Université de Montréal

Étude de la traçabilité entre refactorisations du modèle de classes et refactorisations du code

par
Saliha Bouden

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

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

Décembre, 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Étude de la traçabilité entre refactorisations du modèle de classes et refactorisations du code

présenté par :

Saliha Bouden

a été évalué par un jury composé des personnes suivantes :

Claude Frasson
président-rapporteur

Yann-Gaël Guéhéneuc
directeur de recherche

Stefan Monnier
membre du jury

Mémoire accepté le

RÉSUMÉ

La refactorisation est une technique qui consiste à changer la structure interne d'un programme pour faciliter sa compréhension et sa maintenance sans en changer le comportement externe [Fowler, 1999]. Certains outils de refactorisation ne permettent pas de reporter les changements effectués sur le code vers les modèles (comme UML) et inversement des modèles vers le code. D'autres outils permettent de générer les diagrammes UML à partir du code refactorisé et inversement de générer le code à partir des modèles refactorisés. Cependant, ils n'assurent pas la traçabilité entre la conception d'un programme et son code source, lors de refactorisation.

Le but de notre travail est d'assurer la traçabilité entre le modèle et le code source d'un programme. Nous proposons un catalogue de 55 refactorisations primitives et composites au niveau du modèle de classes et nous définissons pour chaque refactorisation des pré-conditions, des post-conditions et des actions, exprimées dans un pseudo code et nécessaires pour assurer la préservation de "comportement du modèle". Nous distinguons ces refactorisations en refactorisations primitives et composites et nous les classifions afin de mettre en valeur la similarité des opérations associées aux refactorisations. Par ailleurs, nous établissons une correspondance entre les refactorisations du modèle et les refactorisations du code. Enfin, nous avons implanté un sous-ensemble des refactorisations du modèle afin de valider notre étude de la traçabilité entre refactorisations des modèles de classes et refactorisations du code.

Mots clé : préservation du comportement, refactorisation de modèle, refactorisation du code, traçabilité.

ABSTRACT

Refactoring corresponds to a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour [Fowler, 1999]. Some refactoring tools do not allow to report changes made on the source code to (UML) models and inversely on models to source code. Others tools allow to generate models from refactored code and inversely code from refactored models. However, they do not ensure the traceability between the design of a program and its source code.

The goal of our work is to ensure the preservation of the traceability between the model and the source code of a program. We propose a catalog of 55 primitive and composite refactorings at the model level and we define for each refactoring preconditions, postconditions, and actions, expressed in pseudo code and necessary to ensure the preservation of the model "behavior". We distinguish these refactorings as primitive and composite refactorings and we classify them to highlight the similarity of operations associated to refactorings. Moreover, we establish correspondence between model refactorings and code refactorings. Finally, we implement a subset of model refactorings to validate our study of the traceability between model refactorings and code refactorings.

Keywords : behaviour preserving, refactoring model, refactoring code, traceability.

TABLE DES MATIÈRES

RÉSUMÉ . . . . . **iii**

ABSTRACT . . . . . **iv**

TABLE DES MATIÈRES . . . . . **v**

LISTE DES TABLEAUX . . . . . **x**

LISTE DES FIGURES . . . . . **xi**

LISTE DES APPENDICES . . . . . **xii**

DÉDICACE . . . . . **xiii**

REMERCIEMENTS . . . . . **xiv**

CHAPITRE 1 : INTRODUCTION **1**

1.1 Objectifs 3

1.2 Contributions 6

1.3 Organisation du mémoire 6

CHAPITRE 2 : ÉTAT DE L'ART **8**

2.1 Définitions des restucturations et des refactorisations 8

2.2 Applications des restructurations et des refactorisations 10

2.3 Activités de la refactorisation 11

2.4 Préservation de comportement 12

2.4.1 Définition 12

2.4.2 Limites de la définition 12

2.4.3 Techniques et formalismes des refactorisations préservant le comportement 13

2.5 Outils de refactorisation 15

2.6 Refactorisations de modèle et traçabilité entre modèle et code 17

CHAPITRE 3 : REFACTORISATION DES MODÈLES 19

3.1 De l'approche objet à l'ingénierie des modèles 19

3.1.1 Langage de modélisation UML 20

3.1.2 Métamodélisation 21

3.2 Refactorisation des modèles 23

3.2.1 Refactorisations primitives 25

3.2.2 Refactorisations composites 25

CHAPITRE 4 : IMPLANTATION DES REFACTORISATIONS 46

4.1 Refactorisations du code implantées dans Eclipse 46

4.2 Problèmes rencontrés avec les modèles 48

4.3 Refactorisations au niveau des modèles 49

4.3.1 Le métamodèle PADL 49

4.3.2 Sélection des refactorisations 50

4.3.3 Processus de refactorisation 62

4.3.4 Tests unitaires 63

CHAPITRE 5 : TRAÇABILITÉ ENTRE REFACTORISATION DU MODÈLE ET DU CODE 69

5.1 Correspondance entre refactorisations du modèle et du code 69

5.1.1 Refactorisations du code 69

5.2 Notre classification 71

5.3 Refactorisations primitives appliquées sur un paquetage 74

5.3.1 Créer un paquetage 74

5.3.2 Renommer un paquetage 74

5.3.3 Supprimer un paquetage 75

5.3.4 Copier un paquetage 76

5.4 Refactorisations primitives appliquées sur une interface 76

5.4.1 Créer une interface 76

5.4.2 Renommer une interface 77
5.4.3 Supprimer une interface 78
5.4.4 Copier une interface 78
5.5 Refactorisations primitives appliquées sur une classe 79
5.5.1 Créer une classe 79
5.5.2 Renommer une classe 80
5.5.3 Supprimer une classe 80
5.5.4 Copier une classe 81
5.6 Refactorisations primitives appliquées sur un attribut 82
5.6.1 Créer un attribut 82
5.6.2 Renommer un attribut 82
5.6.3 Changer la visibilité d'un attribut 83
5.6.4 Changer le type d'un attribut 84
5.6.5 Supprimer un attribut 85
5.6.6 Copier un attribut 85
5.7 Refactorisations primitives appliquées sur une méthode 86
5.7.1 Créer une méthode 86
5.7.2 Renommer une méthode 87
5.7.3 Changer le type de retour d'une méthode 87
5.7.4 Changer la visibilité d'une méthode 88
5.7.5 Supprimer une méthode 89
5.7.6 Copier une méthode 89
5.8 Refactorisations primitives appliquées sur un paramètre 90
5.8.1 Créer un paramètre 90
5.8.2 Renommer un paramètre 91
5.8.3 Changer le type d'un paramètre 92
5.8.4 Supprimer un paramètre 92
5.8.5 Copier un paramètre 93
5.9 Refactorisations primitives appliquées sur une association 94
5.9.1 Créer une association 94

5.9.2 Renommer une association 95
5.9.3 Changer la multiplicité de la classe cible 95
5.9.4 Changer la multiplicité de la classe source 96
5.9.5 Supprimer une association 97
5.9.6 Copier une association 98
5.10 Refactorisations composites appliquées sur un paquetage 99
5.10.1 Ajouter un paquetage 99
5.10.2 Déplacer un paquetage 100
5.11 Refactorisations composites appliquées sur une interface 101
5.11.1 Ajouter une interface 101
5.11.2 Déplacer une interface 102
5.12 Refactorisations composites appliquées sur une classe 103
5.12.1 Ajouter une classe 103
5.12.2 Déplacer une classe 105
5.13 Refactorisations composites appliquées sur un attribut 106
5.13.1 Ajouter un attribut 106
5.13.2 Déplacer un attribut 107
5.13.3 Monter un attribut 108
5.13.4 Descendre un attribut 109
5.13.5 Encapsuler un attribut 110
5.13.6 Remplacer un attribut par des sous classes 111
5.14 Refactorisations composites appliquées sur une méthode 113
5.14.1 Ajouter une méthode 113
5.14.2 Déplacer une méthode 114
5.14.3 Monter une méthode 115
5.14.4 Descendre une méthode 116
5.15 Refactorisations composites appliquées sur un paramètre 117
5.15.1 Ajouter un paramètre 117
5.15.2 Déplacer un paramètre 119
5.16 Refactorisations composites appliquées sur une association 120

5.16.1 Ajouter une association 120
5.16.2 Déplacer une association 121

CHAPITRE 6 : VALIDATION 123
6.1 Étude de cas 123
6.1.1 JHotDraw 123
6.1.2 Objectifs 123
6.1.3 Méthode 124
6.1.4 Application de la refactorisation *Rename Method* 124

CHAPITRE 7 : CONCLUSION ET TRAVAUX FUTURS 129

BIBLIOGRAPHIE 132

LISTE DES TABLEAUX

3.1 Refactorisations primitives 27
3.2 Refactorisations composites 34

I.1 Traçabilité entre refactorisations du code et refactorisation du modèle . . 138

LISTE DES FIGURES

3.1 Architecture de métamodélisation de l'OMG, composée de quatre couches [Kiczales *et al.*, 1992]. 22
3.2 Exemple d'utilisation de l'architecture de métamodélisation de l'OMG. . . 23

4.1 Sous-ensemble simplifié du métamodèle PADL 51
4.2 Diagramme de refactorisations implantées par apport au métamodèle PADL. 53
4.3 Les trois étapes composant le processus de la refactorisation. 62
4.4 exemple de l'application de la refactorisation de *Rename Method* sur le modèle de classes de QuickUML. 66
4.5 Exemple de la refactorisation *Pull Up Method* appliquée sur le modèle de classes du code précédent. 68

5.1 Classification des refactorisations du modèle de classes 73

6.1 Exemple de l'application de la refactorisation *Rename Method* sur le modèle de classes de JHotDraw 5.2 127
6.2 Schéma synthétique illustrant l'étude de la traçabilité 128

LISTE DES APPENDICES

Annexe I : **Traçabilité entre refactorisations du code et refactorisations du modèle** **138**

À la mémoire de mes parents et mon frère.

REMERCIEMENTS

J'aimerais tout d'abord remercier le bon Dieu qui m'a donné la force de mener cette maîtrise à son terme malgré les pénibles moments que j'ai vécus après la mort de mes plus chers parents.

Je tiens à remercier vivement mon directeur de recherche, Yann-Gaël Guéhéneuc, qui m'a proposé ce sujet, qu'il trouve ici l'expression de ma profonde reconnaissance pour son soutien et sa disponibilité, pour la confiance et la compréhension qu'il m'a témoigné tout au long de cette maîtrise.

Je remercie Naouel pour son soutien précieux et les bons moments qu'on a passé ensemble.

Je remercie aussi tous les membres du laboratoire GEODES et en particulier, les membres de notre équipe Ptidej.

Merci à mes très chers enfants Abir et Aimen ainsi que mon mari pour la compréhension et la patience.

Je n'oublie pas de remercier ma famille et mes amies, qui n'ont jamais cessé de m'encourager et de m'appuyer.

Je tiens aussi à exprimer ma profonde gratitude à mon professeur Kholladdi Mohamed Kheireddine, directeur du département d'informatique de l'unversité de Constantine, pour ses encouragements.

Enfin, je remercie les membres de mon jury, Claude Frasson et Stefan Monnier qui ont accepté de juger ce travail.

CHAPITRE 1

INTRODUCTION

Une propriété intrinsèque d'un programme est son besoin d'évoluer. Au cours de sa vie, les programmeurs sont amenés à le modifier pour corriger des bogues, ajouter de nouvelles fonctionnalités ou améliorer son implantation. Ainsi, le code source du programme devient de plus en plus complexe et dévie de sa conception initiale, ce qui diminue sa qualité. Pour cette raison, la majeure partie du coût de développement du logiciel est consacrée à la maintenance [Coleman, 1994].

Dans l'industrie, la plupart des études menées montrent que la maintenance représente au moins la moitié des coûts de développement [Richner et Ducasse, 1999], tandis que les trois quarts du temps de la maintenance sont consacrés à la compréhension du code source des programmes [Sharon, 1996]. Cependant, les méthodes et les outils de développement actuels ne facilitent pas la compréhension et la maintenance d'un programme orienté objet, parce qu'ils le rendent plus complexe encore[1] [Glass, 1998]. Des techniques qui permettent de réduire la complexité du logiciel en améliorant sa structure interne sont donc nécessaires.

Le domaine de recherche traitant ce problème de complexité croissante en visant à améliorer la qualité interne du logiciel est désigné sous le nom de restructuration [Griswold, 1992] dans le cas d'un langage de programmation procédural et par refactorisation[2] [Opdyke, 1992; Fowler, 1999] dans le cas des programmes orientés objet : "La refactorisation est une technique qui consiste à changer la structure interne d'un programme pour faciliter sa compréhension et sa maintenance sans en changer son comportement externe"[3] [Fowler, 1999].

---

[1] R.Glass dit que : *"the more modern methods you use in building software, the more time you spend maintaining the resulting product"* [Glass, 1998].

[2] La refactorisation est désignée par le terme 'refactoring' en anglais.

[3] *"Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour"* [Fowler, 1999].

La définition de Fowler est fondée sur le principe de redistribution des classes, des variables et des méthodes à travers la hiérarchie de classes afin de faciliter les modifications et adaptations futures. Ainsi, la refactorisation est utile pour maintenir la qualité d'un programme pendant son évolution. Elle aide à la compréhension du code en le rendant plus lisible, extensible et modulaire.

Plusieurs travaux concernant la refactorisation appliquée sur le code source d'une application orientée objet ont contribué à la mise en oeuvre d'une grande variété d'outils pour aider à la refactorisation[4]. Le degré d'automatisation peut varier selon l'outil et le type de support qu'il fournit. Beaucoup d'entre eux se réfèrent directement et exclusivement à un langage de programmation spécifique, par exemple, C# Refactory[5] pour C# et CodeGuide6.0[6] pour Java.

Les refactorisations appliquées sur le code source d'un programme, présentées par Opdyke dans sa thèse de Ph.D. [Opdyke, 1992] et qui ont été mise en application et perfectionnées par Roberts [Roberts, 1999] plus tard, ainsi que celles présentées dans la littérature, s'appliquent essentiellement aux trois concepts suivants : la classe, la méthode et la variable. L'application de ces transformations peut avoir des conséquences sur le processus de calcul, exprimé pour un objet par la séquence d'appels de méthodes et les changements d'états. Il est donc important de connaître de quelle manière elles modifient les spécifications statiques et dynamique du programme : diagrammes de classes, diagrammes d'états ou diagrammes d'activité ; et d'assurer la traçabilité entre refactorisation du code et des modèles.

Sunyé [Sunyé *et al.*, 2001] recapitule les refactorisations appliquées sur le modèle de classes dans les cinq opérations de base suivantes : l'ajout, la suppression, le déplacement, la généralisation et la spécialisation des éléments de la modélisation.

Malheureusement, les éditeurs développés pour faciliter la refactorisation du code ne permettent pas d'étendre les changements effectués sur le code source d'un programme au niveau des modèles et inversement du modèle au code. Parce qu'ils ne gèrent pas

[4]http ://www.refactoring.com
[5]http ://www.xtreme-simplicity.net/
[6]http ://www.omnicore.com

les modèles et s'ils les gèrent, ils n'assurent pas la traçabilité ; les techniques utilisées consistent à regénérer les modèles à partir du code refactorisé afin de répercuter les modifications au niveau des modèles et inversement des modèles au code. Il est donc important d'effectuer des refactorisations au niveau des modèles et du code afin de préserver la traçabilité entre refactorisation du code et des modèles et ainsi assurer la traçabilité entre le code source d'un programme et sa conception.

1.1 Objectifs

Dans ce travail, nous étudions la traçabilité entre modèle et code source d'un programme en définissant d'une part des refactorisations au niveau du modèle de classes et d'autre part la traçabilité entre ces refactorisations du modèle et des refactorisations du code source. Plus précisément, nous avons d'abord défini une liste regroupant 55 refactorisations pour les modèles de classes des programmes orientés objet, exprimés dans un langage de modélisation proche de UML, en nous inspirant des techniques de refactorisation proposées par Fowler [Fowler, 1999] et Tichelaar [Tichelaar *et al.*, 2000a]. Nous définissons des refactorisations primitives et d'autres composites pour chaque élément constituant le modèle de classes, afin de mettre en valeur la similarité des opérations associées aux refactorisations. Nous englobons sous le terme de refactorisation composite une séquence de refactorisations primitives ou composites. Pour chaque élément constituant le modèle de classes (par exemple, les paquetages, les classes, les méthodes, les attributs et les associations), nous proposons les quatre refactorisations primitives suivantes : la création, le renommage, la suppression et la duplication[7] ; et les deux refactorisations composites suivantes : l'ajout et le déplacement[8]. Pour la méthode et l'attribut, nous avons aussi défini les refactorisations composites qui permettent de faire monter ou descendre des méthodes et des attributs[9], une refactorisation composite qui permet d'encapsuler un

[7] La création, le renommage, la suppression et la duplication correspondent respectivement dans notre travail à : *Create*, *Rename*, *Remove* et *Copy*.

[8] L'ajout et le déplacement correspondent respectivement dans notre travail à : *Add* et *Move*.

[9] Monter et Descendre correspondent respectivement dans notre travail à : *Pull Up* et *Push Down*.

attribut[10] et une autre qui remplace un attribut par des sous-classes[11]. Nous avons jugé que ces refactorisations sont des refactorisations composites, car elles peuvent être composées de plusieurs refactorisations primitives (par exemple, le déplacement d'une classe est composé de la séquence des deux refactorisations primitives suivantes : la duplication de la classe dans le paquetage de destination et la suppression de cette classe de son paquetage de définition) ; ou encore de plusieurs refactorisations primitives et composites (par exemple, l'ajout d'une classe est composé de la séquence des deux refactorisations primitives suivantes : la création et le renommage de la classe, ainsi que des deux refactorisations composites suivantes : l'ajout des attributs et l'ajout des méthodes).

Nous proposons ensuite une classification de ces refactorisations du modèle en nous inspirant de la hiérarchie des différents concepts de l'orienté objet. Cette classification est générique, car l'ajout des refactorisations composites est possible par la composition de refactorisations primitives ou composites ou encore la composition des deux types de refactorisations. Nous distinguons aussi, deux catégories de refactorisations : intra-élément qui s'applique au sein d'un élément (par exemple, un paquetage, une classe, une méthode) et inter-élément qui s'appliquent entre plusieurs éléments dans l'architecture d'un programme.

Nous avons aussi défini pour chaque refactorisation des pré-conditions et des post-conditions ainsi qu'une action, exprimée en pseudo code, nécessaire pour assurer la préservation du comportement. Les pré-conditions sont des conditions que le modèle doit satisfaire pour que la refactorisation puisse être appliquée avec succès et les post-conditions servent à vérifier l'état du modèle après avoir appliqué la refactorisation. L'action, quant à elle, définit concrètement la refactorisation pour transformer le modèle de classes.

Pour implanter ces refactorisations, nous avons utilisé le métamodèle PADL (*Pattern and Abstract-Level Description Language*) permettant de modéliser des programmes orientés objet [Guéhéneuc, 2003]. Les refactorisations implantées sont selectionnées parmi

[10] L'encapsulation d'un attribut correspond dans notre travail à : *Encapsulate Field*.

[11] Remplacer un attribut par des sous-classes correspond dans notre travail à : *Replace Attribute with Subclasses*.

la liste des refactorisations définies, en particulier, celles appliquées sur les méthodes. Pour assurer la préservation de la sémantique du modèle, nous avons ajouté au métamodèle des algorithmes utilitaires qui permettent de donner les classes de la hiérarchie d'une classes donnée. L'implantation de ces refactorisations a été réalisée de façon générique et modulaire afin de faciliter l'ajout d'autres types de refactorisations.

Les refactorisations implantées et appliquées sur la méthode sont les suivantes :

1. Renommer une méthode :
   - renommer une méthode dans sa classe de définition seulement avec la surchage de méthode ;
   - renommer une méthode dans sa classe de définition seulement sans la surchage de méthode ;
   - renommer une méthode et propager le renommage à la hiérarchie de classes avec la surcharge de méthodes ;
   - renommer une méthode et propager le renommage à la hiérarchie de classes sans la surcharge de méthodes ;
   - renommer une méthode et propager le renommage aux interfaces avec la surcharge de méthodes ;
   - renommer une méthode et propager le renommage aux interfaces sans la surcharge de méthodes ;
2. Monter une méthode[12] ;
3. Descendre une méthode[13] ;
4. Déplacer une méthode[14].

La plupart des refactorisations appliquées sur les éléments composant le diagramme de classe peuvent avoir des refactorisations correspondantes appliquées au code source. Nous avons donc étudié avec les refactorisations appliquées au niveau code source qui correspondent aux refactorisations appliquées sur les éléments composant le modèle de

[12]Monter une méthode correspond à *Pull Up Method*.
[13]Descendre une méthode correspond à *Push Down Method*.
[14]Déplacer une méthode correspond à *Move Method*.

classes, afin de préserver la traçabilité entre la conception d'un programme et son code source.

Enfin, pour valider d'une part les refactorisations implantées et d'autre part l'étude de la préservation de la traçabilité entre la conception d'un programme et son code source, nous avons appliqués les refactorisations implantées sur les programmes libres suivants : QuickUML et JHotDraw. Des refactorisations ont été déjà appliquées sur le code source de JHotDraw pour passer d'une version à une autre. Ce qui nous a permis de valider l'étude de la traçabilité de façon objective (indépendante).

1.2 Contributions

Ce travail contribue à la préservation de la traçabilité entre refactorisations du modèle et refactorisations du code. L'approche adoptée pour étudier cette traçabilité est innovatrice. Elle consiste d'une part à définir et classifier 55 refactorisations primitives et composites appliquées sur le modèle ; et d'autre part à étudier avec cohérence les refactorisations appliquées sur le code qui correspondent à ces refactorisations du modèle. L'ajout des refactorisations composites est possible par la composition de refactorisations primitives ou composites, grâce à la classification générique des refactorisations du modèle proposée. Pour assurer la préservation de la sémantique du modèle et faciliter la vérification des refactorisations, nous avons ajouté au métamodèle PADL des algorithmes utilitaires qui permettent de donner les classes de la hiérarchie d'une classes donnée. L'implantation de certaines refactorisations a été réalisée de façon générique et modulaire afin de faciliter l'ajout de d'autres types de refactorisations.

1.3 Organisation du mémoire

Ce mémoire est divisé en 7 sections : l'état de l'art, la refactorisation des modèles, l'implantation des refactorisations du modèle, la traçabilité entre refactorisation du modèle et du code, l'expérimentation et la validation et enfin la conclusion.

Dans le chapitre 2 nous présentons un état de l'art des travaux antérieurs dans le domaine de la restructuration et de la refactorisation. Nous introduisons ensuite la métamodélisation dans le chapitre 3 où nous détaillons notre étude de la refactorisation des modèles. Le chapitre 4 décrit les refactorisations implantées et la validation de ces refactorisations sur le programme libre QuickUML. Dans le chapitre 5, nous étudions la cohérence entre les refactorisations du code correspondantes aux refactorisations primitives et composites du modèle définies dans le chapitre 3, afin d'évaluer l'impact des refactorisations du modèle sur le code source d'un programme orienté objet et préserver ainsi la traçabilité entre la conception d'un programme et son code source. Le chapitre 6 met en pratique et évalue concrétement, les refactorisations implantées dans le chapitre 4 et l'étude de la traçabilité entre refactorisations du modèle et refactorisations de code présentée dans le chapitre 5 sur le programe libre JHotDraw. Enfin, nous concluons ce mémoire et proposons de futures directions de recherche.

CHAPITRE 2

ÉTAT DE L'ART

Dans ce chapitre, nous présentons un état de l'art sur la refactorisation. Nous y présentons les premiers travaux qui ont donnés naissance à ce domaine de recherche et discutons ensuite la préservation de comportement au niveau code et la traçabilité entre modèle et code, lors de refactorisations.

2.1 Définitions des restucturations et des refactorisations

Le mot refactorisation vient du terme factorisation en mathématiques. La factorisation d'une expression complexe en une expression équivalente composée de la combinaison d'expressions simples, facilite un certain nombre d'opérations mathématiques[1].

Selon la taxonomie de Chikofsky et Cross [Chikofsky et Cross II, 1990], la restructuration d'un programme est définie comme une transformation source à source qui préserve la sémantique et le comportement externe d'un programme[2]. Bien que la technique de refactorisation ait été pratiquée dans différents langages de programmation depuis des années, le terme refactorisation a été introduit dans la littérature, pour la première fois, par Opdyke dans son mémoire de thèse [Opdyke, 1992] : "Les refactorisations ne modifient pas le comportement des programmes, c'est-à-dire, si un programme est appelé deux fois (avant et après la refactorisation) avec les mêmes arguments, les résultats devraient être les mêmes. Les refactorisations préservent le comportement de sorte que, lorsque les pré- conditions sont vérifiées, les résultats du programme restent les mêmes"[3].

[1] http ://en.wikipedia.org/wiki/Refactoring.

[2] Chikofsky dit : *"the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics). A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system."* [Chikofsky et Cross II, 1990].

[3] Opdyke dit : *"Refactorings do not change the behavior of a program ; that is, if the program is called*

Opdyke s'intéresse aux refactorisations qui préservent le comportement pour faciliter le développement des applications[4] pour le langage de programmation C++. Dans sa thèse de doctorat [Opdyke, 1992], il introduit la notion de refactorisation et présente plusieurs refactorisations pour le langage C++. D'une part, il y a des refactorisations de haut niveau et d'autre part des refactorisations de bas niveau, tel que renommer une variable. L'approche adoptée par Opdyke se base sur le principe des refactorisations de haut niveau qui peuvent être implantées en termes de plusieurs refactorisations de bas niveaux. Si ces dernières sont implantées correctement, alors les refactorisations à niveau élevé sont correctes. Opdyke a défini vingt-six refactorisations primitives ou de bas niveau identifiées en observant les changements effectués par les programmeurs pendant l'évolution de leur code. Et pour compléter la définition de chaque refactorisation primitive, il a établi les pré-conditions nécessaires pour assurer la préservation du comportement. Il propose trois refactorisations complexes en composant des refactorisations primitives. Selon son approche, tant que chaque refactorisation primitive préserve le comportement, alors le résultat des refactorisations complexes préserve le comportement. Ses refactorisations ont été définies pour le langage de programmation C++ mais beaucoup d'entre elles sont applicables à d'autres langages de programmation orienté objet.

Afin de faciliter l'automatisation des refactorisations, Roberts a présenté une nouvelle définition des refactorisations en permettant également des refactorisations qui changent le comportement du programme [Roberts, 1999]. Cette définition se base sur des pré-conditions et des post-conditions qui devraient être vérifiées pour appliquer la transformation du programme[5].

Le livre de Fowler auquel les auteurs de l'outil *Refactoring Browser* [Roberts *et al.*, 1997] ont participé ainsi que Opdyke et Beck, décrit des refactorisations de manière informelle et dont l'application est entièrement manuelle.

---

twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same. Refactorings are behavior preserving so that, when their preconditions are met, they do not 'break' the program." [Opdyke, 1992].

[4] L'application est désignée par le terme '*Framework*' en anglais.

[5] Roberts dit : *"A behaviour-preserving source-to-source program transformation."* [Roberts, 1999].

2.2 Applications des restructurations et des refactorisations

Dans le contexte de l'évolution de logiciel, la restructuration et la refactorisation sont utilisées pour améliorer la qualité du logiciel (par exemple, l'extensibilité, la modularité, la réutilisabilité, la complexité, la maintenance et l'efficacité du logiciel) [Mens et Tourwé, 2004]. La refactorisation et la restructuration sont également utilisées dans le contexte de la réingénierie[6] [Demeyer *et al.*, 2002]. Leur définition formelle est la suivante : "[La] réingénierie est l'inspection et la transformation du contenu d'un système[7], pour le reconstituer dans une nouvelle forme ainsi que l'implantation ultérieure de la nouvelle forme."[8] [Demeyer *et al.*, 2002]. La réingénierie est utilisée pour convertir des programmes orientés objets dont l'implantation est trop complexe en augmentant leur vitesse d'exécution et facilitant leur maintenance.

La refactorisation est utilisée pour convertir un code hérité[9] en un code plus modulaire ou plus structuré [Fanta et Rajlich, 1998]. Fanta et Rajlich ont développé trois outils automatiques pour la refactorisation de programmes écrits avec le langage de programmation C++ : un outil pour l'insertion des fonctions, un outil pour l'encapsulation et un autre pour l'expulsion. Le premier permet d'insérer une fonction en la déplaçant dans une autre classe. Le second permet d'encapsuler des fragments de code consécutifs dans une nouvelle fonction. Quant au troisième, comme son nom l'indique, il permet de déplacer une fonction de sa classe de définition. Afin de faciliter l'implantation de ces transformations, ils ont limité les fonctionnalités des transformations. Dans l'article, ils supposent que l'approche utilisée améliore la réutilisation du code, même s'ils n'ont pas analysé les deux versions de leur code (avant et après la refactorisation) pour comparer le degré de la réutilisation. La refactorisation est aussi utilisée pour migrer un code vers un langage de programmation différent ou même encore vers un paradigme de langage différent. Par exemple, Fanta et Rajlich ont restructuré un code écrit dans le langage de programmation C dans de nouvelles classes du langage de programmation C++ [Fanta

[6] Réingénierie est désignée par le mot "*reengineering*" en anglais.
[7] Entité active qui déclenche les événements modifiant l'état du système.
[8] *"Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."*
[9] Le code hérité correspond au terme '*legacy code*' en anglais.

et Rajlich, 1999]. Ils ont implanté un ensemble d'outils permettant d'exécuter des transformations automatiques afin d'encapsuler des classes à partir d'un code écrit dans un langage de programmation procédural C. Cependant, ils ont utilisé des scénarios qui combinent les outils implantés avec l'intervention humaine pour les tâches compliquées de la restructuration.

Le but principal de l'amélioration d'un système est d'effectuer des modifications importantes au code. Cela est notamment vrai avec les techniques modernes du développement itératif (ou incrémental) où le programme est en phase de modification pratiquement dès le début de son existence. Le cycle de vie itératif est fondé sur la croissance et l'affinement successifs d'un programme par le biais d'itérations multiples [Foote et Opdyke, 1995]. Comme le programme croit avec le temps de façon incrémental, itération par itération, cette méthode porte le nom de développement itératif ou incrémental. La refactorisation est une technique importante dans le développement itératif [Foote et Opdyke, 1995].

La refactorisation est aussi l'une des pratiques de base de la méthodologie XP (*eXtreme Programming*) où il est nécessaire de refactoriser le programme avant d'ajouter la fonctionnalité suivante. En octobre 2000, la publication de l'ouvrage *Extreme Programming Explained* [Beck, 1999], dans lequel Beck expliquait les fondements de sa démarche, a donné naissance à cette méthodologie. La pratique XP propose une démarche de conception continue qui fait émerger la structure de l'application au fur et à mesure du développement.

2.3 Activités de la refactorisation

Le processus de refactorisation se divise en plusieurs activités [Mens et Tourwé, 2004]

1. Identifier les endroits du programme qui devraient être refactorisés.
2. Déterminer quelle(s) refactorisation(s) devraient être appliquée(s) à ces endroits.
3. Garantir que la refactorisation appliquée préserve le comportement du système.
4. Appliquer la refactorisation.
5. Évaluer l'effet de la refactorisation sur les critères de qualité du logiciel (telles que la complexité, la compréhensibilité et la maintenabilité) ou du processus (tels que

la productivité, le coût et l'effort).

6. Maintenir la traçabilité entre le code du programme refactorisé et les autres artifacts du logiciel (tels que la documentation, la conception, les exigences des spécifications et les tests).

2.4 Préservation de comportement

2.4.1 Définition

Une définition initiale de la préservation du comportement a été introduite par Opdyke [Opdyke, 1992] : on doit obtenir les mêmes résultats pour les mêmes données d'entrées[10]. Ainsi, pour assurer cette préservation, il suggère des préconditions appliquées sur le programme transformé [Opdyke, 1992].

2.4.2 Limites de la définition

La définition de la préservation de comportement d'Opdyke, basée sur les entrées/sorties, est insuffisante pour certains domaines car les propriétés du programme considérées intéressantes pour la préservation du comportement peuvent différer d'une application à une autre. Opdyke prend en considération uniquement l'obtention des mêmes résultats des sorties pour les mêmes données d'entrées d'un point de vue calcul seulement sans estimer le temps de calcul, par exemple, dans la plupart des programmes, une transformation est considérée comme préservant le comportement, même si elle rend le programme plus rapide ou plus lent de 10 millisecondes. Mais, dans une application en temps réel, il est impossible de considérer une telle transformation comme préservant le comportement car l'aspect important du comportement d'un système interactif est le temps d'exécution de certaines opérations. Pour les systèmes embarqués[11], les contraintes de mémoire et d'énergie sont également des aspects importants du comportement qui doivent être prises en considération. Et pour les systèmes de sécurité, il y a la notion concrète de sécurité

[10] Opdyke recommande : "For the same set of input values, the resulting set of output values should be the same before and after refactoring."

[11] Les systèmes embarqués sont désignés par "embedded systems" en anglais.

qui doit être préservée par les refactorisations appliquées. En théorie, les refactorisations devraient pouvoir préserver toutes ces propriétés. Dans la pratique, ces propriétés ne sont pas préservées par toutes les entités d'un programme. De plus, peu de langages de programmation possèdent une sémantique formelle pour assurer la préservation des propriétés des programmes.

