HAL
open science

Fiabiliser la réutilisation des patrons par une approche orientée complétude, variabilité et généricité des spécifications

Nicolas Arnaud

To cite this version:

Nicolas Arnaud. Fiabiliser la réutilisation des patrons par une approche orientée complétude, variabilité et généricité des spécifications. Autre [cs.OH]. Université Joseph-Fourier - Grenoble I, 2008. Français. NNT : . tel-00331750v3

HAL Id: tel-00331750

https://theses.hal.science/tel-00331750v3

Submitted on 3 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE JOSEPH FOURIER – GRENOBLE I

THESE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER – GRENOBLE I

Discipline : Informatique

Présentée et soutenue publiquement par

Nicolas ARNAUD

le 13 octobre 2008

TITRE

FIABILISER LA RÉUTILISATION DES PATRONS PAR UNE APPROCHE ORIENTÉE COMPLÉTUDE, VARIABILITÉ ET GÉNÉRICITÉ DES SPÉCIFICATIONS

Composition du jury :

Rapporteurs	:	Mme Isabelle BORNE
	:	Mme Régine LALEAU
Présidente	:	Mme Catherine BERRUT
Examinateur	:	Mr Stéphane COULONDRE
Directrice de thèse	:	Mme Dominique RIEU
Co-directrice	:	Mme Agnès FRONT

Thèse préparée au sein de l'équipe SIGMA du laboratoire LIG

(Laboratoire d'Informatique de Grenoble)

Cette première page est l'occasion d'exprimer mes sincères remerciements à celles et ceux qui ont, au fil des années, joué un rôle dans la réalisation de ce travail.

Je commencerai bien entendu par évoquer ma gratitude à l'égard de mes deux directrices de thèse, Mmes Dominique Rieu et Agnès Front, qui, depuis mon année de licence, ont toujours été présentes avec la justesse, l'attention et la bonne humeur qui les caractérisent. Je les remercie pour les relectures souvent tardives et autres petits tracas et leur transmets toute ma reconnaissance et mon amitié.

Je remercie également Mr Jean-Pierre Giraudin, responsable de l'équipe SIGMA pour m'avoir accueilli au sein de son équipe et offert les moyens de mener ce travail à bien. Si l'excellente ambiance qui règne au sein de l'équipe est le fait de tous ses membres, Jean-Pierre en est toutefois le promoteur indiscutable.

Je tiens à exprimer mes sincères remerciements à Mme Isabelle Borne, Professeur à l'Université Bretagne-Sud, ainsi qu'à Mme Régine Laleau, Professeur à l'université Paris XII, pour l'intérêt qu'elles ont porté à ces travaux en acceptant de les rapporter.

Je souhaite également remercier Mme Catherine Berrut, Professeur à l'Université Joseph Fourier, pour avoir accepté de présider le jury. Je remercie également Mr Stéphane Coulondre, Maître de conférences à l'INSA-Lyon, pour avoir accepté de participer au jury.

Je remercie la région Rhône-Alpes pour avoir financé, par l'intermédiaire du projet TRAFIC, la majeure partie de mes années de doctorat. Je tiens également à remercier Mr François Bernigaud pour toutes les choses que j'ai apprises en sa compagnie sur le domaine des Systèmes d'Information Transport.

Sans les différents services administratifs et techniques, il serait difficile de se consacrer comme il se doit à nos travaux. Merci donc à Carole, Christiane, Pascale, Martine, François et Gilles pour leur disponibilité.

Six années passées au LSR/LIG sont autant d'années de rencontres inoubliables. Je pense notamment aux colocataires successifs du bureau 314 : Cathy, Charlotte, Ibtissem, Ouafa, Emmanuel, Nicolas... ainsi qu'à Céline et Marius, que je remercie pour tous les bons moments passés ensemble.

Des rencontres naissent parfois les amitiés. Mes chaleureux remerciements à ceux qui me supportent également à la ville comme au banquet, et que j'ai l'honneur de compter parmi mes amis : David, Jean-Sébastien et Laurent.

Merci à mes parents de m'avoir subtilement détourné d'une carrière prometteuse dans la vente de pizzas à emporter. Et bien sûr, pour tout le reste...

Enfin, mes remerciements les plus tendres vont à celle qui tint la barre si souvent que cette thèse n'aurait pu aboutir sans elle. Quelle année formidable !

Table des matières

Introduction 9

Chapitre 1 Contexte et problématique par un cas d'étude 13

1 Les patrons....14
1.1 Langage/Système/Catalogue de patrons....14
1.2 Formalisme....15

2 Les patrons dans l'ingénierie logicielle....16
2.1 Classification....16
2.2 Patrons de conception et d'analyse....17
2.2.a Patron « Mémoire d'événement »....17
2.2.b Patron « Observateur »....18
2.2.c D'autres patrons....21

3 Usage des patrons....21
3.1 Aide à la conception et génération de code....22
3.2 Validation et rétro-conception....22
3.3 Documentation....23

4 Cas d'étude....23
4.1 Description du cas d'étude....23
4.2 Les spécifications proposées....25
4.2.a Imitation du patron « Mémoire d'événement »....25
4.2.b Imitation du patron « Observateur »....25
4.2.c Imitation du patron « Composite »....26
4.2.d Imitation du patron « Méthode de fabrication »....27

5 Problématique de la thèse....30
5.1 De la complétude des solutions....31
5.2 De la validité des imitations....31
5.3 De la variabilité des solutions....32
5.4 De la composition des imitations....32
5.5 Bilan des objectifs....33

Chapitre 2 Approches et outils existants 37

1 La variabilité dans l'ingénierie logicielle....38

2 Ingénierie Dirigée par les Modèles....39

3 Cadre de comparaison des approches pour les patrons....40
3.1 Critères fonctionnels....41
3.1.a Critères pour l'usage des patrons....41
3.1.b Critères pour la définition de patrons....42
3.2 Critères traitant des solutions....42
3.3 Critères sur la méta-modélisation....43
3.4 Récapitulatif....43

4 Approches et outils....44

4.1 La proposition de l'OMG : Les collaborations....44
4.2 FACE....46
4.3 Travaux de Mel O`Cinneide....48
4.4 Travaux de France et al. : RBML....50
4.5 DPML....54
4.6 Hook & template....57
4.7 Travaux de Albin-Amiot et Guéhéneuc....60
4.8 Travaux de Gerson Sunyé....63
4.9 Autres approches....65

5 Synthèse....66

Chapitre 3 Formalisation des patrons 69

1 Complétude : la solution comme un mini-système....70
1.1 Vue des cas d'utilisation : les fonctionnalités d'un patron....70
1.2 Vue dynamique : diagrammes de séquence UML2....71
1.3 Relations entre vue fonctionnelle et vue dynamique....72
1.4 Vue statique : Diagrammes de classes....73
1.5 Les limites du mini-système....74

2 Variabilité....74
2.1 Représenter la variabilité dans la vue fonctionnelle....75
2.1.a Un opérateur générique pour la variabilité....76
2.1.b Application de l'opérateur générique à la propriété obligatoire/facultatif....77
2.2 Le mini-système à variantes....78
2.2.a Variabilité dynamique : concepts....78
2.2.b Variabilité dynamique : règle de traduction....80
2.2.c Variabilité dynamique : une autre représentation avec UML2....84
2.2.d Variabilité statique....85
2.3 Variabilité : extensions d'UML....86

3 Généricité....87
3.1 Précision sur le nommage des éléments....88
3.2 Cadre de définition....88
3.2.a Relation entre généricité et variabilité....89
3.2.b Avantages du cadre de définition....90
3.3 Principes et Exemples....90
3.3.a Propriétés génériques....90
3.3.b Mise en œuvre dans le cadre de définition....91
3.3.c Contraintes lors de l'imitation....93
3.4 Abstraction et Visibilité....94
3.4.a Abstraction....94
3.4.b Visibilité....95
3.5 Une autre forme de généricité : les notes « A Faire »....96

4 Processus de spécification....97

5 Application : Patron Composite....98

6 Synthèse....102

Chapitre 4 Processus d'imitation de patrons 105

1 Le processus d'imitation....106
1.1 Terminologie et définitions....106

1.2 Processus de réduction....107
1.2.a Choix des variantes et réduction fonctionnelle....107
1.2.b Réduction des vues dynamique et statique....109
1.3 Processus d'application....111
1.3.a Adaptation....111
1.3.b Résolution....111
1.3.c Composition....112

2 Mise en œuvre....113
2.1 Un premier prototype d'imitation/validation....113
2.2 Ingénierie Dirigée par les Modèles : quelques outils....115
2.3 Nouveau prototype : architecture générale....115
2.4 Utilisation du nouveau prototype....118

3 Synthèse....120

Chapitre 5 Travaux connexes : réflexions sur la composition 123

1 Composition d'imitations : deux opérateurs....124

2 Composants et opérateurs....124
2.1 Le modèle de composants Symphony....124
2.2 Encapsuler une imitation dans un composant....126
2.3 Composition par délégation....127
2.4 Composition par fusion....128
2.4.a Principes....128
2.4.b Application au cas d'étude....129
2.5 Vues d'un système....130

3 Synthèse....131

Conclusion générale et Perspectives 133

1 Rappel des propositions....133

2 Perspectives....135

Annexes 139

1 Patron Composite....140

2 Patron Méthode de Fabrication....142

3 Patron Singleton....144

4 Patron Fabrique Abstraite....145

5 Méta-modèle UMLpi....148

Bibliographie 149

Illustrations et Tableaux

Figure 1.1 : Catalogue du système de patrons du GoF. Figure extraite de (Gamma et al., 1995)...19
Figure 1.2 : Classification des patrons (Conte et al., 2001)....21
Figure 1.3 : Exemple d'imitation du patron « Composite »....26
Figure 1.4 : Données et exemples des modes d'affichage....28
Figure 1.5 : Cas d'utilisation du système....28
Figure 1.6 : Une imitation de « Mémoire d'événement » - Mesure/Capteur....29
Figure 1.7 : Imitation du patron « Observateur » et composition avec l'imitation de « Mémoire d'événement »....30
Figure 1.8 : Imitation du patron « Composite » et composition avec la spécification précédente..31
Figure 1.9 : Imitation du patron « Méthode de fabrication » (première version)....32
Figure 1.10 : Imitation du patron « Singleton » (première version)....33
Figure 1.11 : Imitation du patron « Méthode de fabrication » (deuxième version)....34
Figure 1.12 : Imitation du patron « Singleton » (deuxième version)....34
Figure 1.13 : Récapitulatif des patrons imités....37
Figure 2.1 : Concepts et relations de l'IDM....44
Figure 2.2 : Utilisation des collaborations génériques UML pour la représentation du patron « Composite » (extrait de (Booch et al., 2005))....49
Figure 2.3 : FACE - Principe de fonctionnement....50
Figure 2.4 : FACE – Métaschéma, schéma solution et schéma instance pour le patron « Observateur »....51
Figure 2.5 : Processus d'imitation proposé par (O`Cinneide, 2001)....53
Figure 2.6 : Représentation de la Méta-entité « Rôle » (extrait de (France et al., 2004))....55
Figure 2.7 :SPS et méta-modèle partiel du patron « Observateur » (extrait de (France et al., 2004))....55
Figure 2.8 : Mise en correspondance d'un SPS et d'une imitation (extrait de (France et al., 2004))....56
Figure 2.9 : IPS et méta-modèle partiels pour le patron « Observateur »....57
Figure 2.10 : Illustration du principe de dimension avec le patron « FabriqueAbstraite »....59
Figure 2.11 : Diagramme de spécification du patron « Fabrique Abstraite » avec DPML....60
Figure 2.12 : Imitation du patron « Fabrique abstraite » avec DPML. (extrait de (Maplesden et al., 2002)) 60
Figure 2.13 : Méta-modèle pour la définition des patrons de conception (extrait de(Pagel et Winter, 1996))....62
Figure 2.14 : Le patron « Composite » et une imitation (extrait de (Pagel et Winter, 1996))....62
Figure 2.15: Méta-modèle pour les patrons, figure extraite de (Albin-Amiot et al., 2002)....64
Figure 2.16 : Processus d'instanciation de patron (extrait de (Albin-Amiot et al., 2002))....65
Figure 2.17 : Extrait de la déclaration de la méta-entité « Composite » (extrait de (Albin-Amiot et al., 2002))....66
Figure 2.18 : Choix des compromis d'implémentation pour l'imitation du patron « Observateur » (extrait de (Sunye, 1999))....68
Figure 3.1 : Observateur : vue des cas d'utilisation....75
Figure 3.2 : Observateur : diagramme de séquence du GoF....75

Figure 3.3 : Observateur : diagrammes de séquences avec UML2....76
Figure 3.4 : Observateur : nouvelle vue fonctionnelle....77
Figure 3.5: Observateur : vue statique....78
Figure 3.6 : Vue fonctionnelle à variantes du patron « Observateur »....80
Figure 3.7 : Exemple d'application de l'opérateur générique pour la variabilité fonctionnelle....81
Figure 3.8 : Exemple d'application de l'opérateur générique pour la variabilité fonctionnelle en utilisant une syntaxe simplifiée....81
Figure 3.9 : Vue fonctionnelle à variantes du patron « Observateur »....82
Figure 3.10 : Méta-modèle UML2 : Interactions....84
Figure 3.11 : Traduction dynamique d'une inclusion fonctionnelle classique....84
Figure 3.12 : Traduction dynamique d'un point de variation et ses variantes alternatives....85
Figure 3.13 : Traduction dynamique d'un point de variation avec ses options....85
Figure 3.14 : Traduction dynamique d'une racine....86
Figure 3.15 : Traduction dynamique d'un point de variation regroupant alternatives et options…87
Figure 3.16 : Observateur : vue dynamique à variantes....88
Figure 3.17 : Patron « Observateur » : vue dynamique, notation simplifiée....89
Figure 3.18 : Fragments statiques pour le patron « Observateur »....90
Figure 3.19 : Variabilité : extensions d'UML....91
Figure 3.20 : Cadre de définition des propriétés génériques....93
Figure 3.21 : Extension de la méta-entité ElementImitable pour prendre en compte le nombre d'occurrences et les dimensions....96
Figure 3.22 : Patron « Fabrique abstraite » : dimension et nombre d'occurrences....97
Figure 3.23 : Nouvelles propriétés génériques : abstraction et visibilité....100
Figure 3.24 : Utilisation des notes "A Faire" dans le patron « Observateur »....101
Figure 3.25 : Processus de spécification de la solution d'un patron....102
Figure 3.26: Diagramme de classes original du patron « Composite »....103
Figure 3.27 : Patron « Composite » : ébauche de vue fonctionnelle....103
Figure 3.28 : Patron « Composite » : augmentation de la vue fonctionnelle....104
Figure 3.29 : Patron « Composite » : quelques fragments dynamiques et statiques....104
Figure 3.30 : Patron « Composite » : alternatives « transparence » et « sécurité »....105
Figure 3.31 : Patron « Composite » : Ajouter un composant, vue dynamique et statique....106
Figure 4.1 : Processus d'imitation....111
Figure 4.2 : Sélection des variantes pour le patron « Observateur »....112
Figure 4.3 : Patron « Observateur » : Une réduction fonctionnelle....113
Figure 4.4: Patron « Observateur » : Une réduction dynamique....113
Figure 4.5: Patron « Observateur » : Une réduction statique....114
Figure 4.6 : Patron « Observateur » : Réduction statique, diagramme d'objets....115
Figure 4.7 : Récapitulatif des patrons imités (tiré du cas d'étude)....116
Figure 4.8: Saisie des propriétés génériques du patron « Observateur »....118
Figure 4.9 : Adaptation et validation d'une imitation du patron « Observateur » à l'aide du premier prototype....118
Figure 4.10 : architecture de mise en œuvre....120
Figure 4.11 : Méta-modèle UMLpiBind....121
Figure 4.12: Modèle imitable, correspondance et système cible : un exemple....122

Figure 4.13 : Modèle imitable du patron « Observateur », modèle de correspondance et modèle du système cible....123
Figure 4.14 : Résultat d'une transformation de type τ3....124
Figure 5.1: Illustration du modèle à composant Symphony....129
Figure 5.2 : Structure du patron « Singleton » et imitation encapsulée : le composant SystèmeDeDétection....130
Figure 5.3 : Encapsulation d'une imitation du patron « Observateur » : le composant Détecteur. 131
Figure 5.4 : Encapsulation d'une imitation du patron « Composite » : le composant Structure...131
Figure 5.5 : Application de l'opérateur de délégation au cas d'étude....132
Figure 5.6 : Principes d'utilisation de l'opérateur de fusion....133
Figure 5.7: Fusion entre les imitations des patrons « Mémoire d'événement » et « Observateur ». 134
Figure 5.8 : Vue globale d'un système composé....134

Tableau 1.1 : Description du patron « Mémoire d'événement »....22
Tableau 1.2 : Description du patron « Observateur »....25
Tableau 2.1 : Récapitulatif des critères de comparaison des approches....48
Tableau 2.2 : Récapitulatif des collaborations UML....50
Tableau 2.3 : Récapitulatif de FACE....52
Tableau 2.4 : Récapitulatif des travaux de Mel O`Cinneide....54
Tableau 2.5 : Récapitulatif de (France et al., 2004)....58
Tableau 2.6 : Récapitulatif de DPML....61
Tableau 2.7 : Récapitulatif de « Hook & Template »....64
Tableau 2.8 : Récapitulatif de l'approche de (Albin-Amiot et al., 2002)....67
Tableau 2.9 : Récapitulatif des travaux de Gerson Sunyé....69

INTRODUCTION

L'exigence de qualité des systèmes d'information implique rigueur et continuité dans les différentes phases de développement. Il est donc crucial de capitaliser les savoirs et les savoirs-faire afin de les réutiliser au cours d'autres processus de développement. Nous considérons la réutilisation comme un gage de cette qualité, et particulièrement si elle met en œuvre une traçabilité modulaire des spécifications.

