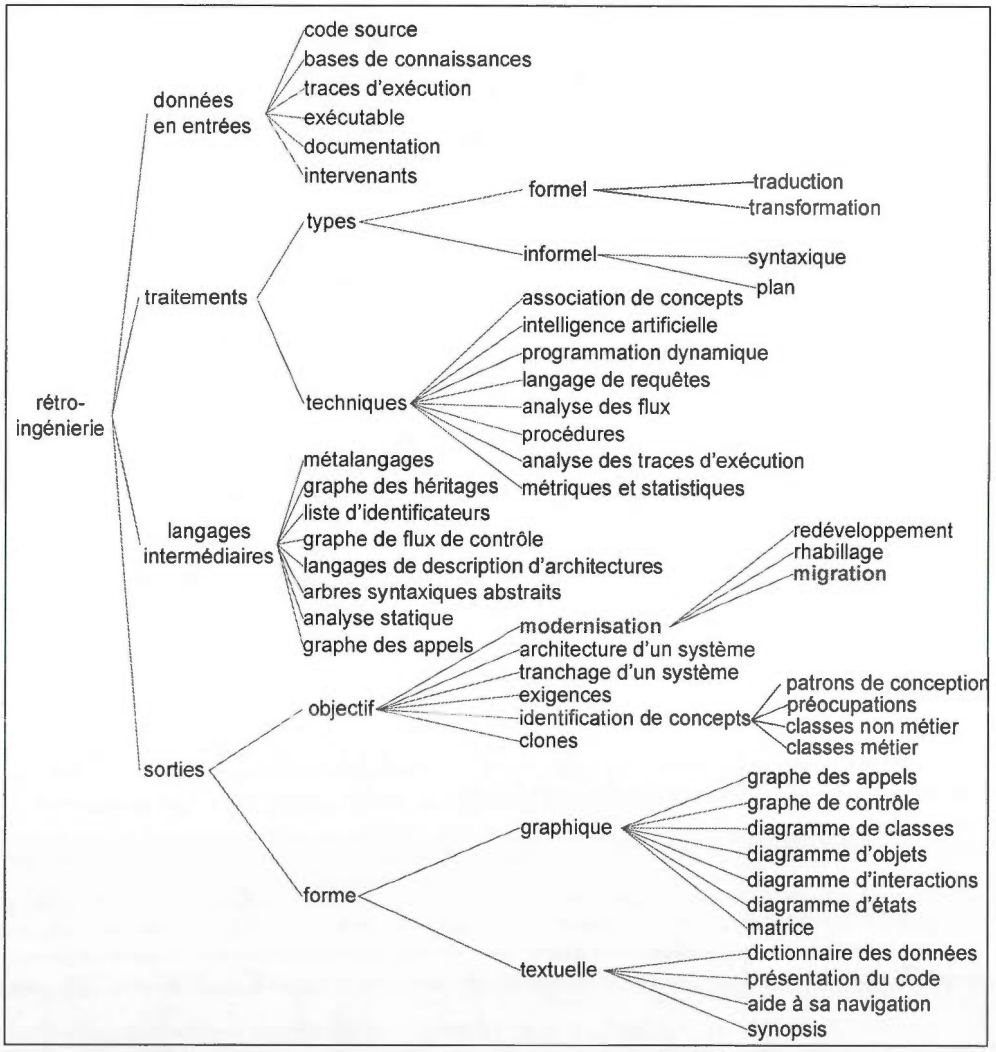

Figure 3.1 Cadre de classification de la rétro-ingénierie

En présentant la figure 3.1, nous avons exploré le domaine de la rétro-ingénierie. Il est important, car nos travaux emploient certaines de ses techniques. Celles-ci, nous permettent de détecter les concepts présents dans des systèmes légataires et les classifier.

Nous présentons donc ces techniques dans la suite de ce chapitre ainsi que des travaux reliés. Nous terminons le chapitre par des approches de migration s'approchant de la nôtre permettant de mieux cerner le domaine dans lequel elle s'insère.

3.2 L'ANALYSE DES IDENTIFICATEURS

Dans nos travaux précédents (Chénard, Khriss et Salah, 2007 ; Khriss et Chénard, 2008), nous avons démontré que l'analyse des identificateurs présents dans le code source d'un logiciel permet d'extraire l'architecture et de distinguer les classes métiers. Nous avons aussi conclu qu'en employant des techniques de traitement du langage naturel, il serait possible d'améliorer les résultats obtenus. Une discussion détaillée sur les différentes avenues d'améliorations est présentée dans (Chénard, 2007). Certains travaux de rétro-ingénierie, comme nous allons voir ci-dessous, ont aussi discuté de cet intérêt.

Anquetil et Lethbridge (Anquetil et Lethbridge, 1999) ont développé une approche pour partitionner un logiciel en sous-systèmes. Pour atteindre leur but, ils se basent sur le nom des différents fichiers contenant le code source du logiciel, la principale problématique qu'ils ont soulevée est que souvent ces noms contiennent des abréviations. Ils ont donc employé pour trouver la signification des abréviations : les commentaires, des dictionnaires, différentes métriques sur le nom des identificateurs et le nom des attributs et opérations dans le fichier.

Shepherd *et al.* (Shepherd *et al.*, 2007) présentent un outil qui aide un usager à découvrir des préoccupations transverses dans un code source. L'usager doit effectuer

une requête représentant la préoccupation recherchée. La requête est alors enrichie par l'outil selon sa connaissance du langage naturel ainsi que du système analysé. Par exemple, il proposera des synonymes se trouvant dans le code ou une autre forme d'un verbe qui y est.

Fry *et al.* (Fry *et al.*, 2008) présentent une approche où ils analysent la signature des opérations d'un système dans le but d'identifier une paire verbe-objet représentant le comportement de l'opération. Ils emploient donc le verbe éventuellement trouvé dans le nom de l'opération et selon son type et sa position, ils énoncent des règles pour détecter son objet.

Pollock *et al.* (Pollock *et al.*, 2007) utilisent le langage naturel pour la détection de préoccupations (reliés à des verbes d'action) et d'aspects (en cherchant des noms opposés comme « open » et « close »).

Caprile et Tonella (Caprile et Tonella, 1999) s'intéressent aux identificateurs contenus dans un système. Ils les analysent d'une manière lexicale, syntaxique et grammaticale. Ils proposent d'utiliser les identificateurs afin d'identifier une grammaire standard représentant les identificateurs dans un logiciel et pour adapter ceux ne la respectant pas. Ils suggèrent que l'approche peut aussi servir à enlever les synonymes dans les noms des identificateurs.

3.3 L'INDEXATION SÉMANTIQUE LATENTE (LSI)

Comme nous l'énonçons à la section précédente, nous avons conclu dans des travaux précédents que les techniques de traitement du langage naturel présentent une voie intéressante pour nos travaux de rétro-ingénierie. L'indexation sémantique latente ou LSI (« Latent semantic indexing ») est l'une de ces techniques que nous avons explorées. LSI est utilisée pour déterminer les similarités entre un ensemble de documents textuels (Deerwester *et al.*, 1990). Un intérêt de cette approche est d'aider à traiter les problèmes de synonymie et de polysémie auxquels nous sommes

confrontés dans notre approche. LSI a été utilisé dans plusieurs approches de génie logiciel nécessitant une analyse textuelle telle que la découverte de traçabilité entre artefacts logiciels, des concepts ou des clones.

Une analyse LSI débute par la création d'une matrice de n par m éléments dans laquelle chaque document (parmi m documents) est représenté par une colonne et chaque terme (parmi n termes) présent dans les documents par une ligne de la matrice. Les valeurs de la matrice sont la fréquence où un terme se retrouve dans un document particulier. Pour régler des problèmes de synonymie et de polysémie, LSI utilise une technique appelée décomposition en valeurs singulières ou SVD qui conserve les k dimensions plus importantes de la matrice originale en créant une matrice tronquée. Toutefois, déterminer la valeur de k n'est pas une tâche aisée. Donner une valeur trop grande à k laissera trop de bruit dans la matrice tronquée. Mais lui donner une valeur trop faible ignorera des dimensions importantes. Plusieurs travaux utilisent une valeur fixe pour k, alors que d'autres utilisent des formules mathématiques pour calculer la valeur de k, comme: $k = (m \times n)^{0,2}$ ou $k = \lfloor \sqrt{m} \rfloor$.

La similitude entre les documents est généralement mesurée par le cosinus entre leurs vecteurs correspondants. Cette mesure donne des valeurs entre -1 et 1. Les valeurs négatives indiquent des documents non liés tandis que les valeurs près du seuil de 1 indiquent des documents similaires. Déterminer ce seuil est aussi problématique que la valeur idéale et dépend du corpus étudié.

Van Der Spek et al. (Van Der Spek, Klusener et Van de Laar, 2008) proposent une approche basée sur LSI afin de découvrir les concepts architecturaux en regroupant les noms de variables en fonction de leur similitude. Les opérations des classes sont utilisées comme contextes dans une matrice LSI. Chaque groupe représente un concept architectural.

Boer et Vliet (Boer et Vliet, 2008) présentent une preuve de concept d'une approche similaire à la découverte de concepts architecturaux. Elle est également basée sur LSI, mais nécessite une intervention manuelle au cours du processus de découverte.

Kuhn et al. (Kuhn, Ducasse et Girba, 2007) présentent une approche basée sur LSI pour regrouper les artefacts d'un code source (par exemple les paquets, des classes ou des opérations) qui utilisent un vocabulaire similaire. Ces groupes sont appelés groupes sémantiques et sont interprétés comme des sujets linguistiques.

Marcus et al. (Marcus *et al.*, 2004) abordent le problème de la localisation de concept en utilisant LSI pour lier des concepts exprimés en langage naturel à des parties pertinentes du code source. Le système peut être utilisé en l'interrogeant directement ou en utilisant l'une des requêtes générées automatiquement.

3.4 LA DÉTECTION DE CLONES

Un des fondements de l'approche que nous présentons dans les prochains chapitres est que la découverte des éléments reliés à une plate-forme d'implémentation est rendue possible par le fait que ces éléments sont implémentés de manières répétitives ou semi-répétitives. Dans nos précédents travaux, nous nous basions principalement sur les identificateurs afin de trouver cette répétitivité, mais il est évident que cette répétitivité peut se refléter d'autre manière. Par exemple, elle peut se refléter dans du code source cloné. Nous avons donc exploité la détection de clones afin d'explorer cette voie.

La détection de clones est un domaine de recherche important dans le domaine de l'évolution des logiciels et a été utilisée avec divers objectifs. Son but est de trouver des fragments de code répétés dans le code source du système légataire. Quatre types de clones ont été documentés dans la littérature (Roy, Cordy et Koschke, 2009) :

- type-1: des fragments de code sont identiques sauf des variations dans les espaces blancs, la mise en page et les commentaires;

- type-2: des fragments de code sont syntaxiquement identiques à l'exception de variations dans les identificateurs, les littéraux, les types, les espaces blancs, la mise en page et les commentaires;
- type-3: des fragments répétés, mais avec des différences telles que : des énoncés modifiés, ajoutés ou supprimés, des variations dans les identificateurs, les littéraux, les types, les espaces blancs, la mise en page et les commentaires;
- type-4: des fragments effectuant la même tâche, mais implémentés avec des variations syntaxiques.

De nombreuses approches ont été proposées pour la détection de ces types de clones et analysent le code de façons différentes. Elles se basent pour la plupart sur l'une des techniques suivantes (Roy, Cordy et Koschke, 2009) :

- basé sur le texte du code source;
- basé sur les jetons;
- basés sur l'arbre syntaxique abstrait;
- basé sur des métriques;
- basé sur le graphe de dépendance.

Bruntink *et al.* (Bruntink *et al.*, 2004) cherchent les préoccupations par la détection de clones. L'idée est intéressante, car une préoccupation devrait être implémentée d'une façon semblable dans les endroits où elle se manifeste. Ils cherchent et regroupent les clones trouvés dans le code source. Ils explorent ensuite les groupes de clones en espérant trouver ceux implémentant une préoccupation qu'ils estiment présente.

Baxter *et al.* (Baxter *et al.*, 1998) présentent une approche où le code source est placé dans un AST où les sous-arbres sont classés dans une table de hachage. Une fonction de hachage plus ou moins floue (selon le degré de similarité désiré) permet de

regrouper les parties de codes qui se ressemblent et ainsi de découvrir des clones. Enfin, l'outil tente de regrouper les clones trouvés afin d'en découvrir de plus gros.

Ducasse, Rieger et Demeyer (Ducasse, Rieger et Demeyer, 1999) présentent une méthode visuelle de détection de clones indépendante du langage. Ils créent une matrice où chaque ligne de code est à la fois une ligne et une colonne et où les lignes de codes identiques sont affichées. Il suffit alors d'inspecter cette matrice pour détecter où sont présents les clones.

Jiang et Hassan (Jiang et Hassan, 2007) emploient des techniques de fouilles de données adaptées pour la détection des clones dans des logiciels de grande taille. La fouille est faite sur un ensemble de candidats produits par un autre outil à l'aide de techniques de mesure de similarités textuelles. Enfin, une abstraction est faite pour aider à les détecter manuellement.

Cornelissen et Moonen (Cornelissen et Moonen, 2007) appliquent des principes de détection de clones aux traces d'exécution dans le but d'en réduire la taille. Donc en détectant les parties de traces semblables et en les gérant correctement, plusieurs approches utilisant une analyse dynamique en profiteraient.

3.5 L'ANALYSE DES FLUX DE DONNÉES

Une des problématiques de notre approche est de trouver des liens entre les éléments du code source d'un système légataire. Par exemple, est-ce qu'un paramètre `x` d'une opération est lié à un attribut `y` d'une classe? Si leurs noms sont identiques, on peut supposer que c'est le cas, sinon une analyse de flux de données peut aider à répondre à cette question et c'est pourquoi notre approche exploite cette voie.

L'analyse de flux de données ou DFA (« Data Flow Analysis ») examine un système afin de déterminer où et comment les données sont définies et utilisées (Khedker, Sanyal et Karkare, 2009). L'objectif de cette analyse peut être l'optimisation du code pendant la compilation (Khedker, Sanyal et Karkare, 2009), l'identification du

concept (Kontogiannis *et al.*, 1994), l'élimination de code mort, la détermination des cas de test (Tsai, Stobart et Parrington, 2001), le tranchage (Davide *et al.*, 2005) ou la sécurité du système (Fetzer et Süßkraut, 2008).

Pour effectuer cette tâche, une analyse de flux de données traverse le graphe de flux de contrôle ou CFG (« Control Flow Graph ») du code source d'un système. Chaque nœud du CFG représente une instruction de code source. Il y a une arête entre deux nœuds, si le flux de contrôle peut passer du nœud source au nœud cible. Une analyse de flux de données associe généralement deux éléments à chaque nœud du CFG : le `inSet` et le `outSet`. Le `inSet` représente un ensemble d'états liés au système à l'entrée d'un nœud. Ces états peuvent représenter, par exemple, les variables initialisées à ce stade du CFG. Le `outSet` représente la même information, mais après l'exécution du nœud.

Ces ensembles sont initialisés et propagés dans le CFG le long des nœuds à partir du nœud de départ en suivant le flux de contrôle (pour une analyse « avant »). L'analyse (propagation) prend fin lorsqu'un point fixe est atteint, c'est-à-dire lorsque les parties itératives sont répétées jusqu'à ce que leurs `inSet` et `outSet` deviennent stables. La possibilité d'un point fixe est assurée par le fait que le domaine des éléments des ensembles est défini et que le contenu des `inSet` et `outSet` doivent être monotone après chaque itération.

Favre (Favre, 2008a) présente une approche ADM où le PSM contient en plus d'un diagramme de classes, des diagrammes d'états transitions décrivant le cycle de vie des objets sous forme d'états du système analysé qui sont extraient à l'aide d'une analyse de flux de données. Pour découvrir les diagrammes de classes, des techniques statiques standards sont employées, mais pour contrer la difficulté d'identifier l'agrégation et la composition, il propose l'utilisation d'analyse dynamique. Pour découvrir les diagrammes d'états transitions, une analyse de flux de données est employée.

L'approche de Kontogiannis et al. (Kontogiannis *et al.*, 1994) vise à détecter une forme de concept, dans ce cas-ci les opérations d'un code source d'un système relié à une variable. Il s'agit donc d'une forme de tranchage. L'analyse de flux de données dans leur cas cherche à déterminer à quelles opérations, une variable est reliée.

Davide et al. (Davide *et al.*, 2005) emploient quant à eux le DFA pour détecter dans un code source où se concrétise l'implémentation d'un aspect en cherchant dans le flux de donnée, un flux représentant l'aspect recherché.

3.6 L'IDENTIFICATION DE CONCEPTS

L'identification de concepts est un domaine de recherche actif depuis le début des années 1990 (Biggerstaff, Mitbander et Webster, 1993 ; Wilde et Gust, 1992). Un concept est défini comme la formulation d'une exigence système du domaine du problème ou du domaine de la solution (Marcus *et al.*, 2005). Une caractéristique (« feature ») est un concept décrivant une exigence fonctionnelle visible et accessible aux utilisateurs. Récemment, avec la popularité de la programmation orientée aspect ou AOP (« Aspect Oriented Programming »), les concepts ont pris des noms différents comme les préoccupations (« concerns ») et les aspects. Une préoccupation est une zone d'intérêt ou de concentration dans un système. Les aspects sont une implémentation possible d'une préoccupation. Les travaux sur le sujet utilisent l'une des deux approches :

- identifier les parties d'un code source qui implémentent un concept donné. Avec cet objectif, le problème est souvent nommé le problème d'affectation (ou localisation) de concepts. Il s'agit d'une approche de haut en bas;
- identifier les concepts mis en œuvre dans un code source. Il s'agit d'une approche de bas en haut.

Une grande partie de notre approche est dévouée à la découverte (identification) de concepts, car nous cherchons entre autres à trouver les concepts représentant la plate-forme d'implémentation et les classes du domaine métier.

3.6.1 Détection des classes non-métiers

Afin d'extraire un modèle indépendant d'une plate-forme d'implémentation, il est nécessaire de pouvoir séparer les classes représentant la logique métier des autres. Les classes utilitaires et celles reliées à l'architecture (ou infrastructure) font partie de la seconde catégorie. Il est donc important de trouver des techniques pour les détecter. Mais peu d'approches s'intéressent à ce problème et souvent quand le problème est mentionné, il n'est fait souvent référence qu'aux classes utilitaires. Il serait donc intéressant que de nouvelles approches s'attaquent à ce problème.

Hamou-Lhadj *et al.* (Hamou-Lhadj *et al.*, 2005) présentent une approche de détection des classes utilitaires qui se base sur des traces d'appels générées statiquement. Ils détectent les utilitaires d'après l'idée qu'un utilitaire est accédé de plusieurs endroits dans un logiciel. Ce qui se concrétise dans un graphe des dépendances entre les classes par le fait que les classes utilitaires vont être des nœuds « puits ». Ils ont créé une métrique basée sur ce fait pour mesurer le taux d'« utilité ». Cette approche nécessite une intervention humaine pour définir ce taux.

Greevy et Ducasse (Greevy et Ducasse, 2005) quant à eux parlent de classes d'infrastructure. Ils définissent les classes d'infrastructure comme étant celles qui participent à au moins la moitié des fonctionnalités offertes par un système. Les classes d'infrastructure sont extraites à partir de l'analyse des traces d'exécution des fonctionnalités du système. Une classe est détectée comme étant une classe d'infrastructure, si elle est présente dans au moins la moitié des ensembles de traces. Il est clair que cette approche pourrait identifier des classes métier comme appartenant à l'infrastructure. En outre, plusieurs classes d'infrastructure ne seront jamais identifiées en tant que tel.

3.6.2 Détection de relations entre les classes

Afin de créer des modèles précis autant spécifiques qu'indépendants de plate-forme, il est important d'identifier les différentes relations pouvant exister entre leurs éléments.

Keschenau (Keschenau, 2004) présente un outil d'analyse de code octet (« byte code ») Java pour en extraire des informations importantes sur les relations entre les classes comme l'héritage, la composition, la dépendance, les associations et leurs multiplicités et la dépendance. Ils emploient des règles telles que : « si une classe est créée dans le constructeur d'une autre inconditionnellement, il y a possibilité d'association 1 à 1 entre elles. Mais, s'il y a une condition, ce serait plutôt 0 à 1 ».

Gueheneuc (Gueheneuc, 2004) s'intéresse à l'identification des associations binaires entre les éléments du code source. Il définit d'abord clairement l'agrégation, l'association et la composition et ce que cela représente à la conception et à l'implémentation. Ensuite, en se basant sur quatre propriétés : l'exclusivité, le site d'invocation, la durée de vie et la multiplicité, il propose des règles les représentants. Ils proposent un algorithme pour la détection de celles-ci.

Sutton et Maletic (Sutton et Maletic, 2007) emploient un ensemble de règles de mappage pour extraire un diagramme de classes d'un code source C++ tout en identifiant plusieurs propriétés des associations. Ces règles s'appliquent à différents niveaux : lors de l'analyse syntaxique, de l'analyse sémantique, de l'analyse statique et à l'aide d'une base de connaissances.

Aucune de ces approches n'effectue une sélection parmi les classes à importer dans le modèle d'UML. Cependant, plusieurs font un bon travail dans la découverte de propriétés UML dont la correspondance n'est pas un à un.

3.6.3 Détection des préoccupations et de patrons de conception

Nous savons que la création d'un modèle indépendant de la plate-forme d'implémentation d'un système implique de pouvoir distinguer ce qui est relié à la logique métier de ce qui ne l'est pas. Les préoccupations (« concerns ») en font partie et devraient donc ne pas être représentées dans le PIM. Celles-ci se définissent comme des fonctionnalités qui, peu importe la décomposition qui est faite d'un système, se trouvent réparties dans plusieurs de ses composantes. La programmation par aspect aborde cette problématique en regroupant en modules des préoccupations communes. Un autre exemple de décisions de conception qui peut être utile à identifier est les patrons de conception.

Breu et Krinke (Breu et Krinke, 2003) analysent les traces d'exécution afin de trouver des aspects. Ils y cherchent des patrons d'appels récurrents à des opérations pour identifier celles réalisant les préoccupations transverses. Ces patrons sont de deux types principaux : quand l'appel d'une opération est toujours précédé de l'appel d'une autre et quand l'appel d'une opération est toujours à l'intérieur d'une autre opération précise.

Nous avons déjà décrit l'approche de Bruntink *et al.* (Bruntink *et al.*, 2004) qui cherche les préoccupations qui peuvent exister dans un système par la détection de clones (voir section 3.4).

Canfora, Cerulo et DiPenta (Canfora, Cerulo et DiPenta, 2006) se basent pour détecter les préoccupations sur l'idée que lors d'une mise à jour d'un logiciel, les lignes qui sont changées sont en général couplées logiquement. Ils analysent donc les différentes versions d'un système dans un dépôt afin d'y détecter les lignes changées conjointement pour trouver les préoccupations qu'ils espèrent identifier.

Marin, Moonen et Deursen (Marin, Moonen et Deursen, 2006) présentent trois techniques de détection d'aspects. D'abord, les opérations qui sont appelées souvent sont analysées pour identifier des préoccupations candidates. Ils identifient aussi des

préoccupations candidates en cherchant des groupes d'entités avec un nom similaire se trouvant dans plusieurs hiérarchies de classes. Finalement, ils cherchent les préoccupations en générant des traces d'exécutions pour une préoccupation qu'ils estiment présente. Ils emploient ensuite ces traces pour trouver les opérations réalisant le concept.

Plusieurs travaux de détection des patrons de conception sont présentés dans la littérature (Heuzeroth *et al.*, 2003 ; Huang *et al.*, 2005 ; Kramer et Prechelt, 1996 ; Philippow *et al.*, 2005 ; Pinali, 1999). La plupart des approches proposent un langage pour spécifier un patron de conception à rechercher. Certaines s'intéressent seulement à la spécification statique d'un patron, comme celle de Krämer et Prechelt (Kramer et Prechelt, 1996). D'autres s'intéressent aussi à la description dynamique des patrons de conception comme le travail d'Heuzeroth et al. (Heuzeroth *et al.*, 2003).

Notons que les travaux qui font de la détection de patrons de conception sont limités par le fait qu'il est difficile sinon impossible de prévoir toutes les implémentations possibles d'un patron de conception. Par exemple, Kim et Benner ont identifié que le patron `Observer` possède au moins douze implémentations différentes (Kim Jung et Benner Kevin, 1996). Notez que c'est pour cette raison que quelques travaux analysent plutôt la présence des micropatrons (Kim, Pan et Whitehead, 2006). Les micropatrons sont des patrons de conception à grains plus fins plus près du code source que les patrons. Les travaux qui identifient les aspects ou les préoccupations sont, quant à eux, très intéressants dans la mesure où ils ne les détectent pas à partir de leur définition, mais plutôt à partir d'une caractéristique assez générale comme le fait qu'ils sont répétitifs.

3.7 LES APPROCHES DE MIGRATION

Comme nous l'expliquons précédemment, à notre connaissance, très peu d'approches proposent la migration d'un environnement non IDM vers un environnement IDM à l'instar de l'approche que nous proposons.

Modisco (Bruneliere *et al.*, 2010) est un cadre d'applications (« framework ») extensible pour le développement orienté modèle ayant comme objectif de soutenir la modernisation des systèmes légataires dans le but de permettre la réutilisation de composantes entre les solutions de modernisation. Son architecture est organisée en trois couches : les cas d'utilisation, les technologies et les infrastructures. La première couche vise à fournir des outils pour des cas d'utilisation de modernisation spécifiques. La deuxième couche offre des composants spécifiques pour une technologie légataire. La dernière couche fournit des composants génériques indépendants de toute technologie légataire.

Boronat, Carsi et Ramos. (Boronat, Carsi et Ramos, 2005) proposent un mécanisme formel de transformation pour traduire une base de données relationnelle en un modèle UML. Le formalisme, appelé MOMENT (Model manageMENT), permet la représentation et la manipulation des modèles en employant la logique de la réécriture des termes.

Doyle et al. (Doyle *et al.*, 2006) présentent une approche pour la migration d'un système développé dans un langage de quatrième génération propriétaire vers IDM en fournissant des transformations de métamodèles en langage propriétaire vers UML. Étant ciblée sur un langage propriétaire, leur approche manque de généralisation.

Favre (Favre, 2008b) décrit un cadre IDM pour la modernisation de systèmes à l'aide d'un langage formel nommé NEREUS. Elle propose la traduction des métamodèles des langages de modélisation et des transformations en langage NEREUS. Ce dernier utilise des théories des méthodes formelles et formalise différents types de transformations de modèle à modèle telles que la décomposition et le réusinage. Elle présente surtout un cadre théorique qui demande à être validé plus à fond.

Reus, Geers et Deursen (Reus, Geers et Deursen, 2006) ont développé une migration vers un environnement IDM d'un système PL/SQL consistant en la découverte d'un modèle d'UML du système de son code source. Aucune abstraction n'est effectuée sur

le modèle extrait ce qui limite donc la praticabilité de l'approche. Mazón et Trujillo (Mazón et Trujillo, 2008) utilisent aussi des principes d'ADM dans la modernisation de bases de données.

Qiao *et al.* (Qiao *et al.*, 2003) discutent de la nécessité de porter les anciens systèmes vers IDM. L'approche est limitée à l'extension de leur travail sur la récupération de l'architecture en y ajoutant une nouvelle phase appelée intégration à l'IDM. Il s'agit simplement de la traduction du modèle architectural en langage SADL (« Simple Architecture Description Language ») vers UML.

Graaf, Weber et Deursen (Graaf, Weber et Deursen, 2008) ont développé une approche IDM qui applique une normalisation aux éléments du modèle source avant sa transformation. La normalisation est nécessaire, car leur modèle de plate-forme provient directement de la documentation et non du code source et il a besoin d'être modifié afin d'être transformable.

Sadovykh et al. (Sadovykh *et al.*, 2009) rapportent un cas réel de modernisation à l'aide d'ADM à partir d'un système légataire C++ traduit en Java. La création de leur PSM est basée sur des experts humains qui décident quelle partie du PSM est liée au domaine d'affaires et celle qui est liée à la plate-forme d'implémentation. Ils affirment qu'il est vraiment difficile d'automatiser cette distinction, car l'avis d'experts du domaine d'affaires et de la plate-forme d'implémentation est nécessaire.

Zhang et al. (Zhang *et al.*, 2009) présentent une stratégie générique de migration dirigée par les modèles pour des systèmes légataires vers SOA (« Service Oriented Architecture »). Leur cadre traite de questions, comme la façon d'extraire le modèle hérité de systèmes existants et la manière de développer une architecture adaptée pour SOA.

Fleurey et al. (Fleurey *et al.*, 2007) présentent une approche semi-automatique d'une migration dirigée par les modèles. Ils extraient automatiquement un PSM à partir d'un système légataire et utilisent des transformations pour générer un PIM et un PSM

représentant la nouvelle plate-forme d'implémentation. Ces transformations sont définies manuellement.

Chen et al. (Chen *et al.*, 2006) présentent une approche qui analyse le code source écrit en assembleur d'un système légataire afin d'extraire un ensemble de modèles représentant ses composantes logicielles et son architecture décrite dans un langage formel. Un PSM est d'abord créé à partir du code source de l'ancien système. Puis, des techniques de décomposition sont utilisées pour restructurer le PSM. Ensuite, les règles d'abstraction sont appliquées afin de créer un PIM à partir du PSM restructuré. Une limite de l'approche est que les règles d'abstraction doivent être définies manuellement.

3.8 SYNTHÈSE

Nous avons montré dans ce chapitre que :

- les approches de migration sont encore limitées pour le moment. Certaines ne font pas tout le travail de migration, d'autres ne s'intéressent qu'à la partie données, parfois il y a un manque de généralisation, le modèle PIM n'est pas toujours extrait du code source ou il manque d'abstraction;
- la rétro-ingénierie présente plusieurs avenues pour améliorer la qualité des modèles extraits du code source dont : la détection des classes non-métiers, la détection des préoccupations ou des aspects et le traitement des langues naturelles;
- une approche de migration combinant ces différentes avenues de rétro-ingénierie pourrait combler certaines faiblesses présentées par les approches existantes. C'est ce que nous proposons de développer au chapitre suivant.

Nous présentons dans le chapitre suivant un aperçu de notre approche.

CHAPITRE IV

APERÇU DE L'APPROCHE DE MODERNISATION

Dans les chapitres précédents, nous décrivons la problématique de la modernisation, l'IDM et l'état de l'art entourant ces sujets.

Dans ce chapitre ayant pour but de présenter notre approche, nous décrivons d'abord, notre vision de la modernisation dirigée par les modèles. Ensuite, nous présentons un système jouet qui sert d'exemple récurant dans le reste de cette thèse. Le chapitre se poursuit alors par la définition d'un ensemble de termes nécessaires à la présentation de notre approche et qui n'ont pas été présentés dans les chapitres précédents. Nous terminons enfin ce chapitre par une vue générale de la première partie de notre processus de modernisation qui est la découverte des modèles, qui sera vue plus en détail dans les trois chapitres suivants.

4.1 NOTRE PROCESSUS DE MODERNISATION DIRIGÉ PAR LES MODÈLES

La figure 4.1[2] décrit notre processus de modernisation ADM pour la migration d'un système légataire développé dans un environnement non IDM vers un environnement IDM. Ce processus est divisé en deux parties.

La première partie utilise la rétro-ingénierie afin de découvrir des modèles abstraits représentant le système légataire. Plus précisément, nous avons :

- une découverte texte à modèle (T2M) afin d'obtenir un PSM à partir du code source;

[2] L'idée d'illustrer le processus de modernisation avec un fer à cheval est inspirée de Seacord, R.C., Plakosh, D. etLewis, G.A. 2003. Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Addison Wesley Professional.

- une découverte modèle à modèle (M2M) afin d'obtenir un PDM à partir du PSM;
- une autre découverte M2M afin d'obtenir un PIM à partir du PSM et du PDM.

La seconde partie du processus de modernisation utilise l'ingénierie en avant (« forward engineering ») afin de créer le code source du nouveau système à l'aide de transformations M2M et modèle à texte (M2T).

Dans notre approche, le PSM est constitué d'un diagramme de classes représentant les éléments constituant le code source de l'application légataire. Le PIM prend lui aussi la forme d'un diagramme de classes, celui-ci représentant les classes du domaine du problème de l'application.

Quant au PDM, nous proposons de le décrire avec deux types de vues. La première vue est un profil UML de la plate-forme d'implémentation. La seconde vue est un ensemble de modèles de transformation. Ces modèles sont des modèles paramétrés qui capturent le code source de cette plate-forme. Ils sont exprimés dans le langage QVT (OMG, 2008). Le choix de QVT est motivé par le fait que MDA le recommande comme langage pour définir des transformations entre les modèles (OMG, 2008).

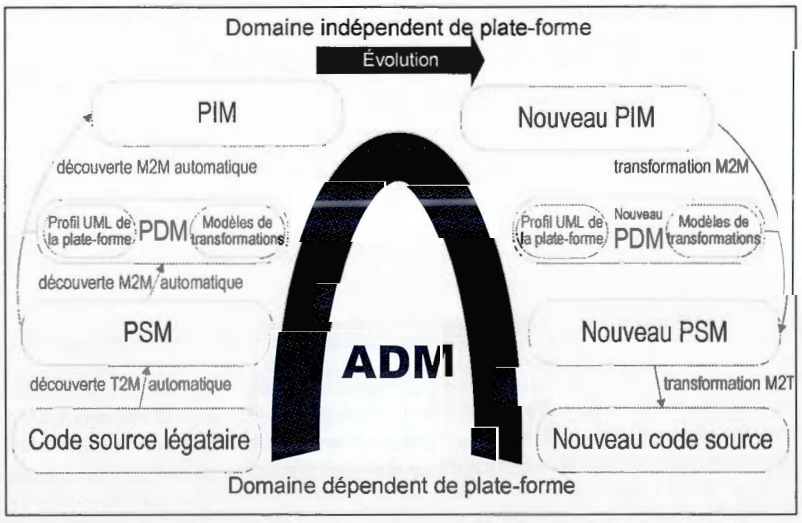

Figure 4.1 Notre processus de modernisation d'un système existant avec ADM

4.2 VUE GÉNÉRALE DE LA PARTIE DE DÉCOUVERTE DES MODÈLES

Dans cette section, nous donnons une vue générale de la partie de découverte des modèles de notre approche. Cette vue est précédée par la présentation d'un petit système jouet qui servira à expliquer notre approche dans le reste de cette thèse et d'un ensemble de définitions qui faciliteront la compréhension de nos descriptions de cette approche.

4.2.1 Le système jouet

Pour illustrer notre approche, nous utilisons un petit système bancaire contenant les 67 classificateurs dont les noms sont listés dans le tableau 4.1 et qui sont représentés dans le diagramme de classes de la figure 4.2. Nous sommes conscients qu'un tel diagramme imprimé à cette échelle est difficilement lisible, mais le but premier de sa présence est de démontrer la complexité que peut prendre un système, même de taille modeste.

Tableau 4.1 Classificateurs du système jouet

Account	AccountBean	AccountCMP
AccountData	AccountHome	AccountUtil
BankManager	BankManagerBean	BankManagerHome
BankManagerSession	BankManagerUtil	BankTransaction
BankTransactionBean	BankTransactionCMP	BankTransactionData
BankTransactionHome	BankTransactionUtil	CheckingAccount
CheckingAccountBean	CheckingAccountCMP	CheckingAccountData
CheckingAccountHome	CheckingAccountUtil	Customer
CustomerBean	CustomerCMP	CustomerData
CustomerHome	CustomerUtil	Deposit
DepositBean	DepositCMP	DepositData
DepositHome	DepositUtil	SavingAccount
SavingAccountBean	SavingAccountCMP	SavingAccountData
SavingAccountHome	SavingAccountUtil	Withdrawal
WithdrawalBean	WithdrawalCMP	WithdrawalData
WithdrawalHome	WithdrawalUtil	

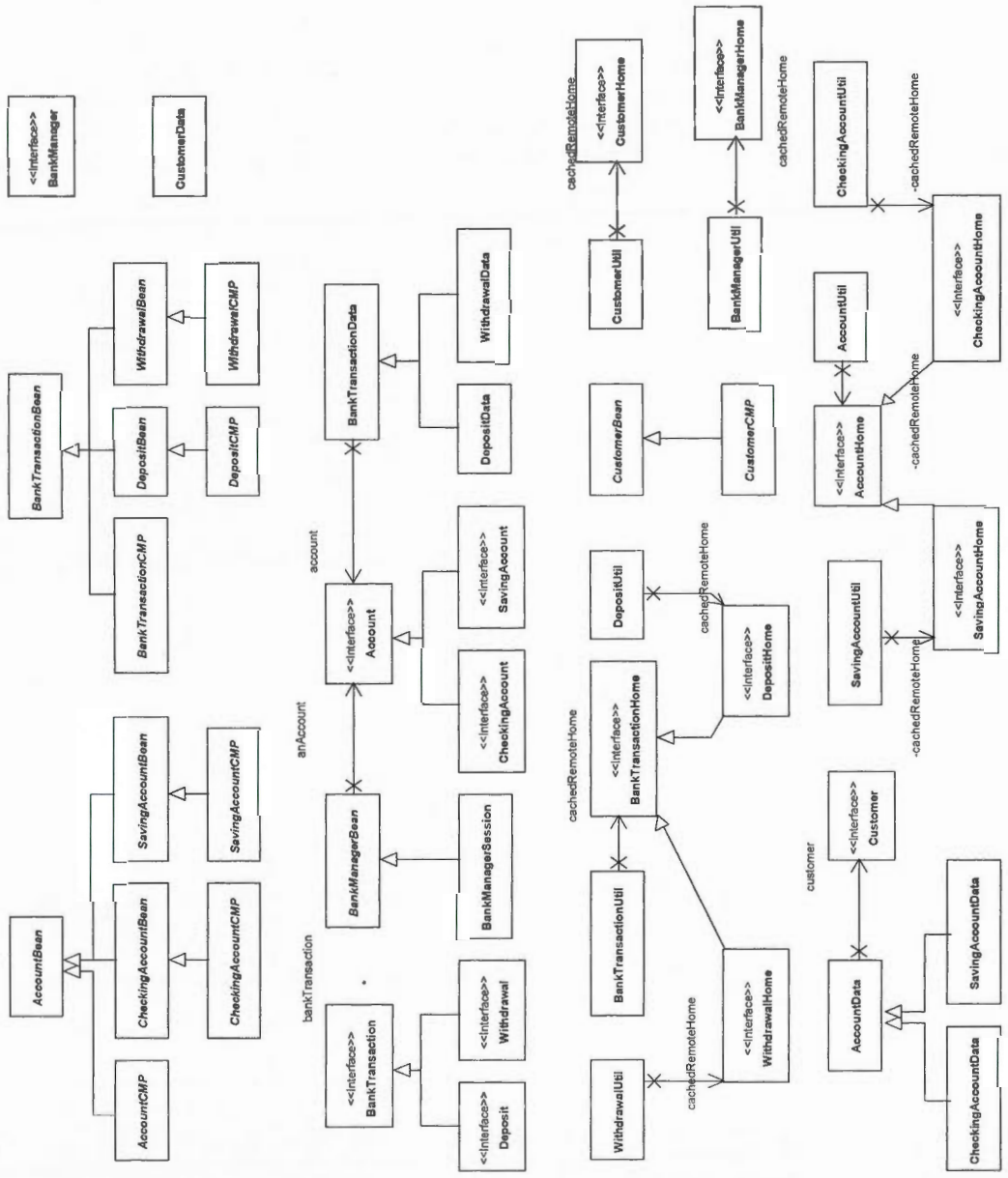

Figure 4.2 Le PSM du système jouet

4.2.2 Définitions

Cette section clarifie certains termes et concepts que nous employions dans la description de notre approche dans le reste de cette thèse. La figure 4.3 décrit le

métamodèle de notre approche et représente plusieurs de ses notions et les liens entre eux.

4.2.2.1 Vocabulaire d'un système légataire

Le vocabulaire d'un système légataire contient l'ensemble des mots contenus dans son code source (excluant les commentaires). Ce vocabulaire (dénoté par V) se compose de : V_{PI} (le vocabulaire représentant le domaine du problème, qui est donc indépendant de la plate-forme), de V_{PS} (le vocabulaire représentant la plate-forme d'implémentation qui est donc spécifique à la plate-forme) et de V_{PL} (le vocabulaire représentant le langage de programmation). Nous avons donc : $V = V_{PI} \cup V_{PS} \cup V_{PL}$.

4.2.2.2 Patron d'un identificateur

Un patron d'un identificateur est un mot (ou une combinaison) qui est une sous-chaîne contenue dans le nom de cet identificateur. Nous obtenons des patrons en divisant les identificateurs en nous basant sur des délimiteurs comme les lettres majuscules (lorsque les patrons des éléments sont séparés selon le principe « camel case ») ou les symboles de soulignement (lorsque les patrons des éléments sont séparés selon le principe « snake case »). Par exemple, l'identificateur `AccountEntity` contient trois patrons : `Account`, `Entity` et `AccountEntity`.