2.4.3 Techniques et formalismes des refactorisations préservant le comportement

La définition précise de la préservation du comportement est un problème récurrent en génie logiciel. Il existe certaines transformations qui sont référées comme des refactorisations préservant le comportement des programmes même si elles les modifient d'une manière spécifique. Par exemple, un programmeur peut remplacer toutes les occurrences d'une fonction par une fonction similaire, mais non identique. Ce genre de transformation du programme est toujours référée comme une refactorisation préservant le comportement.

Dans le domaine de la refactorisation, les travaux existants sont généralement fondés sur une démonstration semi-formelle de la préservation de comportement du code [Opdyke, 1992] ou aucune démonstration de la préservation du comportement n'a été mentionnée [Fowler, 1999]. Fowler suppose que ses refactorisations sont des transformations préservant le comportement d'un programme, même si aucune preuve n'ait été apportée. Pour cela, il se base sur le fait que de telles transformations n'affectent que l'apparence d'un programme sans ajouter de nouvelles fonctionnalités. En effet, les refactorisations définies par Fowler sont appliquées sur un code source dans le but de mieux appréhender le programme ou encore de faciliter des modifications fonctionnelles ultérieures [Fowler, 1999].

Une grande variété de formalismes et des techniques ont été proposés et utilisés pour traiter et garantir la préservation de comportement des programmes refactorisés, parmi celles-ci : l'utilisation des invariants, pré-conditions et post-conditions et la transformation de graphes.

2.4.3.1 Utilisation des invariants, pré-conditions et post-conditions

L'utilisation des invariants a été suggérée pour la première fois dans les travaux de Banerjee pour préserver le comportement des schémas de bases de données orientées objet [Banerjee, 87]. Opdyke a adopté cette approche en y ajoutant des pré-conditions pour les refactorisations [Opdyke, 1992]. Roberts [Roberts, 1999] a spécifié ensuite, ces pré-conditions plus formellement à l'aide du calcul de prédicats du premier ordre.
Les pré-conditions peuvent varier selon la complexité du langage de programmation utilisé. Malheureusement, la vérification statique des pré-conditions peut exiger une analyse très coûteuse ou même peut être impossible. De plus, certaines pré-conditions ne prennent pas en considération la taille ou la structure du programme [Opdyke, 1992]. Par exemple, les programmes C++ peuvent représenter l'arithmétique d'un nombre entier avec l'adresse d'une variable dans une classe. Ceci peut engendrer un problème si la refactorisation change l'ordre physique des variables dans cette classe. Pour surmonter ces problèmes, plusieurs solutions ont été proposées. Roberts suggère d'enrichir les refactorisations avec des post-conditions [Roberts, 1999]. Ces post-conditions sont notamment utiles pour les invariants qui se réfèrent à l'information dynamique. En particulier, si cette dernière est difficile à formuler ou coûteuse à vérifier statiquement avec des pré-conditions.

Il existe une correspondance directe entre l'application des refactorisations et les transformations de graphes.

2.4.3.2 Transformation de graphes

Les arbres et les graphes sont des structures de données utilisées pour représenter des programmes sous forme d'arbres syntaxiques et de graphes de flot de contrôle. De telles représentations permettent de faciliter la manipulation de ces programmes. Les programmes peuvent donc être représentés sous forme de graphes où les refactorisations correspondent à des règles de production de graphe et l'application de refactorisations correspond à des transformations de graphe.

La théorie des transformations de graphes aide à prouver certaines propriétés des

refactorisations du fait que la transformation de graphes s'applique sur une représentation abstraite du programme. Ainsi, la théorie des transformations de graphes a été utilisée pour fournir un support formel pour la refactorisation des logiciels. Mens a utilisé le formalisme de la réécriture de graphe pour prouver la préservation d'accès aux variables, la préservation de mise à jour des variables, ou encore la préservation des appels de méthodes qui peuvent être déduites statiquement du code source [Mens *et al.*, 2002]. La préservation du type est aussi une approche statique de la préservation du comportement d'un programme. Pour pouvoir vérifier des aspects plus précis de la préservation de comportement, il faut prendre en compte des informations dynamiques. Moore a proposé une notion dynamique pour la préservation des appels de méthodes où la transformation garantie que les messages dans une classe doivent être envoyés dans le même ordre [Moore, 1996].

2.5 Outils de refactorisation

Plusieurs outils ont été développés pour aider à la refactorisation[12]. Beaucoup d'entre eux se réfèrent directement à un langage de programmation spécifique. Par exemple, l'outil *Refactoring Browser* [Roberts *et al.*, 1997], développé pour le langage de programmation Smalltalk permet d'exécuter des refactorisations sur le code Smalltalk. Cet outil fournit un environnement pour améliorer la structure des programmes Smalltalk et ainsi réduire le coût de la réutilisation des logiciels.
D'autres outils sont aussi disponibles, tels que C# Refactory[13] pour C#, CodeGuide6.0[14] pour Java et Xrefactory[15] qui aide dans la modification de code C++ et Java.
Le degré d'automatisation des outils cités peut varier selon les activités de la refactorisation (citées dans la section 2.3) supportées par l'outil. Les outils tels que *Refactoring Browser*, *Xrefactory*, et *jFactor*[16] supportent l'approche semi-automatique. Cette approche a été adoptée par les environnements de développement de logiciels industriels tels

[12]http ://www.refactoring.com
[13]http ://www.xtreme-simplicity.net/CSharpRefactory.html
[14]http ://www.omnicore.com
[15]http ://www.xref-tech.com/xrefactory
[16]http ://www.instantiations.com/jfactor

que VisualWorks[17], Together[18], JBuilder[19] et Eclipse[20]. Cependant, la tâche d'identification des parties du programme qui ont besoin d'une refactorisation reste la responsabilité du développeur, de même que la sélection des refactorisations appropriées à appliquer. En se basant sur des tests effectués sur deux études de cas non triviales écrites en C++, Tokuda et Batory [Tokuda et Batory, 2001] indiquent qu'une approche semi-automatique peut augmenter la productivité (en termes de temps de codage et de débogage) en la comparant avec une refactorisation effectuée à la main. Un autre avantage primordial des outils de refactorisation est que la préservation du comportement du code refactorisé réduit de manière significative le besoin de réécrire les tests unitaires.
D'autres chercheurs se sont intéressés aux approches automatiques en développant des outils automatiques pour la refactorisation, tels que l'outil *Guru* développé par Moore [Moore, 1995; Moore, 1996]. Cet outil analyse et restructure une hiérarchie d'héritage exprimée dans le langage de programmation *SELF*. Un autre outil supportant l'approche automatique a été proposé par Casais [Casais, 1994]. Ce dernier a proposé des algorithmes pour restructurer la hiérarchie de classes suivant certaines règles du programme sans prendre en considération la demande de changement du programmeur.

L'approche semi-automatique pour le domaine de recherche de la refactorisation reste l'approche la plus utile dans la pratique du fait que les outils de refactorisation automatiques effectuent parfois trop de tâches et que certaines parties du logiciel refactorisées deviennent plus difficiles à comprendre. Une partie significative de la connaissance exigée pour exécuter la refactorisation ne peut pas être extraite à partir du logiciel, mais demeure implicitement à la charge du programmeur.

Tous les outils cités fournissent une variété de refactorisations, par exemple renommer et extraire des méthodes. Cependant, aucun outil ne gère les modèles conceptuels des programmes et autres artifacts. Le processus de rétroconception est utilisé dans Fujaba [Niere *et al.*, 2003], où l'utilisateur peut reconstruire un modèle après avoir effectué un ensemble de changements au niveau code. Une option plus efficace devrait permettre

[17]http ://smalltalk.cincom.com/
[18]http ://www.borland.com/fr/products/together
[19]http ://www.borland.com/fr/products/jbuilder
[20]Eclipse development platform home page. http ://eclipse. org

de prévoir les effets de la refactorisation sur les différentes parties du modèle. Ceci est plus facilement réalisé sur les modèles structuraux, où les transformations de tels diagrammes sont équivalents par notation à la transformation lexicale effectuée sur le code source, plutôt que sur les spécifications de comportements.

2.6 Refactorisations de modèle et traçabilité entre modèle et code

Dans la littérature, plusieurs chercheurs se concentrent sur l'idée de pouvoir produire un outil pour la refactorisation indépendant des langages de programmation orientés objet. Ainsi, Tichelaar [Tichelaar *et al.*, 2000b] propose le métamodèle FAMIX qui contient les informations utiles pour représenter les programmes orientés objet et y appliquer des refactorisations. Ces informations sont basées sur différents langages objets. Il y a cependant des différences fondamentales entre les langages objets, par exemple le typage dynamique en Smalltalk ou statique en Java, qui font que certaines refactorisations sont intrinsèquement spécifiques à un langage (ou a une famille de langage).

Dans [Sunyé *et al.*, 2001], les auteurs définissent un ensemble de refactorisations préservant le comportement appliquées sur le modèle afin d'améliorer la conception des applications orientées objets sans ajouter de nouvelles fonctionnalités. Ces refactorisations se résument aux opérations de base suivantes : l'ajout, la suppression et le déplacement des constituents des modèles de classes (par exemple, l'attribut, la méthode ou la classe) ainsi que la généralisation et la spécialisation. Afin de préserver le comportement des modèles, les transformations des diagrammes sont indiquées par des préconditions et post-conditions écrites comme des contraintes OCL (Object Constraint Language). Cependant, il n'est pas possible de répercuter ces transformations au niveau code source.

Enfin, les éditeurs de diagrammes de classe actuels n'étendent pas les changements à tous les autres types de diagrammes, limitant ainsi l'automatisation au code source seulement.

À notre connaissance, seuls les travaux de Bottoni [Bottoni *et al.*, 2005] porte sur la préservation de la traçabilité entre code source d'un programme et son modèle ; où une

approche basée sur la transformation de graphes a été proposée. Elle permet d'établir une correspondance entre les arbres de syntaxe abstraits représentant le code et les instances du métamodèle UML. Bien que l'approche ait été illustrée en utilisant les langages de programmation Java et JavaML, il est possible de l'appliquer sur d'autres langages de programmation orientés objet, à condition de pouvoir établir une correspondance non-ambiguë entre l'arbre de syntaxe et les composants du modèle de classes. Cependant, Bottoni et al. n'ont pas validé cette approche sur un outil de refactorisation pour démontrer leur résultats.

CHAPITRE 3

REFACTORISATION DES MODÈLES

Nous proposons de pallier aux limites des refactorisations portant sur leur traçabilité entre code et modèles. Pour cela, nous proposons d'abord un ensemble de refactorisations au niveau modèle avant d'étudier leur traçabilité avec des refactorisations au niveau code.

Ce chapitre décrit les refactorisations des modèles de classes que nous avons définies, notre distinction entre refactorisations primitives et refactorisations composites et l'utilisation des invariants comme approche pour la préservation du comportement des programmes. Dans la première section, nous introduisons l'approche objet, le langage de modélisation UML et, spécifiquement, les diagrammes de classes ainsi que la métamodélisation. Ensuite, nous présentons les refactorisations primitives et composites du modèle de classes que nous avons définies et leurs invariants.

3.1 De l'approche objet à l'ingénierie des modèles

Le paradigme objet, approche fondée sur l'encapsulation, le polymorphisme, la délégation, l'héritage et, particulièrement, les classes, est devenu le standard en matière de programmation et de développement de logiciels.

Grâce à une forte demande en développement, une diversité de méthodes d'analyse, de conception et de développement ont été conçues dans le but de simplifier le développement et la maintenance d'un logiciel ; en particulier, l'approche objet permet une meilleure réutilisation des composants (classes, méthodes, modules) développés. L'approche objet a donc été largement adoptée dans le monde industriel et elle est appuyée par de nombreux langages de programmation (tels que Smalltalk [Goldberg et Robson, 1983], Eiffel [Meyer, 1992], Java [Gosling *et al.*, 2000]), outils : bibliothèques de classes ou de composants (tels que Corba[1] [OMG, march 2004], COM[2], JavaBeans [Monson-Haefel,

[1]Corba. Common Object Request Broker Architecture

[2]COM. Component Object Model. http ://www.microsoft.com/com/.

2001]) et environnements de développement (tels que Eclipse[3] et NetBeans[4]).

La modélisation est une étape fondamentale dans la conception de logiciels. Elle consiste à représenter des éléments du monde réel auxquels on s'intéresse, sans se préoccuper de l'implantation, donc indépendamment d'un langage de programmation. De nombreux langages de modélisation sont apparus mais seulement trois méthodes ont véritablement influencé la modélisation objet : la méthode OMT (Object Modeling Technique) de Rumbaugh, la méthode BOOCH de Booch et la méthode OOSE (Object Oriented Software Engineering) de Jacobson. En 1996, Booch, Rumbaugh et Jacobson unissent leurs différents langages de modélisation pour unifier et donner naissance au langage de modélisation objet UML.

3.1.1 Langage de modélisation UML

UML (Unified Modeling Language, que l'on peut traduire par "langage de modélisation objet unifié") est une notation permettant de modéliser un programme de façon standard. En 1997, UML est devenu une norme OMG (Object Management Group)[5], désormais la référence en terme de modélisation objet.

Un modèle est une abstraction de la réalité. Modéliser est le processus qui consiste à identifier les caractéristiques intéressantes d'une entité en vue d'une utilisation précise dans un contexte donné. UML est donc un moyen d'exprimer des modèles objet en faisant abstraction de leur implantation, c'est-à-dire que le modèle fourni par UML devrait être valable pour n'importe quel langage de programmation orienté objet. En effet, il propose une multitude de concepts pour représenter divers vues d'un système, tant statiques que dynamiques. Un diagramme UML est une représentation graphique qui s'intéresse à une vue précise du modèle. On distingue donc deux types de vues : les vues statiques et les vues dynamiques. La première vue permet de représenter le programme physiquement à l'aide des diagrammes d'objets, des diagrammes de classes, des diagrammes de cas

[3]http ://eclipse. org/.

[4]http ://www.netbeans.org/.

[5]OMG : est une organisation à but non lucratif, dont le but est de mettre au point des standards garantissant la compatibilité entre des applications programmées à l'aide de langages objet et fonctionnant sur des réseaux hétérogènes.. Pour plus de détails consulter la page http ://www.omg.org/

d'utilisation, des diagrammes de composants et des diagrammes de déploiement. Quant à la seconde vue, elle permet de montrer le fonctionnement du programme à l'aide des diagrammes de séquence, des diagrammes de collaboration, des diagrammes d'états et des diagrammes d'activités.

Le diagramme de classes, cœur du langage de modélisation UML, est le diagramme le plus utilisé par les développeurs de programmes. Il permet de définir les composants d'un programme orienté objet. En général, on constate sa disponibilité tôt dans le cycle de vie d'un programme et en plus, il est souvent la seule forme de documentation.

Les diagrammes de classes expriment de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes [Rumbaugh *et al.*, 1999]. Une classe est une description abstraite d'un ensemble d'objets du domaine d'application. Elle définit leur structure, leur comportement et leurs relations. Dans la notation UML, la classe est représentée par un rectangle divisé en trois parties. Le nom de la classe, qui est unique dans un paquetage, se trouve dans la première partie. Dans la deuxième partie se trouvent les attributs de la classe. Dans la troisième partie prennent place toutes les méthodes (les comportements) de la classe. On distingue quatre types de relations : l'héritage, l'association, l'agrégation, et la composition [Guéhéneuc et Albin-Amiot, 2004]. Dans un diagramme de classes, les paquetages servent à organiser les différentes classes en espaces de nommages et en sous-systèmes.

3.1.2 Métamodélisation

La force du langage de modélisation UML provient du fait qu'il s'appuie sur un métamodèle, un modèle de plus haut niveau qui définit les éléments d'UML (les concepts utilisables) et leur sémantique (leur signification et leur mode d'utilisation). Le métamodèle permet de se placer à un niveau d'abstraction supérieur car il est plus abstrait que le modèle qu'il permet de construire.

UML est définie par une architecture en niveaux de métamodélisation. On appelle M0 le niveau des données ou des objets du monde réel qu'on cherche à modéliser au niveau d'abstraction supérieur M1. Pour exprimer les modèles de M1, on utilise un langage proche d'UML dont on définit la syntaxe par modélisation, au niveau M2, ce modèle

d'UML est donc un métamodèle. En pratique le langage de métamodélisation défini au niveau M3 est suffisamment abstrait et minimal pour se décrire lui-même, il est appelé MOF pour UML. Le MOF (Meta-Object Facility) est devenu une notation standard de description des métamodèles [Object Management Group, Inc., 2002] pour l'OMG[6].

La figure 3.1 illustre les quatre couches de l'architecture de métamodélisation de

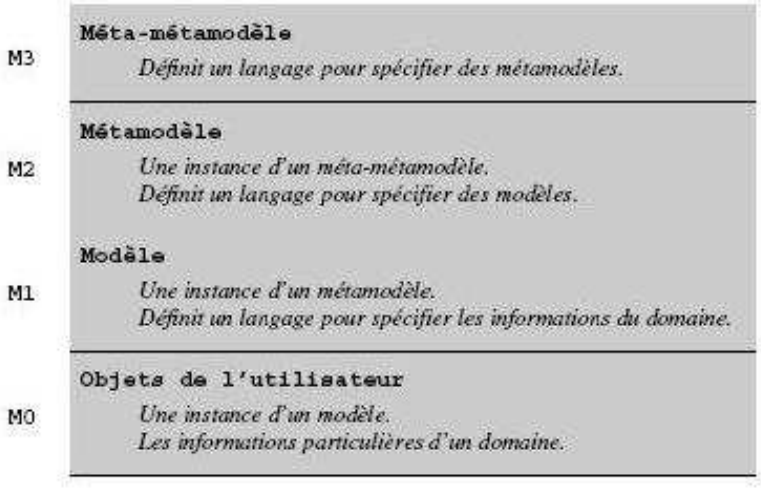

FIG. 3.1: Architecture de métamodélisation de l'OMG, composée de quatre couches [Kiczales *et al.*, 1992].

Un exemple d'utilisation de cette architecture en quatre couches est illustré par la figure 3.2

Dans l'approche objet, les concepts de modularité et de réutilisabilité sont prépondérants, pour faciliter la gestion des programmes de taille et de complexité importantes. Beaucoup de changements effectués sur un programme orienté objet, tels qu'ajouter une nouvelle classe ou renommer une méthode, sont donc possibles. De telles modifications sont référées comme des exemples de la refactorisation du code. Plusieurs refactorisations du code sont largement connues et appliquées sur le code source d'un programme. Cependant, il est important de connaître dans quelle manière elles modifient les spécifications

[6]http ://www.omg.org/

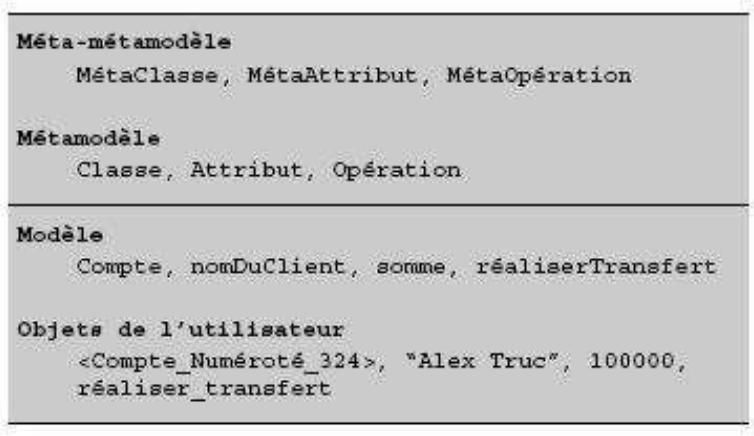

FIG. 3.2: Exemple d'utilisation de l'architecture de métamodélisation de l'OMG.

statiques et dynamiques du programme : diagrammes de classes, diagrammes d'états ou diagrammes d'activité car les transformations issues de la refactorisation de code ne sont pas toutes réellement référées de nouveau à la conception. Il est donc nécessaire d'appliquer le processus de refactorisation aux modèles de conception afin de préserver la traçabilité entre le code source d'un programme et sa spécification, exprimée par des diagrammes UML. Nous décrivons donc la refactorisation du modèle de classes en fonction des niveaux de métamodélisation décrites plus haut en mettant en application l'exemple illustré par la figure 3.2.

3.2 Refactorisation des modèles

La refactorisation des modèles UML est utile pour les trois raisons principales suivantes [Astels, 2001] :

1. Les concepteurs, les développeurs et spécifiquement les mainteneurs préfèrent visualiser leurs programmes sous forme de modèles pour simplifier la compréhension des programmes orientés objet.
2. Manipuler directement le code à un niveau plus abstrait (c'est-à-dire méthodes, variables, et classes plutôt que des caractères) peut rendre la refactorisation plus efficace.

3. Visualiser le code en utilisant des diagrammes de classes, spécifiquement le contenu des classes et les liens entre elles, peut aider dans la détection des mauvaises odeurs[7]. Beck et Fowler ont décrit les mauvaises odeurs comme certaines structures dans un code qui suggèrent la possibilité d'appliquer une refactorisation[8] [Fowler, 1999].

Selon Astels [Astels, 2001], le diagramme de classes et le diagramme de séquences prouvent particulièrement leur utilité pour le processus de refactorisation de façon complémentaire puisque le diagramme de classes donne une vue statique du programme (les classes composant le système, leur contenu et leurs liens) alors que le diagramme de séquences donne une vue dynamique (l'appel des méthodes et l'accés aux champs et aux objets).

Dans notre travail, nous s'interessons à la traçabilité entre refactorisation du modèle de classes et refactorisation du code. La refactorisation d'un modèle de classes peut donc concerner tous les éléments le constituant : les paquetages, les classes, les méthodes, les attributs et les associations en proposant des refactorisations primitives (par exemple, la création, le renommage, la suppression, la duplication des constituents du modèle de classes) et des refactorisations composites (par exemple, l'ajout, le déplacement des constituents du modèle de classes ainsi que monter, descendre des méthodes et des attributs, encapsuler un attribut et remplacer un attribut par des sous-classes). Nous définissons ainsi une liste de refactorisations utiles pour les modèles de classes des programmes orientés objet, exprimés dans un langage de modélisation proche de UML. La définition de ces refactorisations est inspirée des techniques de refactorisation proposées par Fowler [Fowler, 1999].

Il est important d'évaluer l'impact de l'application des refactorisations du modèle de classes sur le code source. Ceci est possible du fait que la plupart des refactorisations appliquées sur les constituants d'un diagramme de classes peuvent avoir une correspondance directe avec des refactorisations appliquées au code source. La préservation de la traçabilité entre refactorisation du modèle de classes et du code est donc possible en définissant avec cohérence les refactorisations correspondantes appliquées sur le code

[7]Les mauvaises odeurs est la traduction du mot *code smells* en anglais

[8]Fowler et Beck disent : *"certain structure in code that suggest the possibility of refactoring."*

source pour chaque refactorisation du modèle définie.

3.2.1 Refactorisations primitives

Nous proposons pour chaque constituant d'un modèle de classes, les cinq refactorisations primitives suivantes :

1. La création : permet de créer une instance du méta-élément constituant le modèle de classes (par exemple, méta-paquetage, méta-interface, méta-classe, méta-attribut, méta-méthode, méta-paramètre et méta-association du méta-modèle UML) avec comme nom, un nom donné par défaut.
2. Le renommage : permet de changer le nom d'un constituant d'un modèle de classes (par exemple, le paquetage, l'interface, la classe, l'attribut, la méthode, le paramètre et l'asssociation) par un nouveau nom choisi par l'utilisateur et qui ne doit pas déjà exister dans la portée du constituant.
3. La suppression : permet de supprimer un constituant d'un modèle de classes (par exemple, le paquetage, l'interface, la classe, l'attribut, la méthode, le paramètre et l'asssociation) après avoir vérifié que ce constituant ainsi que les éléments qui le composent ne sont plus référés dans le modèle de classes.
4. La duplication : permet de copier un constituant d'un modèle de classes (par exemple, le paquetage, l'interface, la classe, l'attribut, la méthode, le paramètre et l'asssociation) après avoir vérifié que le nom de ce constituant n'existe pas déjà dans l'emplacement de destination.

3.2.2 Refactorisations composites

Nous proposons aussi pour chaque constituant d'un modèle de classes les deux refactorisations composites suivantes :

1. L'ajout : permet d'ajouter un constituant d'un modèle de classes (par exemple, le paquetage, l'interface, la classe, l'attribut, la méthode, le paramètre et l'asssociation) après avoir vérifié que le nom de ce constituant est inexistant. Cette refactorisation est une refactorisation composite puisqu'elle est composée d'une séquence

de refactorisations primitives (par exemple, la refactorisation de l'ajout d'un paquetage qui est composée de deux refactorisations primitives suivantes : la création d'un paquetage et le renommage du paquetage) ou de refactorisations primitives et composites (par exemple, la refactorisation de l'ajout d'une classe qui est composée des refactorisations primitives suivantes : la création d'une classe et le renommage de la classe ainsi que des refactorisations composites suivantes : l'ajout des attributs et l'ajout des méthodes).

2. Le déplacement : permet de déplacer un constituant d'un modèle de classes (par exemple, le paquetage, l'interface, la classe, l'attribut, la méthode, le paramètre et l'asssociation) de son emplacement courant à un emplacement de destination après avoir vérifié que le nom de ce constituant est inexistant dans son nouvel emplacement. Cette refactorisation est une refactorisation composite puisqu'elle est composée des deux refactorisations primitives suivantes appliquées séquentiellement : la duplication du constituant dans l'emplacement de destination et la suppression de ce constituant de l'emplacement courant (par exemple, la refactorisation de déplacement d'une classe est composée de deux refactorisations primitives suivantes : la duplication de la classe dans le paquetage de destination et la suppression de la classe de son paquetage de définition).

Pour les méthode et les attributs, nous proposons les refactorisations composites spécifiques suivantes :

1. Monter une méthode ou un attribut : permet de déplacer un comportement (une méthode) ou une caractéristique (un attribut) dupliqué dans des sous-classes vers une superclasse commune pour éliminer la duplication.
2. Descendre une méthode ou un attribut : permet de déplacer un comportement (une méthode) ou une caractéristique (un attribut) vers une superclasse à une sous-classe ou plusieurs sous classes.
3. Encapsuler un attribut : permet de changer la visibilité d'un attribut défini public[9]

[9]La visibilité public est la traduction de public visibility en anglais

en privée[10] et de fournir des méthodes accesseurs[11] pour accéder et mettre à jour cet attribut.

4. Remplacer un attribut par des sous-classes : permet de remplacer les attributs définis avec un type numérique constant ou une énumération par des sous classes. La meilleure façon pour manipuler le comportement variant d'une classe est l'utilisation du polymorphisme. Ainsi, si on ajoute un nouvel invariant à la classe, on a seulement besoin d'ajouter une nouvelle sous-classe.

Nous avons aussi défini pour chaque refactorisation primitive et composite des pré-conditions et des post-conditions ainsi qu'une action, exprimée en pseudo code, nécessaires pour assurer la préservation du comportement. Les pré-conditions sont des conditions que le modèle doit satisfaire pour que la refactorisation puisse être appliquée avec succès et les post-conditions servent à vérifier l'état du modèle après avoir appliqué la refactorisation. L'action, quant à elle, définit concrètement la refactorisation pour transformer le modèle de classes.

Pour les refactorisations primitives, nous n'avons pas mis beaucoup de contraintes afin d'élargir leurs utilisations, contrairement aux refactorisations composites qui sont plus complexes et très spécifiques dans certains cas, nous avons définis des contraintes plus restreintes. Les deux tableaux suivants présentent en détails les deux types de refactorisations.

TAB. 3.1: Refactorisations primitives

Refactorisation primitive	**Pré-conditions**	**Action**	**Post-conditions**
Create package (*model*)	le métamodèle UML *meta-model* doit exister	créer une instance du méta-paquetage du métamodèle UML avec le nom donné par défaut au nouveau paquetage	le paquetage avec comme nom, le nom donné par défaut est créé dans le modèle UML

[10]La visibilité privé est la traduction de private visibility en anglais
[11]Les méthodes accesseurs correspondent à getter et setter en anglais

Refactorisation primitive	**Pré-conditions**	**Action**	**Post-conditions**
Rename Package (*package*, *newName*)	aucun paquetage ne doit avoir le même nom *newName* dans le modèle	renommer le paquetage *package* avec le nom *newName*	– le paquetage *package* porte le nom *newName* – mettre à jour toutes les références avec le nouveau nom *newName*
Remove package (*package*)	– le paquetage ne possède pas des classes ou bien ses classes ne sont pas référencées – le paquetage ne possède pas d'interfaces ou bien ses interfaces ne sont pas référencées – le paquetage ne doit pas avoir de lien avec les autres paquetages du modèle	supprimer le paquetage *package* du modèle	le paquetage *package* est supprimé
Copy Package (*package*, *packageName*, *model*)	le nom *packageName* ne doit pas déjà exister dans le modèle *model*	dupliquer le paquetage *package* dans le modèle *model*	le paquetage *package* est copié dans le modèle *model*
Create Interface (*package*)	le paquetage *package* doit exister dans le métamodèle UML	instancier la méta-interface du métamodèle UML avec le nom donné par défaut à la nouvelle interface	l'interface avec comme nom, le nom donné par défaut est créée dans le paquetage *package* du modèle de classes UML
Rename Interface (*interface*, *newName*)	aucune interface ne doit avoir le même nom *newName* dans le paquetage	changer le nom de l'interface *interface* par le nom *newName*	– l'interface *interface* porte le nom *newName* – mettre à jour toutes les références avec le nouveau nom *newName*

Refactorisation primitive	**Pré-conditions**	**Action**	**Post-conditions**
Remove Interface (*interface*)	l'interface ne doit pas avoir de lien avec les classes et les interfaces des paquetages du modèle	supprimer l'interface *interface* du modèle de classes UML	l'interface *interface* est supprimée
Copy Interface (*interface*, *interfaceName*, *package*)	le nom *interfaceName* de l'interface dupliquée ne doit pas exister dans le paquetage *package*	dupliquer l'interface *interface* avec le nom *interfaceName* dans le paquetage *package*	l'interface *interface* est copiée dans le paquetage *package*
Create Class (*package*)	Le paquetage *package* doit exister dans le métamodèle UML	instancier la méta-classe du métamodèle UML avec le nom donné par défaut à nouvelle classe	la classe avec comme nom, le nom donné par défaut est créée dans le paquetage *package* du modèle de classes UML
Rename Class (*class*, *newName*)	aucune classe ne doit avoir le même nom *newName* dans la même portée	changer le nom de la classe *class* par le nom *newName*	– la classe *class* porte le nom *newName* – mettre à jour toutes les références avec le nouveau nom *newName*
Remove Class (*class*)	– la classe ne possède pas des attributs ou bien ses attributs ne sont pas référencés – la classe ne possède pas des méthodes ou bien ses méthodes ne sont pas référencées – la classe ne doit pas avoir de lien avec les classes et les interfaces du modèle	supprimer la classe *class* du modèle de classes UML	– la classe *class* est supprimée – les superclasses de la classe supprimée sont devenues les superclasses de ses sous-classes

Refactorisation primitive	**Pré-conditions**	**Action**	**Post-conditions**
Copy Class (*class*, *className*, *package*)	le nom *className* de la classe dupliquée ne doit pas exister dans le paquetage *package*	dupliquer la classe *class* avec le nom *className* dans le paquetage *package*, ainsi que les méthodes et les champs qui la compose	la classe *class* est copiée dans le paquetage *package*
Create Attribute (*class*)	la classe *class* doit exister dans le modèle	créer une instance du méta-attribut du métamodèle UML avec le nom donné par défaut au nouveau attribut	l'attribut avec comme nom, le nom donné par défaut est créé dans la classe *class* du modèle de classes UML
Rename Attribute (*attribute*, *newName*)	– aucun attribut ne doit avoir le même nom *newName* dans la classe qui comporte l'attribut *attribute* – les sous-classes ne doivent pas contenir un attribut portant le nom *newName*	changer le nom de l'attribut *attribute* par le nom *newName*	– l'attribut *attribute* porte le nom *newName* – toutes les attributs qui redéfinissent l'attribut *attribute* dans les sous-classes portent le nom *newName* – les méthodes qui utilisent l'attribut *attribute* sont mises à jour
Set Visibility Attribute (*attribute*, *newVisibility*)	la visibilité *newVisibility* doit appartenir à l'ensemble des visibilités définies par le standard UML	changer la visibilité de l'attribut *attribute* par *newVisibility*	– la visibilité de l'attribut *attribute* est *newVisibility* – vérifier que l'attribut *attribute* est utilisé conformememt à sa nouvelle visibilité dans toute la hiérarchie de classes