Si le développement complet d'un système à base de réutilisation est un vœu pieu, il n'en reste pas moins que le principe de réutilisation est applicable de la phase d'expression des besoins à la phase de recette quelles que soient les techniques de modélisation et/ou d'implémentation. Les deux caractéristiques fondamentales d'un « fragment » de spécification réutilisable sont la complétude et l'adaptabilité : le fragment se suffit à lui-même, de façon à pouvoir être documenté efficacement, et fournit clairement des modes d'adaptation au contexte de réutilisation. Les patrons sont, sous cet éclairage, particulièrement pertinents.

Un patron décrit un problème fréquemment rencontré dans un contexte ainsi que la solution consensuelle qui le résout. Dans le domaine des systèmes informatiques et pour ce qui nous intéresse dans celui de la conception de systèmes d'information, on peut bien évidemment citer les patrons de conception et parmi les plus connus ceux du Gang of Four (GoF) et de Peter Coad, qui nous serviront à la fois de référence et d'exemple tout au long de ce document.

Notre objectif est de proposer un processus fiable de réutilisation d'un patron, que nous appelons *processus d'imitation*, et qui permet d'extraire la solution du patron et de l'appliquer dans un système en construction. Cependant, il n'y a pas de bonne imitation s'il n'y a pas de bonne spécification de la solution. Une bonne spécification est avant tout une spécification complète qui, dans le monde orienté-objet, ne peut être atteinte en recourant seulement à la modélisation statique (diagrammes de classes). C'est pourquoi nous pensons qu'il est judicieux, à l'instar d'un système d'information classique, de prendre en compte plusieurs aspects de la solution. Dans ce

travail, nous tirons partie de trois vues complémentaires : la vue fonctionnelle, la vue dynamique et la vue statique.

De plus, rendre sa solution semi-formelle plus complète n'est pas suffisant pour exprimer tous les apports d'un patron. Dans de nombreux cas, et particulièrement pour les patrons destinés aux phases d'analyse ou de conception, la description du patron est clairsemée d'informations exprimant des variantes possibles de cette solution, et ceci selon tous les aspects : fonctionnels, dynamiques ou statiques.

Enfin, si la variabilité permet d'exprimer des solutions différentes, ou, nous le verrons, des problèmes différents, il n'en reste pas moins qu'une solution doit pouvoir être adaptée finement pour correspondre à certains besoins spécifiques du contexte d'imitation.

Nos propositions ont donc pour objectif de donner une spécification plus opérationnelle à la solution originellement offerte par un patron. Dans ce sens, le processus d'imitation que nous proposons garantit que l'imitation obtenue reste conforme à la solution. Pour autant, cette spécification opérationnelle ne se substitue pas aux solutions originelles qui restent un excellent support didactique et facilitent d'autant la compréhension du patron.

Deux aspects complémentaires sont particulièrement traités dans notre approche : variabilité et généricité des spécifications. En effet, ces dernières prennent en compte certaines variantes auparavant décrites de manière textuelles tout en proposant une description plus générique de certaines propriétés de manière à affiner l'adaptabilité de la solution.

S'agissant de la définition des patrons ou de leur usage, nous partons du principe qu'il faut, pour que notre approche soit réellement utilisable, qu'elle soit le moins intrusive possible. C'est pourquoi nous avons choisi d'étendre les langages de modélisation plutôt que d'en créer de nouveaux. Plus particulièrement, puisque nous limitant aux patrons orientés-objets destinés au génie logiciel, nous utilisons les concepts de la méta-modélisation pour étendre le langage UML et y intégrons les notions de variabilité et de généricité. Nous proposons également deux processus pour supporter la spécification et l'imitation de ces « nouvelles » solutions de patrons.

Le premier chapitre de ce mémoire débute par quelques rappels sur la notion de patrons et les concepts associés. Par la suite, nous introduisons un cas d'étude sous la forme du déroulement de la phase d'analyse/conception d'un système de détection d'activité sismique. Ce cas d'étude nous permet d'illustrer la problématique de cette thèse en terme de complétude, variabilité et généricité des solutions de patrons.

Le second chapitre est dédié à l'état de l'art sur la variabilité dans le génie logiciel, l'ingénierie dirigée par les modèles et les approches d'aide à la conception à l'aide de patrons. Une fois un cadre de comparaison défini, chaque approche est décrite puis replacée dans ce cadre. Cette comparaison démontre en particulier que contrairement aux aspects génériques, la variabilité, et dans une moindre mesure la complétude, ont rarement été traitées dans les approches à patrons.

Les constatations des deux premiers chapitres ouvrent la voie d'un troisième détaillant nos propositions en vue d'améliorer la spécification des solutions de patrons à travers les trois leviers que sont la complétude, la variabilité et la généricité. Nous y proposons également un processus de spécification destiné à assister les ingénieurs de patrons dans leur activité.

Si le chapitre trois insiste sur le contenu des solutions de patrons et sur la méthodologie à adopter pour sa spécification, le chapitre quatre précise quant à lui la manière d'en tirer partie lors de l'imitation. Nous proposons aux ingénieurs d'applications un processus de réutilisation qui, à partir d'une solution de patron spécifiée selon les recommandations précédentes, lui permet de produire une imitation en tenant plus précisément compte de son problème (traitement de la variabilité) et en en garantissant la validité dans le temps (traitement de la généricité). Dans un deuxième temps, nous présentons deux prototypes (l'un axé variabilité, l'autre axé généricité) basés sur les principes de l'ingénierie dirigée par les modèles.

Enfin, nous présentons dans un cinquième et dernier chapitre, une approche de composition d'imitations basée sur des concepts issus du monde des composants métier. Cette approche distingue deux types de composition : la « délégation » et la « fusion », illustrées à l'aide du cas d'étude.

Cette thèse a été en partie financée grâce au projet TRAFIC (TRAFIC, 2008), soutenu par la région Rhône-Alpes. Une des tâches de ce projet était de donner, en collaboration avec la société ACTOLL (ACTOLL, 2008) spécialiste des systèmes d'information dans le domaine du transport, une description du domaine de la billettique (dans le transport de personne) sous la forme de composants métier. De là est née l'idée d'appliquer nos propositions à d'autres types d'objets réutilisables, comme par exemple les composants métier.

CONTEXTE	***Chapitre 1 : Contexte et problématique par un cas d'étude***	Analyse et conception à l'aide de patrons : intérêts et problèmes rencontrés sur un cas d'étude.
	Chapitre 2 : Approches et outils existants	État de l'art sur la variabilité, l'IDM et les approches pour la conception à l'aide de patrons.
PROPOSITIONS	***Chapitre 3 : Formalisation des patrons***	Complétude, variabilité et généricité des spécifications des solutions de patrons : concepts, méta-modèle et méthodologie pour les ingénieurs de patrons.
	Chapitre 4 : Processus d'imitation	Réutilisation des patrons par les ingénieurs d'applications : processus et prototypes pour l'imitation des spécifications proposées dans le chapitre trois.
TRAVAUX CONNEXES	***Chapitre 5 : Travaux connexes : réflexions sur la composition***	Proposition pour la composition d'imitations.

CHAPITRE 1 CONTEXTE ET PROBLÉMATIQUE PAR UN CAS D'ÉTUDE

Dans ce chapitre, nous présentons dans un premier temps quelques généralités sur les patrons et illustrons quelques patrons de Coad (Coad, 1995) et du GoF (Gamma et al., 1995) tels qu'ils ont été décrits originellement. Nous décrivons ensuite la spécification d'un système par un ingénieur d'applications maîtrisant les patrons précédemment décrits. Nous revenons enfin sur les différentes étapes de ce travail en insistant sur les points qui peuvent (ou auraient pu) amener l'ingénieur d'applications à faire des erreurs ou à un questionnement délicat. Cette discussion permet de dégager la problématique de nos travaux, et de fournir un cas d'étude permettant, dans la suite de cette thèse, d'illustrer nos propositions.

1 LES PATRONS

Au cours des années soixante-dix, l'architecte Christopher Alexander (Alexander, 1979), (Alexander et al., 1977) s'est interrogé sur la qualité des constructions en cherchant à déterminer des critères irréfutables garantissant une architecture de qualité. En étudiant divers bâtiments, il a pu déterminer que les bonnes solutions à une même sorte de problèmes architecturaux ont souvent des points communs qui peuvent être décrits au travers d'objets destinés à servir de support pour la réalisation d'autres constructions similaires : les patrons. C. Alexander en donne la définition suivante (Alexander, 1979) :

> *« Chaque patron décrit à la fois un problème qui se produit très fréquemment dans votre environnement et l'architecture de la solution à ce problème de telle façon que vous puissiez utiliser cette solution des millions de fois sans jamais l'adapter deux fois de la même manière ».*

Les patrons peuvent comporter deux types de connaissance. Les patrons qui visent à donner la spécification des artefacts constituants d'une solution au problème sont appelés des patrons « produit ». Les patrons dits « processus » présentent un savoir-faire sous forme d'actions et/ou de tâches à suivre pour la résolution du problème. Plus simplement, les patrons produit traitent du savoir tandis que les patrons processus traitent du savoir-faire.

Un patron est plus communément défini comme une **solution** consensuelle à un **problème** récurrent dans un **contexte** où :

- le **contexte** se réfère à un ensemble de situations récurrentes,
- le **problème** définit un but à atteindre,
- la **solution** décrit un « quoi » ou un « comment » permettant de résoudre ce problème, en l'adaptant à un contexte particulier.

1.1 Langage/Système/Catalogue de patrons

Dans « A Pattern Language » (Alexander et al., 1977), C. Alexander et ses collègues mettaient en avant la notion de collection de patrons. On distingue trois types principaux de collection de patrons : les langages, les systèmes et les catalogues de patrons (Buschmann et al., 1996).

D'après The Hillside Group, *« un langage de patron définit une collection de patrons ainsi que les règles pour les combiner »* (The Hillside Group, 2007). Les langages de patrons décrivent des cadriciels ou des familles de systèmes. Ils ont pour objectif d'aider celui qui les utilise, au cours d'une démarche d'ingénierie plus ou moins bien définie, à construire une solution pour son problème complexe d'ingénierie et se composent donc de patrons produits et processus. La définition d'un processus d'ingénierie comme un support pour la construction de systèmes appartenant à la même famille est particulièrement proche de la vision qu'avait Alexander de ses langages de patrons qui devaient permettre à un novice en architecture de concevoir un bâtiment.

Les langages de patrons ont un objectif de complétude dont les systèmes de patrons sont exempts car non destinés à la construction intégrale d'un système. Néanmoins, si les systèmes de patrons ne peuvent être complets, ils n'en restent pas moins détaillés et cohérents, et présentent

un ensemble de patrons ainsi que des relations de combinaison, composition, raffinement, etc. Dans la pratique, les systèmes de patrons sont principalement des recueils de bonnes pratiques dans un domaine d'application et/ou pour une technologie donnée.

En fonction de leur formalisme, certaines collections de patrons peuvent être perçues comme des catalogues de patrons. Un catalogue de patrons est un système de patrons dont certaines rubriques du formalisme servent à une classification interne des patrons du système. Ainsi, la classification portée/but du célèbre système de patrons du GoF[1] peut être représentée grâce au catalogue de la figure 1.1.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Figure 1.1 : Catalogue du système de patrons du GoF. Figure extraite de (Gamma et al., 1995).

1.2 Formalisme

Tout patron peut être défini par un triplet "*problème, contexte, solution*", qui exprime une relation entre un problème et sa solution dans un contexte donné. Pour représenter ce triplet, de nombreux formalismes ont été proposés. Nous entendons par formalisme la structure adoptée par le concepteur de patrons pour représenter des patrons. On peut distinguer deux classes de formalismes : les formalismes narratifs tels que ceux de C. Alexander (Alexander, 1979) et ceux du Portland Pattern Repository (Portland, 2007), et les formalismes structurés tels que ceux de M. Fowler (Fowler, 1997) et de P. Coad (Coad, 1995) pour les patrons d'analyse, du GoF (Gamma et al., 1995) pour les patrons de conception, ou de S.W. Ambler (Ambler, 1999) pour les patrons processus. Par opposition aux formalismes narratifs par nature peu structurés et informels, le second type de formalisme offre une meilleure représentation des patrons. Ces formalismes structurés, composés chacun d'un ensemble de rubriques qui leur est propre, sont globalement équivalents dans la mesure où ils expriment tous le triplet "*problème, contexte, solution*". Ils diffèrent cependant par le nombre de rubriques proposées et le degré de détail de celles-ci.

La plupart des formalismes de représentation sont dédiés soit à l'expression des patrons produit en mettant l'accent sur la représentation des solutions modèles (c'est le cas par exemple des formalismes du GoF et de Coad), soit à l'expression des patrons processus en mettant l'accent sur la représentation des solutions démarches (c'est le cas du formalisme de Ambler). Néanmoins,

1 GoF est l'abréviation pour « Gang of Four » : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. (Gamma et al., 1995).

certains formalisme comme P-SIGMA (Conte et al., 2001) permettent de combiner, au sein d'un même système, patrons produits et processus. Ces représentations mixtes sont particulièrement intéressantes pour la description de démarches de développement puisqu'elles permettent de décrire les activités et les artefacts de manière homogène.

2 Les patrons dans l'ingénierie logicielle

Les premières applications du concept de patron au domaine du génie logiciel datent des années quatre-vingt. En particulier, dans le domaine de la programmation orientée objet, Kent Beck et Ward Cunningham posent à la conférence OOPSLA'87 les jalons d'un langage de patrons destiné aux ingénieurs d'applications programmant en SmallTalk (Beck et Cunningham, 1987). Même si la distinction entre patrons produit et patrons processus n'y est pas explicite, ce langage se situe entre un recueil de bonnes pratiques et une démarche de développement.

C'est au début des années 90 que la notion de patron dans le génie logiciel prend de l'ampleur. En marge de la conférence OOPSLA'92 se forme le « Gang of Four » à l'origine du plus fameux recueil de patrons de conception orientée objet (Gamma et al., 1995). Dans le même temps, James O. Coplien catalogue des « idiomes » de programmation spécifiques au langage C++ (Coplien, 1992) et initialise la collection « Software Patterns Series » (Coplien et al., 1995) (Coplien et al., 1996).