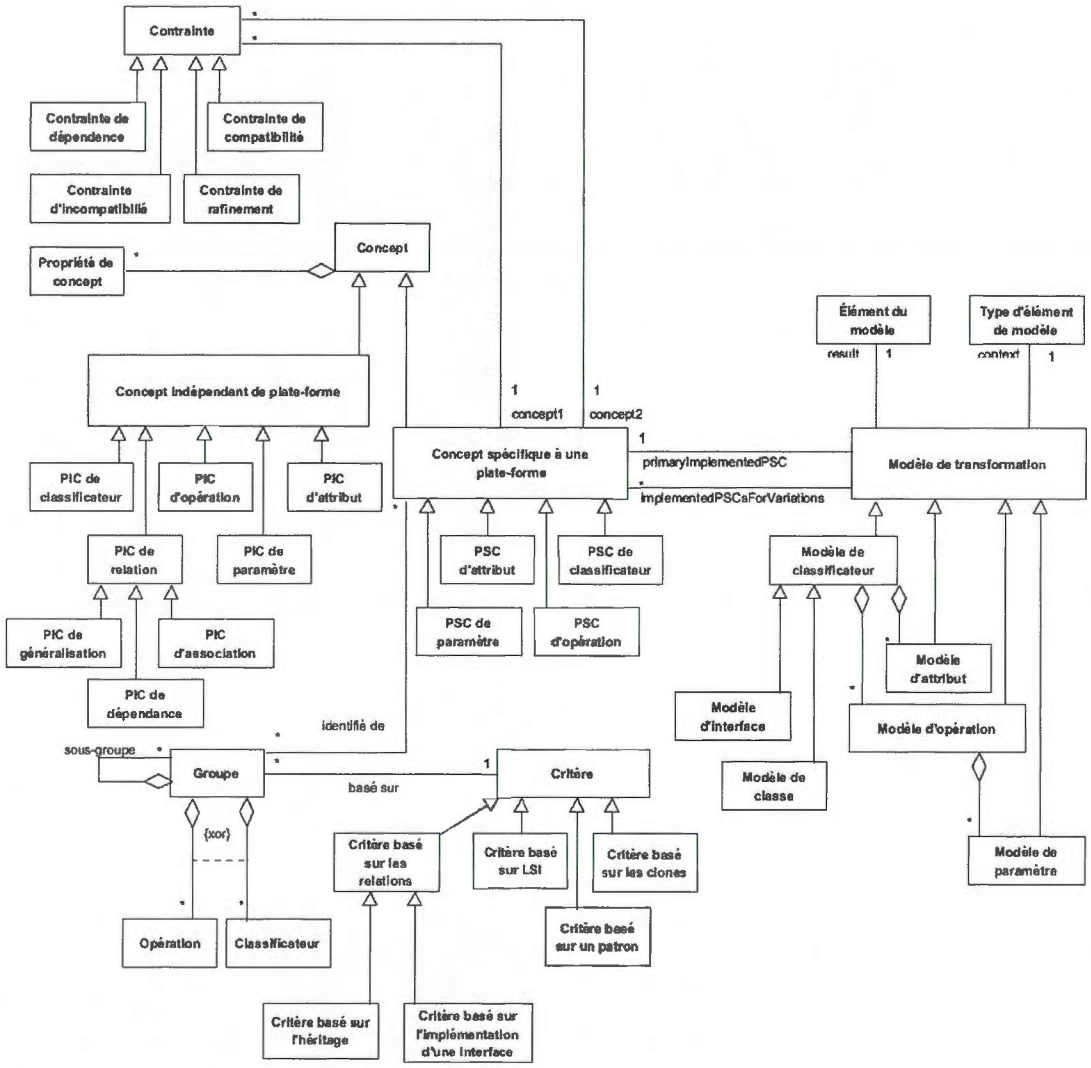

Figure 4.3 Métamodèle de notre approche

4.2.2.3 Dérivation

Une chaîne de caractère `S1` est dérivée d'une autre chaîne de caractère `S2` si `S1` peut être obtenue par l'application d'une ou de plusieurs fonctions sur `S2`. Ces fonctions peuvent être par exemple : `firstUpperCase`, `firstLowerCase`, `camelCase`, `snakeCase` et `abbreviation`.

Une variable `V1` peut être dérivée d'une autre variable `V2` sans que leurs noms comportent de similitude. Supposons que `V1` et `V2` sont des variables du code source

d'un système. `v2` est dérivée par analyse de flux de données de `v1` si une analyse de flux de données du code source montre que `v2` prend la valeur de `v1`.

4.2.2.4 Concept

Un concept est la formulation d'une exigence système du point de vue du domaine du problème ou du domaine de la solution. Il est identifié par un nom. Comme le montre la figure 4.3, nous distinguons deux types de concepts: concept indépendant de plate-forme ou PIC (« Platform Independent Concept ») et concept spécifique à une plate-forme ou PSC (« Platform Specific Concept »). Un PIC est un concept du domaine du problème alors qu'un PSC est un concept du domaine de la solution.

Un PIC appartient à V_{PI}. Il indique un type d'élément requis afin de respecter une exigence du domaine de problème et il est de l'un des types suivants :

- classificateur (classe ou interface);
- attribut;
- opération;
- paramètre;
- relation (association, généralisation ou dépendance).

Un PSC appartient à V_{PS}. Il indique un type d'élément requis afin de respecter une exigence de la plate-forme d'implémentation et il est de l'un des types suivants :

- de classificateur (classe ou interface);
- d'attribut;
- d'opération;
- de paramètre.

Par exemple, il est facile de constater en observant son nom qu'une classe nommée `AccountEntity` est liée au PIC `Account` car elle représente la classe du domaine du problème nommée `Account`. Elle est aussi liée au PSC `Entity` car elle implémente ce concept de la plate-forme d'implémentation.

4.2.2.5 Concept indéfini

Un concept est indéfini s'il s'agit d'un concept détecté par notre approche, mais que celle-ci n'arrive pas à nommer automatiquement.

4.2.2.6 Concepts prédéfinis

Dans notre approche, nous avons prédéfini un ensemble de PSCs :

- `PICInterface` (respectivement, `PICClass`) : ce concept indique que la plate-forme d'implémentation exige la présence d'au moins une interface dans le PSM (respectivement, classe) dont le nom appartient à V_{PI};
- `PICAttribute` (respectivement, `PICOperation`) : ce concept indique que la plate-forme d'implémentation exige la présence d'au moins un attribut (respectivement, opération), appartenant à un classificateur du PSM, dont le nom appartient à V_{PI};
- `PICParameter` : ce concept indique que la plate-forme d'implémentation exige la présence d'au moins un paramètre, appartenant à une opération du PSM, dont le nom appartient à V_{PI}.

Par exemple, l'interface du PSM nommée `Account` est liée au PSC prédéfini `PICInterface` car elle représente directement une classe du domaine du problème nommée `Account`.

4.2.2.7 Groupes de classificateurs ou d'opérations

Un groupe de classificateurs (respectivement opérations) comme le montre la figure 4.3 est un ensemble de classificateurs (respectivement opérations) regroupés selon un critère. Afin d'alléger la description de notre approche, nous tenons pour acquis qu'un groupe contient toujours au moins deux éléments, sinon c'est qu'il ne s'agit pas d'un groupe. Un critère, pour créer l'un de ces groupes, peut être l'un des éléments suivants:

- basé sur LSI, il permet le regroupement des classificateurs similaires selon LSI;
- basé sur les clones, il permet le regroupement des opérations qui partagent du code cloné;
- basé sur les patrons, il permet le regroupement des classificateurs ou des opérations ayant un patron commun dans leur nom;
- basé sur les relations, il permet le regroupement des classificateurs implémentant une interface commune ou héritant d'un classificateur commun.

Un groupe peut être subdivisé en sous-groupes selon l'un des critères définis ci-dessus. Bien sûr, un classificateur ou une opération peut appartenir à différents groupes ou sous-groupes. Par exemple, le groupe des classificateurs du PSM basé sur le patron `Account` contient les éléments suivants : `Account`, `AccountEntity`, `AccountFacade`, `AccountHome`, `AccountHomeImpl`, `AccountPersistence`, `AccountTransaction`, `AccountTransactionSession`, `AccountNode` et `AccountType`.

4.2.2.8 Propriété de concept

Comme le montre la figure 4.3, un concept peut avoir une ou plusieurs propriétés de concept ou PRP (« PRoPerty concept »). Un PRP permet de préciser des détails du

concept le contenant dans la formulation d'une exigence du système, autant du domaine du problème que du domaine de la solution. Un PRP possède deux propriétés : `nom` et `valeur`. Ainsi, un PSC d'attribut possède le PRP prédéfini `initialValue`. Celui-ci indique la valeur initiale d'un attribut du PSM. Un PSC d'opération possède le PRP prédéfini `operationBody` qui indique le code source d'une opération du PSM.

4.2.2.9 Contrainte

Une contrainte peut exister entre deux PSCs `C1` et `C2` et peut être de l'un des types suivants comme le montre la figure 4.3 :

- dépendance, si l'implémentation de `C1` exige l'implémentation de `C2`;
- compatibilité, elle si l'implémentation de `C1` est compatible avec l'implémentation de `C2`;
- incompatibilité si l'implémentation de `C1` est incompatible avec l'implémentation de `C2`;
- raffinement, si l'implémentation de `C2` présente des variations et que l'implémentation de `C1` est l'une de ces variations.

4.2.2.10 Modèle de transformation

Un modèle de transformation est un modèle utilisé afin de décrire comment effectuer la transformation d'un élément du PIM vers un (ou des) élément(s) du PSM. Comme le montre la figure 4.3, un modèle de transformation implémente un PSC primaire et possiblement un ensemble de PSCs auxiliaires. Ces PSCs auxiliaires sont la représentation de variations de son implémentation et sont donc des raffinements du PSC primaire. Nous définissons un modèle de transformation par quatre propriétés :

- `primaryImplementedPSC` : le PSC primaire mis en application par le modèle;

- `implementedPSCsForVariations` : l'ensemble des PSCs représentant les variations d'implémentation du PSC primaire;
- `context` : le type de l'élément de modèle en entrée;
- `result` : l'élément résultant de la transformation.

Un modèle de transformation peut être de l'un des cinq types suivants : classe, interface, attribut, opération ou paramètre.

Un modèle de transformation de type classe (respectivement interface, attribut, opération et paramètre) est un modèle dont le résultat de son application est un élément du PSM de type classe (respectivement interface, attribut, opération et paramètre).

Un modèle de transformation de type classificateur (classe ou interface) contient un ensemble d'appels à des modèles de type attribut et d'opération responsables de la création des attributs et des opérations du classificateur du PSM résultant.

Un modèle de transformation de type opération contient un ensemble d'appels à des modèles de transformation de type paramètre responsables de la création des paramètres de l'opération résultante dans le PSM.

4.2.3 Découverte des modèles

Nous présentons dans cette section la première partie de notre processus de modernisation : la découverte des modèles. Elle représente la partie gauche de la figure 4.1. Dans les trois chapitres suivants, nous vous présenterons chacune des phases de cette partie en détail.

Comme décrit au chapitre II, la partie de gauche de la figure représente la première partie du processus ADM, celle qui utilise la rétro-ingénierie. Elle a pour but de découvrir les modèles abstraits représentant le système légataire. Quant au contenu de la partie de droite de la figure 4.1, elle représente la seconde partie du processus

ADM, celle qui utilise l'ingénierie en avant. Elle représente l'application des modèles du PDM à un PIM marqué avec les PSCs du profil de la plate-forme d'implémentation du PDM afin de générer un PSM et ultimement un code source. Cette partie du processus ne représente pas une problématique particulière et n'est pas présentée en détail dans la thèse. Mais, elle est employée dans notre cadre de validation afin de vérifier la qualité des modèles découverts.

La figure 4.4 reprend la partie gauche de la figure 4.1 sous une forme représentant plus en détail nos phases de rétro-ingénierie. Les flèches entre les différentes boîtes représentent nos différentes transformations. Une flèche entre un modèle et un autre indique que la découverte du modèle de droite utilise le modèle de gauche en entrée. La figure indique aussi quel ensemble des règles présentées aux chapitres suivants correspond à chacune des transformations.

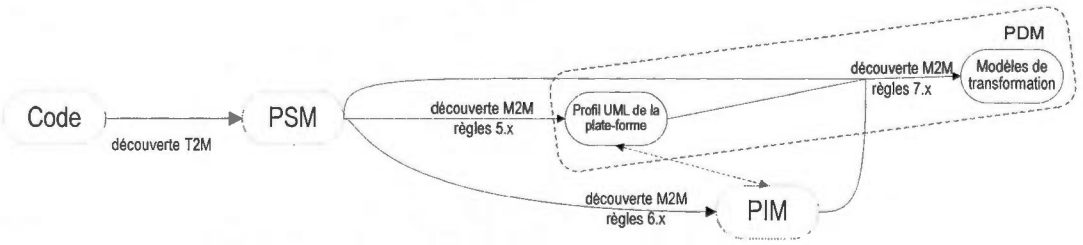

Figure 4.4 La partie de découverte des modèles plus en détails

4.2.3.1 Découverte du modèle spécifique à une plate-forme (PSM)

La première découverte, texte à modèle, représente la découverte du modèle spécifique à une plate-forme. Comme la flèche la plus à gauche dans la figure 4.4 l'indique, elle utilise comme intrant le code source du système légataire. Le PSM extrait prend la forme d'un diagramme de classes. Dans notre implémentation courante, il s'agit de code source Java. Mais l'approche pourrait être étendue à d'autres langages.

Comme cette découverte ne pose pas de problématique particulière, elle ne sera pas présentée plus en détail dans cette thèse. Aussi comme l'on peut le voir dans la figure

4.4, ce PSM est utilisé par les trois autres découvertes. Nous tiendrons pour acquis dans la présentation de ces découvertes que le PSM est déjà disponible.

4.2.3.2 Découverte du profil UML de la plate-forme d'implémentation

La seconde découverte, modèle à modèle, représente la découverte du profil UML de la plate-forme d'implémentation. Comme la flèche entre « PSM » et « Profil » l'indique dans la figure 4.4, elle utilise comme intrant le PSM découvert, mais on note aussi une flèche pointillée indiquant que cette découverte se fait en parallèle avec la découverte du PIM, car ces deux découvertes ont besoin de collaborer pour s'effectuer.

Ce premier point de vue du modèle de plate-forme montre les concepts (PSCs) de la plate-forme d'implémentation d'un système légataire dans un profil UML. Le profil UML de notre système jouet est exprimé à la figure 4.5. Cette figure représente une partie du profil UML de la plate-forme d'implémentation du système jouet où l'on peut observer différents concepts présents dans sa plate-forme d'implémentation. Notez les différents préfixes, capturant les types des concepts, ont été ajoutés afin de garantir l'unicité des identificateurs des classes dans le profil UML.

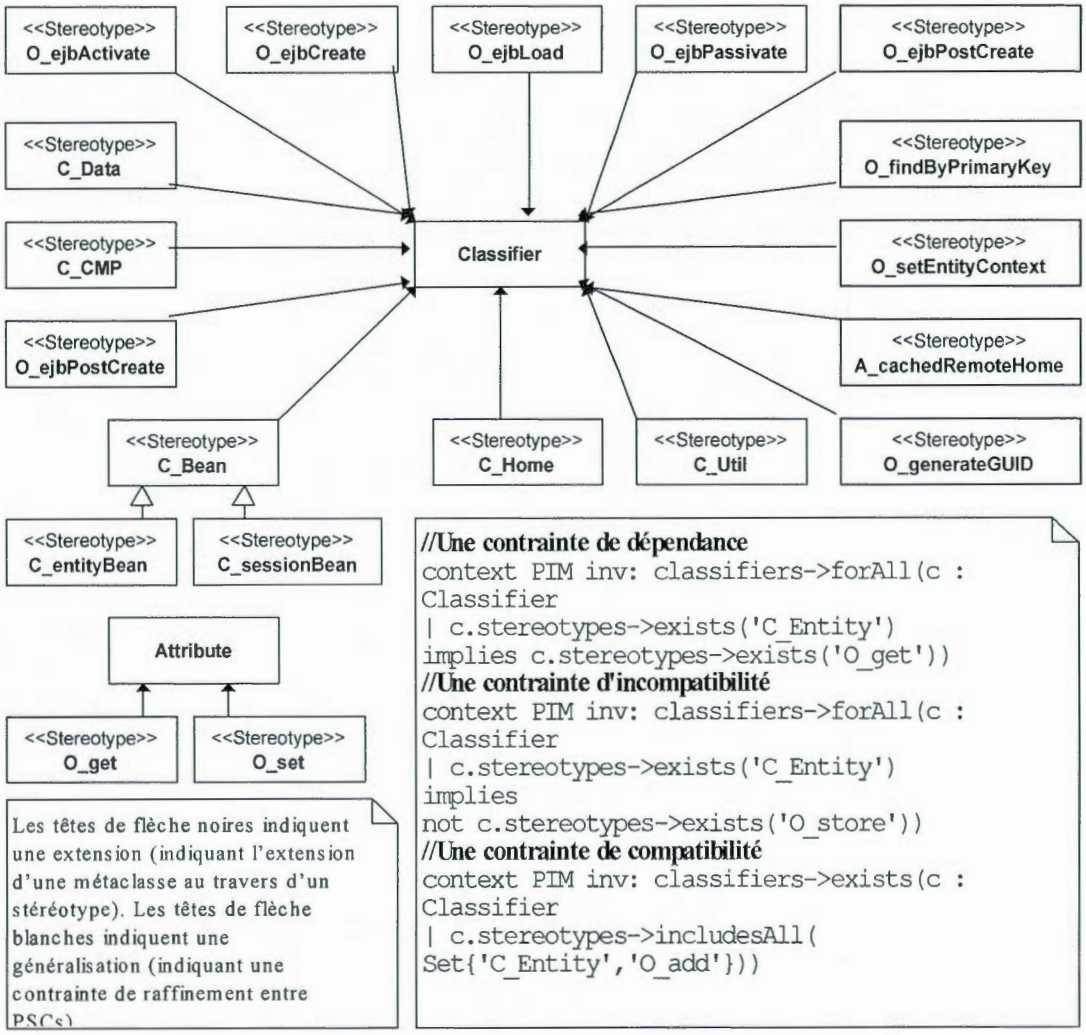

Figure 4.5 Extrait du profil UML de la plate-forme du système jouet

Nous voyons aussi des exemples de contraintes existant entre ces concepts. Par exemple, on peut voir à la figure 4.5 que l'on retrouve, entre autres, les contraintes suivantes :

- tous les classificateurs qui implémentent le concept de classificateur `Entity` implémentent aussi le concept d'opération `get`;
- aucun classificateur n'implémente à la fois le concept de classificateur `Entity` et le concept d'opération `store`;

- un classificateur qui implémente le concept de classificateur `Entity` peut (ou ne pas) implémenter le concept d'opération `add`;
- les concepts de classificateur `EntityBean` et `SessionBean` sont des raffinements du concept de classificateur `Bean`.

La découverte du profil UML de la plate-forme d'implémentation est expliquée en détail au chapitre 5.

4.2.3.3 Découverte du modèle indépendant d'une plate-forme (PIM)

La troisième découverte, modèle à modèle, représente la découverte du modèle indépendant d'une plate-forme. Comme la flèche dans la figure 4.4 l'indique, elle utilise aussi le PSM découvert.

Notre modèle indépendant de plate-forme décrit l'ensemble des classes métiers, leurs attributs et opérations et leurs relations dans un diagramme de classes UML.

Notre approche permet la découverte du PIM à partir d'une élimination par filtrage du code relié à l'infrastructure de la plate-forme d'implémentation. Dans la figure 4.6, nous montrons le PIM obtenu pour notre système jouet où tous les éléments spécifiques à la plate-forme d'implémentation sont enlevés.

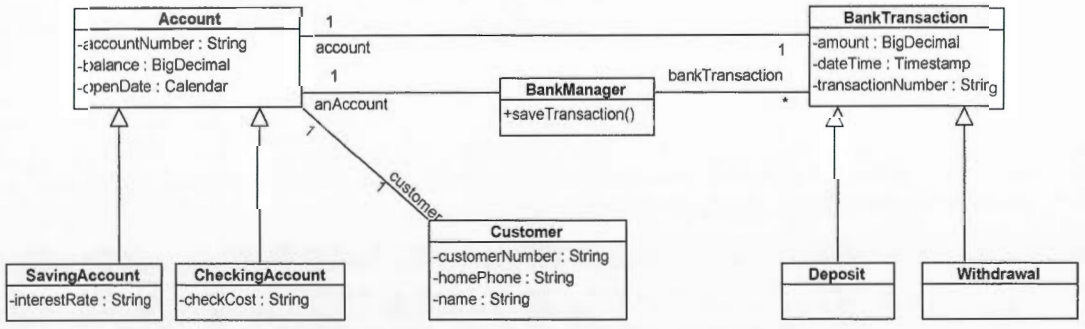

Figure 4.6 Le PIM du système jouet

La découverte du modèle indépendant d'une plate-forme est expliquée en détail au chapitre 6.

4.2.3.4 Découverte des modèles de transformation de la plate-forme d'implémentation

La dernière découverte, modèle à modèle, est celle de la « découverte des modèles de transformation de la plate-forme d'implémentation ». Comme la flèche à trois arcs dans la figure 4.4 l'indique, cette découverte emploie les trois modèles découverts dans les phases précédentes.

Ce second point de vue du modèle de plate-forme prend la forme d'un ensemble de modèles de transformation. Ces modèles sont des modèles paramétrés capturant le code source de la plate-forme d'implémentation du système. Ils sont ultimement exprimés en QVT.

Le tableau 4.2 présente un exemple de modèle de transformation découvert pour le système jouet. Ce modèle est sérialisé en transformation QVT. On voit que cette transformation crée, pour chaque classe du PIM liée au concept de plate-forme `Home`, une interface. Le nom de cette interface reprend le nom de sa classe d'origine dans le PIM (représentée par `context`) accolé au suffixe `Home`. La visibilité de l'interface sera publique et son paquetage sera le même que `context` et elle n'importe pas d'autres classificateurs. Dans ce modèle de transformation, il y a deux types d'implémentation du concept `Home`, qui varient par le nom de leur super classe. En fait, une analyse manuelle de cette transformation ainsi obtenue par notre approche par un architecte en J2EE peut facilement identifier que nous n'avons pas besoin de deux nouveaux concepts de classificateur (`Home_Variation_1` et `Home_Variation_2`) pour capturer cette variation puisqu'elle est simplement conditionnelle au fait qu'une classe du PIM possède ou non une super classe. Le tableau 4.2 montre aussi que le modèle de transformation appelle des modèles de transformation d'attribut et d'opération afin de créer ces éléments pour les différentes interfaces créées par cette transformation.

Tableau 4.2 Exemple de modèle de transformation sérialisé en QVT

```
mapping Class::classToJavaInterface_Home() : Interface
when{ self.stereotypeList->includes('C_Home');}
{
  var context           := self;
  name                  := context.name+'Home';
  visibility            := 'public';
  packageName           := context.packageName;
  if(self.stereotypeList->includes('C_Home_Variation_1'))then
  {
    superClassifierName    := 'javax.ejb.EJBHome'
  }
  else
  {
    if(self.stereotypeList->includes('C_Home_Variation_2'))then
    {
      superClassifierName    := context.packageName+'.'+context.superClassifierName+'Home'
    }
    endif;
  }
  endif;

  importPackageList     += null;
  attributeList         += self->map classToJavaAttribute_COMP_NAME(self);
  attributeList         += self->map classToJavaAttribute_JNDI_NAME(self);
  operationList         += self->map classToJavaOperation_create(self);
  operationList         += self->map classToJavaOperation_findAll(self);
  operationList         += self->map classToJavaOperation_findByPrimaryKey(self);
}
```

La découverte des modèles de transformation de la plate-forme d'implémentation est expliquée en détail au chapitre 7.

4.3 SYNTHÈSE

Nous avons présenté dans ce chapitre :

- notre approche de modernisation basée sur ADM;
- le système jouet qui nous aide à expliquer cette approche;
- un ensemble de définitions afin d'aider à comprendre la description de cette approche;
- une vue de la partie de découverte des modèles représentant un système légataire qui sont:
    - le modèle spécifique à une plate-forme (PSM);

- le modèle de plate-forme d'implémentation (PDM) qui est constitué de deux sous modèles : son profil UML et un ensemble de modèles de transformation;
- le modèle indépendant d'une plate-forme (PIM).

Nous présentons dans les trois chapitres suivants, l'approche en détail.

CHAPITRE V

DÉCOUVERTE DU PROFIL UML DE LA PLATE-FORME D'IMPLÉMENTATION

Dans les chapitres précédents, nous décrivons la problématique de la modernisation, l'ingénierie dirigée par les modèles, l'état de l'art entourant ces sujets et un aperçu global de notre approche de modernisation.

Dans ce chapitre, nous présentons la phase de la découverte du profil UML de la plate-forme d'implémentation qui est l'un des trois modèles que nous découvrons dans notre processus de modernisation. Nous donnons d'abord une vue générale de cette phase avant de l'expliquer en détail à l'aide de règles et d'exemples. Nous décrivons ensuite notre processus de validation et ses résultats pour terminer par une discussion sur ces résultats.

5.1 VUE GÉNÉRALE

La phase de découverte du profil UML de la plate-forme d'implémentation est composée de deux sous-phases principales. La première est la découverte du profil UML de la plate-forme d'implémentation du système légataire. La seconde est la sérialisation de ce profil dans un modèle XMI.

Comme la seconde sous-phase ne constitue pas un défi particulier et que notre principale contribution se situe dans la première sous-phase, nous allons décrire celle-ci en détail dans la suite de ce chapitre.

5.2 SOUS-PHASE 1 : DÉCOUVERTE DU PROFIL UML DU MODÈLE DE PLATE-FORME

Dans cette section nous décrivons la sous-phase de découverte du profil UML de la plate-forme d'implémentation du système légataire à l'aide de règles et d'exemples d'application de celles-ci sur le système jouet décrit au chapitre 4.

Comme l'illustre la figure 5.1, cette sous-phase est composée de six étapes et débute par l'étape de la découverte des PSCs de classificateur. Cette première étape est composée de 5 sous-étapes. La première sous-étape est la découverte des premiers PSCs de classificateur. Ensuite, une sous-étape de filtrage des PSCs de classificateur qui auraient pu être découverts à tort est effectuée. La sous-étape suivante effectue la découverte des PICs de classe que contient le système légataire, car ils sont nécessaires à l'exécution des sous-étapes suivantes. Les résultats de cette découverte sont utilisés à la sous-étape suivante afin de filtrer les PSCs découverts. À la dernière sous-étape, les PICs peuvent conduire à la découverte de nouveaux PSCs de classificateur. Si c'est le cas, la première étape est reprise depuis sa deuxième sous-étape. Sinon la sous-phase se poursuit par les étapes de découverte des PSCs d'attribut, d'opération et de paramètre pour se terminer par l'étape de la découverte des contraintes entre les PSCs. La seconde sous-phase est quant à elle responsable de la création du profil UML de la plate-forme qui contient ces PSCs et ces contraintes.

5.2.1 Étape 1.1 : Découverte des PSCs de classificateur

La première étape découvre les PSCs de classificateur contenus dans le PSM. Ces PSCs peuvent être des PSCs d'interface ou des PSCs de classe. Ceux-ci indiquent les classificateurs nécessaires afin de répondre aux exigences de la plate-forme d'implémentation.

Cette étape utilise d'abord la règle 5.1 afin de découvrir un premier groupe de PSCs de classificateur. Cette première règle exploite l'idée que les classificateurs implémentant un même PSC devraient partager un vocabulaire ou du code commun.

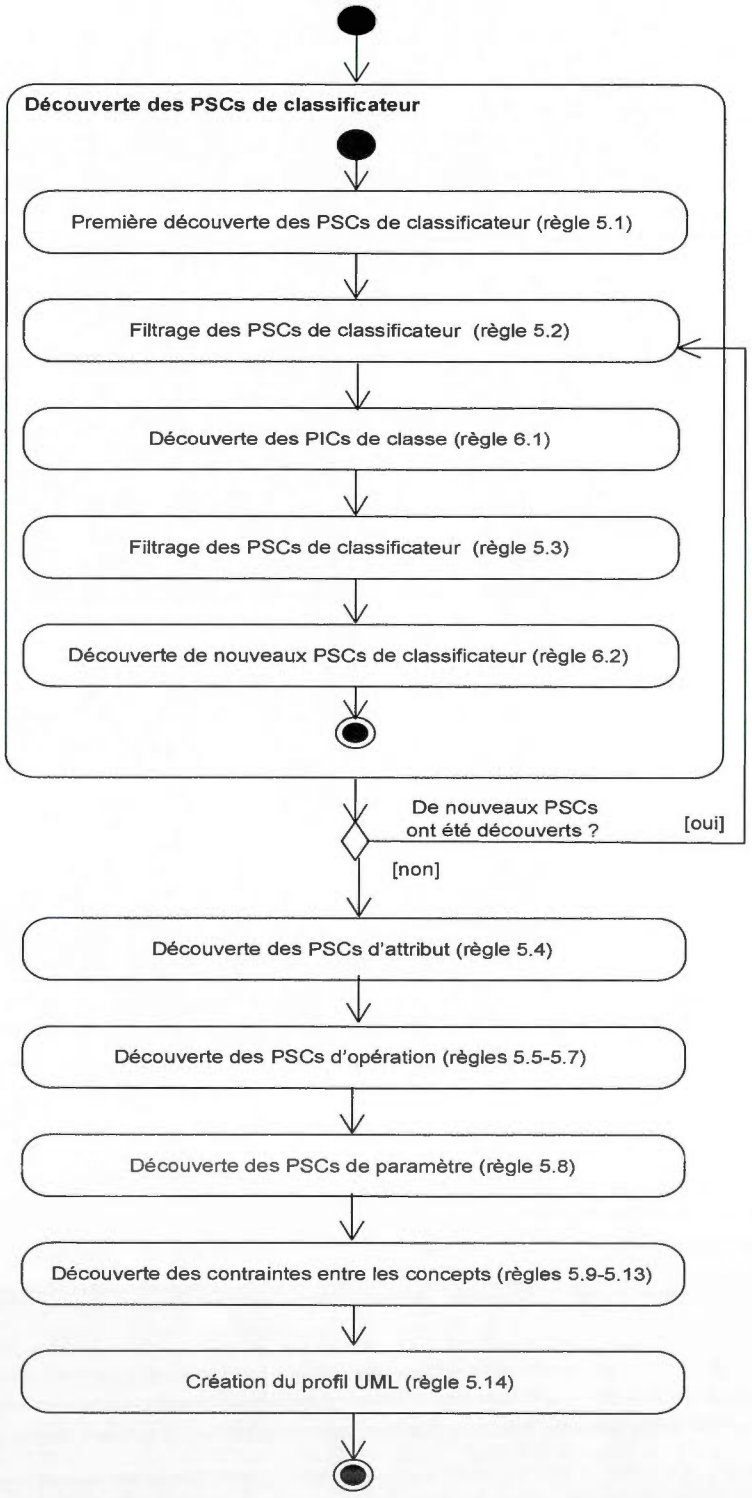

Figure 5.1 La sous-phase de découverte du profil UML

Ensuite, la règle 5.2 est utilisée afin de filtrer des PSCs non pertinents qui auraient pu être identifiés par la règle 5.1. La règle 5.2 exploite l'idée qu'un concept dont le nom contient le nom d'un second risque de contenir aussi des mots du vocabulaire du domaine du problème.

Entre l'application de la règle 5.2 et de la règle 5.3, nous procédons à la découverte des PICs de classe (règle 6.1). En effet, même si le but de la sous-phase de découverte du profil UML de la plate-forme d'implémentation n'est pas de découvrir les PICs du système, il est nécessaire d'y découvrir les PICs de classe du PIM représenté dans le PSM, car cette information est utilisée par certaines règles lors de la découverte du profil UML. La découverte de PICs de classe du PIM est présentée à la section 6.2.

Finalement, la règle 5.3 retire les PSCs de classificateur qui ne sont plus liés à un PIC de classe une fois ceux-ci découverts. En effet, il est facile d'imaginer le cas où des classificateurs seulement liés à la plate-forme d'implémentation et contenant un patron commun à la fin de leur nom auraient été regroupés par LSI, ce patron deviendrait un faux PSC de classificateur qui après la découverte des PICs de classe ne serait lié à aucun de ceux-ci.

Règle 5.1

Chaque groupe basé sur LSI (respectivement basé sur les clones) représente un PSC de classificateur. Si tous les classificateurs du groupe (respectivement classificateurs contenant les opérations) ont un patron commun dans leurs noms (qui n'est pas le nom d'un classificateur du PSM.), le PSC est nommé par le plus long patron commun. S'il n'y a pas de patron commun, le nom du PSC demeure indéfini.

Cette règle isole dans notre système jouet les cinq PSCs de classificateur suivants: `Bean`, `CMP`, `Data`, `Home` et `Util`. Par exemple, tous les classificateurs du groupe basé sur LSI { `AccountBean`, `BankTransactionBean`, `CheckingAccountBean`,

`CustomerBean, DepositBean, WithdrawalBean` } ont dans leur nom le patron commun `Bean`. Ce patron devient alors le nom d'un PSC de classificateur.

Règle 5.2

Si le nom d'un PSC de classificateur `C1` est une sous-chaîne du nom d'un autre PSC de classificateur `C2` et que le groupe des classificateurs reliés à `C2` est un sous-groupe de ceux reliés à `C1` alors `C2` est retiré de la liste des PSCs.

Dans notre système jouet, si un groupe LSI avait été créé contenant seulement les classificateurs `CheckingAccountHome` et `SavingAccountHome` alors un PSC de classificateur nommé `AccountHome` serait créé dans un premier temps. Si le PSC de classificateur nommé `Home` est aussi détecté et que son groupe relié contient ces deux classificateurs, la règle 5.2 va retirer le PSC `AccountHome`.

Règle 5.3

Chaque PSC de classificateur qui n'est pas lié à un PIC de classe est retiré de la liste des PSCs.

Dans le système BJDBC, que nous décrivons à l'appendice A et qui nous servira à décrire certaines règles ne s'appliquant pas à notre exemple jouet, les PSCs de classificateur `Iterator` et `Search` sont enlevés, car ils ne sont pas liés à un PIC de classe détecté dans les règles expliquées au chapitre suivant lors de la découverte du PIM.

5.2.2 Étape 1.2 : Découverte des PSCs d'attribut

La deuxième étape découvre les PSCs d'attribut du PSM en recherchant les mots du vocabulaire du PSM les représentant. Ces PSCs indiquent une représentation abstraite des attributs nécessaires à certains classificateurs pour répondre aux exigences de la plate-forme d'implémentation.

Cette étape utilise la règle 5.4 qui exploite un des fondements de notre approche, à savoir que les éléments reliés à une plate-forme d'implémentation sont implémentés de manières répétitives ou semi-répétitives. C'est pourquoi nous prenons comme hypothèse que si un attribut se retrouve dans tous les classificateurs d'un sous-groupe de classificateurs implémentant un même PSC de classificateur alors cet attribut représente probablement un PSC d'attribut. Notez que nous vérifions dans un sous-groupe et non pas dans tous les classificateurs implémentant un PSC de classificateur, car celui-ci peut présenter des variations d'implémentation.

Règle 5.4

Un attribut d'un classificateur du PSM représente un PSC d'attribut, s'il existe un PSC de classificateur `C` où cet attribut appartient à tous les classificateurs de son groupe ou à l'un de ses sous-groupes en lien avec le critère d'implantation d'une interface ou d'héritage d'un classificateur. Un sous-groupe doit contenir au moins deux classificateurs.

Dans notre système jouet, tous les classificateurs du PSM implémentant le PSC de classificateur nommé `Util` possèdent l'attribut `cachedRemoteHome`. Alors, cet attribut représente un PSC d'attribut.

5.2.3 Étape 1.3 : Découverte des PSCs d'opération

La troisième étape découvre les PSCs d'opération de la plate-forme d'implémentation en recherchant les mots du vocabulaire du PSM les représentant. Ces PSCs indiquent une représentation abstraite des opérations nécessaires à certains classificateurs pour répondre aux exigences de la plate-forme implémentation. Cette étape utilise les règles 5.5, 5.6 et 5.7.

L'objectif de la règle 5.5 est de discerner les opérations utilitaires liées à des éléments du PIM comme les « getters » et « setters » en découvrant ces liens. L'idée derrière les règles 5.6 et 5.7 est que les opérations contenant une occurrence commune d'un

mot du vocabulaire dans leur nom ou un clone représentent probablement un PSC d'opération.

Règle 5.5

Une opération `O` d'un classificateur `C1` du PSM est liée à un attribut `A` d'un classificateur `C2` du PSM si le nom de `O` contient le nom de `A` et que l'une des conditions suivantes est satisfaite :

1. `C1` et `C2` sont les mêmes;
2. `C1` est lié à un PIC de classe `P` et `C2` appartient au groupe basé sur le patron du nom de `P`;
3. `C1` a un parent (direct ou indirect) `S`, `S` est lié à un PIC de classe `P` et `C2` appartient au groupe basé sur le patron `P`.

Dans notre système jouet, l'opération `getAccountNumber` du classificateur `AccountData` du PSM est liée à l'attribut `accountNumber` car ce dernier est un attribut de ce même classificateur (condition 1). Le classificateur `CheckingAccountCMP` hérite indirectement de `AccountBean` (par l'intermédiaire de `CheckingAccountBean`); le classificateur `AccountBean` est lié au PIC de classe `Account`; le classificateur `AccountData` qui appartient au groupe basé sur le patron `Account`, possède l'attribut `accountNumber`; on conclut donc que l'opération `setAccountNumber` du classificateur `CheckingAccountCMP` est liée à l'attribut `accountNumber` du classificateur `AccountData` (condition 3).

Règle 5.6

Chaque opération liée à un verbe `V` d'un groupe basé sur LSI représente un PSC d'opération. Si cette opération est liée à un attribut, alors ce PSC est nommé par le verbe `V`, sinon, il est nommé par le nom de l'opération.

Dans notre système jouet, le verbe `get` devient un PSC d'opération, car l'opération `getAccountNumber` du classificateur `AccountData` est liée à son attribut `accountNumber`. L'opération `ejbStore`, du classificateur `DepositBean`, qui contient le verbe `store` n'est pas liée à un attribut; `ejbStore` représente alors un PSC d'opération et non le verbe `store`.

Règle 5.7

Chaque groupe basé sur les clones représente un PSC d'opération. Son nom dépend de la première des sous-règles suivantes s'appliquant :

- si toutes les opérations du groupe ont le même nom, il est nommé par le nom commun;
- si toutes les opérations ont un verbe commun, il est nommé par ce verbe;
- si aucune des sous-règles précédentes ne s'applique, son nom demeure indéfini.

Dans notre système jouet, le groupe de clones basé sur de l'opération `toString` {`AccountData`, `BankTransactionData`, `CustomerData`} représente un PSC d'opération désigné par le nom de l'opération commune (ici, `toString`).

5.2.4 Étape 1.4 : Découverte des PSCs de paramètre

La quatrième étape découvre les PSCs de paramètre de la plate-forme d'implémentation en recherchant les mots du vocabulaire du PSM les représentant. Ces PSCs indiquent une représentation abstraite des paramètres nécessaires à certaines opérations pour répondre aux exigences de la plate-forme implémentation.

Cette étape utilise la règle 5.8 dont l'objectif est de discerner les paramètres des opérations liées à des éléments du PIM.

Règle 5.8

Chaque paramètre P d'une opération O1 d'un classificateur du PSM C1 lié à une classe C2 du PIM représente un PSC de paramètre homonyme si P n'est pas lié à un élément du PIM E. Le lien entre un paramètre P d'une opération O d'un classificateur du PSM C1 lié à une classe C2 du PIM et un élément du PIM dépend de la première condition suivante qui s'applique :

- s'il existe un attribut A1 de C1 où A1 est lié à un attribut A2 (respectivement une extrémité d'association AE) de C2 et A1 est relié à P par l'analyse de flux de données, alors P est lié à A2 (respectivement AE);
- s'il existe un attribut A (respectivement une extrémité d'association AE) de C2 dont le nom est le patron le plus long figurant dans le nom de P, alors P est lié à A (respectivement AE);
- si le nom de C2 est le patron le plus long figurant dans le nom de P, alors P est lié à C2.

Dans notre système jouet, le paramètre balance de l'opération setBalance de la classe Account ne représente pas un PSC de paramètre, car son nom contient le nom d'un attribut de la classe du PIM relié au classificateur contenant cette opération. Par contre, le paramètre ctx de l'opération setEntityContext de la classe AccountCMP devient un PSC de paramètre puisqu'il n'est relié à aucun élément du PIM.

5.2.5 Étape 1.5 : Découverte des contraintes entre les concepts

La cinquième étape met en évidence les contraintes qui existent entre les PSCs. Cette étape utilise les règles 5.9 à 5.13. Les trois premières règles utilisent les relations entre les éléments du PSM afin d'identifier les contraintes. L'idée derrière les règles 5.12 et 5.13 est qu'un PSC peut avoir des variations dans son implémentation. Ces variations se concrétisent par l'existence de plus d'un groupe associé à ce PSC. Chaque variation peut représenter un nouveau PSC qui affine l'original.