Refactorisation primitive	Pré-conditions	Action	Post-conditions
Set Type Attribute (*attribute*, *newType*)	le type *newType* doit appartenir à l'ensemble des types définies par le standard UML	changer le type de l'attribut *attribute* par *newType*	– le type de l'attribut *attribute* est *newType* – vérifier que les attributs appropriés ont le même type *newType* dans la hiérarchie de classes
Remove Attribute (*attribute.class*)	l'attribut *attribute* n'est pas référencé dans le modèle	supprimer l'attribut *attribute* de la classe *class*	l'attribut *attribute* est supprimé de la classe *class*
Copy Attribute (*attribute*, *attributeName*, *class*)	le nom *attributeName* de l'attribut dupliqué ne doit pas exister dans la classe *class* et dans les sous classes	dupliquer l'attribut *attribute* avec le nom *attributeName* dans la classe *class*	l'attribut *attribute* est copié dans la classe *class*
Create Method (*class*)	la classe *class* doit exister dans le modèle	créer une instance de la méta-méthode du métamodèle UML avec le nom donné par défaut à la nouvelle méthode	la méthode avec comme nom, le nom donné par défaut est créée dans la classe *class* du modèle de classes UML
Rename Method (*method*, *newName*)	aucune méthode ne doit porter le même nom *newName* dans l'hiérarchie d'héritage	changer le nom de la méthode *method* par le nom *newName*	– la méthode porte le nom *newName* – les méthodes appropriées dans l'hiérarchie d'héritage ont le nom *newName*
Set Return Type Method (*method*, *newType*)	le type *newType* doit appartenir à l'ensemble des types définies par le standard UML	modifier le type de retour de la méthode *method* par le type *newType*	– le type de retour de la méthode *method* est *newType* – vérifier que les méthodes appropriées dans l'hiérarchie de classes ont le même type de retour *newType*

Refactorisation primitive	Pré-conditions	Action	Post-conditions
Set Visibility Method (*method*, *newVisibility*)	la visibilité *newVisibility* doit appartenir à l'ensemble des visibilités définies par le standard UML	changer la visibilité de l'attribut *attribute* par *newVisibility*	– la visibilité de la méthode *method* est *newVisibility* – vérifier que la méthode *method* est utilisée conformemmt à sa nouvelle visibilité dans toute la hiérarchie de classes
Remove Method (*method*)	la méthode *method* n'est pas référencée dans le modèle	supprimer la méthode *method* de la classe	la méthode *method* est supprimée de la classe.
Copy Method (*method*, *methodName*, *class*)	le nom *methodName* de la méthode dupliquée ne doit pas exister dans la classe *class* et dans l'hiérarchie d'héritage	dupliquer la méthode *method* ayant comme nom *methodName* dans la classe *class*	la méthode *method* est copiée dans la classe *class*
Create Parameter (*method*)	la méthode *method* doit exister dans le modèle	créer une instance du méta-paramètre du métamodèle UML avec le nom donné par défaut au nouveau paramètre	le paramètre avec comme nom, le nom donné par défaut est créé dans la méthode *method* du modèle de classes UML
Rename Parameter (*parameter*, *newName*)	– aucun paramètre ne doit porter le même nom *newName* dans la liste des paramètres de la méthode qui l'utilise – aucun attribut ne doit porter le nom *newName* dans la classe qui comporte la méthode qui utilise le parmètre *parameter*	changer le nom du paramètre *parameter* par le nom *newName*	– le paramètre *parameter* porte le nom *newName* – le paramètre *parameter* qui appartient aux méthodes appropriées dans l'hiérarchie d'héritage porte le nom *newName*

Refactorisation primitive	Pré-conditions	Action	Post-conditions
Set Type Parameter (*parameter*, *newType*)	le type *newType* doit appartenir à l'ensemble des types définies par le standard UML	changer le type du paramètre *parameter* par *newType*	le type du paramètre *parameter* est *newType*
Remove Parameter (*parameter*, *method*)	le paramètre *parameter* n'est pas référencé dans la méthode *method* qui l'utilise	supprimer le paramètre *parameter* de la méthode *method*	– le paramètre *parameter* est supprimé de la méthode *method* ainsi que de toute les méthodes appropriées dans l'hiérarchie d'héritage
Copy parameter (*parameter*, *parameterName*, *method*)	le nom *parameterName* du paramètre dupliqué ne doit pas déjà exister dans les paramètres de la méthode *method*	copier le paramètre *parameter* avec le nom *parameterName* dans les paramètres de la méthode *method*	le paramètre *parameter* portant le nom *parameterName* est copié dans les paramètres de la méthode *method*
Create Association (*SourceClass*, *TargetClass*, *package*)	la classe source *SourceClass* et la classe cible *TargetClass* doivent exister dans le paquetage *package*	instancier la méta-association du métamodèle UML avec le nom donné par défaut à la nouvelle association	l'association avec comme nom, le nom donné par défaut est créée dans le paquetage *package*
Rename Association (*association*, *newName*)	– aucune association ne doit avoir le même nom *newName* dans l'hiérarchie d'héritage ou le paquetage – les classes qui sont liées à cette association *association* ne doivent pas comporter aucune variable portant le nom *newName*	changer le nom de l'association *association* par le nom *newName*	l'association *association* porte le nom *newName*

Refactorisation primitive	Pré-conditions	Action	Post-conditions
Set Multiplicity Source Class (*associationName*, *newMultiplicity*)	la multiplicité *newMultiplicity* doit appartenir à l'ensemble des multiplicités définies par le standard UML	changer la multiplicité de la classe source par *newMultiplicity*	la multiplicité de la classe source est *newMultiplicity*
Set Multiplicity Target Class (*associationName*, *newMultiplicity*)	la multiplicité *newMultiplicity* doit appartenir à l'ensemble des multiplicités définies par le standard UML	changer la multiplicité de la classe cible par *newMultiplicity*	la multiplicité de la classe cible est *newMultiplicity*
Remove Association (*association*)	– l'association *association* liant deux classes n'est pas utilisée – l'association ne doit avoir aucun lien avec les autres classes du paquetage	supprimer l'association *association* du modèle UML	l'association *association* est supprimée
Copy Association (*association*, *associationName*, *SourceClass*, *TargetClass*)	le nom *associationName* de l'association dupliquée ne doit pas exister dans le paquetage	dupliquer l'association *association* avec le nom *associationName*	l'association association est copiée dans le paquetage

TAB. 3.2: Refactorisations composites

Refactorisation composite	Pré-conditions	Action	Post-conditions
Add Package (*packageName*)	aucun paquetage ne doit porter le même nom *packageName* dans le modèle de classes	1. **Create Package** (*metamodel*) 2. **Rename Package** (*package*, *newName*)	le nouveau paquetage avec le nom *packageName* est ajouté au modèle de classes UML

Refactorisation composite	Pré-conditions	Action	Post-conditions
Move Package (*package*, *packageName*, *model*)	– le paquetage *package* ne doit pas avoir aucun lien avec les autres paquetages – le sous paquetage lié (où on veut déplacer *package*) qui appartient au modèle *model* ne doit pas contenir aucun paquetage portant le même nom *packageName* du paquetage déplacé	1. **Copy Package** (*package*, *packageName*, *model*) 2. **Remove Package** (*package*)	– le paquetage *package* qui porte le nom *packageName* est déplacé au sous paquetage requis – le paquetage *package* n'est plus définie dans le sous paquetage de définition original
Add Interface (*interfaceName*, *package*, *subclasses*)	aucune interface ne doit porter le même nom *interfaceName* dans le paquetage *package*	1. **create Interface** (*package*) 2. **Rename Interface** (*interface*, *newName*)	– l'interface *interface* est ajoutée au paquetage *package* du modèle de classes UML – les sous classes *subclasses* sont les sous classe de l'interface *interfaceName*
Move Interface (*interface*, *interfaceName*, *package*)	– l'interface *interface* ne doit pas avoir aucun lien avec les autres classes du paquetage de définition – le nom *interfaceName* ne doit pas exister dans le paquetage *package* où on veut déplacer l'interface *interface*	1. **Copy interface** (*interface*, *interfaceName*, *package*) 2. **Remove interface** (*interface*)	– l'interface *interface* qui porte le nom *interfaceName* est déplacée au paquetage *package* – l'interface *interface* n'est plus définie dans le paquetage de définition originale

Refactorisation composite	**Pré-conditions**	**Action**	**Post-conditions**
Add Class (*className*, *package*, *superclasses*, *subclasses*)	– aucune classe ne doit porter le même nom *className* dans l'hiérarchie d'héritage – toutes les sous-classes sont des sous-classes de toutes les superclasses	1. **Create Class** (*package*) 2. **Rename Class** (*class*, *newName*) 3. **Tant que** (attribute n'est pas un ensemble vide) **faire** – **Add Attribute** (*attributeName*, *class*) **Fin tant que** 4. **Tant que** (methode n'est pas un ensemble vide) **faire** – **Add Method** (*methodName*, *class*) **Fin tant que**	– la nouvelle classe est ajoutée à l'hiérarchie ayant *superclasses* comme superclasses et *subclasses* comme sous-classes – les subclasses héritent de la nouvelle classe et non des superclasses
Move Class (*class*, *className*, *package*)	– la classe *class* ne doit pas avoir aucun lien avec les autres classes – le paquetage *package* ne doit pas contenir aucune classe qui a le même nom *className*	1. **Copy Class** (*class*, *className*, *package*) 2. **Remove Class** (*class*)	– la classe *class* ayant comme nom *className* est déplacée au paquetage *package* – la classe *class* n'est plus définie dans le paquetage de définition originale

Refactorisation composite	**Pré-conditions**	**Action**	**Post-conditions**
Add Attribute (*attributeName*, *newVisibility*, *newType*, *class*)	aucun attribut ne doit porter le même nom *attributeName* dans la classe *class*	1. **Create Attribute** (*class*) 2. **Rename Attribute** (*attribute*, *newName*) 3. **Set Visibility Attribute** (*attribute*, *newVisibility*) 4. **Set Type Attribute** (*attribute*, *newType*)	l'attribut attribute est ajouté à la classe class du modèle de classes UML
Move Attribute (*attribute*, *attributeName*, *class*)	– l'attribut *attribute* ne doit pas être utilisé dans la classe où il est défini – l'attribut *attribute* ne doit pas être utilisé par les autres classes de l'hiérarchie d'héritage – le nom *attributeName* de l'attribut ne doit pas exister dans la classe *class* (où on veut le déplacer)	1. **Copy Attribute** (*attribute*, *attributeName*, *class*) 2. **Remove Attribute** (*attribute*)	– l'attribut *attribute* est déplacé à la classe *class* – l'attribut *attribute* n'est plus défini dans la classe de définition originale
Pull Up Attribute (*attribute*, *superclass*)	– chaque attribut dans les sous-classes portant le même nom de l'attribut *attribute* doit avoir le même type de l'attribut *attribute* – le nom de l'attribut *attribute* ne doit pas déjà exister dans la superclasse *superclass*	1. **Copy Attribute** (*attribute*, *attributeName*, *superclass*) 2. **Si** (la visibilité de *attribute* est *private*) **Alors** – **Set Visibility Attribute** (*attribute*, *protected*) **Fin Si** 3. **Tant que** (subclasses n'est pas un ensemble vide) **faire** – **Remove attribute**	– l'attribut *attribute* est défini dans la superclasse *superclass* – l'attribut *attribute* est supprimé des sous-classes

Refactorisation composite	Pré-conditions	Action	Post-conditions
Push Down Attribute (*attribute*, *subclasses*)	– l'attribut *attribute* n'est pas utilisé dans la superclasse où il est défini – les sous-classes *subclasses* de la superclasse utilisant cet attribut ne doivent pas comporter un attribut avec un nom identique au nom de l'attribut *attribute*	1. **Tant que** (*subclasses* n'est pas un ensemble vide) **faire** – **Copy Attribute** (*attribute*, *attributeName*, *subclasses*) **Fin tant que** 2. **Remove attribute** (*attribute*)	– l'attribut *attribute* est défini dans toutes les sous-classes *subclasses* qui l'utilisent – l'attribut *attribute* est supprimé de sa classe de définition
Encapsulate Field (*attribute*, *class*)	l'attribut *attribute* est déclaré avec une visibilité public[12] dans la classe *class*	1. **Set Visibility Attribute**(*attribute*, *private*) 2. **Add Method** (*method*, *get*, *class*) 3. **Add Method** (*method*, *set*, *class*)	– l'attribut *attribute* est déclaré avec une visibilité privée dans la classe *class* – les méthodes *get* et *set* sont définies dans la classe *class*
Replace Attribute with Subclasses(*class*, *attributes*)	– les attributs de la classe *class* ont un type numérique constant ou une énumération – toutes les valeurs des types des attributs ne doivent pas changer après la création d'un objet – aucune classe appartenant à l'hiérarchie d'héritage ne doit avoir le même nom que les attributs de la classe *class*	1. **Tant que** (attributes n'est pas un ensemble vide) **faire** – **Add Class** (*attributeName*, *package*, *class*, *subclasses*) **Fin tant que** 2. **Tant que** (attributes n'est pas un ensemble vide) **faire** – **Remove Attribute** (*attribute*, *class*) **Fin tant que**	– tous les attributs avec un type constant appartenant à la classe *class* sont supprimés – les sous-classes portant les noms des attributs de la classe *class* sont des sous-classes de la classe *class*

Refactorisation composite	**Pré-conditions**	**Action**	**Post-conditions**
Add Method (*methodName*, *newVisibility*, *newType*, *class*)	aucune méthode ne doit porter le même nom *methodName* dans la classe *class*	1. **Create Method** (*class*) 2. **Rename Method** (*method*, *newName*) 3. **Set Visibility Method** (*method*, *newVisibility*) 4. **Set Return Type Method** (*method*, *newType*) 5. **Tant que** (parameter n'est pas un ensemble vide) **faire** – **Add Parameter** (*parameterName*, *method*) **Fin tant que**	la méthode *method* est ajoutée à la classe *class* du modèle de classes UML
Move Method (*method*, *methodName*, *class*)	– la méthode *method* ne doit pas être invoquée dans la classe où elle est définie – la classe *class* (où on veut déplacer la méthode) ne doit pas contenir une méthode ayant la même signature de method	1. **Copy Method** (*method*, *methodName*, *class*) 2. **Remove Method** (*method*)	– la méthode *method* est déplacée à la classe *class* – la méthode *method* n'est plus définie dans la classe de définition originale

Refactorisation composite	Pré-conditions	Action	Post-conditions
Pull Up Method (*method*, *superclass*)	– la méthode *method* doit avoir la même signature dans les sous classes – la méthode ne doit pas utiliser les attributs de sa classe de définition – la superclasse ne doit pas contenir une méthode avec une signature similaire à celle de la méthode *method*	1. **Copy Method** (*method*, *methodName*, *superclass*) 2. **Si** (la visibilité de *method* est *private*) **Alors** – **Set Visibility Method** (*method*, *protected*) **Fin Si** 3. **Tant que** (*subclasses* n'est pas un ensemble vide) **faire** – **Remove Method** (*method*) **Fin tant que**	– la méthode *method* est définie dans la superclasse *superclass* – la méthode *method* n'est plus définie dans les sous classes de l'hiérarchie d'héritage
Push Down Method (*method*, *subclasses*)	– la méthode n'est pas utilisée dans sa classe de définition – les sous-classes directes de la superclasse (où la méthode est définie) ne contiennent pas de méthodes avec une signature identique à la méthode *method*	1. **Tant que** (*subclasses* n'est pas un ensemble vide) **faire** – **Copy Method** (*method*, *methodName*, *subclass*) **Fin tant que** 2. **Remove Method** (*method*)	– la méthode *method* est définie dans les sous classes directes de la classe de définition originale – la méthode *method* n'est plus définie dans la superclasse de définition originale

Refactorisation composite	Pré-conditions	Action	Post-conditions
Add Parameter (*parameterName*, *newType*, *method*)	aucun paramètre ne doit porter le même nom *parameterName* dans la méthode *method*	1. **Create Parameter** (*method*) 2. **Rename Parameter** (*parameter*, *newName*) 3. **Set Type Parameter** (*parameter*, *newType*)	le paramètre *parameter* est ajouté à la méthode *method*
Move Parameter (*parameter*, *parameterName*, *method*)	– le paramètre *parameter* ne doit pas être utilisé par la méthode qui le contient – le nom *parameterName* du paramètre ne doit pas exister dans les paramètres de la méthode de destination (où on veut le déplacer)	1. **Copy parameter** (*parameter*, *parameterName*, *method*) 2. **Remove parameter** (*parameter*)	– le paramètre *parameter* est déplacé dans les paramètres de la méthode *method* – le paramètre *parameter* n'appartient plus aux paramètres de la méthode originale
Add Association (*associationName*, *SourceClass*, *TargetClass*, *newMultiplicitySourceClass*, *newMultiplicityTargetClass*, *package*)	aucune association ne doit porter le même nom *associationName* dans le paquetage *package*	1. **Create Association** (*SourceClass*, *TargetClass*, *package*) 2. **Rename Association** (*association*, *newName*) 3. **Set Multiplicity Source** (*associationName*, *newMultiplicitySourceClass*) 4. **Set Multiplicity Target** (*associationName*, *newMultiplicityTargetClass*)	la nouvelle association portant le nom *associationName* est ajoutée au paquetage *package* pour lier la classe source *SourceClass* et la classe cible *TargetClass*

Refactorisation composite	**Pré-conditions**	**Action**	**Post-conditions**
Move Association (*association*, *associationName*, *SourceClass*, *TargetClass*, *package*)	– l'association *association* n'est pas utile pour les classes liées – le paquetage *package* ne doit pas contenir aucune association qui a le même nom de l'association déplacée	1. **Copy Association** (*association*, *associationName*, *SourceClass*, *TargetClass*) 2. **Remove Association** (*association*, *SourceClass*, *TargetClass*)	– l'association *association* portant le nom *associationName* est déplacée au paquetage *package* – l'association *association* n'est plus définie dans le paquetage de définition original

CHAPITRE 4

IMPLANTATION DES REFACTORISATIONS

Ce chapitre décrit les refactorisations implantées et pouvant être appliquées sur un modèle de classes, les outils qui ont été intégrés et la validation de ces refactorisations sur le programme libre, QuickUML.

4.1 Refactorisations du code implantées dans Eclipse

Actuellement, plusieurs environnement de développement offrent des outils pour faciliter la refactorisation du code. Parmi ces environnements, Eclipse, un environnement de développement intégré est le plus populaire[1] dont le but est de fournir une plate-forme de développement écrite en Java. Eclipse est un logiciel avec un code source libre[2] et a été créé par OTI[3], maintenant IBM Ottawa Labs. Il utilise le concept de "plugiciel"[4] dans son architecture. Ce concept permet aux développeurs d'ajouter de nouvelles fonctionnalités qui ne sont pas fournies en standard par Eclipse par l'intermediaire de ce mécanisme d'extension de la plate-forme. Les principaux plugiciels fournis en standard avec Eclipse concernent le langage Java mais d'autres plugiciels sont aussi disponibles pour d'autres langages de programmation par exemple C++, Cobol, et d'autres aspects du développement (par exemple la modélisation avec UML et les bases de données). De plus, Eclipse intégre d'autres outils comme CVS et JUnit.

Eclipse supporte les refactorisations de renommage qui permettent de renommer une unité de compilation[5] (qui est le fichier), un type, une méthode, une variable ou un paramètre en modifiant toutes les références de l'élément renommé. D'autres refactorisations supportées par Eclipse permettent de déplacer un code, extraire des méthodes et

[1] L'environnement de développement intégré est la traduction de "Integrated Development Environment (IDE)" en anglais

[2] code source libre est la traduction de "open source" en anglais

[3] L'OTI est le sigle de "Object Technology International" en anglais

[4] Plugiciel est la traduction du mot "plug-ins"

[5] Unité de compilation est la traduction de *compilation unit* en anglais.

encapsuler des champs, etc.

La transformation s'applique à tous les fichiers qui font référence à la zone sélectionnée, sauf si les préfèrences de l'utilisateur disent le contraire. Voici les principales fonctions de refactorisation proposées par Eclipse 3.1.

- Type 1 : Changer la structure physique du code :
  - Renommer (*Rename*).
  - Déplacer (*Move*).
  - Changer la Signature d'une Méthode (*Change Method Signature*).
  - Convertir une Classe Anonyme en une classe Interne (*Convert Anonymous Class to Nested*).
  - Déplacer un Type Membre dans un Nouveau Fichier (*Move Member Type to New File*).
- Type 2 : Changer la structure logique du code au niveau classe :
  - Descendre (*Push Down*).
  - Monter (*Pull Up*).
  - Extraire une Interface (*Extract Interface*).
  - Généraliser un Type (*Generalize Type*).
  - Utiliser un Supertype si Possible (*User Supertype Where Possible*).
  - Inférer un Type Générique aux arguments (*Infer Generic Type Arguments*).
- Type 3 : Changer la structure du code à l'intérieur d'une classe :
  - Inline.
  - Extraire une Méthode (*Extract Method*).
  - Extraire une Variable Locale (*Extract Local Variable*).
  - Extraire une Constante (*Extract Constant*).
  - Introduire un Paramètre (*Introduce Parameter*).
  - Introduire une Fabrique (*Introduce Factory*).
  - Convertir une Variable locale en un Champ (*Convert Local Variable to Field*).
  - Encapsuler un Champ (*Encapsulate Field*).

Un exemple trivial de refactorisation est le renommage d'un constituant (par exemple une classe, une méthode, une variable ou un paramètre). D'autres formes de refactori-

sation plus complexes sont possibles. Par exemple, l'extraction d'un morceau de code pour créer une nouvelle méthode ou l'introduction d'une interface à la place d'une classe. Certaines refactorisations sont parfois difficiles à effectuer, en pratique, sur un gros programme. Même lorsqu'elles sont considérées triviales, par exemple, renommer une méthode, car une recherche est nécessaire pour trouver tous les endroits où cette méthode est utilisée c'est-à-dire appelée ou redéfinie pour les mettre à jour.

Théoriquement, toute refactorisation devrait pouvoir préserver la sémantique (le sens du programme transformé). Malheureusement ceci n'est pas toujours vérifié. Il faut donc vérifier que le programme fonctionne toujours (préserve sa sémantique) après l'application de chaque refactorisation, par exemple, par des tests unitaires.

Les éditeurs développés pour faciliter la refactorisation du code ne permettent pas d'étendre les changements effectués sur le code source d'un programme au niveau des modèles parce qu'ils ne gèrent pas les modèles et s'ils les gèrent, ils n'assurent pas la traçabilité. Il est donc important d'effectuer des refactorisations au niveau des modèles afin de préserver la traçabilité entre refactorisation du code et des modèles.

4.2 Problèmes rencontrés avec les modèles

Dans la litterature, on parle souvent de la refactorisation du code et rarement de refactorisation du modèle. Les refactorisations les plus connues sont celles définies par Fowler [Fowler, 1999]. Nous avons donc pris du temps pour comprendre toutes ces refactorisations du code pour savoir quelles sont celles qui peuvent être appliquées sur le modèle de classes. Ensuite, nous avons défini nos propres refactorisations appliquées sur le modèle de classes et ainsi définir les pré-conditions et les post-conditions pour assurer la préservation du comportement du modèle de classes (voir chapitre précédent). Une fois ces refactorisations bien définies, leur implantation ne nous a pas pris beaucoup de temps. Le problème majeur rencontré était d'extraire certaines informations du modèle de classes. Par exemple, dans la refactorisation de déplacement, avant de déplacer une méthode d'une classe à une autre, il faut d'abord s'assurer que cette méthode n'est pas invoquée dans sa classe de définition ce qui n'est pas évident dans un modèle de classes.

4.3 Refactorisations au niveau des modèles

Pour implanter les refactorisaions primitives et composites définies dans le chapitre précédent, nous avons utilisé le métamodèle PADL[6] [Guéhéneuc, 2003] permettant de modéliser des programmes orientés objet.

4.3.1 Le métamodèle PADL

Le métamodèle PADL (*Pattern and Abstract-level Description Language*) permet de décrire des programmes orientés-objet [Guéhéneuc, 2003]. Ce métamodèle est une extension du métamodèle PDL[7] présenté dans [Albin-Amiot, 2003] et permettant de décrire les motifs de conception. Le métamodèle PADL propose un ensemble de constituants nécessaires à la description des niveaux idiomatique[8] et conception[9] des programmes orientés objet. Ces constituants permettent de décrire la structure et une partie du comportement d'un programme.

Le métamodèle PADL comporte deux types différents de constituants, les entités (classes, interfaces) et les éléments (méthodes, champs, paramètres, associations). Il offre également un ensemble d'analyseurs pour construire des modèles à partir de différentes sources. Il est possible d'ajouter d'autres analyseurs en utilisant le patron de conception constructeur[10]. Plusieurs extensions ont été apporté au métamodèle PADL pour supporter d'autres langages de programmation orientés objets tel que C++ [Flores et Robidoux, 2004] ou le langage objet abstrait AOL[11] [Antoniol et Guéhéneuc., 2005] ou encore d'autres paradigme comme la programmation orientée aspect avec AspectJ [Guyo-

[6]PADL est l'acronyme anglais de Pattern and Abstract-level Description Language : langage de description des patrons et de niveaux d'abstraction.

[7]PDL est l'acronyme anglais de Pattern Description Language : langage de description des motifs

[8]Le niveau idiomatique est un niveau d'abstraction intermédiaire entre les niveaux implémentation et conception. ce niveau d'abstraction est nommé niveau idiomatique en référence aux idiomes de programmation qui abstraient des constituants du niveau implémentation et qui sont des motifs intermédiaires entre les niveaux implémentation et conception. Le modèle du programme au niveau idiomatique est décrit avec un diagramme de classes dont les constituants sont les classes, les interfaces, les méthodes, les champs et les relations entre classes et interfaces [Guéhéneuc, 2003].

[9]Le niveau de conception est un niveau d'abstraction qui décrit le modèle du programme avec des diagrammes de classes et des motifs des conception [Guéhéneuc, 2003].

[10]Le patron de conception constructeur est la traduction de "Builder design pattern" en anglais. C'est un des patrons de conception décrits par [Gamma *et al.*, 1994]

[11]Le langage objet abstrait est la traduction de "Abstract Object Language (AOL)" en anglais

marc'h, 2006]. La figure 4.1 présente un sous-ensemble simplifié du métamodèle PADL.

Nous avons choisi ce métamodèle parce que ces constituants nous permettent de bien caractériser et gérer les éléments du modèle de classes d'un programme orienté objet et qu'il est facilement disponible.

4.3.2 Sélection des refactorisations

Nous avons implanté un sous-ensemble de la liste des refactorisations définies dans le chapitre 3, en particulier, celles appliquées sur les méthodes. Toutes ces refactorisations exigent des vérifications pour préserver la sémantique du modèle et d'éviter les erreurs. Par exemple, avant de renommer ou d'ajouter une nouvelle méthode à une classe, une vérification est nécessaire pour s'assurer qu'aucune méthode appartenant à la hiérarchie des classes ne porte la même signature ; dans ce cas, la surcharge des méthodes[12] est désirée par l'utilisateur. Dans l'autre cas, où la surcharge n'est pas désirée par l'utilisateur, la vérification concerne seulement le nom de la méthode.

Les refactorisations implantées sont :

1. Renommer une méthode :
   - renommer une méthode dans sa classe de définition seulement avec la surchage de méthode ;
   - renommer une méthode dans sa classe de définition seulement sans la surchage de méthode ;
   - renommer une méthode et propager le renommage à la hiérarchie de classes avec la surcharge de méthodes ;
   - renommer une méthode et propager le renommage à la hiérarchie de classes sans la surcharge de méthodes ;
   - renommer une méthode et propager le renommage aux interfaces avec la surcharge de méthodes ;

[12] la surcharge des méthodes est le principe qui permet de donner le même nom à des méthodes comportant des paramètres différents et simplifie donc l'écriture de méthodes sémantiquement similaires sur des paramètres de type différent.

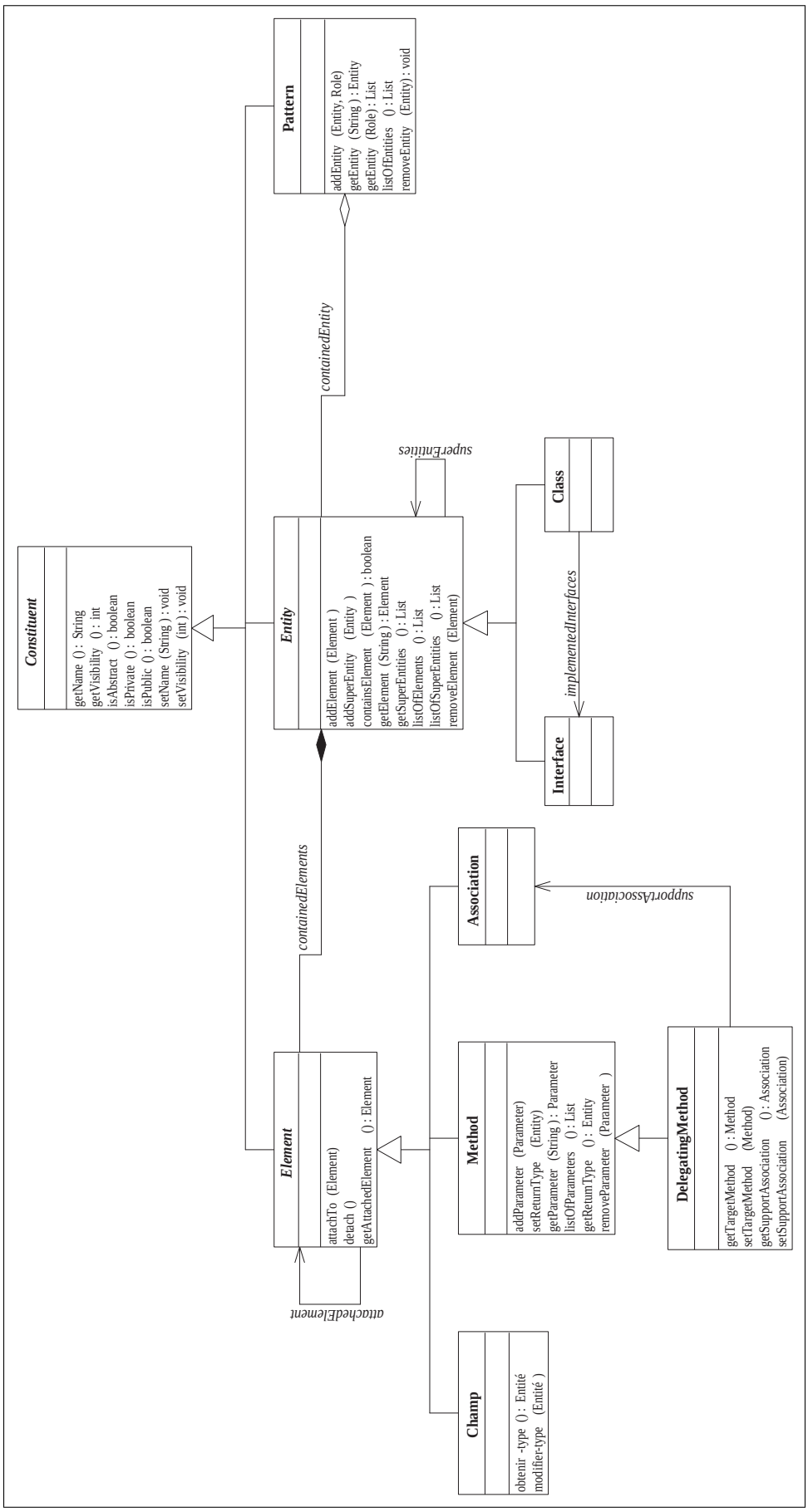

FIG. 4.1: Sous-ensemble simplifié du métamodèle PADL

   - renommer une méthode et propager le renommage aux interfaces sans la surcharge de méthodes ;
2. Monter une méthode ;
3. Descendre une méthode ;
4. Déplacer une méthode.

L'application de ces refactorisations exigent des vérifications nécessaires pour assurer la préservation de la sémantique du modèle. Ce qui nous a permis d'ajouter des algorithmes utilitaires au métamodèle PADL qui permettent de donner les classes de la hiérarchie d'une classes donnée car le métamodèle PADL fournissait seulement la liste des superclasses et des sous-classes directes d'une classe donnée.

4.3.2.1 Algorithmes utilitaires

Nous avons ajouté au métamodèle PADL des algorithmes qui facilitent la refactorisation au niveau modèle, en particulier, celles appliquée sur les méthodes. Nous avons aussi ajouté au métamodèle des algorithmes utilitaires qui permettent de donner les classes de la hiérarchie d'une classes donnée pour faciliter la vérification des refactorisations. Comme illustré dans la figure 4.2, l'implantation de ces refactorisations a été réalisée de façon générique et modulaire afin de faciliter l'ajout de d'autres types de refactorisations.

La figure 4.2 suivante présente le diagramme de refactorisations implantées par apport au métamodèle PADL.

4.3.2.2 Les refactorisations implantées

Les refactorisations appliquées sur les méthodes du modèle généré par le métamodèle PADL ont été implantées en Java. Ces refactorisations sont les suivantes :

1. **Renommer une méthode** : cette refactorisation permet de changer le nom d'une méthode, par exemple, son nom n'indique pas son but. Pour cette refactorisation, nous avons choisi de traiter tous les cas possibles en autorisant la surcharge des méthodes ou en l'interdisant, selon les préférences de l'utilisateur.