Au début des années 90, Peter Coad distingue patrons de conception et patrons d'analyse (Coad, 1992) et publie, en 1995, un livre regroupant des « Stratégies » – assimilables à des patrons processus – ainsi que des patrons orientés objet applicables pendant la phase d'analyse (Coad, 1995).

Plus récemment, Michael Mahemoff a publié un livre consacré à des patrons de conception dans la technologie AJAX (Michael Mahemoff, 2006) et a mis en ligne un site web collaboratif dédié à ce sujet (AjaxPatterns, 2007).

Les exemples précédents concernent principalement des patrons produit, mais certains patrons sont orientés processus. Ainsi, Scott. W. Ambler propose des patrons processus généraux couvrant un cycle de développement de logiciels, dans la technologie orientée-objet (Ambler, 1998) (Ambler, 1999). Les patrons proposés par L. Gzara (Gzara, 2000) sont, eux, spécifiques à un domaine : les systèmes d'information produit. Ils en couvrent les étapes d'analyse et de conception en combinant patrons produit et patrons processus.

Le génie logiciel n'est pas le seul domaine à avoir vu un intérêt à l'utilisation des patrons. De nombreux travaux sont menés à ce sujet dans le domaine de l'interaction homme-machine (Borchers, 2000) (Welie et Gerrir, 2003) (Tarpin-Bernard et al., 1998) ou encore dans celui des applications temps-réel (Konrad et Cheng, 2005).

2.1 Classification

La classification proposée par (Conte et al., 2001) est originalement destinée aux composants en général. Néanmoins, les auteurs précisent que trois critères permettent de classifier les patrons :

- le type de connaissances : il peut s'agir de capitaliser des spécifications ou des implantations de produit – un produit correspondant au but à atteindre - ou de capitaliser des spécifications ou des implantations de processus – un processus correspondant au chemin à parcourir pour atteindre le résultat.
- la couverture : il peut s'agir de patrons généraux (resp. domaine, entreprise) si le problème traité est fréquent dans de nombreux domaines d'applications (resp. dans un domaine d'application, dans une entreprise particulière).
- la portée : la portée d'un patron est évaluée en fonction de l'étape d'ingénierie (analyse, conception, implantation, etc..) à laquelle le composant s'adresse. Ce critère est assimilable au critère de granularité de (Buschmann et al., 1996).

La figure 1.2 place les différents systèmes de patrons du génie logiciel énoncés plus haut dans l'espace défini par ces trois critères.

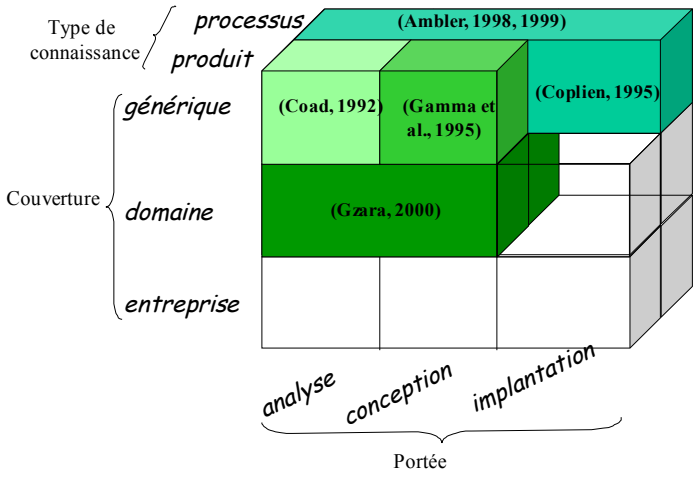


Figure 1.2 : Classification des patrons (Conte et al., 2001).

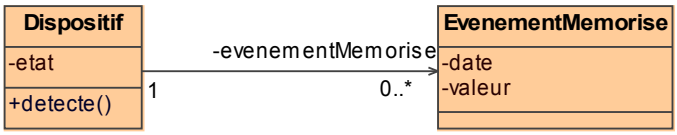
2.2 Patrons de conception et d'analyse

UML n'existant pas encore, les patrons orientés-objet proposés au début des années 90 décrivaient leurs solutions dans des langages comme OMT (i.e. les patrons du GoF) ou dans une notation propre (i.e. les patrons de Coad). Dans cette section, comme tout au long de cette thèse, nous présentons des traductions en UML des solutions initialement proposées.

2.2.a Patron « Mémoire d'événement »

Le patron « Mémoire d'événement » (event logging) est décrit ci-dessous dans le formalisme de Coad, constitué de cinq rubriques. Comme tous les patrons de la collection, ce patron produit est générique et sa portée se situe dans la phase d'analyse.

Nom : Mémoire d'événement
Solution modèle :



Le patron :

Un objet « dispositif » surveille un dispositif externe, l'objet est responsable de la détection d'un événement qui est survenu ; l'objet est également responsable du déclenchement de la réponse à l'événement. Une partie de la réponse peut être de mémoriser l'occurrence de l'événement. Lorsque cela est nécessaire, l'objet « dispositif » envoie un message de création à la classe « événement mémorisé » pour créer une nouvelle instance de cette classe contenant des valeurs chronologiques. Le dispositif doit connaître les objets « événement mémorisé ».

Exemple :

Un objet *SondeThermique* surveille la température courante d'une sonde dans le but de détecter des violations de seuil. Pour faire son travail, l'objet *SondeThermique* doit connaître ses états opérationnels et son seuil. Lorsqu'il détecte qu'une violation de seuil est survenue, il envoie un message à la classe *ViolationSeuil* pour créer une nouvelle instance de cette classe avec les valeurs de date et d'heure, la valeur mesurée et le seuil de détection.

Directives d'utilisation :

A utiliser quand un événement est détecté et que vous devez enregistrer son occurrence dans le but d'en faire une analyse a posteriori.

Tableau 1.1 : Description du patron « Mémoire d'événement ».

2.2.b Patron « Observateur »

Onze des treize rubriques du formalisme de (Gamma et al., 1995) sont utilisées ci-dessous dans la traduction du patron « Observateur » des mêmes auteurs.

Nom : Observateur

Classification : Objet – Comportemental

Intention : Définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour.

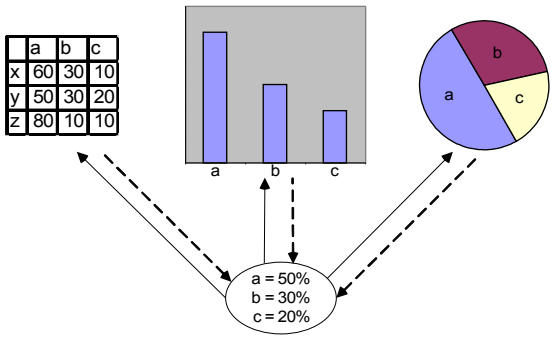
Alias : Dépendants, Diffusion – Souscription

Motivation :

Un effet couramment induit par le partitionnement d'un système en une collection de classes coopérantes est l'obligation de maintenir la cohérence des objets en relation. Il n'est pas souhaitable d'obtenir cette cohérence au prix d'un couplage étroit entre les classes, car cela réduirait leur réutilisabilité.

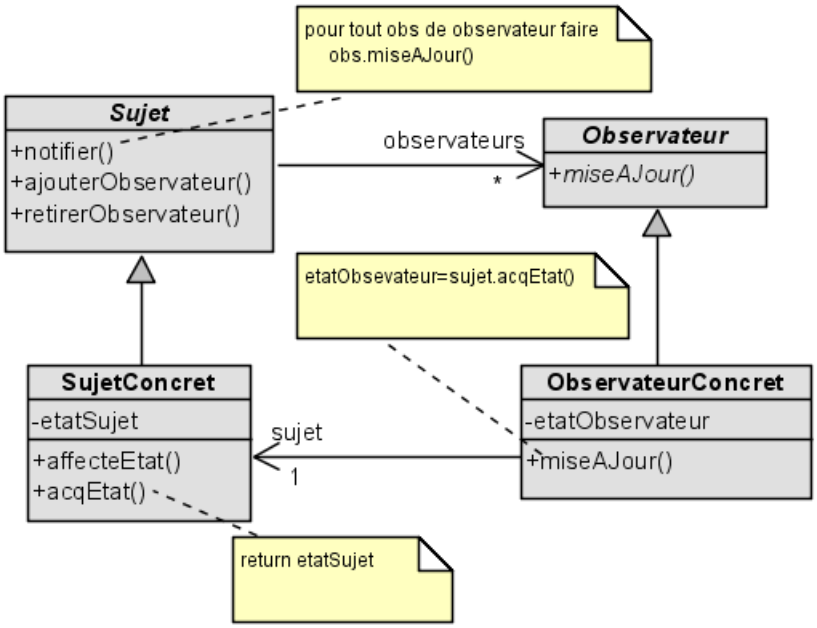
Par exemple, de nombreux outils d'interface graphique utilisateurs séparent les aspects de présentation de l'interface utilisateur des données sous-jacentes de l'application. Les classes qui définissent les données d'application et les présentations peuvent être réutilisées indépendamment. Elles peuvent également travailler ensemble. Un objet tableur, un objet diagramme en cheminées d'usine, peuvent tous deux décrire l'information du même objet donné de l'application, avec des représentations différentes. Le tableur et le diagramme s'ignorent mutuellement, laissant toute latitude de choix de réutilisation. Mais ils se comportent comme s'ils se connaissaient. Quand l'utilisateur modifie l'information dans le tableur, le diagramme fait immédiatement apparaître les modifications et vice versa.

[…]



[…]

Structure :



Constituants :

- **Sujet** : il connait ses observateurs. Un nombre quelconque d'observateurs peut observer un sujet. Il fournit une interface pour attacher et détacher les objets observateurs.
- **Observateur** : il définit une interface de mise à jour pour les objets qui doivent être notifiés des changements du sujet.
- **SujetConcret** : il mémorise les états qui intéressent les objets ObservateurConcret. Il envoie une notification à ses observateurs lorsqu'il change d'état.
- **ObservateurConcret** : il gère une référence sur un objet SujetConcret. Il mémorise l'état qui doit rester pertinent pour le sujet. Il fait l'implémentation de l'interface de mise à jour de

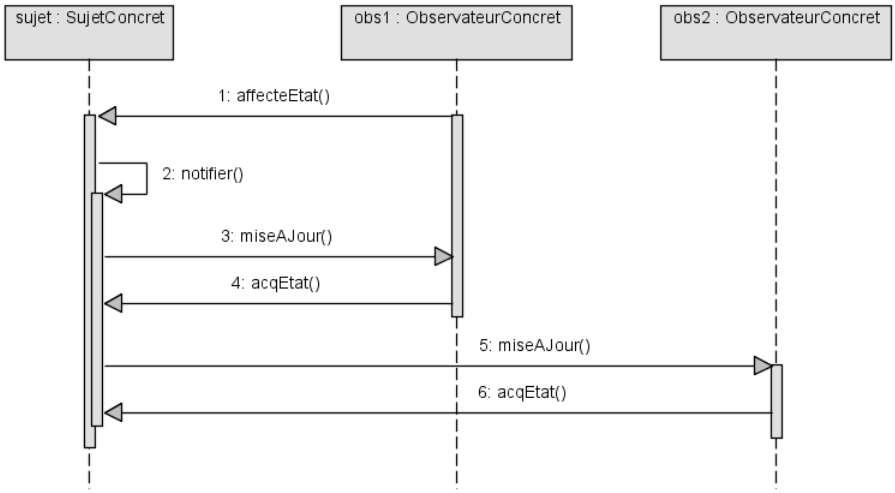
l'Observateur pour conserver la cohérence de son état avec le sujet.

Indications d'utilisation :

On utilisera le modèle observateur dans les situations suivantes :

- quand un concept a deux représentations, l'une dépendant de l'autre. Encapsuler ces deux représentations dans des objets distincts permet de les réutiliser et de les modifier indépendamment.
- quand la modification d'un objet nécessite de modifier les autres, et que l'on ne sait pas combien sont ces autres.
- quand un objet doit être capable de faire une notification à d'autres objets sans faire d'hypothèse sur la nature de ces objets. En d'autres termes, quand ces objets ne doivent pas être trop fortement couplés.

Collaborations :



- Le SujetConcret notifie ses observateurs de tout changement se produisant qui pourrait rendre l'état de ses observateurs incompatible avec le sien propre.
- Après avoir été informé d'un changement dans le sujet concret, un objet ObservateurConcret peut faire une demande d'information au sujet. L'ObservateurConcret utilise ces informations pour mettre son état en conformité avec celui du sujet.

Conséquences :

Le modèle Observateur permet de modifier les sujets et les observateurs indépendamment. On peut réutiliser les sujets sans réutiliser les observateurs, et réciproquement. On peut ajouter un nouvel observateur sans avoir à modifier les autres et sans avoir à modifier le sujet.

Au nombre des autres avantages et contingences du modèle Observateur, citons les suivants :

- isolation du couplage entre Sujet et Observateur. Tout ce que sait un sujet est qu'il possède une liste des observateurs, chacun de ceux-ci se conformant à l'interface simple de la classe abstraite Observateur. Le sujet ne connaît aucune classe concrète d'observateur. De ce fait, le couplage entre sujets et observateurs est abstrait et minimal.
- support de diffusion. A la différence d'une requête ordinaire, la notification émise par un sujet n'a pas à spécifier ses destinataires. La notification est automatiquement diffusée à tous les objets qui ont souscrit. […]
- mises à jour inopinées. Du fait que les observateurs ignorent la présence les uns des autres, ils peuvent être complètement aveugles au coût final d'une modification de sujet. Une opération sur le sujet, en apparence inoffensive, peut engendrer une cascade de mises à

jour sur les observateurs et sur les objets qui en dépendent.
Implémentation : - *Qui déclenche la mise à jour ?* 1. Disposer dans Sujet d'opérations de positionnement d'état qui appellent *notifier()* après avoir changé l'état du sujet. L'avantage de cette approche est de soulager les clients d'avoir à se souvenir d'appeler *notifier()* sur le sujet. Dans le cas de plusieurs opérations successives, il y aurait autant de mises à jour, ce qui peut être inutile et inefficace. 2. Confier au client la responsabilité d'invoquer *notifier()* au moment adéquat. L'avantage, ici, est que le client peut attendre pour déclencher une mise à jour jusqu'à ce qu'une série de changements d'état ait eu lieu, évitant ainsi des mises à jour intermédiaires inutiles. Le désavantage est qu'il échoit au client la responsabilité supplémentaire d'avoir à déclencher la mise à jour. Ceci rend les erreurs plus probables, le client risquant d'oublier d'invoquer *notifier()*. - *Protocoles spécifiques de mise à jour.* 1. D'un coté c'est le « modèle push » : le sujet envoie aux observateurs une information détaillée sur les modifications, que ceux-ci en aient besoin ou non. 2. De l'autre c'est le « modèle pull » : le sujet n'envoie rien sauf l'indication strictement minimale, et les observateurs réclament ensuite explicitement les détails à fournir […] - *Références en l'air aux sujets détruits*. La suppression d'un sujet ne doit pas laisser de références périmées dans ses observateurs. - *Observation de plusieurs sujets*. Dans certaines situations, il peut être intéressant pour un observateur de dépendre de plusieurs sujets. - *S'assurer avant notification, que l'état du sujet est adéquat*. Il est important de s'assurer que l'état du sujet a un sens avant d'appeler *notifier()*. [...]

Tableau 1.2 : Description du patron « Observateur ».

2.2.c D'autres patrons

Les patrons « Composite » et « Fabrique Abstraite », « Singleton » et « Méthode de Fabrication » du GoF sont donnés en annexe.

3 Usage des patrons

D'une manière générale, on peut identifier deux types d'acteurs qui manipulent les patrons : l'ingénieur de patrons qui les définit et l'ingénieur d'applications qui les utilise.

L'ingénieur de patrons a pour objectif de concevoir de nouveaux patrons par identification de problèmes récurrents et de solutions à ces problèmes. Ces patrons doivent ensuite être intégrés dans une collection et être reliés aux patrons qui la composent. Pour mener à bien cette activité, l'ingénieur de patrons peut être amené à définir de nouveaux formalismes.