Règle 5.9

Une contrainte de dépendance existe entre :

- un PSC de classificateur `C` et un PSC d'attribut `A` (respectivement un PSC d'opération `O`) si tous les classificateurs implémentant `C` implémentent également `A` (respectivement `O`) ;
- un PSC d'opération `O` et un PSC de paramètre `P` si toutes les opérations implémentant `O` implémentent également `P`.

Dans notre système jouet, tous les classificateurs implémentant le PSC de classificateur `Bean` implémentent également le PSC d'opération `ejbPostCreate`.

Règle 5.10

Une contrainte de compatibilité existe entre deux PSCs `P1` et `P2` si certains des classificateurs implémentant `P1` implémentent aussi `P2`.

Dans notre système jouet, certains classificateurs implémentent le PSC de classificateur `Util` et le PSC d'attribut `cachedRemoteHome`.

Règle 5.11

Une contrainte d'incompatibilité existe entre deux PSCs `P1` et `P2` s'il n'existe pas de classificateur implémentant à la fois `P1` et `P2` dans le PSM.

Dans notre système jouet, aucun classificateur n'implémente à la fois le PSC de classificateur `Util` et le PSC d'opération `ejbPostCreate`.

Règle 5.12

Si un PSC `P` est associé à plus d'un groupe, que chaque groupe est basé sur LSI et qu'au moins deux groupes implémentent ou héritent d'un classificateur différent C_i (un groupe implémente ou hérite d'un classificateur `C` si tous les classificateurs du

groupe implémentent C) qui n'est pas lié à un PIC de classe, alors tous les classificateurs C_i représentent un nouveau PSC du même type, et il est nommé du nom du classificateur C_i (après avoir enlevé toute occurrence d'un mot du vocabulaire du domaine du problème). Une contrainte de raffinement est ensuite ajoutée entre chacun de ces nouveaux PSCs et P.

Dans notre système jouet, le PSC de classificateur `Bean` est associé aux groupes suivants: {`AccountBean`, `BankTransactionBean`, `CheckingAccountBean`, `CustomerBean`, `DepositBean`, `WithdrawalBean`} et {`BankManagerBean`}; le classificateur du groupe avec un unique élément {`BankManagerBean`} implémente l'interface `javax.ejb.SessionBean` alors que tous les classificateurs de l'autre groupe implémentent l'interface `javax.ejb.EntityBean`; deux PSCs de classificateur sont alors ajoutés: `EntityBean` et `SessionBean`. Deux contraintes de raffinement sont également ajoutées: l'une entre `EntityBean` et `Bean` et l'autre entre `SessionBean` et `Bean`.

Règle 5.13

Si le PSC P est associé à plus d'un groupe et que chaque groupe est basé sur les clones alors, chaque groupe représente un nouveau PSC du même type. Le nom de ces nouveaux PSCs est indéfini et une contrainte de raffinement est ajoutée entre chacun d'eux et P.

Dans notre exemple, le PSC d'opération `getData` est associé aux groupes suivants :

- {`AccountCMP.getData`, `BankTransactionCMP.getData`, `CustomerCMP.getData`} ;
- {`AccountCMP.getData`, `BankTransactionCMP.getData`, `CheckingAccountCMP.getData`, `CustomerCMP.getData`, `DepositCMP.getData`, `SavingAccountCMP.getData`, `WithdrawalCMP.getData`} ;

- {CheckingAccountCMP.getData, DepositCMP.getData, SavingAccountCMP.getData, WithdrawalCMP.getData}.

Les trois groupes vont générer trois nouveaux PSCs d'opération. Chacun de ces nouveaux PSCs est un raffinement du PSC d'opération `getData`. Le code source de l'opération `getData` suit le même schéma pour tous les classificateurs d'opération implémentant le PSC de classificateur `CMP`. Un extrait du code source de l'opération est donné dans la figure 5.2 pour le classificateur `AccountCMP`. Le nombre de lignes du code de l'opération dépend du nombre d'attributs dans un classificateur. Le fragment de code est de type-3 (voir section 3.4). Comme l'outil que nous utilisons pour détecter les clones ne prend pas en charge ce type, nous obtenons trois groupes au lieu d'un seul groupe. Les trois nouveaux PSCs sont alors tous de faux positifs, car il n'y a pas de variation dans l'implémentation de l'opération.

```
public test.AccountData getData()
{
 test.AccountData dataHolder = null;
 try
 {
  dataHolder = new test.AccountData();
  dataHolder.setAccountNumber( getAccountNumber());
  dataHolder.setBalance( getBalance() );
  dataHolder.setCustomer( getCustomer() );
  dataHolder.setOpenDate( getOpenDate() );
 }
 catch (RuntimeException e)
```

Figure 5.2 Extrait du code source de l'opération `getData` de `AccountCMP`

5.2.6 Étape 1.6 : Création du modèle de profil UML

La sixième étape crée le modèle du profil UML de la plate-forme d'implémentation. Cette étape utilise la règle 5.14.

Règle 5.14

Le diagramme de classes UML décrivant le profil de la plate-forme d'implémentation est créé en appliquant en ordre les points suivants :

1. Une classe avec un stéréotype `Stereotype` est créée pour chaque PSC de classificateur dont le nom est défini. La classe est nommée par le nom du PSC préfixé de `C_`. Une relation d'extension est ajoutée entre la nouvelle classe et la classe `Classifier`.
2. Une classe avec un stéréotype `Stereotype` est créée pour chaque PSC d'attribut dont le nom est défini. La classe est nommée par le nom du PSC préfixé de `A_`. Une relation d'extension est ajoutée entre la nouvelle classe et la classe `Classifier`.
3. Une classe avec un stéréotype `Stereotype` est créée pour chaque PSC d'opération dont le nom est défini. La classe est nommée avec le nom du PSC préfixé de `O_`. Une relation d'extension est ajoutée entre la nouvelle classe et la classe `Attribute` (respectivement, la classe `Classifier`) si le PSC est généré à partir d'un verbe (respectivement si ce n'est pas le cas).
4. Une classe avec un stéréotype `Stereotype` est créée pour chaque PSC de paramètre dont le nom est défini. La classe est nommée par le nom du PSC préfixé de `P_`. Une relation d'extension est ajoutée entre la nouvelle classe et la classe `Paramètre`.
5. Les contraintes de dépendance, de compatibilité et d'incompatibilité sont exprimées en OCL.
6. Pour chaque contrainte de raffinement entre deux PSCs `P1` et `P2`, une relation d'héritage est créée entre leurs classes correspondantes.

La figure 4.5 présente une partie du profil UML pour notre système jouet.

5.3 VALIDATION DU TRAVAIL

Dans cette section, nous présentons notre cadre de validation, ensuite, nous présentons les résultats obtenus lors de notre processus de validation et enfin, nous

discutons de plusieurs aspects de l'approche proposée et soulignons certains travaux futurs.

5.3.1 Cadre de validation

Nous avons développé un prototype afin de valider notre approche mettant en application les règles présentées pour les étapes de notre approche. Nous avons développé notre propre analyse LSI et nous utilisons l'outil CCFinder (Kamiya, Kusumoto et Inoue, 2002) pour découvrir des groupes d'opérations clonées (voir appendice C).

Notre prototype prend en entrée du code source Java. Il ne supporte pas actuellement d'autres types d'artefacts systèmes tels que les fichiers de descripteur de déploiement. Par conséquent, certains concepts d'implémentation d'une plate-forme ne peuvent pas être découverts s'ils ne sont présents que dans ces artefacts non pris en charge.

Afin de vérifier la pertinence de notre approche, nous avons donc appliqué ce prototype à différentes applications et analysé les résultats afin d'en tirer des statistiques et des enseignements. Le tableau 5.1 présente un sommaire des résultats obtenus pour la découverte des PSCs.

La recherche d'éléments par la rétro-ingénierie peut mener à trois résultats possibles (Philippow *et al.*, 2005). Un élément identifié par les résultats de la recherche est un « vrai positif » s'il fait bien partie des résultats attendus. Il est un « faux positif » s'il ne fait pas partie des résultats attendus. Enfin, un « faux négatif » est un élément qui devrait s'y retrouver, mais qui ne l'est pas. En se basant sur ces résultats, il est possible de développer des métriques. Dans notre cas, nous employons les deux métriques suivantes : la précision et le rappel.

Dans ce qui va suivre, nous allons définir ces deux propriétés, puis nous allons parler des éléments que nous voulons mesurer. Nous allons aussi mentionner le protocole que nous avons suivi pour obtenir ces mesures.

5.3.1.1 Précision

Une première mesure intéressante à extraire est la précision. Elle cherche à évaluer si les éléments identifiés le sont vraiment. Par exemple si nous avons identifié 50 PSCs de classificateur et que sur ces 50, 5 n'en sont pas vraiment nous avons donc une précision de 90 % (45/50).

5.3.1.2 Couverture

Une seconde mesure intéressante à extraire est le rappel. Il s'agit d'évaluer quel pourcentage des éléments recherchés est identifié. Ainsi, si nous savons (ou estimons) que dans le modèle original il y a 50 PSCs de classificateur et que l'outil en ignore 5, nous pouvons dire que nous avons un rappel de 90 % (45/50).

5.3.2 Éléments mesurés et protocole suivi

Le but de notre processus de validation est d'évaluer si les PSCs extraits appartiennent vraiment à la plate-forme d'implémentation. Nous mesurons les données suivantes : nombre d'éléments découverts correctement (vrai positif), nombre d'éléments découverts à tort (faux positif) et nombre d'éléments non découverts (faux négatif). Nous calculons ensuite à l'aide de ces éléments la précision et le rappel. Notez que l'extraction de contraintes entre les PSCs n'a pas été validée puisque ses règles produisent de façon systématique des contraintes en se basant sur les relations entre les éléments du code source du système analysé. Par conséquent, les contraintes trouvées reflètent fidèlement le code source et ne posent pas de défis particuliers.

Nous avons automatisé le processus de validation. Les PSCs attendus de tous les systèmes testés ont été extraits après avoir analysé manuellement leur code source. La disponibilité de la documentation EJB et la familiarité du domaine d'affaires de ces systèmes ont aidé à cette analyse. Toutefois, comme cette tâche est manuelle et que souvent une plate-forme d'implémentation n'est pas toujours implémentée rigoureusement, nous ne pouvons pas affirmer que l'information est 100 % correcte

ou complète. Ces PSCs sont donc placés dans un fichier. Lorsque notre prototype termine son travail, il compare ses résultats avec le contenu de ce fichier. Différentes traces sont également générées pour des fins de débogage.

5.3.3 Les logiciels analysés

Pour prouver la pertinence de notre approche, nous avons appliqué notre outil à six systèmes :

1. Le système jouet (Jouet) ;
2. Un système de gestion des ventes (SGV) généré à partir d'un modèle UML en utilisant l'outil MDA SourceCafe[3];
3. Un second système bancaire « Banking JDBC » (BJDBC) présenté à l'appendice A[4];
4. Un centre commercial virtuel (VSM) trouvé sur le site d'Oracle[5];
5. Un Service de courtage en ligne (OFBS) également trouvé sur le site d'Oracle[6];
6. Le EJB Pet Store de Sun (Pet)[7].

La plate-forme d'implémentation des deux premiers systèmes est basée exclusivement sur EJB. La plate-forme d'implémentation des systèmes 4, 5 et 6 est basée en partie sur EJB. Le troisième système utilise une plate-forme ad hoc.

Notez aussi que même si les deux premiers systèmes sont basés exclusivement sur EJB, ils ne contiennent pas exactement le même ensemble de PSCs puisque chacun d'eux utilise une implémentation différente d'EJB.

---

[3] www.sourcecafe.com, 09-27-2010
[4] www.oracle.com/technology/sample_code/tutorials/vsm1.3, 09-27-2010
[5] www.oracle.com/technology/sample_code/tech/java/j2ee/fbs10g, 09-27-2010
[6] java.sun.com/developer/releases/petstore, 09-27-2010
[7] www.oracle.com/technology/sample_code/tutorials/vsm1.3, 09-27-2010

5.3.4 Résultats

Le tableau 5.1 présente les résultats obtenus pour la découverte des PSCs des six logiciels analysés. Nous revenons ensuite plus en détail pour chacun de ces logiciels dans le reste de cette section.

Tableau 5.1 Résultat de la découverte des PSCs

	Jouet	SGV	BJDBC	VSM	OFBS	Pet
Nombre de lignes de code	1124	1091	2076	3198	3082	6190
Nombre de classificateurs	47	29	67	89	74	209
Nombre d'attributs	55	53	130	146	132	411
Nombre d'opérations	430	207	375	503	427	1049
VP PSC de classe	6	6	8	10	15	8
FP PSC de classificateur	0	0	1	5	3	32
FN PSC de classificateur	0	0	2	3	1	17
Rappel PSC de classificateur	100,00	100,00	80,00	76,92	93,75	32,00
Précision PSC de classificateur	100,00	100,00	88,89	66,67	83,33	20,00
VP PSC d'opération	27	29	23	54	26	82
FP PSC d'opération	0	0	40	19	18	100
FN PSC d'opération	2	2	2	75	56	143
Rappel PSC d'opération	93,10	93,55	92,00	41,86	31,71	36,44
Précision PSC d'opération	100,00	100,00	36,51	73,97	59,09	45,05
VP PSC d'attribut	5	3	6	3	7	17
FP PSC d'attribut	0	1	4	1	5	8
FN PSC d'attribut	0	0	0	54	41	135
Rappel PSC d'attribut	100,00	100,00	100,00	5,26	14,58	11,18
Précision PSC d'attribut	100,00	75,00	60,00	75,00	58,33	68,00
VP PSC de paramètre	15	5	15	27	31	85
FP PSC de paramètre	0	1	12	12	46	19
FN PSC de paramètre	0	0	6	40	20	105
Rappel PSC de paramètre	100,00	100,00	71,43	40,30	60,78	44,74
Précision PSC de paramètre	100,00	83,33	55,56	69,23	40,26	81,73

5.3.4.1 Le système jouet (Jouet)

Cette première mise en œuvre de notre algorithme s'effectuait donc sur un exemple assez facile qui est l'application bancaire présentée au chapitre 4. Cet exemple

implante une partie de l'architecture J2EE. Ce premier système était relativement facile pour nos règles, car il s'agit d'un système totalement généré. Mais ce système contient néanmoins une embuche. En effet, la classe du PIM nommée `BankManager` est la seule représentée dans le PSM sous la forme d'un « `Session Bean` », ce qui signifie que certains des PSCs propres à ce « `Bean` » ne sont pas implémentés de façon répétitive ce qui est l'une des prémisses de nos règles. Cette embuche explique les deux faux négatifs de PSC d'opération. Il s'agit des PSCs d'opération `setSessionContext` et `unsetSessionContext` qui n'ont pas été détectés, car ils sont implantés qu'une seule fois.

Il est aussi intéressant de noter que grâce à LSI et à la règle 5.1 nous avons découvert le PSC d'opération `findAll`. En effet, la règle 5.1 découvre un groupe de classificateurs contenant les interfaces `BankTransactionHome`, `AccountHome` et `CustomerHome`. On pourrait être ainsi tenté de penser que ce groupe représente le PSC `Home`. Oui, mais pas seulement, car d'autres classes implémentent aussi ce PSC. Par contre, ces trois interfaces possèdent la particularité d'être les seules à implémenter le PSC d'opération `findAll`.

5.3.4.2 Le système de gestion des ventes (SGV)

Nous avons ensuite soumis à notre outil un exemple de code généré fourni avec un générateur de code nommé « SourceCafe » de la compagnie EJD Technologies (EJDTechnologies, 2006). Tout comme le système précédent, notre outil a été mis en échec par une classe qui est la seule à implémenter certains PSCs d'opération, ici, la classe `SalesOrderItem` avec les PSCs d'opération `hashcode` et `equals`. Ceux-ci représentent les deux faux négatifs de PSC d'opération.

Quant à lui, le faux positif de PSC d'attribut est causé par la détection des clones et la règle 5.2. Un groupe de clones a été créé contenant seulement les classificateurs `SalesOrderItemsModel` et `SalesOrderModel` pour leur opération `toString` et donc un PSC de classificateur est créé avec seulement ces deux classificateurs dans son

groupe. Comme ces deux classificateurs partagent un attribut nommé `orderId`, celui-ci devient donc un PSC d'attribut selon la règle 5.4.

Le faux positif de PSC de paramètre représente le paramètre `orderID` de l'opération `load` du classificateur du PSM `SalesOrderDAO`. Il est découvert, car son lien avec un PIC d'attribut n'est pas trouvé. Cela est une conséquence du faux positif de PSC d'attribut `orderID` qui a été décrit précédemment. Celui-ci empêche la découverte du PIC d'attribut `orderID` et donc on ne peut trouver son lien avec ce paramètre.

5.3.4.3 Banking JDBC (BJDBC)

Nous sommes ici en présence d'un premier système qui n'a pas été généré et l'on peut observer que la qualité de la découverte des PSCs est alors un peu moins intéressante.

Ce qui est intéressant dans ce système c'est le fait que même si le système est relativement petit et simple, il complique les choses. Le nombre de classes présent dans son PIM n'est que de cinq et seulement deux de ces classes sont représentées par plus d'un classificateur dans le PSM, ce qui met un peu en échec notre prémisse de répétitivité.

Ainsi, dans le PIM, se trouvent les classes `Account` et `AccountType`. Comme la classe `AccountType` n'est liée qu'à un seul classificateur du PSM et que `Account` est détecté comme un PIC de classe, nos règles détectent `Type` comme représentant un PSC de classificateur et `AccountType` est alors liée au PIC de classe `Account` et au PSC de classificateur `Type`.

On voit aussi que les faux positifs de PSC d'opération et d'attribut augmentent. Ce qui est logique, car notre outil recherche les PSCs d'opération reliés aux faux PSCs de classificateur détectés.

5.3.4.4 Le centre commercial virtuel (VSM)

VSM (Virtual Shopping Mail) est une application implantant un centre commercial virtuel (Oracle, 2006b). Le premier problème avec ce système est le rappel pour les PSCs d'attribut qui est de seulement 5,26 %. Une grande part de ces faux négatifs vient du fait que ces PSCs d'attribut sont implémentés qu'une seule fois dans le PSM. Ce qui va être un problème aussi avec les systèmes suivants. En fait 49 des 57 PSCs d'attribut recherchés, ne sont implémentés qu'une fois. Et comme nos règles se basent beaucoup sur la répétition, ces concepts sont difficilement détectables par celles-ci. Il y a aussi un grand nombre de faux négatifs d'opérations, pour la même raison avec 85 PSCs d'opération sur 129 implémentés une seule fois.

5.3.4.5 Le service de courtage en ligne (OFBS)

OTN Financial Brokerage Service (OFBS) (Oracle, 2006a) est un logiciel qui simule un service de courtage en ligne et où les utilisateurs peuvent échanger des actions, lire des informations financières et contrôler leur portefeuille. Nous voyons que c'est ce système qui obtient le plus faible rappel pour les PSCs d'opération. Encore là, le fait qu'une majorité des PSCs d'opération ne sont implémentés qu'une seule fois ne facilite pas le travail de notre outil.

5.3.4.6 Le EJB Pet Store (PET)

Java Pet Store est un exemple fourni par Sun, illustrant l'application de la technologie J2EE (Singh, Stearns et Johnson, 2002). En regardant le tableau 5.1, on constate que notre outil a rencontré des difficultés avec ce système. Une des problématiques avec l'évaluation de ce système a été de déterminer les PSCs et le PIM que contient celui-ci. En effet même si ce système se veut un exemple de bonnes pratiques, l'architecture qu'il contient est loin d'être claire et en créant nos modèles nous avons éprouvé de la difficulté à la comprendre et encore plus à la représenter en terme de PSCs et de PICs.

Une autre des particularités de ce système est que son PSM contient des classificateurs portant le nom de PSCs de classificateur recherchés. Ce fait empêche la règle 5.1 de détecter ces PSCs, car elle spécifie bien que le nom d'un PSC de classificateur ne peut être le même que le nom d'un classificateur du PSM. Par exemple, un PSC de classificateur recherché se nomme `EJBAction` alors qu'un classificateur du PSM porte ce nom.

5.3.5 Discussion

Les différents essais, que nous venons de présenter, nous aident à esquisser des réflexions et des constatations sur notre approche. Certaines nous font voir les causes des limites de notre approche alors que d'autres sont une piste de réflexion afin d'y remédier.

5.3.5.1 Complexité des algorithmes

La phase de découverte du profil UML de la plate-forme d'implémentation consiste en plusieurs étapes que nous avons exprimées dans un ensemble de règles. Plusieurs facteurs en rapport au nombre d'éléments dans le PSM entrent en compte dans le calcul de la complexité de l'application de ces règles : le nombre de classificateurs (n_c), le nombre d'opérations (n_o), le nombre d'attributs (n_a), le nombre de paramètres (n_p), le nombre d'éléments (classificateurs, attributs, opérations et paramètres) (n_e), le nombre de jetons (n_t) et le nombre d'instructions (n_i) dans le code des opérations. Nous avons choisi d'estimer la complexité dans le pire des cas pour chacune de ces règles.

Trois des algorithmes employés par ces règles pourraient sembler particulièrement couteux. Le premier algorithme est l'application de LSI. Son temps d'exécution est proportionnel au nombre de termes dans la matrice (n_o) et le nombre de dimensions retenues (k) lors de la réduction SVD (Bingham et Mannila, 2001). Ce qui donne dans notre cas $O(n_c \times n_o \times k)$. Car, au pire des cas, il y a dans la matrice une colonne

pour chaque classificateur et un verbe pour chaque opération, si chacun contient un verbe distinct. Le second algorithme qui pourrait être couteux est la détection des opérations clonées. Son temps d'exécution en employant l'outil CCFinder est proportionnel à $O(n \log n)$ où n est le nombre d'éléments à comparer (Kamiya, Kusumoto et Inou, 2001). Ce qui donne dans notre cas $O(n_t \log n_t)$ car la comparaison se fait au niveau des jetons. Le troisième algorithme est l'application de DFA. Son temps d'exécution est proportionnel au nombre d'instructions dans le code des opérations et plus spécifiquement celles de branchement. Ce qui donne dans notre cas une complexité de $O(n_p n_i)$ car une analyse de flux de données est effectuée pour chaque paramètre de chaque opération de chaque classificateur du PSM.

Le tableau 5.2 résume une estimation de la complexité pour chaque règle. En analysant ces chiffres, nous pouvons estimer que le pire des cas lors de l'exécution serait proportionnel au carré du nombre d'éléments dans le PSM, $O(n_e^2)$ ce qui est supérieur au temps d'exécution de LSI ou de la détection des clones. De plus, il est à noter que la complexité peut être augmentée, car les règles 5.2 et 5.3 peuvent être appelées à répétition comme nous l'avons expliqué dans ce chapitre. Dans le pire des cas cette répétition pourrait être exécutée n_c fois, mais en pratique, ce n'est jamais plus que quelquefois. Donc, nous pouvons estimer la complexité de cette phase entre $O(n_e^2)$ et $O(n_c \times n_e^2)$, mais très probablement le plus près de la borne inférieure.

5.3.5.2 La répétitivité de l'implémentation des PSCs

Nos règles sont basées pour beaucoup sur l'idée qu'un PSC appartenant à une plate-forme d'implémentation est implémenté de façon répétitive ou semi-répétitive dans le code source d'un système qui adopte cette plate-forme. Les deux premiers systèmes analysés nous prouvent que c'est clairement le cas dans des systèmes générés, comme il était facile de l'imaginer. Par contre, l'on voit qu'avec des systèmes bénéficiant d'une architecture moins uniformément implémentée ou même sans une vraie

architecture, c'est plus difficile. La répétitivité peut signifier deux choses : uniformité et pluralité.

Tableau 5.2 Complexité des règles de la phase de découverte du profil UML

Règle	Complexité	Motivation (le pire des cas)
5.1	$O(n_c^2 + n_o^2)$	Si LSI crée un groupe pour chaque paire possible de classificateurs du PSM et si chaque paire d'opérations est clonée entre elles.
5.2	$O(n_c)$	Si chaque couple de classificateurs représente un PSC de classificateur.
5.3	$O(n_c)$	Idem.
5.4	$O(n_a)$	Si chaque classificateur du PSM est lié à un PIC de classe, alors une itération sur chaque attribut du PSM est effectuée.
5.5	$O(n_o\ n_a)$	Si chaque classificateur est lié à un PIC de classe, alors une itération sur chaque attribut pour chaque opération du PSM est effectuée.
5.6	$O(n_c^2\ n_o)$	Le cas se produit si LSI crée un groupe par paire de classes et qu'il y a un verbe par opération.
5.7	$O(n_o^2)$	Si un groupe de clones par paire d'opérations est créé.
5.8	$O(n_p\ n_i)$	Pour chaque paramètre, une analyse de flux de données est effectuée.
5.9	$O(n_e^2)$	S'il y a un PIC pour chaque élément du PSM.
5.10	$O(n_e^2)$	Idem.
5.11	$O(n_e^2)$	Idem.
5.12	$O(n_e^2)$	Idem.
5.13	$O(n_e^2)$	Idem.
5.14	$O(n_e)$	Au pire des cas, il y aura un élément dans le profil pour chaque élément du PSM.

Quand on parle de pluralité, l'exemple de VSM avec ses 49 PSCs d'attribut sur 57 implémentés une seule fois en démontre la problématique. Or l'absence de pluralité peut venir de deux raisons. La première raison vient du fait que le concept peut se retrouver de façon unique dans le code source d'un système l'implémentant. Un exemple courant de cas de figure est les classes utilitaires. La deuxième raison peut venir du fait que c'est la nature du système analysé (par exemple, si le PIM est petit) qui a fait en sorte que l'implémentation n'est faite qu'une seule fois. Ces deux raisons nous poussent à penser qu'avec un gros système, l'impact de ses deux raisons sur les

résultats serait moindre de ce que nous avons eu avec nos systèmes de test. Quant à l'uniformité, il s'agit du fait qu'un PSC soit toujours représenté de la même manière dans le code source. Une variation dans l'implémentation peut venir d'une différence dans le nommage du PSC ou dans son implémentation. C'est pourquoi nous avions décidé d'utiliser des techniques tels que la détection de clones et l'usage de LSI afin de remédier au problème d'absence d'uniformité. Dans les deux prochaines sous-sections, nous allons discuter de l'utilité de ces deux techniques.

5.3.5.3 La détection de clones

Comme le démontre le tableau 5.3 un grand nombre de groupes de clones sont détectés pour chaque système. Surtout comparativement au nombre de groupes détectés par LSI.

Tableau 5.3 Nombre de groupes découverts par LSI et détection de clones

	Jouet	SGV	BJDBC	VSM	OFBS	Pet
Nombre de groupes LSI	7	5	5	15	6	17
Nombre de groupes de clone	35	160	234	387	329	341

D'abord avec un grand nombre, nous augmentons la chance de découverte de faux positifs. Par exemple, nous avons choisi de créer un groupe de PSCs de classificateur pour chaque groupe de classificateurs partageant une opération clonée. Avec le système SGV nous avons vu que cela porte à la découverte d'un faux PSC d'opération. De même, il est possible que nous rations la découverte d'un PSC si l'information représentée par les groupes de clones n'est pas exploitée adéquatement. Ce qui nous mène à deux voies à explorer pour mieux exploiter la détection des clones.

La première voie et l'amélioration de la création des groupes de clones. La création des groupes de clones présente plusieurs pistes à explorer. Pour le moment, nous groupons les classificateurs possédants du code cloné dans leur opération. Mais

d'autres pistes sont à explorer comme la taille minimale du code cloné entre deux classificateurs, si des opérations doivent être clonées en entier ou en partie ou le type de clones recherchés.

La seconde voie est de mieux exploiter la signification des groupes de clones. Nous groupons les classificateurs qui partagent du code faisant partie d'une même classe de clones en supposant que cette classe de clones représente à la fois un PSC de classificateur et un PSC d'opération. Il faudrait d'abord mieux spécifier de quel type de PSC il s'agit : classificateur, opération ou autre. Les classificateurs d'un groupe liés à un même PSC de classificateur devraient partager du code commun, mais du code commun peut être partagé par des classificateurs qui ne sont pas liés à un même PSC de classificateur comme l'indique l'exemple cité précédent de `toString`. Aussi quand les classificateurs possèdent un patron commun dans leur nom, nous identifions ce PSC de classificateur par ce nom, sinon nous ignorions comment l'identifier.

5.3.5.4 LSI

Nous avons exploré l'utilisation de LSI afin de lutter contre le problème de manque d'uniformité de nommage des identificateurs liés aux PSCs, notamment des problèmes de synonymies. Nous voyons dans le tableau 5.3 que le nombre de groupes détectés par l'approche LSI est moins élevé que ceux détectés par l'approche de détection de clones.

Avec LSI, il y a la même problématique qu'avec les clones, celle de détecter la signification en termes de PSC d'un groupe de classificateurs détecté. Nous voyons dès le premier exemple que LSI nous aide à améliorer la découverte en nous aidant à trouver le PSC d'opération `findAll`. Mais dans ce cas-ci, il ne s'agit pas de problème de nommage, car les verbes contenus dans ces trois classificateurs sont les mêmes.

Un autre exemple où LSI a bien performé, c'est qu'il a permis de regrouper les classificateurs dont le nom se termine par `TD` et `TransitionDelegate` dans le système PET. Mais on ne peut pas dire qu'il résout directement un problème de synonymie de la façon dont nous le désirons, c'est-à-dire dans le nom des opérations.

En analysant les matrices de cooccurrences, on observe que peu de problèmes de synonymes ont été réglés. Par contre, l'on constate que l'analyse regroupe ensemble des classificateurs possédant des opérations ayant des verbes communs dans leur nom. Peut-être que c'est plus l'analyse TF/DF qui fait les corrections. Il serait aussi intéressant de voir comment un treillis de concept s'en serait tiré dans le même contexte.

Une problématique que nous avons eue dans l'utilisation de LSI est la détection des verbes que nous plaçons dans la matrice des termes. Par exemple si le nom d'opération contient le patron `message` ou `order`, ces noms peuvent être des verbes ou des noms. Une solution trouvée a été de ne pas considérer le nom du constructeur dans la recherche des verbes, car il est rare qu'un classificateur contienne un verbe dans son nom, cette restriction a réglé la plupart des occurrences de ce problème.

On peut aussi s'interroger sur la pertinence de n'utiliser que le nom des verbes contenu dans le nom des opérations dans la matrice de cooccurrence. Ce choix a été fait sur l'idée que les PSCs d'opération devraient souvent être reliés à un verbe. Cela avait pour but de permettre d'utiliser LSI en cherchant à placer dans la matrice des termes liés au PSC. Un défaut de cette idée est que le nombre de termes s'en trouve réduit réduisant l'efficacité de la découverte.

Aussi on pourrait imaginer d'utiliser LSI de façon inverse, c'est-à-dire ne placer que les noms dans la matrice afin de rechercher des PICs au lieu de PSCs.

5.3.5.5 L'enchainement des erreurs et l'interactivité

Comme nous le démontrons dans l'analyse des résultats de BJDBC avec les faux positifs de PSCs d'attribut et d'opération liés aux faux positifs de PSCs de classificateur, une erreur causée par la découverte erronée d'un PSC par une règle peut amener d'autres erreurs en cascade dans les règles suivantes. Un autre exemple de cette problématique est que comme nous l'avons constaté, la découverte de faux PSCs de classificateur tend à causer la découverte de faux PSCs d'opération et PSCs d'attribut qui leurs sont reliés. Évidemment, l'idéal est d'éviter ces erreurs, ce qui est notre but premier.

Une idée qui vient à l'esprit serait un mécanisme permettant une certaine interaction afin d'indiquer au système les erreurs afin qu'il reprenne ses recherches de PSCs aux endroits appropriés. C'est d'ailleurs ce que fait automatiquement l'algorithme en rappelant l'étape de filtrage des PSCs quand il découvre de nouveaux PSCs ou des PSCs inutiles. Mais cette interaction pourrait aussi prendre la forme d'un mécanisme d'interaction entre le système et un expert où ce dernier pourrait indiquer après l'exécution de chaque règle du système ses erreurs. Dans le cas de faux positifs, le système devra ignorer les PSCs faussement détectés indiqués par l'expert et dans le cas de faux négatifs ajouter aux PSCs détectés ceux que l'expert suggère. Bien entendu, un tel système devra afficher de façon efficace et intuitive ses résultats, expliquer comment ils ont été découverts et offrir un mécanisme convivial pour les modifier.

5.3.5.6 Règles de nommage des identificateurs

Nous prenons parti que le nom des identificateurs dans le code source d'un système analysé peut donner une idée à la fois du PSC implémenté et le cas échéant du PIC implémenté. Si nous nous limitions aux EJBs nous pourrions supposer que le nom du PIC est toujours suivi du nom du PSC dans le nom d'un classificateur et l'inverse

dans celui d'une opération. Dans notre approche, nous n'avons pas fait ce genre d'hypothèses, mais en le faisant nous augmenterions la qualité des résultats comme dans le cas d'`Exception` de BJDBC. On pourrait alors imaginer que le système tente de détecter ces patrons de nommage ou que ceux-ci soient fournis en entrée.

Nous avions tenté d'appliquer une règle tentant de résoudre en partie cette problématique. Elle était basée sur l'idée que si le nom d'un PSC de classificateur ne se retrouve pas toujours à la même position dans le nom d'un classificateur du PSM, on est devant l'une des deux situations : soit il ne s'agit pas d'un PSC de classificateur soit nous sommes devant un problème de nommage (nommage non uniforme). On présageait le premier cas en énonçant la règle comme ceci : « Chaque PSC de classificateur dont le nom ne se trouve pas seulement au début, au milieu ou à la fin du nom d'un classificateur du PSM est retiré de la liste des PSCs. » Malheureusement sous cette forme elle apportait plus de problèmes que de solutions. Mais il aurait été sans doute possible de la raffiner.

5.4 SYNTHÈSE

Nous avons présenté dans ce chapitre notre algorithme qui permet la découverte d'un profil UML de la plate-forme d'implémentation à partir du code source d'un système légataire. Par la suite, nous avons présenté le protocole que nous avons suivi lors de notre processus de validation de notre algorithme. Nous avons aussi fourni les résultats de ce processus. Nous avons conclu ce chapitre par une discussion sur des éléments de notre algorithme et des pistes d'améliorations à suivre.

CHAPITRE VI

DÉCOUVERTE DU MODÈLE INDÉPENDANT DE PLATE-FORME

Dans les chapitres précédents, nous décrivons la problématique de la modernisation, l'ingénierie dirigée par les modèles, l'état de l'art entourant ces sujets, un aperçu global de notre approche de modernisation et sa première phase qui est la découverte du profil UML de la plate-forme d'implémentation.

Dans ce chapitre, nous présentons la phase de la découverte du modèle indépendant de plate-forme qui est le second des trois modèles que nous découvrons dans le processus de modernisation. Nous donnons d'abord une vue générale de cette phase avant de l'expliquer en détail à l'aide de règles et d'exemples. Nous parlons ensuite du processus de validation et de ses résultats et nous terminons par une discussion sur ces résultats.

6.1 VUE GÉNÉRALE

La phase de la découverte du modèle indépendant de plate-forme est composée de deux sous-phases principales. La première sous-phase est la découverte du modèle indépendant de plate-forme et utilise le PSM du système légataire et le profil UML de la plate-forme d'implémentation. La seconde sous-phase est la sérialisation de ce PIM dans un modèle XMI. Comme la seconde sous-phase ne constitue pas de défi particulier et que notre principale contribution se situe dans la première sous-phase, nous allons décrire celle-ci en détail dans la suite de ce chapitre.

6.2 SOUS-PHASE 1 : DÉCOUVERTE DU MODÈLE INDÉPENDANT DE PLATE-FORME

Dans cette section nous décrivons notre sous-phase de découverte des concepts indépendants de plate-forme à l'aide de règles et d'exemples d'applications de celles-ci sur le système jouet décrit au chapitre 4 et pour certaines sur le système Banking JDBC présenté en appendice A. Comme l'illustre la figure 6.1, cette sous-phase est divisée en sept étapes et débute par la découverte des PICs de classe. Si cette première étape amène la découverte de nouveaux PSCs alors une étape de « raffinement » du profil de la plate-forme est appelée, qui consiste à réexécuter la découverte de ce profil à partir de la règle 5.2. Sinon les étapes suivantes sont la découverte des PICs d'attribut et la découverte des PICs d'opération. Ensuite, une étape de filtrage des PICs de classe est appelée, qui peut rappeler l'étape de « raffinement » du profil de la plate-forme. La dernière étape est la découverte des PICs de relation représentant les relations entre les PICs découverts. Dans ce qui suit, nous allons détailler six de ces sept étapes. Les règles 5.12-5.13 de l'étape de « raffinement » du profil de la plate-forme sont décrites dans le chapitre 5.

6.2.1 Étape 1.1 : Découverte des PICs de classe

La première étape découvre les PICs de classe du système en recherchant les mots du vocabulaire du PSM les représentant. Cette étape utilise les règles 6.1 à 6.3. La règle 6.1 est motivée par le fait que les noms des classificateurs du PSM sont composés de mots venant du domaine du problème (donc du PIM) et de la solution (donc du PDM) et qu'en filtrant les noms des PSCs de classificateur de ces derniers, il est possible de découvrir ceux des PICs de classe.

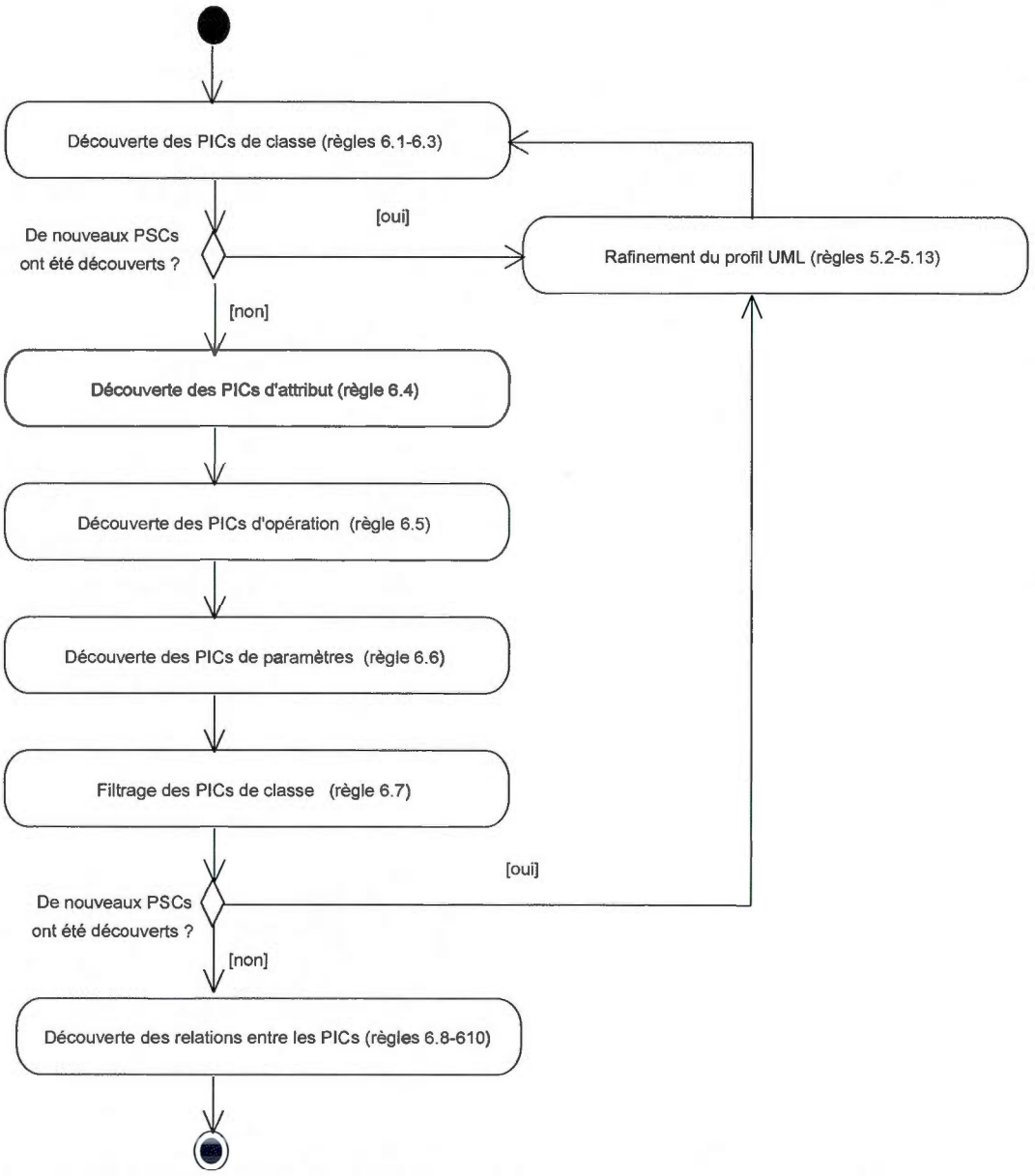

Figure 6.1 La sous-phase de découverte du modèle indépendant de plate-forme

La règle 6.2, quant à elle est motivée par le fait que l'ensemble des PSCs de classificateur du profil UML de la plate-forme d'implémentation donné en entrée est possiblement incomplet. Elle cherche de nouveaux PSCs de classificateur en examinant les classificateurs du PSM qui ne sont encore liés à aucun PSC de classificateur. Si le nom de ces classificateurs contient celui d'un PIC de classe alors

possiblement, il contient aussi le nom d'un PSC de classificateur non encore découvert. Si de nouveaux PSCs sont découverts ici, il est utile de réactiver la découverte du profil en l'enrichissant de ces PSCs et d'ensuite recommencer la découverte du PIM avec ce profil enrichi.