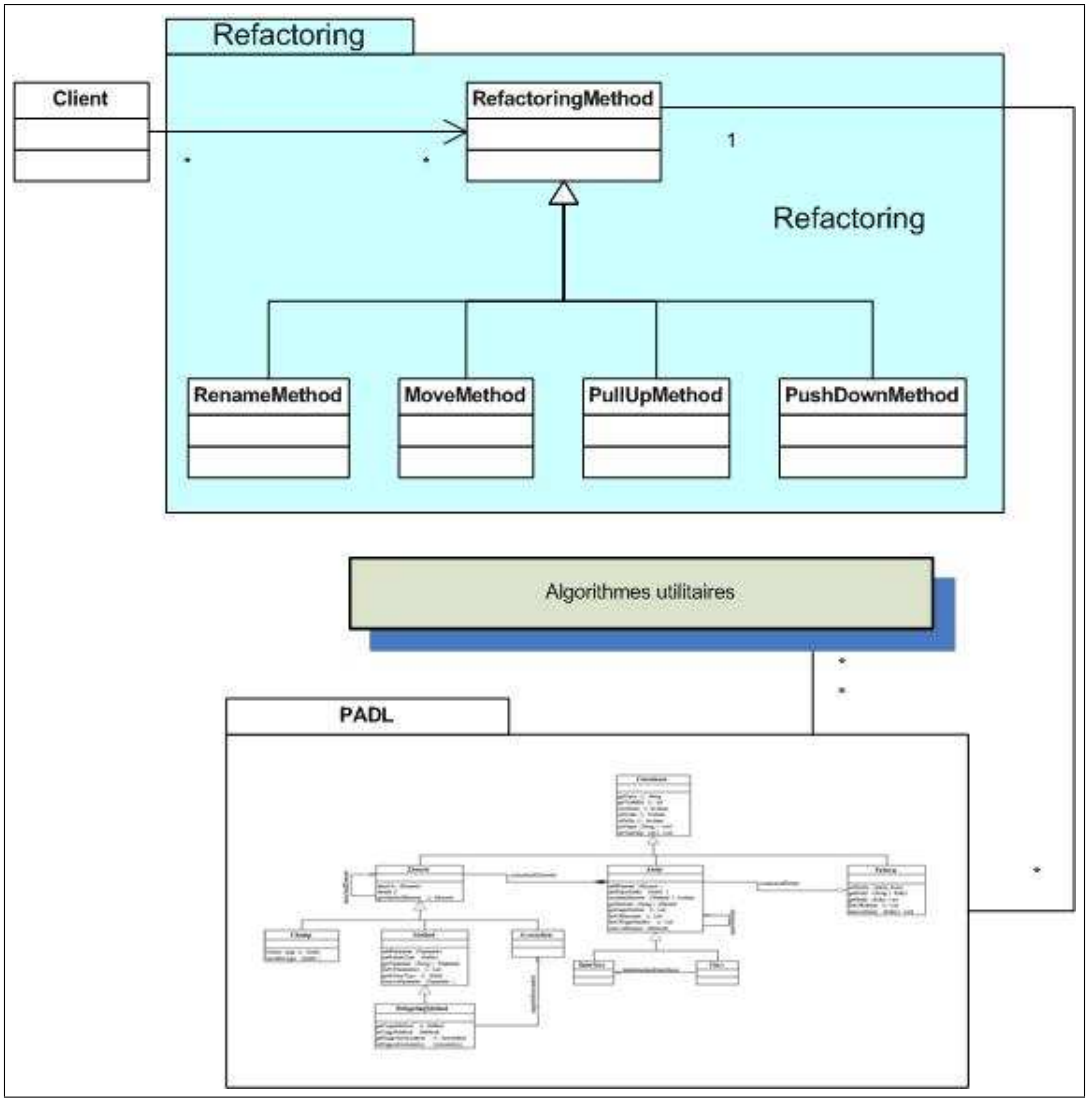

FIG. 4.2: Diagramme de refactorisations implantées par apport au métamodèle PADL.

– **Renommer une méthode dans sa classe de définition seulement sans la surchage des méthodes** correspond dans notre programme à la refactorisation **Rename Method Without Overloading** (*className*, *methodName*, *newName*) : le changement du nom est effectué seulement dans la classe portant le nom *className* où la méthode (qu'on veut renommer) est définie. Avant de changer le nom de la méthode, une vérification est requise pour s'assurer que le nouveau nom de la méthode n'existe pas dans la classe même (où elle est définie) et dans la hiérarchie des classes. Dans ce cas, la surcharge des méthodes n'est pas autorisée (c'est-à-dire, qu'il n'est pas possible d'appeler plusieurs méthodes avec le même nom).

Pré-conditions :

– aucune méthode ne doit porter le même nom *newName* dans la hiérarchie des classes.

Action : changer le nom de la méthode *method* par le nom *newName* juste dans la classe portant le nom *className*.

Post-conditions :

– la méthode *method* porte le nom *newName*.

– les méthodes appropriées dans la hiérarchie des classes n'ont pas changé de nom.

– **Renommer une méthode dans sa classe de définition seulement avec la surchage des méthodes** correspond dans notre programme à la refactorisation **Rename Method Accept Overloading**(*className*, *methodName*, *newName*) : le changement est effectué seulement dans la classe avec le nom *className* où la méthode portant le nom *methodName* est définie. Avant de changer le nom de la méthode, une vérification est requise pour s'assurer qu'aucune méthode n'a la même signature de cette méthode dans la classe de définition et dans la hiérarchie des classes. Dans ce cas, la surcharge des méthodes est autorisée (c'est-à-dire, qu'il est possible d'appeler plusieurs méthodes avec le même nom à condition que leurs arguments diffèrent en type et/ou en nombre).

Pré-conditions :

  - aucune méthode ne doit porter la même signature que la méthode portant le nom *methodName* dans la hiérarchie des classes.

  Action : changer le nom *methodName* de la méthode par le nom *newName* juste dans la classe où elle est définie.

  Post-conditions :

  - la méthode renommée porte le nom *newName*.
  - les méthodes appropriées dans la hiérarchie des classes n'ont pas changé de nom.

- **Renommer une méthode et propager le renomage à la hiérarchie de classes sans la surcharge des méthodes** correspond dans notre programme à la refactorisation **Rename Method With Propagation To Hierarchy Without Overloading** (*className*, *methodName*, *newName*) : le changement est effectué dans la classe de définition de la méthode portant le nom *methodName* ainsi que dans toute la hiérarchie de classes, sans prendre en considération les interfaces. Avant de changer le nom de la méthode, une vérification est requise pour s'assurer que le nouveau nom *newName* de la méthode n'existe pas dans la classe de définition et dans la hiérarchie des classes. Dans ce cas, la surcharge des méthodes n'est pas autorisée.

  Pré-conditions :

  - aucune méthode ne doit porter le même nom *newName* dans la hiérarchie des classes.

  Action : changer le nom *methodName* de la méthode par le nom *newName* dans toute la hiérarchie de classes.

  Post-conditions :

  - la méthode renommée porte le nom *newName*.
  - les méthodes appropriées portant le même nom *methodName* et appartenant à la hiérarchie des classes portent aussi le nom *newName*.

- **Renommer une méthode et propager le renomage à la hiérarchie de classes avec la surcharge des méthodes** correspond dans notre programme à la refactorisation **Rename Method With Propagation To Hierarchy Ac-**

cept Overloading(*className*, *methodName*, *newName*) : le changement est effectué dans la classe de définition de la méthode avec le nom *methodName* ainsi que dans toute la hiérarchie de classes, sans prendre en considération les interfaces. Avant de changer le nom de la méthode, une vérification est requise pour s'assurer qu'aucune méthode n'a la même signature que cette méthode dans la classe de définition et dans la hiérarchie des classes. Dans ce cas, la surcharge des méthodes est autorisée.

Pré-conditions :

- aucune méthode ne doit porter la même signature que la méthode portant le nom *methodName* dans la hiérarchie des classes.

Action : changer le nom *methodName* de la méthode par le nom *newName* dans toute la hiérarchie de classes.

Post-conditions :

- la méthode renommée porte le nom *newName*.
- les méthodes appropriées portant le même nom *methodName* et appartenant à la hiérarchie des classes portent aussi le nom *newName*.

– **Renommer une méthode et propager le renomage aux interfaces sans la surcharge de méthodes** correspond dans notre programme à la refactorisation **Rename Method With Propagation To Interfaces Without Overloading** (*className*, *methodName*, *newName*) : le changement est effectué dans la classe de définition de la méthode portant le nom *methodName* ainsi que dans toute la hiérarchie de classes et en propageant le renomage jusqu'aux interfaces. Avant de changer le nom de la méthode, une vérification est requise pour s'assurer que le nouveau nom *newName* de la méthode n'existe pas dans la classe de définition et dans la hiérarchie des classes incluant les interfaces. Dans ce cas, la surcharge des méthodes n'est pas autorisée.

Pré-conditions :

- aucune méthode ne doit porter le même nom *newName* dans la hiérarchie des classes, incluant les interfaces.

Action : changer le nom *methodName* de la méthode par le nom *newName* dans

toute la hiérarchie de classes incluant les interfaces.

Post-conditions :

- la méthode *method* porte le nom *newName*.
- les méthodes appropriées portant le même nom *methodName* et appartenant à la hiérarchie des classes, incluant les interfaces, portent aussi le nom *newName*.

– **Renommer une méthode et propager le renomage aux interfaces avec la surcharge de méthodes** correspond dans notre programme à la refactorisation **Rename Method With Propagation To Interfaces Accept Overloading**(*className*, *methodName*, *newName*) : le changement est effectué dans la classe de définition de la méthode ayant comme nom *methodName* ainsi que dans toute la hiérarchie de classes et en propageant le renomage jusqu'aux interfaces. Avant de changer le nom de la méthode, une vérification est requise pour s'assurer qu'aucune méthode n'a la même signature que cette méthode dans la classe même (où elle est définie) et dans la hiérarchie des classes, incluant les interfaces. Dans ce cas, la surcharge des méthodes est autorisée.

Pré-conditions :

- aucune méthode ne doit porter la même signature que la méthode portant le nom *methodName* dans la hiérarchie des classes incluant les interfaces.

Action : changer le nom *methodName* de la méthode par le nom *newName* dans toute la hiérarchie de classes incluant les interfaces.

Post-conditions :

- la méthode *method* porte le nom *newName*.
- les méthodes appropriées portant le même nom *methodName* et appartenant à la hiérarchie des classes, incluant les interfaces, portent aussi le nom *newName*.

2. **Monter une méthode** : cette refactorisation permet de déplacer des méthodes dupliquées dans des sous-classes dans une superclasse communes afin d'éliminer la duplication. Pour cette refactorisation, nous avons traité les deux cas qui permettent de monter une méthode en acceptant la surcharge de méthodes ou en l'interdisant, selon les préférences de l'utilisateur.

– **Monter une méthode sans la surcharge de méthodes** correspond dans

notre programme à la refactorisation **Pull Up Method Without Overloading**(*methodName*, *superClassName*) : cette refactorisation permet de déplacer la méthode portant le nom *methodName* définie dans les sous-classes dans une superclasse commune avec le nom *superClassName*. Dans ce cas, la surcharge des méthodes n'est pas autorisée. Le déplacement est donc effectué avec succès après avoir vérifié les pré-conditions suivantes.

Pré-conditions

- la méthode portant le nom *methodName* doit avoir la même signature dans les sous classes.
- la méthode ne doit pas utiliser les attributs de sa classe de définition.
- la superclasse ne doit pas contenir une méthode avec le même nom *methodName*.

Actions

- Ajouter la méthode avec le nom *methodName* à la superclasse portant le nom *superClassName*.
- Supprimer la méthode avec le nom *methodName* dans toutes les sous-classes.

Post-conditions

- la méthode avec le nom *methodName* est définie dans la superclasse portant le nom *superclassName*.
- la méthode avec le nom *methodName* n'est plus définie dans sa classe de définition originale.

3. **Monter une méthode avec la surcharge de méthodes** correspond dans notre programme à la refactorisation **Pull Up Method Accept Overloading**(*methodName*, *superClassName*) : cette refactorisation permet de déplacer la méthode portant le nom *methodName* définie dans les sous-classes dans une superclasse commune ayant comme nom *superClassName*. Dans ce cas, la surcharge des méthodes est autorisée. Le déplacement est donc effectué avec succès après avoir vérifié les pré-conditions suivantes.

   Pré-conditions

   - la méthode portant le nom *methodName* doit avoir la même signature dans les

sous classes.

- la méthode ne doit pas utiliser les attributs de sa classe de définition.
- la superclasse ne doit pas contenir une méthode avec une signature similaire à la signature de cette méthode.

Actions

- Ajouter la méthode avec le nom *methodName* à la superclasse portant le nom *superClassName*.
- Supprimer la méthode avec le nom *methodName* dans toutes les sous-classes.

Post-conditions

- la méthode avec le nom *methodName* est définie dans la superclasse portant le nom *superclassName*.
- la méthode avec le nom *methodName* n'est plus définie dans sa classe de définition originale.

4. **Descendre une méthode** : cette refactorisation permet de déplacer le comportement d'une superclasse à une sous-classe spécifique. Nous avons traité les deux cas suivants qui permettent de descendre une méthode en acceptant la surcharge de méthodes ou en l'interdisant, selon les préférences de l'utilisateur.
   - **Descendre une méthode sans la surcharge de méthodes** correspond à la refactorisation **Push Down Method Without Overloading**(*nameMethod*, *superClassName*, *subClassName*) : dans ce cas, la surcharge des méthodes n'est pas autorisée. Pour effectuer ce déplacement avec succès, une vérification des pré-conditions suivantes est requise.

     Pré-conditions

     - la méthode n'est pas invoquée dans sa classe de définition.
     - les sous-classes directes de la super-classe (où la méthode est définie) ne contiennent pas de méthodes avec le même nom *nameMethod* que la méthode à déplacer.

     Actions

     - ajouter la méthode avec le nom *nameMethod* à la sous-classe portant le nom *subClassName*.
     - supprimer la méthode avec le nom *nameMethod* de la super-classe portant le

nom *superClassName*.

Post-conditions

- la méthode qui porte le nom *nameMethod* est définie dans les sous-classes directes de la super-classe avec le nom *subClassName*.
- la méthode déplacée n'est plus définie dans la super-classe.

5. **Descendre une méthode avec la surcharge de méthodes** correspond à la refactorisation **Push Down Method Accept Overloading** (*nameMethod*, *superClassName*, *subClassName*) : dans ce cas, la surcharge des méthodes est autorisée. Pour effectuer ce déplacement avec succès, une vérification des pré-conditions suivantes est requise.

   Pré-conditions

   - la méthode n'est pas invoquée dans sa classe de définition.
   - les sous-classes directes de la superclasse (où la méthode est définie) ne contiennent pas de méthodes avec une signature identique à celle de la méthode à déplacer.

   Actions

   - ajouter la méthode avec le nom *nameMethod* à la sous-classe portant le nom *subClassName*.
   - supprimer la méthode avec le nom *nameMethod* de la super-classe portant le nom *superClassName*.

   Post-conditions

   - la méthode qui porte le nom *nameMethod* est définie dans les sous-classes directes de la super-classe et avec le nom *subClassName*.
   - la méthode déplacée n'est plus définie dans la super-classe.

6. **Déplacer une méthode** : cette refactorisation permet de déplacer une méthode d'une classe qui ne l'utilise pas dans une autre classe qui l'utilise davantage. Nous avons traité les deux cas suivants permettant de déplacer une méthode en acceptant la surcharge de méthodes ou en l'interdisant, selon les préférences de l'utilisateur.
   - **Déplacer une méthode sans la surcharge de méthodes** correspond à la refcatorisation **Move Method**(*methodName*, *sourceClassName*, *targetClassName*) : dans ce cas, la surcharge des méthodes n'est pas autorisée. Afin d'effec-

tuer cette refactorisation avec succès, une vérification des pré-conditions suivantes est requise.

Pré-conditions

- la méthode ayant comme nom *methodName* ne doit pas être invoquée dans sa classe de définition.
- la classe de destination portant le nom *targetClassName* ne doit pas contenir une méthode ayant le même nom de la méthode à déplacer.

Actions

- ajouter la méthode qui porte le nom *methodName* à la classe avec le nom *targetClassName*.
- supprimer la méthode avec le nom *methodName* de sa classe de définition portant le nom *sourceClassName*.

Post-conditions

- la méthode avec le nom *methodName* est déplacée dans la classe qui porte le nom *targetClassName*.
- la méthode déplacée n'est plus définie dans sa classe de définition originale portant le nom *sourceClassName*.

7. **Déplacer une méthode avec la surcharge de méthodes** correspond à la refcatorisation **Move Method Accept Overloading**(*methodName*, *sourceClassName*, *targetClassName*) : dans ce cas, la surcharge des méthodes est autorisée. Afin d'effectuer cette refactorisation avec succès, une vérification des pré-conditions suivantes est requise.

   Pré-conditions

   - la méthode ayant comme nom *methodName* ne doit pas être invoquée dans sa classe de définition.
   - la classe de destination portant le nom *targetClassName* ne doit pas contenir une méthode ayant la même signature que la méthode à déplacer.

   Actions

   - ajouter la méthode qui porte le nom *methodName* à la classe avec le nom *targetClassName*.

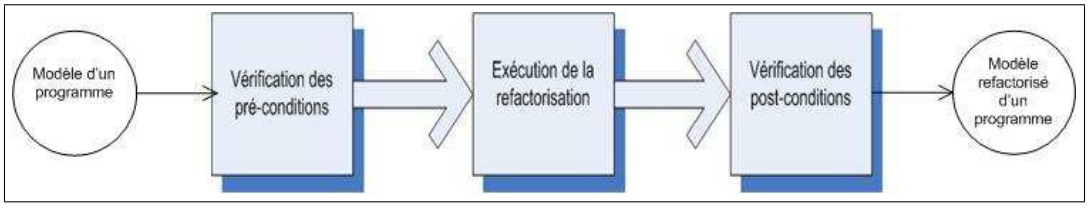

FIG. 4.3: Les trois étapes composant le processus de la refactorisation.

– supprimer la méthode avec le nom *methodName* de sa classe de définition portant le nom *sourceClassName*.

Post-conditions

– la méthode avec le nom *methodName* est déplacée dans la classe qui porte le nom *targetClassName*.

– la méthode déplacée n'est plus définie dans sa classe de définition originale portant le nom *sourceClassName*.

4.3.3 Processus de refactorisation

Le processus de la refactorisation se compose des trois étapes suivantes :

– vérification des pré-conditions et la possibilité d'appliquer la refactorisation : dans cette étape, nous vérifions les pré-conditions nécessaires à l'application de la refactorisation choisie par l'utilisateur.

– exécution de la refactorisation : cette étape permet d'exécuter les actions définies pour l'application de la refactorisation choisie par l'utilisateur.

– vérification des post-conditions : cette étapes permet de vérifier les post-conditions définies pour la refactorisation choisie par l'utilisateur, afin d'assurer la préservation du comportement du modèle et de confirmer aussi l'application de la refactorisation.

La figure 4.3 suivante illustre les trois étapes du processus de la refactorisation.

4.3.3.1 Messages d'erreurs

Si la refactorisation n'a pas été exécutée avec succès, un message d'erreur est affiché pour indiquer à l'utilisateur la cause de l'échec de l'application de la refactorisation choisie. Par exemple, si l'utilisateur choisit d'appliquer la refactorisation qui permet de

renommer une méthode sans la surcharge des méthodes (c'est-à-dire qu'il n'est pas possible d'appeler plusieurs méthodes avec le même nom) et donne un nom déjà existant . Dans ce cas, un message d'erreur est affiché indiquant que le nom choisi existe déjà.

4.3.4 Tests unitaires

JUnit[13] [Gamma et Beck, 1998] est une application[14] qui facilite l'écriture des tests unitaires. Elle permet de vérifier les différentes parties d'un programme pour en vérifier le bon fonctionnement et signaler une erreur dans le cas contraire. Elle fournit une infrastructure permettant d'exécuter tous les tests contenus dans un jeu de tests donné. JUnit définit ainsi comment structurer les tests et fournit des outils pour les exécuter.

Suivant la méthodologie eXtreme Programming *XP*, on a tout d'abord commencé à utiliser l'application JUnit implanté pour Java, avant même d'implanter nos algorithmes de refactorisation (avant d'écrire le code) afin d'assurer le bon fonctionnement de nos refactorisations. Elle nous a permis de mettre en place plus d'une vingtaine de tests unitaires qui nous ont suivis tout au long des phases de l'implantation et de la valdation.

Ceci nous a permis de définir pour chaque refactorisation implantée une classe de test regroupant un certain nombre de méthodes de test pour tester chaque refactorisation individuellement.

Nous avons ainsi ajouté le projet ***Refactoring Tests***, dans lequel on trouve le paquetage *refactoring.method.test* qui regroupe les quatres classes suivantes : ***RefactoringRenameMethodTest***, ***RefactoringMoveMethodTest***, ***RefactoringPullUpMethodTest*** et ***RefactoringPushDownMethodTest***. Chaque classe contient des suites de tests pour vérifier les conditions de chaque refactorisation.

4.3.4.1 Étude de cas : QuickUML

Pour valider les refactorisations implantées, nous utilisons le programme QuickUML, un éditeur UML développé par Excel Software[15]. Nous avons choisi ce programme car son

[13] http ://junit.org/
[14] Application est la traduction de *"Framework"* en anglais
[15] http ://www.excelsoftware.com/

code source est disponible. Nous avons aussi créé certains programmes pour les besoins des refactorisations qui permettent de monter et descendre des méthodes. L'application de ces refactorisations a été effectué automatiquement.

Des exemples de refactorisations sont données dans la section suivante pour valider l'application des refactorisations implantées. Certaines refactorisations (telles que, **Rename Method** et **Move Method**) s'appliquent sur le modèle de classes du programme QuickUMl décrit avec le métamodèle PADL obtenu par la rétro-conception. Les autres refactorisations (telles que, *Pull Up Method* et *Push Down Method*) s'appliquent sur le modèle de classes des programmes que nous avons créé.

Pour renommer la méthode *addDiagramDataListener* qui appartient à la classe *AbstractDiagramModel* du paquetage *diagram*, nous exécutons ces trois refactorisations, selon les trois cas suivants :

1. La refactorisation *RenameMethodWithPropagationToInterfaceWithoutOverloading ("diagram.AbstractDiagramModel", "addDiagramDataListener", "newName")* : dans ce cas, le renommage de la méthode *addDiagramDataListener* par le nouveau nom *newName* s'effectue avec succès dans toute la hiérarchie de classes.
2. La refactorisation *RenameMethodWithPropagationToInterfaceAcceptOverloading ("diagram.AbstractDiagramModel", "addDiagramDataListener", "add")* : dans ce cas, le renommage de la méthode *addDiagramDataListener* par le nouveau nom *add* s'effectue avec succès puisque la surcharge des méthodes est accéptée (le nom de la méthode *add* existe déjà dans la classe *DefaultDiagramModel* qui étend la classe *AbstractDiagramModel*) .
3. La refactorisation *RenameMethodWithPropagationToInterfaceWithoutOverloading ("diagram.AbstractDiagramModel", "addDiagramDataListener", "add")* : dans ce cas, un message d'erreur est produit indiquant que le nouveau nom existe déjà dans la hiérarchie de classes (le nom de la méthode *add* existe déjà dans la classe *DefaultDiagramModel* qui étend la classe *AbstractDiagramModel*). Ceci, confirme l'échec de l'application de la refactorisation du renommage.

La figure suivante 4.4 présente une capture d'écran de l'exécution de l'application des

trois refactorisations précédentes sur le modèle de classes de QuickUML. Avant l'application des trois refactorisation, le programme affiche dans la console les noms des paquetages suivis des noms des classes et des interfaces qui contiennent la méthode à refactoriser dans toutes la hiérarchie, incluant les interfaces : ***diagram.AbstractDiagramModel*** et ***diagram.DiagramModel*** ainsi que le nom ***addDiagramDataListener*** de cette méthode. Après l'application de ces trois refactorisations, le programme affiche, selon les trois cas, les résultats de l'application des refactorisation. Pour la refactorisation ***RenameMethodWithPropagationToInterfaceWithoutOverloading*** *("diagram.AbstractDiagramModel", "addDiagramDataListener", "newName")* et ***RenameMethodWithPropagationToInterfaceAcceptOverloading*** *("diagram.AbstractDiagramModel", "addDiagramDataListener", "add")*, le renommage s'effectue avec succès en affichant dans la console les noms des classes et des interfaces avec le nouveau nom de la méthode pour s'assurer que le nom est modifié dans toutes les classes et les interfaces qui contiennent l'ancien nom de la méthode renommée. Dans le premier cas, la méthode porte le nouveau nom *newName* après avoir vérifié que ce nom n'existe pas déjà dans toute la hiérarchie de classes en refusant la surcharge de méthodes. Dans le deuxième cas, comme la surcharge de méthodes est autorisée, le programme affiche la classe *DefaultDiagramModel* en plus de la classe *AbstractDiagramModel* et de l'interface *DiagramModel* qui contiennent la méthode renommée *add* car le nouveau nom *add* existe déjà dans la classe *DefaultDiagramModel* qui étend la classe *AbstractDiagramModel*. Pour la refactorisation ***RenameMethodWithPropagationToInterfaceWithoutOverloading****("diagram.AbstractDiagramModel", "addDiagramDataListener", "add")*, un message d'erreur est produit indiquant que le nouveau nom *add* existe déjà dans la hiérarchie de classes.

4.3.4.2 Autres exemples des refactorisations implantées

Les refactorisations *Pull Up Method* et *Push Down Method*) s'appliquent sur le modèle de classes des programmes que nous avons créé.

Voici le code du programme créé pour valider l'implantation de la refactorisation *Pull Up Method*.

```
package refactoring.method.test.data.PullUpMethod;
```

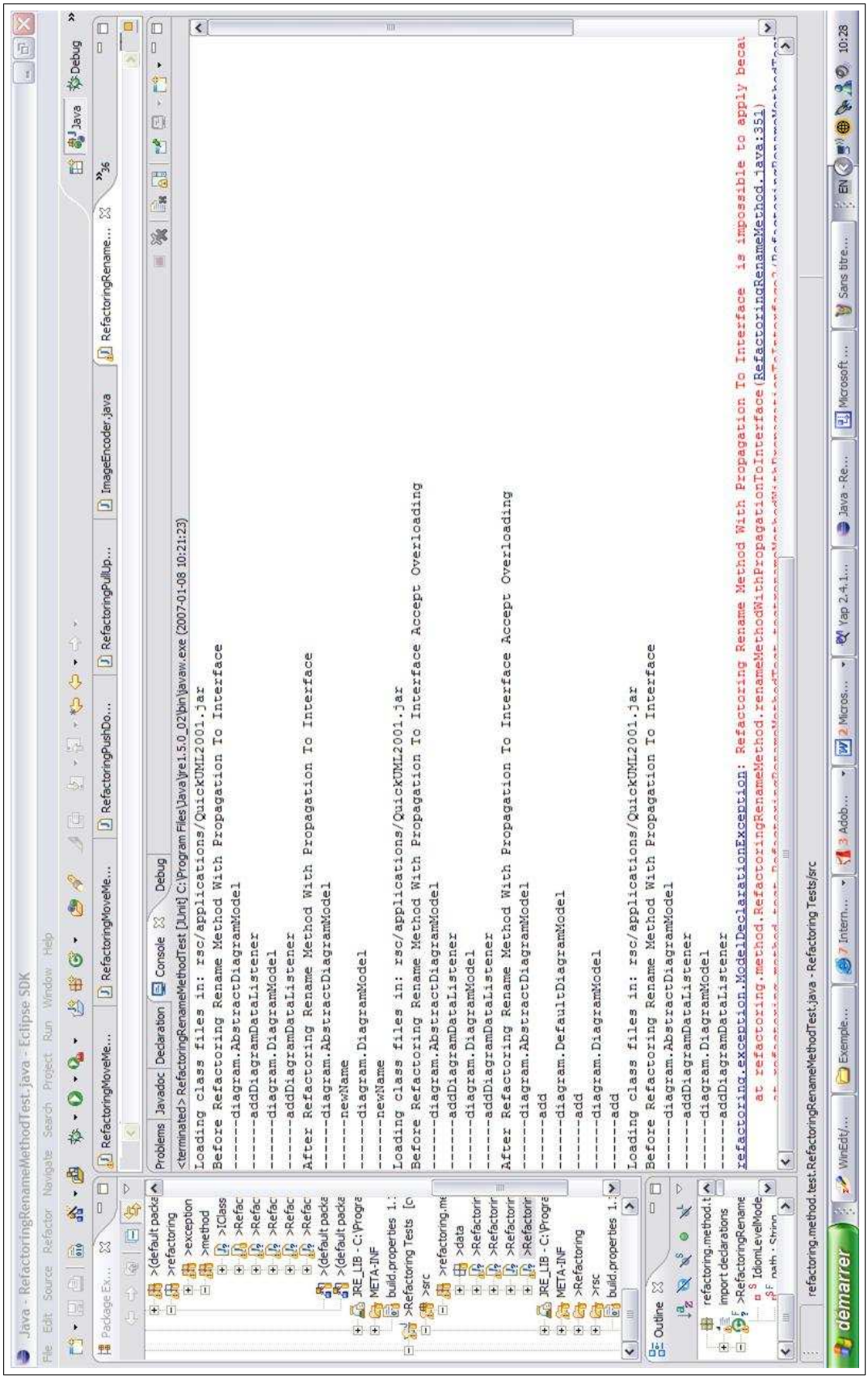

FIG. 4.4: exemple de l'application de la refactorisation de *Rename Method* sur le modèle de classes de QuickUML.

```
public class classA { public  void fooo(){
        System.out.println("je suis la classe A ");
    }
    }
public class classB extends classA{
    public void foo(){
        System.out.println("je suis foo() ");
    }
    }
public class classC extends classA{
    public void foo(){
        System.out.println("je suis foo() ");
} }
```

La figure suivante 4.5 présente une capture d'écran de l'exécution de l'application de la refactorisation de ***Pull Up Method*** sur le modèle de classes du code précédent décrit avec le métamodèle PADL obtenu par la rétro-conception. Avant l'application de cette refactorisation, le programme affiche dans la console les noms des paquetages suivis des noms des deux sous-classes qui contiennent la méthode à refactoriser : ***refactoring.method.test.data.PullUpMethod.classB*** et ***refactoring.method.test.data.PullUpMethod.classC***, ainsi que le nom ***foo*** de cette méthode . Après l'application de la refactorisation, le programme affiche le nom de la superclasse qui contient la méthode déplacée et le nom ***foo*** de la méthode refactorisée. Les noms des sous-classes ne sont pas produits puisque la méthode ***foo*** a été supprimée des sous-classes. Ce qui assure le succès l'application de cette refactorisation.

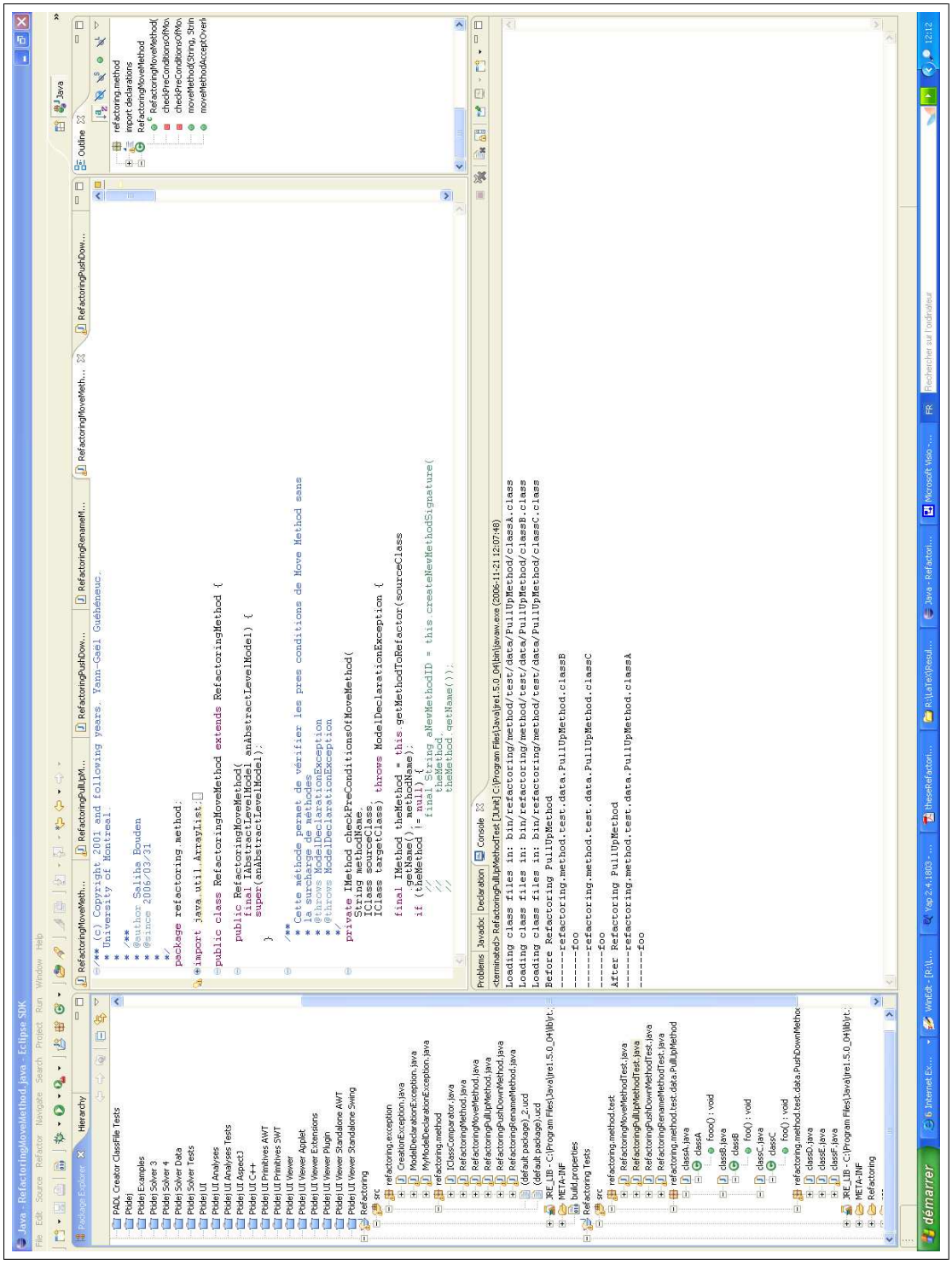

FIG. 4.5: Exemple de la refactorisation *Pull Up Method* appliquée sur le modèle de classes du code précédent.