Dans (Florijn et al., 1997), les auteurs déclinent trois types d'objectifs visés par les outils basés sur les patrons. La détection de patrons met en évidence les imitations de patrons dans le code ou les

spécifications correspondantes. La génération est, quant à elle, une démarche volontaire d'application d'un patron sous la forme de code où l'ingénieur d'applications prodigue les paramètres nécessaires à l'imitation. Enfin, la reconstruction unifie génération et détection pour extraire une imitation de spécifications originales.

Dans (Borne et Revault, 1999) et (Conte et al., 2001b), les auteurs proposent une décomposition intentionnelle de l'usage des patrons, où l'ingénieur d'applications peut avoir recours aux patrons dans plusieurs buts : aide à la conception et/ou génération de code, validation et rétro-conception et enfin documentation des spécifications. Ces trois types d'usage sont détaillés ci-dessous.

3.1 Aide à la conception et génération de code

L'objectif principal des patrons est d'aider les ingénieurs d'applications à l'analyse et à la conception des systèmes. Les patrons apportent des solutions efficaces à des problèmes d'analyse et de conception. Pour bénéficier du savoir (ou savoir-faire) décrit par le patron, l'ingénieur d'application doit ***imiter*** le patron, et plus particulièrement la solution, en prenant soin de l'adapter à son contexte. La plupart du temps, cette adaptation doit tenir compte des spécifications déjà présentes au sein du système en cours de développement. On parle alors de ***composition*** de l'imitation. La majorité des approches présentées dans le chapitre 3 propose des mécanismes de génération des imitations (avec ou sans composition).

La figure 1.3 présente un exemple d'imitation de la solution du patron « Composite » du GoF dans le contexte d'un système de gestion de fichiers. Les classes *Raccourci* et *Fichier* sont des feuilles tandis que *Repertoire* est la classe composite.

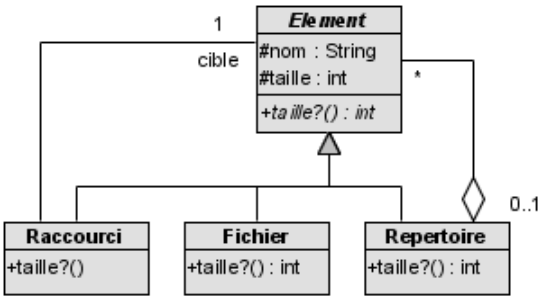


Figure 1.3 : Exemple d'imitation du patron « Composite ».

3.2 Validation et rétro-conception

Un autre objectif de l'utilisation des patrons est la validation des spécifications. En effet, les patrons permettent de valider la conception de modèles existants et de vérifier si l'architecture et les structures proposées par certains patrons ont été respectées. Un moyen de réaliser cette validation est de ***détecter*** la présence d'imitations dans les spécifications (ou dans le code).

La validation des spécifications d'un système peut également s'opérer par la validation des imitations explicitement réalisées pour construire ce système. Par nature, la solution d'un patron doit être assez générique et documentée de manière à pouvoir être adaptée dans différents

contextes d'imitation. Néanmoins, elle comporte certaines propriétés qui en représentent le sens, on parle aussi d'essence (Arnaud et al., 2004) ou de *« leitmotiv »* (Eden, 2000). Pour être valide, une imitation doit donc respecter ces propriétés, et ce, quelle que soit l'adaptation dont elle est issue.

3.3 Documentation

Une fois le patron imité, l'ingénieur d'applications peut souhaiter garder une trace de cette réutilisation afin de pouvoir ***documenter*** une partie de son système (et donc justifier certains choix) à l'aide de la description du patron. Néanmoins, cette documentation « automatique » n'a de sens que si l'imitation est valide (cf. 3.2).

4 Cas d'étude

4.1 Description du cas d'étude

Notre cas d'étude se situe lors de la conception d'un système de surveillance de bâtiments industriels où les températures ambiantes de certains locaux particulièrement sensibles doivent être contrôlées. Il s'agit plus précisément de permettre à un opérateur d'afficher les températures relevées toutes les minutes par les capteurs. Le recueil des besoins définit les propriétés suivantes.

- Les locaux comportent un ou plusieurs capteurs. Les bâtiments sont divisés en zones, elle-mêmes décomposables en plusieurs zones et/ou en plusieurs locaux.
- Toutes les mesures effectuées par tous les capteurs, qu'elles aient fait l'objet d'un affichage ou non, doivent être enregistrées.
- L'opérateur doit pouvoir disposer de trois modes d'affichage :
  - Forme Simple : une fenêtre présentant l'identifiant du local, d'un capteur et la température qu'il mesure en temps réel (cf. figure 1.4, à droite en haut).
  - Forme Graphique : un graphique présentant l'évolution de la température mesurée par un capteur depuis le démarrage de l'affichage de ce dernier par l'opérateur (cf. figure 1.4, à gauche, en bas). Ce graphique évolue également en temps réel.
  - Forme Historique : un rapport comprenant un tableau et un graphique, qui récapitulent, pour un local donné, l'ensemble des mesures effectuées par tous ses capteurs entre deux dates données. (cf. figure 1.4, à droite, en bas). C'est un instantané qui n'évolue pas en temps réel.
- Pour obtenir un affichage de type Simple ou Graphique, l'opérateur parcourt les différentes zones et locaux afin de sélectionner un capteur.
- Pour obtenir un affichage de type Historique, l'opérateur parcourt les différentes zones afin de sélectionner le local qui l'intéresse.

Les cas d'utilisation du système, illustrés figure 1.5, sont décrits comme suit :

- *Ajouter un afficheur temps-réel.* Pour consulter la température mesurée par un capteur, une fois ce dernier sélectionné en parcourant les zones et les locaux, l'utilisateur choisit entre deux types d'affichage (voir ci-dessus) : simple ou graphique (voir ci-dessus). Ensuite l'afficheur restera visible tant que l'utilisateur n'aura pas demandé sa suppression.
- *Supprimer un afficheur temps-réel.* Une fois que l'utilisateur juge qu'un afficheur n'est plus pertinent, il le sélectionne et demande sa suppression.
- *Générer un historique.* Pour obtenir un historique, l'utilisateur sélectionne le local qu'il désire visualiser ainsi qu'un intervalle de dates et le système génère un graphique relevant les valeurs mesurées par tous les capteurs de ce local durant l'intervalle.

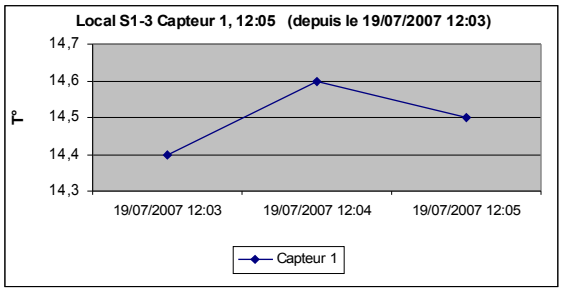
Local S1-3

Heure	Capteur 1	Capteur 2
19/7/07 12:00	14,2	13,8
19/7/07 12:01	14,2	13,9
19/7/07 12:02	14,3	13,8
19/7/07 12:03	14,4	13,9
19/7/07 12:04	14,6	14,1
19/7/07 12:05	14,5	14,1

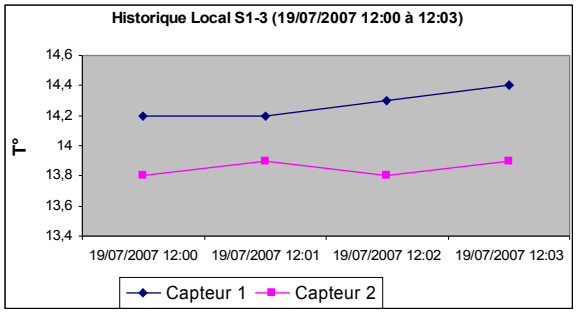
Données de l'exemple

Local S1-3 *12:05*
Capteur 1
Température : **14,5°C**

Afficheur de type « Simple »



Afficheur de type « Graphique »



« Historique » d'un local

Figure 1.4 : Données et exemples des modes d'affichage.

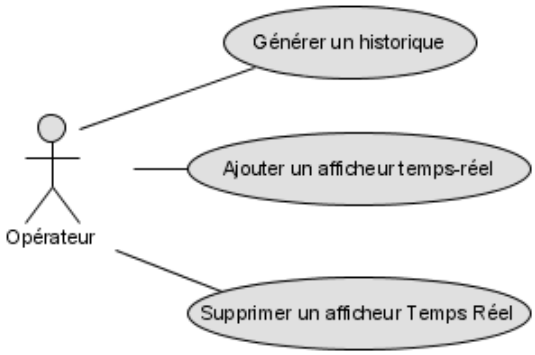


Figure 1.5 : Cas d'utilisation du système.

4.2 Les spécifications proposées

Sans pour l'instant entrer dans les détails, certains concepts-clés du système sont intuitivement identifiables, comme par exemple : les locaux, les zones, les afficheurs, les capteurs, les mesures. Dans notre exemple, nous ne distinguons pas les phases de développement logiciel, et plus particulièrement celles d'analyse et de conception. Il faut considérer les spécifications qui vont suivre, comme produites lors d'un développement rapide de système. L'ingénieur d'application possède, en outre, une bonne compétence sur les patrons et distingue rapidement dans les besoins exposés ci-dessus la possibilité de réutiliser certains patrons.

Pour trouver un patron à appliquer, il faut tout d'abord définir le problème que l'on veut résoudre et tenter de le faire correspondre à un problème résolu par un patron. Dans les patrons de Coad et du GoF, on trouve une description concise du problème dans les rubriques « Directives d'utilisation » pour Coad et « Intention » et« Indication d'utilisation » pour le GoF.

4.2.a Imitation du patron « Mémoire d'événement »

Le premier patron à apparaître clairement est « Mémoire d'événement » de Coad (cf. 2.2.a). En effet, d'après ses directives d'utilisation, ce patron est *« A utiliser quand un événement est détecté et que vous devez enregistrer son occurrence dans le but d'en faire une analyse a posteriori. »*. Ce patron semble effectivement pertinent pour le système en construction, afin de stocker les différentes mesures effectuées par les capteurs de température. Ainsi, l'ingénieur d'application peut imiter ce patron comme montré dans le diagramme de classe de la figure 1.6 :

- Une mesure (classe *Mesure*) est constituée d'une température (attribut *valeur*) ainsi que de l'instant (attribut *date*) à laquelle elle a été mesurée.
- Un capteur (classe *Capteur*) gère la collection de toutes ses mesures (rôle *mesure* et méthode *ajouterMesure*). Conformément à la solution du patron « Mémoire d'événement », le *Capteur* est chargée de l'instanciation des *Mesures* qu'il stocke.
- La méthode *mesurer* ajoute une nouvelle mesure à la collection. Elle sera invoquée par une horloge dont nous ne détaillerons pas le fonctionnement.

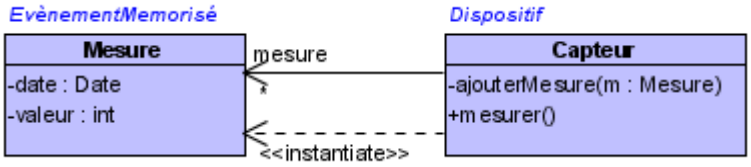


Figure 1.6 : Une imitation de « Mémoire d'événement » - Mesure/Capteur.

N.B. 1 : La dépendance stéréotypée « instantiate » signifie que la classe située à l'origine de la flèche est responsable de l'intanciation des objets de la classe désignée par la pointe de flèche. Dans la figure 1.6 , c'est donc le Capteur qui instancie les Mesures.

N.B. 2 : Le nom placé juste au dessus d'une classe désigne la classe du patron à laquelle elle est assimilée.

4.2.b Imitation du patron « Observateur »

Dans le système à construire, il est également question de mise à jour en temps-réel des afficheurs. Ce problème semble proche de l'intention du patron « Observateur » du

GoF : *« Définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour. »*. Tout comme pour le patron « Mémoire d'événement », l'ingénieur d'applications va donc imiter le patron « Observateur » et tenter de le composer avec les spécifications précédemment réalisées. La figure 1.7 présente le diagramme de classes résultant. Les correspondances qu'il effectue sont détaillées ci-dessous.

Les afficheurs s'apparentent à des *« objets dépendants »* (et donc « observateurs ») d'un capteur de température dont la valeur mesurée, qui représente l'état de ce dernier, change et provoque donc la mise à jour des afficheurs. Le système nécessitant deux sortes d'afficheurs « temps réel », l'ingénieur d'application définit deux « observateurs concrets » : *AfficheurSimple* et *AfficheurGraphique*.

- Pour ces deux observateurs, l'objet observé (le « sujet concret ») est donc le *Capteur*, déjà présent dans l'imitation du patron « Mémoire d'événement ». Grâce à cette imitation, ce dernier dispose déjà d'un « état », c'est à dire la liste des mesures qu'il a déjà effectuées (rôle *mesure*).
- L'*AfficheurSimple* a besoin de la dernière mesure effectuée pour remplir sa tâche (rôle *dernière_mesure*). L'*AfficheurGraphique* a, quant à lui, besoin d'avoir accès à toutes les mesures effectuées depuis qu'il a été démarré (attribut *dateDebut*). Il gère lui-même sa liste de mesures par l'intermédiaire du rôle *mesure_à_afficher*.
- Finalement, la seule information nécessaire aux observateurs (qu'ils soient de type simple ou graphique) lors leur mise à jour est la dernière mesure réalisée par le capteur. L'ingénieur opte pour le modèle « push » de notification (cf. rubrique « Implémentation »). Ainsi, il définit cette mesure comme paramètre de *miseAJour* et de *notifier*.

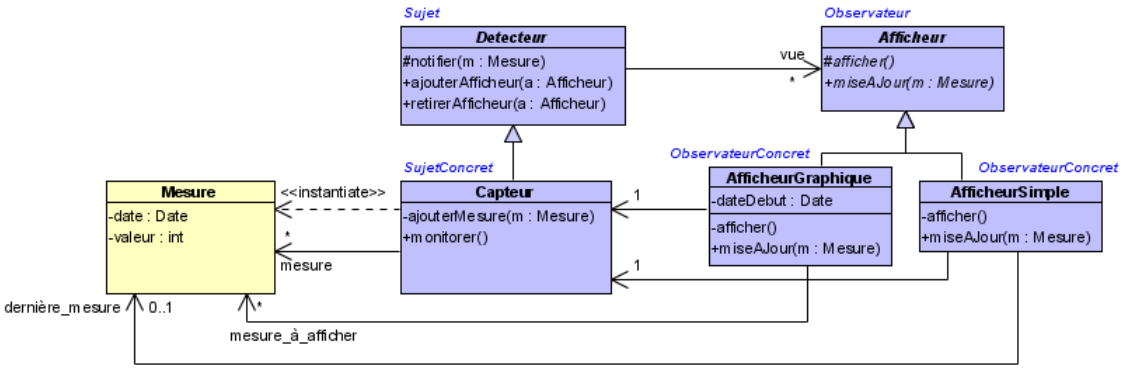


Figure 1.7 : Imitation du patron « Observateur » et composition avec l'imitation de « Mémoire d'événement ».

4.2.c Imitation du patron « Composite »

La décomposition du site surveillé en zones et locaux amène également notre ingénieur d'application à considérer le patron « Composite » du GoF. En effet, ce patron a pour intention de *« composer des objets dans des structures d'arbres pour représenter des hiérarchies tout-partie »*. Dans son système, les feuilles des arbres seront les locaux (classe *Local*) tandis que les objets composites seront les zones (classe *Zone*). Toute *Structure* (*Local* ou *Zone*) porte un nom et dispose de deux « opérations spécifiques » (cf.) :

- *détecteurs* : qui retourne l'ensemble des détecteurs (capteurs) qu'elle contient.
- *emplacement* : qui retourne une chaîne de caractères présentant la localisation de la structure. Pour un local X contenu dans une zone B elle-même contenue dans une zone A, la chaine retournée est *« zone A – zone B – local X »*.