La règle 6.3 procède de la même façon que la règle 6.2 à la différence que cette fois-ci, nous cherchons à identifier des PICs de classe non encore découverts.

Règle 6.1

Un classificateur `C` du PSM est lié à un PIC de classe `P` dont le nom est celui de `C` auquel toute occurrence du nom d'un PSC de classificateur a été retirée. Ce lien existe à condition que `P` soit lié à plus d'un classificateur du PSM.

Dans notre système jouet, nous découvrons les PICs de classe suivants liés à des classificateurs du PSM : `Account`, `BankManager`, `BankTransaction`, `CheckingAccount`, `Customer`, `Deposit`, `SavingAccount`, et `Withdrawal`. Par exemple, nous découvrons que le PIC de classe `Account` est lié à au moins deux classificateurs du PSM (`AccountBean` et `AccountHome`) après avoir retiré de leur nom les occurrences du nom des PSCs de classificateur `Bean` et `Home`.

Règle 6.2

Chaque classificateur `C` du PSM qui n'est lié à aucun PSC de classificateur et dont le nom contient celui d'un PIC de classe `P` est lié à celui-ci. Un nouveau PSC de classificateur est créé et nommé en retirant du nom de `C` celui de `P`. Le nom de ce nouveau PSC ne doit pas être le même que celui d'un PIC de classe. Si de nouveaux PSCs ont été détectés, l'approche doit reprendre à la règle 6.1 après avoir relancé la découverte du profil avec ces nouveaux PSCs à la règle 5.2.

Dans notre système jouet, cette règle découvre un nouveau PSC de classificateur nommé `Session` grâce au classificateur orphelin `BankManagerSession` dont le nom contient celui du PIC de classe nommé `BankManager`.

Règle 6.3

Chaque classificateur `C` du PSM qui n'est lié à aucun PIC de classe et dont le nom contient celui d'un PSC de classificateur `P` est lié à un nouveau PIC de classe. Ce nouveau PIC est nommé en retirant du nom de `C` celui de `P`, à condition que ce nom ne contienne pas le nom d'autres PSCs de classificateur.

Dans notre système jouet, si la classe du PIM `Account` n'implémentait que le PSC de classificateur `Util` alors ce PIC n'aurait pas été découvert par la règle 6.1. La règle découvrirait alors ce PIC grâce au classificateur du PSM `AccountUtil`.

6.2.2 Étape 1.2 : Découverte des PICs d'attribut

La seconde étape de la découverte des PICs a pour objectif de découvrir, pour chaque PIC de classe, les PICs d'attribut qui lui sont liés en recherchant les mots du vocabulaire du PSM susceptibles de les représenter. Cette étape est basée sur la règle 6.4.

Règle 6.4

Un attribut d'un classificateur du PSM lié à un PIC de classe est lié à un PIC d'attribut homonyme si un attribut du même nom n'appartient pas à tous les classificateurs de l'un des sous-groupes d'un PSC de classificateur en lien avec le critère d'implémentation d'une interface ou d'héritage d'un classificateur.

Dans notre système jouet, l'attribut `accountNumber` du classificateur du PSM `AccountData` lié au PIC de classe `Account` est quant à lui lié au PIC d'attribut `accountNumber`, car aucun sous-groupe d'un PSC de classificateur ne voit tous ses classificateurs partager un attribut nommé « `accountNumber` ». Cette règle découvre

également que l'attribut `cachedRemoteHome` du classificateur du PSM `AccountUtil` lui aussi lié au PIC de classe `Account` n'est pas un PIC d'attribut, car tous les sous-groupes du PSC de classificateur `Util` contiennent un attribut nommé « `cachedRemoteHome` ».

6.2.3 Étape 1.3 : Découverte des PICs d'opération

La troisième étape de la sous-phase de découverte du PIM a comme objectif d'identifier, pour chaque PIC de classe, les PICs d'opération qui lui sont liés en recherchant les mots du vocabulaire du PSM susceptibles de les représenter. Cette étape est représentée par la règle 6.5.

Règle 6.5

Une opération d'un classificateur du PSM lié à un PIC de classe est liée à un PIC d'opération homonyme si une opération du même nom n'appartient pas à tous les classificateurs de l'un des sous-groupes d'un PSC de classificateur en lien avec le critère d'implémentation d'une interface ou de l'héritage d'un classificateur et que le nom de cette opération n'est pas lié avec un PSC d'opération. Toute occurrence d'un nom d'un PSC dans le nom et le type de chaque paramètre de cette opération est enlevée.

Dans notre système jouet, puisque tous les classificateurs du groupe lié au PSC de classificateur `Bean` ont une opération nommée `ejbActivate`, cette opération n'est pas un PIC d'opération lié au PIC de classe `Account`. Par contre, l'opération nommée `saveTransaction` du classificateur du PSM `BankManager` représente un PIC d'opération lié au PIC de classe `BankManager`, car aucun des groupes liés au PSC `PICClass` ne voit tous ses membres avoir une opération homonyme.

6.2.4 Étape 1.4 : Découverte des PICs de paramètre

La quatrième étape de la découverte du PIM a comme objectif d'identifier, pour chaque PIC d'opération, les PICs de paramètre qui lui sont liés en recherchant les mots du vocabulaire du PSM susceptibles de les représenter. Cette étape est représentée par la règle 6.5.

Règle 6.6

Un paramètre `P1` d'une opération `O1` d'un classificateur `C1`du PSM, où `O1` est lié à une opération homonyme `O2` d'une classe `C2` du PIM, représente un PIC de paramètre homonyme si `P1` est lié à un élément du PIM, selon la première des conditions suivantes s'appliquant :

- s'il existe un attribut `A1` de `C1` où `A1` est lié à un attribut `A2` (respectivement une extrémité d'association `AE`) de `C2` et `A1` est relié à `P1` par l'analyse de flux de données, alors `P1` est lié à `A2` (respectivement `AE`);
- s'il existe un attribut `A` (respectivement une extrémité d'association `AE`) de `C2` dont le nom est le patron le plus long figurant dans le nom de `P1`, alors `P1` est lié à `A` (respectivement `AE`);
- si le nom de `C2` est le patron le plus long figurant dans le nom de `P1`, alors `P1` est lié à `C2`.

Dans le système Banking JDBC, le paramètre `amt` de l'opération `credit` du classificateur du PSM `Account` lié au PIC d'opération `credit` du PIC de classe `Account` devient un PIC de paramètre. Car il est lié par DFA à l'attribut `balance` de la classe du PIC `Account`.

6.2.5 Étape 1.5 : Filtrage des PICs de classe

Le fait qu'un PIC de classe qui a été détecté ne soit pas lié à des PICs d'attribut ou à des PICs d'opération est un indice qu'il ne s'agit probablement pas d'un vrai PIC de

classe. La règle 6.7 cherche aussi avec ces PICs de classe, d'autres PICs de classe et potentiellement de nouveaux PSCs de classificateur.

Règle 6.7

Un PIC de classe qui n'est lié à aucun PIC d'attribut ou d'opération et dont le nom contient le nom d'un autre PIC de classe est retiré du PIM. Le nom de chacun des classificateurs du PSM qui lui étaient liés est analysé afin de trouver de nouveaux PSCs de classificateur en cherchant le patron le plus long qui ne contient pas le nom de PSC de classificateur, de PIC de classe et de classificateur du PSM du classificateur. Si de nouveaux PSCs ont été détectés, l'approche doit reprendre à la règle 5.2 après avoir relancé la découverte du profil avec ces nouveaux PSCs.

Dans le système Banking JDBC, le PIC de classe `ReuseException` est enlevé, car il n'est lié à aucun PIC d'attribut ou PIC d'opération.

6.2.6 Étape 1.6 : Découverte des relations entre les éléments du PIM

La dernière étape de la sous-phase de découverte du PIM consiste à découvrir les relations existantes entre les PICs. Cette étape est basée sur les règles 6.8, 6.9 et 6.10.

Règle 6.8

Si le type d'un PIC d'attribut `A` lié à un PIC de classe `C1` est un autre PIC de classe `C2` alors ce PIC d'attribut est remplacé par une association entre `C1` et `C2`.

Dans notre système jouet, le PIC attribut `customer` lié au PIC de classe `Account` est du type du PIC de classe `Customer`, alors une association entre ces deux PICs de classe est créée et `customer` n'est plus un PIC attribut lié à la classe `Account`.

Règle 6.9

S'il existe une relation d'héritage entre un classificateur `C1` du PSM relié à un PIC de classe `P1` et un autre classificateur `C2` du PSM relié à un PIC de classe `P2`, il y a une relation d'héritage entre `P1` et `P2` dans le PIM.

Dans notre système jouet, une relation d'héritage est créée entre le PIC de classe `CheckingAccount` et le PIC de classe `Account`, car le classificateur du PSM `CheckingAccount` qui est lié au PIC de classe `CheckingAccount` hérite du classificateur du PSM `Account`. Ce dernier classificateur étant lié au PIC de classe `Account`.

Règle 6.10

S'il existe une exception `E` jetée par une opération d'un classificateur `C1` du PSM lié à un PIC de classe `P2` et le type de `E` est celui d'un classificateur `C2` du PSM lié à un PIC de classe `P2` alors une relation de dépendance est créée où `P1` est la source et `P2` est la cible.

Dans le système JDBC, la classe du PIM `Account` a une dépendance de la classe du PIM `InsufficientFundsException`, car son classificateur lié dans le PSM `AccountTransaction` possède la méthode `transfer` qui jette une exception de ce type.

6.3 VALIDATION DU TRAVAIL

6.3.1 Cadre de validation

Le cadre de validation utilisé est le même que celui décrit à la section 5.3.1 à la différence qu'au lieu d'utiliser pour la validation une liste de PSCs, nous donnons au système le PIM du système légataire que nous avons créé avec l'outil Visual

Paradigm[8]. Une remarque importante quant à la qualité de nos résultats, est que le profil de la plate-forme d'implémentation employé est celui découvert dans le processus automatisé présenté au chapitre précédent. Nous avons choisi de procéder de cette façon, car les deux processus sont très reliés comme nous l'avons déjà expliqué.

Mais il est évident que si nous avions fourni une liste de PSCs idéals à l'outil nous aurions obtenu des résultats forcément meilleurs.

6.3.2 Résultats

Le tableau 6.1 présente les résultats obtenus pour la découverte des PICs des six logiciels analysés. Nous revenons ensuite plus en détail pour chacun de ces logiciels dans le reste de cette section en analysant des points spécifiques de leur résultat.

6.3.2.1 Le système jouet (Jouet)

Notre système jouet nous donne d'excellents résultats, comme nous pouvions nous y attendre. Les deux faux positifs de PIC d'opération sont là, car ceux-ci n'ont pas été détectés comme PSCs d'opération alors qu'ils le sont. Ils sont donc retenus comme des PICs d'opération. Il s'agit des opérations `setSessionContext` et `unsetSessionContext` de `BankManagerSession` qui représentent des PSCs et non des PICs. Mais, ils ne sont implémentés qu'une seule fois et ils ne sont donc pas détectables comme PSCs et deviennent donc des PICs.

6.3.2.2 Le système de gestion des ventes (SGV)

Avec ce système, il y a trois problèmes :

- le faux négatif de PIC d'attribut qui est relié au faux positif de PSC d'attribut `orderId` qui l'empêche d'être détecté comme un PIC d'attribut;

[8] http://www.visual-paradigm.com/

- les deux faux positifs de PIC d'opération qui quant à eux, sont reliés aux faux négatifs de PSC d'opération `hashcode` et `equals` qui les font détecter comme des PICs d'opération;
- les trois faux négatifs de PIC d'association qui viennent du fait que même si ces associations existent bien dans le PIM et dans la logique de l'application, elles ne peuvent être détectées par nos règles. Car ces associations prennent la forme d'un attribut dans la classe qui est associé à l'autre. Par exemple dans le PIM, la classe `SalesOrder` est liée à `Customer`, mais dans le PSM cette association se concrétise par l'attribut `customerID` qui est de type entier, ce qui fait que ce lien ne peut être détecté par nos règles. Ce problème se produit régulièrement dans les autres systèmes analysés;
- les deux faux positifs de PIC d'attribut sont quant à eux reliés aux faux négatifs de PIC d'association car ils représentent des attributs représentant ces associations.

6.3.2.3 Banking JDBC (BJDBC)

Comme nous le mentionnons en commentant les résultats de la découverte des PSCs de classificateur, seulement deux des classes de son PIM sont implémentées plus de deux fois et c'est précisément celles-ci qui ont été trouvées comme vrai positif de PIC de classe. Les autres 3 faux négatifs de PIC de classe sont donc les autres classes du PIM non détectées. Les 16 faux positifs de PIC d'opération comme la majorité des faux PICs d'attribut et de paramètre sont liés à de faux PICs de classes. Quant aux 6 faux positifs de PIC de classe :

Tableau 6.1 Résultat de la découverte des PICs

	Jouet	SGV	BJDBC	VSM	OFBS	Pet
Nombre de lignes de code	1124	1091	2076	3198	3082	6190
Nombre de classificateurs	47	29	67	89	74	209
Nombre d'attributs	55	53	130	146	132	411
Nombre d'opérations	430	207	375	503	427	1049
VP PIC de classe	8	4	2	14	10	6
FP PIC de classe	0	0	6	16	4	16
FN PIC de classe	0	0	3	5	5	31
Rappel PIC de classe	100,00	100,00	40,00	73,68	66,67	16,22
Précision PIC de classe	100,00	100,00	25,00	46,67	71,43	27,27
VP PIC d'attribut	11	21	5	42	24	4
FP PIC d'attribut	0	2	20	44	11	82
FN PIC d'attribut	0	1	3	22	25	79
Rappel PIC d'attribut	100,00	95,45	62,50	65,63	48,98	4,82
Précision PIC d'attribut	100,00	91,30	20,00	48,84	68,57	4,65
VP PIC d'opération	1	0	4	16	10	1
FP PIC opération	2	2	16	36	19	73
FN PIC opération	0	0	0	77	44	17
Rappel PIC opération	100,00	100,00	100,00	17,20	18,52	5,56
Précision PIC opération	33,33	0,00	20,00	30,77	34,48	1,35
VP PIC de paramètre	0	0	3	0	1	0
FP PIC de paramètre	0	0	1	7	0	4
FN PIC de paramètre	0	0	5	30	46	19
Rappel PIC de paramètre	100,00	100,00	37,50	0,00	2,13	0,00
Précision PIC de paramètre	100,00	100,00	75,00	0,00	100,00	0,00
VP PIC d'association	4	0	1	1	2	3
FP PIC d'association	0	0	2	5	9	7
FN PIC d'association	0	3	2	13	14	106
Rappel PIC d'association	100,00	0,00	33,33	7,14	12,50	2,75
Précision PIC d'association	100,00	100,00	33,33	16,67	18,18	30,00
VP PIC d'héritage	4	0	0	0	0	0
FP PIC d'héritage	0	0	2	10	0	3
FN PIC d'héritage	0	0	0	0	0	0
Rappel PIC d'héritage	100,00	100,00	100,00	100,00	100,00	100,00
Precision PIC d'héritage	100,00	100,00	0,00	0,00	100,00	0,00
VP PIC de dépendance	0	0	0	1	2	0
FP PIC de dépendance	0	0	2	12	0	6
FN PIC de dépendance	0	0	1	0	3	1
Rappel PIC de dépendance	100,00	100,00	0,00	100,00	40,00	0,00
Precision PIC de dépendanc	100,00	100,00	0,00	7,69	100,00	0,00

- il y a d'abord `base`, qui s'il était un vrai positif, serait lié à trois classificateurs du PSM : `BaseEntity`, `BaseFacade`, `BaseHome` qui sont liés à des PSCs de

classificateur réels et à une quatrième qui ajoute le faux positif de PSC de classificateur `Session` avec la classe liée `BaseSession`. Il s'agit de classificateurs purement reliés à la plate-forme qui implémentent des PSCs de classificateur. En fait, ce point soulève un point important : un PSC de classificateur peut aussi implémenter d'autres PSCs de classificateur. Nous reviendrons sur ce point dans notre discussion à la fin de ce chapitre;

- le faux positif PIC de classe `Exception` qui est aussi lié avec deux vrais PSCs d'opération;
- les 4 autres faux positifs de PIC de classe sont découverts en appliquant la règle 6.3. Ces faux positifs sont liés à des classificateurs du PSM dont le nom contient le nom de vrais PSCs de classificateur.

Les faux positifs de PIC d'attribut sont causés par des variations dans le nom des attributs comme `FIRST_NAME` et `firstName` qui sont tous les deux détectés pour la classe `Customer` du PIM. Les autres faux positifs d'attribut et d'opération sont surtout reliés à de faux positifs de PIC de classe. Les trois faux négatifs de PIC d'attribut sont ceux qui seraient reliés au faux négatif de PIC de classe `AccountType`.

Un des deux faux négatifs de PIC d'association se produit, car l'une des ses extrémités n'est pas détectée comme PIC de classe alors que l'autre se produit, car le lien entre les PICs de classe `Customer` et `Account` se concrétise dans le PSM par un `ArrayList` (qui contiendra les `Accounts`), ce qui n'est pas détectable par nos règles. Quant aux faux positifs d'association, ce sont des associations entre de faux PICs de classe.

Le faux négatif de PIC de dépendance vient du fait que le PIC de classe `InsufficientFundsException` n'a pas été détecté et que donc la dépendance de `Account` pour cette classe ne peut être découverte. Les deux faux positifs de PIC de dépendance sont causés par des PICs faussement détectés, et les dépendances vont

vers eux. Les autres problèmes de dépendance sont semblables dans les autres systèmes.

C'est aussi dans ce système que nous voyons les premiers faux positifs d'héritage. Encore une fois, le problème vient d'héritages entre PICs de classe faussement détectés. Ici, la classe de plate-forme `Base` est faussement détectée comme PIC de classe et certains classificateurs du PSM liés à un PIC de classe héritent du classificateur du PSM lié au PIC `Base`, un héritage est alors détecté entre ces PICs de classe. Ce cas se reproduit dans les systèmes qui suivent.

C'est d'abord dans cet exemple que se pointe un problème, la découverte des PICs de paramètre. D'abord, le faux positif est lié à un faux PIC d'opération. Mais, le problème majeur dans la découverte des PICs de paramètre vient du fait que pour détecter ceux-ci, tout comme les PSCs de paramètre, nous cherchons les liens entre les éléments du PIM et ceux-ci. Quand les noms sont identiques, ce n'est pas trop problématique. Par contre pour plusieurs cas c'est plus problématique, par exemple, le lien entre le paramètre `acct` du PIC d'opération `deposit` et la classe du PIM `Account` n'a pu être découvert ni par l'analyse des identificateurs ni par DFA. De même, le paramètre `amt` de l'opération `widthraw` du classificateur du PSM `Account` n'a pas son lien détecté par DFA avec l'attribut `balance` de la classe du PIM `Account` car notre analyse de flux de données devrait être plus élaborée. Encore là, cette problématique se reproduit dans les systèmes qui suivent.

6.3.2.4 Le centre commercial virtuel (VSM)

Ce système a aussi plusieurs faux positifs de PIC d'opération, beaucoup de ces faux positifs sont en lien avec des PICs d'attribut non découverts. Comme ces PICs d'attribut ne sont pas détectés, le lien entre les deux ne peut être découvert. Par exemple, la classe du PIM `Cart`, ne voit pas son attribut `amount` découvert, alors l'opération du PSM `getAmount` ne peut y être liée.

6.3.2.5 Le service de courtage en ligne (OFBS)

Le nombre de faux négatifs de PIC d'association est élevé, car il n'est pas possible de trouver des liens entre des PICs de classe non découverts. Nous voyons aussi que c'est ce système qui a le rappel le plus faible pour les PICs d'attribut (exception faite du système Pet). Pour beaucoup de ceux-ci, ce sont de PICs de classe non détectés qui contiennent plusieurs attributs et donc les PICs d'attribut qui leur sont reliés ne sont pas détectés.

Aussi ce système nous présente un problème intéressant pour lier un paramètre à un élément du PSM qu'une analyse de flux de données ne peut répondre. Par exemple, le paramètre `symbol` de l'opération `getLatestRate` du classificateur du PSM `StockRate` est difficilement associable à un attribut, car il est utilisé seulement dans une requête SQL dans le code de cette opération.

6.3.2.6 Le EJB Pet Store (PET)

Comme, nous pouvions nous y attendre, ce système a obtenu les moins bons résultats des six. En grande partie, car il recevait déjà en entrée un profil UML de mauvaise qualité.

6.3.3 Discussion

Les différents essais, que nous venons de présenter, nous aident à esquisser des réflexions et des constatations sur notre approche. Certaines nous font voir les causes des limites de notre approche alors que d'autres sont une piste de réflexion afin d'y remédier. Certains points de la discussion au sujet de la découverte du profil UML de la plate-forme d'implémentation s'appliquent aussi à cette phase et ne seront donc pas répétés.

6.3.3.1 Complexité des algorithmes

La phase de la découverte du modèle indépendant de plate-forme consiste en plusieurs étapes que nous avons exprimées dans un ensemble de règles. Plusieurs facteurs en rapport au nombre d'éléments dans le PSM entrent en compte dans le calcul de la complexité de l'application de ces règles : le nombre de classificateurs (n_c), le nombre d'opérations (n_o), le nombre d'attributs (n_a), le nombre de paramètres (n_p), le nombre d'import dans le PSM (n_{im}), le nombre de jetons (n_t) et le nombre d'instructions (n_i) dans le code des opérations. Nous avons choisi d'estimer la complexité dans le pire des cas pour chacune de ces règles. Nous avons déjà expliqué la complexité de l'analyse de flux de données dans la section correspondante du chapitre 5.

Le tableau 6.2 résume une estimation de la complexité pour chaque règle. En analysant ces chiffres, nous pouvons estimer que le pire des cas lors de l'exécution serait proportionnel au carré du nombre de classificateurs dans le PSM, $O(n_c^2)$. De plus, il est à noter que la complexité peut être augmentée, car la règle 6.2 et les suivantes peuvent être appelées à répétition comme nous l'avons expliqué dans ce chapitre. Dans le pire des cas cette répétition pourrait être exécutée n_c fois, mais en pratique, ce n'est jamais plus que quelquefois. Donc, nous pouvons estimer la complexité de cette phase entre $O(n_c^2)$ et $O(n_c^3)$, mais très probablement, le plus près de la borne inférieure.

Tableau 6.2 Complexité des règles de la phase de découverte du PIM

Règle	Complexité	Motivation (le pire des cas)
6.1	$O(n_c)$	Pour chaque classificateur du PSM.
6.2	$O(n_c^2)$	Le nombre de PSCs est proportionnel au nombre de classificateurs par le nombre de classificateurs.
6.3	$O(n_c)$	Pour chaque classificateur du PSM.
6.4	$O(n_a)$	Pour chaque attribut du PSM.
6.5	$O(n_o)$	Pour chaque opération du PSM.
6.6	$O(n_c^2)$	Le nombre de PICs de classe est proportionnel au nombre de classificateurs du PSM tout comme le nombre de groupes reliés.
6.7	$O(n_p n_i)$	Pour chaque paramètre, une analyse de flux de données est effectuée.
6.8	$O(n_a)$	Le nombre de PICs d'attribut est proportionnel au nombre d'attributs du PSM.
6.9	$O(n_c^2)$	Pour chaque classificateur du PSM « par » pour chaque classificateur du PSM.
6.10	$O(n_{im} + n_o)$	Proportionnel au nombre d'import et d'opération.

6.3.3.2 Importance de la qualité du profil de la plate-forme

Comme nous l'avons expliqué précédemment, nous avons utilisé le profil de la plate-forme d'implémentation tel que découvert par notre outil par les règles décrites au chapitre 5. Comme les résultats de cette découverte ne sont pas optimaux, les résultats de la phase de découverte des concepts indépendants de plate-forme ne peuvent pas être parfaits. Donc, toute amélioration dans la découverte du profil UML de la plate-forme d'implémentation serait profitable à la phase de découverte des concepts indépendants de plate-forme.

6.3.3.3 Difficulté de cerner des PICs de classe reliés à un unique classificateur du PSM

Nous avons introduit la règle 6.3 afin de détecter les classes du PIM représentées une seule fois dans le PSM. Cette règle fonctionne bien dans plusieurs des cas. Mais elle entraine la découverte de plusieurs faux positifs de PIC de classe comme nous avons pu le constater avec JDBC. Car parfois un classificateur du PSM possède dans son nom le nom d'un PSC de classificateur vrai ou faux positif, mais n'est relié qu'à la plate-forme d'implémentation.

Cette règle ne détecte pas aussi les PICs de classe ne contenant pas le nom d'un PSC de classificateur ou d'un PSC de classificateur non découvert. Il faut donc trouver une technique plus efficace et comportant moins d'effets secondaires afin d'isoler ces PICs de classe.

6.3.3.4 Le nom des éléments du PIM épelé différemment que dans le PSM

Nous avons aussi noté que des éléments du PIM et du PSM correspondant peuvent ne pas être écrits de façon identique, par exemple tout en majuscule quand elle correspond à une constante dans le PSM, mais en minuscule dans le PIM. Le problème est relativement simple à corriger dans les cas simples, comme le cas majuscule et minuscule. Mais on pourrait imaginer aller plus loin en cherchant à détecter les fautes d'orthographe, voir des synonymes.

6.3.3.5 La difficulté de découvrir un PIM plus ou moins existant

Il va de soi qu'une application contient des éléments représentant les objets du problème et de la solution. Malheureusement souvent les choses ne sont pas définies aussi clairement. Dans ce cas, il devient difficile de détecter ce PIM si celui-ci est vague et diffus dans le code source. On pourrait imaginer que dans ce cas la découverte du PIM pourrait faire appel à un expert du domaine afin de corriger ses

résultats, en procédant en quelque sorte à une réingénierie du système en plus d'extraire directement un PIM du système.

6.3.3.6 Des PSCs de classificateur en implémentant d'autres

Nous avons détecté certains faux positifs de PIC de classe, car des PSCs de classificateur peuvent implémenter d'autres PSCs de classificateur. Rappelons le cas du PSC de classificateur `Base` qui implémente les PSCs de classificateur `Entity`, `Facade`, `Home` et `Session`. Les fondements de notre approche font en sorte que ce genre de scénario n'est pas supporté. Dans l'état actuel, nous ne voyons pas d'autres solutions qu'une intervention manuelle pour signaler que `Base` est un PSC de classificateur et non un PIC de classe.

6.4 SYNTHÈSE

Nous avons décrit dans ce chapitre notre algorithme de découverte du PIM d'un code source légataire. Nous avons aussi présenté l'outil que nous avons développé pour implanter notre approche, le protocole que nous avons suivi lors de notre processus de validation et les résultats de notre processus de validation. Nous avons conclu ce chapitre par une discussion sur notre processus de validation et des pistes d'améliorations à suivre.

CHAPITRE VII

DÉCOUVERTE DES MODÈLES DE TRANSFORMATION

Dans les chapitres précédents, nous décrivons la problématique de la modernisation, l'ingénierie dirigée par les modèles, l'état de l'art entourant ces sujets, un aperçu global de notre approche de modernisation et ses deux premières phases qui sont la découverte du profil UML de la plate-forme d'implémentation et la découverte du modèle indépendant de plate-forme.

Dans ce chapitre, nous présentons la phase de découverte des modèles de transformation. Nous donnons d'abord une vue générale de cette étape avant de l'expliquer en détail à l'aide de règles et d'exemples. Nous parlons ensuite du processus de validation et de ses résultats et nous terminons par une discussion sur ces résultats.

7.1 VUE GÉNÉRALE

La phase de découverte des modèles de transformation est divisée en trois sous-phases. La figure 7.1 nous montre les intrants à ces sous-phases tout comme les liens entre elles en plus de les présenter d'une façon graphique. Comme le montre le haut de cette figure, ces sous-phases utilisent comme intrant le PSM représentant le système légataire, le profil UML du modèle de sa plate-forme d'implémentation et son PIM obtenus dans les phases précédentes. L'objectif de cette figure est de faire ressortir les liens entre ces différentes sous-phases.

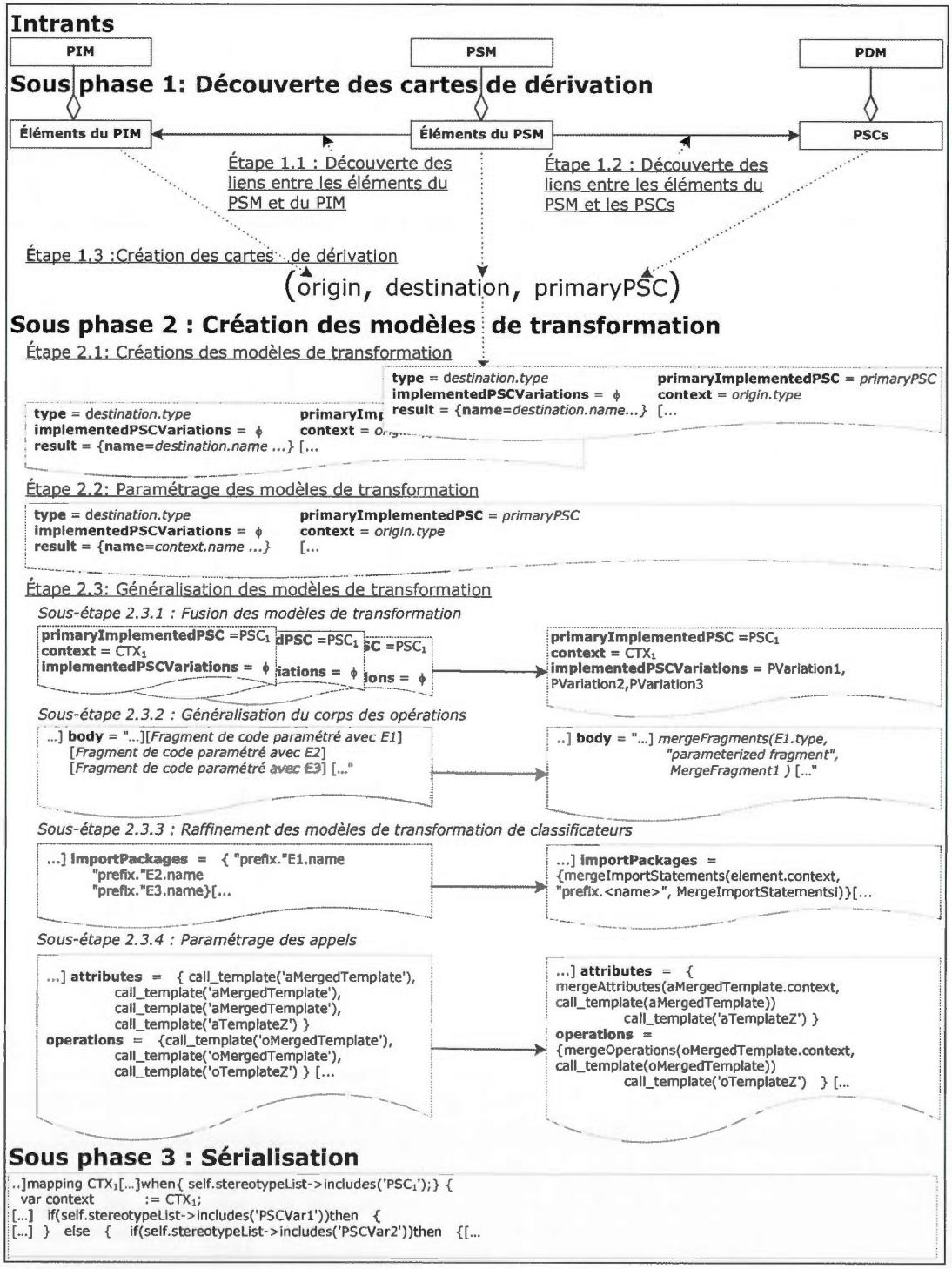

Figure 7.1 Les sous-phases de découverte des modèles de transformation

Les deux figures suivantes détaillent les deux premières sous-phases avec une colonne indiquant les idées clefs et une autre les techniques employées. La figure 7.2 représente la première sous-phase qui est la découverte d'un ensemble de cartes de dérivation afin d'exprimer les liens entre un élément du PSM et un élément du PIM et un PSC. La figure 7.3 représente la seconde sous-phase qui est la découverte d'un ensemble de modèles de transformation paramétrant le code source de la plate-forme d'implémentation du système. Pour illustrer notre approche, nous utilisons le système Banking JDBC présenté à l'appendice A.

7.2 SOUS-PHASE 1 : DÉCOUVERTE DES CARTES DE DÉRIVATION

L'objectif de la sous-phase de découverte des cartes de dérivation est de découvrir un ensemble de cartes de dérivation indiquant pour les éléments du PSM ses liens avec un PSC et avec un élément du PIM, si ces liens existent. Cette sous-phase est composée de trois étapes comme le montrent les points 1.1 à 1.3 de la figure 7.2 :

1.1 découverte des liens entre les éléments du PSM et du PIM;

1.2 découverte des liens entre les éléments du PSM et les PSCs;

1.3 création des cartes de dérivation.

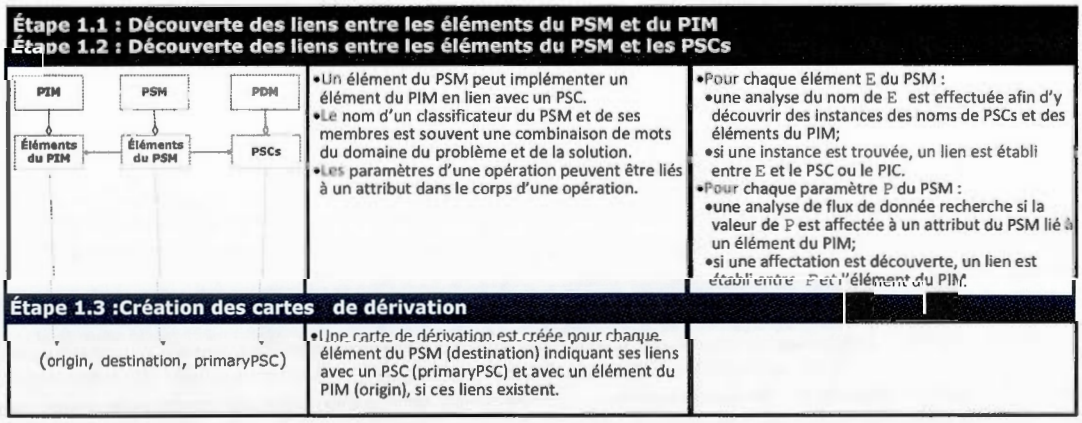

Figure 7.2 La première sous-phase de la phase de découverte des modèles

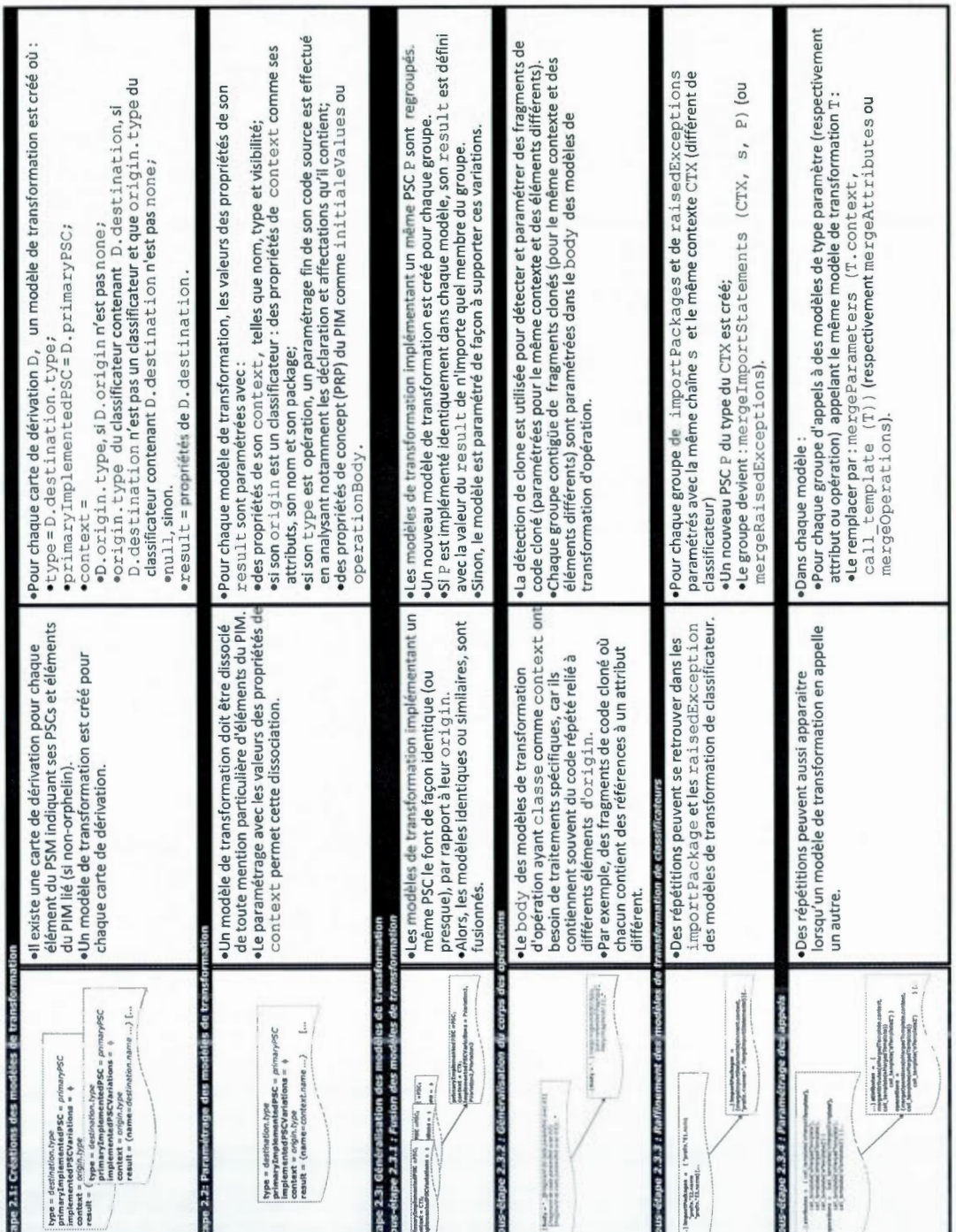

Figure 7.3 La seconde sous-phase de la phase de découverte des modèles

7.2.1 Étape 1.1 : Découverte des liens entre les éléments du PSM et du PIM.

Notre approche est basée sur le fait que des liens existent entre certains éléments du PSM et ceux du PIM. En effet, un PSM représente la mise en œuvre d'une plate-forme d'implémentation pour un PIM et donc les éléments du PIM doivent se refléter dans ceux du PSM. Un élément du PSM est lié à un élément du PIM s'il implémente un concept du domaine métier. Un élément du PSM, qui est uniquement lié à la plate-forme d'implémentation, n'est pas lié à un élément du PIM, et il est alors considéré comme orphelin. Cette étape utilise les règles 7.1 à 7.4. L'idée derrière ces règles est que nous pouvons trouver des indices dans le code source permettant de découvrir ces liens, principalement en effectuant des analyses d'identificateur et de flux de données.

Remarquez que nous ne cherchons pas à découvrir de liens entre les éléments du PSM qui sont des membres d'une classe orpheline, c'est-à-dire ses attributs, opérations et paramètres.

Règle 7.1

Un classificateur `C1` du PSM est lié à la classe du PIM ayant le nom le plus long contenu dans celui de `C1`. Si un tel élément du PIM n'existe pas, `C1` est considéré comme orphelin et il est lié au mot clé `none`.

Dans notre exemple, le classificateur du PSM `AccountEntity` est lié à la classe du PIM `Account` alors que le classificateur du PSM `BaseEntity` est orphelin, car il représente exclusivement une exigence de la plate-forme d'implémentation et qu'il n'existe pas de classe en lien dans le PIM.

Règle 7.2

Le lien entre un attribut `A1` d'un classificateur `C1` du PSM lié à une classe `C2` du PIM et un élément du PIM dépend de la première condition suivante qui s'applique :

- s'il existe un attribut `A2` de `C2` dont le nom est le patron le plus long figurant dans le nom de `A1`, alors `A1` est lié à `A2`;
- s'il existe une extrémité d'association `AE` de `C2` dont le nom de rôle est le patron le plus long figurant dans le nom de `A1`, alors `A1` est lié à `AE`;
- si le nom de `C2` est le patron le plus long figurant dans le nom de `A1`, alors `A1` est lié à `C2`;
- si aucune des conditions précédentes ne s'applique, `A1` est lié au mot clé `none`.

Dans notre exemple, l'attribut `balance` du classificateur du PSM `AccountEntity` est lié à l'attribut `balance` de la classe du PIM `Account` alors que l'attribut `customer` du classificateur du PSM `Account` est lié à l'extrémité d'association `customer` de la classe du PIM `Account`. L'attribut `parent` du classificateur du PSM `AccountNode` est orphelin et par conséquent, il s'agit exclusivement d'une exigence de la plate-forme d'implémentation.