CHAPITRE 5

TRAÇABILITÉ ENTRE REFACTORISATION DU MODÈLE ET DU CODE

Dans ce chapitre, nous étudions maintenant pour chaque refactorisation du modèle définie dans le chapitre 3, la refactorisation du code correspondante afin de préserver la traçabilité entre le modèle d'un programme refactorisé et son code source, et inversement du code au modèle. Nous voulons identifier les refactorisations à effectuer sur le code quand une refactorisation est appliquée sur le modèle et inversement.

5.1 Correspondance entre refactorisations du modèle et du code

La plupart des refactorisations appliquées sur les entités et éléments composants le modèle du programme peuvent avoir une correspondance directe avec des refactorisations appliquées au code source. Nous étudions les refactorisations du code qui correspondent aux refactorisations du modèle de classes afin de préserver la traçabilité entre la conception d'un programme et son code source.

5.1.1 Refactorisations du code

Comme mentionner dans le chapitre 1, les refactorisations de Fowler [Fowler, 1999] sont manuelles et appliquées sur le code source d'un programme écrit en Java et forment l'ensemble de refactorisations le plus complet et le plus étudiés à notre connaissance. Il est donc possible d'étudier la correspondance entre nos refactorisations du modèle et les refactorisations de Fowler pour montrer les différences et similitudes qui existent entre les deux types de refactorisations. Ces différences sont liées avec la spécification du langage de programmation et du langage de modélisation. Par exemple, lors de l'application de la refactorisation de renomage d'une méthode au niveau code, une vérification supplémentaire est requise pour les appels de la méthode renommée qui doivent être actualisées alors que pour une refactorisation de renomage d'une méthode au niveau

modèle, cette vérification est inutile du fait que l'appel de méthode n'est pas mentionné dans un modèle statique mais dans un modèle dynamique (par exemple, diagramme de séquence) que nous n'abordons pas dans ce mémoire.

Fowler présente dans son livre [Fowler, 1999] 74 refactorisations regroupées dans deux catégories différentes, les refactorisations primitives et les refactorisations composites[1]. Toutes ces refactorisations sont définies de manière informelle et avec une classification qui ne reflète pas la hiérarchie des différents concepts de l'orienté objet. Nous constatons aussi que certaines opérations s'appliquent seulement sur certains éléments (par exemple, Add Parameter et Remove Parameter qui s'appliquent que sur les paramètres et pas sur d'autres éléments d'un programme Java, comme par exemple les méthodes). Cependant, la catégorisation des refactorisations du code suivante ne facilite pas la recherche et l'application des refactorisations appropriées.

- refactorisations primitives :
  - composition ou décomposition des méthodes[2] (9 refactorisations) : par exemple, Extract Method, Remove Assignments To Parameters et Substitute Algorithm.
  - déplacement des caractéristiques entre les objets[3] (8 refactorisations) : par exemple, Move Method, Move Field et Extract Class.
  - organisation des données[4] (16 refactorisations) : par exemple, Encapsulate Field, Replace Subclass with Fields et Duplicate Observed Data.
  - simplification des expressions conditionnelles[5] (8 refactorisations) : par exemple, Decompose Conditional, Replace Conditional with Polymorphism et Introduce Assertion.
  - simplification des appels de méthodes[6] (15 refactorisations) : par exemple, Re-

[1] Dans le livre [Fowler, 1999], les refactorisations composites sont nommées par les grandes refactorisations (en anglais "big refactorings").

[2] La composition ou la décomposition des méthodes est la traduction de "Composing or Decomposing Methods".

[3] Le déplacement des caractéristiques entre les objets est la traduction de "Moving Features Between Objects".

[4] L'organisation des données est la traduction de "Organising Data".

[5] La simplification des expressions conditionnelles est la traduction de "Simplifying Conditional Expressions".

[6] La simplification des appels de méthodes est la traduction de "Simplifying Method Calls".

name Method, Add Parameter et Replace Parameter With Method.

- opération avec la généralisation[7](12 refactorisations) : par exemple, Pull Up Field, Push Down Method et Extract Interface.

- refactorisations composites :
  - Tease apart inheritance ;
  - Extract hierarchy ;
  - Convert procedural design to objects ;
  - Separate domain from presentation.

Pour plus de détails sur les refactorisation de Fowler [Fowler, 1999] ainsi que celles définies par plusieurs contibuteurs et présentées dans son site Web[8], consulter l'annexe 1. Dans cet annexe, nous présentons dans un tableau de toutes ces refactorisations avec leur descriptions et la possibilité de leur application au niveau modéle sachant que toutes les refactorisation de Fowler s'appliquent sur un code écrit en Java.

5.2 Notre classification

Contrairement à Fowler, nous avons spécifié explicitement les refactorisations associées à la création des éléments vides (par exemple, créer une classe vide dont la refactorisation correspondante est **Create Class**). De plus, Fowler ne fournit pas pour tous les éléments composant un programme orienté objet une liste exhaustive de tous les types de refactorisations (par exemple, il propose la refactorisation **Rename Method** pour le renommage d'une méthode seulement). Aussi, nous distinguons deux catégories de refactorisations : intra-élément qui s'applique au sein d'un élément (par exemple, un paquetage, une classe, une méthode) et inter-élément, qui s'appliquent entre plusieurs éléments dans l'architecture d'un programme. Cette classification est générique, car l'ajout des refactorisations composites est possible par la composition de refactorisations primitives ou composites ou encore la composition des deux types de refactorisations. Par exemple,

[7]L'opération avec la généralisation est la traduction de "Dealing with Generalisation".
[8]http ://www.refactoring.com/catalog/index.html

si nous ajoutons la refactorisation Extraire une classe[9], nous appliquons séquentiellement les refactorisations primitives et composites suivantes :

1. Créer une Classe ;
2. Renommer la Classe ;
3. Tant que (l'attribut n'est pas un ensemble vide) faire Déplacer l'Attribut ;
4. Tant que (la méthode n'est pas un ensemble vide) faire Déplacer la Méthode.

La figure 5.1 représente la classification des refactorisation du modèle de classes. Les refactorisations soulignées avec une seule ligne représentent les refactorisations pouvant être appliquées sur les modèles de classes et le code. Les refactorisations soulignées avec deux lignes représentent les refactorisations pouvant être appliquées sur les modèles de classes et le code et qui sont déjà définies par Fowler (appliquée sur le code). La refactorisation non soulignée, *Rename Association* est la seule refactorisation pouvant être appliquée sur le modèle de classes et qui n'a pas de correspondance au niveau du code car les associations ne sont pas explicite au niveau du code.

Dans la section suivante, nous reprenons la liste des refactorisations primitives et composites (55 refactorisations) présentées dans le chapitre 2 et discutons les refactorisations correspondantes au niveau du code source. Pour chaque refactorisation, nous suivons un patron présentant successivement :

- nom de la refactorisation au niveau du modèle : correspond au nom de la refactorisation du modèle présentée dans le chapitre 2.
- comparaison entre refactorisation du modèle et refactorisation du code : pour montrer les opérations nécessaires lors de l'application de la refactorisation au niveau modèle et code et comparer leurs applications.
- nom de la refactorisation au niveau du code correspondante définie par fowler : correspond au nom de la refactorisation du code définie par Fowler [Fowler, 1999] quand celle-ci existe (Fowler présente le nom des refactorisations sans paramètres) et quand elle n'existe pas nous proposons notre refactorisation avec ses paramètres.

[9]Extraire une Classe correspond à *Extract Class*

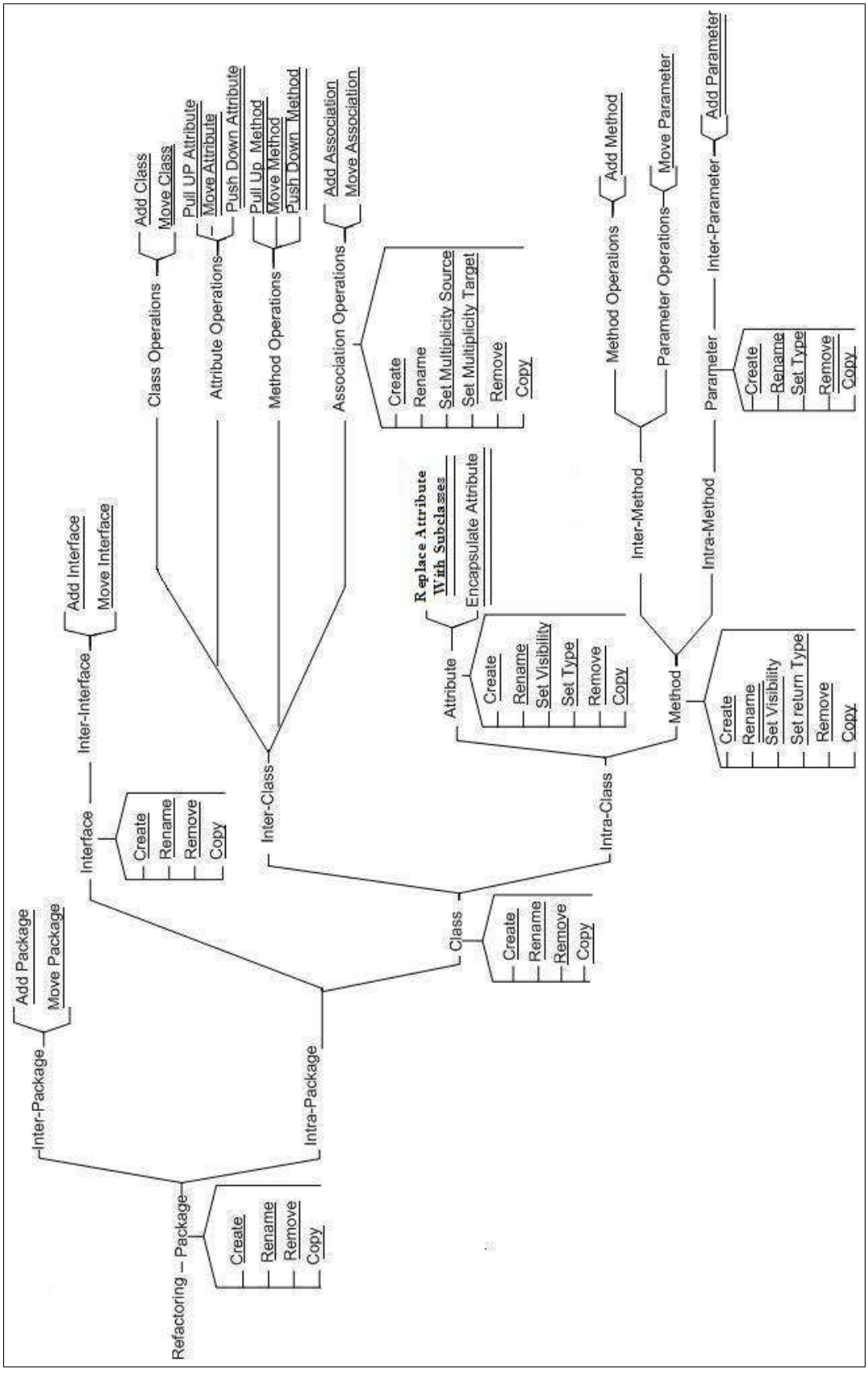

FIG. 5.1: Classification des refactorisations du modèle de classes

5.3 Refactorisations primitives appliquées sur un paquetage

5.3.1 Créer un paquetage

- **Nom de la refactorisation au niveau du modèle** : Create Package (*model*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation sur le modèle, une instance de méta-paquetage du méta-modèle UML est créée avec comme nom, le nom donné par défaut au nouveau paquetage selon la spécification du langage de modélisation UML. Cette refactorisation est exécutée avec succès au niveau du modèle après s'être assuré que le méta-modèle UML *meta-model* existe.
  - pour la refactorisation du code, l'application de cette refactorisation permet de créer un dossier dont le nom est le nom du paquetage modifié pour créer des sous dossiers. Pour appliquer cette refactorisation avec succès au niveau code, le modèle UML *model* doit exister.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent de créer des éléments vides. Mais la refactorisation correspondante que nous proposons est : Create Package (*model*).

5.3.2 Renommer un paquetage

- **Nom de la refactorisation au niveau du modèle** : Rename Package (*package*, *newName*)
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer qu'aucune instance des méta-éléments paquetage dans le modèle ne possède pas déjà le nom *newName*.
  - pour appliquer cette refactorisation sur le code, il faut s'assurer qu'aucun paquetage appartenant au programme ne porte déjà le nom *newName*. Après avoir renommer le paquetage, une mise à jour pour tous les fichiers comportant la

déclaration des classes qui se trouvent dans le paquetage renommé est nécessaire pour changer le nom de ce dernier avec le nouveau nom *newName*. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle.

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent le renommage. Mais la refactorisation correspondante que nous proposons est : Rename Package (*package*, *newName*).

5.3.3 Supprimer un paquetage

- **Nom de la refactorisation au niveau du modèle** : Remove Package (*package*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - avant de supprimer un paquetage du modèle, il faut s'assurer que ce dernier n'a pas des relations de dépendance ou de généralisation avec les autres paquetages du paquetage englobant, de même pour ses composants. Lors de l'application de cette refactorisation sur le modèle, tous les éléments (par exemple, les paquetages emboîtés, les classes, les interfaces, les associations, les généralisations et les implantations) contenus dans le paquetage supprimé sont aussi supprimés.
  - Avant de supprimer un paquetage il faut d'abord s'assurer qu'il n'est référencé dans le programme. Lors de l'application de la refactorisation de suppression d'un paquetage au niveau code, tous les fichiers qui composent le paquetage supprimé sont aussi supprimés.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent la suppression. Mais la refactorisation correspondante que nous proposons est : Remove Package (*package*).

5.3.4 Copier un paquetage

- **Nom de la refactorisation au niveau du modèle** : Copy Package (*package*, *packageName*, *model*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer la refactorisation de duplication d'un paquetage au niveau du modèle, il faut s'assurer qu'aucune instance des méta-éléments paquetage dans le modèle ne possède pas déjà le nom *packageName*. Lors de l'application de la refactorisation de duplication d'un paquetage au niveau du modèle, tous les éléments (par exemple, les paquetages emboîtés, les classes, les interfaces, les associations et les généralisations) contenus dans le paquetage dupliqué sont aussi dupliqués dans le modèle.
  - lors de l'application de cette refactorisation au niveau code, tous les fichiers contenus dans ce paquetage sont aussi copiés. Cette refactorisation est exécuté avec succès après avoir vérifier qu'aucun paquetage dans le programme ne possède pas déjà le nom *packageName*.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent la duplication des éléments d'un programme orienté objet. Mais la refactorisation correspondante que nous proposons est : Copy Package (*package*, *packageName*).

5.4 Refactorisations primitives appliquées sur une interface

5.4.1 Créer une interface

- **Nom de la refactorisation au niveau du modèle** : Create Interface (*package*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation sur le modèle, une instance de méta-interface est créée avec le nom donné par défaut à la nouvelle interface selon la spécification du langage de modélisation UML après s'être assuré que le pa-

quetage qui va comporter la nouvelle interface doit exister dans le métamodèle UML, et aucune interface de même nom ne doit pas exister.

  - pour la refactorisation du code, l'application de cette refactorisation permet de créer un fichier conformement à la déclaration d'une interface dans le langage Java et en attribuant un nom par défaut à la nouvelle interface. Pour appliquer cette refactorisation, une vérification est nécessaire pour s'assurer que le paquetage qui va comporter la nouvelle interface doit exister sur disque et aucune interface de même nom ne doit pas exister.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent de créer des éléments vide. Mais la refactorisation correspondante que nous proposons est : Create Interface (*package*).

5.4.2 Renommer une interface

- **Nom de la refactorisation au niveau du modèle** : Rename Interface (*interface*, *newName*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer qu'aucune instance des méta-éléments interface dans le paquetage ne possède pas déjà le nom *newName*.
  - pour appliquer cette refactorisation sur le code, il faut s'assurer qu'aucune interface dans le paquetage ne porte le nom *newName*. Après avoir renommer l'interface *interface*, une mise à jour pour toutes les classes qui implémentent l'interface renommé et toutes les interfaces qui l'étendent est nécessaire. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui

permettent le renommage. Mais la refactorisation correspondante que nous proposons est : Rename Interface (*interface*, *newName*).

5.4.3 Supprimer une interface

- **Nom de la refactorisation au niveau du modèle** : Remove Interface (*interface*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation au niveau du modèle, la relation d'implémentation et de généralisation existantes respectivement entre l'interface supprimée et les interfaces et les classes du modèle sont aussi supprimées.
  - lors de l'application de cette refactorisation au niveau code, une mise à jour pour toutes les classes qui implémentent cette interface ou les classes qui l'étendent nécessaire pour supprimer de leur déclarations le nom de l'interface supprimée, après avoir supprimer le fichier comportant la déclaration de cette interface du paquetage. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent la suppression. Mais la refactorisation correspondante que nous proposons est : Remove Interface (*interface*).

5.4.4 Copier une interface

- **Nom de la refactorisation au niveau du modèle** : Copy Interface (*interface*, *interfaceName*, *package*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation au niveau du modèle, il faut s'assurer qu'aucune instance des méta-éléments interface dans le modèle ne possède pas déjà le nom *interfaceName*.

  - lors de l'application de cette refactorisation au niveau code, tous les attributs et les signatures des méthodes contenus dans cette interface sont aussi copiés. Cette refactorisation est exécuté avec succès après avoir vérifier qu'aucune interface dans le paquetage ne possède pas déjà le nom *interfaceName*.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent la duplication des éléments d'un programme orienté objet. Mais la refactorisation correspondante que nous proposons est : Copy Interface (*interface*, *interfaceName*, *package*).

5.5 Refactorisations primitives appliquées sur une classe

5.5.1 Créer une classe

- **Nom de la refactorisation au niveau du modèle** : Create Class (*package*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation sur le modèle, une instance de méta-classe est créée avec un nom donné par défaut à la nouvelle classe selon la spécification du langage de modélisation UML après s'être assuré que le paquetage qui va comporter la nouvelle classe doit exister dans le métamodèle UML, et aucune classe de même nom ne doit pas exister.
  - pour la refactorisation du code, l'application de cette refactorisation permet de créer un fichier conformement à la déclaration d'une classe dans le langage Java et en attribuant un nom par défaut à la nouvelle classe. Pour appliquer cette refactorisation, une vérification est nécessaire pour s'assurer que le paquetage qui va comporter la nouvelle classe doit exister sur disque et aucune classe de même nom ne doit pas exister.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des

refactorisations qui permettent de créer des éléments vide. Mais la refactorisation correspondante que nous proposons est : Create Class (*package*).

5.5.2 Renommer une classe

- **Nom de la refactorisation au niveau du modèle** : Rename Class (*class*, *newName*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer qu'aucune instance des méta-éléments classe dans le paquetage ne possède pas déjà le nom *newName*.
  - pour appliquer cette refactorisation sur le code, il faut aussi s'assurer qu'aucune classe dans le paquetage ne porte le nom *newName*. Après avoir renommer la classe *class*, une mise à jour pour toutes les classes qui étendent la classe renommé est nécessaire. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent le renommage. Mais la refactorisation correspondante que nous proposons est : Rename Class (*class*, *newName*).

5.5.3 Supprimer une classe

- **Nom de la refactorisation au niveau du modèle** : Remove Class (*class*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation au niveau du modèle, la relation de généralisation ou d'implémentation existante respectivement entre la classe supprimé et ses sous-classes ou ses intefaces sont aussi supprimées. ce qui signifie que les superclasses de la classe supprimée sont devenues les superclasses de ses sous-classes.

  - lors de l'application de cette refactorisation au niveau code, une mise à jour pour toutes les classes qui étendent cette classe est nécessaire pour supprimer de leur déclarations le nom de la classe supprimée, après avoir supprimer le fichier comportant la déclaration de cette classe du paquetage. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent la suppression. Mais la refactorisation correspondante que nous proposons est : Remove Class (*class*).

5.5.4 Copier une classe

- **Nom de la refactorisation au niveau du modèle** : Copy Class (*class*, *className*, *package*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation au niveau du modèle, il faut s'assurer qu'aucune instance des méta-éléments classe dans le modèle ne possède pas déjà le nom *className*. Lors de l'application de cette refactorisation, tous les attributs et les méthodes contenus dans cette classe sont copiés.
  - lors de l'application de cette refactorisation au niveau code, tous les attributs et les méthodes contenus dans cette classe sont aussi copiés. Cette refactorisation est exécuté avec succès après avoir vérifier qu'aucune classe dans le paquetage ne possède pas déjà le nom *className*.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent la duplication des éléments d'un programme orienté objet. Mais la refactorisation correspondante que nous proposons est : Copy Class (*class*, *className*, *package*).

5.6 Refactorisations primitives appliquées sur un attribut

5.6.1 Créer un attribut

- **Nom de la refactorisation au niveau du modèle** : Create Attribute (*class*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation sur le modèle, une instance de méta-attribut est créée en donnant des noms et valeurs par défaut aux champs (visibilité, le nom et le type de l'attribut) après s'être assuré que la classe qui va comporter l'attribut créé doit exister dans le modèle, et aucun attribut de même nom ne doit pas exister.
  - pour la refactorisation du code, l'application de cette refactorisation permet de créer un attribut conformement à la déclaration d'un attribut dans le langage Java et en attribuant des noms et valeurs par défaut aux champs (visibilité, le nom et le type l'attribut). Pour appliquer cette refactorisation, une vérification est nécessaire pour s'assurer que la classe qui va comporter le nouveau attribut doit exister dans le programme et aucun attribut de même nom ne doit pas exister.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent de créer des éléments vide. Mais la refactorisation correspondante que nous proposons est : Create Attribute (*class*).

5.6.2 Renommer un attribut

- **Nom de la refactorisation au niveau du modèle** : Rename Attribute (*attribute*, *newName*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer qu'aucune instance des méta-éléments attribut dans la classe ne possède pas déjà le nom *newName*. Lors de l'application de cette refactorisation

au niveau du modèle, les paramètres de toutes les méthodes des classes utilisant l'attribut renommé sont mis à jour.

- pour appliquer cette refactorisation sur le code, il faut s'assurer qu'aucun attribut dans la classe ne porte le nom *newName*. Lors de l'application de cette refactorisation au niveau du code, tous les appels de méthodes ainsi que les paramètres et le corps de toutes les méthodes utilisant l'attribut renommé sont mis à jour.

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent le renommage. Mais la refactorisation correspondante que nous proposons est : Rename Attribute (*attribute*, *newName*).

5.6.3 Changer la visibilité d'un attribut

– **Nom de la refactorisation au niveau du modèle** : Set Visibility Attribute (*attribute*, *newVisibility*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

- pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer que la nouvelle visibilité *newVisibility* appartient à l'ensemble des visibilités définies par le standard UML.
- pour appliquer cette refactorisation sur le code, il faut s'assurer que la nouvelle visibilité *newVisibility* appartient à l'ensemble des visibilités : public, protégé, privé[10] définies par le langage Java pour déterminer la portée des variables. Lors de l'application de cette refactorisation au niveau du code, une vérification pour tous les appels de méthodes dans toutes les classes est nécessaire pour utiliser l'attribut conformement à son nouveau type de visibilité (c'est à dire si la visibilité a été changé de protéger à priver, alors l'attribut doit être utilisé dans sa classe

[10] L'ensemble des visibilités public, protégé, privé est la traduction respectivement de public, protected, private en anglais

de définition seulement et ainsi interdire son utilisation par les sous-classes). Cela peut donc “casser” le code et devrait empêcher l’application de la refactorisation. Cette opération est inutile lors de l’application de cette refactorisation sur le modèle, car les attributs ne sont pas référencés dans les diagrammes de classes mais dans d’autres diagrammes (par exemple, interactions).

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Dans le catalogue [Fowler, 1999], Fowler a proposé une refactorisation qui permet d’encapsuler un champs en changeant sa visibilité de public à priver, un cas spécial que nous allons citer plus tard. Donc, Fowler n’a pas proposé une refactorisation qui traite le cas général. La refactorisation correspondante que nous proposons est : Set Visibility Attribute (*attribute*, *newVisibility*).

5.6.4 Changer le type d’un attribut

– **Nom de la refactorisation au niveau du modèle** : Set Type Attribute (*attribute*, *newType*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :
  – pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s’assurer que le nouveau type *newType* est accépté par le standard UML. Lors de l’application de cette refactorisation sur le modèle, tous les paramètres des méthodes des classes utilisant l’attribut *attribute* sont mis à jour pour utiliser cet atribut avec son nouveau type.
  – lors de l’application de cette refactorisation sur le code, une mise à jour pour tous les paramètres des appels de méthodes ainsi que les paramètres et le corps des méthodes de toutes les classes est nécessaire pour utiliser l’attribut avec son nouveau type, si possible afin d’éviter les erreurs de compilation. Cette opération est inutile lors de l’application de cette refactorisation sur le modèle, car les attributs ne sont pas référencés dans les diagrammes de classes mais dans d’autres diagrammes (par exemple, interactions).

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du

code ne correspond à notre refactorisation puisque Fowler n'a pas proposé une refactorisation qui permet de changer le type d'un attribut dans le cas général (la refactorisation *Encapsulate Field* proposée par Fowler traite le changement de la visibilité de public à privée, donc un cas spécifique). Mais la refactorisation correspondante que nous proposons est : Set Type Attribute (*attribute*, *newType*).

5.6.5 Supprimer un attribut

- **Nom de la refactorisation au niveau du modèle** : Remove Attribute (*attribute*, *class*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer que l'attribut *attribute* n'est pas référencé dans le modèle.
  - lors de l'application de cette refactorisation sur le code, une mise à jour pour tous les paramètres des appels de méthodes ainsi que les paramètres et le corps des méthodes de toutes les classes est nécessaire pour s'assurer que l'attribut supprimé n'est plus utilisé. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle de classes.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent la suppression. Mais la refactorisation correspondante que nous proposons est : Remove Attribute (*attribute*, *class*).

5.6.6 Copier un attribut

- **Nom de la refactorisation au niveau du modèle** : Copy Attribute (*attribute*, *attributeName*, *class*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation au niveau du modèle, il faut s'assurer qu'aucune instance des méta-éléments attribut dans la classe *class* ne possède pas déjà

le nom *attributeName*. Lors de l'application de cette refactorisation, l'attribut est copié avec sa visibilité et son type.

- pour appliquer cette refactorisation au niveau code, il faut s'assurer qu'aucun attribut dans la classe *class* ne possède pas déjà le nom *attributeName*. Lors de l'application de cette refactorisation, l'attribut est copié avec sa visibilité et son type.

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent la duplication des éléments d'un programme orienté objet. Mais la refactorisation correspondante que nous proposons est : Copy Attribute (*attribute*, *attributeName*, *class*).

5.7 Refactorisations primitives appliquées sur une méthode

5.7.1 Créer une méthode

– **Nom de la refactorisation au niveau du modèle** : Create Method (*class*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

- lors de l'application de cette refactorisation sur le modèle, une instance de méta-méthode est créée en donnant des noms et valeurs par défaut aux champs (la visibilité, le type de retour, le nom et les paramètres de la méthode) après s'être assuré que la classe qui va comporter la méthode doit exister dans le modèle et qu'aucune méthode de même nom n'existe.
- pour la refactorisation du code, l'application de cette refactorisation permet de créer une méthode conformement à la déclaration d'une méthode dans le langage Java et en attribuant des noms et valeurs par défaut aux champs (la visibilité, le type de retour, le nom et les paramètres de la méthode) et un corps vide à la nouvelle méthode créée. Pour appliquer cette refactorisation, une vérification est nécessaire pour s'assurer que la classe qui va comporter la nouvelle méthode doit exister dans le programme et aucune méthode de même nom ne doit pas exister.

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent de créer des éléments vide. Mais la refactorisation correspondante que nous proposons est : Create Method (*class*).

5.7.2 Renommer une méthode

- **Nom de la refactorisation au niveau du modèle** : Rename Method (*method*, *newName*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer qu'aucune instance des méta-éléments méthode dans la classe ne possède pas déjà le nom *newName*.
  - pour appliquer cette refactorisation sur le code, il faut s'assurer qu'aucune méthode dans la classe ne porte le nom *newName*. Lors de l'application de cette refactorisation au niveau du code, une mise à jour pour tous les appels de méthodes qui se réfèrent à l'ancien nom de la méthode est nécessaire pour les référer au nouveau nom. Cette mise à jour est inutile lors de l'application de cette refactorisation sur le modèle, car l'appel de méthode n'est pas mentionné dans le modèle de classes.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Rename Method.

5.7.3 Changer le type de retour d'une méthode

- **Nom de la refactorisation au niveau du modèle** : Set Return Type Method (*method*, *newType*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer que le nouveau type *newType* est accépté par le standard UML.
  - lors de l'application de cette refactorisation sur le code, une mise à jour pour tous les appels de méthodes est nécessaire pour s'assurer que cette méthode est

invoquée avec son nouveau type de retour, si possible afin d'éviter les erreurs de compilation. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle, car l'appel de méthode n'est pas mentionné dans le modèle de classes.

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé une refactorisation qui permet de changer le type de retour d'une méthode. Mais la refactorisation correspondante que nous proposons est : Set Return Type Method (*method*, *newType*).

5.7.4 Changer la visibilité d'une méthode

- **Nom de la refactorisation au niveau du modèle** : Set Visibility Method (*method*, *newVisibility*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - l'application de cette refactorisation sur le modèle permet de changer la visibilité de la méthode *method*, après avoir vérifier que la nouvelle visibilité est accéptée par le standard UML.
  - pour appliquer cette refactorisation sur le code, il faut s'assurer que la nouvelle visibilité *newVisibility* appartient à l'ensemble des visibilités : public, protégé, privé[11] définies par le langage Java pour déterminer la portée des variables. Lors de l'application de cette refactorisation au niveau du code, une vérification pour tous les appels de méthode dans toutes les classes utilisant cette méthode est nécessaire pour l'utiliser conformement à son nouveau type de visibilité (c'est à dire si la visibilité a été changée de protégée à privée, alors la méthode doit être utilisée dans sa classe de définition seulement et ainsi interdire son utilisation par les sous-classes). Cette opération est inutile lors de l'application de cette refactorisation sur le modèle, car l'appel de méthode n'est pas mentionné dans

[11] L'ensemble des visibilités public, protégé, privé est la traduction respectivement de public, protected, private en anglais

le modèle de classes mais dans le diagramme de séquence.

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Dans le catalogue [Fowler, 1999], Fowler a proposé une refactorisation qui permet de cacher une méthode en changeant sa visibilité de public à priver, un cas spécial, Hide Method. Donc, Fowler n'a pas proposé une refactorisation qui traite le cas général. La refactorisation correspondante que nous proposons est : Set Visibility Method (*method*, *newVisibility*).

5.7.5 Supprimer une méthode

- **Nom de la refactorisation au niveau du modèle** : Remove Method (*method*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer que la méthode *method* n'est pas référencé dans le modèle.
  - lors de l'application de cette refactorisation sur le code, une mise à jour pour tous les appels de méthode dans les classes utilisant cette méthode est nécessaire pour s'assurer que la méthode supprimée n'est plus utilisée. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle, car l'appel de méthode n'est pas mentionné dans le modèle de classes.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent la suppression. Mais la refactorisation correspondante que nous proposons est : Remove Method (*method*).

5.7.6 Copier une méthode

- **Nom de la refactorisation au niveau du modèle** : Copy Method (*method*, *methodName*, *class*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation au niveau du modèle, il faut s'assurer qu'au-

cune instance des méthodes de la classe *class* ne possède déjà le nom *methodName*. Lors de l'application de cette refactorisation, la méthode est copiée avec sa visibilité, son type de retour et la liste de ses paramètres.

  - pour appliquer cette refactorisation au niveau code, il faut s'assurer qu'aucune méthode dans la classe *class* ne possède déjà le nom *methodName*. Lors de l'application de cette refactorisation, la méthode est copiée avec son corps et sa signature (la visibilité, le type de retour et la liste des paramètres).

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent la duplication des éléments d'un programme orienté objet. Mais la refactorisation correspondante que nous proposons est : Copy Method (*method*, *methodName*, *class*).