Puisque seuls les locaux comportent des capteurs, la composition de cette imitation avec la spécification existante se fait en associant un ensemble de *Detecteur* à un *Local*. La figure 1.8 montre le résultat de cette composition.

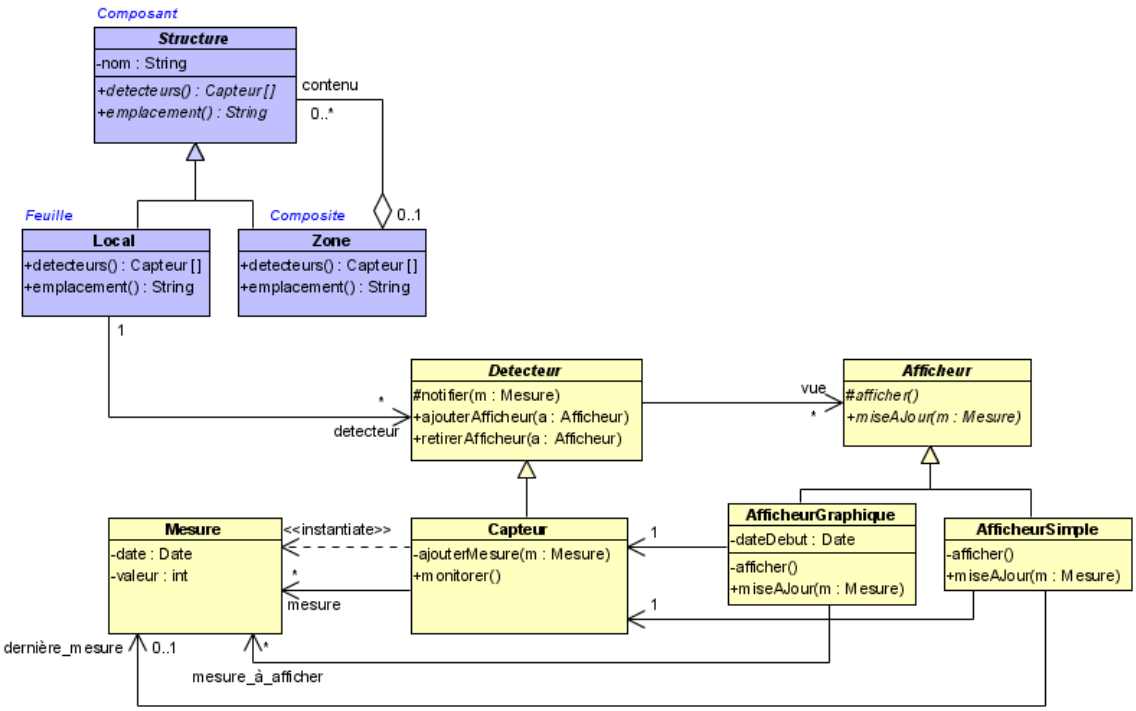


Figure 1.8 : Imitation du patron « Composite » et composition avec la spécification précédente.

L'ingénieur d'applications doit également déterminer la responsabilité de la création des afficheurs et la manière de mettre en œuvre cette dernière. La présence de paramètres de type *Afficheur* dans les méthodes *ajouterAfficheur* et *retirerAfficheur* (classe Detecteur) issues de l'imitation du patron « Observateur » suggère que l'instanciation des afficheurs n'est pas réalisée par les détecteurs. La section ci-dessous présente comment le patron « Méthode de fabrication » peut aider à résoudre ce problème.

4.2.d Imitation du patron « Méthode de fabrication »

L'intention du patron « Méthode de fabrication » (cf. Annexes) est la suivante « *Définit une interface pour la création d'objet, mais laisse les sous-classes décider de la classe à instancier* ». Tout en imitant ce patron, le système peut donc être complété par un créateur abstrait (*CreateurAfficheur*) et deux créateurs concrets (*CreateurAffSimple* et *CreateurAffGraphique*) dédiés chacun à l'instanciation d'un type d'afficheur (produits concrets). Cela permettra de créer les afficheurs de manière indépendante et homogène, quelque soit leur type. La figure 1.9 illustre le nouveau diagramme de classes du système.

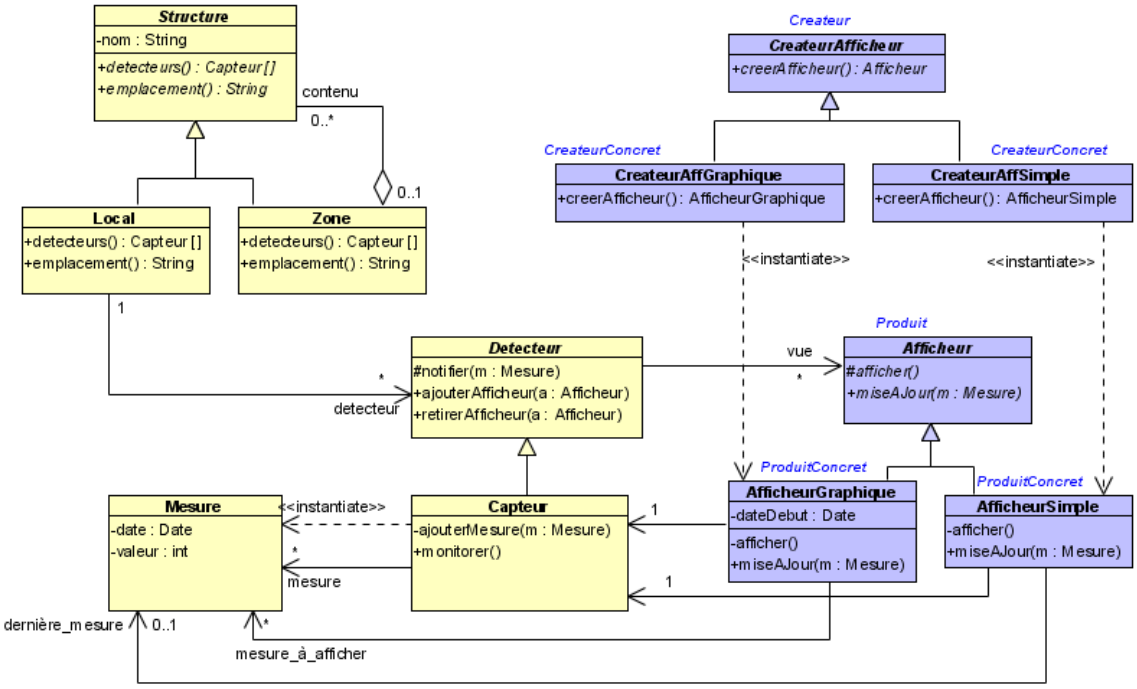


Figure 1.9 : Imitation du patron « Méthode de fabrication » (première version).

Pour compléter le système, il est nécessaire de produire une spécification qui met en œuvre les différents cas d'utilisation présentés au 4.1. Dans ce but, le concepteur ajoute une classe « Application » qui coordonne leur réalisation et gère l'accès aux *Structures* et donc aux *Capteurs*. Cette classe (*SystemeDeDetection*) réalise tous les cas d'utilisation du système *(demarrerAfficheur, supprimerAfficheur, afficherHistorique*). Ces méthodes n'ont pas de paramètre car elles réalisent un scénario de haut niveau.

Il ne peut y avoir qu'une seule instance de cette classe lors d'une exécution du système. Cette propriété correspond en fait à l'intention du patron « Singleton » (cf. Annexes) du GoF : « *S'assurer qu'une classe n'a qu'une seule instance, et en fournir un point d'accès global* ». La classe *SystemeDeDetection* sera donc une imitation du patron « Singleton ».

Dans les deux sections suivantes, nous détaillons deux manières d'intégrer la classe *SystemeDeDetection* aux spécifications précédentes en considérant plus particulièrement la mise en œuvre de la création des afficheurs.

Première version : délégation de la création des afficheurs

Une première solution amènerait la classe *SystemeDeDetection* à demander la création de l'afficheur à un *CreateurAfficheur* (instancié à chaque fois) en fonction de la méthode appelée (*demarrerAfficheurSimple* ou *demarrerAfficheurGraphique*). On obtiendrait alors le diagramme de classes de la figure 1.10 ci-dessous. Il est à noter que l'usage de méthodes statiques permettrait d'éviter l'instanciation systématique d'un créateur.

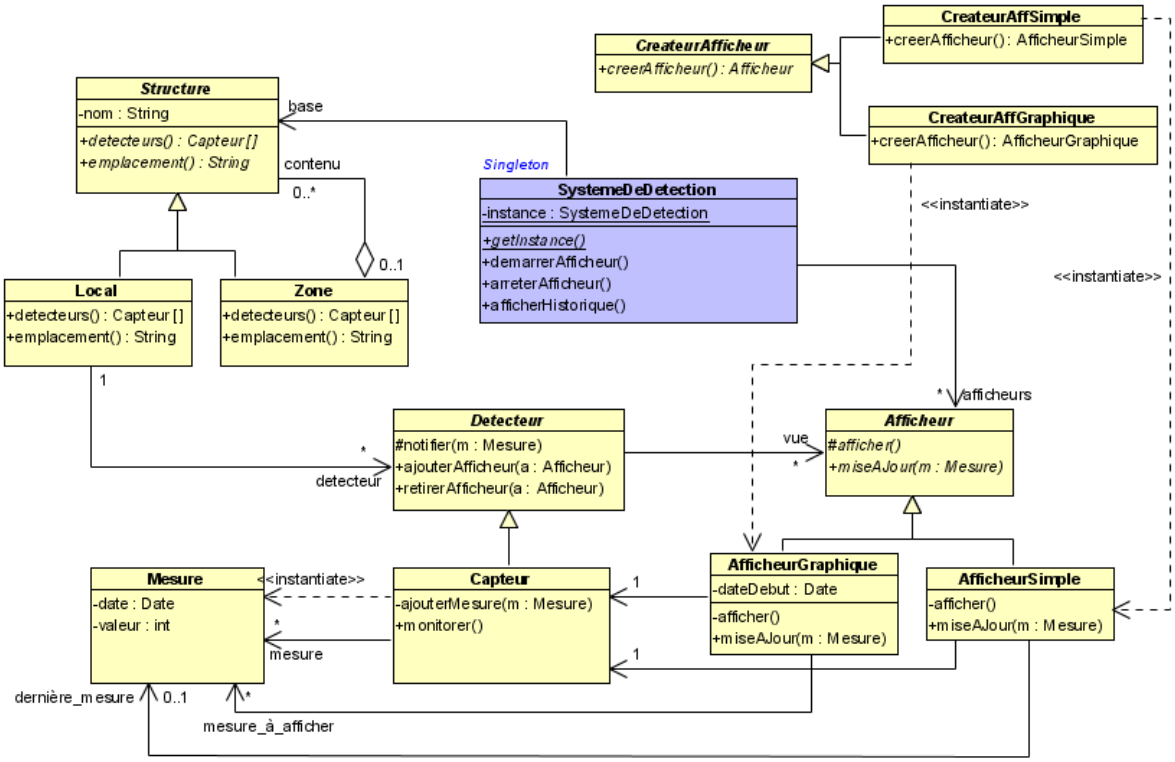


Figure 1.10 : Imitation du patron « Singleton » (première version).

Deuxième version : la classe application crée elle-même les afficheurs

Une autre solution est de donner à la classe application la responsabilité de la création des afficheurs. Afin de tirer partie des spécifications précédemment réalisées, il peut paraître normal de considérer la classe *SystemeDeDetection* comme un créateur d'afficheurs et donc de tenter de fusionner *CreateurAfficheur* et *SystèmeDeDetection*. Cela pose néanmoins quelques problèmes :

- Les deux classes n'ont pas le même niveau d'abstraction. En effet, *CreateurAfficheur* est une classe abstraite tandis que *SystemeDeDetection* est destinée à être concrète.
- Si on rendait la classe *SystemeDeDetection* abstraite, le lien de généralisation/spécialisation entre le créateur abstrait et ses deux sous-classes amènerait le système à comporter deux objets servant d'instance de classe « application ». L'une ne pourrait pas mettre en œuvre une création d'afficheur simple et l'autre une création d'afficheur graphique. Ceci est bien entendu contraire à la volonté originelle de l'ingénieur d'applications de manipuler tout le système à partir d'un seul objet.

Il existe cependant un moyen pour permettre cette fusion, mais il faut revenir sur l'imitation du patron « Méthode de fabrication ». En effet, il est question, dans la rubrique *Implémentation* de ce patron, d'utiliser une méthode de création paramétrée. Pour nous, cette « variante » est en fait l'expression d'un patron avec un problème légèrement différent où il n'est plus utile, dans la solution, d'avoir une famille de créateurs mais seulement un créateur doté d'une méthode avec un paramètre de type chaîne (par exemple) désignant le type de produit concret à instancier. La figure 1.11 illustre l'imitation de ce patron pour le système de détection.

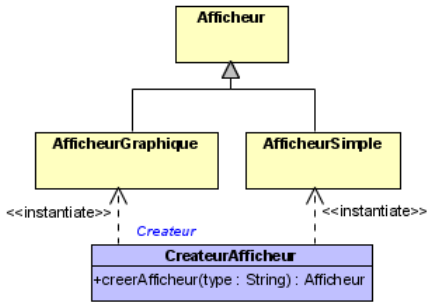


Figure 1.11 : Imitation du patron « Méthode de fabrication » (deuxième version).

La fusion de *CréateurAfficheur* et *SystemeDeDetection* ne pose dès lors plus de problème particulier. On obtient alors le diagramme de classes de la figure 1.12.

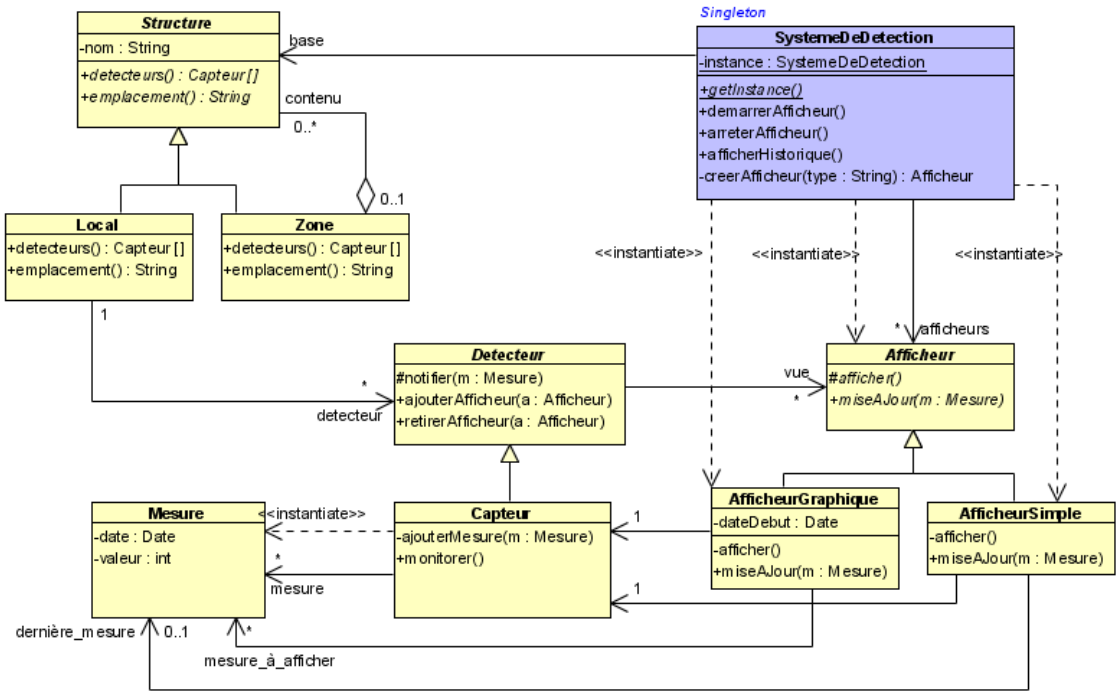


Figure 1.12 : Imitation du patron « Singleton » (deuxième version).