Règle 7.3

Le lien entre une opération `O1` d'un classificateur du PSM `C1` lié à une classe `C2` du PIM et un élément du PIM dépend de la première condition suivante qui s'applique :

- si le nom de `C2` est le patron le plus long figurant dans le nom de `O1`, alors `O1` est lié à `C2`;
- s'il existe une opération `O2` de `C2` dont le nom est le patron le plus long figurant dans le nom de `O1`, alors `O1` est lié à `O2`;
- s'il existe un attribut `A` de `C2` dont le nom est le patron le plus long figurant dans le nom de `O1`, alors `O1` est lié à `A`;
- s'il existe une extrémité d'association `AE` de `C2` dont le nom de rôle est le patron le plus long figurant dans le nom `O1`, alors `O1` est lié à `AE`;

- si aucune des conditions précédentes ne s'applique, O1 est lié au mot clé none.

Dans notre exemple, le constructeur AccountEntity() du classificateur du PSM AccountEntiy est lié à la classe du PIM Account. L'opération credit(id : Identifier , amt : double) du classificateur du PSM Account est liée à l'opération credit (amt : double) de la classe du PIM Account. L'opération getNumber(id : Identifier) du classificateur du PSM AccountEntity est liée à l'attribut nombre de la classe du PIM Account. L'opération create (id : Identifie, t : AccountType, cust : CustomerFacade) du classificateur du PSM AccountEntity est orpheline et par conséquent, elle représente exclusivement une exigence de la plate-forme d'implémentation.

Règle 7.4

Le lien entre un paramètre P1 d'une opération O1 d'un classificateur du PSM C1 lié à une classe C2 du PIM et un élément du PIM dépend de la première condition suivante qui s'applique :

- s'il existe un paramètre P2 d'une opération O2 de C2 dont le nom est le patron le plus long figurant dans le nom de P1, alors P1 est lié à P2;
- s'il existe un attribut A1 de C1 où A1 est lié à un attribut A2 (respectivement une extrémité d'association AE) de C2 et A1 est relié à P1 par l'analyse de flux de données, alors P1 est lié à A2 (respectivement AE);
- s'il existe un attribut A (respectivement une extrémité d'association AE) de C2 dont le nom est le patron le plus long figurant dans le nom de P1, alors P1 est lié à A (respectivement AE);
- si le nom de C2 est le patron le plus long figurant dans le nom de P1, alors P1 est lié à C2;
- si aucune des conditions précédentes ne s'applique, P1 est lié au mot clé none.

Dans notre exemple, le paramètre `AMT` de l'opération `credit (id : Identifier, amt : double)` du classificateur du PSM `AccountEntity` est lié au paramètre `amt` de l'opération `credit (amt : double)` de la classe du PIM `Account`. Le paramètre `t` de l'opération `create (id : Identifier, t : AccountType, cust : CustomerFacade)` du classificateur du PSM `AccountEntity` est lié à l'extrémité d'association `type` de la classe du PIM `Account` parce que l'attribut `type` de `AccountEntity` est lié au paramètre `t` par l'analyse de flux de données. Le paramètre `id` de l'opération `create (id : Identifier, t : AccountType, cust : CustomerFacade)` du classificateur du PSM `AccountEntity` est orphelin et par conséquent, il s'agit exclusivement d'une exigence de la plate-forme d'implémentation.

7.2.2 Étape 1.2 : Découverte des liens entre les éléments du PSM et les PSCs.

La deuxième étape est la découverte des liens entre les éléments du PSM et les PSCs. Cette étape utilise la règle 7.5.

Règle 7.5

Le lien entre un élément `E1` du PSM et un PSC dépend de la première condition suivante qui s'applique :

- si `E1` est lié au mot clé `none` et qu'il existe un PSC `PS` du même type que `E1` où le nom de `PS` est le même que celui de `E1` alors `E1` est lié à `PS`;
- si `E1`est lié à un élément du PIM `E2` et que le nom de `E1` est le même que celui de `E2`, alors :
    - si `E1` est une interface, `E1` est lié au PSC prédéfini `PICInterface`;
    - si `E1` est une classe, `E1` est lié au PSC prédéfini `PICClass`;
    - si `E1` est un attribut, `E1` est lié au PSC prédéfini `PICAttribute`;

  - si E1 est une opération, E1 est lié au PSC prédéfini PICOperation;
  - si E1 est un paramètre, E1 est lié au PSC prédéfini PICParameter.
- si E1est lié à un élément du PIM E2 et que le nom de E1 est différent de celui de E2 et qu'il existe un PSC PS du même type que E1 dont le nom est le patron le plus long figurant dans le nom de E2, alors E1 est lié à PS;
- si aucune des conditions précédentes ne s'applique, E1 est lié au mot clé none.

Dans notre exemple, le classificateur du PSM AccountEntity est lié au PSC de classificateur Entity. L'attribut customer du classificateur du PSM Account est lié au PSC prédéfini PICAttribute. L'opération create (id : Identifier, t : AccountType, cust : CustomerFacade) du classificateur du PSM AccountEntity est liée au PSC d'opération create.

7.2.3 Étape 1.3 : Création des cartes de dérivation

La troisième étape utilise les liens découverts dans les sous-étapes précédentes pour créer les cartes de dérivation. Une carte de dérivation indique pour un élément du PSM à quel élément du PIM et à quel PSC il est lié. Nous définissons une carte de dérivation par un quadruplet (origin, destination, primaryPSC, variationPSCs). Les propriétés origin et destination représentent respectivement des éléments du PIM et du PSM. Les propriétés primaryPSC et variationPSCs décrivent respectivement le PSC primaire et les variantes d'implémentations.

La règle 7.6 est utilisée pour créer les cartes de dérivation. Notez que dans cette règle la valeur de la quatrième propriété prend la valeur ensemble vide (ϕ) pour toutes les cartes de dérivation créées. Ces ensembles vides seront peuplés dans les étapes de la deuxième sous-phase.

Règle 7.6

Pour chaque élément du PSM `E`, une nouvelle carte de dérivation (le nom qualifié du PIC lié à `E`, le nom qualifié de `E`, le PSC liés à `E`, ϕ) est créée.

Dans notre exemple, une carte de dérivation (`com.imaginary.bank.Account`, `com.imaginary.bank.AccountEntity`, `entité`, ϕ) est créée pour le classificateur du PSM `AccountEntity`. Une carte de dérivation (`com.imaginary.bank.Account.customer`, `com.imaginary.bank.Account.CUSTOMER`, `PICAttribute`, ϕ) est créée pour l'attribut `customer` du classificateur du PSM `Account`. Une carte de dérivation (`none`, `com.imaginary.bank.AccountEntity.create(id: Identifier, t: AccountType, cust: CustomerFacade)`, `create`,ϕ) est créée pour l'opération `create (id: Identifier, t: AccountType, cust: CustomerFacade)` du classificateur du PSM `AccountEntity`.

7.3 SOUS-PHASE 2 : CRÉATION DES MODÈLES DE TRANSFORMATION

L'objectif de cette sous-phase est d'extraire un ensemble de modèles de transformation paramétrés représentant le code source de la plate-forme d'implémentation du système légataire. Cette sous-phase est composée de trois étapes comme le montrent les points 2.1 à 2.3 de la figure 7.3 :

2.1 création des modèles de transformation;

2.2 paramétrage des modèles de transformation;

2.3 généralisation des modèles de transformation.

7.3.1 Étape 2.1 : Création de modèles de transformation

Cette première étape crée un modèle de transformation pour chaque carte de dérivation et utilise la règle 7.7.

Règle 7.7

Un modèle de transformation `T (primaryImplementedPSC, implementedPSCsForVariations, context, result)` est créé pour chaque carte de dérivation `D (origin, destination, primaryPSC, variationPSCs)` avec les valeurs suivantes :

- type du modèle = le type de l'élément de destination;
- `primaryImplementedPSC` = `primaryPSC`;
- `implementedPSCsForVariations` = `variationPSCs`;
- la valeur de `context` dépend de la première condition qui s'applique :
    - si `origin` n'est pas `none`, `context` est le type de l'élément dans `origin`;
    - sinon :
        - si `destination` est un attribut ou une opération et que l'`origin` du classificateur contenant `destination` n'est pas `none`, alors `context=class`;
        - si `destination` est un paramètre alors `context = none`.
    - si aucune des conditions précédentes ne s'applique, alors `context = none`.
- `result` = `destination`.

Le tableau 7.1 présente le modèle de transformation de classe créé pour la carte de dérivation (`com.imaginary.bank.Account`, `com.imaginary.bank.AccountEntity`, `entity`, ϕ).

Tableau 7.1 Exemple d'un modèle de transformation de classe

```
primaryImplementedPSC = 'Entity'
implementedPSCsForVariations = ϕ
context = class
result = the class
  name = 'AccountEntity'
  packagename = 'com.imaginary.bank'
  visibility = 'public'
  superClassifier = 'BaseEntity'
  importPackages =
    {
     'com.imaginary.lwp.BaseEntity',
     'com.imaginary.lwp.Identifier',
     'com.imaginary.lwp.SequenceGenerator',
     'com.imaginary.lwp.SequenceException',
     'com.imaginary.lwp.TransactionException',
     'com.imaginary.lwp',
     'java.rmi.RemoteException'
    }
  implementInterfaces = {'Account'}
  attributes = {balance,customer,  number, type}
  operations =
  {
     AccountEntity(),
     create(Identifier id, AccountType t, CustomerFacade cust),
     credit(Identifier id, double amt),
     getBalance(Identifier id),
     getCustomer(Identifier id),
     getNumber(Identifier id),
     getType(Identifier id)
  }
```

Le tableau 7.2 présente le modèle de transformation d'attribut créé pour la carte de dérivation (`com.imaginary.bank.Account.Customer`, `com.imaginary.bank.Account.CUSTOMER`, `PICAttribute`, ϕ).

Tableau 7.2 Exemple d'un modèle de transformation d'attribut

```
primaryImplementedPSC = 'PICAttribute'
implementedPSCsForVariations = ϕ
context = associationEnd
result = the attribute
  name = 'CUSTOMER'
  visibility = 'public'
  type = 'String'
  changeability = 'static final'
  initialValue = 'customer'
```

Le tableau 7.3 présente le modèle de transformation d'opération créé pour la carte de dérivation (none, `com.imaginary.bank.AccountEntity.create (id : Identifier, t : AccountType, cust : CustomerFacade)`, `create`, ϕ).

Tableau 7.3 Exemple d'un modèle de transformation d'opération

```
primaryImplementedPSC = 'create'
implementedPSCsForVariations = ϕ
context = class
result = the operation
  name = 'create'
  visibility = 'public'
  returnType = 'void'
  changeability = ''
  parameters = {'Identifier id','AccountType t','CustomerFacade cust'}
  raiseExceptions = {'TransactionException'}
  body = 'prepareCreate(id);
   try {
    number = (int) SequenceGenerator
    .generateSequence("ACCT_NUM");}
   catch( SequenceException e ) {
    throw new TransactionException
              (e.getMessage());}
   type = t;
   customer = cust;    '
```

Le tableau 7.4 présente le modèle de transformation de paramètre créé pour la carte de dérivation (`com.imaginary.bank.Account.customer`, `com.imaginary.bank.AccountEntity.create.parameter (cust : CustomerFacade)`, `PICParameter`, ϕ).

Tableau 7.4 Exemple d'un modèle de transformation de paramètre

```
primaryImplementedPSC = 'PICParameter'
implementedPSCsForVariations = ϕ
context = associationEnd
result = the parameter
  name = 'cust'
  type = 'CustomerFacade'
  directionKind = ''
```

7.3.2 Étape 2.2 : Paramétrage des modèles de transformation.

Cette étape paramètre les valeurs de la propriété `result` des modèles de transformation créés à l'étape précédente et utilise les règles 7.8 à 7.12. Le

paramétrage d'un modèle de transformation `T (primaryImplementedPSC, implementedPSCsForVariations, context, result)` créé pour une carte de dérivation `D (origin, destination, primaryPSC, variationPSCs)` consiste principalement à paramétrer les occurrences des valeurs des propriétés d'`origin` contenues dans les valeurs des propriétés de `result` avec les propriétés correspondantes de `context`.

Règle 7.8

Les propriétés de `result` du modèle de transformation de classificateur (interface ou classe) `T (primaryImplementedPSC, implementedPSCsForVariations, context, result)` créé pour une carte de dérivation `D (origin, destination, primaryPSC, variationPSCs)` sont paramétrées comme suit :

- `result.name` en substituant chaque occurrence d'`origin.name` par `context.name`;
- `result.packageName` en substituant chaque occurrence d'`origin.name` par `context.name` et d'`origin.packageName` par `context.packageName`;
- `result.superClassifierName` en substituant chaque occurrence d'`origin.superClassifierName` par `context.superClassifierName`, d'`origin.name` par `context.name`, d'`origin.superClassifierPackageName` par `context.superClassifierPackageName` et d'`origin.packageName` par `context.packageName`;
- `result.implementedInterfaceList` en substituant chaque occurrence d'`origin.name` par `context.name` et d'`origin.packageName` par `context.packageName`.
- `result.importPackages` en substituant chaque occurrence de :

- `origin.superClassifierName` par `context.superClassifierName` et d'`origin.superClassifierPackageName` par `context.superClassifierPackageName;`
- `origin.associationEnds['AE'].type.name` par `context.associationEnds['AE'].type.name` et d'`origin.associationEnds['AE'].type.packageName` par `context.associationEnds['AE'].type.packageName` pour chaque extrémité de l'association `AE` de `D.origin;`
- `origin.dependencies['d'].target.name` par `context.dependencies['d'].target.name` et de `origin.dependencies['d'].target.packageName` par `dependencies de context.['d'].target.packageName` pour chaque dépendance `d` de `D.origin`.

- Les valeurs des propriétés de la propriété `result` des attributs et opérations sont paramétrées en appelant chaque modèle de transformation capturant un attribut ou une opération du classificateur de `destination`.

Dans notre exemple, le paramétrage du modèle de transformation décrit dans le tableau 7.1 est donné dans le tableau 7.5.

Tableau 7.5 Exemple d'un modèle de transformation de classe paramétré

```
primaryImplementedPSC = 'Entity'
implementedPSCsForVariations = ϕ
context = class
result = the class
  name = context.name + 'Entity'
  packagename = context.packagename
  visibility = 'public'
  superClassifier = 'BaseEntity'
  importPackages = {
      'com.imaginary.lwp.BaseEntity',
      'com.imaginary.lwp.Identifier ',
      'com.imaginary.lwp.SequenceGenerator',
      'com.imaginary.lwp.SequenceException',
      'com.imaginary.lwp.TransactionException',
      'com.imaginary.lwp',
      'java.rmi.RemoteException'
    }
  implementInterfaces = {context.name}
  attributes = {
      call_template('com.imaginary.bank.AccountEntity.balance'),
      call_template('com.imaginary.bank.AccountEntity.customer'),
      call_template('com.imaginary.bank.AccountEntity.number'),
      call_template('com.imaginary.bank.AccountEntity.type')
   }
  operations = {
   call_template('com.imaginary.bank.AccountEntity.AccountEntity()'),
   call_template('com.imaginary.bank.AccountEntity. create(Identifier id, AccountType t, CustomerFacade cust)'),
   call_template('com.imaginary.bank.AccountEntity.credit(Identifier id, double amt)'),
   call_template('com.imaginary.bank.AccountEntity.getBalance(Identifier id)'),
   call_template('com.imaginary.bank.AccountEntity.getCustomer(Identifier id)'),
   call_template('com.imaginary.bank.AccountEntity.getNumber(Identifier id)'),
   call_template('com.imaginary.bank.AccountEntity.getType(Identifier id)')
  }
```

Règle 7.9

Les propriétés de `result` du modèle de transformation d'attribut `T` (`primaryImplementedPSC`, `implementedPSCsForVariations`, `context`, `result`) créé pour une carte de dérivation `D` (`origin`, `destination`, `primaryPSC`, `variationPSCs`) sont paramétrées comme suit :

- `result.name` en substituant chaque occurrence d'`origin.name` par `context.name`;
- si `origin` est un attribut ou une extrémité d'association :

- result.visibility, en remplaçant chaque occurrence d'origin.visibility par context.visibility;
- result.changeability en substituant chaque occurrence d'origin.changeability par context.changeability;
- result.type en substituant chaque occurrence d'origin.type par context.type.

- si origin est une classe :
  - result.type en substituant chaque occurrence d'origin.name par context.name.
  - result.initialValue en substituant chaque occurrence du nom d'un attribut d'origin A par origin.attribute['A'].name et en substituant chaque occurrence du nom d'une association d'origin AE par origin.association['AE'].name
- si origin est un attribut :
  - result.initialValue en substituant chaque occurrence d'origin.initialValue par context.initialValue.
- si origin est une extrémité d'association ou une classe :
  - si result.initialValue est défini, la valeur de initialValue de result est paramétrée par la valeur du PRP prédéfinit initialValue de PSC (context.PSCs[PSC].PRP['initialValue'].value).
- si origin est none et l'origin de destination du classificateur n'est pas none :
  - result.type en substituant chaque occurrence d'origin.classifier.name par context.name;

- si `result.initialValue` est défini alors la valeur de la propriété `initialValue` de `result` est paramétrée en :
  - remplaçant chaque occurrence d'`origin.classifier.name` par `context.name` si `origin.classifier.name` est un patron de `result.initialValue;`
  - sinon par la valeur du PRP prédéfini `initialValue` de `PSC (context.PSCs[PSC].PRP['initialValue'].value).`

Dans notre exemple, le paramétrage du modèle de transformation décrit dans le tableau 7.2 est donné dans le tableau 7.6.

Tableau 7.6 Exemple d'un modèle de transformation d'attribut paramétré

```
primaryimplementedPSC = 'PICAttribute'
implementedPSCsForVariations = ϕ
context = associationEnd
result = the attribute
  name = context.name.toUpper()
  visibility = 'public'
  type = 'String'
  changeability = 'static final'
  initialValue = context.PSCs[' PICAttribute'].PRP['initialValue'].value
```

Règle 7.10

Les propriétés de `result` du modèle de transformation d'opération `T (primaryImplementedPSC, implementedPSCsForVariations, context, result)` créé pour une carte de dérivation `D (origin, destination, primaryPSC, variationPSCs)` sont paramétrées comme suit :

- `result.name` en substituant chaque occurrence d'`origin.name` par `context.name;`
- si `origin` est une opération :
  - `origin.visibility` en substituant chaque occurrence d'`origin.visibility` par `context.visibility;`

- `origin.returnType` en substituant chaque occurrence d'`origin.returnType` par `context.returnType`.

- si `origin` est un attribut :
  - `origin.returnType` en substituant chaque occurrence d'`origin.type` par `context.type`.

- si `origin` est une extrémité d'association :
  - `origin.returnType` en substituant chaque occurrence d'`origin.type.name` par `context.type.name`.

- si `origin` est une opération, un attribut ou une extrémité d'association :
  - `origin.raisedExceptions` en substituant chaque occurrence d'`origin.classifier.dependencies['d'].target.name` par `context.classifier.dependencies['d'].target.name` et d'`origin.classifier.dependencies['d'].target.packageName` par `context.classifier.dependencies['d'].target.packageName` pour chaque dépendance du classificateur d'origine.

- si `origin` est une classe (respectivement, `origin` est `none` et l'origine de la `destination` du classificateur n'est pas `none`) :
  - `origin.returnType` en substituant chaque occurrence d'`origin.name` (respectivement, `origin.classifier.name`) par `context.name`;
  - `origin.raisedExceptions` en substituant chaque occurrence d'`origin.dependencies.target.name` (respectivement, `origin.classifier.dependencies['d'].target.name`) par `context.dependencies['d'].target.name` et d'`origin.dependencies.['d'].target.packageName` (respectivement,

origin.classifier.dependencies['d'].target.packageName) par context.dependencies['d'].target.packageName pour chaque dépendance du classificateur d'origine.

- result.body est paramétré par l'application de la règle 7.11.

Règle 7.11

La propriété result du body du modèle de transformation d'opération T (primaryImplementedPSC, implementedPSCsForVariations, context, result) créé pour une carte de dérivation D (origin, destination, primaryPSC, variationPSCs) est paramétré comme suit :

- si origin est une opération, alors la valeur de la propriété body est paramétrée par la valeur du PRP prédéfini operationBody de PSC (context.PSCs[PSC].PRP['operationBody'].value);
- si origin n'est pas une opération, alors pour chaque instruction S :
  - si origin est un attribut ou une extrémité d'association A, alors body est paramétrée en remplaçant chaque occurrence du nom de A dans son nom par le nom de la propriété context de T;
  - si origin est un classificateur C, alors body est paramétré en remplaçant chaque occurrence du nom de chaque attribut de C ou d'extrémité d'association par respectivement le nom d'un attribut ou d'une extrémité d'association de la propriété context de T.
- si origin n'est pas une opération, pour chaque instruction S qui déclare une variable V :
  - si origin est un attribut A (respectivement, une extrémité d'association AE) et A (respectivement, AE) est relié par analyse de

flux de données à V, alors le type de V est paramétré par le type de la propriété context de T;

- si origin est un classificateur C et qu'il y a un attribut A de C (respectivement, une extrémité d'association AE de C) où A (respectivement, AE) est relié par analyse de flux de données à V, alors le type de V est paramétré par le type d'un attribut (respectivement, une extrémité d'association) de la propriété context de T.

- si origin n'est pas une opération, alors pour chaque instruction S qui affecte une variable V par une expression expr :
  - si origin est un attribut A et A est relié à par analyse de flux de données à V :
    - si expr est une *expression littérale* et :
      - expr est égal à A, alors la valeur initiale de expr est paramétrée avec la valeur initiale de la propriété context de T;
      - expr n'est pas égal à A, alors la valeur initiale de expr est paramétrée avec la valeur du PRP initialValue de PSC.
    - si expr est une *expression de transtypage* (« cast ») et que son type de transtypage est égal au type de A, alors expr est paramétré avec le type de la propriété context de T.
  - si origin est une extrémité d'association AE et que AE est relié à V par l'analyse de flux de données :

- si `expr` est une expression littérale, alors `expr` est paramétré avec la valeur du PRP `initialValue` de `PSC`;
- si `expr` est une *expression de transtypage* et que son type de transtypage est égal au nom du type de `AE`, alors `expr` est paramétré avec le nom du type de la propriété `context` de `T`.

o si `origin` est un classificateur `C` et qu'il y a un attribut `A` de `C` relié par analyse de flux de données à `V` :

- si `expr` est une *expression littérale* et que :
  - `expr` est égal à la valeur initiale de `A`, alors `expr` est paramétré avec la valeur initiale de l'attribut `A` de la propriété `context` de `T`;
  - `expr` n'est pas égal à la valeur initiale de `A`, alors `expr` est paramétré avec la valeur du PRP `initialValue`, un PSC au nom non défini et associé à l'attribut `A` de la propriété `context` de `T` (`context.attributes[A].PSCs['undefined'].PRP['initialValue'].value`). Ce PSC sera identifié dans l'étape suivante.
- si `expr` est une *expression de transtypage* et que son type de transtypage est égale au type de `A`, alors `expr` est paramétré avec le type de l'attribut `A` de la propriété `context` de `T`.

o si `origin` est un classificateur `C` et que `C` possède une extrémité d'association `AE` où `AE` est relié à par analyse de flux de données à `V` :

- si `expr` est une *expression littérale,* alors `expr` est paramétré avec la valeur du PRP `initialValue`, un PSC indéfini est associé à l'extrémité d'association `AE` de la propriété `context`

de `T (context.associationEnds[AE].PSCs['undefined'].PRP['initialValue'].value)`

- si `expr` est une *expression de transtypage* et que son type de transtypage est égal au nom du type de `AE`, alors `expr` est paramétré avec le type de l'extrémité d'association `AE` de la propriété `context` de `T`.

Dans notre exemple, le paramétrage du modèle de transformation décrit dans le tableau 7.3 est donné dans le tableau 7.7.

Tableau 7.7 Exemple d'un modèle de transformation d'opération paramétré

```
primaryImplementedPSC = 'create'
implementedPSCsForVariations = ϕ
context = class
result = the operation
  name = 'create'
  visibility = 'public'
  returnType = 'void'
  changeability = ''
  parameters = {
    call _template('com.imaginary.bank.AccountEntity.create.parameter(Identifier id)'),
    call_template('com.imaginary.bank.AccountEntity.create.parameter(AccountType t)'),
    call_template('com.imaginary.bank.AccountEntity.create.parameter(CustomerFacade cust)')
  }
  raiseExceptions = {'TransactionException'}
  body = 'prepareCreate(id);
  try {
   context.attributes['number'].name = (context.attributes['number'].type) SequenceGenerator
   .generateSequence("ACCT_NUM");}
  catch( SequenceException e ) {
   throw new TransactionException
             (e.getMessage());}
  context.associationEnds['type'].name = context.associationEnds['type'].name.abbreviation();
  context.associationEnds['customer'].name  = context.associationEnds['customer'].abbreviation();
  '
```

Règle 7.12

Les propriétés de la propriété `result` du modèle de transformation de paramètre `t` créé pour chaque carte de dérivation `D (origin, destination, primaryPSC, variationPSCs)` sont paramétrées comme suit :

- `result.name` en substituant chaque occurrence d'`origin.name` par `context.name;`
- si `origin` est un attribut :
    - `result.type` en substituant chaque occurrence d'`origin.type` par `context.type.`
- si `origin` est une extrémité d'association :
    - `result.type` en substituant chaque occurrence d'`origin.type.name` par `context.type.name.`
- si `origin` est une classe :
    - `result.type` en substituant chaque occurrence d'`origin.name` par `context.name.`
    - `result.name` en substituant chaque occurrence du nom d'un attribut d'`origin A` par `origin.attribute['A'].name` et en substituant chaque occurrence du nom d'une association d'`origin AE` par `origin.association['AE'].name`
- si `origin` est un paramètre et `result.directionKind` et `origin.directionKind` sont les mêmes :
    - `result.directionKind` en remplaçant chaque occurrence d'`origin.directionKind` par `context.directionKind.`

Dans notre exemple, le paramétrage du modèle de transformation décrit dans le tableau 7.4 est donné dans le tableau 7.8.

Tableau 7.8 Exemple d'un modèle de transformation de paramètre paramétré

```
primaryImplementedPSC = 'PICParameter'
implementedPSCsForVariations = ϕ
context = associationEnd
result = the parameter
  name = context.name.abbreviation()
  type = context.type.name+'Facade'
  directionKind = ''
```

7.3.3 Étape 2.3 : Généralisation des modèles de transformation.

La troisième étape de la deuxième sous-phase généralise les modèles de transformation obtenue lors des étapes précédentes.

Cette étape est composée de quatre sous-étapes comme le montrent les points 2.3.1 à 2.3.4 de la figure 7.3 :

2.3.1 fusion des modèles de transformation;

2.3.2 généralisation du corps des opérations;

2.3.3 raffinement des modèles de transformation de classificateur;

2.3.4 paramétrage des appels.

7.3.3.1 Sous-étape 2.3.1 : Fusion des modèles de transformation

L'objectif principal de cette sous-étape est de fusionner les modèles de transformation implémentant un même PSC primaire et ayant un contexte identique dans un seul modèle de transformation. Pour ce faire, la description des modèles doit être dissociée de toute mention particulière aux éléments du PIM. Cette sous-étape utilise la règle 7.13 et est résumée dans la figure 7.4.

Notez que, dans cette étape, la valeur de la propriété `implementedPSCsForVariations` d'un modèle de transformation est définie, chaque fois que le processus de généralisation découvre qu'un PSC est implémenté par des variations dans le code source.

Règle 7.13

Les membres de chaque groupe `G` de `n` modèles de transformation de paramètre (respectivement, attribut, opération et classificateur) {`Ti(primaryImplementedPSC, implementedPSCsForVariations, context, result, 1≤i≤n`} implémentant un même PSC primaire `P` et ayant le même contexte `CTX` sont généralisés comme suit :

- les modèles de transformation d'attribut et d'opération qui ont la même origine et qui sont appelés par un même modèle de transformation de classificateur sont placés dans des groupes distincts;
- si `G` est un groupe de modèles de transformation d'opération, alors les propriétés `body` et `parameters` de la propriété `result` de ses membres sont généralisées en appliquant les règles 7.14 et 7.15;

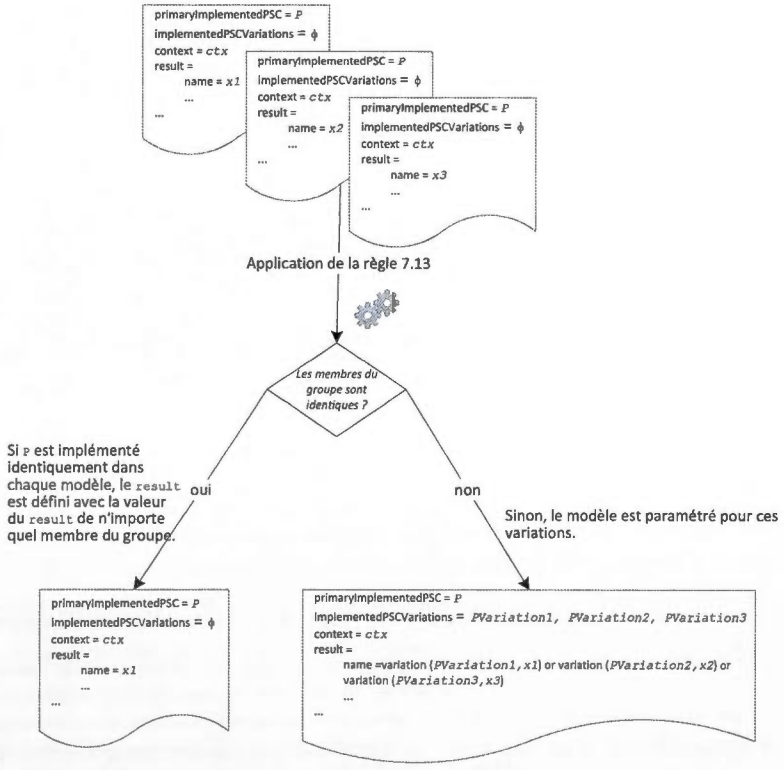

Figure 7.4 Fusion des modèles de transformation

- si `G` est un groupe de modèles de transformation de classificateur, alors les propriétés `importPackages`, `raisedExceptions`, `attributes` et `operations` de la propriété `result` de ses membres sont généralisées en appliquant les règles 7.16 et 7.17;
- les `n` modèles de transformation de paramètre (respectivement, attribut, opération et classificateur) de chaque groupe de `G` sont fusionnés en un seul modèle de transformation de paramètre (respectivement, attribut, opération et classificateur) `T(primaryImplementedPSC, implementedPSCsForVariations, context, result)` comme suit :
  - `T.primaryImplementedPSC = P;`
  - `T.context = CTX.`
- si `P` est implémenté en `m` variations (`m≤n`, avec une valeur différente pour au moins une propriété de `result`) et soit T_k`(primaryImplementedPSC, implementedPSCsForVariations, context, result)` est le modèle de transformation représentatif de la variation d'implémentation `k` (`1≤k≤m`) :
  - `m` nouveaux PSCs de paramètre (respectivement, attribut, opération et classificateur) $varPSC_j$ (`1≤j≤m`) sont créés et nommés par la chaîne obtenue par la concaténation du nom de `P`, le mot clé `variation` et l'index `j`. Ces nouveaux PSCs sont enregistrés dans la propriété `T.implementedPSCsForVariations;`
  - dans la valeur de chaque propriété `PR` de `T` de la propriété `result`, nous enregistrons la formule suivante :

$$\bigvee_{k=1}^{m} variation(t_k.result.PR, varPSC_k) \quad (1)$$

- si `P` n 'est pas implémenté avec des variations, alors la valeur `result` de `T` est définie selon la valeur de la propriété `result` d'un modèle de transformation de `G`.

Le tableau 7.9 présente un exemple de modèle de transformation paramétré obtenue en fusionnant des modèles ayant des différences pour les propriétés `superClassifier`, `importPackages` et `implementInterfaces`.

Tableau 7.9 Exemple d'un modèle de transformation paramétré avec variations

```
...]
primaryImplementedPSC = 'PICClass'
implementedPSCsForVariations = {'PICClasse_Variation_1',
'PICClasse_Variation_2'}
context = class
result = the class
  name = context.name
  packageName = context.packageName'
  visibility = 'public'
  superClassifier          =          variation('java.lang.Exception',
PICClasse_Variation_1) or variation ('', PICClasse_Variation_2)
  importPackages = variation({}, PICClasse_Variation_1) or variation
({'java.io.Serializable'}, PICClasse_Variation_2)
  implementInterfaces  =  variation({},  PICClasse_Variation_1)  or
variation ({'java.io.Serializable'}, PICClasse_Variation_2)
 [...
```

7.3.3.2 Sous-étape 2.3.2 : Généralisation du corps des opérations

L'objectif principal de cette sous-étape est de paramétrer le corps des modèles de transformation d'opération. En effet, ceux-ci nécessitent des traitements spécifiques.

Lorsque le contexte d'un modèle de transformation d'opération est une classe, la propriété `body` peut contenir, par exemple, des fragments de code référençant chacun un attribut différent. Par conséquent, outre les techniques d'analyse utilisées précédemment, nous utilisons également la détection de clones afin de mieux parvenir à la découverte des modèles de transformation d'opération. Cette sous-étape utilise la règle 7.14 et est résumée dans la figure 7.5.

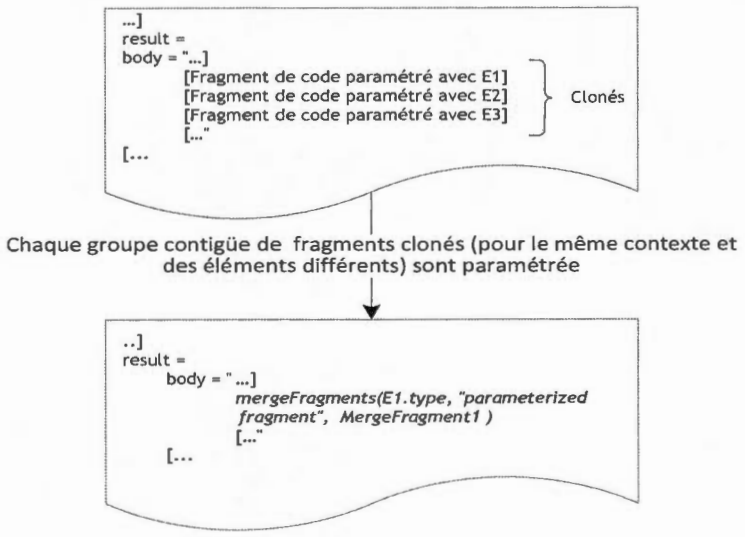

Figure 7.5 Sous-étape de généralisation du corps des opérations

Règle 7.14

Pour chaque modèle de transformation d'opération paramétré `T(primaryImplementedPSC, implementedPSCsForVariations, context, result)` dont le `context` est une classe, `result.body` est généralisé comme suit :

- Les portions de code correspondant à un groupe G_i ($1 \leq i \leq m \leq_{\text{nombre de lignes dans body original}}$) de portions de code cloné contiguës $clone_j$($1 \leq j \leq m \leq_{\text{nombre de lignes dans body original}}$) de type 2 (un code cloné peut être une instruction d'une seule ligne) dans `body` avant sa paramétrisation sont fusionnés dans un $fragment_i$ si :
  - chaque $fragment_j$ correspondant paramétré de `body` le sont pour un même contexte CTX_i;
  - et que ce contexte réfère à un élément différent E_j;
  - $fragment_i$ est obtenu en prenant n'importe quel fragment $fragment_j$ et en y remplaçant :
    - toutes les occurrences de E_i par son type d'élément;

- toute variation var_l (où k est le nombre de ces variations et $1 \leq l \leq k$) entre les $fragment_j$ par la valeur d'un nouveau PRP (nommé par la concaténation de la chaîne `varFragmentProperty` et de l'indice de l) associé à un PSC indéfini.

- pour chacun des groupes G_i fusionnés :
  - un nouveau PSC P_i du type de CTX_i est créé et nommé par la concaténation `T.primaryImplementedPSC.name`, le mot clé `mergeFragment` et l'index `i`. Ensuite, toutes les occurrences de PSC indéfinies dans $fragment_i$ sont remplacées par le nom de P_i;
  - les portions de code paramétré de `Gi` sont remplacées par la formule : `mergeFragments (`CTX_i`, `$fragment_i$`, `P_i`)`.

- Si `body` est toujours paramétré avec des propriétés d'un élément E_r (attribut ou association) d'`origin` :
  - que q représente le nombre de ces éléments ($1 \leq r \leq q$);
  - alors, toute occurrence de E_r est remplacée par la formule : `element(type de `E_r`, `P_r`)`;
  - où P_r est un nouveau PSC du type de E_r et nommé en concaténant `T.primaryImplementedPSC.name`, le mot clé `element` et l'index `r`.

Le tableau 7.10 nous montre un extrait d'un `body` contenant une portion clonée alors que le tableau 7.11 nous montre un extrait du même `body` où ce clone a été paramétré et remplacé par une formule `mergedFragment`.

7.3.3.3 Sous-étape 2.3.3 : Raffinement des modèles de transformation de classificateur

Cette sous-étape cherche à généraliser la répétition pouvant se trouver dans les propriétés `importPackage` et `raisedException` d'un modèle de transformation de classificateur. Cette sous-étape utilise la règle 7.15 et est résumée dans la figure 7.6.

Règle 7.15

La propriété `importPackages` (respectivement `raisedExceptions`) d'un modèle de classificateur `T` est généralisée comme suit :

- les membres de chaque groupe G_i ($1 \leq i \leq$ nombre d'éléments dans `importPackages` (respectivement, `raisedExceptions`)) d'`importPackages` (respectivement, `raisedExceptions`) où chaque élément est paramétré par la même chaîne `S` et le même contexte `CTX` (qui n'est pas un classificateur) sont fusionnés dans une seule instruction `importPackages` groupée (respectivement, `raisedExceptions`).
- pour chacun des groupes G_i fusionnés :
    - un nouveau PSC `Pi` du type de `CTX` est créé, nommé par la concaténation du nom de `T.primaryImplementedPSC.name`, le mot clé `mergeImportStatements` (respectivement `mergeRaisedExceptions`) et l'index `i`;
    - les éléments du groupe G_i, sont remplacés par la formule : `mergeImportStatements (CTX, S, Pi)` (respectivement, `mergeRaisedExceptions`).

Tableau 7.10 Extrait d'un body contenant du code cloné

```
...]
body = ...[ 'String '+context.attribute[name='name']->first().name+ '
=          request.getParameter("'+context.attributeList[name='name']-
>first().name+';'
+ 'String '+context.attribute[name=' description ']->first().name+ '
=  request.getParameter("'+context.attributeList[name='  description
']->first().name+';'+ [...
```

Tableau 7.11 Extrait du body de la figure précédente généralisé

```
...]
body = ...[ +  mergedFragment (attribute, 'String '+
context.attribute[x].name+' = request.getParameter(' +
context.attributet[x].name+';', doGet MergeFragment5) + [...
```

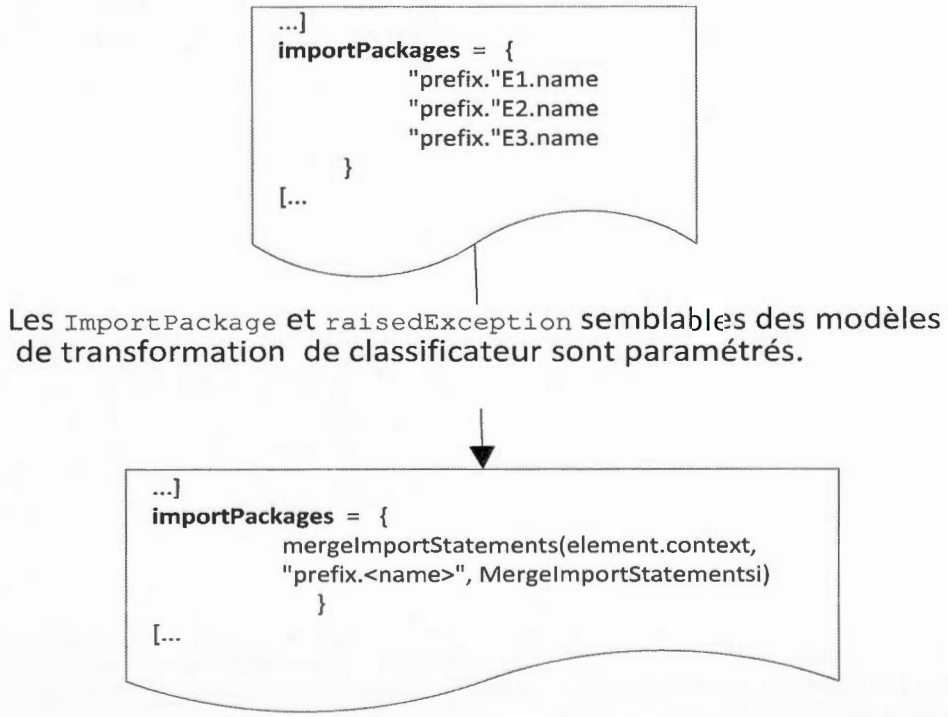

Figure 7.6 Raffinement des modèles de transformation de classificateur

7.3.3.4 Sous-étape 2.3.4 : Paramétrage des appels

Cette sous-étape cherche à généraliser la répétition dans les appels de modèles que l'on trouve dans les modèles de transformations de classificateur pour les propriétés

`attributeList` et `operationList` et pour les modèles de transformations d'opération pour la propriété `parameterList`. Cette sous-étape utilise les règles 7.16 à 7.17. La règle 7.16 est résumée dans la figure 7.7 et la règle 7.17 dans la figure 7.8.