5.8 Refactorisations primitives appliquées sur un paramètre

5.8.1 Créer un paramètre

- **Nom de la refactorisation au niveau du modèle** : Create Parameter (*method*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation sur le modèle, une instance de méta-paramètre est créée en donnant des noms et valeurs par défaut aux champs (le nom et le type de l'attribut) après s'être assuré que la méthode qui va comporter le paramètre créé doit exister, et aucun paramètre de même nom ne doit pas exister parmi la liste des paramètres de cette méthode.
  - pour la refactorisation du code, l'application de cette refactorisation permet de créer un paramètre conformement à la déclaration d'un paramètre dans le langage Java et en attribuant des noms et valeurs par défaut aux champs (le nom et le type l'attribut). Pour appliquer cette refactorisation, une vérification est nécessaire pour s'assurer que la méthode qui va avoir le nouveau paramètre doit exister dans la classe et qu'aucun paramètre de même nom n'existe parmi la liste des

paramètres de cette méthode.

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent de créer des éléments vides. Mais la refactorisation correspondante que nous proposons est : Create Parameter (*method*).

5.8.2 Renommer un paramètre

- **Nom de la refactorisation au niveau du modèle** : Rename Parameter (*parameter*, *newName*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer qu'aucune instance des méta-éléments attribut dans la classe ne possède pas déjà le nom *newName*. Lors de l'application de cette refactorisation au niveau du modèle, les paramètres de toutes les méthodes des classes utilisant l'attribut renommé sont mis à jour.
  - pour appliquer cette refactorisation sur le code, il faut s'assurer qu'aucun paramètre dans la classe ne porte le nom *newName*. Lors de l'application de cette refactorisation au niveau du code, tous les paramètres des appels de méthodes ainsi que les paramètres et le corps de toutes les méthodes utilisant l'attribut renommé sont mis à jour pour utilser le paramètre renommé avec son nouveau nom. Certains langages de programmation et de modélisation peuvent interdire un paramètre d'avoir le même nom qu'un attribut existant, mais ce n'est pas le cas pour UML et Java.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent le renommage. Mais la refactorisation correspondante que nous proposons est : Rename Parameter (*parameter*, *newName*).

5.8.3 Changer le type d'un paramètre

- **Nom de la refactorisation au niveau du modèle** : Set Type Parameter (*parameter*, *newType*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer que le nouveau type *newType* est accépté par le standard UML.
  - lors de l'application de cette refactorisation sur le code, une mise à jour pour le corps de la méthode qui comporte le paramètre *parameter* est nécessaire ainsi que pour tous les appels de méthodes pour s'assurer que cette méthode est invoquée avec son paramètre *parameter* ayant comme type *newType*, si possible afin d'éviter les erreurs de compilation. Cette vérification est inutile lors de l'application de cette refactorisation sur le modèle, car les paramètres d'une méthode sont utilisés dans son corps.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé une refactorisation qui permette de changer le type d'un paramètre. Mais la refactorisation correspondante que nous proposons est : Set Type Parameter (*parameter*, *newType*)

5.8.4 Supprimer un paramètre

- **Nom de la refactorisation au niveau du modèle** : Remove Parameter (*parameter*, *method*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer que le paramètre *parameter* n'est pas référencé dans le modèle.
  - pour appliquer cette refactorisation sur le code, une vérification est requise pour s'assurer que le paramètre *parameter* n'est pas référencé dans le corps de la méthode *method*. Lors de l'application de cette refactorisation sur le code, une mise à jour pour tous les appels de méthode est nécessaire pour s'assurer que cette

méthode est invoquée sans le paramètre supprimé. Cette opération est inutile lors de l'application de cette refactorisation sur le modèle, car les paramètres d'une méthode sont utilisés dans son corps.

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Remove Parameter (*parameter*, *method*) [Fowler, 1999].

5.8.5 Copier un paramètre

– **Nom de la refactorisation au niveau du modèle** : Copy parameter (*parameter*, *parameterName*, *method*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

  – pour appliquer cette refactorisation au niveau du modèle, il faut s'assurer qu'aucun paramètre dans la méthode *method* ne possède déjà le nom *parameterName*. Lors de l'application de cette refactorisation, le paramètre est copié avec son type.

  – pour appliquer cette refactorisation au niveau code, il faut s'assurer qu'aucun paramètre de la méthode *method* ne possède pas déjà le nom *parameterName* et aucune variable locale dans le corps de cette méthode ne possède pas déjà le nom *parameterName*. Cette vérification est inutile lors d'une refactorisation du modèle, car les méthodes ne possèdent pas de corps. Lors de l'application de cette refactorisation, le paramètre est copié avec son type et une mise à jour pour tous les appels de méthode est nécessaire pour invoquer la méthode *method* avec le paramètre *parameterName*.

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent la duplication des éléments d'un programme orienté objet. Mais la refactorisation correspondante que nous proposons est : Copy parameter (*parameter*, *parameterName*, *method*).

5.9 Refactorisations primitives appliquées sur une association

5.9.1 Créer une association

- **Nom de la refactorisation au niveau du modèle** : Create Association (*SourceClass*, *TargetClass*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation sur le modèle, une instance de méta-association est créée en donnant des noms et valeurs par défaut aux champs (le nom de l'association, la multiplicité de la classe source et la multiplicité de la classe cible) selon la spécification du langage de modélisation UML, après s'être assuré que la classe source et la classe cible existent dans le modèle et qu'aucune association de même nom n'existe. La création d'une association au niveau du modèle signifie la création d'un lien sémantique entre deux classes.
  - la création d'une association au niveau code se traduit par la création d'un ou de deux attributs dépendemment de la navigabilité associée aux rôles désignant les extrémités des association. La navigabilité est une propriété qui indique qu'il est possible de naviguer de façon unidirectionnelle ou bidirectionnelle entre les objets de la classe source et ceux de la classe cible. Généralement la navigablté modifie la visibilité de l'attribut. La création d'une association unidirectionnelle au niveau code se traduit donc par la création d'un attribut dans la classe source. Dans le cas d'une association bidirectionnelle, ceci se traduit par la création d'un attribut dans la classe source et d'un autre dans la classe cible de type de la classe cible et des méthodes d'accés.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent de créer des éléments vides et les associations sont implicites au niveau code. Mais la refactorisation correspondante que nous proposons est : Create Association (*SourceClass*, *TargetClass*).

5.9.2 Renommer une association

- **Nom de la refactorisation au niveau du modèle** : Rename Association (*association*, *newName*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est requise pour s'assurer qu'aucune association dans la classe ne possède déjà le nom *newName*.
  - l'application de la refactorisation du renommage d'une association au niveau du code n'affecte pas ce dernier puisque le nom de l'association ne correspond à rien au niveau code.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : aucune refactorisation du code ne correspond à cette refactorisation du modèle puisque le nom de l'association dans le modèle de classes ne correspond à rien au niveau code.

5.9.3 Changer la multiplicité de la classe cible

- **Nom de la refactorisation au niveau du modèle** : Set Multiplicity Target Class (*associationName*, *newMultiplicity*)
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - l'application de cette refactorisation sur le modèle permet seulement de changer les valeurs de la multiplicité de la classe cible. Ce changement est trivial sur le diagramme de classes mais impacte fortement les diagrammes d'objet et intéraction. La multiplicité indique généralement une contrainte du domaine que nous voulons pouvoir contrôler dans le code du programme.
  - lors de l'application de cette refactorisation au niveau du code, des mises à jour sont nécessaires pour modifier l'attribut de référence et les méthodes de la classe source. Jackson [Jackson et Waingold, 2001] se sont intéressés à cible de l'association unidirectionnelle, parce qu'elle est plus simple à gérer. Ils ne traitent pas la multiplicité de la classe source. Pour étudier l'impact du changement de la mu-

tiplicité de la classe source et de la classe cible d'une association bidirectionnelle sur le code source d'un programme, nous suivons le même principe que Jackson en divisant cette association en deux associations unidirectionnelles. Nous étudions donc en premier l'association de la classe A à la classe B représentée par le couple {la multiplicité de la classe source, la multiplicité de la classe cible}. Si la multiplicité change de (1,*) (c'est à dire, pour chaque instance de A, plusieurs instances de B existent) à (1,1) (c'est à dire, pour chaque instance de A, une seule instance de B existe), il faut mettre à jour l'attribut référence (l'attribut référençant une instance de la classe cible) pour modifier sa déclaration à privée statique et interdire aux méthodes de la classe A de retourner un objet de la classe B. On est donc assuré qu'une seule instance de la classe B a été créée dans la classe A. Si la multiplicité change de (1,1) à (1,*), dans ce cas, une mise à jour pour l'attribut de référence est nécessaire pour modifier sa visibilité à privée sans statique et ainsi permettre à plusieurs instances de la classe B d'être créée dans la classe A.

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : les refactorisations apppliquées sur l'associaton et proposées par Fowler sont : changer une association bidirectionnelle en une association unidirectionnelle et changer une association unidirectionnelle en une association bidirectionnelle. Mais la refactorisation correspondante que nous proposons est : Set Multiplicity Target Class (*TargetClass*, *newMultiplicity*)

5.9.4 Changer la multiplicité de la classe source

- **Nom de la refactorisation au niveau du modèle** : Set Multiplicity Source Class (*associationName*, *newMultiplicity*)
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - l'application de cette refactorisation sur le modèle permet seulement de changer les valeurs de la multiplicité de la classe source.
  - lors de l'application de cette refactorisation au niveau code, une mise à jour pour les attributs et les méthodes de la classe cible est nécessaire pour les modifier

afin de refléter dans le programme le changement de la valeur de la mutiplicité qui indique une contrainte du domaine, comme dans la sous section précédente : 5.9.3

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : la refactorisation correspondante que nous proposons est : Set Multiplicity Source Class (*SourceClass*,*newMultiplicity*)

5.9.5 Supprimer une association

- **Nom de la refactorisation au niveau du modèle** : Remove Association (*association*, *SourceClass*, *TargetClass*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - lors de l'application de cette refactorisation au niveau du modèle, la supression de la relation d'association existante entre la classe source et la classe cible signifie la supression du lien sémantique qui existait entre ces deux classes. C'est au concepteur de décider si cette association est utilisée ou non.
  - pour appliquer cette refactorisation sur le code, des vérifications dans la classe source et cible sont nécessaires pour s'assurer que les attributs de références ne sont pas utilisés dans leurs classes de définitions. L'application de cette refactorisation au niveau code se traduit par la suppression de l'attribut déclaré dans la classe source et référençant l'instance de la classe cible. Une vérification dans toutes les méthodes de la classe de définition de l'attribut de référence est aussi nécessaire pour supprimer les méthodes accesseurs et toutes les affectations à cet attribut. De même, l'attribut de la classe cible référençant l'instance de la classe source est aussi supprimé ainsi que les méthodes accesseurs et toutes les affectations à cet attribut. Sinon, leur présence empêche l'application de cette refactorisation.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui

permettent la suppression. Mais la refactorisation correspondante que nous proposons est : Remove Association (*SourceClass*, *TargetClass*).

5.9.6 Copier une association

- **Nom de la refactorisation au niveau du modèle** : Copy Association (*association*, *associationName*, *SourceClass*, *TargetClass*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation au niveau du modèle, il faut s'assurer qu'aucune association dans la classe ne possède déjà une association avec comme nom *associationName*. Lors de l'application de cette refactorisation, les multiplicités de la classe source et de la classe cible sont aussi copiés.
  - l'application de cette refactorisation au niveau du code est beaucoup plus complexe de celle appliquée au niveau modèle, car elle nécessite plusieurs opérations. Il faut d'abord copier l'attribut de référence dans la classe source, après avoir vérifié que son nom n'existe pas déjà dans cette dernière, et lui affecter l'instance de la nouvelle classe cible. Ensuite, il faut créer des méthodes accesseurs dans la classe source. Puisque, nous traitons le cas général d'une association bidirectionnelle, il faut aussi copier l'attribut de référence dans la classe cible, après avoir vérifié que son nom n'existe pas déjà dans cette dernière, lui affecter l'instance de la nouvelle classe source et ensuite créer des méthodes accesseurs à la classe source.
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé des refactorisations qui permettent la duplication des éléments d'un programme orienté objet. Mais la refactorisation correspondante que nous proposons est : Copy Association (*SourceClass*, *TargetClass*).

5.10 Refactorisations composites appliquées sur un paquetage

5.10.1 Ajouter un paquetage

- **Nom de la refactorisation au niveau du modèle** : Add Package (*packageName*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est nécessaire pour s'assurer qu'aucun paquetage ne doit avoir le même nom *packageName* dans le modèle. L'application de cette refactorisation permet d'ajouter un paquetage avec le nom *packageName* au modèle UML en créant un paquetage vide et en lui attribuant le nouveau nom *packageName* choisi par l'utilisateur et qui ne doit pas exister déjà dans le modèle. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Create Package** (*meta-model*) ;
    2. **Rename Package** (*package*, *newName*) ;
  - pour appliquer cette refactorisation sur le code, une vérification est nécessaire pour s'assurer qu'aucun paquetage appartenant au programme ne doit avoir déjà le même nom *packageName*. L'application de cette refactorisation permet d'ajouter un paquetage avec le nom *packageName* au programme en créant un paquetage vide et en lui attribuant un nouveau nom choisi par l'utilisateur et qui ne doit pas exister déjà dans le code. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Create Package** (*model*) ;
    2. **Rename Package** (*package*, *newName*) ;
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent l'ajout. Mais la refactorisation correspondante que nous proposons est :

Add Package (*packageName*).

5.10.2 Déplacer un paquetage

- **Nom de la refactorisation au niveau du modèle** : Move Package (*package*, *packageName*, *model*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer la refactorisation de déplacement du paquetage sur le modèle, des vérifications sont nécessaires pour s'assurer que ce paquetage n'a pas des relations de dépendance ou de généralisation avec les autres paquetages du paquetage englobant ou d'autres paquetages du modèle. Le nom *packageName* ne doit pas exister déjà dans le paquetage englobant de destination (où l'on veut déplacer le paquetage). L'application de cette refactorisation permet de déplacer le paquetage portant le nom *packageName* avec tous ses éléments (les paquetages emboîtés, les classes, les interfaces). Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Copy Package** (*package*, *packageName*, *model*) ;
    2. **Remove Package** (*package*) ;
  - pour appliquer cette refactorisation sur le code, des vérifications pour tous les fichiers du programme sont nécessaires pour s'assurer que ces fichiers n'utilisent pas des classes et des interfaces de ce paquetage, et le nom *packageName* ne doit pas exister déjà dans le programme. L'application de cette refactorisation permet de déplacer le paquetage portant le nom *packageName* avec tous ses éléments (les classes et les interfaces). Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Copy Package** (*package*, *packageName*) ;
    2. **Remove Package** (*package*) ;
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour

tous les éléments constituant un programme orienté objet, des refactorisations qui permettent le déplacement. Mais la refactorisation correspondante que nous proposons est : Move Package (*package*, *packageName*).

5.11 Refactorisations composites appliquées sur une interface

5.11.1 Ajouter une interface

- **Nom de la refactorisation au niveau du modèle** : Add Interface (*interfaceName*, *package*, *subclasses*)
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est nécessaire pour s'assurer qu'aucune interface ne doit avoir le même nom *interfaceName* dans le paquetage *package*. L'application de cette refactorisation permet d'ajouter une interface avec le nom *interfaceName* au paquetage *package* en créant une interface vide et en lui attribuant le nouveau nom *interfaceName* choisi par l'utilisateur. Les sous classes *subclasses* sont les classes implantant l'interface ajoutée et ainsi des relations de réalisation sont ajoutées. Dans le cas où les classes sont des interfaces, une relation de généralisation est ajoutée. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Create Interface** (*package*) ;
    2. **Rename Interface** (*interface*, *newName*) ;
  - pour appliquer cette refactorisation sur le code, une vérification est nécessaire pour s'assurer qu'aucune interface appartenant au programme n'a déjà le même nom *interfaceName*. L'application de cette refactorisation permet d'ajouter une interface avec le nom *interfaceName* au programme en créant une interface vide et en lui attribuant le nouveau nom *interfaceName* choisi par l'utilisateur et qui ne doit pas exister déjà dans le programme. L'ajout d'une nouvelle interface au niveau du code signifie l'ajout d'un fichier qui comporte la déclaration de cette interface et la mise à jour pour tous les fichiers qui comportent la déclaration des classes implantant cette interface pour ajouter à leur déclarations, selon le lan-

gage Java, le mot clé qui définie l'implantation de la nouvelle interface. Si les sous classes sont des interfaces, dans ce cas ces dernières étendent l'interface ajoutée. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :

1. **Create Interface** (*package*) ;
2. **Rename Interface** (*interface*, *newName*) ;

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent l'ajout. Mais la refactorisation correspondante que nous proposons est : Add Interface (*interfaceName*, *package*, *subclasses*).

5.11.2 Déplacer une interface

– **Nom de la refactorisation au niveau du modèle** : Move Interface (*interface*, *interfaceName*, *package*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

– pour appliquer cette refactorisation sur le modèle, des vérifications sont nécessaires pour s'assurer que l'interface *interface* n'a pas de lien avec les autres classes ou les autres interfaces du paquetage de définition (où cette interface est définie). Et le nom *interfaceName* ne doit pas exister déjà dans le paquetage *package* de destination (où l'on veut déplacer cette interface). L'application de cette refactorisation permet de déplacer l'interface portant le nom *interfaceName* au paquetage *package*. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :

1. **Copy interface** (*interface*, *interfaceName*, *package*) ;
2. **Remove interface** (*interface*) ;

– pour appliquer cette refactorisation sur le code, des vérifications pour tous les fichiers du programme sont nécessaires pour s'assurer que ces fichiers n'utilisent

pas l'interface *interface*, et le nom *interfaceName* ne doit pas exister déjà dans le paquetage *package*. L'application de cette refactorisation permet de déplacer l'interface portant le nom *interfaceName* au paquetage *package*. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :

1. **Copy interface** (*interface*, *interfaceName*, *package*) ;
2. **Remove interface** (*interface*) ;

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet, des refactorisations qui permettent le déplacement. Mais la refactorisation correspondante que nous proposons est : Move Interface (*interface*, *interfaceName*, *package*).

5.12 Refactorisations composites appliquées sur une classe

5.12.1 Ajouter une classe

– **Nom de la refactorisation au niveau du modèle** : Add Class (*className*, *package*, *superclasses*, *subclasses*)

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

– pour appliquer cette refactorisation sur le modèle, une vérification est nécessaire pour s'assurer qu'aucune classe ne doit avoir le même nom *className* dans le paquetage *package*. L'application de cette refactorisation permet d'ajouter une classe avec le nom *className* au paquetage *package* en créant une classe vide et en lui attribuant le nouveau nom *classeName* choisi par l'utilisateur et qui ne doit pas exister déjà dans le paquetage. Des relations de généralisation sont ajoutées pour rendre la nouvelle classe une superclasse pour les sous classes *subclasses* et une sous classe pour les superclasses *superclasses*. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Create Class** (*package*) ;

2. **Rename Class** (*class*, *newName*) ;
3. **Tant que** (attribute n'est pas un ensemble vide) **faire**
   – **Add Attribute** (*attributeName*, *class*) ;
   Fin tant que
4. **Tant que** (methode n'est pas un ensemble vide) **faire**
   – **Add Method** (*methodName*, *class*) ;
   Fin tant que

– pour appliquer cette refactorisation sur le code, une vérification est nécessaire pour s'assurer qu'aucune classe appartenant au programme ne doit avoir déjà le même nom *className*. L'application de cette refactorisation sur le code permet d'ajouter un fichier comportant la déclaration de la nouvelle classe avec comme nom, le nom *className* choisi par l'utilisateur et comme superclasses, les superclasses *superclasses*. Une mise à jour pour tous les fichiers qui contiennent la déclaration des sous classes est donc nécessaire pour rendre ces dernières, selon la spécification du langage Java, les classes qui étendent la nouvelle classe. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Create Class** (*package*) ;
2. **Rename Class** (*class*, *newName*) ;
3. **Tant que** (attribute n'est pas un ensemble vide) **faire**
   – **Add Attribute** (*attributeName*, *class*) ;
   Fin tant que
4. **Tant que** (methode n'est pas un ensemble vide) **faire**
   – **Add Method** (*methodName*, *class*) ;
   Fin tant que

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet des refactorisations qui

permettent l'ajout. Mais la refactorisation correspondante que nous proposons est : Add Class (*className*, *package*, *superclasses*, *subclasses*).

5.12.2 Déplacer une classe

- **Nom de la refactorisation au niveau du modèle** : Move Class (*class*, *className*, *package*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, des vérifications sont nécessaires pour s'assurer que la classe *class* n'a pas de lien avec les autres classes du paquetage de définition. Et le nom *className* ne doit pas exister déjà dans le paquetage *package* de destination (où l'on veut déplacer cette classe). L'application de cette refactorisation permet de déplacer la classe portant le nom *className* dans le paquetage *package* avec tous ses éléments (les attributs et les méthodes). Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Copy Class** (*class*, *className*, *package*) ;
    2. **Remove Class** (*class*) ;
  - pour appliquer cette refactorisation sur le code, des vérifications pour tous les fichiers du programme sont nécessaires pour s'assurer que ces fichiers n'utilisent pas la classe *class*, et le nom *className* ne doit pas exister déjà dans le paquetage *package*. L'application de cette refactorisation permet de déplacer la classe portant le nom *className* au paquetage *package* avec ses éléments (les attributs, les signatures et le corps des méthodes). Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Copy Class** (*class*, *className*, *package*) ;
    2. **Remove Class** (*class*) ;
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Move Class[12]

[12]http ://www.refactoring.com/catalog/moveClass.html

5.13 Refactorisations composites appliquées sur un attribut

5.13.1 Ajouter un attribut

- **Nom de la refactorisation au niveau du modèle** : Add Attribute (*attributeName*, *newVisibility*, *newType*, *class*)
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est nécessaire pour s'assurer qu'aucun attribut n'a le même nom *attributeName* dans la classe *class*. L'application de cette refactorisation permet d'ajouter un attribut avec le nom *attributeName* à la classe *class* en créant un attribut vide et en lui attribuant le nom *attributeName*, la visibilité *newVisibility* et le type *newType* choisis par l'utilisateur. Le nom choisi *attributeName* ne doit pas exister déjà dans la classe et la visibilité *newVisibility* et le type *newType* doivent respectivement appartenir à l'ensemble des visibilités et des types autorisés. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Create Attribute** (*class*) ;
    2. **Rename Attribute** (*attribute*, *newName*) ;
    3. **Set Visibility Attribute** (*attribute*, *newVisibility*) ;
    4. **Set Type Attribute** (*attribute*, *newType*) ;
  - pour appliquer cette refactorisation sur le code, une vérification est nécessaire pour s'assurer qu'aucun attribut ne doit avoir déjà le même nom *attributeName* dans la classe *class*. L'application de cette refactorisation permet d'ajouter un attribut avec le nom *attributeName* à la classe *classs* en créant un attribut vide et en lui attribuant le nom *attributeName*, la visibilité *newVisibility* et le type *newType* choisis par l'utilisateur. Le nom choisi ne doit pas exister déjà dans la classe et la visibilité et le type doivent respectivement appartenir à l'ensemble des visibilités et des types autorisés. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Create Attribute** (*class*) ;

2. **Rename Attribute** (*attribute*, *newName*) ;
3. **Set Visibility Attribute** (*attribute*, *newVisibility*) ;
4. **Set Type Attribute** (*attribute*, *newType*) ;

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet des refactorisations qui permettent l'ajout. Mais la refactorisation correspondante que nous proposons est : Add Attribute (*attributeName*, *newVisibility*, *newType*, *class*).

5.13.2 Déplacer un attribut

– **Nom de la refactorisation au niveau du modèle** : Move Attribute (*attribute*, *attributeName*, *class*).
– **Comparaison entre refactorisation du modèle et refactorisation du code** :
  – pour appliquer cette refactorisation sur le modèle, l'attribut *attribute* ne doit pas être utilisé dans sa classe de définition ainsi que dans les autres classes de sa hiérarchie, et le nom *attributeName* de cet attribut ne doit pas exister dans la classe de destination *class* (où l'attribut va être déplacé). L'application de cette refactorisation permet de déplacer l'attribut portant le nom *attributeName* à la classe *class*. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Copy Attribute** (*attribute*, *attributeName*, *class*) ;
    2. **Remove Attribute** (*attribute*) ;
  – pour appliquer cette refactorisation sur le code, des vérifications pour le fichier de la classe où l'attribut *attribute* est défini et pour tous les fichiers de la hiérarchie d'héritage sont nécessaires pour s'assurer que ces fichiers n'utilisent pas cet attribut. De plus, le nom *attributeName* ne doit pas exister déjà dans la classe *class* qui va comporter l'attribut déplacé. L'application de cette refactorisation permet de déplacer l'attribut portant le nom *attributeName* à la classe *class*. Ceci peut

être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :

1. **Copy Attribute** (*attribute*, *attributeName*, *class*) ;
2. **Remove Attribute** (*attribute*) ;

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Move Field[13]

5.13.3 Monter un attribut

– **Nom de la refactorisation au niveau du modèle** : Pull Up Attribute (*attribute*, *superclass*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

– pour appliquer cette refactorisation sur le modèle, toutes les sous classes de *superclass* doivent avoir le même attribut *attribute* avec le même nom et le même type. De plus, le nom de l'attribut ne doit pas exister déjà dans la superclasse *superclass*. L'application de cette refactorisation permet de déplacer l'attribut *attribute* qui est identique dans les sous classe dans une superclasse commune *superclass*. Si la visibilité de l'attribut déplacé est privée, il faut changer sa visibilité à protégée pour permettre aux sous classes de d'y accéder. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Copy Attribute** (*attribute*, *attributeName*, *superclass*) ;
2. **Si** (la visibilité de *attribute* est *private*) **Alors**
   – **Set Visibility Attribute** (*attribute*, *protected*) ;
   Fin Si
3. **Tant que** (subclasses n'est pas un ensemble vide) **faire**
   – **Remove attribute** (*attribute*) ;
   Fin tant que

– pour appliquer cette refactorisation sur le code, une vérification pour tous les fichiers des sous classes est nécessaire pour s'assurer que l'attribut *attribute* est

[13]http ://www.refactoring.com/catalog/moveField.html

déclaré avec le même nom et le même type dans toutes les sous classes. De plus, le nom de l'attribut *attribute* ne doit pas exister déjà dans la définition de *superclass*. L'application de cette refactorisation permet de déplacer l'attribut *attribute* qui est identique dans les sous classe en une superclasse commune *superclass*. Si la visibilité de l'attribut déplacé est privée, il faut changer sa visibilité à protégée pour permettre aux sous classes de l'utiliser. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Copy Attribute** (*attribute*, *attributeName*, *superclass*) ;
2. **Si** (la visibilité de *attribute* est *private*) **Alors**
   – **Set Visibility Attribute** (*attribute*, *protected*) ;
   Fin Si
3. **Tant que** (subclasses n'est pas un ensemble vide) **faire**
   – **Remove attribute** (*attribute*) ;
   Fin tant que

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Pull Up Field (*attribute*, *superclass*) [Fowler, 1999].

5.13.4 Descendre un attribut

– **Nom de la refactorisation au niveau du modèle** : Push Down Attribute (*attribute*, *subclasses*).
– **Comparaison entre refactorisation du modèle et refactorisation du code** :
  – pour appliquer cette refactorisation sur le modèle, l'attribut *attribute* ne doit pas être utilisé dans la superclasse (où il est défini). De plus, le nom de l'attribut *attribute* ne doit pas déjà exister dans les sous-classes *subclasses* qui l'utilisent. L'application de cette refactorisation permet de déplacer l'attribut *attribute* de la superclasse aux sous-classes *subclasses*. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
    1. **Tant que** (*subclasses* n'est pas un ensemble vide) **faire**
       – **Copy Attribute** (*attribute*, *attributeName*, *subclasses*) ;

Fin tant que

2. **Remove attribute** (*attribute*) ;

– pour appliquer cette refactorisation sur le code, une vérification dans la définition de la superclasse est nécessaire pour s'assurer que l'attribut *attribute* n'est pas utilisé. De plus, le nom de l'attribut *attribute* ne doit pas déjà exister dans une sous-classe des *subclasses*. L'application de cette refactorisation permet de déplacer l'attribut *attribute* de la superclasse aux sous-classes *subclasses*. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Tant que** (*subclasses* n'est pas un ensemble vide) **faire**
   – **Copy Attribute** (*attribute*, *attributeName*, *subclasses*) ;
   Fin tant que
2. **Remove attribute** (*attribute*) ;

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Push Down Field (*attribute*, *subclasses*) [Fowler, 1999].

5.13.5 Encapsuler un attribut

– **Nom de la refactorisation au niveau du modèle** : Encapsulate Attribute (*attribute*, *class*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

– pour appliquer cette refactorisation sur le modèle, l'attribut *attribute* doit être défini avec une visibilité publique dans la classe *class*. L'application de cette refactorisation permet de modifier la visibilité de l'attribut *attribute* de publique à privée et d'ajouter la signature des méthodes accesseurs à la classe *class* pour permettre l'accès à cet attribut. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Set Visibility Attribute**(*attribute*, *private*) ;
2. **Add Method** (*method*, *get*, *class*) ;
3. **Add Method** (*method*, *set*, *class*) ;

- pour appliquer cette refactorisation sur le code, l'attribut *attribute* doit être déclaré avec une visibilité publique dans le fichier de la classe *class*. L'application de cette refactorisation permet de modifier la visibilité de l'attribut *attribute* de publique à privée et d'ajouter les méthodes accesseurs (la signature et le corps) à la classe *class* pour permettre l'accès à cet attribut. Une mise à jour pour des références externes à l'attribut *attribute* est nécessaire pour utiliser les accesseurs de méthode au lieu d'accéder au champ directement, ce qui nécessite une vérification pour le appels et le corps des méthodes. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
  1. **Set Visibility Attribute**(*attribute*, *private*) ;
  2. **Add Method** (*method*, *get*, *class*) ;
  3. **Add Method** (*method*, *set*, *class*) ;
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Encapsulate Field (*attribute*, *class*) [Fowler, 1999].

5.13.6 Remplacer un attribut par des sous classes

- **Nom de la refactorisation au niveau du modèle** : Replace Attribute with Subclasses (*class*, *attributes*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, les attributs de la classe *class* doivent avoir un type numérique constant ou une énumération et toutes les valeurs des types des attributs ne doivent pas changer après la création d'un objet. De plus, les noms des attributs ne doivent pas exister déjà dans l'hiérarchie des classes. L'application de cette refactorisation permet d'ajouter pour chaque attribut *attribute* défini dans la classe *class*, avec un type numérique constant ou une énumération, une classe avec comme nom, le nom de l'attribut *attribut* et comme superclasse, la classe *class*. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
    1. **Tant que** (attributes n'est pas un ensemble vide) **faire**

– **Add Class** (*attributeName*, *package*, *class*, *subclasses*) ;

Fin tant que

2. **Tant que** (attributes n'est pas un ensemble vide) **faire**

– **Remove Attribute** (*attribute*, *class*) ;

Fin tant que

– pour appliquer cette refactorisation sur le code, les attributs de la classe *class* doivent avoir un type numérique constant ou une énumération et toutes les valeurs des types des attributs ne doivent pas changer après la création d'un objet. De plus, les noms des attributs ne doivent pas exister déjà dans l'hiérarchie des classes de la classe *class*. L'application de cette refactorisation permet d'ajouter pour chaque attribut *attribute* défini dans la classe *class* avec un type numérique constant (déclaré statique et final) ou une énumération une classe avec comme nom, le nom de l'attribut *attribut* et comme superclasse, la classe *class*. Une vérification est effectuée dans la classe *class* pour déplacer les méthodes (signature et corps) et les attributs, qui se réfèrent seulement à un type particulier de la classe *class*, à la classe appropriée ajoutée en appliquant les refactorisations composites *Push Down Method* et *Push Down Attribute*. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Tant que** (attributes n'est pas un ensemble vide) **faire**

– **Add Class** (*attributeName*, *package*, *class*, *subclasses*) ;

Fin tant que

2. **Tant que** (attributes n'est pas un ensemble vide) **faire**

– **Remove Attribute** (*attribute*, *class*) ;

Fin tant que

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Replace Type Code with Subclasses (*class*, *attributes*) [Fowler, 1999].

5.14 Refactorisations composites appliquées sur une méthode

5.14.1 Ajouter une méthode

- **Nom de la refactorisation au niveau du modèle** : Add Method (*methodName*, *newVisibility*, *newType*, *class*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est nécessaire pour s'assurer qu'aucune méthode définie dans la classe *class* ne doit avoir une signature identique à celle de la méthode dont le nom est *methodName*. L'application de cette refactorisation permet d'ajouter à la classe *class*, une méthode avec le nom *methodName* en créant une méthode vide et en lui attribuant le nom *methodName*, la visibilité *newVisibility* et le type de retour *newType* choisis par l'utilisateur ainsi qu'en ajoutant ses paramètres. La visibilité *newVisibility* et le type de retour *newType* doivent respectivement appartenir à l'ensemble des visibilités et des types autorisés. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
    1. **Create Method** (*class*) ;
    2. **Rename Method** (*method*, *newName*) ;
    3. **Set Visibility Method** (*method*, *newVisibility*) ;
    4. **Set Return Type Method** (*method*, *newType*) ;
    5. **Tant que** (parameter n'est pas un ensemble vide) **faire**
       - **Add Parameter** (*parameterName*, *method*) ;

       Fin tant que
  - pour appliquer cette refactorisation sur le code, une vérification est nécessaire pour s'assurer qu'aucune méthode ne doit avoir une signature identique à celle de la méthode dont le nom est *methodName*. L'application de cette refactorisation permet d'ajouter une méthode avec le nom *methodName* en créant une méthode vide et en lui attribuant le nom *methodName*, la visibilité *newVisibility* et le type de retour *newType* choisis par l'utilisateur ainsi qu'en ajoutant ses paramètres

et son corps. La visibilité *newVisibility* et le type de retour *newType* doivent respectivement appartenir à l'ensemble des visibilités et des types autorisés. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Create Method** (*class*) ;
2. **Rename Method** (*method*, *newName*) ;
3. **Set Visibility Method** (*method*, *newVisibility*) ;
4. **Set Return Type Method** (*method*, *newType*) ;
5. **Tant que** (parameter n'est pas un ensemble vide) **faire**
   - **Add Parameter** (*parameterName*, *method*) ;

   Fin tant que

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet des refactorisations qui permettent l'ajout. Mais la refactorisation correspondante que nous proposons est : Add Method (*methodName*, *newVisibility*, *newType*, *class*).