5 Problématique de la thèse

Le paragraphe 4.2 a montré comment un ingénieur d'applications, ayant acquis une pratique de l'imitation et possédant une connaissance certaine de quelques patrons, peut produire des spécifications à l'aide de leurs solutions, mais également du contenu d'autres rubriques qui les composent. Cette démonstration naïve de l'usage des patrons pour l'aide à la conception nous permet de mettre en avant les principaux problèmes inhérents à l'utilisation des patrons et soulève certaines questions quant au contenu même des solutions en vue d'un accompagnement de l'ingénieur d'applications lors de l'imitation et de la composition des imitations dans le système.

5.1 De la complétude des solutions

Dans le monde objet, l'immense majorité des solutions des patrons est constituée uniquement d'un diagramme de classes. Le comportement des entités présentes dans la solution semi-formelle est souvent décrit dans des notes apposées sur le diagramme de classes (cf. solution du patron « Observateur » au), parfois décrit de manière textuelle, et quelquefois omis par l'ingénieur de patrons. Pourtant, lorsqu'ils construisent un système, les ingénieurs d'applications ont besoin d'en spécifier les aspects comportementaux.

Dans le patron « Mémoire d'événements » (cf. 2.2.a) imité au début du cas d'étude, le comportement des objets, et plus particulièrement de la méthode *detecte,* est décrit de manière textuelle. Il est précisé que « *l'objet est également responsable du déclenchement de la réponse à l'événement. Une partie de la réponse peut être de mémoriser l'occurrence de l'événement* ». Même si rien n'est dit au sujet du reste de la réponse, il est raisonnable de penser qu'il va dépendre du contexte d'imitation du patron. D'ailleurs, dans le cas d'étude, après avoir intégré une imitation du patron « Observateur », l'ingénieur d'applications complète en fait la réponse par la notification aux observateurs (afficheurs). Si la solution de « Mémoire d'événements » avait comporté également un modèle dynamique (sous la forme, par exemple, d'un diagramme de séquence), ce dernier aurait été imité, et la spécification du comportement de la réponse en aurait été grandement facilitée par un simple ajout de message.

En généralisant, on peut supposer qu'une spécification plus **complète** des solutions de patron, c'est à dire en prenant en compte d'autres aspects que les seules propriétés statiques, améliorerait la qualité de l'imitation.

5.2 De la validité des imitations

Les problèmes rencontrés au paragraphe 4.2.d lors de la tentative de composition d'une imitation du patron « Singleton » avec une imitation du patron « Méthode de fabrication » sont dûs à des conflits entre des propriétés « fortes » (tel le niveau d'abstraction d'une classe) dans chacun des deux patrons. Ce genre de conflits est particulièrement important car il permet à l'ingénieur d'applications de détecter des incompatibilités entre ses désirs d'imitation et son système, mais également, dans certains cas, de se prémunir d'erreurs conceptuelles graves.

Lorsque l'on étudie un patron, il est normal de se poser des questions sur ces « règles » d'utilisation. Que dire d'une imitation du patron « Composite » où l'agrégation entre composite et composant disparaîtrait suite à l'adaptation ? Et si la cardinalité du rôle *composant* était changée en 1..*, serait-on toujours en présence d'une imitation du patron « Composite » ? Les fondements du patron seraient-ils toujours présents dans l'imitation ? Dans le cas d'étude, la méthode *operationSpecifique* a en fait été imitée deux fois. Est-ce contraire au sens du patron ?

Dans une imitation du patron « Observateur », il est clair qu'il peut y avoir plusieurs *ObservateurConcrets* (cf. cas d'étude) tandis que les classes abstraites *Sujet* et *Observateur* semblent ne pas devoir être imitées plus d'une fois si l'on veut conserver l'uniformité comportementale affichée par le patron.

En résumé, même si la solution d'un patron a pour but d'être adaptée au contexte d'imitation, l'ingénieur de patrons conçoit implicitement (ou explicitement dans les notes textuelles) des limites à l'application de son patron, au-delà desquelles l'utilisation du patron est contraire au sens qu'il lui donne, et en général au problème qu'il résout. Ces contraintes expriment ce que nous appelons le caractère **générique** de la solution et méritent d'être explicitées directement au sein de la solution de manière à être prises en compte dans le processus d'imitation que nous voulons définir.

5.3 De la variabilité des solutions

Dans le cas d'étude, lors de l'imitation du patron « Observateur », l'ingénieur d'applications s'est basé sur la rubrique « Implémentation » du patron pour imiter une solution où, contrairement à la solution initialement proposée, le message de notification comporte les informations nécessaires à la mise à jour des observateurs. Ce mode « push » est considéré par le GoF comme une variante d'implémentation, n'ayant aucune implication sur la manière dont le « Client » manipule les objets, autrement dit sa portée est limitée à la solution elle même.

De plus, le patron « Observateur » définit, comme précisé dans l'intention, la mise à jour des observateurs automatique : « ... *soient notifiés et automatiquement mis à jour.* ». Pourtant, toujours dans la rubrique « Implémentation », les auteurs discutent de la responsabilité du déclenchement de la notification (cf. tableau 1.2). Outre le déclenchement implicite en fin de la méthode modifiant l'état du sujet, il est question d'un déclenchement explicite par le « Client ». La variante « explicite » modifiera donc la manière d'intégrer l'imitation au système, et n'est pas tout à fait cohérente avec l'intention du patron, le problème étant légèrement différent. Nous remarquons que la définition d'un modèle comportemental pour chacun des deux cas faciliterait là encore la tâche de l'ingénieur d'applications. Dans le cas d'étude, c'est la variante nominale (i.e. notification implicite) qui est utilisée par l'ingénieur d'applications.

Qu'elles se situent au niveau de l'**implémentation** ou bien à un niveau plus **fonctionnel** (raffinement/généralisation du problème résolu par le patron), les variantes proposées par l'ingénieur de patrons devraient être systématiquement prises en compte lors de l'imitation. C'est pourquoi la notion de **variabilité** dans les patrons doit s'appliquer à notre processus d'imitation.

5.4 De la composition des imitations

La figure 1.13 ci-dessous présente, de manière symbolique, les implications de toutes les imitations dans la structure du système final (deuxième version). La **composition** d'une imitation, qu'elle soit ou non effectuée dans un système comportant déjà des imitations, est génératrice de problèmes de représentation, mais pose également le problème de la validation des imitations qui peuvent être impactées. En effet, si une imitation, lorsqu'elle est produite, doit être valide, et elle doit le rester.

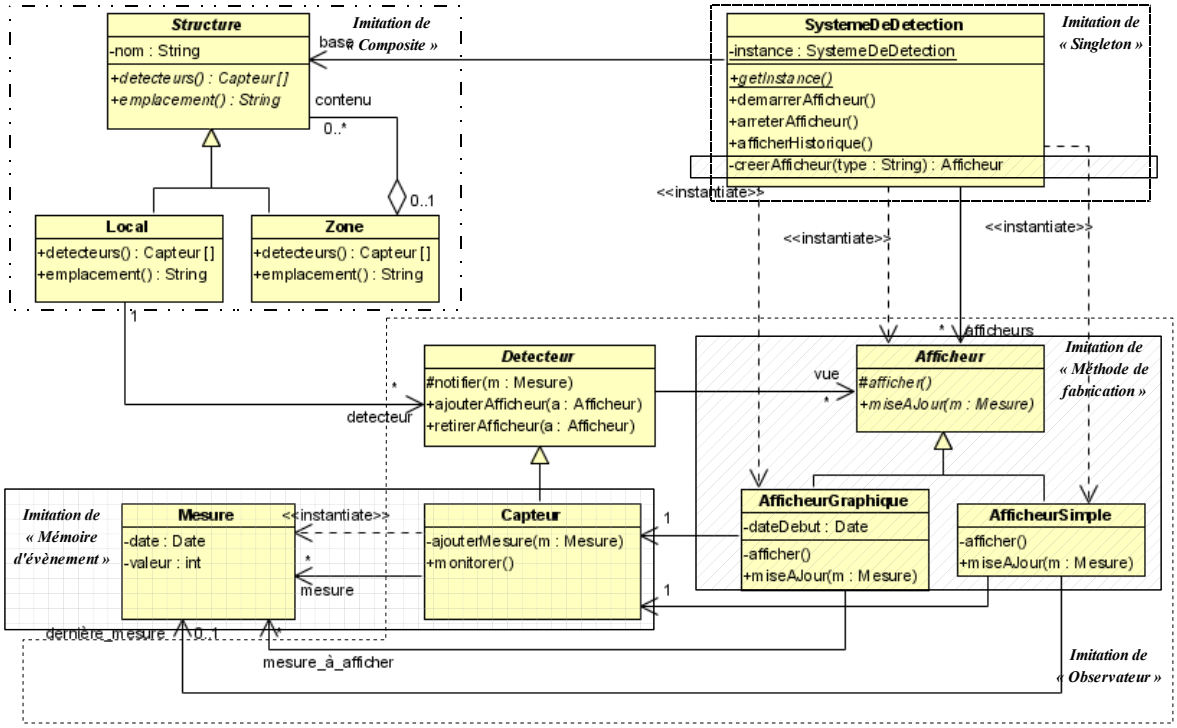


Figure 1.13 : Récapitulatif des patrons imités.

Lorsqu'il manipule son système, l'ingénieur d'applications est amené à faire des modifications sur les spécifications d'ores et déjà produites. Ces modifications peuvent avoir des conséquences fâcheuses. Par exemple, la suppression de la méthode de *notification* dans une imitation du patron « Observateur » rendrait cette imitation invalide. De plus, nous l'avons vu plus haut, l'imitation d'un autre patron peut provoquer des conflits.

Si les actions de l'ingénieur d'applications rendent une imitation invalide, il ne faut pas pour autant en déduire que ce qu'il veut faire est incorrect. En fait, il est probable que, finalement, le patron utilisé ne convienne pas, ou plutôt qu'après avoir reconsidéré son problème, l'ingénieur d'applications s'aperçoive qu'il ne convient plus complètement. Pour autant, la majorité des spécifications induites par son imitation conviennent toujours au « vrai » problème de l'ingénieur et doivent être conservées, mais plus au sein d'une imitation.

5.5 Bilan des objectifs

Notre objectif principal et de fournir aux ingénieurs d'applications un processus d'imitation de patron qui exploite des solutions complètes, variables et génériques tout en garantissant une traçabilité accrue dans un contexte où les imitations seront composées entre elles, ou avec des spécifications originales du système cible.

Pour arriver à ces fins, nous devons tout d'abord fournir aux ingénieurs de patrons les moyens de spécifier de telles solutions. Nous proposons donc une nouvelle forme de spécification des solutions de patrons intégrant des aspects fonctionnels, dynamiques et statiques complétés par l'expression de variantes et de propriétés génériques qui définissent les bornes de réutilisabilité d'un patron.

Le cycle de vie d'une imitation est une donnée primordiale pour la définition d'un processus d'imitation performant. C'est pourquoi nous décriront précisément les différentes étapes qui amène l'ingénieur d'applications à produire et à maintenir une imitation valide et adaptée à ses besoins. Dans un soucis d'utilisabilité de nos propositions, nous cherchons également à rendre automatisables certaines de ces étapes en nous appuyant sur l'ingénierie dirigée par les modèles.

CHAPITRE 2 APPROCHES ET OUTILS EXISTANTS

Variabilité et généricité au sein des patrons sont les données fondamentales de notre problématique. La généricité des solutions de patrons a fait l'objet de nombreuses propositions tandis que la variabilité n'a été que peu étudiée dans ce domaine. Plus généralement, les principes de variabilité dans l'ingénierie logicielle ne sont pas encore bien établis. C'est pourquoi nous commençons ce chapitre par une rapide définition de cette notion.

La plupart des approches présentées proposent ou étendent des langages de modélisation afin de prendre en compte les spécificités (ie. aspects génériques) des solutions de patrons ainsi que leur usage le plus courant : la génération d'imitations. Ces deux points nous incitent à considérer ces approches, ainsi que la nôtre, dans le contexte de l'ingénierie dirigée par les modèles. La deuxième section de ce chapitre fera quelques rappels sur ce domaine.

Toutes les approches présentées traitent d'un ou plusieurs usages des patrons, et, par extension, de la manière de définir les solutions exploitables. Nous présentons, dans une troisième section, un cadre de comparaison des approches insistant sur les aspects de définition et d'usage des patrons, du contenu des solutions ainsi que de la méta-modélisation. Ce cadre de comparaison est utilisé afin d'expliciter les approches dans la quatrième section et d'en synthétiser les résultats dans la cinquième section.

1 La variabilité dans l'ingénierie logicielle

Dans l'ingénierie logicielle, la variabilité peut être définie comme *« la capacité d'un système à être changé, personnalisé et configuré en fonction d'un contexte spécifique »* (VanGrup et al., 2001). Il convient de différencier la variabilité liée à l'exécution de celle liée à la conception. Il y a néanmoins une sorte de consensus autour de deux concepts principaux de la variabilité : les points de variation et les variantes. Un point de variation est *« un endroit du système où il y a une variation »* (Czarnecki et Eisenecker, 2000), c'est-à-dire où des choix devront être faits afin d'identifier les variantes à utiliser.

Deux grandes catégories de variabilité se distinguent : la variabilité liée à l'exécution (choix lors du déploiement, de la compilation, (...) et la variabilité liée à la conception (choix lors du développement du système). La première concerne plus particulièrement la personnalisation et la gestion de configurations tandis que la seconde est proche de la notion de famille de produits et de la réutilisation. Dans cette thèse, c'est cette variabilité pour la réutilisation qui est étudiée.

Si l'approche fondatrice, basée sur les diagrammes de « features » est FODA (Kang et al., 1990), d'autres travaux ont cherché à définir la variabilité dans l'ingénierie logicielle. Ainsi, pour (Bachmann et Bass, 2001), il existe plusieurs types de variantes associables à un point de variation. Chacun de ces types définit une cardinalité de choix de variante parmi un ensemble de variantes :

- *Variante optionnelle*. Il s'agit de sélectionner, ou non, une variante isolée. Cela revient à choisir zéro ou une variante parmi une. (noté *0..1 | 1*)
- *Variante alternative*. Les variantes alternatives (au moins deux par point de variation), partagent le même objectif, mais une et une seule doit être sélectionnée. Il faudra choisir une et une seule variantes parmi n. (noté *1..1 | n*)
- *Alternatives optionnelles*. Ces alternatives expriment une propriété optionnelle réalisable de plusieures manière, dont une seule peut être sélectionnée. C'est donc un choix entre zéro ou une variante parmi n. (noté *0..1 | n*)
- *Ensembles d'alternatives*. A la différence de « variante alternative », plusieurs variantes peuvent être sélectionnées. Il faudra choisir entre une et m variantes parmi n. (noté *1..n | n*).
- *Ensemble optionnel d'alternatives*. Il s'agit d'un ensemble d'alternatives qui peut donner lieu à un choix vide. Il s'agit de choisir zéro à n variantes parmi n. (noté *0..n | n*).

Deux remarques peuvent être associées à cette taxonomie :

- Pour « ensemble d'alternatives » et « ensemble optionnel d'alternatives », la cardinalité maximale pourrait être inférieure au nombre total de variantes. De plus, dans une ensemble d'alternatives, la cardinalité minimale pourrait être supérieure à un.
- Les variantes optionnelles d'un même point de variation se présentent de manière indépendante mais pourraient tout aussi bien s'exprimer sous la forme d'un ensemble optionnel avec une cardinalité *0..n | n*. En effet, si l'on regroupe n options,

il faudra choisir zéro ou n variantes parmi elles. Dans cette thèse, nous nous autorisons à regrouper les variantes optionnelles d'un même point de variation en sachant toutefois qu'il ne s'agit pas d'un ensemble optionnel d'alternatives, puisque ces variantes ne sont justement pas alternatives entre elles.

Dans le cadre de la variabilité pour la conception, plusieurs travaux ont cherché à introduire les différents concepts ci-dessus au sein de spécifications UML de lignes de produits. D'un point de vue fonctionnel, comme les cas d'utilisation de (Van der Maßen et Lichter, 2002), statique, comme les diagrammes de classes de (Clauss, 2001), ou dynamique, comme les diagrammes de séquences de (Ziadi et al., 2003).