Règle 7.16

La propriété `parameters` d'un modèle de transformation d'opération `T` est généralisée comme suit :

- les membres de chaque groupe G_i ($1 \leq i \leq$ nombre de paramètres) de paramètres consécutifs appelant le même modèle de transformation de paramètre T_i sont fusionnées dans un seul appel groupé;
- pour chacun des groupes G_i fusionnés :
    - les éléments du groupe G_i, sont remplacés par la formule suivante : `mergeParameters(`T_i`.context, call_template (`T_i`))`.

Le tableau 7.12 représente un appel à un même template de paramètre répété pour plusieurs attributs à l'intérieur d'un modèle de transformation d'opération alors que le tableau 7.13 nous montre ces appels remplacés par une formule `mergeParameters`.

Tableau 7.12 Un appel à un template de paramètre répété

```
...]
parameters = callTemplate(context.name) //Pour le paramètre customerId
parameters += callTemplate(context.name) //Pour le paramètre name
parameters += callTemplate(context.name) // Pour le paramètre email
parameters += callTemplate(context.name) // Pour le paramètre street
parameters += callTemplate(context.name) // Pour le paramètre companyName
[...
```

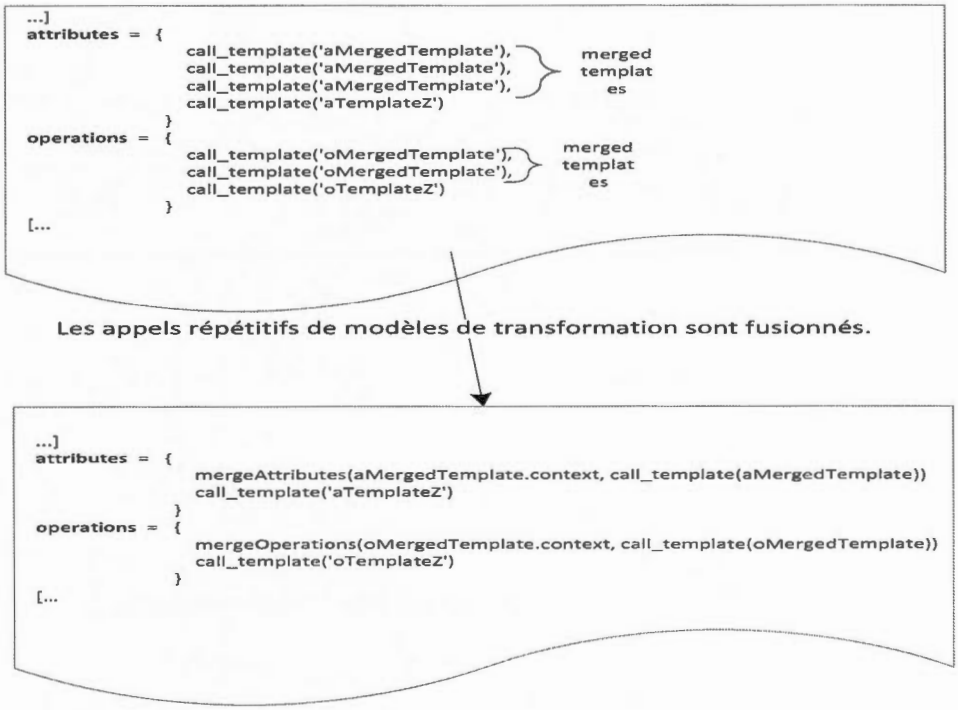

Figure 7.7 Généralisation des propriétés attributes et operations

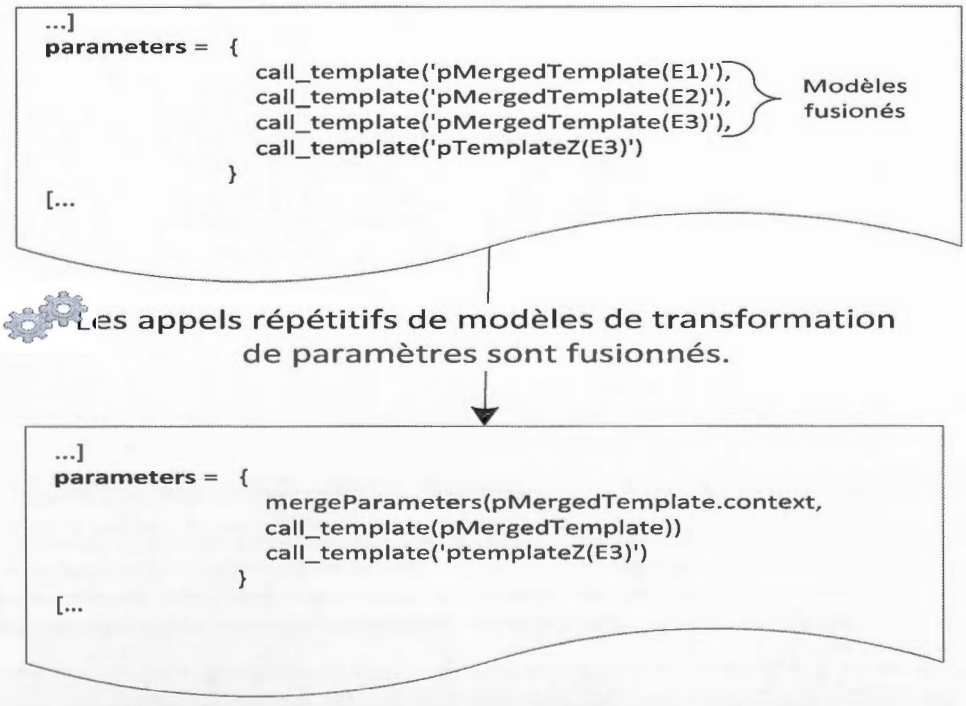

Figure 7.8 Généralisation de la propriété parameters

Tableau 7.13 Un appel à un template de paramètre généralisé

```
...]
parameters = mergeParameters(Attribute, call_template(context.name))
[...
```

Règle 7.17

La propriété `attributes` (respectivement, `operations`) d'un modèle de transformation de classificateur `T` est généralisée comme suit :

- les membres de chaque groupe G_i ($1 \leq i \leq$ nombre d'attributs (respectivement, opérations)) d'attributs (respectivement, opérations) appelant le même modèle de transformation d'attribut (respectivement, opération) T_i sont fusionnés dans un seul appel groupé;
- pour chacun des groupes G_i fusionnés :
    - les éléments du groupe G_i, sont remplacés par la formule suivante : `mergeAttributes(`T_i`.context, call_template(`T_i`))` (respectivement, `mergeOperations`)

Le tableau 7.14 représente des appels répétés aux mêmes templates d'attribut et d'opération à l'intérieur d'un modèle de transformation de classificateur alors que le tableau 7.15 nous montre des appels à des modèles d'attribut et d'opération qui remplacent ceux-ci à l'intérieur du même modèle de classificateur.

7.4 SOUS-PHASE 3 : SÉRIALISATION DES MODÈLES DE TRANSFORMATION

Cette sous-phase génère un document QVT décrivant les modèles de transformation extraits lors de la phase précédente. Le tableau 7.16 montre un extrait du document QVT généré pour notre système.

La sérialisation d'un modèle de transformation `T(primaryImplementedPSC, implementedPSCsForVariations, context, result)` se divise en trois sections :

- une signature qui suit le modèle : « <T.context>:: <templateName> : <T.result.type> »;
- une garde (un « when clause ») qui contient la ligne `self.stereotype.includes(' T.primaryImplementedPSC');`
- un corps de transformation qui contient les instructions nécessaires afin de réaliser les transformations pour générer le `result`.

Les formules `mergeAttributes`, `mergeOperations` et `mergeParameters` sont exprimées dans le corps des transformations par l'expression d'appel d'une autre transformation à l'aide du mot clé `map`. Par exemple, les formules `mergeAttributes ('attribut', call_template(T))` sont traduites en QVT par l'expression `attribute->map T()`.

Tableau 7.14 Appels à des modèles de transformation répétés

```
…]
attributes = callTemplate(context.name.firstToLower()) //Pour
l'attribut customer
attributes += callTemplate(context.name.firstToLower()) //Pour
l'attribut salesOrder
attributes += callTemplate(context.name.firstToLower()) // Pour
l'attribut salesOrderItems
[…]
operations = callTemplate('get'+context.name.firstToUpper())) //Pour
l'opération getLineId
operations += callTemplate('get'+context.name.firstToUpper()))
//Pour l' opération SalesOrderItems
[…
```

Tableau 7.15 Appels à des modèles de transformation généralisés

```
attributes = mergeAttributes(attribute,
call_template(context.name.firstToLower())
operations = mergeOperations(operation,
call_template('get'+context.name.firstToUpper()))
```

Tableau 7.16 Une partie du document QVT généré pour le système

```
mapping  Class::classToJavaClassPersistence():Class
when{self.stereotypeList->includes('Persistence');}
{
  var context       := self;
  name              := context.name+'Persistence';
  packageName       := context.packageName;
  superClassifierName:='JDBCSupport';

  [...]

  if(self.stereotypeList->includes('PersistenceVariation2'))then
  {
    importPackageList:=List {'java.util.ArrayList'};
  }
  endif;
  importPackageList += List  {'com.imaginary.lwp.BaseEntity',[...]};
  attributeList += self->map  classToJavaAttribute_CREATE();

  [...]

  operationList +=self->map classToJavaOperation_getPrimaryTable();
  [...]
}

mapping Class::classToJavaAttribute_CREATE():Attribute
when{ self.stereotypeList->includes('CREATE');}
{
  name                  := 'CREATE';
  visibility            := 'private';
  changeability         := 'static final';
  type                  := 'String';

  [...]
}

mapping Class::classToJavaOperation_getPrimaryTable():Operation
when{ self.stereotypeList->includes('getPrimaryTable');}
{
  var context   := self;
  name          := 'getPrimaryTable';
  returnType    := 'String';
  specification := 'return "'
                   + context.name.toUpper()'";';

  [...]
}
```

Les formules `mergeImportStatements`, `mergeRaiseExceptions` et `mergeFragments` sont exprimées en QVT par une expression `forEach`. Par exemple, la formule `mergeFragment('associationEnd', 'associationEnd.name = associationEnd.name.abbreviation());');', 'PSC')` dans le `body` d'un modèle de transformation est traduite en QVT par le code exprimé dans le tableau 7.17.

Tableau 7.17 Traduction en QVT d'une formule `mergeFragment`

```
this.associationEnd->forEach(assEnd| assEnd.stereotype-
>includes('PSC'))
{
  body  := body + assEnd.name + ' = '
  assEnd.name.abbreviation())+(';');
};
```

Les variations dans les propriétés sont exprimées en QVT en utilisant des expressions « `if/else` ». Par exemple les variations : `variation('ArrayList', 'variation_1')`, `variation(context.type.name+'Facade', 'variation_2')` et `variation('context.type.name', 'variation_3')` de la propriété `type` d'un modèle de transformation d'attribut sont traduites en QVT par le code exprimé au tableau 7.18.

Tableau 7.18 Traduction en QVT de variations

```
if (self.type.stereotype->includes('variation_1'))) then
    type := 'ArrayList'
  else
    if(self.type.stereotype->includes('variation_2'))then
      type := self.type.name+'Facade'
    else
if(self.type.stereotype>includes('
   variation_3'))then
        type := self.type.name
      endif
    endif
   endif;
```

La formule `element(element type of` E_r, P_r`)` est exprimée à l'aide des opérations `select` et `first` sur les collections d'OCL. Par exemple, la formule

`element('attribute', 'create_element_1')` d'attribut est traduite en QVT par le code exprimé dans le tableau 7.19.

Tableau 7.19 Traduction en QVT de element

```
attributeList->select(stereotypeList->includes('create_element_1'))-
>first().
```

7.5 VALIDATION DU TRAVAIL

Dans cette section, nous présentons notre cadre de validation, ensuite, nous présentons les résultats obtenus lors de notre processus de validation et enfin, nous discutons de plusieurs aspects de l'approche proposée et soulignons certains travaux futurs.

7.5.1 Cadre validation

Nous avons développé un prototype d'outil afin de valider notre approche mettant en application les règles présentées dans ce chapitre. Notre prototype prend en entrée un PSM, un PIM et le profil UML de la plate-forme du système représenté par le PSM.

Contrairement aux deux autres phases de notre approche, celle-ci est indépendante. Nous avons donc choisi de lui fournir en entrée un PIM et un profil UML « idéal » plutôt que ceux découverts dans les phases précédentes, afin de faire ressortir plus clairement les faiblesses et les forces de cette phase et quelle ne soient pas obscurcies par celles des deux autres phases. Nous avons appliqué notre outil aux six systèmes déjà présentés au chapitre 5.

Notre processus de validation pour tous les systèmes testés suit les étapes suivantes :

- génération du code QVT représentant les modèles de transformation;
- annotation des éléments du PIM afin d'indiquer comment ils doivent être transformés. Ils sont annotés avec les PSCs associés en utilisant les informations contenues dans les cartes de dérivation. Nous avons utilisé les

stéréotypes pour les PSCs et les paires étiquette-valeur (« tagged values ») pour les PRPs;

- régénération d'un PSM grâce au plugin Eclipse « Model To Model[9] » (M2M), un sous-projet du projet Eclipse Modeling, nous appliquons le code QVT au PIM annoté pour recréer un PSM du système;
- calcul d'un ensemble de métriques sur les modèles de transformation découverts et le PSM régénéré. Ces métriques sont décrites ci-dessous.

7.5.1.1 Taux de réduction

La première métrique calculée est le taux de réduction. Elle indique, en pourcentage, le rapport entre le nombre d'éléments dans le PSM et le nombre de modèles de transformation. Par exemple pour le système jouet, nous avons besoin de 7 modèles de transformation de classificateur pour représenter les 47 classificateurs du système. Le taux de réduction est donc de 85,11 %.

7.5.1.2 Nombre de variations

La deuxième métrique calculée est la variation qui capture le nombre de variations dans les modèles de transformation. Cette métrique donne deux types de renseignements. Tout d'abord, elle renseigne sur les variations dans l'implémentation d'un PSC. Ensuite, elle évalue la complexité des modèles de transformation obtenue. En fait, chaque variation dans un modèle apparait dans le code QVT par un bloc conditionnel, ce qui peut compliquer la maintenabilité.

7.5.1.3 Taux de paramétrage des modèles de transformation

La troisième métrique que nous mesurons est le nombre de références à des éléments du PIM qui demeurent présentes dans nos modèles de transformation paramétrés,

[9] http://www.eclipse.org/m2m/

bien entendu l'idéal est aucune. Nous indiquons le nombre de modèles qui en contiennent ainsi que le pourcentage.

7.5.1.4 Taux de similitude entre le PSM en entrée et celui régénéré

La quatrième métrique permet de mesurer le taux de similitude entre le PSM en entrée et celui régénéré. Elle est déterminée en calculant la moyenne du taux de similitude de chaque classificateur correspondant des deux PSMs. Deux classificateurs sont identiques si chacune de leurs propriétés (incluant les attributs et opérations) est identique, la similitude sera alors de 100 %. Pour évaluer la similitude de deux classificateurs, nous mesurons la similitude de chacune de ses propriétés une à une (incluant ses attributs et opérations) sur la même échelle et faisons leur moyenne. Le taux de similitude des opérations et des attributs (qui est employé dans le calcul de celui des classificateurs) se fait de la même façon, mais pour leurs propres propriétés. Si un attribut ou une opération est en trop ou manquant, nous déterminons une similitude de 0 pour cet élément.

7.5.2 Résultats

Le tableau 7.20 présente les résultats obtenus pour la découverte des modèles de transformation des six logiciels analysés. Nous revenons ensuite plus en détail pour chacun de ces logiciels dans le reste de cette section en analysant des points spécifiques de leur résultat.

Pour la métrique de la variation, le tableau donne pour chaque système, la moyenne et le nombre maximal de variations selon le type de modèle de transformation. Une valeur de 1 indique qu'un PSC est implémenté avec une variation (deux possibilités).

Tableau 7.20 Résultat de la découverte des modèles de transformation

	Jouet	SGV	BJDBC	VSM	OFBS	Pet
Nombre de lignes de code	1124	1091	2076	3198	3082	6190
Nombre de classificateurs	47	29	67	89	74	209
Nombre d'attributs	55	53	130	146	132	411
Nombre d'opérations	430	207	375	503	427	1049
Nombre de paramètres	234	250	332	486	556	858
Nombre de modèles de classificateurs	7	11	59	50	44	131
Nombre de modèles d'attributs	7	17	108	68	60	281
Nombre de modèles d'opérations	60	52	315	264	163	633
Nombre de modèles de paramètres	77	46	245	264	300	564
Réduction classificateurs	85,11	62,07	11,94	43,82	40,54	37,32
Réduction attributs	87,27	67,92	16,92	53,42	54,55	31,63
Réduction opérations	86,05	74,88	16,00	47,51	61,83	39,66
Réduction paramètres	66,95	81,60	26,20	45,68	46,04	34,27
Variations moyennes classificateurs	0,71	0,27	0,09	0,48	0,41	0,58
Variations moyennes attributs	0,14	0,24	0,06	0,22	0,25	0,18
Variations moyennes opérations	0,77	0,90	0,09	0,26	0,68	0,92
Variations moyennes paramètres	0,14	0,50	0,04	0,10	0,21	0,11
Maximum de variations classificateurs	2	1	1	10	3	16
Maximum de variations attributs	1	2	4	8	10	19
Maximum de variations opérations	6	3	2	16	27	42
Maximum de variations paramètres	6	1	2	5	8	4
Taux de paramétrage des modèles (nombre)	0	10	24	84	83	57
Taux de paramétrage des modèles (pourcentage)	100,00	92,06	96,70	87,00	85,36	96,46
Taux de similitude des PSM original et régénéré	97,26	99,27	99,55	98,63	97,40	98,41

7.5.2.1 Le système jouet (Jouet)

Dans ce système, il est intéressant de noter qu'il n'y a pas de classificateur orphelin (c'est-à-dire purement reliés à la plate-forme d'implémentation) et que donc chacun des 7 modèles de transformation de classificateur est relié directement à un des 6 PSCs de classificateur en plus de celui correspondant au PSC prédéfini `PICInterface`.

Des répétitions n'ont pas été détectées dans certains corps de méthode, car elles contiennent à la fois des références à des attributs et à des associations. Par exemple,

le fragment de code « `"account=" + getAccount() + " " + "amount=" + getAmount()` » de l'opération `toString` de `BankTransactionData`. Cette répétition évidente n'est pas détectée, car `account` est une extrémité d'association alors qu'`amount` est un attribut. Donc, on ne peut les regrouper avec un `forEach` qui itère sur un même type d'élément.

Une des causes d'une similitude non totale entre le code original et le code régénéré vient du fait que le type de certains attributs ou opérations nécessite le nom complet (qualifié) du type alors que d'autres non. Par exemple, dans le classificateur du PSM `Account`, le type de l'opération `getBalance` devient dans le PSM régénéré `BigDecimal` au lieu du `java.math.BigDecimal` attendu. Il s'agit d'un problème facile à résoudre.

Un autre problème se produit lorsqu'un ensemble d'éléments est créé avec le même template pour plusieurs éléments du PIM. Dans ces cas, nous ne pouvons garantir dans quel ordre ils seront créés. Par exemple, ce cas se produit lorsqu'une opération a un paramètre créé pour chaque attribut de la classe contenant l'opération. Comme c'est le cas pour l'un des constructeurs de la classe `AccountData` du PSM. Il s'agit d'une difficulté avec QVT que nous devrons contourner.

Une autre difficulté avec nos modèles de transformation QVT qui induit une dissimilitude se produit lorsqu'un classificateur possède des éléments dérivés de ceux de sa super classe. Ainsi, le classificateur `CheckingAccount` hérite de `Account` et les modèles de transformation de `CheckingAccount` dépendent d'éléments de `Account` qui ne sont pas accessibles dans les transformations QVT telles que nous les avons conçues. Cette difficulté devrait être contournable en modifiant légèrement nos modèles de transformation QVT.

Ces raisons d'absence de similitude totale entre le PSM original et régénéré s'appliquent aussi aux autres systèmes testés et ne seront pas répétées dans leur analyse.

7.5.2.2 Le système de gestion des ventes (SGV)

Ce qui est à noter particulièrement dans ce système, c'est le faible nombre de variations pour chacun des modèles de transformation découverts. Dans ce système comme dans les autres, les modèles de transformations contenant le plus de variation sont ceux d'opération, car la partie de paramétrisation et de généralisation du corps des opérations est la plus difficile à laquelle nous avons à faire face. Le maximum de 3 variations représente les modèles de transformation d'opération pourrait être encore réduit si nous pouvions améliorer la détection des répétions dans le code source. Par exemple, les 3 répétitions dans le modèle de transformation relié au PSC `load` pourraient être détectées et fusionnées dans une seule variation.

Nous avons aussi relevé une problématique dans la qualité du paramétrage des propriétés des modèles de transformations reliés au classificateur du PSM `Table`. En effet, celui-ci n'est pas relié à un élément du PIM, mais les noms de certains de ses attributs réfèrent directement à des éléments du PIM, mais ne sont pas paramétrés. C'est ce qui cause le taux de paramétrage de 92,06% pour ce système.

7.5.2.3 Banking JDBC (BJDBC)

Le taux de réduction pour les classificateurs de ce système est faible comparé aux autres systèmes testés, car son PIM contient seulement 5 classes et des 67 classificateurs de son PSM, seulement 20 sont liés à ces classes. Le reste du système se compose de 47 classificateurs représentant exclusivement des exigences de la plate-forme d'implémentation et donc pour chacun d'eux un modèle de transformation est créé. Nous pouvons déduire qu'un système plus large mettant en œuvre la même plate-forme d'implémentation aura un plus grand rapport de réduction.

Ses modèles de transformation de classificateur contiennent une moyenne de 0,09 variation et un maximum d'une seule variation. Le faible nombre de variations est

causé par les 47 classificateurs qui représentent exclusivement des exigences de la plate-forme d'implémentation et qui par conséquent ne contiennent pas de variation. Les autres ne sont liés au maximum qu'à deux classificateurs du PSM, donc ils ne peuvent avoir plus qu'une seule variation. Ce maximum de 1 représente cinq modèles de transformation comme celui représentant le PSC `PICinterface` où une variation importe une seule interface et une autre en importe quatre.

On voit aussi que le modèle de transformation créé pour son classificateur `BankFrame` conserve le nom d'éléments du PIM. Le problème est que ces noms réfèrent à des attributs de classes du PIM autres que celles reliées à `BankFrame`, ce qui présente une des limites de notre approche.

7.5.2.4 Le centre commercial virtuel (VSM)

Ce système obtient une réduction intéressante pour les paramètres. La cause en est que beaucoup de ses paramètres implémentent un PSC commun, ce qui fait que leurs modèles de transformations sont fusionnés. Par exemple, il y a 129 paramètres d'opérations de classificateur du PSM qui sont liés au PSC de paramètre `id`. Leurs modèles de transformations sont fusionnés en un seul modèle.

7.5.2.5 Le service de courtage en ligne (OFBS)

Parmi les systèmes non générés, c'est ce système qui obtient la meilleure réduction pour les opérations. Ce qui signifie que plusieurs opérations du PSM partagent un même modèle de transformation. Donc contrairement à BJDBC, plusieurs PSCs d'opération sont implémentés à répétition. Notamment, le PSC d'opération représentant les « getters » est implémenté 55 fois et celui les « setters » 51 fois. Ce qui fait qu'une fois fusionnés, 104 modèles de transformation d'opération sont éliminés.

Mais surtout, on remarque que c'est ce système qui contient le plus grand nombre de modèles de transformations contenant encore des références directes à des éléments

du PIM. Une grande cause de cette problématique (37 des 83 modèles de transformation problématiques) est un problème évoqué précédemment. Ce problème est celui des modèles de transformation qui créent des éléments du PSM qui ne sont pas reliés à des éléments du PIM et pour qui on ne peut donc paramétrer ces modèles.

7.5.2.6 Le EJB Pet Store (PET)

C'est ce système qui possède le plus grand nombre de modèles de transformation de paramètre. Cela est causé par le fait qu'il possède plusieurs modèles de transformation d'opération de plate-forme ayant des noms de paramètre différents, ce qui crée autant de modèles de transformation de paramètre. De plus, on observe qu'une grande majorité de ces modèles de transformation de paramètres ne sont pas du tout paramétrés.

7.5.3 Discussion

Les différents essais, que nous venons de présenter, nous aident à esquisser des réflexions et des constatations sur notre approche. Certaines nous font voir les causes des limites de notre approche alors que d'autres sont une piste de réflexion afin d'y remédier.

7.5.3.1 Complexité des algorithmes

La phase de découverte des modèles de transformation consiste en plusieurs étapes que nous avons exprimées dans un ensemble de règles. Plusieurs facteurs en rapport au nombre d'éléments dans le PSM entrent en compte dans le calcul de la complexité de l'application de ces règles : le nombre de classificateurs (n_c), le nombre d'opérations (n_o), le nombre d'attributs (n_a), le nombre de paramètres (n_p), le nombre d'import(n_{import}), le nombre d'exceptions (n_{raised}), le nombre d'éléments (classificateurs, attributs, opérations et paramètres) (n_e), le nombre de jetons (n_t) et le nombre d'instructions (n_i) dans le code des ses opérations. Le nombre de PICs

d'attribut et d'associations dans le PIM (n_{pa}) entre aussi en compte. Nous avons choisi d'estimer la complexité dans le pire des cas pour chacune de ces règles.

Deux des algorithmes employés par ces règles pourraient sembler particulièrement couteux. Le premier algorithme est l'application de DFA. Son temps d'exécution est proportionnel au nombre d'instructions dans le code des opérations et plus spécifiquement celles de branchement. Ce qui donne dans notre cas $O(n_i)$. Le second algorithme qui pourrait être couteux est la détection des opérations clonées. Son temps d'exécution a été évalué à la section 5.3.5.1 à $O(n_t \log n_t)$.

Le tableau 7.21 résume une estimation de la complexité pour chaque règle. Donc, nous pouvons estimer la complexité de cette phase à $O(n_e) + O(n_t \log n_t) + O(n_{pa} n_t)$.

7.5.3.2 Les classificateurs qui ne sont reliés qu'à la plate-forme d'implémentation

On voit que dans plusieurs systèmes, des modèles de transformation de classificateur qui sont reliés à un unique classificateur du PSM et qui ne sont pas reliés à un élément du PIM. Ces classificateurs orphelins représentent purement une exigence de la plate-forme d'implémentation. Pour le moment, notre approche crée pour ces classificateurs plusieurs modèles de transformation de tous les types, ce qui augmente grandement le nombre des modèles de transformation. Une idée serait peut-être de définir de tels classificateurs ailleurs que dans les modèles de transformation, car dans ce cas, il s'agit plus de création que de transformation. On pourrait imaginer ajouter au PDM une troisième vue qui représente ces classificateurs directement.

Une autre problématique avec ces classificateurs arrive quand certaines de leurs propriétés réfèrent à des éléments du PIM, mais de la façon dont sont construites nos règles, ceux-ci ne peuvent être paramétrés adéquatement. Il sera facile d'étendre nos règles afin de supporter ces cas.

Tableau 7.21 Complexité des règles de la phase de découverte des modèles

Règle	Complexité	Motivation (le pire des cas)
7.1	$O(n_c)$	Pour chaque classificateur du PSM.
7.2	$O(n_a)$	Pour chaque attribut du PSM.
7.3	$O(n_o)$	Pour chaque opération du PSM.
7.4	$O(n_p)$	Pour chaque paramètre du PSM.
7.5	$O(n_e)$	Pour chaque élément du PSM.
7.6	$O(n_e)$	Pour chaque élément du PSM.
7.7	$O(n_e)$	Pour chaque élément du PSM.
7.8	$O(n_c)$	Pour chaque modèle de transformation créé pour chaque classificateur du PSM.
7.9	$O(n_a)$	Pour chaque modèle de transformation créé pour chaque attribut du PSM.
7.10	$O(n_o)$	Pour chaque modèle de transformation créé pour chaque opération du PSM.
7.11	$O(n_{pa}\, n_t)$	Pour chaque modèle de transformation créé pour chaque opération du PSM, on examine chacun de ses jetons en rapport avec les PICs d'attribut et d'association
7.12	$O(n_p)$	Pour chaque modèle de transformation créé pour chaque paramètre du PSM.
7.13	$O(n_e)$	Chaque modèle de transformation est fusionné en couple.
7.14	$O(n_t)$	La quantité de code cloné est proportionnelle au nombre de jetons.
7.15	$O(n_{import}+ n_{raised})$	Proportionnel à ces éléments
7.16	$O(n_p)$	Il peut avoir un appel pour chacun de ces éléments.
7.17	$O(n_a + n_o)$	Il peut avoir un appel pour chacun de ces éléments.

7.5.3.3 La découverte des liens entre les éléments

Notre approche fonctionne bien pour découvrir les liens entre les éléments quand leurs noms sont écrits de façon conséquente et que les liens sont explicites dans ces noms. Mais ce n'est pas toujours le cas, avec des abréviations, des synonymes et des fautes d'orthographe. Par exemple, des éléments liés au PIC d'attribut `socialSecurity` de Banking JDBC sont liés à des éléments du PSM contenant parfois `ssn` et parfois `socialSecurity` dans leur nom. Dans ces cas, même si nous

donnons les meilleurs modèles possible en entrée, il est impossible de détecter ces liens en se basant sur les identificateurs.

Une solution qui a été explorée est l'analyse de flux de données. Elle résout la problématique de l'exemple précédant. Mais ce n'est pas suffisant et nous devons explorer d'autres voies afin d'améliorer l'identification des liens entre les éléments du PSM et du PIM afin d'améliorer la qualité des modèles de transformation.

Un autre problème avec nos règles se produisant lors de la découverte des liens entre les éléments arrive lorsque des propriétés d'attributs d'autres classificateurs ou des propriétés de classificateurs autres que celle du classificateur contenant l'élément à paramétrer sont référées dans cet élément. Il faudra donc étendre notre approche pour répondre à cette problématique.

7.5.3.4 La réduction et la taille du système

Comme nous avons pu le constater, la réduction n'est pas en lien directement avec la taille du système. Mais plutôt avec le nombre d'éléments du PIM partageant un PSC commun. Donc plus un système aura une plate-forme implémentée uniformément pour un PIM et plus le taux de réduction sera élevé.

7.5.3.5 Les variations et la taille du système

Le tableau 7.20 démontre que le nombre de variations par modèle de transformation tend à augmenter avec la taille du système analysé. Cela est causé par le fait que plus un système est grand, plus un PSC risque d'être implémenté avec plusieurs variations.

Le nombre de variations va aussi diminuer en améliorant la paramétrisation. Car si différents modèles de transformation liés au même PSC sont paramétrés de façon idéale, la possibilité que ses propriétés paramétrées soient identiques (donc sans variation) est plus élevée.

Une autre voie pour réduire le nombre de variations serait de détecter des PSCs ayant des noms identiques et implémentant un différent concept et non seulement qui sont un raffinement d'un autre. Cela réduirait le nombre de variations par modèle de transformation. Le nombre de modèles augmenterait, mais ils seraient moins complexes.

7.5.3.6 Le paramétrage du corps des opérations

Le paramétrage du corps des opérations repose pour beaucoup sur la détection des portions de code clonées et sur l'analyse de flux de données. Ces deux parties de notre algorithme pourraient être améliorées de façon à mieux détecter des liens entre des éléments et les portions de code répétitifs.

Un de nos problèmes avec la détection des fragments de code répété est la difficulté d'utiliser les outils de détection de clones pour repérer des petites portions de code cloné pertinentes. Une idée à explorer serait l'emploi d'expressions régulières. De plus comme nos analyses de flux de données sont relativement simples, il serait intéressant de les pousser plus loin afin de découvrir comment exploiter encore plus cette technique.

7.5.3.7 La quantité de modèles de transformation de paramètres

Il y a beaucoup de modèles de transformation de paramètres. Une idée que nous énoncions précédemment pour les réduire est que quand ils ne sont pas liés à des éléments du PIM, de ne pas créer de modèles de transformations, mais de créer directement les paramètres dans les modèles de transformation d'opération.

7.5.3.8 Complexité des modèles de transformation

Un intérêt de l'approche MDA est de simplifier la maintenance logicielle en maintenant des modèles plutôt que du code source. Mais il faut que la maintenance de ces modèles soit simple. Donc, un minimum de modèles à maintenir est nécessaire.

Mais une autre nécessité est que ces modèles ne comportent pas un nombre exagéré de variations.

On pourrait ainsi imaginer créer un seul modèle de transformation par type d'éléments pour tout un système, mais dans ce cas nous serions face à des modèles ayant un grand nombre de variations et très difficiles à maintenir. À l'inverse, nous pourrions imaginer créer un modèle de transformation pour chaque élément du PSM, dans ce cas les modèles seraient très simples, mais leur nombre complexifierait encore la maintenance. La création des modèles de transformation doit donc concilier au mieux ces deux exigences un peu contradictoires.

7.5.3.9 Problème lors de la régénération

Lors de la validation de notre approche, nous avons levé plusieurs petits problèmes qui devront être corrigés, mais qui ne posent pas de défi particulier. Le premier soulevé est que nous n'avons pas prévu dans les modèles de transformation de distinction entre noms qualifiés et non. Une autre problématique est quand nous employons une boucle « forEach » pour créer des éléments et que nous appelons un même modèle de transformation pour créer plusieurs éléments, nous ne somme pas certain de l'ordre dans lequel ceux-ci vont être créé et nous ne pouvons donc pas garantir que ces éléments seront dans la même position dans le PSM régénéré.

7.5.3.10 Qualité de l'architecture et du modèle métier

Si le système ne repose pas sur un PIM et une plate-forme clairement définie et bien implémentée dans un PSM, il devient problématique pour notre approche de détecter des liens entre des éléments quand ceux-ci sont flous ou imprécis. Tout comme il est difficile de fournir un profil UML de la plate-forme d'implémentation et un PIM de qualité à l'outil. Donc la définition de modèles de transformation est tributaire de la qualité des modèles en entrée.

7.6 SYNTHÈSE

Nous avons présenté dans ce chapitre l'outil que nous avons développé pour valider notre approche, le protocole que nous avons suivi lors de notre processus de validation, les résultats de notre processus de validation. Nous avons conclu ce chapitre par une discussion sur notre processus de validation et des pistes d'améliorations à suivre.

CONCLUSION

Dans cette thèse, nous avons proposé une nouvelle approche de modernisation fondée sur ADM permettant la migration de systèmes légataires développés dans des environnements non IDM vers un environnement IDM. Dans ce qui suit, nous allons résumer nos contributions et présenter les extensions possibles à notre recherche.

UN PROCESSUS DE MODERNISATION ADM REVISITÉ

Nous avons proposé un nouveau processus de modernisation ADM. Nous y avons défini les différents modèles nécessaires et les transformations à réaliser pour passer d'un modèle à l'autre. Ce processus permet de passer du code source d'un système légataire à un ensemble de modèles à plus haut niveau d'abstraction et ainsi permettre d'utiliser l'IDM pour moderniser ce système.

Dans ce processus, afin de surmonter les problèmes que pose la représentation de la plate-forme d'implémentation et son application à un modèle métier, nous avons proposé de la représenter par deux modèles. Le premier modèle représentant la plate-forme d'implémentation est un profil UML décrivant les concepts de la plate-forme sur laquelle le système a été implémenté. Ce premier modèle décrit aussi les contraintes entre les concepts en employant le langage OCL. Le second modèle représentant la plate-forme d'implémentation est un ensemble de modèles de transformations paramétrés capturant le code d'infrastructure de la plate-forme d'implémentation. Les modèles de transformation sont décrits à l'aide du langage QVT.

Afin de faciliter l'application de ce processus, nous avons proposé des algorithmes pour la découverte automatique des trois principaux modèles employés à partir du code source d'un système légataire.

D'abord, nous avons proposé un algorithme permettant la découverte du profil UML de la plate-forme d'implémentation d'un système légataire. Ensuite, nous avons proposé un autre algorithme pour la découverte du modèle représentant les éléments du domaine du problème contenu dans le système légataire. Ce second modèle, sous forme d'un diagramme de classes UML, donne une vue du système d'un point de vue indépendant de toute plate-forme d'implémentation. Finalement, nous avons proposé un troisième algorithme pour la découverte du troisième modèle qui contient un ensemble de modèles de transformations paramétrés capturant le code d'infrastructure de la plate-forme d'implémentation.

L'importance de nos contributions réside dans le fait qu'à notre connaissance, notre travail est le premier qui est allé aussi loin dans la migration d'un système légataire vers un environnement IDM. Même si l'objectif de la thèse était au départ très ambitieux, nous avons voulu relever le défi pour voir jusqu'où nous pourrions aller.

Un des fondements de notre approche est que la découverte des éléments reliés à une plate-forme d'implémentation est rendue possible par le fait que ces éléments sont implémentés de manières répétitives ou semi-répétitives. Afin de trouver cette répétitivité, nous avons exploré différentes techniques de rétro-ingénierie nous permettant d'identifier cette répétitivité. Notre approche a combiné plusieurs techniques telles que l'analyse des identificateurs, l'indexation sémantique latente, la détection des clones et l'analyse des flux de données.

EXTENSIONS POSSIBLES

Notre approche a été validée sur plusieurs systèmes Java et a donné de bons résultats pour ceux bien structurés ayant un bon style de programmation. Les autres posent des défis à notre approche qui peuvent être remédiés par trois types d'extensions.

Le premier type d'extension consiste à améliorer les travaux actuels. En effet, notre processus de validation nous a montré plusieurs pistes à suivre. D'abord, la découverte des modèles est rendue compliquée par la non-répétitivité souvent présente dans l'implémentation d'un système légataire. Nous avons exploré plusieurs voies afin de confronter cette problématique comme nous l'avons déjà mentionné, mais il y a encore place à amélioration. Il faut donc pousser plus loin ces techniques et en explorer d'autres. Une autre limite de notre approche est le manque de souplesse de nos règles qui sont souvent trop rigides. Il faut donc aussi explorer ces pistes.

Un deuxième type d'extension est de rendre l'approche interactive. Deux voies principales sont à explorer. D'abord, permettre lors de l'application de nos algorithmes un guidage par un expert qui permettrait de signaler une erreur à une étape qui en éviterait une cascade d'autres aux étapes suivantes. L'expert pourrait aussi apporter d'emblée des informations sur le système existant. La seconde interactivité désirable avec un expert serait de permettre la modification des modèles découverts afin de réaliser la tâche de modernisation. Une dernière extension possible est de découvrir comment les techniques de remaniement de code («refactoring») pourraient permettre d'enlever une des limites de notre approche du fait qu'elle se base sur l'analyse des identificateurs sur plusieurs de ses aspects. En effet, un système légataire ne suivant pas une bonne politique de nommage posera des défis importants. Nous croyons que cette limite n'est pas aussi handicapante qu'on peut le croire puisqu'un tel système peut passer au préalable par un processus de remaniement de code. Reste donc à voir si cette solution est possible à mettre en pratique surtout pour les gros systèmes.

APPENDICE A

LE SYSTÈME BANKING JDBC

Dans certains de nos exemples, nous utilisions un second système que nous nommons Banking JDBC ou BJDBC. Nous utilisons ce système, car certaines de nos règles s'appliquent à des systèmes ayant une architecture implémentée de façon moins uniforme et aucun exemple ne se trouve dans les classes du système jouet pour celles-ci.

Ce petit système bancaire contient les 67 classificateurs listés dans le tableau A.1. Il utilise une plate-forme d'implémentation ad hoc présentée dans le livre de Reese sur JDBC (Reese, 2000). La figure A.1 décrit le PSM de ce système. Nous sommes conscients qu'un tel diagramme imprimé à cette échelle est difficilement lisible, mais le but premier de sa présence est de montrer la différence entre le PSM et le PM qui est décrit dans la figure A.2. La figure A.3 représente une partie du profil UML de sa plate-forme d'implémentation.