5.14.2 Déplacer une méthode

– **Nom de la refactorisation au niveau du modèle** : Move Method (*method*, *methodName*, *class*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

  – pour appliquer cette refactorisation sur le modèle, des vérifications sont nécessaires pour s'assurer que la méthode *method* n'est pas utilisée dans sa classe et dans ses sous classe, et la signature de cette méthode ne doit pas exister déjà dans la classe de destination *class*. L'application de cette refactorisation permet de déplacer la méthode portant le nom *methodName* à la class *class*. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :

    1. **Copy Method** (*method*, *methodName*, *class*) ;

2. **Remove Method** (*method*) ;

– pour appliquer cette refactorisation sur le code, une vérification pour les appels de méthodes dans tous les fichiers qui comportent la déclaration de la classe de définition de la méthode *method* et des sous classes est nécessaire pour s'assurer que cette méthode n'est pas appelée. De plus, la signature de cette méthode ne doit pas exister déjà dans la classe de destination *class*. L'application de cette refactorisation permet de déplacer la méthode (la signature et le corps de la méthode) avec le nom *methodName* à la class *class*. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :

1. **Copy Method** (*method*, *methodName*, *class*) ;
2. **Remove Method** (*method*) ;

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Move Method (*method*, *methodName*, *class*) [Fowler, 1999]

5.14.3 Monter une méthode

– **Nom de la refactorisation au niveau du modèle** : Pull Up Method (*method*, *superclass*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

– pour appliquer cette refactorisation sur le modèle, la méthode *method* doit être définie avec une signature identique dans toutes les sous classes et elle ne doit pas utiliser les attributs de sa classe de définition. De plus, la signature de cette méthode ne doit pas exister déjà dans la superclasse *superclass*. L'application de cette refactorisation permet de déplacer la méthode *method* ayant une signature identique dans toutes les sous classe en une superclasse commune *superclass*. Si la visibilité de la méthode déplacée est privée, il faut changer sa visibilité à protégée pour permettre aux sous classes de l'utiliser. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :

1. **Copy Method** (*method*, *methodName*, *superclass*) ;
2. **Si** (la visibilité de *method* est *private*) **Alors**

– **Set Visibility Method** (*method*, *protected*) ;

Fin Si

3. **Tant que** (subclasses n'est pas un ensemble vide) **faire**

– **Remove Method** (*method*) ;

Fin tant que

– pour appliquer cette refactorisation sur le code, une vérification pour toutes les sous classes est nécessaire pour s'assurer que la méthode *method* a une signature identique dans toutes les sous classes et qu'elle n'utilise pas utiliser dans son corps les attributs et les méthodes de sa classe de définition. De plus, la signature de cette méthode ne doit pas exister déjà dans la superclasse *superclass*. L'application de cette refactorisation permet de déplacer la méthode *method* ayant une signature identique dans toutes les sous classe en une superclasse commune *superclass*. Si la visibilité de la méthode déplacée est privée, il faut changer sa visibilité à protégée pour permettre aux sous classes de l'utiliser. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :

1. **Copy Method** (*method*, *methodName*, *superclass*) ;
2. **Si** (la visibilité de *method* est *private*) **Alors**

– **Set Visibility Method** (*method*, *protected*) ;

Fin Si

3. **Tant que** (*subclasses* n'est pas un ensemble vide) **faire**

– **Remove Method** (*method*) ;

Fin tant que

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Pull Up Method (*method*, *superclass*) [Fowler, 1999].

5.14.4 Descendre une méthode

– **Nom de la refactorisation au niveau du modèle** : Push Down Method (*method*, *subclasses*).

- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, la méthode *method* ne doit pas être utilisé dans la superclasse. De plus, la signature de la méthode *method* ne doit pas exister déjà dans les sous-classes *subclasses* qui utilisent cette méthode. L'application de cette refactorisation permet de déplacer la méthode *method* de la superclasse aux sous-classes *subclasses*. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
    1. **Tant que** (*subclasses* n'est pas un ensemble vide) **faire**
       - **Copy Method** (*method*, *methodName*, *subclass*) ;

       Fin tant que
    2. **Remove Method** (*method*) ;
  - pour appliquer cette refactorisation sur le code, une vérification pour tous les appels de méthode dans la superclasse est nécessaire pour s'assurer que la méthode *method* n'est pas appelée. De plus, la signature de la méthode *method* ne doit pas exister déjà dans les fichiers des sous-classes *subclasses* qui utilisent cette méthode. L'application de cette refactorisation permet de déplacer la méthode *method* de la superclasse aux sous-classes *subclasses*. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
    1. **Tant que** (*subclasses* n'est pas un ensemble vide) **faire**
       - **Copy Method** (*method*, *methodName*, *subclass*) ;

       Fin tant que
    2. **Remove Method** (*method*) ;
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Push Down Method (*method*, *subclasses*) [Fowler, 1999].

5.15 Refactorisations composites appliquées sur un paramètre

5.15.1 Ajouter un paramètre

- **Nom de la refactorisation au niveau du modèle** : Add Parameter (*parameterName*, *newType*, *method*).

- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est nécessaire pour s'assurer que le nom *parameterName* ne doit pas exister déjà dans la liste des paramètres de la méthode *method*. L'application de cette refactorisation permet d'ajouter un paramètre avec le nom *parameterName* à la méthode *method* en créant un paramètre vide et en lui attribuant le nom *parameterName* et le type *newType* choisis par l'utilisateur. Le nom choisi ne doit pas exister déjà dans la liste des paramètres de la méthode *method* et le type doit appartenir à l'ensemble des types valides. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
    1. **Create Parameter** (*method*) ;
    2. **Rename Parameter** (*parameter*, *newName*) ;
    3. **Set Type Parameter** (*parameter*, *newType*) ;
  - pour appliquer cette refactorisation sur le code, une vérification est nécessaire pour s'assurer que le nom *parameterName* ne doit pas exister déjà dans la liste des paramètres et dans le corps de la méthode *method*. L'application de cette refactorisation permet d'ajouter un paramètre avec le nom *parameterName* à la méthode *method* en créant un paramètre vide et en lui attribuant le nom *parameterName* et le type *newType* choisis par l'utilisateur. Le nom choisi ne doit pas déjà exister dans la liste des paramètres et comme variable locale dans la méthode *method* et le type *newType* doit appartenir à l'ensemble des types valides. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
    1. **Create Parameter** (*method*) ;
    2. **Rename Parameter** (*parameter*, *newName*) ;
    3. **Set Type Parameter** (*parameter*, *newType*) ;
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : Add Parameter (*parameterName*, *newType*, *method*) [Fowler, 1999].

5.15.2 Déplacer un paramètre

- **Nom de la refactorisation au niveau du modèle** : Move Parameter (*parameter*, *parameterName*, *method*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, des vérifications sont nécessaires pour s'assurer que le paramètre *parameter* n'est pas utilisé dans la méthode (qui le définie parmi la liste de ses paramètres). De plus, le nom *parameterName* ne doit pas exister déjà dans la liste des paramètre de la méthode *method* (la méthode qui va contenir le paramètre déplacé). L'application de cette refactorisation permet de déplacer le paramètre portant le nom *parameterName* dans les paramètres de la méthode *method*. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Copy parameter** (*parameter*, *parameterName*, *method*) ;
    2. **Remove parameter** (*parameter*) ;
  - pour appliquer cette refactorisation sur le code, des vérifications sont nécessaires pour s'assurer que le paramètre *parameter* n'est pas utilisé dans le corps de la méthode (qui le définie parmi la liste de ses paramètres). De plus, le nom *parameterName* ne doit pas déjà exister dans la liste des paramètre et dans le corps de la méthode *method* (la méthode qui va contenir le paramètre déplacé). L'application de cette refactorisation permet de déplacer le paramètre portant le nom *parameterName* dans les paramètres de la méthode *method*. Une mise à jour est aussi nécessaire pour les appels à la méthode d'origine. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
    1. **Copy parameter** (*parameter*, *parameterName*, *method*) ;
    2. **Remove parameter** (*parameter*) ;
- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet des refactorisations qui

permettent le déplacement. Mais la refactorisation correspondante que nous proposons est : Move Parameter (*parameter*, *parameterName*, *method*).

5.16 Refactorisations composites appliquées sur une association

5.16.1 Ajouter une association

- **Nom de la refactorisation au niveau du modèle** : Add Association (*associationName*, *SourceClass*, *TargetClass*, *newMultiplicitySourceClass*, *newMultiplicityTargetClass*, *package*).
- **Comparaison entre refactorisation du modèle et refactorisation du code** :
  - pour appliquer cette refactorisation sur le modèle, une vérification est nécessaire pour s'assurer que le nom *associationName* n'existe pas déjà dans la classe *SourceClass*. L'application de cette refactorisation permet d'ajouter une association avec le nom *associationName* à la classe *SourceClass* en créant une association vide entre la classe source *SourceClass* et la classe cible *TargetClass*. Et en attribuant respectivement le nom *associationName* et les valeurs *newMultiplicitySourceClass* et *newMultiplicityTargetClass* choisis par l'utilisateur. Ceci peut être effectué en appliquant séquentiellement les refactorisations primitives suivantes :
    1. **Create Association** (*SourceClass*, *TargetClass*, *package*) ;
    2. **Rename Association** (*association*, *newName*) ;
    3. **Set Multiplicity Source** (*associationName*, *newMultiplicitySourceClass*) ;
    4. **Set Multiplicity Target** (*associationName*, *newMultiplicityTargetClass*) ;
  - l'application de cette refactorisation au niveau du code est plus complexe que celle appliquée au niveau du modèle puisqu'elle nécessite l'ajout des attributs références et de leur méthodes accesseurs. Par exemple, l'ajout d'une association bidirectionnelle au niveau du code se traduit par l'ajout à la classe source d'un attribut référençant une instance de la classe cible et des méthodes accesseurs à cet attribut, et l'ajout à la classe cible, d'un attribut référençant une instance de la classe source et des méthodes accesseurs à cet attribut. Ceci peut être effectué en appliquant séquentiellement les refactorisations composites suivantes :

1. **Add Attribute** (*attributeName*, *private*, *newType*, *Sourceclass*) ;
2. **Add Method** (*getAttributeName*, *public*, *newType*, *Sourceclass*) ;
3. **Add Method** (*setAttributeName*, *public*, *newType*, *Sourceclass*) ;
4. **Add Attribute** (*attributeName*, *private*, *newType*, *Targetclass*) ;
5. **Add Method** (*getAttributeName*, *public*, *newType*, *Targetclass*) ;
6. **Add Method** (*setAttributeName*, *public*, *newType*, *Targetclass*) ;

– **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet des refactorisations qui permettent l'ajout et ne traite pas les associations. Mais la refactorisation correspondante que nous proposons est : Add Association (*SourceClass*, *TargetClass*, *package*).

5.16.2 Déplacer une association

– **Nom de la refactorisation au niveau du modèle** : Move Association (*association*, *associationName*, *SourceClass*, *TargetClass*, *package*).

– **Comparaison entre refactorisation du modèle et refactorisation du code** :

– pour appliquer cette refactorisation sur le modèle, des vérifications sont nécessaires pour s'assurer que l'association avec le nom *associationName* n'est plus utile pour les classes liées et que la classe *TargetClass* ne contient pas une association qui a le même nom *associationName*. L'application de cette refactorisation permet de déplacer l'association portant le nom *associationName* au paquetage *package* pour lier la classe source *SourceClass* et la classe cible *TargetClass*. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :

1. **Copy Association** (*association*, *associationName*, *SourceClass*, *TargetClass*) ;
2. **Remove Association** (*association*, *SourceClass*, *TargetClass*) ;

- pour appliquer cette refactorisation sur le code, des vérifications sur les classes source et cible sont nécessaires pour s'assurer que les attributs de références des deux classes ne sont pas référencés dans leurs classes de définition. De plus, les noms des attributs de références qui vont être déplacés dans la classe source *SourceClass* et cible *TargetClass* ne doivent pas exister déjà dans ces dernières. L'application de cette refactorisation sur le code se traduit par le déplacement des attributs de références ainsi que leurs méthodes accesseurs respectivement dans la classe source *SourceClass* et cibe *TargetClass*. Ceci peut être effectué en appliquant séquentiellement les deux refactorisations primitives suivantes :
  1. **Copy Association** (*association*, *associationName*, *SourceClass*, *TargetClass*) ;
  2. **Remove Association** (*association*, *SourceClass*, *TargetClass*) ;

- **Nom de la refactorisation au niveau du code correspondante définie par Fowler** : dans le catalogue de Fowler [Fowler, 1999], aucune refactorisation du code ne correspond à notre refactorisation puisque Fowler n'a pas proposé pour tous les éléments constituant un programme orienté objet des refactorisations qui permettent le déplacement et ne traite pas les associations. Mais la refactorisation correspondante que nous proposons est : Move Association (*SourceClass*, *TargetClass*, *package*).

CHAPITRE 6

VALIDATION

Pour valider les refactorisations implantées et l'étude de la préservation de la traçabilité entre refactorisations du modèle et refactorisations du code, nous appliquons au niveau modèle du programme libre JHotDraw 5.2 les refactorisations correspondantes au niveau code et montrons que le résultat est conforme à la refactorisation manuelle du code de JHotDraw 5.2 vers JHotDraw 5.3.

6.1 Étude de cas

Nous utilisons une étude de cas pour valider l'étude de la traçabilité entre les refactorisations appliquées au niveaux modèle de classes et celles appliquées au niveau code : JHotDraw. Nous avons choisi ce programme car plusieurs refactorisations ont été déjà appliquées sur son code source, en plus de la disponibilité de sa documentation, de ses informations architecturales et de son code source libre.

6.1.1 JHotDraw

JHotDraw [Gamma et Eggenschwiler, 1998] est un programme de dessin vectoriel dont l'architecture met en œuvre de nombreux patrons de conception. Plusieurs versions existe déjà marquant ainsi son évolution. Nous pouvons donc connaître les refactorisations appliquées.

6.1.2 Objectifs

Nous voulons montrer que notre implantation des refactorisations et l'étude de la traçabilité permet d'assurer la traçabilité entre modèle et code d'un programme après applications de refactorisation au niveau du code comme du modèle.

6.1.3 Méthode

Afin de valider l'étude de la traçabilité entre refactorisation du modèle et refactorisation du code, nous appliquons certaines refactorisations implantées sur le modèle du programme JHotDraw, la version 5.2 et montrons que nos refactorisations assurent la traçabilité entre code et modèle. Nous avons donc besoin d'une version modifiée de JHotDraw 5.2 pour laquelle les refactorisations sont identifiées. Nous cherchons des refactorisations indépendantes de nos outils pour éviter au maximum les biais.

Pour trouver ces refactorisations, nous avons contacté Dig et Johnson qui ont réalisé l'outil *RefactoringCrawler* [Dig et Johnson, 2006] pour la détection des refactorisations produites entre deux versions différentes de l'application : JHotDraw 5.2 et JHotDraw 5.3. Dig nous a fourni la liste des refactorisations détectées. Ces refactorisations se focalisent sur les deux refactorisations suivantes : renommer une méthode et changer la signature d'une méthode.

Dans la section suivante, nous choisissons la refactorisation *Rename method* détectée par l'outil *RefactoringCrawler* pour l'appliquer sur le modèle de JHotDraw 5.2.

6.1.4 Application de la refactorisation *Rename Method*

L'outil *RefactoringCrawler* a détecté 5 refactorisations de renommage d'une méthode. Parmi celles-ci, nous choisissons la refactorisation suivante :

```
<refactoring name="RenamedMethods">
<parameter name="new name">CH.ifa.draw.standard.CreationTool.getCreatedFigure</parameter>
<parameter name="old name">CH.ifa.draw.standard.CreationTool.createdFigure</parameter>
</refactoring>
```

Cette refactorisation a été appliquée sur le code source de JHotDraw 5.2 pour renommer la méthode ***createdFigure*** qui appartient à la classe ***CreationTool*** du paquetage ***CH.ifa.draw.standard*** avec le nouveau nom ***getCreatedFigure*** dans la version JHotDraw 5.3.

Grâce à l'étude de la préservation de la traçabilité entre refactorisations du modèle et refactorisations du code effectuée dans le chapitre 5, nous sommes en mesure de montrer

que nos refactorisations assurent la traçabilité entre le modèle et le code source d'un programme.

Nous appliquons ainsi la même refactorisation appliquée sur le code source de JHotDraw 5.2, au niveau de son modèle décrit avec le métamodèle PADL obtenu par la rétro-conception. La refactorisation implantée et pouvant être appliquée sur le modèle de classes de JHotDraw 5.2 correspond à la refactorisation : ***RenameMethodWithPropagationToInterfaceWithOverloading*** *("CH.ifa.draw.standard.CreationTool", "createdFigure", "getCreatedFigure")*. Cette refactorisation permet de changer le nom de la méthode ***createdFigure*** appartennant à la classe ***CreationTool*** du paquetage ***CH.ifa.draw.standard*** par le nouveau nom ***getCreatedFigure*** qui indique mieux son but. Nous avons choisi cette refactorisation car nous voulons renommer la méthode ***createdFigure*** dans sa classe de définition : ***CreationTool*** et propager le renommage dans toute la hiérarchie de classes, incluant les interfaces, en autorisant la surcharge de méthodes.

6.1.4.1 Tests Unitaires

Comme mentionner dans le chapitre 4, nous utilisons JUnit pour mettre en place des tests unitaires pour notre étude de la traçabilité. Il nous a suffi d'ajouter à la classe ***RefactoringRenameMethodTest*** qui appartient au paquetage *refactoring.method.test* du projet ***Refactoring Tests*** une suite de tests pour vérifier les conditions de la refactorisation en question.

La figure 6.1 montre l'application de la refactorisation ***RenameMethodWithPropagationToInterfaceWithOverloading*** *("CH.ifa.draw.standard.CreationTool", "createdFigure", "getCreatedFigure")* sur le modèle de classes de JHotDraw 5.2. Avant l'application de cette refactorisation, le programme affiche dans la console le nom de la classe qui contient la méthode à renommer en la précédant du nom de son paquetage de définition : ***CH.ifa.draw.standard.CreationTool***, ainsi que le nom ***createdFigure*** de cette méthode. Après l'application de la refactorisation, le programme affiche le nom de la classe précédées du nom de son paquetage de définition et le nouveau nom ***getCreatedFigure*** de la méthode renommée pour s'assurer que la refactorisation a été appliquée avec succès.

Après avoir appliqué la refactorisation du renommage sur le modèle de classes de JHotDraw 5.2, nous avons obtenu le même modèle de classes que pour JHotDraw 5.3. Ceci valide notre étude de la traçabilité entre refactorisations des modèles et refactorisations du code, c'est-à-dire, l'application d'une refactorisation sur le modèle peut être tracée au niveau code source d'un programme et vice-versa.

Pour mieux illustrer l'étude de la traçabilité entre refactorisations du modèle et refactorisations du code, nous utilisons le schéma synthétique 6.2 suivant. Les modèle de classes de JHotDraw 5.2 et JHotDraw 5.3 sont obtenus par la rétro-conception produite par l'outil Ptidej [Guéhéneuc, 2005].

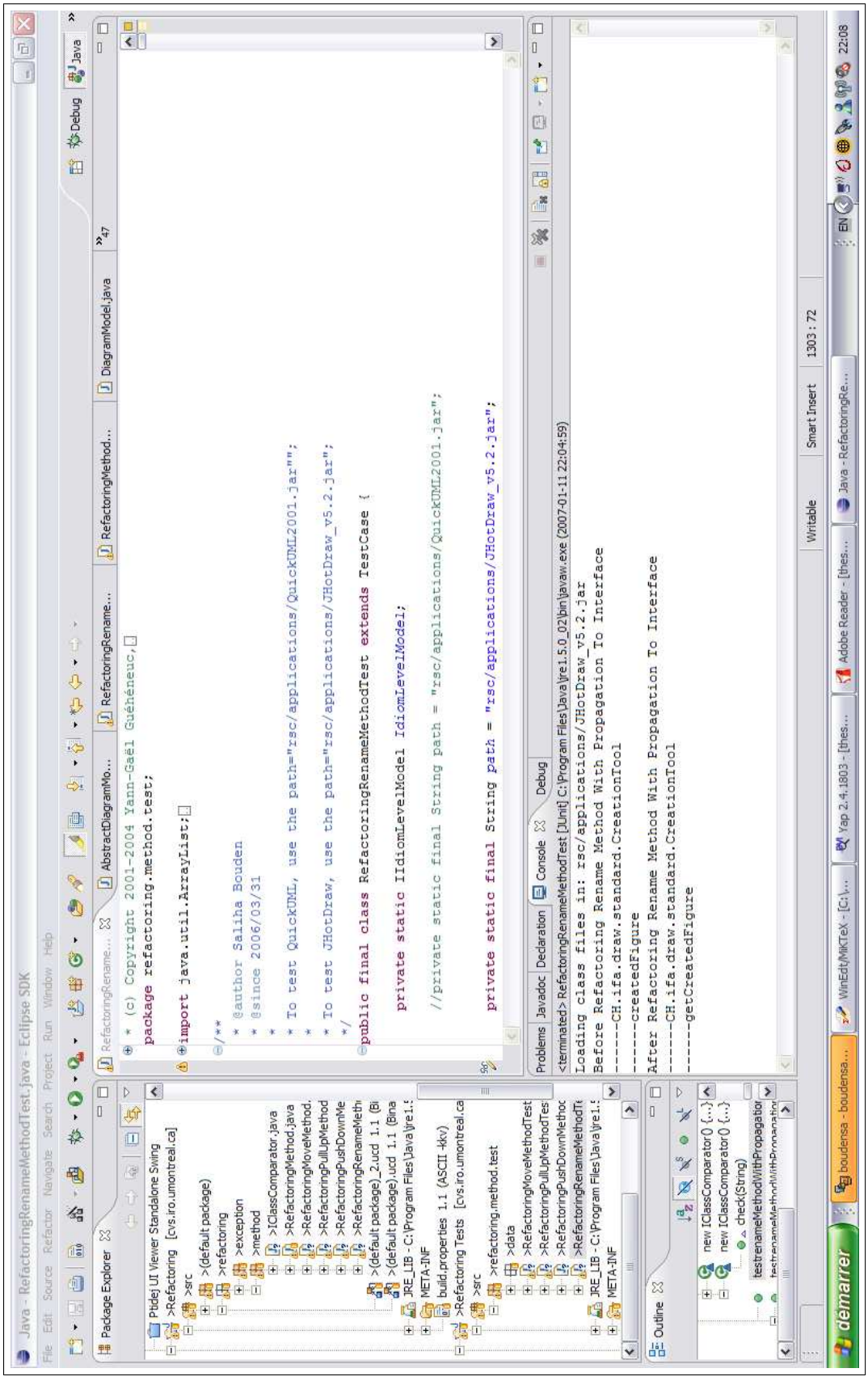

FIG. 6.1: Exemple de l'application de la refactorisation *Rename Method* sur le modèle de classes de JHotDraw 5.2

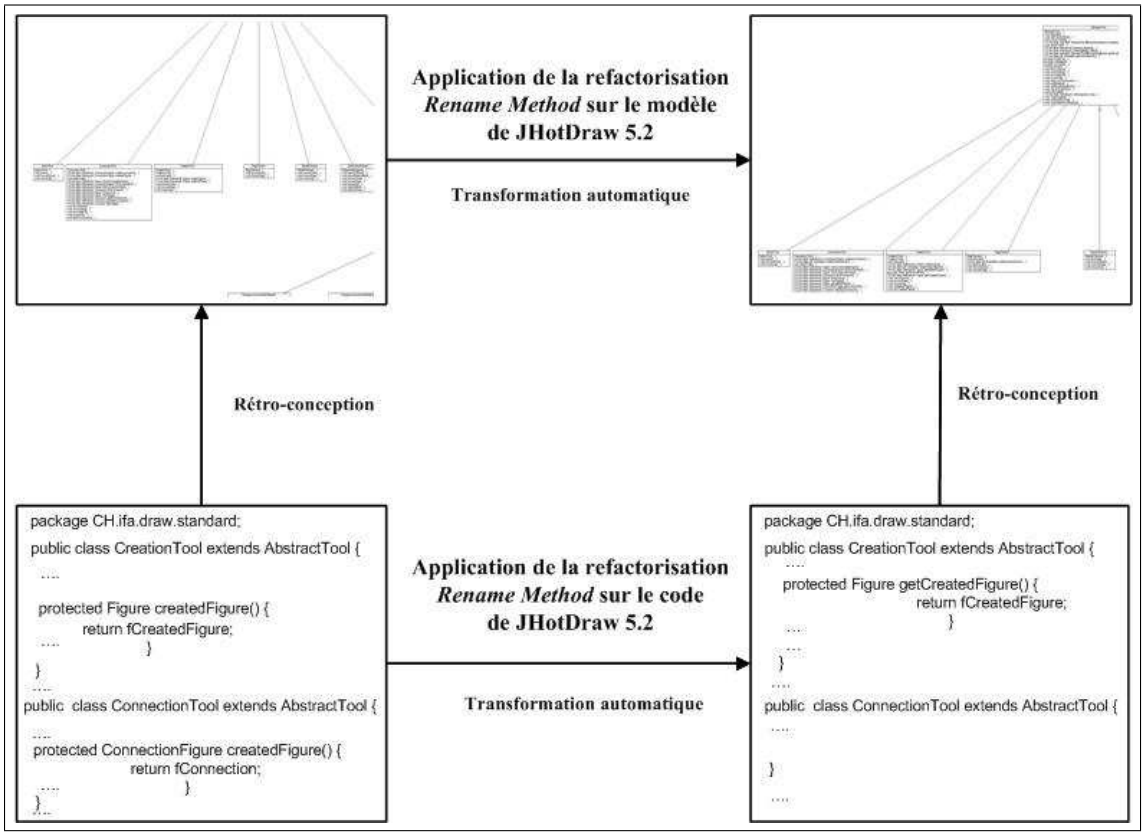

FIG. 6.2: Schéma synthétique illustrant l'étude de la traçabilité

CHAPITRE 7

CONCLUSION ET TRAVAUX FUTURS

Le travail présenté dans ce mémoire avait pour but d'étudier la préservation de la traçabilité entre refactorisations du modèle et refactorisations du code. L'approche adoptée pour étudier cette traçabilité consiste d'une part à définir et classifier 55 refactorisations primitives et composites appliquées sur le modèle et d'autre part à étudier la correspondance entre les refactorisations appliquées sur le code et celles sur le modèle.

La refactorisation est une technique importante dans le développement itératif. Elle consiste à changer la structure interne d'un programme pour faciliter sa compréhension et sa maintenance sans en changer son comportement externe [Fowler, 1999]. Mais habituellement, elle est appliquée sur le code source d'un programme et rarement sur le modèle.

Nous avons présenté dans le chapitre de l'état de l'art les travaux antérieurs dans le domaine de la refactorisation en nous focalisant sur les refactorisations qui préservent le comportement des programmes et des modèles. Certains outils développés pour aider à la refactorisation ne permettent pas de reporter les changements effectués sur le code source d'un programme au niveau des modèles et inversement des modèles au niveau du code. D'autres outils permettent de générer des modèles à partir du code refactorisé et inversement de générer le code à partir des modèles refactorisés. Cependant, ils n'assurent pas la traçabilité entre le modèle d'un programme et son code source.

Il est donc important d'effectuer des refactorisations au niveau des modèles et du code en parallèle afin de préserver la traçabilité entre refactorisations du code et des modèles et ainsi assurer la traçabilité entre la conception et le code source d'un programme. Plusieurs refactorisations sont connues et appliquées sur le code source d'un programme [Fowler, 1999] ou sur les modèles [Tichelaar *et al.*, 2000a]. Nous nous sommes inspirés de ces refactorisations pour définir un catalogue qui regroupe 55 refactorisations pour les modèles de classes des programmes orientés objet, exprimés dans un langage de modélisation proche de UML. Nous avons proposé une classification de ces refactorisations préservant les

modèles en nous inspirant de la hiérarchie des différents concepts de l'orienté objet. Nous avons défini des refactorisations primitives et d'autres composites pour chaque élément constituant le modèle, afin de mettre en valeur la similarité des opérations associées aux refactorisations. Les refactorisations composites ont été définies comme une séquence de refactorisations primitives ou composites. Aussi, nous avons distingué deux catégories de refactorisations : intra-élément qui s'applique au sein d'un élément (par exemple, une classe, une méthode) et inter-élément, qui s'appliquent entre plusieurs éléments dans l'architecture d'un programme. La classification des refactorisations du modèle proposée est générique, car l'ajout des refactorisations composites est possible par la composition de refactorisations primitives ou composites ou encore la composition des deux types de refactorisations.

Pour chaque élément constituant le modèle de classes (par exemple, les paquetages, les classes, les méthodes, les attributs et les associations), nous avons proposé les cinq refactorisations primitives suivantes : la création, le renommage, la suppression et la duplication ; et les deux refactorisations composites suivantes : l'ajout, et le déplacement. Pour la méthode et l'attribut, nous avons aussi défini les refactorisations composites qui permettent de faire monter ou descendre des méthodes et des attributs. En plus, une refactorisation composite qui permet d'encapsuler un attribut et une autre qui remplace un attribut par des sous-classes.

Nous avons défini aussi, pour chaque refactorisation primitive et composite, des pré-conditions et des post-conditions ainsi qu'une action, exprimée dans un pseudo code, nécessaires pour assurer la préservation du comportement.

Afin d'assurer la traçabilité entre la conception et le code source d'un programme, nous avons étudié la correspondance entre refactorisations du modèle et refactorisations du code en définissant les refactorisations correspondantes appliquées sur le code source pour chaque refactorisation du modèle définie.

Nous avons implanté un sous-ensemble des refactorisations du modèle en utilisant le métamodèle PADL (*Pattern and Abstract-Level Description Language*) qui permet de modéliser des programmes orientés objet [Guéhéneuc, 2003]. Nous avons ajouté à ce métamodèle des algorithmes utilitaires qui permettent de donner les classes de la

hiérarchie d'une classes donnée pour assurer la préservation de la sémantique du modèle. L'implantation de ces refactorisations a été réalisée de façon générique et modulaire afin de faciliter l'ajout de d'autres types de refactorisations.

Pour valider les refactorisations implantées et l'étude de la préservation de la traçabilité entre refactorisations du modèle et refactorisations du code, nous avons appliqué les refactorisations implantées sur les programmes libres suivants : QuickUML et JHotDraw. La détection de certaines refactorisations produites entre deux versions différentes de l'application : JHotDraw 5.2 et JHotDraw 5.3, nous a beaucoup facilité la tâche de la validation de l'étude de la traçabilité. Nous avons ainsi appliqué la même refactorisation appliquée déjà sur le code source de JHotDraw 5.2, au niveau de son modèle décrit avec le métamodèle PADL. Le modèle de classes refactorisé était identique au modèle de classes de JHotDraw 5.3. Ceci valide notre étude de la traçabilité entre refactorisations des modèles et refactorisations du code, c'est-à-dire, l'application de la même refactorisation sur le modèle et le code source d'un programme assure la préservation de la traçabilité entre la conception et le code source d'un programme. Les résultats obtenus sont donc très satisfaisants et encourageants pour ajouter et terminer l'implantation de toutes les refactorisations définies dans notre catalogue.

Grâce à l'étude de la traçabilité et l'implantation des refactorisations au niveau du modèle, nous sommes de position de créer un outil semi-automatique de refactorisations des modèles et du code qui assurent la traçabilité entre les refactorisations des modèles et les refactorisations du code.

Par ailleurs, ces refactorisations vont être utilisées pour corriger les défauts de conception détectés dans les travaux de Moha [Moha et Guéhéneuc, 2005] pour améliorer la qualité des programmes [Moha *et al.*, 2006].

Il sera possible d'ajouter d'autres refactorisations en utilisant et composant les refactorisations primitives et composites définies dans le chapitre 3.

Enfin, il sera important d'implanter toutes les refactorisations primitives et composites définies dans le chapitre 3 et de les tester sur d'autres applications.

BIBLIOGRAPHIE

[Albin-Amiot, 2003]

Hervé Albin-Amiot. *Idiomes et Patterns Java : Application à la Synthèse de Code et à la Détection*. Thèse de doctorat, université de Nantes, février 2003.

[Antoniol et Guéhéneuc., 2005]

Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification : A novel approach and a case study. In Tibor Gyim'othy et Vaclav Rajlich, editor, *proceedings of the 21st International Conference on Software Maintenance*. IEEE Computer Society Press, Septembre 2005.