2 Ingénierie Dirigée par les Modèles

Au début des années 2000, l'Object Management Group (OMG) propose une approche de développement et de maintenance de systèmes nommée Model Driven Architecture (MDA). L'idée principale du MDA est d'utiliser le standard UML pour modéliser les parties dépendantes (avec des PSM, *Platform Specific Models*) et indépendantes (avec des PIM, *Platform Independant Models*) des plate-formes et de définir des passerelles (i.e. transformations) entre elles.

Très vite, il s'est avéré que l'utilisation d'un unique méta-modèle (celui d'UML) limitait l'intérêt de l'approche, et que l'utilisation d'un ensemble de langages de modélisation était plus approprié. Toutefois, afin de garantir une certaine interopérabilité des différents langages (à des fins de transformations), l'OMG a proposé un méta-méta-modèle (MOF, *Meta-Object Facility*) (OMGb, 2005) auquel tous les méta-modèles (qui représentent un langage) utilisés dans un développement MDA doivent être conformes, ainsi qu'un standard pour la définition des transformations (MOF-QVT, *Query/View/Transformation*) (OMGc, 2005).

Pour (Favre et al., 2006), l'ingénierie dirigée par les modèles (IDM) vise à « *favoriser un génie logiciel plus proche des métiers en autorisant une appréhension des applications selon différents points de vues (modèles) exprimés séparément.* ». Elle intègre également « *la mise en cohérence de ces perspectives [..] et se veut productive en automatisant la prise en charge des outils relatifs à la validation des modèles, les transformations et les générations de code* ». De plus, contrairement à l'approche MDA, l'IDM ne se focalise pas sur une seule technologie, mais insiste sur les différents domaines de compétences (métier, technologie, gestion,...) d'un projet de développement, leurs besoins, ainsi que sur les relations qu'ils entretiennent.

Les systèmes, les modèles, les méta-modèles et les langages sont les concepts principaux de l'IDM. Certaines relations entre ces concepts, détaillées ci-dessous. sont également fondamentales.

- La relation *ModéleDe* (notée μ) définit, par exemple, qu'un méta-modèle représente un langage, ou qu'un modèle représente un système.
- La relation *ElementDe* (notée ε) peut préciser qu'un modèle est une partie (i.e. Un mot au sens « théorie de langages ») d'un langage.

- Si un méta-modèle MM représente (μ) un langage L, et que, un modèle M est un élément de (ε) L alors on peut dire que M est conforme à MM. La relation de conformité se note (χ).
- La relation *TransforméEn* (notée τ) permet, par exemple, de produire un modèle M2 par application d'une transformation sur le modèle M1. Le couple (M1, M2) est nommé une *instance de transformation*. Les transformations peuvent être décrites sous la forme de modèles de transformation, à partir des méta-modèles source et destination, de façon à les rendre automatisables.

La figure est une illustration des principaux concepts et relations de l'IDM. *M1* et *M2* sont des modèles représentant le même système *S1*. Ces deux modèles sont respectivement conformes aux méta-modèles *MM1* et *MM2* qui représentent les langages *L1* et *L2*. Une transformation est définie entre les deux méta-modèles. Le couple (M1, M2) est une instance de cette transformation.

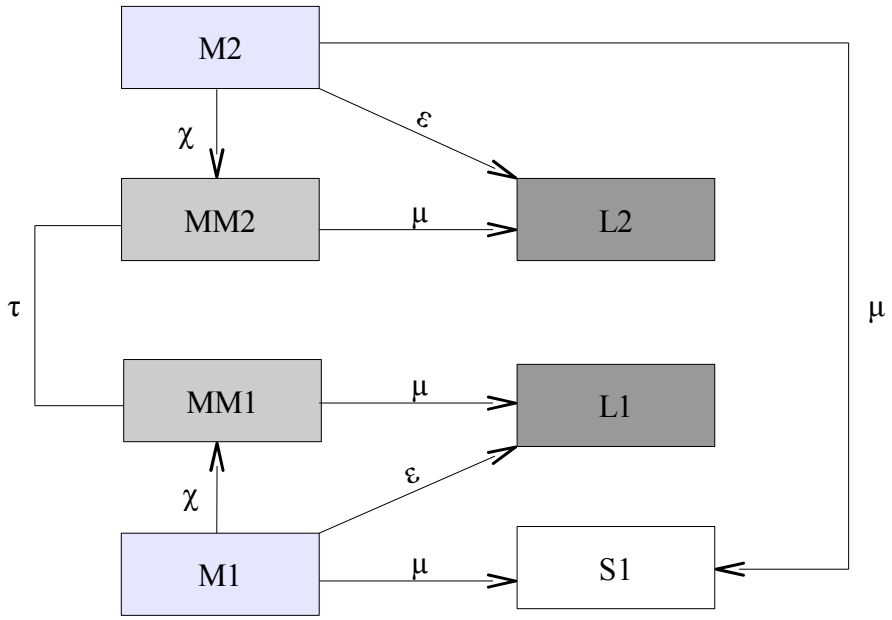


Figure 2.1 : Concepts et relations de l'IDM.

L'IDM est une « *approche intégrative générale* » (Favre et al., 2006) du génie logiciel, et même si certains sont antérieurs au MDA et au courant IDM, la plupart des travaux que nous présentons peuvent être considérés par rapport aux concepts fondamentaux que sont les modèles, méta-modèles et transformations. Nos propositions, dans leur définition comme dans leur mise en œuvre, se placent également dans le contexte de l'IDM.

3 Cadre de comparaison des approches pour les patrons

Afin de comparer les différents outils et approches liés à la réutilisation de patrons de conception, nous définissons un ensemble de critères répartis dans trois domaines :

- le domaine des fonctionnalités, qui comporte les critères exprimant les buts fonctionnels de l'approche.
- le domaine des solutions, composé de critères servant à qualifier les solutions de patrons manipulées par l'approche.
- le domaine de la méta-modélisation, qui compare les approches sur la manière dont elles utilisent l'IDM et la méta-modélisation.

3.1 Critères fonctionnels

S'il est entendu que les patrons de conception sont avant tout destinés aux ingénieurs d'applications, certaines des approches étudiées peuvent également intervenir au niveau des activités d'ingénierie de patrons. Il est donc intéressant de déterminer si l'approche est destinée à la réutilisation de patrons ou à la spécification de patrons, ou encore à la combinaison des deux. Dans les tableaux récapitulatifs, ce critère nommé « cible » est potentiellement multivalué et prendra les valeurs ***usage*** et/ou ***définition.*** Il conditionnera la pertinence des autres critères fonctionnels.

Afin de compléter notre grille d'analyse, nous proposons trois critères supplémentaires pour identifier les approches orientées vers l'usage des patrons ainsi que deux critères permettant de préciser leurs éventuelles contributions en terme de définition de patrons.

3.1.a Critères pour l'usage des patrons

Les fonctions d'usage proposées par les approches, c'est-à-dire ce qu'elles apportent à l'ingénieur d'applications désireux de travailler à l'aide de patrons, sont diverses. Il peut s'agir d'apports en terme de ***documentation*** des patrons, ou de réutilisation à proprement parler tels que la ***génération*** d'imitations, ou la génération avec ***composition***. La ***détection*** de patrons peut également être une préoccupation des ingénieurs d'applications.

De plus, les solutions de patrons de conception sont données la plupart du temps sous la forme d'un ou plusieurs diagrammes décrivant un modèle. Cependant, les besoins de l'ingénieur d'applications se situent également à des niveaux d'abstraction différents. C'est pourquoi certaines approches se consacrent tout particulièrement à l'utilisation des patrons dans le code. Nous complétons donc les types d'usages en les combinant avec les deux niveaux d'abstraction dans l'usage des patrons : ***modèle*** et ***code***.

Le premier critère d'usage des patrons est un ensemble de combinaisons des deux propriétés précédentes. Par exemple, la valeur *génération/code, détection/code* signifie que l'approche permet à l'ingénieur d'applications de générer des imitations sous forme de code ainsi que de détecter des imitations de patrons dans du code source.

La mise en œuvre des approches orientées usage implique la plupart du temps l'utilisation d'un processus. S'il est explicitement décrit, ce dernier peut être ***non, implicite, formalisé*** (par exemple sous la forme d'une démarche), ou même ***assisté*** par le support d'une application logicielle. Ce critère est important car il est fortement lié à l'utilisabilité de l'approche et à son appropriation par les ingénieurs d'applications.

Enfin, le dernier critère d'usage est le niveau de compétence des ingénieurs d'applications que nécessite l'approche. La diversité des approches étudiées induit un spectre assez large de connaissances relatives à leur bonne mise en œuvre. Tout d'abord, on distingue trois types d'artefacts potentiellement manipulés par l'ingénieur d'applications : ***modèle***, ***méta-modèle*** et ***code***. Ensuite, le savoir-faire sur ces artefacts est décomposable en trois activités graduelles : ***comprendre***, ***utiliser*** et ***produire***.

Le niveau de compétence de l'ingénieur d'applications requis est défini comme un ensemble de combinaisons des deux énumérations précédentes. Par exemple, un niveau de compétence *utiliser/modèle, produire/code* indique que la mise en œuvre des fonctionnalités d'usage de l'approche requiert de l'ingénieur d'applications qu'il sache comprendre et appliquer un modèle, ainsi que programmer. Les langages, qu'ils soient de modélisation ou de programmation ne constituent pas des critères discriminants, et seront, le cas échéant, précisés pour chaque approche.

3.1.b Critères pour la définition de patrons

Les deux critères liés à la définition de patrons par les ingénieurs de patrons sont la formalisation d'un processus de définition et le niveau de compétence requis pour les ingénieurs de patrons. Ces critères sont similaires à ceux présentés pour l'usage des patrons. Ainsi, le processus de définition des patrons, s'il est décrit par l'approche, peut être ***non, implicite***, ***formalisé*** ou ***assisté***. A l'instar de celles de l'ingénieur d'applications, les compétences requises pour l'ingénieur de patrons sont identifiables à l'aide de couples activité/artefact (cf. 3.1.a)

3.2 Critères traitant des solutions

Nous l'avons vu, l'ingénieur de patrons donne habituellement la solution de ses patrons de conception sous la forme d'une structure statique, éventuellement complétée par des aspects dynamiques. Nous différencions donc les approches en fonction de la complétude des solutions manipulées. Le critère de complétude (multivalué) précise si ces solutions comportent des aspects ***fonctionnels***, ***dynamiques*** et/ou ***statiques***.

La plupart des approches s'appuie sur la capacité d'application des solutions au contexte d'imitation. Cette capacité peut être étudiée selon deux axes : variabilité et généricité (cf. chapitre 1). Les solutions des patrons, dans le cadre de leur définition comme de leur utilisation, peuvent l'être à l'aide de **variabilité** et/ou de **généricité**.

Le critère de variabilité des solutions se base sur deux types de variabilité qui ont déjà été présentées dans le chapitre 1, il s'agit de la variabilité d'***implémentation*** et de la variabilité ***fonctionnelle***.

La généricité des solutions peut s'exprimer à travers un grand nombre de propriétés, dont certaines sont spécifiques aux approches et à leurs auteurs. Il ne nous semble pas pertinent de comparer les approches sur chaque propriété, c'est pourquoi le critère de généricité que nous proposons indique simplement si l'approche se base sur l'expression et/ou l'usage de généricité dans la solution.

3.3 Critères sur la méta-modélisation

Si elle ne sont pas explicitement fondée sur l'ingénierie dirigée par les modèles, la plupart des approches permettant la définition de patrons en utilisent les principaux concepts que sont les modèles et la méta-modèles. Nous dégageons deux dimensions pour comparer ces approches. Premièrement, le nombre de méta-modèles : ***un méta-modèle par patron*** ou ***un méta-modèle générique*** assez complet pour permettre de représenter tous les patrons. Ensuite, la dépendance entre l'imitation et la solution qui, si elle est effective, s'exprime par un lien de ***généralisation/spécialisation***, une ***délégation*** ou une ***instanciation*** (équivalent à relation χ, cf. section 2).

3.4 Récapitulatif

Le tableau 2.3 ci-dessous récapitule l'ensemble des critères et donne, pour chacun, l'ensemble des valeurs possibles.

DENOMMINATION			DESCRIPTION	VALEURS POSSIBLES
FONCTIONNALITÉS		**Cible**	Pour l'usage ou la définition de patrons ?	***usage / définition /usage & définition***
	USAGE	**Type**	Types d'usage pour l'ingénieur d'applications ?	*A*: documentation, génération, composition, détection | *B*: modèle, code ***Valeurs : AB***
		Processus	Sous quelle forme est donné le processus d'usage ?	***non / implicite / formalisé / assisté***
		Compétences	Quelles compétences doit avoir l'ingénieur d'applications ?	*A*: comprendre, utiliser, produire | *B*: méta-modèle, modèle, code ***Valeurs : AB****
	DÉFINITION	**Processus**	Sous quelle forme est donné le processus de définition ?	***non / implicite / formalisé / assisté***
		Compétences	Quelles compétences doit avoir l'ingénieur de patrons ?	*A*: comprendre, utiliser, produire | *B*: méta-modèle, modèle, code ***Valeurs : AB****
SOLUTIONS		**Complétude**	Quelles vues sur le patron sont gérées ?	*A*: fonctionnelle | dynamique | statique ***Valeurs : A****
		Généricité	Y-a-t'il des propriétés exprimant la généricité ?	***oui / non***
		Variabilité	Le cas échéant, quel type de	*A*

		variabilité est pris en compte ?	non | implémentation | fonctionnelle ***Valeurs : A****
MODÉLISAITONMÉTA-	**Nombre de Méta-modèles**	Combien de méta-modèles l'approche met-elle en œuvre ?	***un par patron | un pour tous les patrons***
	Dépendance	Comment le lien entre solution et imitation est-il mis en œuvre ?	*A* généralisation/spécialisation délégation instanciation ***Valeurs : A****

Tableau 2.1 : Récapitulatif des critères de comparaison des approches.

4 Approches et outils

4.1 La proposition de l'OMG : Les collaborations

Principes

Depuis sa version 1.3, le langage UML comporte un mécanisme de définition et de réutilisation de structures récurrentes. L'approche s'appuyait sur l'utilisation des structures de collaborations proposées dans les versions précédentes d'UML. Dans sa version 2.0 d'UML (OMGa, 2005), une *Collaboration* décrit *« un ensemble de participants qui coopérent à la réalisation d'une tâche donnée »*. Chacun de ces participants est un *Classifier* jouant un rôle de la collaboration. De plus, il est possible de définir des contraintes sur les instances des rôles définis.

Une C*ollaboration* n'est pas instanciable, mais « réutilisable » sous la forme d'une *CollaborationUse*, qui représente son application à un contexte donné. Cette application se fait en précisant quel classifier du contexte « cible » joue quel rôle de la collaboration. Cette application implique que l'ensemble des entités impactées collaborent à la réalisation de la fonction définie dans la collaboration et que, par conséquent, chaque classe liée à un rôle doit comporter les propriétés définies au sein de ce rôle. Les contraintes exprimées dans la collaboration s'appliqueront sur les instances des classes liées.

Dans (OMGa, 2005), il est précisé que les collaborations sont « *particulièrement utiles comme moyen de capture des patrons de conception »*. Cette approche est donc particulièrement pertinente dans le cadre de la définition et l'usage des patrons. La figure 2.2 présente la définition d'une collaboration représentant le patron « Composite », à partir d'une spécification réalisée par un ingénieur de patrons.