Tableau A.1 Les classificateurs de Banking JDBC

Account	AccountEntity	AccountFacade
AccountHome	AccountHomeImpl	AccountPersistence
AccountTransaction	AccountTransactionSession	AccountNode
AccountType	Customer	CustomerEntity
CustomerFacade	CustomerHome	CustomerHomeImpl
CustomerPersistence	CustomerNode	InsufficientFundsException
BankFrame	BankModel	RootNode
TellerApp	AuthenticationException	AuthenticationRole
Authenticator	BaseEntity	BaseFacade
BaseHome	BaseSession	ConfigurationException
Entity	FacadeReuseException	FindException
Home	Identifier	JDBCAuthenticator
JDBCGenerator	JDBCJoin	JDBCSupport
JDBCTransaction	JDBCTransactionImpl	LookupException
LWPProperties	Memento	ObjectServer
ObjectServerImpl	PersistenceException	PersistenceSupport
Persistent	SearchBinding	SearchBoolean
SearchCriteria	SearchOperator	SequenceException
SequenceGenerator	Session	Transaction
TransactionException	WorkerThread	ClientIterator
DistributedIterator	DistributedIteratorImpl	DistributedList
FifoStack	LifoStack	PropertyReader
Stack		

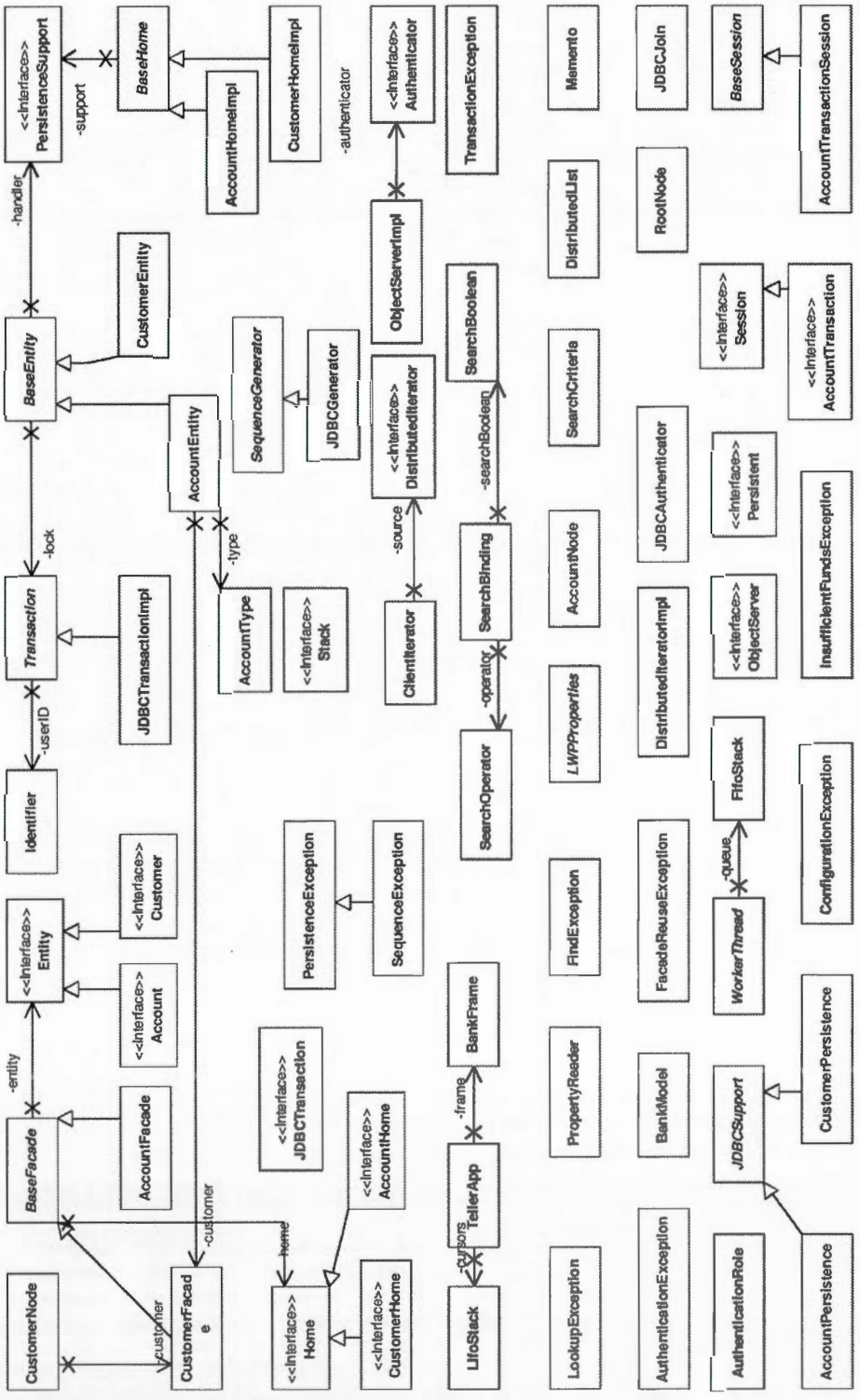

Figure A.1 Le PSM de Banking JDBC

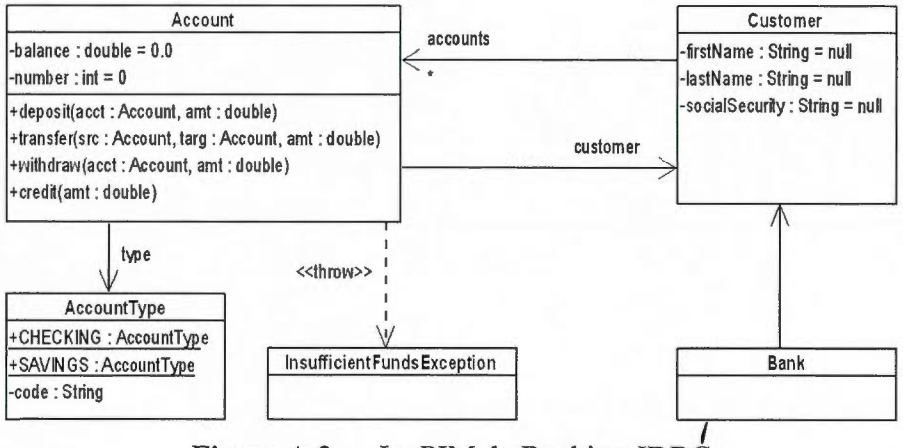

Figure A.2 Le PIM de Banking JDBC

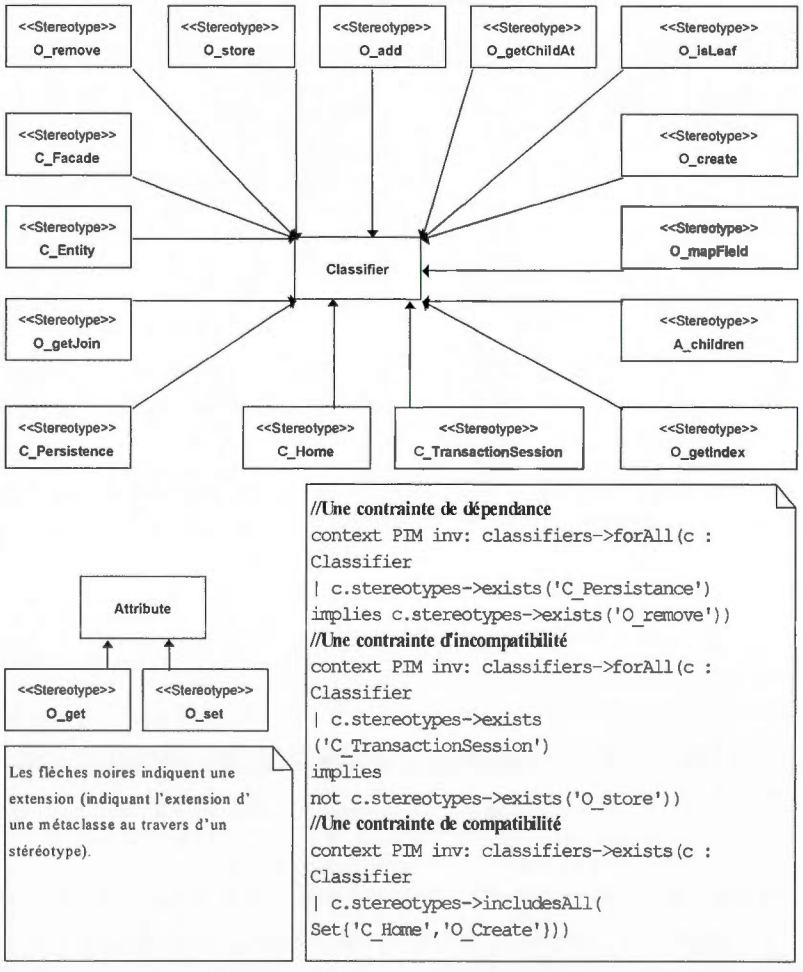

Figure A.3 Une partie du profil UML de la plate-forme d'implémentation de Banking JDBC

APPENDICE B

PSEUDO-CODE

B.1 PHASE DE DÉCOUVERTE DU PROFIL UML DU MODÈLE DE PLATE-FORME

B.1.1 Découverte des PSCs de classificateur

Règle 5.1

```
Découverte des PSCs de classificateur par LSI
   grouper les classificateurs du PSM avec LSI
   pour chaque groupe qui contient au moins deux classificateurs ayant des noms (non qualifiés)
   différents faire
     si les noms des classificateurs du groupe ont au moins un patron commun qui n'est pas le nom
     d'un classificateur du PSM alors
       P := patron commun le plus long dans le nom des classificateurs du groupe
       si P n'est pas le nom d'un PSC de classificateur déjà trouvé alors
         ajouter un nouveau PSC de classificateur nommé d'après P
         associer le groupe au PSC
       sinon si le groupe est différent de tous les groupes déjà associés au PSC de classificateur
       nommé d'après P alors
         associer le groupe au PSC
       fin si
     sinon
       ajouter un PSC de classificateur au nom indéfini
       associer le groupe au PSC
     fin si
   fin pour
Fin découverte des PSCs de classificateur par LSI

Découverte des PSCs de classificateur par la détection de clones
   grouper les opérations du PSM avec la détection de clones
   pour chaque groupe faire
     sortir les classificateurs reliés aux opérations des groupes
     s'il y a au moins deux classificateurs ont des noms (non qualifiés) différents alors
         P := patron commun le plus dans le nom des classificateurs du groupe
         si P n'est pas le nom d'un PSC de classificateur déjà trouvé alors
           ajouter un nouveau PSC de classificateur nommé d'après P
           associer le groupe au PSC
         sinon si le groupe est différent de tous les groupes déjà associés au PSC de classificateur
         nommé d'après P alors
           associer le groupe au PSC
         fin si
       sinon
         ajouter un PSC de classificateur au nom indéfini
         associer le groupe au PSC
       fin si
     fin si
   fin pour
Fin découverte des PSCs de classificateur par la détection de clones
```

Règle 5.2

```
Filtrage des PSCs de classificateur
   pour chaque couple de PSC de classificateur C1 et C2 faire
     si le nom de C1 est une sous-chaîne du nom de C2 et que le groupe des classificateurs reliés à C2
     est un sous-groupe de celui relié à C1 alors
       enlever C2 de la liste des PSCs
     fin si
   fin pour
Filtrage des PSCs de classificateur
```

Règle 5.3

```
Filtrage des PSCs de classificateur
   pour chaque PSC de classificateur C1 faire
     si C1 est relié à un aucun PIC de classe alors
       enlever C1 de la liste des PSCs
     fin si
   fin pour
Filtrage des PSCs de classificateur
```

B.1.2 Découverte des PSCs d'attribut

Règle 5.4

```
Découverte des PSCs d'attribut
  pour chaque classificateur du PSM C relié à un PIC de classe faire
    pour chaque attribut A de C faire
      si A appartient à tous les classificateurs d'un sous-groupe d'un PSC de classificateur alors
        si le nom de A n'est pas le nom d'un PSC d'attribut déjà trouvé alors
            ajouter un nouveau PSC d'attribut nommé d'après A
            créer un nouveau groupe lié au PSC et contenant A
        sinon
          ajouter A au groupe lié à son PSC d'attribut homonyme
        fin si
      fin si
    fin pour
  fin pour
Fin découverte des PSCs d'attribut
```

B.1.3 Découverte des PSCs d'opération

Règle 5.5

```
Lier les opérations du PSM aux attributs
  pour chaque classificateur du PSM C1 relié à un PIC de classe faire
    pour chaque opération O de C1 faire
      pour chaque classificateur du PSM C2  faire
        si (C2 = C1) ou (C1 est lié au PIC de classe P et C2 appartient au groupe basé sur le patron du
        nom de P) ou (C1 a un parent (direct ou indirect) S et  S est lié à un PIC de classe P et C2
        appartient au groupe basé sur le patron P)  alors
        pour chaque attribut A de C2 faire
          si le nom de A  est le nom d'attribut d'une des classes répondant aux conditions précédentes
          alors
            O est lié à A
          fin si
        fin pour
      fin pour
    fin pour
  fin pour
Fin lier opérations du PSM aux attributs


Découverte des PSCs d'opération
    pour chaque PIC de classe P  faire
      pour chaque classificateur du PSM C relié à P faire
        pour chaque opération O de C faire
          si O n'est pas liée à un attribut faire
              // Le cas du constructeur, par exemple il va trouver le PSC Home pour le constructeur d'Account
              Home
              si le nom de O est le même que celui de C et différant de celui de P alors
                ajouter un nouveau PSC d'opération nommé en enlevant du nom de C celui de P
              sinon si O appartient à  tous les classificateurs d'un des sous-groupes d'un PSC de
              classificateur alors
                ajouter un nouveau PSC d'opération nommé d'après O
            fin si
          fin si
      fin pour
    fin pour
Fin découverte des PSCs d'opération
```

Règle 5.6

```
Découverte des PSCs d'opération par LSI
   grouper les classificateurs du PSM avec LSI
   pour chaque groupe G faire
     si les classificateurs contenant les opérations de G ont au moins deux noms (non qualifiés)
     différents alors
       pour chaque verbe présent dans les classificateurs du PSM faire
         chercher toutes les opérations reliées au verbe dans le groupe
         pour chaque opération reliée au verbe faire
           si le nom de l'opération contient celui d'un attribut A d'un classificateur du groupe G ou le
           nom d'un PIC de classe alors
             ajouter un PSC d'opération nommé d'après le verbe
             associer le groupe G au PSC
             noter cette opération comme reliée à l'attribut A
           sinon si l'opération est marquée comme reliée à un attribut alors
             ajouter un PSC d'opération nommé d'après le verbe
             associer le groupe G au PSC
             noter cette opération comme reliée au verbe
           sinon si l'opération n'est pas marquée comme reliée à un classificateur ou une opération
           alors
             ajouter un PSC d'opération nommée d'après l'opération
             associer le groupe G au PSC
           fin si
         fin pour
       fin pour
     fin si
   fin pour
Fin découverte des PSCs d'opération par LSI
```

Règle 5.7

```
Découverte des PSCs d'opération par la détection de clones
   grouper les opérations du PSM par la détection de clones
   pour chaque groupe G d'opérations clonées faire
      si les classificateurs contenant les opérations de G ont au moins deux noms (non qualifiés)
      différents alors
        si les noms des opérations sont les mêmes alors
          si ce nom commun n'est pas celui d'un PSC déjà présent alors
            ajouter un PSC d'opération nommé d'après ce nom
            associer le groupe G au PSC
          sinon
            associer le groupe G au PSC déjà présent
          fin si
        sinon si les noms des opérations contiennent un verbe commun alors
          si un PSC d'opération nommée d'après le verbe n'existe pas déjà alors
            ajouter un PSC d'opération nommé d'après le verbe
          fin si
          associer le groupe G au PSC
        sinon
          ajouter un PSC d'opération au nom indéfini
          associer le groupe G au PSC
        fin si
      fin si
   fin pour
Fin découverte des PSCs d'opération par la détection de clones
```

B.1.4 Découverte des PSCs de paramètre

Règle 5.8

```
Découverte des PSCs de paramètre
  pour chaque classificateur du PSM C1 relié à un PIC de classe C2faire
    pour chaque opération O de C faire
      pour chaque paramètre P de O faire
      si (il n'existe pas un attribut A1 de C1 où A1 est lié à un attribut de C2 ou à une extrémité
      d'association de C2 et A1 est relié à P par l'analyse de flux de données)
        et (il n'existe pas un attribut ou d'extrémité d'association de C2 dont le nom est dans le nom de
      P)
        et (le nom de C2 ne figure pas dans le nom de P) alors
            si un PSC de paramètre nommé d'après le nom de  P n'existe pas déjà alors
              ajouter un PSC de paramètre nommé d'après P
            fin si
            associer le P au PSC
          fin si
        fin si
      fin si
    fin pour
  fin pour
Fin découverte des PSCs de paramètre
```

B.1.5 Découverte des contraintes entre les PSCs

Règle 5.9

```
Découverte des contraintes de dépendance
    pour chaque PSC de classificateur CC, sortir l'ensemble des classificateurs EC l'implémentant
    faire
      pour chaque PSC d'attribut CA, sortir l'ensemble des classificateurs EA l'implémentant faire
        si EA contient  tous les classificateurs de EC alors
          ajouter une contrainte de dépendance entre CC et CA
        fin si
    fin pour
    pour chaque PSC d'opération CO, sortir l'ensemble des opérations EO l'implémentant faire
      pour chaque PSC de paramètre CP, sortir l'ensemble des opérations EP l'implémentant faire
        si EO contient toutes les opérations de EP alors
          ajouter une contrainte de dépendance entre CO et CP
        fin si
    fin pour
Fin découverte des contraintes de dépendance
```

Règle 5.10

Découverte des contraintes de compatibilité
pour chaque PSC C1, sortir l'ensemble des éléments E1 l'implémentant **faire**
pour chaque PSC C2, sortir l'ensemble des éléments E2 l'implémentant **faire**
si E1 contient certains éléments de E2 **alors**
ajouter une contrainte de compatibilité entre C1 et C2
fin si
fin pour
Fin découverte des contraintes de compatibilité

Règle 5.11

Découverte des contraintes d'incompatibilité
pour chaque PSC C1, sortir l'ensemble des éléments E1 l'implémentant **faire**
pour chaque PSC C2, sortir l'ensemble des éléments E2 l'implémentant **faire**
si E1 ne contient aucun élément de E2 **alors**
ajouter une contrainte d'incompatibilité entre C1 et C2
fin si
fin pour
Fin découverte des contraintes d'incompatibilité

Règle 5.12

Découverte des contraintes de raffinement entre les PSCs
pour chaque PSC P qui est associé à plus d'un groupe **faire**
si les groupes sont basés sur LSI **et** au moins deux groupes implémentent ou héritent d'un classificateur différent C qui n'est pas lié à un PIC de classe **alors**
tous les classificateurs C_i représentent un nouveau PSC du même type, et il est nommé du nom du classificateur C_i (après avoir enlevé toute occurrence d'un mot du vocabulaire du domaine du problème).
chaque différence devient un nouveau PSC C2 de type classificateur
ajouter une contrainte de raffinement entre chaque nouveau PSC et P
fin si
fin pour
Fin découverte des contraintes de raffinement entre les PSCs

Règle 5.13

```
Découverte des contraintes de raffinement entre les PSCs
   pour chaque PSC C1 qui est associé à plusieurs groupes faire
      sinon si les groupes sont des groupes des clones d'opérations alors
         trouver les différences entre ces clones
         chaque différence devient un nouveau PSC C2 de type classificateur
         ajouter une contrainte de raffinement entre C2 et C1
      fin si
   fin si
   fin pour
Fin découverte des contraintes de raffinement entre les PSCs
```

B.2 PHASE DE DÉCOUVERTE DU MODÈLE INDÉPENDANT DE PLATE-FORME

B.2.1 Découverte des PICs de classe

Règle 6.1

```
Découverte des PICs de classe
   pour chaque classificateur du PSM faire
     nomCandidat := retirer le nom des PSCs de classificateur du nom de la classe
     si nomCandidat est obtenue pour au moins deux classificateurs du PSM ayant des noms non
     qualifiés différents alors
       ajouter un PIC de classe nommée d'après nomCandidat
     fin si
     si la classe n'appartient à aucun des groupes associés aux PSCs contenus dans son nom alors
       créer un nouveau groupe contenant la classe
       associer le groupe au PSC
     fin si
   fin pour
Fin découverte des PICs de classe
```

Règle 6.2

```
Découverte des PSCs de classificateur orphelins reliés à des PICs de
 classe
   pour chaque PIC de classe CM faire
     si on peut trouver des classificateurs C2 non reliés à un PIC de classe et qui contiennent le nom
     du PIC de classe CM alors
       pour chaque classe de C2 faire
         nomCandidat := retirer de son nom celui de CM
         si nomCandidat n'est pas déjà le nom d'un PIC de classe alors
           ajouter un nouveau PSC de classificateur nommé d'après nomCandidat
           créer un nouveau groupe contenant la classe C2
           associer le nouveau groupe au nouveau PSC
         fin si
       fin pour
     fin si
   fin pour
   si trouvé d'autres PSCs de classificateur alors
    redémarrer l'algorithme à partir de la règle 5.2
   fin si
Fin découverte des PSCs de classificateur orphelins reliés à des
PICs de classe
```

Règle 6.3

```
Découverte des classificateurs orphelins reliés à un PSC de
 classificateur
   pour chaque classificateur du PSM orphelin faire
     si on peut trouver dans son nom celui d'un PSC de classificateur alors
       nomCandidat := retirer du nom du classificateur celui du PSC de classificateur
       si nomCandidat ne contient pas un PSC de classificateur alors
         ajouter un nouveau PSC de classificateur nommé d'après nomCandidat
       fin si
     fin si
   fin pour
Fin découverte des classificateurs orphelins reliés à un PSC de
classificateur
```

B.2.2 Découverte des PICs d'attribut

Règle 6.4

```
Découverte des PICs d'attribut
  pour chaque classificateur du PSM C relié à un PIC de classe P faire
    pour chaque attribut A de C faire
      si A n'est pas lié à un PSC d'attribut alors
        ajouter un nouveau PSC d'attribut nommé d'après A qui est lié à P
      fin si
    fin pour
  fin pour
Fin découverte des PICs d'attribut
```

B.2.3 Découverte des PICs d'opération

Règle 6.5

```
Découverte des PICs d'opération
  pour chaque classificateur du PSM C relié à un PIC de classe P faire
    pour chaque opération O de C faire
      si O n'est pas une opération qui est liée à un PSC d'opération alors
        ajouter un nouveau PSC d'opération nommé d'après O qui est lié à P
      fin si
    fin pour
  fin pour
Fin découverte des PICs d'opération
```

B.2.4 Filtrage des PICs de classe

Règle 6.7

```
Filtrage des PICs de classe
   pour chaque PIC de classe C faire
      si cette classe ne contient ni PIC de d'attribut ni PIC d'opération alors
        si le nom de cette classe contient un patron qui est celui d'un PIC de classe alors
          enlever C de la liste des PICs de classe
          pour chacune des classes qui ont servie à trouver ce PIC de classe faire
            si on peut trouver un ensemble de patrons Ps de C qui ne sont ni un PSC de classificateur,
            ni un PIC de classe et ni un classificateur du PSM alors
                ajouter le patron de P qui contient le plus de classe dans son groupe relié (le groupe
                doit contenir 2 classificateurs et plus qui ne sont pas reliés au PIC de classe inclus) ni
                un sous-patron du nom des PSCs détectés dans les classes reliées au PIC de classe.
                créer un nouveau groupe pour chaque classe du groupe
            fin si
          fin pour
        fin si
      fin si
   fin pour
   si trouvé d'autres PSCs de classificateur alors
    redémarrer l'algorithme à partir de la règle 5.2
   fin si
Fin filtrage des PICs de classe
```

B.2.5 Découverte des relations entre les éléments du PIM

Règle 6.8

```
Découverte des associations
   pour chaque PIC de classe C1 faire
      pour chaque PIC d'attribut A lié à C1 faire
        si le type de A est un PIC de classe C2 et C1!=C2 alors
          enlever A des PICs d'attribut liés à C1
          ajouter une association de C1 vers C2
        fin si
      fin pour
   fin pour
Fin découverte des associations
```

Règle 6.9

```
Découverte des héritages
   pour chaque classificateur du PSM C1 relié à un PIC de classe P1 faire
     pour chaque classificateur du PSM C2 relié à un PIC de classe P2 faire
        si il existe une relation d'héritage C1 et  C2 alors
          créer une relation d'héritage entre P1 et P2
        fin si
      fin pour
   fin pour
Fin découverte des héritages
```

Règle 6.10

```
Découverte des dépendances
   pour chaque classificateur du PSM C1 relié à un PIC de classe P1 faire
     pour chaque classificateur du PSM C2 relié à un PIC de classe P2 faire
        si C1 contient une opération qui jette une exception du type de C2 alors
          créer une relation de dépendance entre P1 et P2
        fin si
      fin pour
   fin pour
Fin découverte des dépendances
```

B.3 PHASE DE DÉCOUVERTE DES CARTES DE DÉRIVATION

B.3.1 Sous-phase 1 : Découverte des cartes de dérivation

B.3.1.1 Étape 1.1 : Découverte des liens entre les éléments du PSM et du PIM.

Règle 7.1

```
Découverte des liens entre les classificateurs du PSM et les PICs
   pour chaque classificateur C1 du PSM faire
      C1.linkedPIC := le PIC de classe avec le nom le plus long contenu dans son nom s'il existe
      sinon null
   fin pour
Fin découverte des liens entre les classificateurs du PSM et les
PICs
```

Règle 7.2

```
Découverte des liens entre les attributs du PSM et les PICs
   pour chaque classificateur C du PSM relié à un PIC de classe P faire
     pour chaque attribut A de C faire
        A.linkedPIC :=
           l'attribut de P avec le nom le plus long contenu dans son nom
          ou
           l'extrémité d'association de P avec le nom le plus long contenu dans son nom
          ou
           P si son nom est contenu dans celui de A
          ou
           null
     fin pour
   fin pour
Fin découverte des liens entre les attributs du PSM et les PICs
```

Règle 7.3

```
Découverte des liens entre les opérations du PSM et les PICs
   pour chaque classificateur C du PSM relié à un PIC de classe P faire
     pour chaque opération O de C faire
       O.linkedPIC :=
           P si son nom est contenu dans celui de A
         ou
           l'opération de P avec le nom le plus long contenu dans son nom
         ou
           l'attribut de P avec le nom le plus long contenu dans son nom
         ou
           l'extrémité d'association de P avec le nom le plus long contenu dans son nom
         ou
           null
     fin pour
   fin pour
Fin découverte des liens entre les opérations du PSM et les PICs
```

Règle 7.4

```
Découverte des liens entre les paramètres du PSM et les PICs
   pour chaque classificateur C1 du PSM relié à un PIC de classe C2 faire
     pour chaque opération O de C où O.linkedPIC != null faire
       pour chaque paramètre P de O faire
       P.linkedPIC :=
              le paramètre P2 d'une opération O2 de P dont le nom est le patron le plus long
              figurant dans le nom de P
            ou
              l'attribut A2 de C2 où A1 est lié à un attribut A2 (ou une extrémité d'association AE)
              de C2 et A1 est relié à P par l'analyse de flux de données
            ou
              l'extrémité d'association AE de C2 dont le nom est le patron le plus long figurant dans
              le nom de P
            ou
              l'attribut A de P dont le nom est le patron le plus long figurant dans le nom de P
            ou
              C2 si son nom est le patron le plus long figurant dans le nom de P
            ou
              null
       fin pour
     fin pour
   fin pour
Fin découverte des liens entre les paramètres du PSM et les PICs
```

B.3.1.2 Étape 1.2 : Découverte des liens entre les éléments du PSM et les PSCs.

Règle 7.5

```
Découverte des liens entre les classificateurs du PSM et les PSCs
  pour chaque classificateur C1 du PSM faire
    si C1 . linkedPIC = null et il existe un PSC de classificateur PS homonyme alors
         C1.primaryPSC := PS
    sinon si C1 est une interface et un C1.name = C1 . linkedPIC.name alors
       C1. primaryPSC := "PICInterface"
    sinon si C1 est une classe et un C1.name = C1 . linkedPIC.name alors
       C1. primaryPSC := "PICClasse"
    sinon si C1.linkedPIC.name != null et C1.name n'est pas dérivé de C1 . linkedPIC.name et il
    existe un PSC de classificateur PS dont le nom est le patron le plus long figurant dans le nom de
    C1.linkedPIC.name alors
       C1. primaryPSC := PS
    sinon
       C1. primaryPSC := null
    fin si
  fin pour
Fin découverte des liens entre les classificateurs du PSM et les
PSCs
```

```
Découverte des liens entre les attributs du PSM et les PSCs
  pour chaque attribut A1d'un classificateur du PSM C1  faire
    si A1 . linkedPIC = null et il existe un PSC d'attribut PS homonyme  alors
        A1.primaryPSC := PS
    sinon si A1.name = A1 . linkedPIC.name alors
        A1. primaryPSC := "PICAttribute"
    sinon si A1.linkedPIC.name != null et A1.name n'est pas dérivé de A1 . linkedPIC.name et il
    existe un  PSC d'attribut PS dont le nom est le patron le plus long figurant dans le nom de
    A1.linkedPIC.name alors
        A1. primaryPSC  := PS
    sinon
        A1. primaryPSC  := null
    fin si
  fin pour
Fin découverte des liens entre les attributs du PSM et les PSCs


Découverte des liens entre les opérations du PSM et les PSCs
  pour chaque opération O1d'un classificateur du PSM C1  faire
    si O1 . linkedPIC = null et il existe un PSC d'opération PS homonyme alors
        O1.primaryPSC := PS
    sinon si O1.name = O1 . linkedPIC.name alors
        O1. primaryPSC := "PICOperation"
    sinon si O1.linkedPIC.name != null et O1.name n'est pas dérivé de O1 . linkedPIC.name et il
    existe un  PSC d'opération PS dont le nom est le patron le plus long figurant dans le nom de
    O1.linkedPIC.name alors
        O1. primaryPSC := PS
    sinon
        O1. primaryPSC := null
    fin si
  fin pour
Fin découverte des liens entre les opérations du PSM et les PSCs
```

```
Découverte des liens entre les paramètres du PSM et les PSCs
  pour chaque paramètre P1  d'un classificateur du PSM C1  faire
    si P1 . linkedPIC = null et il existe un PSC de paramètre PS homonyme  alors
        P1.primaryPSC := PS
    sinon si P1.name = P1 . linkedPIC.name alors
        P1. primaryPSC := "PICParameter"
    sinon si P1.linkedPIC.name != null et P1.name n'est pas dérivé de P1 . linkedPIC.name et il
    existe un  PSC d'paramètre PS dont le nom est le patron le plus long figurant dans le nom de
    P1.linkedPIC.name alors
        P1. primaryPSC  := PS
    sinon
        P1. primaryPSC  := null
    fin si
  fin pour
Fin découverte des liens entre les paramètres du PSM et les PSCs
```

B.3.1.3 Étape 1.3 : Création des cartes de dérivation

Règle 7.6

```
Création des cartes de dérivation de classificateur
pour chaque classificateur C1 du PSM dans PSM faire
      si C1.linkedPIC != null alors
            créer une carte de dérivation de classificateur CD
            CD.origin := C1.linkedPIC
            CD.destination := C1
            CD.primaryPSC := C1.PSC
            CD.variationPSCs := ϕ
      fin si
   fin pour
Fin création des cartes de dérivation de classificateur
```

```
Création des cartes de dérivation d'attribut
   pour chaque carte de dérivation de classificateur CDC  faire
      pour chaque attribut A dans CDC.destination.attributeList faire
         créer une carte de dérivation d'attribut CDA
         si A.linkedPIC != null alors
           CDA.origin := A.linkedPIC
         sinon
           CDA.origin := null
         fin si
         CDA.destination := A
         CDA.PrimaryPSC := A.PSC
         CDA.AuxiliaryPSC := ϕ
      fin pour
   fin pour
Fin création des cartes de dérivation d'attribut


Création des cartes de dérivation d'opération
   pour chaque carte de dérivation de classificateur CDC  faire
      pour chaque opération O dans CDC.destination.operationList faire
         créer une carte de dérivation d'opération CDO
         si O.linkedPIC != null alors
           CDO.origin := O.linkedPIC
         sinon
           CDO.origin := null
         fin si
         CDO.destination := O
         CDO.PrimaryPSC := O.PSC
         CDO.AuxiliaryPSC := ϕ
      fin pour
   fin pour
Fin création des cartes de dérivation d'opération

Création des cartes de dérivation de paramètres
   pour chaque carte de dérivation d'opération CDO  faire
      pour chaque paramètre P dans CDO.destination.parameterList faire
         créer une carte de dérivation de paramètre CDP
         si P.linkedPIC != null alors
           CDP.origin := O.linkedPIC
         sinon
           CDP.origin := null
         fin si
         CDP.destination := P
         CDP.PrimaryPSC := P.PSC
         CDP.AuxiliaryPSC := ϕ
      fin pour
   fin pour
Fin création des cartes de dérivation de paramètres
```

B.3.2 Sous-phase 2 : Création des modèles de transformation

B.3.2.1 Étape 2.1 : Création de modèles de transformation

Règle 7.7

```
Création des modèles de transformation de classificateur
   pour chaque carte de dérivation de classificateur CDC faire
      si CDC est une interface alors
         créer un modèle de transformation d'interface MT
      sinon
         créer un modèle de transformation de classe MT
         MT.implementedInterface :=
         contextualizeClassifierMappingTemplateImplementedInterface(CDC)
      fin si
      réglerChampsDeBase(CDC, MT)
      MT.name := contextualizeClassifierMappingTemplateName(CDC)
      MT.packageName := contextualizeClassifierMappingTemplatePackageName(CDC)
      MT.superClassifierName := contextualizeClassifierMappingTemplateSuperClassifier(CDC)
      MT.ImplementedInterface := contextualizeClassifierMappingTemplateImplementedInterface(CDC)
      MT.importPackage= contextualizeClassifierMappingTemplateImportPackage(CDC)
      MT.visibility := CDC.destination.visibility
   fin pour aDerivationClassifierMapElement
Fin création des modèles de transformation de classificateur


Régler champs de base (CD, MT)
   MT.primaryImplementedPSC := CD.primaryPSC
   MT.auxiliaryImplementedPSCs := CD.auxiliaryPSCs
   MT.result := CD.destination
   si CD.origin n'est pas null alors
      MT.context := CD.origin.type
   sinon
      si CD.destination est un attribut ou une opération et CD.destination.containingClassifier.origin
      n'est pas null alors
         MT.context := CD.destination.containingClassifier.type
      sinon si CD.destination est un paramètre et
      CD.destination.containingOperation.containingClassifier.origin n'est pas null alors
        MT.context := CD.destination.containingOperation.containingClassifier.origin
      fin si
   fin si
   MT.PSMElements.add(aDerivationMapElement.qualifiedName)
   CD.mappingTemplate := MT
Fin régler champs de base
```

```
Création des modèles de transformation d'attribut
   pour chaque carte de dérivation d'attribut CDA faire
     créer un modèle de transformation d'attribut MT
     réglerChampsDeBase(CDA, MT)
     MT.name := contextualizeAttributeMappingTemplateName(CDA)
     MT.visibility := contextualizeAttributeMappingTemplateVisibility(CDA)
     MT.changeability := contextualizeAttributeMappingTemplateChangeability(CDA)
     MT.type := contextualizeAttributeMappingTemplateType(CDA)
     MT.initialValue := contextualizeAttributeMappingTemplateInitialValue(CDA)
     fin pour
Fin création des modèles de transformation d'attribut

Création des modèles de transformation d'opération
   pour chaque carte de dérivation d'attribut CDO faire
     créer un modèle de transformation d'opération MT
     réglerChampsDeBase(CDO, MT)
     MT .name := contextualizeOperationMappingTemplateName(CDO)
     MT.returnType := contextualizeOperationMappingTemplateReturnType(CDO)
     MT.visibility := contextualizeOperationMappingTemplateVisibility (CDO)
     MT.raisedException := contextualizeOperationMappingTemplateRaisedException (CDO)
     MT.specification := contextualizeOperationMappingTemplateSpecification (CDO)
   fin pour
Fin Création des modèles de transformation d'opération

Création des modèles de transformation de paramètre
   pour chaque carte de dérivation de paramètre CDP faire
     créer un modèle de transformation de paramètre MT
     réglerChampsDeBase(CDO, MT)
     MT.name := contextualizeParameterMappingTemplateName(CDP)
     MT.directionKind := contextualizeParameterMappingTemplateDirectionKind (CDP)
     MT.type := contextualizeParameterMappingTemplateType(CDP)
   fin pour
Fin création des modèles de transformation de paramètre
```

B.3.2.2 Étape 2.2 : Paramétrage des modèles de transformation.