[Astels, 2001]

D. Astels. Refactoring with uml, 2001.

[Beck, 1999]

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1^{st} edition, October 1999. ISBN: 0-201-61641-6.

[Bottoni *et al.*, 2005]

Paolo Bottoni, Francesco Parisi-Presicce, George Mason, and Gabriele Taentzer. Specifying coherent refactoring of software artefacts with distributed graph transformations. In Patrick van Bommel, editor, *Transformation of Knowledge, Information and Data*, pages 95–125. Information Science Publishing (an imprint of Idea Group Inc.), USA, 1^{st} edition, 2005. ISBN: ISBN: 1-59140-527-0 (hard cover), 1-59140-528-9 (soft cover).

[Casais, 1994]

Eduardo Casais. Automatic reorganization of object-oriented hierarchies: A case study. *Object-Oriented Systems*, 1(2):95–115. Chapman & Hall, 1994.

[Chikofsky et Cross II, 1990]

Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[Coleman, 1994]

John Coleman. Commentary on Bird and Klein. *CL*, 20(3):492, 1994.

[Demeyer *et al.*, 2002]

Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. ISBN: 1-55860-639-4.

[Dig et Johnson, 2006]

Dany Dig and Ralph Johnson. Toward automatic upgrades of component-based applications. In *Proceedings of the Finals of cross-disciplinary ACM Student Research Competition*. ACM Press, May 2006.

[Fanta et Rajlich, 1998]

Richard Fanta and Vaclav Rajlich. Reengineering object-oriented code. In *Proceedings of the International Conference on Software Maintenance*, 1998.

[Fanta et Rajlich, 1999]

R. Fanta and V. Rajlich. Restructuring legacy c code into c++. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 77, Washington, DC, USA, 1999. IEEE Computer Society. ISBN: 0-7695-0016-1.

[Flores et Robidoux, 2004]

Ward Flores and Sébastien Robidoux. C++ parser for padl. Rapport technique 31 pages, Diro, Université de Montréal, octobre 2004.

[Foote et Opdyke, 1995]

Brian Foote and William F. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In *Pattern languages of program design*, pages 239–257, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. ISBN: 0-201-60734-4.

[Fowler, 1999]

Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999. ISBN: 0-201-48567-2.

[Gamma *et al.*, 1994]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns –*

Elements of Reusable Object-Oriented Software. Addison-Wesley, 1^{st} edition, 1994. ISBN: 0-201-63361-2.

[Gamma et Beck, 1998]

Erich Gamma and Kent Beck. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50. SIGS Publications, July 1998.

[Gamma et Eggenschwiler, 1998]

Erich Gamma and Thomas Eggenschwiler. JHotDraw. Web site, 1998.

[Glass, 1998]

Robert L. Glass. Maintenance: Less is not more. *IEEE Software*, 15(4):67–68, 1998.

[Goldberg et Robson, 1983]

Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN: 0-201-11371-6.

[Gosling *et al.*, 2000]

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000. ISBN: 0-201-31008-2.

[Griswold, 1992]

William G. Griswold. *Program restructuring as an aid to software maintenance*. Ph.D. thesis, University of Washington, Seattle, WA, USA, 1992.

[Guéhéneuc et Albin-Amiot, 2004]

Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *proceedings of the* 19^{th} *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.

[Guéhéneuc, 2003]

Yann-Gaël Guéhéneuc. *Un cadre pour la traçabilité des motifs de conception*. Thèse de doctorat, École des Mines de Nantes et Université de Nantes, juin 2003.

[Guéhéneuc, 2005]

Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *proceedings of*

the 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005. **Submitted for publication.**

[Guyomarc'h, 2006]

Jean-Yves Guyomarc'h. Une architecture pour l'évaluation qualitative de l'impact de la programmation orientée aspect. thèse de maîtrise, Diro, Université de Montréal, Mai 2006.

[Jackson et Waingold, 2001]

D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. *Software Engineering, IEEE Transactions on*, 27(2):156–169, 2001.

[Kiczales *et al.*, 1992]

Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. The art of the meta-object protocol. *SIGPLAN Not.*, 27(2):9, New York, NY, USA, 1992. ACM Press. Reviewer-Alan Wexelblat.

[Mens *et al.*, 2002]

T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations, 2002.

[Mens et Tourwé, 2004]

Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, Piscataway, NJ, USA, 2004. IEEE Press.

[Meyer, 1992]

Bertrand Meyer. *Eiffel : The Language*. Prentice Hall., 1992. ISBN: 0-132-47925-7.

[Moha *et al.*, 2006]

Naouel Moha, Saliha Bouden, and Yann-Gaël Guéhéneuc. Correction of high-level design defects with refactorings. In Serge Demeyer, Stéphane Ducasse, Yann-Gaël Guéhéneuc, Kim Mens, and Roel Wuyts, editors, *Proceedings of the* 7^{th} *ECOOP Workshop on Object-Oriented Reengineering*, July 2006.

[Moha et Guéhéneuc, 2005]

Naouel Moha and Yann-Gaël Guéhéneuc. On the automatic detection and correction of design defects. In Serge Demeyer, Kim Mens, Roel Wuyts, and Stéphane Ducasse,

editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. Springer-Verlag, July 2005. **To be submitted for publication in June 2005.**

[Monson-Haefel, 2001]

Richard Monson-Haefel. *Enterprise JavaBeans*. O'relly, 2001. ISBN: 1-56592-605-6.

[Moore, 1995]

Ivan Moore. a tool for automatic restructuring of self inheritance hierarchies. In *In TOOLS USA 1995*, pages 267–275, 1995.

[Moore, 1996]

Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 235–250, New York, NY, USA, 1996. ACM Press. ISBN: 0-89791-788-X.

[Niere *et al.*, 2003]

Jörg Niere, Holger Giese, Sven Burmester, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level within the fujaba tool suite, 2003.

[Object Management Group, Inc., 2002]

Object Management Group, Inc. *MOF. Meta-Object Facility specification, version 1.4*, April 2002.

[OMG, march 2004]

OMG. CORBA 3.0.3, Common Object Request Broker Architecture (Core Specification), 2004-03-01, march 2004.

[Opdyke, 1992]

William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Urbana-Champaign, IL, USA, 1992.

[Richner et Ducasse, 1999]

Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White,

editors, *proceedings of* 7^{th} *International Conference on Software Maintenance*, pages 13–22. IEEE Computer Society Press, August 1999.

[Roberts *et al.*, 1997]

Donald Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, New York, NY, USA, 1997. John Wiley & Sons, Inc.

[Roberts, 1999]

Donald Bradley Roberts. *Practical analysis for refactoring.* Ph.D. thesis, University of Illinois at Urbana-Champaign, 1999. Adviser-Ralph Johnson. ISBN: 0-599-46857-2.

[Rumbaugh *et al.*, 1999]

James Rumbaugh, Robert Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual.* Addison-Wesley, 1^{st} edition, January 1999. ISBN: 0-201-30998-X.

[Sharon, 1996]

David Sharon. Meeting the challenge of software maintenance. *IEEE Softw.*, 13(1):122–126, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[Sunyé *et al.*, 2001]

Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jezequel. Refactoring UML models. In *The Unified Modeling Language*, pages 134–148, 2001.

[Tichelaar *et al.*, 2000a]

Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. *Proceedings of ISPSE*, 2000.

[Tichelaar *et al.*, 2000b]

Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pages 157–167. IEEE, 2000.

[Tokuda et Batory, 2001]

Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. In *Automated Software Engineering*, pages 174–, 2001.

Annexe I

Traçabilité entre refactorisations du code et refactorisations du modèle

Cette annexe présente la liste des refactorisations listées du catalogue de Fowler [Fowler, 1999] [1]. Ces refactorisations s'appliquent sur le code source d'un programme écrit en Java. Nous étudions dans le tabeau suivant la possibilité d'appliquer ces refactorisations (81 refactorisations) sur le modèle d'un programme, afin de compléter notre étude sur la traçabilité entre refactorisations du code et refactorisation du modèle.

TAB. I.1: Traçabilité entre refactorisations du code et refactorisation du modèle

Nom de la refactorisation	**Description**	**Code source**	**Modèle**
Add Parameter	Ajouter un paramètre à une méthode	√	√
Change Bidirectional Association to Unidirectional	Changer une association bidirectionnelle en une association unidirectionnelle si une classe n'a plus besoin des services de l'autre classe	√	√
Change Reference to Value	Si un objet de référence est petit, immuable et difficile à contrôler. Transformer le en objet de valeur [2]	√	√
Change Unidirectional Association to Bidirectional	Si deux classes doivent utiliser les services de chacune des deux, mais il y a seulement un lien à sens unique. Changer l'association unidirectionnelle en une association bidirectionnelle, en changeant les modificateurs pour mettre à jour les deux ensembles	√	√

[1] http ://www.refactoring.com/catalog/index.html

Nom de la refactorisation	Description	Code source	Modèle
Change Value to Reference	Si une classe possède beaucoup d'instances identiques et qu'on veut la remplacer par un objet simple. Il vaut mieux transformer l'objet en objet de référence [3]	√	√
Collapse Hierarchy	La superclasse et la sous-classe ne sont pas très différentes. Alors, on peut les fusionner	√	√
Consolidate Conditional Expression	On a un ordre de structure conditionnelle avec le même résultat. Combiner les dans une seule expression conditionnelle simple	√	
Consolidate Duplicate Conditional Fragments	Si un fragment de code est dupliqué dans toutes les branches d'une expression conditionnelle. Déplacer le à l'extérieur de cette expression	√	
Convert Dynamic to Static Construction (par Gerard M. Davison)	Si un code charge d'autres classes dynamiquement. Ceci peut produire un code plus fragile. Il faut le remplacer par un code statique [4]	√	
Convert Static to Dynamic Construction (par Gerard M. Davison)	Cette méthode peut être employée avec succès dans les applications (frameworks) où on ignore quelle classe a besoin d'être compilé. Si les classes ont des dépendances à la compilation sur des classes qui peuvent être seulement bâti sur une plateforme spécifique. Exécuter le paquetage (*java.lang.reflect*) pour détruire la dépendance statique [5]	√	
Decompose Conditional	si une expression conditionnelle (*if-then-else*) est compliquée. Extraire les méthodes de la condition (*if*), ensuite de (*then*) et de (*else*)	√	

Nom de la refactorisation	Description	Code source	Modèle
Duplicate Observed Data	Si les données du domaine sont disponibles seulement dans un objet de la GUI et les méthodes du domaine en ont besoin, copier les données dans un objet du domaine. Ensuite, installer un observateur pour synchroniser les deux morceaux de données	√	√
Encapsulate Collection	Si une méthode retourne une collection. Changer son retour en lecture seulement et fournir des méthodes d'ajout et de suppression (*add*/*remove*). En effet, la collection est correctement encapsulée, ce qui réduit le couplage	√	√
Encapsulate Downcast	Si une méthode retourne un objet qui doit être trans-typé par ses appelants. Déplacer le *downcast* à l'intérieur de la méthode	√	
Encapsulate Field	Si un champ (une variable) est déclarée public, changer sa déclaration en privé et fournir les méthodes (*get* et *set*)	√	√
Extract Class	Si une classe effectue le travail qui devrait être effectué par deux. Créer une nouvelle classe et déplacer les champs et les méthodes appropriés à la nouvelle classe	√	√
Extract Interface	Plusieurs clients utilisent le même sous-ensemble de l'interface d'une classe, ou deux classes ont une partie de leurs interfaces en commun. Extraire le sous- ensemble dans une interface	√	√

Nom de la refactorisation	Description	Code source	Modèle
Extract Method	Si un fragment de code peut être groupé ensemble. Transformer le fragment en méthode dont le nom explique le but de la méthode	√	
Extract Package (par Gerard M. Davison)	Si un paquetage a trop de classes, devient trop grand et difficile à comprendre. Extraire un paquetage secondaire selon les dépendances ou l'utilisation	√	√
Extract Subclass	Si une classe possède des caractéristiques utilisées seulement par certaines instances. Créer une sous-classe pour ce sous-ensemble de caractéristiques	√	√
Extract Superclass	Si deux classes possèdent des caractéristiques semblables. Créer une superclasse et déplacer les caractéristiques communes à la superclasse	√	√
Form Template Method	Si deux méthodes dans des sous-classes exécutent des étapes semblables dans le même ordre, mais que les étapes sont différentes. Il faut saisir ces étapes dans des méthodes avec une même signature, de sorte que les méthodes originales deviennent les mêmes et ensuite les faire monter	√	√
Hide Delegate	Un client appelle une classe déléguée d'un objet. Si cette dernière change, le client doit être changé aussi. Pour enlever cette dépendance, il faut créer des méthodes sur le serveur pour cacher le délégué. Ainsi, les changements deviennent limités au serveur et ne se propagent pas au client	√	√

Nom de la refactorisation	Description	Code source	Modèle
Hide Method	Si une méthode n'est utilisée par aucune autre classe. Il est préférable de rendre cette méthode privée. Si une méthode de getting ou setting est déclarée priver et on utilise un accès de variable direct. Dans ce cas, on peut supprimer la méthode	√	√
Inline Class	Si une classe n'est pas utile, déplacer tous ses caractéristiques dans une autre classe et supprimer la. La refactorisation *Inline class* est l'inverse de la refactorisation *Extract class*	√	√
Inline Method	Si le corps d'une méthode est aussi clair que son nom. Mettre le corps de la méthode dans le corps de ses appelants et ensuite supprimer la méthode	√	
Inline Temp	La plus part du temps, Inline Temp est employée en tant qu'élément de la refactorisation Replace Temp with Query. Elle est employée seule dans le cas où la valeur d'un appel de méthode est assignée à une variable temporaire. Dans ce cas, il faut remplacer toutes les références de temp par l'expression	√	
Introduce Assertion	Une assertion est une instruction conditionnelle. L'échec d'une assertion indique les erreurs du programmeur. Les assertions ne devraient jamais être employées par d'autres parties du système. L'assertion facilite la communication et le déboguage. Ajouter une assertion est toujours considéré comme une préservation de comportement du moment qu'elle n'affecte pas le fonctionnement du système. Utiliser les assertions seulement pour vérifier les choses qui doivent être vraies. Son utilisation avec excès peut mener à dupliquer la logique qui est difficile à maintenir	√	

Nom de la refactorisation	Description	Code source	Modèle
Introduce Explaining Variable	Si une expression est complexe et devient difficile à lire. Mettre le résultat de l'expression ou une partie de cette dernière dans une variable temporaire avec un nom qui explique le but	√	
Introduce Foreign Method	Si une classe serveuse (*server class*) ne répond pas aux besoins. Dans ce cas, une méthode additionnelle est nécessaire, mais on ne peut pas modifier cette classe. Alors, on doit créer une méthode et l'introduire dans une classe cliente avec une instance de la classe serveuse comme son premier argument	√	√
Introduce Local Extension	Si une classe serveuse utilisée a besoin de plusieurs méthodes additionnelles, mais on ne peut pas modifier cette classe. Créer une nouvelle classe qui contient ces méthodes supplémentaires. Rendre la classe d'extension en une sous-classe (ou une encapsulation) de la classe originale	√	√
Introduce Null Object	Si une vérification pour une valeur nulle est répétée. Remplacer la valeur nulle par un objet nul	√	√
Introduce Parameter Object	Si un groupe particulier de paramètres tend à être passés ensemble. Plusieurs méthodes peuvent utiliser ce groupe, ainsi qu'une classe ou plusieurs classes. Alors, il faut le remplacer par un objet. Cette refactorisation est utile, puisqu'elle réduit la taille des listes de paramètre. Les listes de paramètre longues sont difficiles à comprendre	√	√

Nom de la refactorisation	Description	Code source	Modèle
Move Class (par Gerard M. Davison)	Si une classe est crée dans un paquetage contenant d'autres classes qui ne sont pas reliées à cette classe. Il vaut mieux déplacer la classe à un paquetage plus approprié ou créer un nouveau paquetage qui est nécessaire pour des utilisations futures. En effet, ceci peut aider à enlever les dépendances complexes au niveau des paquetages et à faciliter la réutilisation et la recherche des classes par les développeurs	√	√
Move Field	Si un champ est utilisé davantage par d'autres classe que la classe dans laquelle il est défini. Il faut créer un nouveau champ dans la classe cible, et changer tous ses utilisateurs. Une autre raison pour utiliser la refactorisation *Move Field* est lorsque la refactorisation *Extract Class* est appliquée, dans ce cas, les champs vont être déplacés d'abord et puis les méthodes	√	√
Move Method	Cette refactorisation est utilisée quand les classes ont une multitude de comportement ou quand les classes collaborent trop et sont fortement couplées. Si une méthode est utilisée davantage par d'autres classe que la classe dans laquelle elle est définie. Il faut créer une nouvelle méthode avec un corps semblable et l'introduire dans la classe qui l'utilise davantage. Ensuite, transformer la vieille méthode en une délégation simple, ou la supprimer entièrement	√	√

Nom de la refactorisation	Description	Code source	Modèle
Parameterize Method (Contribuant : Bernd Kahlbrandt)	Plusieurs méthodes exécutent les mêmes instructions mais avec différentes valeurs contenues dans le corps de la méthode. Dans ce cas, il faut créer une seule méthode qui utilise un paramètre pour les différentes valeurs. En effet, une telle refactorisation supprime le code dupliqué et augmente la flexibilité, puisque cela facilite de traiter d'autres variations en ajoutant des paramètres	√	√
Preserve Whole Object	Si on a besoin de plusieurs valeurs d'un objet et on veut passer ces valeurs comme des paramètres dans un appel de méthode. Dans ce cas, il vaut mieux envoyer l'objet entier. Ceci permet de rendre la liste de paramètres plus robuste aux changements, et le code plus lisible. Il est peut être difficiles de travailler avec une longue listes de paramètres, du fait que la méthode appelée et l'appelant doivent se rappeler des valeurs passées. D'une autre part, le passage de la longue liste de paramètres encourage également la duplication de code, du fait que l'objet appelé ne peut pas tirer profit d'aucune autre méthode existante dans l'objet entier pour calculer des valeurs intermédiaires. quand on fait un passage par valeurs, l'objet appelé a une dépendance sur les valeurs, mais il n'y a aucune dépendance à l'objet à partir duquel les valeurs ont été extraites. Le passage dans l'objet requis cause une dépendance entre l'objet requis et l'objet appelé. Si ceci va gâcher la structure de dépendance, n'employer pas cette refactorisation. Une autre raison pour ne pas utiliser cette refactorisation est quand l'objet appelé a besoin seulement d'une seule valeur de l'objet requis, il est meilleur de passer la valeur que de passer l'objet entier	√	√

Nom de la refactorisation	Description	Code source	Modèle
Pull Up Constructor Body	On a des constructeurs dans des sous-classes avec des corps identiques la plupart du temps. Créer un constructeur dans la superclasse et l'appeler à partir des méthodes déclarées dans les sous-classes. Dans ce cas, on ne peut pas utiliser la refactorisation *Pull Up Method*, parce qu'on ne peut pas hériter des constructeurs	√	√
Pull Up Field	On trouve souvent des caractéristiques dupliquées si des sous-classes ont été développé indépendamment, ou combiner en utilisant la refactorisation. Particulièrement, certains champs peuvent être dupliqués. Si ces derniers sont utilisés de la même façon, donc on peut les déplacer dans la superclasse	√	√
Pull Up Method	L'élimination de la duplication de comportement est importante. Si on a des méthodes ayant des résultats identiques dans les sous-classes. Déplacer les à la superclasse	√	√
Push Down Field	La refactorisation *Push Down Field* est l'opposé de la refactorisation *Pull Up Field*. Cette refactorisation est utilisée quand un champ n'est pas utilisé par la superclasse mais seulement par certaines sous-classes. Pour cela, il faut déplacer le champ dans la sous-classe	√	√

Nom de la refactorisation	Description	Code source	Modèle
Push Down Method	La refactorisation *Push Down Method* est l'opposé de la refactorisation *Pull Up Method*. Son utilisation est requise quand on a besoin de déplacer le comportement d'une superclasse à une sous-classe spécifique. On le fait souvent quand on utilise *Extract Subclass*	√	√
Reduce Scope of Variable (par Mats Henricson)	Une variable locale est déclarée dans une portée plus grande de son utilisation. Il vaut mieux réduire la portée de cette variable de sorte qu'elle soit seulement visible dans la portée où elle est utilisée, afin d'éviter les erreurs	√	
Remove Assignments to Parameters	Pour éviter la confusion entre le passage de paramètres par valeur et par référence. Le langage Java utilise le passage de paramètres par valeur exclusivement. Si un code est affecté à un paramètre d'une méthode, utiliser une variable temporaire pour affecter le paramètre à cette variable. Bien sur, cette règle n'est pas nécessairement appliquée à d'autres langages de programmation qui utilisent les paramètres de sorties. Bien que même, il est préférable d'utiliser les paramètres de sorties le moins possible	√	
Remove Control Flag	Si le déroulement d'une instruction d'itération est contrôlé au moyen d'une variable qui agit en tant que drapeau de contrôle pour une série d'expressions booléennes. Remplacer cette variable par une instruction break ou return pour sortir de la boucle sans exécuter la suite des instructions	√	

Nom de la refactorisation	Description	Code source	Modèle
Remove Double Negative (contribuant : Ashley Frieze)	Si un code contient une expression conditionnelle négative double. Remplacer cette dernière par une expression conditionnelle négative simple pour éviter les erreurs de la confusion et pour rendre le code plus facile à comprendre	√	
Remove Middle Man	Si une classe effectue beaucoup de délégation simple. Dans ce cas, obliger le client à appeler le délégué directement	√	√
Remove Parameter	Si un paramètre n'est plus utilisé par le corps d'une méthode, supprimer le	√	√
Remove Setting Method (contribuant : Paul Haahr)	Fournir une méthode *Setting* indique qu'un champ peut être changé. Si on ne veut pas que ce champ change une fois l'objet est créé, alors il faut éviter de fournir la méthode *setting* et ainsi déclarer le champ final. Par contre, on ne peut pas utiliser un champ déclaré final si l'initialisation est faite dans une méthode séparée au constructeur	√	√
Rename Method	Le nom d'une méthode n'indique pas son but. Changer le nom de la méthode	√	√
Replace Array with Object	Un tableau est une structure de données qui contient seulement une collection d'objets semblables, organisées dans un certain ordre. Cependant, si un tableau est déclaré avec des différent types. Remplacer le par un objet possédant un champ pour chaque élément	√	

Nom de la refactorisation	Description	Code source	Modèle
Replace Assignment with Initialization (contribuant : Mats Henricson)	Si un code déclare d'abord une variable et lui assigne ensuite une valeur. Transformer ce code en une initialisation directe	√	
Replace Conditional with Polymorphism	Si une instruction conditionnelle choisit des comportements différents, dépendant du type d'objets. Déplacer alors, chaque branche de l'instruction conditionnelle à une méthode de redéfinition [6] (spécialisation) dans une sous-classe. Et rendre ensuite la méthode originale abstraite (*abstract*). Du fait que, le polymorphisme évite d'écrire une instruction conditionnelle explicitement quand les objets dont le comportement change selon leurs types. L'avantage du polymorphisme se produit quand un ensemble de conditions apparaît dans beaucoup d'endroits dans le programme. Si on veut ajouter un nouveau type, on doit trouver et mettre à jour toutes les conditions. Mais avec des sous-classes on a juste besoin de créer une nouvelle sous-classe et ensuite, fournir les méthodes appropriées	√	√
Replace Conditional with Visitor (contribuant : Ivan Mitrovic)	Si une instruction conditionnelle "agressive" se développe de façon improbable (aléatoire ou dynamique), choisit des comportements différents, selon le type d'objets et se répète dans tout le code. Créer alors une instance concrète de l'objet Visiteur pour chaque type de données dans l'instruction conditionnelle. Créer ensuite, une instance concrète du visiteur qui encapsule la logique de chaque condition et visite les objets. Cette refactorisation ne s'intéresse pas au nombre de conditions *switch* basées sur le même type de code qui existent dans le système	√	√

Nom de la refactorisation	Description	Code source	Modèle
Replace Constructor with Factory Method	Lors de la création d'un objet, si on veut effectuer plus qu'une instanciation. Alors, on a besoin de remplacer le constructeur par une méthode Factory car le constructeur ne possède aucune valeur de retour	√	√
Replace Data Value with Object	Souvent durant les premières phases du processus de développement, on représente des faits simples en tant que données élémentaires simples. Mais rapidement, on se rend compte que ces données élémentaires ont besoin de données ou de comportements additionnelles. On a donc besoin de transformer les données en objet données	√	√
Replace Delegation with Inheritance	Si on utilise toutes les méthodes de la classe déléguée et on est obligé de réécrire toutes ces méthodes. Dans ce cas, il est préférable de remplacer la délégation par l'héritage. Cependant, il est recommandé d'éviter d'appliquer cette refactorisation dans les deux cas suivants : – si on n'utilise pas toutes les méthodes de la classe à laquelle on délègue et une sous-classe devrait toujours suivre l'interface de la superclasse, – si le délégué est partagé par plusieurs objets d'une part et mutable d'une autre part. Dans ce cas, on ne peut pas utiliser cette refactorisation parce que les données ne sont plus partagées	√	√

Nom de la refactorisation	Description	Code source	Modèle
Replace Error Code with Exception	De nombreux langages de programmations utilisent des sorties (résultats) spéciales pour indiquer l'erreur. Unix et le langage C utilisent un code de retour (instruction return) pour signaler le succès ou l'échec d'un sous-programme (routine). Le langage Java a une meilleure manière pour gérer les erreurs basée sur les exceptions qui séparent clairement un traitement normal d'un traitement d'erreur	√	
Replace Exception with Test	Si un code lance une exception en fonction d'une condition que l'appelant pourrait vérifiée d'abord. Changer l'appelant pour faire le test en premier	√	
Replace Inheritance with Delegation	Si une sous-classe utilise seulement une partie de l'interface des superclasses ou refuse d'hériter des données. Dans ce cas, il faut créer un champ pour la superclasse, adapter ensuite les méthodes pour déléguer à la superclasse et à la fin supprimer l'héritage	√	√
Replace Iteration with Recursion (contribuant : Dave Whipp)	Le terme "invariant" est utilisé pour décrire la condition résultante de chaque itération. Un invariant peut être ajouté au code sous forme de commentaires ou d'assertions. Si dans une boucle, il est difficile d'établir ce que chaque itération fait, alors la solution est de remplacer l'itération par la récursivité. En effet, contrairement à la plupart construction de boucles procédurales, un appel de fonction récursif permet de donner un nom significatif qui devrait refléter la boucle invariante	√	

Nom de la refactorisation	Description	Code source	Modèle
Replace Magic Number with Symbolic Constant	Si on a un nombre magique avec une signification particulière. Créer une constante, ayant un nom significatif et ensuite remplacer le nombre magique par ce nom dans tout le code	√	
Replace Method with Method Object (contribuant : Marnix Klooster)	Si un code comprend une longue méthode qui utilise des variables locales ne permettant pas ainsi d'appliquer la refactorisation *Extract Method*. Pour résoudre ce problème, il faut transformer la méthode en objet de sorte que toutes les variables locales deviennent des champs de cet objet. De ce fait, la méthode peut être décomposé en d'autres méthodes dans le même objet	√	√
Replace Nested Conditional with Guard Clauses	Si une méthode a un comportement conditionnel qui ne suit pas clairement le chemin normal de l'exécution. Utiliser les assertions [7] pour tous les cas spéciaux. Une assertion est la désignation de la vérification conditionnelle inhabituelle basée sur la vérification d'une condition. Si cette condition est vraie, une instruction de retour est exécutée	√	
Replace Parameter with Explicit Methods	Si une méthode exécute du code différent qui dépende des valeurs d'un paramètre énuméré. Dans ce cas, créer une méthode séparée pour chaque valeur du paramètre	√	

Nom de la refactorisation	Description	Code source	Modèle
Replace Parameter with Method	Si un objet invoque une méthode pour passer le résultat comme paramètre pour une autre méthode et le receveur peut également appeler cette méthode. Supprimer alors le paramètre et laisser le receveur appeler la méthode	√	
Replace Record with Data Class	Les structures d'enregistrements sont un dispositif commun des environnements de programmation. Il y a de diverses raisons de les introduire dans un programme orienté objet pour communiquer un enregistrement structuré avec un API de programmation traditionnelle, ou un enregistrement de base de donnée. Dans ces cas, il est indispensable de créer une classe d'interfaçage pour traiter cet élément externe. En effet, il est plus simple de rendre la classe semblable à l'enregistrement externe et déplacer après, d'autres champs et méthodes dans la classe	√	
Replace Recursion with Iteration (contribuant : Ivan Mitrovic)	Bien que la récursivité soit souvent plus élégante et plus facile à comprendre que la solution itérative. Toutefois, la récursivité complexe est difficile à comprendre et devrait être considérée comme une "mauvaise odeur" dans le code. Ainsi, il faut la remplacer par l'itération	√	
Replace Static Variable with Parameter (contribuant : Marian Vittek)	Si une fonction dépend d'une variable statique et cette fonction doit être réutilisée dans un contexte plus général. Pour cela, il faut ajouter un nouveau Paramètre à la fonction et remplacer toutes les références de la variable statique dans la fonction par ce nouveau paramètre	√	

Nom de la refactorisation	Description	Code source	Modèle
Replace Subclass with Fields	Si les sous-classes d'une hiérarchie comportent seulement des méthodes qui retournent des données constantes. Afin d'améliorer la complexité de la hiérarchie, il faut déplacer ces méthodes dans les champs de la superclasse et supprimer ensuite les sous-classes complètement	√	√
Replace Temp with Query	Si une variable temporaire est utilisée pour sauvegarder le résultat d'une expression. Il faut extraire cette expression dans une méthode et remplacer toutes les références de la variable par l'expression. Dans ces conditions, la nouvelle méthode peut être utilisée par d'autres méthodes dans la classe. Souvent cette refactorisation est utilisée avant d'appliquer la refactorisation *Extract Method* car les variables locales rendent difficile l'application de cette dernière. En effet, le remplacement des variables temporaires par des méthodes est favorisé	√	
Replace Type Code with Class	Si une classe a un code de type numérique ou une énumération qui n'affecte pas son comportement. Remplacer le par une nouvelle classe. L'application de cette refactorisation est favorisée seulement dans le cas ou le type du code est une donnée pure qui ne cause pas des comportements différents à l'intérieur d'une instruction conditionnelle. Dans le langage Java par exemple, l'instruction *switch* permet de faire plusieurs tests de valeurs sur le contenu d'une même variable entière seulement, pas sur une classe arbitraire, ainsi dans ce cas le remplacement n'est pas possible	√	√

Nom de la refactorisation	Description	Code source	Modèle
Replace Type Code with State/Strategy	Cette refactorisation est similaire à la refactorisation *Replace Type Code with subclasses*. Elle est appliquée si le type du code affecte le comportement d'une classe, mais on ne peut pas appliquer une sous classification. Dans ce cas, il faut remplacer ce type par un objet d'état en employant le patron d'état ou de stratégie. Ces derniers sont très semblables. Si on veut simplifier un algorithme simple avec la refactorisation Replace Conditional with Polymorphism, la stratégie est la plus recommandée. Cependant, si on veut déplacer des données d'état spécifiques et l'objet change d'état, le patron d'état est le mieux adopté	√	√
Replace Type Code with Subclasses	Si un code a un type immuable qui affecte le comportement d'une classe. Remplacer le donc par des sous-classes. La présence des structures conditionnelles est l'élément déclencheur pour cette refactorisation. Son avantage est de permettre le déplacement de la connaissance du comportement variable des clients de la classe à la classe elle-même. Si on veut ajouter de nouvelles variantes, il suffit d'ajouter une sous-classe. Cette refactorisation est particulièrement valable quand les variantes continuent à changer	√	√

Nom de la refactorisation	Description	Code source	Modèle
Reverse Conditional (contribuant : Bill Murphy)	Si une instruction conditionnelle serait plus facile à comprendre si on inverse son sens. Inverser donc le sens de la condition et réorganiser à nouveau les clauses de l'instruction conditionnelle	√	
Self Encapsulate Field	Si l'accès à un champ est direct, mais le couplage est faible. Créer alors des méthodes *getting* et *setting* pour le champ et utiliser ces méthodes seulement pour accéder au champ	√	√
Separate Query from Modifier	Si une méthode renvoie une valeur mais change également l'état d'un objet. Dans ce cas, il faut créer deux méthodes, une pour la requête et une autre pour la modification	√	
Split Loop	Si un code comprend une boucle qui fait deux choses à la fois. Il est préférable de dupliquer la boucle et de diviser les tâches. En effet, exécuter la boucle deux fois va doubler le travail et sera mal jugé par la majorité des programmeurs. En optimisation, faire deux choses différentes dans une boucle est moins clair que les faisant séparément	√	
Split Temporary Variable	Si une variable temporaire est assignée plus d'une fois, mais elle n'est pas une variable de boucle ni une collection de variable temporaire. Pour chaque assignement ou responsabilité, il faut produire une variable temporaire séparée. En effet, utiliser une variable temporaire pour deux choses différentes est très confondant pour le lecteur et rend la compréhension plus difficile encore	√	

Nom de la refactorisation	Description	Code source	Modèle
Substitute Algorithm	Si on veut remplacer un algorithme par un autre plus clair. On doit remplacer le corps de la méthode par le nouvel algorithme. Ceci est fréquemment utilisé quand on veut changer un algorithme par un autre algorithme légèrement différent	√	