Règle 7.8

```
ContextualizeClassifierMappingTemplateName(CDC) : String
 retourne CDC.destination.name.replace(CDC.origin.name, "context.name")
Fin contextualizeClassifierMappingTemplateName

contextualizeClassifierMappingTemplatePackageName(CDC) : String
 resultat := CDC.destination.packageName.replace(CDC.origin.name, "context.name")
 retourne resultat.replace(CDC.origin.packageName, "context.packageName")
Fin contextualizeClassifierMappingTemplatePackageName
```

```
contextualizeClassifierMappingTemplateSuperClassifier(CDC) : String
  resultat := CDC.destination.superclassifierName.replace(CDC.origin.Name, "context.Name")
  resultat := resultat.replace(CDC.origin.packageName, "context.packageName")
  resultat := resultat.replace(CDC.origin.superclassifierName, "context.superClassifierName")
  retourne resultat.replace(CDC.origin.superClassifier.packageName,
  "context.superClassifier.packageName")
Fin contextualizeClassifierMappingTemplateSuperClassifier


contextualizeClassifierMappingTemplateImplementedInterface     (CDC) :
 Vector
  resultat := nouveau Vector
  pour chaque Interface I  CDC.destination.implementedInterface faire
    nom := I.name.replace(CDC.origin.name, "context.name")
    nom := nom.replace(CDC.origin.packageName, "context.packageName")
    nom := nom.replace(CDC.origin.superClass.name, "context.superclass.name")
    ajouter nom à resultat
  fin pour
  retourne resultat
Fin contextualizeClassifierMappingTemplateImplementedInterface
```

```
contextualizeClassifierMappingTemplateImportPackage (CDC) : Vector
 si CDC.origin == null alors
   retourne CDC.destination.importPackage
 fin si
 pour chaque importPackage  IP CDC.destination.importPackage faire
   IP := IP.replace (CDC.origin.packageName, "context.packageName")
   IP := IP.replace(CDC.origin.name, "context.name")
   pour chaque AssociationEnd AE CDC.origin.associationEndList faire
     IP := IP.replace(AE.type.name, "context.associationEnds['AE'].type.name")
     IP := IP.replace(AE.type.packageName, "context.associationEnds['AE'].type.packageName")
   fin pour
   pour chaque Dependency D CDC.origin.dependency faire
     IP := IP.replace(origin.dependencies['D'].target.name, "context.dependencies['D'].target.name")
     IP := IP.replace(origin.dependencies['D'].target. packageName, "context.dependencies['D'].target.packageName")
   fin pour
   pour chaque Attribute A CDC.origin.attributeList faire
     IP := IP.replace(A.type, "A.type")
   fin pour
   pour chaque AssociationEnd AE CDC.origin.associationList faire
     IP := IP.replace(AE.type, "AE.type")
   fin pour
   IP := IP.replace( CDC.origin.superclassifier.name, "context.superclass.name")
   // Le package du parent de la classe d'origine
   IP := IP.replace(CDC.origin.superclassifier.packageName, "context.superclass.packageName")
   // Le parent  de la classe dérivée
   IP := IP.replace(CDC.destination.superclassifier.name, "this.superclass.name")
   // Les implements de la classe dérivée
   pour chaque Interface I CDC.destination. implementedInterface faire
     IP := IP.replace(I, "this.ImplementedInterface['I']")
   fin pour
   pour chaque Attribute A CDC.destination.attributeList faire
     IP := IP.replace(A.type, "this.Attribute['A'].type")
   fin pour
   pour chaque Operation O CDC.destination.operation faire
     IP := IP.replace(O.type, "this.Operation['O']..type")
     pour chaque Parameter P dans O.parameter faire
       IP := IP.replace(P r.type, " this.Operation['O'].Parameter['P'].type "
     fin pour
   fin pour
 fin pour importPackage
Fin contextualizeClassifierMappingTemplateImportPackage
```

Règle 7.9

```
contextualizeAttributeMappingTemplateName(CDA) : String
 si CDA.origin = null alors
   retourne CDA.destination.name
 sinon
   retourne CDA.destination.name.replace(CDA.origin.name, "context.name")
 fin si
Fin contextualizeAttributeMappingTemplateName


contextualizeAttributeMappingTemplateVisibility(CDA) : String
 resultat =CDA.destination.visibility
 si CDA.origin est un attribut ou une extrémité d'association alors
   retourne CDA.destination.visibility.replace(CDA.origin.visibility, "context.visibility")
 fin si
 retourne resultat
Fin contextualizeAttributeMappingTemplateVisibiltiy


contextualizeAttributeMappingTemplateChangeability(CDA) : String
 resultat = CDA.destination.changeability
 si CDA.origin est un attribut ou un extrémité d'associationalors
   resultat = resultat.replace( CDA.origin.changeability, "context.changeability")
 fin si
 retourne resultat;
Fin contextualizeAttributeMappingTemplateChangeability


contextualizeAttributeMappingTemplateType(CDA) : String
 resultat = CDA.destination.type
 si CDA.origin est un classificateur alors
   resultat = resultat.replace( CDA.origin.name, "context.name")
 sinon si CDA.origin est un attribut alors
   resultat = resultat.replace( CDA.origin.type, "context.type ")
 sinon si CDA.origin est une extrémité d'association alors
   resultat = resultat.replace( CDA.origin.type.name, "context.type.name")
 sinon si CDA.origin = null et CDA.destinationClasifier n'est pas null alors
   resultat = resultat.replace( "origin.classifier.name", "context.name")
 fin si
 retourne resultat
Fin contextualizeAttributeMappingTemplateType
```

```
contextualizeAttributeMappingTemplateInitialValue(CDA) : String
  resultat := CDA.destination.initialValue
  si CDA.origin est un attribut alors
    resultat = resultat.replace( CDA.origin.initialValue, "context.initialValue")
  sinon si CDA.origin est une extrémité d'association ou un classificateur alors
    resultat = resultat.replace( CDA.origin.initialValue, "context.PSCs[PSC].PRP['initialValue'].value")
  sinon si CDA.origin = null et CDA.destinationClasifier n'est pas null alors
    si resultat contains CDA.destinationClasifier.name alors
       resultat.replace("origin.classifier.name", "context.name")
    sinon
       resultat.replace(CDA.origin.initialValue, "context.PSCs[PSC].PRP['initialValue'].value")
    fin si
  fin si
  retourne resultat
Fin contextualizeAttributeMappingTemplateInitialValue
```

Règle 7.10

```
contextualizeOperationMappingTemplateName(CDO):String
  si CDO.origin = null alors
    retourne CDO.destination.name
  fin si
  retourne CDO.destination.name.replace(CDO.origin.name, "context.name")
Fin contextualizeOperationMappingTemplateName


contextualizeOperationMappingTemplateVisibility (CDO) : String
  resultat := CDO.destination.visibility
  si CDO.origin est un opération alors
        resultat := resultat.replace( CDO.origin.visibility, "context.visibility")
  fin si
  retourne resultat
Fin contextualizeOperationMappingTemplateVisibility
```

```
contextualizeOperationMappingTemplateReturnType(CDO):String
resultat := CDO.destination.returnType
 si CDO.origin est une classe alors
   resultat := resultat.replace( origin.name, "context.name")
 sinon si CDO.origin est un attribut alors
   resultat := resultat.replace( CDO.origin.type, "context.type")
 sinon si CDO.origin est une extrémité d'association alors
   resultat := resultat.replace( CDO.origin.type.name, "context.type.name")
 sinon si CDO.origin est une opération alors
   resultat := resultat.replace( CDO.origin. returnType, "context.returnType")
 sinon si CDO.origin = null et CDO.destinationClasifier n'est pas null alors
   resultat := resultat.replace( "origin.classifier.name", "context.name")
 fin si
 retourne resultat;
Fin contextualizeOperationMappingTemplateReturnType.

contextualizeOperationMappingTemplateRaisedException (CDO) : Vector
 resultat := CDO.destination.raisedException
 pour chaque currentRaisedException dans resultat faire
   si anOperatioDerivationMapElement.origin est un attribut ou une extrémité d'association ou une
   opération alors
      pour chaque Dependency D dans
      anOperatioDerivationMapElement.origin.classifier.dependency faire
         importPackage.replace(D.target.name, "context.classifier.dependencies['D'].target.name")
         importPackage.replace(D.target. packageName, "context.classifier.dependencies['D'].target.packageName")
      fin pour
   sinon si CDO.origin est une classe alors
      pour chaque Dependency D dans anOperatioDerivationMapElement.origin.dependency faire
         importPackage.replace( D.target.name, "context.dependencies['D'].target.name")
         importPackage.replace(D.target. packageName, "context.dependencies['D'].target.packageName")
      fin pour
   sinon si anAttributeDerivationMapElement.origin = null et CDO.destinationClasifier n'est pas null
   alors
      pour chaque Dependency D dans
      anOperatioDerivationMapElement.origin.classifier.dependency faire
         importPackage.replace( D.target.name, "context.dependencies['D'].target.name")
         importPackage.replace( D.target. packageName, "context.dependencies['D'].target.packageName")
      fin pour
   fin si
 fin pour
 retourne resultat
Fin contextualizeOperationMappingTemplateRaisedException
```

Règle 7.11

```
contextualizeOperationMappingTemplateSpecification (CDO):String
   resultat := CDO.destination.specification
   destination := CDO.destination
   origin := CDO.origin
   originClassifier := le classificateur d'origine du classificateur dérivé contenant l'opération
   dérivée
   si origin est une opération alors
     resultat := context.PSCs[PSC]. PRP['operationBody'].value
   sinon
     si origin est un attribut ou une extrémité d'association alors
       resultat := contextualizeTokens (origin.name, "context.name", resultat)
       contextualizeVariableDeclarationTypeByAttributeOrAssociationEndType(
       origin,"context.type")
       contextualizeVariableAssignation (origin,"context.")
     sinon si origin est un classificateur alors
       pour chaque attribut A dans origin trié par taille descendante faire
         contextualizeTokens (A.name, "context.attribute[].name", resultat)
       fin pour
       pour chaque associationEnd AE dans origin trié par taille descendante faire
         contextualizeTokens (AE.name, "context.AssociationEnd[].name", resultat
       fin pour
       contextualizeVariableDeclarationTypeByOriginClasseAttributeAndAssociationEndType(
       originClassifier)
       contextualizeVariableAssignationTypeByOriginClasseAttributeAndAssociationEndType(
       originClassifier)
     fin si
   fin si
   resultat := contextualizeOperationMappingTemplateSpecificationWithParameters(destination ,
   resultat)
   resultat := contextualizeSpecificationWithOriginClassifier(destination, PIM, resultat)
   resultat := refineBodyParametrization(CDO, destination, resultat)
   retourne resultat
Fin contextualizeOperationMappingTemplateSpecification.


contextualizeTokens (toReplace, aReplacement, aSpecification)
  pour chaque aToken dans aSpecification faire
    aToken.text := contextualizeReplaceString(aToken.text, toReplace, aReplacement)
  fin pour
Fin contextualizeTokens
```

```
contextualizeVariableDeclarationTypeByAttributeOrAssociationEndType(
 Identifier argIdentifier, String argContextualization)
 pour chaque déclaration D d'une variable V faire
  si V est relié par DFA à argIdentifier alors
   D.type := D.type.replace(argIdentifier.type, argContextualization)
  fin si
 fin pour
Fin contextualizeVariableDeclarationTypeByAttributeOrAssociationEndType


contextualizeVariableDeclarationTypeByOriginClasseAttributeAndAssoci
 ationEndType(Classe originClassifier)
 pour chaque attribut A dans originClassifier faire
  contextualizeVariableDeclarationTypeByAttributeOrAssociationEndType(A,"context.attribute[].ty
  pe")
 fin pour
 pour chaque associationEnd AE dans origin originClassifier faire
  contextualizeVariableDeclarationTypeByAssociationEndOrAssociationEndType(AE,"context.Asso
  ciationEnd[].type")
 fin pour
Fin contextualizeVariableDeclarationTypeByAttributeOrAssociationEndType


contextualizeVariableAssignation (Identifier argIdentifier, String
argContextualization)
 pour chaque assignation A d'une valeur X à une variable V faire
  si V est relié par DFA à argIdentifier alors
   si X est un littéral alors
    si argIdentifier est un attribut et X == argIdentifier.initialValue alors
     X:= X . replace(argIdentifier.initialValue, argContextualization+".initialValue")
    sinon
     X:= argContextualization+".PSCs[PSC].PRP['initialValue'].value))")
    fin si
   sinon si X est une expression de cast et cast type = argIdentifier. type alors
    X:= X . replace(argIdentifier.initialValue, argContextualization+".type")
   fin si
  fin si
 fin pour
Fin contextualizeVariableAssignationTypeByAttributeOrAssociationEndType

contextualizeVariableAssignationTypeByOriginClasseAttributeAndAssoci
ationEndType(Classe originClassifier)
 pour chaque attribut A dans originClassifier faire
    contextualizeVariableAssignation (anAttribute,"context.attribute[]")
 fin pour
 pour chaque associationEnd AE dans origin originClassifier faire
    contextualizeVariableAssignation (anAssociationEnd,"context.AssociationEnd[]")
 fin pour
Fin contextualizeVariableAssignationTypeByAttributeOrAssociationEndType
```

Règle 7.12

```
contextualizeParameterMappingTemplateName(CDP) : String
 si CDP.origin = null alors
   retourne CDP.destination.name
 sinon
   retourne CDP.destination.name.replace(CDP.origin.name, "context.name")
 fin si
Fin contextualizeParameterMappingTemplateName


contextualizeParameterMappingTemplateDirectionKind (CDP) : String
 resultat := CDP.destination.directionKind
 si CDP.origin est un paramètre alors
   si CDP.origin directionKind = CDP.destination.directionKind alors
     resultat := resultat.replace(CDP.origin.directionKind, "context.directionKind")
   fin si
 fin si
retourne resultat
Fin contextualizeParameterMappingTemplateDirectionKind


contextualizeParameterMappingTemplateType(dans CDP) : String
 resultat := CDP.destination.type
 fin si
 resultat := resultat.replace( CDP.origin.name, "context.name")
 si CDP.origin est un un attribut alors
   resultat := resultat.replace( CDP.origin.type, "context.type")
 sinon si si CDP.origin est une extrémité d'association alors
   resultat := resultat.replace( CDP.origin.type.name, "context.type.name")
 sinon si CDP.origin est un classificateur alors
   resultat := resultat.replace( CDP.origin.name, "context. name")
 fin si
 retourne resultat
Fin contextualizeParameterMappingTemplateType
```

B.3.2.3 Étape 2.3 : Généralisation des modèles de transformation.

Sous-étape 2.3.1 : Fusion des modèles de transformation

Règle 7.13

```
Fusion des modèles de transformations
  groupesDeModèleDeTransformationSimilaires := grouper les modèles de
  transformation implémentant le même primaryPSC avec le même context
  pour chaque groupe G dans groupesDeModèleDeTransformationSimilaires faire
    créer un nouveau modèle de transformation du type des éléments de G nommé MT
    MT.primaryImplementedPSC := G.primaryImplementedPSC
    MT.context := G.context
    MT.PSMElements = union de tous les PSMElments des membres de G

    groupesDeModèleDeTransformationIdentiques := grouper les modèles de
    transformation de G dont les propriétés ont des valeurs identiques
    si groupesDeModèleDeTransformationIdentiques contient un seul élément alors
        MT.properties = G.firstElement.properties
    sinon
      pour chaque champ C  dans MT faire
        s'il y a des variations dans les membres de groupesDeModèleDeTransformationSimilaires
        alors
          pour chaque variation V dans C   pour les membres de
          groupesDeModèleDeTransformationSimilaires faire
             nouveauPSC := un nouveau PSC du type de  MT
             nouveauPSC.name := MT.primaryPSC.name
                                +«_Variation_»
                                + (index unic)
             ajouter  nouveauPSC à  MT.implementedPSCsForVariations
             ajouter (nouveauPSC → V) à MT.curentField
          fin pour
        fin si
      fin pour
    fin si
  fin pour
Fin fusion des modèles de transformations
```

Sous-étape 2.3.2 : Généralisation du corps des opérations

Règle 7.14

```
Généraliser corps des modèles de transformation d'opération
  pour chaque corps original CO d'un mode de transformation paramétré MT avec context =
  classe faire
    pour chaque groupe Gi de clones contigus dans CO ayant comme context classe faire
      si les fragments de Gi sont paramétrés avec CTXi alors
        si les fragments de Gi réfèrent à des éléments différents de contexte Ej alors
          Pi = un nouveau PSC du type de CTXi
          Pi.name = MT.primaryImplementedPSC.name + "MergeFragment" + i
          fragmenti = Gi[0].replace(Ej, type d'élément de Ej)
          pour chaque variation var1 between fragmentj faire
            newPRP = create a new PRP
            newPRP.PSC = Pi
            newPRP.name = "varFragmentProperty" + 1
            newPRP.value = var1
          fin pour
          remplacer chaque fragmenti par mergeFragment(CTXi, fragmenti, Pi)
        fin si
      fin si
    fin pour
    pour chaque référence refq à un attribut ou une extrémité d'association dans
    MT.contextualizedBody faire
      Pr = un nouveau PSC du type de refq
      Pr.name = "element_" + r
      remplacer refq par element(refq.type, refq, Pr)
    fin pour
  fin pour
Fin généraliser corps des modèles de transformation d'opération
```

Sous-étape 2.3.4 : Raffinement des modèles de transformation de classificateur

Règle 7.15

```
Généraliser ImportPackage
  pour chaque modèle de transformation de classificateur MT faire
    pour chaque group Gi d'énoncé import de MT paramétrés par une même chaîne S et pour le
    même context ctx et ctx n'est pas un classificateur faire
      fusionner les membres de Gi dans un import groupé
      Pi:= un nouveau PSC du type de CTXi
      Pi.name := MT.primaryImplementedPSC.name + « mergeImportStatements » + i
      enregistrer dans Gi mergeImportStatements(ctx, S, Pi)
    fin pour
  fin pour
Fin généraliser ImportPackage
```

```
Généraliser RaisedException
   pour chaque modèle de transformation de classificateur MT faire
      pour chaque group Gi d'énoncé raisedException de MT paramétrés par une même chaîne S et
      pour le même context ctx et ctx n'est pas un classificateur faire
        fusionner les membres de Gi dans un RaisedException groupé
        Pi:= un nouveau PSC du type de CTXi
        Pi.name := MT.primaryImplementedPSC.name + « raisedExceptionStatements » + i
        enregistrer dans Gi raisedExceptionStatements(ctx, S, Pi)
      fin pour
   fin pour
Fin généraliser RaisedException
```

Sous-étape 2.3.5 : paramétrage des appels

Règle 7.16

```
Généraliser paramètres
   pour chaque modèle de transformation d'opération MT faire
      pour chaque groupe Gi d'appels consécutifs au même modèle de transformation de paramètre
      MTPi faire
        fusionner les membres de Gi dans un seul appel
        enregistrer dans Gi mergeParameters(MTPi.context, call_template(MTPi))
      fin pour
   fin pour
Fin généraliser paramètres
```

Règle 7.17

```
Généraliser attributs
   pour chaque modèle de transformation de classificateur MT faire
      pour chaque group Gi d'appel au même modèle de transformation d'attribut MTAi faire
        fusionner les membres de Gi dans un seul appel
        enregistrer dans Gi mergeAttributes(MTAi.context, call_template(MTAi))
      fin pour
   fin pour
Fin généraliser attributs


Généraliser opérations
   pour chaque modèle de transformation de classificateur MT faire
      pour chaque group Gi d'appel au même modèle de transformation d'opération MTAi faire
        fusionner les membres de Gi dans un seul appel
        enregistrer dans Gi mergeOpérations(MTAi.context, call_template(MTAi))
      fin pour
   fin pour
Fin généraliser opérations
```

APPENDICE C

LES TECHNIQUES D'ANALYSE UTILISÉES

Dans plusieurs aspects de notre approche, nous supposons que le code source lié à l'implémentation d'un même PSC possède des caractéristiques particulières la distinguant des autres concepts : elle est répétitive ou semi-répétitive. Nous pouvons alors supposer que les classificateurs implémentant un même PSC partagent un vocabulaire commun et/ou du code cloné. Par conséquent, nous avons choisi des techniques tels que l'indexation sémantique latente ou LSI (« Latent semantic indexing »), la détection de clones et l'analyse de flux de données afin d'identifier cette caractéristique. Dans cet appendice, nous donnons des détails sur la façon dont nous les avons mises en œuvre.

C.1 LSI

Notre implémentation LSI suit les principes de cette technique tels que décrits à la section 3.3. Pour effectuer les calculs sur les matrices, nous utilisons la bibliothèque ParallelColt[10] qui offre un ensemble d'opérations pour l'algèbre linéaire et fonctionne en parallèle augmentant la vitesse de calcul sur de grandes matrices si l'ordinateur sur lequel il s'exécute possède plusieurs processeurs.

La valeur de notre `k` est déterminée empiriquement pour chaque système analysé et le seuil de similitude a été établi à 0,995 pour tous les systèmes.

Les groupes selon LSI sont découverts comme suit. Nous construisons d'abord une matrice représentant le système analysé. Les colonnes sont des classificateurs du système et les lignes sont les verbes extraits du nom des opérations des classificateurs (les opérations héritées sont également prises en considération pour un classificateur). La raison de l'utilisation des verbes vient du fait que le verbe dans le nom d'une opération doit refléter la tâche effectuée par l'opération. Les valeurs de la matrice sont alors la fréquence des verbes dans les classificateurs après avoir été normalisées en

10 sites.google.com/site/piotrwendykier/software/parallelcolt, 09-27-2010

utilisant tf-idf[11]. La similitude entre les classificateurs est mesurée par le cosinus entre les vecteurs correspondants à chaque classificateur. Chaque ensemble de classificateurs partageant une similarité franchissant le seuil est placé dans un groupe commun.

C.2 DÉTECTION DE CLONES

Les groupes d'opérations basés sur les clones sont extraits en utilisant l'outil CCFinder[12]. Dans la phase de la découverte du profil de la plate-forme d'implémentation, nous cherchons les clones entre les opérations de différents classificateurs et nous cherchons donc à créer des groupes d'opérations partageant du code cloné. Dans la phase de la découverte des modèles de transformation, nous cherchons les clones à l'intérieur d'une même opération et donc à créer des groupes de jetons clonés à l'intérieur d'une même opération. Dans les deux cas, nous utilisons l'outil CCFinder avec les paramètres appropriés aux types de clones désirés afin qu'il génère des groupes de clones (ou classes de clones) en joignant les paires de clones détectés.

C.3 ANALYSE DE FLUX DE DONNÉES

Nous procédons à deux analyses de flux de données qui suivent les principes de LSI que décrite à la section 3.5. La première analyse cherche à déterminer pour chacun des paramètres de chacune des opérations du PSM si sa valeur peut être affectée à un attribut d'un classificateur. La seconde analyse cherche à déterminer si la valeur d'un attribut peut être affectée à une variable locale, c'est-à-dire à l'intérieur d'une opération. Pour effectuer ces analyses de flux de données, nous utilisons l'outil Soot[13].

---

[11] TF-IDF ajuste l'importance d'un terme par rapport à sa fréquence relative dans le document, pondéré par le nombre de documents qui le contiennent.

[12] www.ccfinder.net, 09-27-2010

[13] www.sable.mcgill.ca/soot/,

BIBLIOGRAPHIE

ADM Domain Task Force (2003). Why Do We Need Standards for the Modernization of Existing Systems?

Alam, A., et T. Padenga. 2010. *Application Software Re-Engineering*: Pearson Education India, 254 p.

Anquetil, N., et T. C. Lethbridge. 1999. «Recovering software architecture from the names of source files». *Journal of Software Maintenance: Research And Practice*, vol. 11, no 3, p. 201-221.

Baxter, I. D., A. Yahin, L. Moura, M. Sant'Anna et L. Bier. 1998. «Clone Detection Using Abstract Syntax Trees». In *Proceedings of the International Conference on Software Maintenance* (Bethesda, MD, USA, novembre), p. 368-377. Bethesda, MD, USA: IEEE Computer Society Press.

Bézivin, J., M. Blay, M. Bouzhegoub, J. Estublier, J.-m. Favre, S. Gérard et J. M. Jézéquel (2004). Rapport de synthèse de l'AS CNRS sur le MDA, CNRS

Biggerstaff, T. J., B. G. Mitbander et D. E. Webster. 1993. «The concept assignment problem in program understanding». In *Proceedings of the international conference on Software Engineering* (Baltimore, USA, 17-21 mai), p. 482-498. Baltimore, USA: IEEE Computer Society.

Bingham, E., et H. Mannila. 2001. «Random projection in dimensionality reduction: applications to image and text data». In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining* (San Francisco, CA, USA, 26-29 août), p. 245-250. San Francisco, CA, USA: ACM.

Bisbal, J., D. Lawless, W. Bing et J. Grimson (1999). Legacy System Migration: A Brief Review of Problems, Solutions and Research Issues. Dublin, Irlande, Computer Science Department, Trinity College

Bisbal, J., D. Lawless, W. Bing, J. Grimson, V. Wade, R. Richardson et D. O'Sullivan. 1997. «An overview of legacy information system migration». In *Proceedings of the Software Engineering Conference, 1997. Asia Pacific and International Computer Science Conference 1997* (Clear Water Bay, Hong Kong, 2-5 décembre), p. 529-530. Clear Water Bay, Hong Kong.

Bodhuin, T., E. Guardabascio et M. Tortorella. 2003. «Migration of non-decomposable software systems to the Web using screen proxies». In *Proceedings of the 10th Working Conference on Reverse Engineering* (Victoria, Canada, 13-16 novembre), p. 165. Victoria, Canada: IEEE Computer Society.

Boer, R. C. d., et H. v. Vliet. 2008. «Architectural knowledge discovery with latent semantic analysis: Constructing a reading guide for software product audits». *Journal of Systems and Software*, vol. 81, no 9, p. 1456-1469.

Boronat, A., J. A. Carsi et I. Ramos. 2005. «Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine». In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering* (Manchester, UK, 21- 23 mars), p. 228-231. Manchester, UK: IEEE Computer Society.

Breu, S., et J. Krinke. 2003. «Aspect mining using dynamic analysis». *GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft fur Informatik*, vol. 23, p. 21-22.

Bruneliere, H., J. Cabot, J. Frédéric et M. Frédéric. 2010. «MoDisco: a generic and extensible framework for model driven reverse engineering». In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (Anvers, Belgique, 20-24 septembre), p. 173-174. Anvers, Belgique: ACM.

Bruntink, M., A. v. Deursen, T. Tourwe et R. vanEngelen. 2004. «An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns». In *Proceedings of the 20th IEEE International Conference on Software Maintenance* (Chicago, IL, USA, 11-17 septembre), p. 200-209. Chicago, IL, USA: IEEE Computer Society.

Canfora, G., L. Cerulo et M. DiPenta. 2006. «On the Use of Line Co-change for Identifying Crosscutting Concern Code». In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (Dublin, Irlande, 24-27 septembre), p. 213-222. Dublin, Irlande: IEEE Computer Society.

Caprile, B., et P. Tonella. 1999. «Nomen Est Omen: Analyzing the Language of Function Identifiers». In *Proceedings of the Sixth Working Conference on Reverse Engineering* (Atlanta, GA, USA, 8 octobre), p. 112-122. Atlanta, GA, USA: IEEE Computer Society Press.

Chen, F., H. Yang, B. Qiao et W. C.-C. Chu. 2006. «A Formal Model Driven Approach to Dependable Software Evolution». In *Proceedings of the International Computer Software and Applications Conference* (Chicago, IL, USA, 17-21 septembre), p. 205-214. Chicago, IL, USA: IEEE Computer Society.

Chénard, G. 2007. «Vers une approche automatique pour l'extraction des règles d'affaires d'une application». Université du Québec à Montréal. En ligne. <http://www.archipel.uqam.ca/764/>. Consulté le 21 décembre 2012.

Chénard, G., I. Khriss et A. Salah. 2007. «Towards a New Approach for Extracting Business Logic of an Object-Oriented Application». In *Proceedings of 20th International Conference on Software & Systems Engineering and their Applications* (Paris, France, 4-6 décembre), p. 1-13. Paris, France.

--------. 2010. «Towards the Discovery of Implementation Platform Description Models of Legacy Object-Oriented Systems». In *Workshop on Processes for Software Evolution and Maintenance (WoPSEM 2010)* (Boston, MA, USA, 13-16 octobre), p. 1-10. Boston, MA, USA: IEEE. En ligne. <http://ikhriss.uqar.ca/papers/CKS_WOPSEM2010.pdf>. Consulté le 21 décembre 2012.

--------. 2012. «Towards the Automatic Discovery of Platform Transformation Templates of Legacy Object-Oriented ». In *Models and Evolution (ME) 2012 workshop a satellite event at MoDELS 2012* (Insbrusck, Autriche, 30 septembre), p. 51-56. Insbrusck, Autriche: ACM.

Chia-Chu, C., et C. Bayrak. 2006. «Legacy Software Modernization». In *IEEE International Conference on Systems, Man and Cybernetics* (Taipei, Taiwan, 8-11 octobre), p. 1304-1309. Taipei, Taiwan: IEEE.

Chikofsky, E. J. 1990. «Reverse engineering and design recovery: A taxonomy». *IEEE Software*, vol. 7, no 1, p. 13-17.

Cimitile, A., A. De Lucia, G. A. Lucca et A. R. Fasolino. 1999. «Identifying objects in legacy systems using design metrics». *Journal of Systems and Software*, vol. 44, no 3, p. 199-211.

Comella-Dorda, S., K. Wallnau, R. Seacord et J. Robert. 2000a. «A Survey of Black-Box Modernization Approaches for Information Systems». In *Proceedings of the International Conference on Software Maintenance (ICSM'00)* (San Jose, CA, USA, 11-14 octobre), p. 173-183. San Jose, CA, USA: IEEE Computer Society.

-------- (2000b). A Survey of Legacy System Modernization Approaches. Software Engineering Institute, Carnegie Mellon University

Cornelissen, B., et L. Moonen. 2007. «Visualizing Similarities in Execution Traces». In *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA).* (Vancouver, Canada, 29 octobre), p. 6-10. Vancouver, Canada.

Corrêa, P. L. P., P. J. Moacir et M. J. José. 2005. «Architecture to Application Gateway to access Electronic Government Legacy System ». In *Proceedings of the 1st International Conference on Interoperability of eGovernment Services* (Genève, Suisse, 23 -24 février), p. 6-10. Genève, Suisse.

Davide, B., D. Antonio Castaldo, apos, Ursi, C. Luca, M. Mattia, C. Antonio et C. D'ursi Luca (2005). Slicing AspectJ Woven Code En ligne. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.6405>.

De Lucia, A., R. Francese, G. Scanniello, G. Tortora et N. Vitiello. 2006. «A Strategy and an Eclipse Based Environment for the Migration of Legacy Systems to Multi-tier Web-based Architectures». In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (Philadelphia, PA, USA,

24-27 septembre), p. 438-447. Philadelphia, PA, USA: IEEE Computer Society.

Deerwester, S., T. D. Susan, W. F. George, T. K. Landauer, H. Richard et V. Karakostas. 1990. «Indexing by latent semantic analysis». *Journal of the American Society for Information Science*, vol. 41, no 6, p. 391-407.

Doyle, D., H. Geers, B. Graaf, A. v. Deursen et J. M. Favre. 2006. «Migrating a domain-specific modeling language to MDA technology». In *Proceedings of the International Workshop on Metamodels* (Genoa, Italie, 1 octobre 2006), p. 47-54. Genoa, Italie.

Ducasse, S., M. Rieger et S. Demeyer. 1999. «A Language Independent Approach for Detecting Duplicated Code». In *Proceedings of the IEEE International Conference on Software Maintenance* (Oxford, UK, 30 août-3 septembre), p. 109-118. Oxford, UK: IEEE Computer Society Press.

EJDTechnologies. 2006. «SourceCafe». EJD Technologies En ligne. <http://www.decompile.com/curtis/articles/JavaPro/index.htm>. Consulté le 26 octobre 2013.

Favre, L. 2008a. «Formalizing MDA-Based Reverse Engineering Processes». In *Proceeding of Sixth International Conference on Software Engineering Research, Management and Applications* (20-22 août), p. 153-160: IEEE.

Favre, L. 2008b. «Modernizing Software & System Engineering Processes». In *Proceedings of the International Conference on Systems Engineering* (Las Vegas, NV, USA, 19-21 août), p. 442-447. Las Vegas, NV, USA: IEEE Computer Society.

Fetzer, C., et M. Süßkraut. 2008. «Switchblade: enforcing dynamic personalized system call models». *SIGOPS Oper. Syst. Rev.*, vol. 42, no 4, p. 273-286.

Fleurey, F., E. Breton, B. Baudry, A. Nicolas et J.-M. Jézéquel. 2007. «Model-Driven Engineering for Software Migration in a Large Industrial Context». In *Proceedings of the international conference on Model Driven Engineering Languages and Systems* (Nashville, TN, USA, 30 septembre - 5 octobre), p. 482-497. Nashville, TN, USA: Springer.

Fry, Z. P., D. Shepherd, E. Hill, L. Pollock et K. Vijay-Shanker. 2008. «Analysing source code: looking for useful verb-direct object pairs in all the right places». *IET Software Special Issue on Natural Language in Software Development*, vol. 2, no 1, p. 27-36.

Graaf, B., S. Weber et A. v. Deursen. 2008. «Model-driven migration of supervisory machine control architectures». *Journal of Systems and Software*, vol. 81, no 4, p. 517-535.

Greevy, O., et S. Ducasse. 2005. «Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces». In *Proceedings of International Workshop on Object-Oriented Reengineering* (Glasgow, U.K, juillet 2005), p. 2-7. Glasgow, U.K.

Gueheneuc, Y. 2004. «A Systematic Study of UML Class Diagram Constituents for their Abstract and Precise Recovery». In *Proceeding of the 11th Asia-Pacific software engineering conference* (Busan, Corée, 30 novembre-3 décembre), p. 265-274. Busan, Corée: IEEE Computer Society Press.

Hamou-Lhadj, A., E. Braun, D. Amyot et T. C. Lethbridge. 2005. «Recovering Behavioral Design Models from Execution Traces». In *Proceedings of the European Conference on Software Maintenance and Reengineering* (Manchester, UK, 21-23 mars), p. 112-121. Manchester, UK: IEEE Computer Society Press.

Heuzeroth, D., T. Holl, G. Högström et W. Löwe. 2003. «Automatic Design Pattern Detection». In *Proceedings of the 11th IEEE International Workshop on Program Comprehension* (Portland, OR, USA, mai), p. 94-103. Portland, OR, USA: IEEE Computer Society Press.

Huang, H., S. Zhang, J. Cao et Y. Duan. 2005. «A practical pattern recovery approach based on both structural and behavioral analysis». *Journal of Systems and Software*, vol. 75, no 1/2, p. 69-87.

ISO/IEC, S. E. (1999). Software Maintenance FDIS 14764. Genève, Suisse, International Standards Organization: 38 p

Jacobson, I., et F. Lindstr. 1991. «Reengineering of old systems to an object-oriented architecture». *ACM Sigplan Notices*, vol. 26, no 11, p. 340-350.

Jiang, Z. M., et A. E. Hassan. 2007. «A Framework for Studying Clones In Large Software Systems». In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation* (Paris, France, 30 septembre-1 octobre), p. 203-212. Paris, France: IEEE Computer Society.

Jouault, F., J. Bézivin et M. Barbero. 2009. «Towards an advanced model-driven engineering toolbox». *Innovations in Systems and Software Engineering*, vol. 5, no 1, p. 5-12. En ligne. <http://dx.doi.org/10.1007/s11334-009-0082-7>.

Kajko-Mattsson, M., M. Ta et L. Wilczek. 2007. «State of Modernization Practice in Four Swedish Organizations». In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007) - Volume 01* (Beijing, Chine, 23-27 juillet), p. 549-556. Beijing, Chine: IEEE Computer Society.

Kamiya, T., S. Kusumoto et K. Inou. 2001. «A Token-based Code Clone Detection Technique and Its Evaluation». *IEICE SS2000-42*, vol. 100, no 570, p. 41-49.

Kamiya, T., S. Kusumoto et K. Inoue. 2002. «CCFinder: a multilinguistic token-based code clone detection system for large scale source code». *IEEE Transactions on Software Engineering*, vol. 28, no 7, p. 654-670.

Keschenau, M. 2004. «Reverse engineering of UML specifications from java programs». In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (Vancouver, Canada, 24-28 octobre), p. 326 - 327 Vancouver, Canada: ACM Press.

Khedker, U., A. Sanyal et B. Karkare. 2009. *Data Flow Analysis: Theory and Practice*: CRC Press, Inc., 395 p.

Khriss, I., et G. Chénard. 2008. «Vers une nouvelle approche d'extraction de la logique métier d'une application orientée objet». In *Proceedings of the 14ème Colloque International sur les Langages et Modèles à Objets* (Montréal, Québec, 3-7 mars), p. 63-76. Montréal, Québec.

Khriss, I., G. Chénard et A. Salah. 2008. «Achieving Object-Oriented Systems Modernization through Model-Driven Architecture Migration». In *Proceedings of 21th International Conference on Software & Systems Engineering and their Applications* (Paris, France, 9-11 décembre), p. 1-10. Paris, France.

Kim, H. S., et C. H. Kim. 2004. «A use-case driven approach to component mining for legacy modernization». In *Proceedings of the IASTED International Conference on Software Engineering* (Innsbruck, Autriche, 17-19 février), p. 303-308. Innsbruck, Autriche: IASTED/ACTA Press.

Kim Jung, J., et M. Benner Kevin. 1996. «Implementation patterns for the observer pattern». In *Pattern languages of program design,* p. 75-86: Addison-Wesley Longman Publishing Co. Inc.

Kim, S., K. Pan et J. E. J. Whitehead. 2006. «Micro pattern evolution». In *Proceedings of the 2006 international workshop on Mining software repositories* (Shanghai, China, 22-23 mai), p. 40-46 Shanghai, China: ACM Press.

Kleppe, A. G., J. Warmer et W. Bast. 2003. *MDA Explained: The Model Driven Architecture: Practice and Promise*: Addison-Wesley Longman Publishing Co., Inc., 170 p.

Kontogiannis, K., R. Di Mori, M. Bernstein et E. Merlo. 1994. «Localization of Design Concepts in Legacy Systems». In *Proceedings of the International Conference onSoftware Maintenance* (Victoria, Canada, septembre), p. 414-423. Victoria, Canada: IEEE Computer Society Press.

Koskinen, J., J. Ahonen J., H. Sivula, T. Tilus, H. Lintinen et I. Kankaanpaa. 2005. «Software Modernization Decision Criteria: An Empirical Study». In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering* (Manchester, UK, 21-23 mars), p. 324-331. Manchester, UK: IEEE Computer Society.

Kramer, C., et L. Prechelt. 1996. «Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software». In *Proceedings of the Third Working Conference on Reverse Engineering* (Monterey, CA, USA, novembre), p. 208-215. Monterey, CA, USA: IEEE Computer Society Press.

Kuhn, A., S. Ducasse et T. Girba. 2007. «Semantic clustering: Identifying topics in source code». *Information and Software Technology*, vol. 49, no 3, p. 230-243.

Law, K. C. K., H. H. S. Ip et F. Wei. 1998. «Web-enabling legacy applications». In *Proceedings of 1998 International Conference on Parallel and Distributed System* (Tainan, Taiwan, 14-16 décembre), p. 218-225. Tainan, Taiwan.

Li, M., O. F. Rana, M. S. Shields et D. W. Walker. 2000. «A wrapper generator for wrapping high performance legacy codes as Java/CORBA components». In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (Dallas, TX, USA, 5-10 novembre), p. 13. Dallas, TX, USA: IEEE Computer Society.

Mansurov, N., et D. Campara. 2005. «Managed Architecture of Existing Code as a Practical Transition Towards MDA». In *UML Modeling Languages and Applications,* p. 219-233.

Marcus, A., R. Koschke, A. v. Deursen, A. Rajlich, P. Tonella et H. Sneed. 2005. «Identification of concepts, features, and concerns in source code». In *Panel Discussion at the International Conference on Software Maintenance* (Budapest, Hongrie, 25-30 septembre), p. 1. Budapest, Hongrie.

Marcus, A., A. Sergeyev, V. Rajlich et J. I. Maletic. 2004. «An Information Retrieval Approach to Concept Location in Source Code». In *Proceedings of the Working Conference on Reverse Engineering* (Delft, Pays-Bas, 8-12 novembre), p. 214-223. Delft, Pays-Bas: IEEE Computer Society.

Marin, M., L. Moonen et A. v. Deursen. 2006. «A common framework for aspect mining based on crosscutting concern sorts». In *Proceedings of the 13th Working Conference on Reverse Engineering* (Benevento, Italie), p. 29-38. Benevento, Italie: IEEE Computer Society.

Mazón, J. N., et J. Trujillo. 2008. «A Model Driven Modernization Approach for Automatically Deriving Multidimensional Models in Data Warehouses». In *Conceptual Modeling - ER 2007,* p. 56-71.

Miller, J., et J. Mukerji (2003). MDA Guide Version 1.0.1, Object Management Group: 62 p

OMG (2003). UML 2.0 OCL - 2nd Revised Submission En ligne. <http://www.omg.org/cgi-bin/doc?ad/2003-01-07>. Consulté le 1 novembre 2008.

-------- (2006). Architecture-Driven Modernization Task Force. Architecture-Driven Modernization scenarios En ligne. <http://adm.omg.org/>. Consulté le 1 novembre 2008.

-------- (2008). MOF 2.0 Query/View/transformation Specification. Version 1.0 En ligne. <www.omg.org >. Consulté le 1 novembre 2008.

Oracle. 2006a. «OTN Financial Brokerage Service 10g». Oracle. En ligne. <http://www.oracle.com/technology/sample_code/tech/java/j2ee/fbs10g/index.html>. Consulté le June 15, 2010.

--------. 2006b. «Virtual Shopping Mall 1.3 ». Oracle. En ligne. <http://www.oracle.com/technology/sample_code/tutorials/vsm1.3/over/design.htm>. Consulté le June 15, 2010.

Philippow, I., D. Streitferdt, M. Riebisch et S. Naumann. 2005. «An approach for reverse engineering of design patterns». *Software and Systems Modeling*, vol. 4, no 1, p. 55-70.

Pinali, D. O. 1999. «Design Pattern Extraction for Software Documentation». Vrije, Belgium, Computer Science Department, Universiteit Brussel.

Pollock, L., K. Vijay-Shanker, D. Shepherd, E. Hill, Z. P. Fry et K. Maloor. 2007. «Introducing natural language program analysis». In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (San Diego, CA, USA, 13-14 juin), p. 15-16. San Diego, CA, USA: ACM.

Qiao, B., H. Yang, W. Chu et B. Xu. 2003. «Bridging Legacy Systems to Model Driven Architecture». In *Proceedings of the International Conference on Computer Software and Applications* (Washington, DC, USA, 3-6 novembre), p. 304-309. Washington, DC, USA: IEEE Computer Society.

Reese, G. 2000. *Database Programming with JDBC and Java, Second Edition*: O'Reilly & Associates, Inc., 328 p.

Reus, T., H. Geers et A. v. Deursen. 2006. «Harvesting Software Systems for MDA-Based Reengineering». In *Proceeding European Conference on Model Driven Architectures: Foundations and Applications* (Bilbao, Espagne, 10-13 juillet), p. 213-225. Bilbao, Espagne: Springer-Verlag.

Roy, C. K., J. R. Cordy et R. Koschke. 2009. «Comparison and evaluation of code clone detection techniques and tools: A qualitative approach». *Science of Computer Programming*, vol. 7, no 74, p. 470-495.

Sadovykh, A., L. Vigier, A. Hoffmann, J. Grossmann, T. Ritter, E. Gomez et O. Estekhin. 2009. «Architecture Driven Modernization in Practice - Study Results». In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems* (Potsdam, Allemagne, 2-4 juin), p. 50-57. Potsdam, Allemagne: IEEE Computer Society.

Schmidt, D. C. 2006. «Guest Editor's Introduction: Model-Driven Engineering». *Computer*, vol. 39, no 2, p. 25.

Seacord, R. C., D. Plakosh et G. A. Lewis. 2003. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*: Addison Wesley Professional, 352 p.

Shepherd, D., Z. P. Fry, E. Hill, L. Pollock et K. Vijay-Shanker. 2007. «Using natural language program analysis to locate and understand action-oriented concerns». In *Proceedings of the 6th international conference on Aspect-oriented software development* (Vancouver, Canada, 12 - 16 mars), p. 212-224. Vancouver, Canada: ACM.

Singh, I., B. Stearns et M. Johnson. 2002. *Designing enterprise applications with the J2EE platform*: Addison-Wesley Longman Publishing Co., Inc., 417 p.

Sutton, A., et J. I. Maletic. 2007. «Recovering UML class models from C++: A detailed explanation». *Information and Software Technology*, vol. 49, no 3, p. 212-229.

Tsai, B. Y., S. Stobart et N. Parrington. 2001. «Employing data flow testing on object-oriented classes». *Software, IEE Proceedings* -, vol. 148, no 2, p. 56-64. En ligne. <10.1049/ip-sen:20010448>.

Van Der Spek, P., S. Klusener et P. Van de Laar. 2008. «Towards Recovering Architectural Concepts Using Latent Semantic Indexing». In *Proceedings of the European Conference on Software Maintenance and Reengineering* (Athens, Grèce, 1-4 avril), p. 253-257. Athens, Grèce: IEEE Computer Society.

Weiderman, N. H., J. K. Bergey, D. B. Smith et S. R. Tilley (1997). Approaches to Legacy System Evolution. Pittsburgh, Pa, Software Engineering Institute, Carnegie Mellon University

Wilde, N., et T. Gust. 1992. «Locating user functionality in old code». In *Proceedings of IEEE International Conference on Software Maintenance* (Orlando, FL, USA, 25-29 novembre), p. 200-205. Orlando, FL, USA: IEEE Computer Society.

Zhang, W., A. Berre, D. Roman et H. Huru. 2009. «Migrating Legacy Applications to the Service Cloud». In *Proceedings of Object Oriented Programming Systems Languages and Applications* (Orlando, FL, United States, 25-29 octobre), p. 56-68. Orlando, FL, United States.