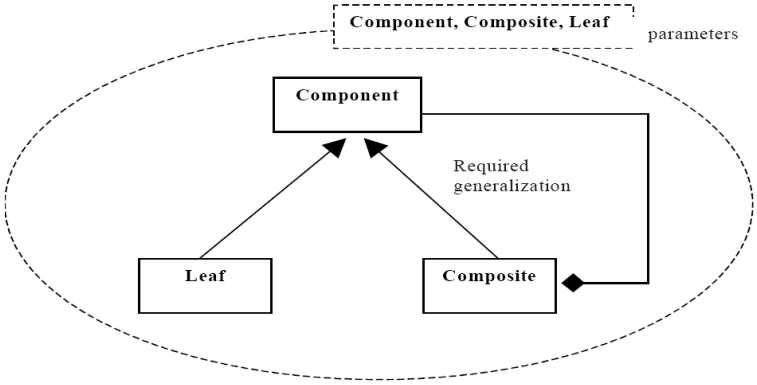


Figure 2.2 : Utilisation des collaborations génériques UML pour la représentation du patron « Composite » (extrait de (Booch et al., 2005))

Dans (Sunye et al., 2000), les auteurs démontrent en quoi l'utilisation des collaborations (au sens UML du terme, cf. ci-dessus) n'est pas satisfaisante, particulièrement en raison de l'impossibilité de spécifier correctement le *leitmotiv* - le terme *essence* est également employé - du patron et les contraintes (i.e. contraintes génériques) qui en découlent, puisque ces dernières ne s'appliquent pas sur les éléments imités mais sur les instances de ces derniers.

Tableau récapitulatif

Cible		Usage et définition sont traités.	***usage & définition***
USAGE	**Type**	L'objectif est de pointer au plus tôt dans le développement les endroits où l'application d'un patron est nécessaire. Cette application se fera par la suite soit sur le modèle, soit directement dans le code.	***documentation-modèle génération-code génération-modèle***
	Processus	Le processus n'est pas détaillé.	***non***
	Compétences	L'ingénieur d'applications doit maîtriser la notion de collaboration UML.	***comprendre-métamodèle, produire modèle***
DÉFINITION	**Processus**	Aucun processus de définition n'est proposé.	***non***
	Compétences	L'ingénieur de patrons doit maîtriser la notion de collaboration UML.	***produire-modèle***
Complétude		Une collaboration est avant tout une structure qui encapsule un comportement. Aspects statiques et dynamique des solutions sont traités.	***Statique, dynamique***
Généricité		La généricité n'est pas traitée.	***non***
Variabilité		La variabilité n'est pas traitée.	***non***
Nombre de Méta-Modèles		Il y a un méta-modèle général : UML.	***un pour tous les patrons***
Dépendance		Une collaboration (i.e. une solution de patron) est « instancié » sous la forme d'une *CollaborationUse*.	***instanciation***

Tableau 2.2 : Récapitulatif des collaborations UML.

4.2 FACE

Principes

Les travaux de recherche autour de FACE (Framework Adaptive Composition Environment) (Meijler et al, 1997) sont partis du constat qu'il existe un écart important entre la modélisation et le code source d'un programme. En effet, même si l'utilisation de patrons dans des outils de modélisation permet de générer une partie du code source à partir d'un modèle, l'écart reste grand entre le modèle informatisable et le code source. Le développement d'une application suit un cycle itératif pour lequel ce genre d'outils n'est pas adapté : la re-génération de code remplace systématiquement le code généré précédemment et, pire, le code modifié ou ajouté manuellement par le développeur.

Pour Meijler et al., l'origine du problème est la différence d'abstraction entre la modélisation, permettant l'imitation de patrons, et le code source d'un programme. Les auteurs proposent de supprimer le code source correspondant à une imitation pour n'utiliser, même lors de l'exécution du programme, que le modèle décrit dans un format spécifique. Ainsi, au lieu de travailler sur deux niveaux d'abstraction distincts, un seul niveau sera utilisé.

La spécification d'un patron avec FACE se compose de deux modèles : le métaschéma et le schéma solution. Le modèle issu de l'imitation d'un patron est appelé instance du patron. L'organisation de ces trois modèles est présentées figure 2.3.

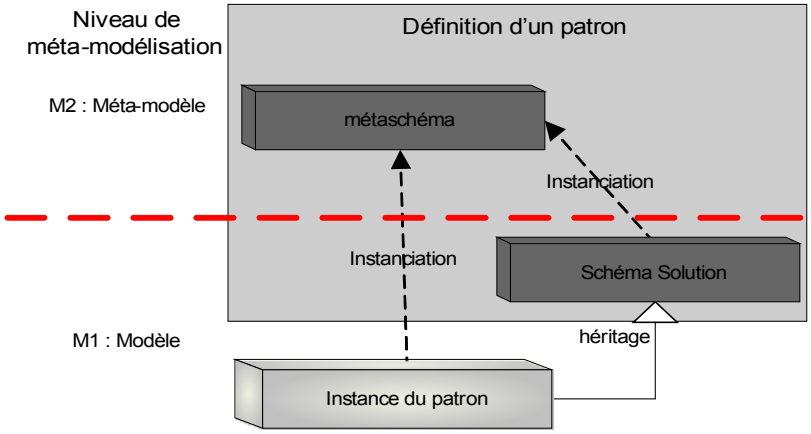


Figure 2.3 : FACE - Principe de fonctionnement

Le métaschéma est un méta-modèle spécifique à chaque patron et qui sert à définir le type des éléments utilisés dans les schémas solution (et les imitations). Il permet de typer fortement les éléments constituants d'une solution de patron, à savoir principalement les classes et les opérations.

Le schéma solution est constitué d'éléments instances du métaschéma qui sont considérés comme essentiels au patron. Ce schéma, partiellement conforme au métaschéma, est également nommé

schéma primaire. Il sert de base à la réalisation du schéma d'instance du patron qui sera, lui, conforme au métaschéma.

Pour débuter une imitation, il est nécessaire de commencer par spécialiser chacune des classes du schéma Solution. Enfin, l'instance (imitation) du patron est constituée uniquement d'instances des éléments définis dans le métaschéma.

Tous les diagrammes réalisés dans le cadre de cette approche utilisent une version légèrement étendue de la notation OMT. Un exemple partiel d'application au patron « Observateur » du GoF est détaillé ci-dessous. La figure 2.4 présente le couple métaschéma / schéma solution de ce patron ainsi qu'un schéma instance, en mettant en évidence les relations entre les différents schémas. Seules les classes Sujet et Observateur et les méthodes de Sujet sont présentées.

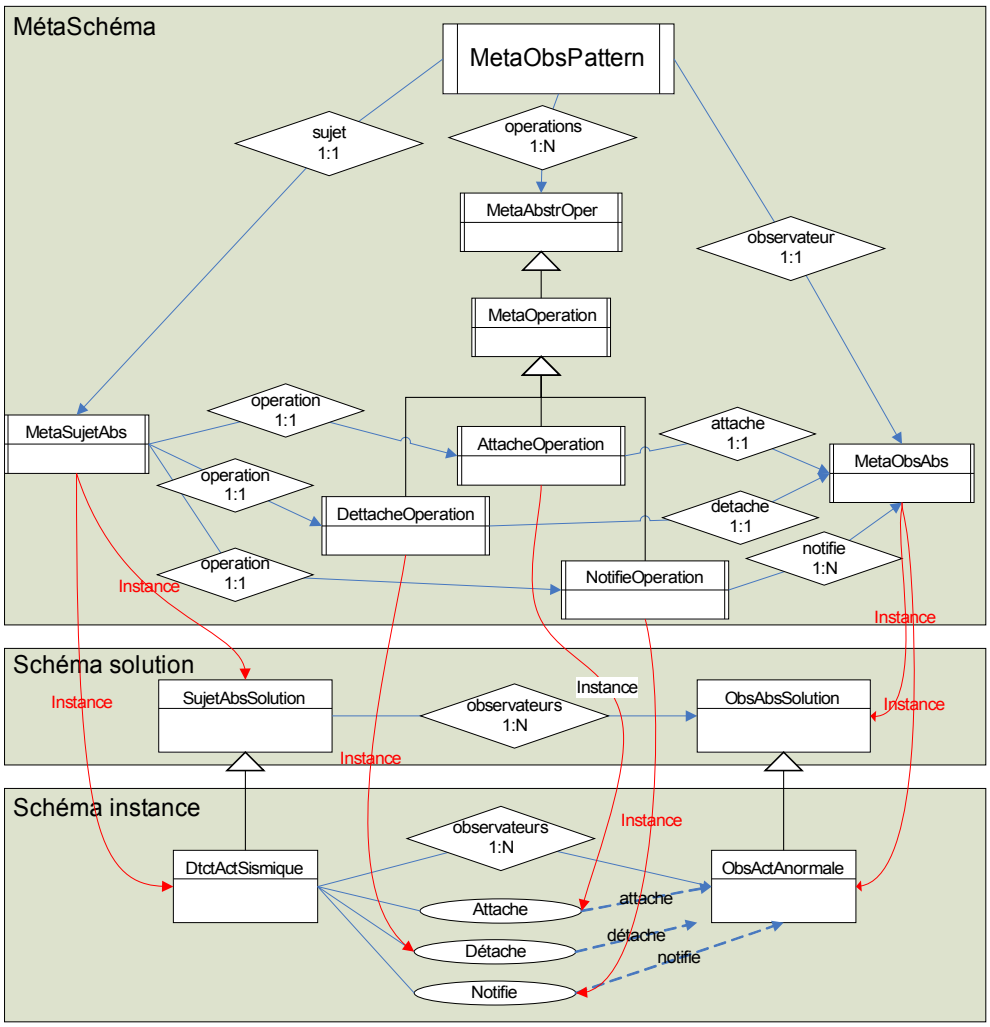


Figure 2.4 : FACE – Métaschéma, schéma solution et schéma instance pour le patron « Observateur ».

Tableau récapitulatif

	Cible	Usage et définition sont traités.	***usage & définition***
USAGE	**Type**	L'objectif est de générer du code.	***documentation-modèle génération-code***

	Processus	Le processus n'est pas détaillé.	***non***
	Compétences	L'ingénieur d'applications doit savoir faire un modèle OMT et comprendre un méta-modèle OMT.	***comprendre-métamodèle, produire modèle***
DÉFINITION	**Processus**	Aucun processus de définition n'est proposé.	***non***
	Compétences	L'ingénieur de patrons doit produire un méta-modèle ainsi qu'un modèle OMT pour chaque patron.	***produire-métamodèle, produire-modèle***
Complétude		Seuls les aspects statiques sont traités.	***statique***
Généricité		Une seul règle : tout ce qui n'est pas dans le schéma solution n'est pas obligatoire.	***oui (limitée)***
Variabilité		La variabilité n'est pas traitée	***non***
Nombre de Méta-Modèles		Il y a un méta-modèle par patron.	***un par patron***
Dépendance		Les entités de l'imitation (schéma instance) spécialisent les entités de la solution (schéma solution).	***généralisation/spécialisation***

Tableau 2.3 : Récapitulatif de FACE.

4.3 Travaux de Mel O`Cinneide

Principes

Les travaux de thèse de Mel O`Cinneide (O`Cinneide, 2001) traitent de l'application automatique de patrons de conception dans des systèmes préexistants et fonctionnels. Les patrons sont donc utilisés ici non pas pour générer les spécifications d'un système mais pour les compléter afin d'améliorer le système, sans modifier le comportement existant.

Cette approche se base sur les notions de mini-patrons, de mini-transformations et de précurseurs.

- Un mini-patron est défini comme « *un motif de conception de bas niveau se rencontrant fréquemment* » ; c'est en fait un patron de conception de très petite taille, résolvant un problème conceptuel très concret. Les mini-patrons peuvent être combinés afin de produire différents « vrais » patrons.

- Une mini-transformation est définie comme un ensemble de préconditions, un algorithme de description de la transformation, et un ensemble de postconditions qui permettent d'appliquer un mini-patron à un contexte donné, appelé précurseur.

- Un précurseur est une structure de conception où l'on doit retrouver les « participants » de l'intention du patron.

L'auteur propose et décrit précisément une « méthodologie pour les patrons de conception » qui est en fait un processus d'imitation (cf. figure 2.5). Après avoir choisi le patron qu'il veut « intégrer » au système existant, l'ingénieur d'applications doit choisir le précurseur (éventuellement composé de plusieurs éléments) sur lequel il veut appliquer le patron. Ensuite, il doit déterminer l'écart de conception entre le précurseur et l'état de ses constituants après l'imitation. De cet écart, il peut déterminer l'ensemble des mini-patrons à appliquer, et donc des mini-transformations à exécuter.

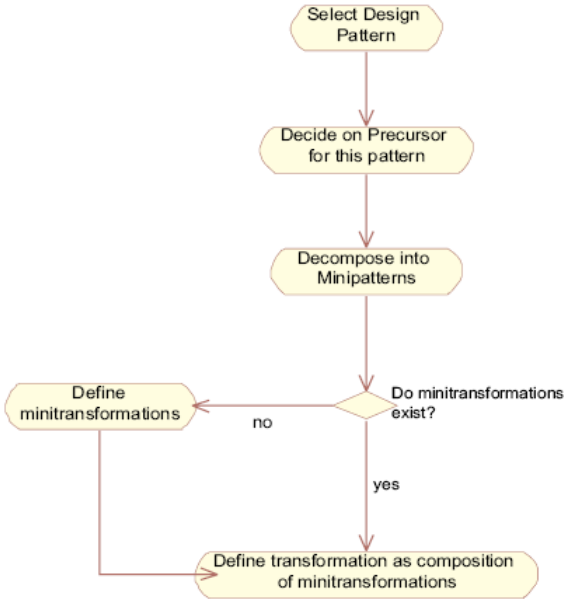


Figure 2.5 : Processus d'imitation proposé par (O`Cinneide, 2001)

Dans ses travaux de thèse (Eden, 2000), Amnon Eden a proposé un langage visuel (LePUS) destiné à définir des motifs récurrents dans le monde orienté-objet, et en particulier pour les patrons de conception. Ce langage propose en quelques sorte un ensemble de mini-patrons (*Abstraction, Délégation,...*) dont (O`Cinneide, 2001) s'est inspiré dans ses travaux. Ce langage visuel, aujourd'hui en version 3, est utilisé conjointement avec un langage de spécifications formelles (dédié à l'orienté-objet) nommé *Class-Z (LePUS, 2008)*. Un outil nommé *TTP* est également fourni et permet de définir (graphiquement ou formellement) et détecter des structures récurrentes (dont les patrons du GoF) dans du code Java.

Tableau récapitulatif

Cible		Usage et définition sont traités.	***usage & définition***
USAGE	**Type**	L'objectif est de générer des imitations au sein des spécifications de systèmes préexistants.	***génération-composition-modèles***
	Processus	Le processus est détaillé.	***formalisé***
	Compétences	L'ingénieur d'applications doit comprendre les mini-transformations.	***comprendre-modèle***

DÉFINITION	**Processus**	Le processus de définition n'est pas vraiment détaillé.	***implicite***
	Compétences	L'ingénieur de patrons doit produire du code de transformation ainsi que des contraintes OCL (préconditions, postconditions).	***comprendre méta-modèle, produire-code***
Complétude		Seuls les aspects statiques sont traités.	***statique***
Généricité		La généricité n'est pas traitée.	***non***
Variabilité		La variabilité n'est pas traitée.	***non***
Nombre de Méta-Modèles		Il n'y a pas de méta-modèle spécifique, mais, pour être utilisées, les transformations doivent s'appuyer sur le méta-modèle du système à modifier.	***?***
Dépendance		Il n'y a pas de lien direct entre solution et imitation.	***?***

Tableau 2.4 : Récapitulatif des travaux de Mel O`Cinneide.

4.4 Travaux de France et al. : RBML

Principes

L'approche proposée par France et al. (France et al., 2003) (France et al., 2004) est basée sur la méta-modélisation, plus particulièrement sur la spécialisation du méta-modèle UML. Cette dernière s'opère d'une part par le sous-typage des classes du méta-modèle et d'autre part, par la définition de règles, sous forme de contraintes OCL, qui affinent la sémantique du patron. Ces travaux différencient clairement les aspects structurels et d'interaction qui définissent la solution d'un patron. Les premiers sont décrits dans la spécification structurelle du patron (SPS : Structural Pattern Specification) et les seconds dans la spécification d'interaction du patron (IPS : Interaction Pattern Specification).

Dans le SPS, il faut définir une structure de rôles. Cette dernière comporte les sous-types des classes du méta-modèle UML ainsi que des contraintes pour décrire les éléments structurels du patron et leur sémantique. Le sous-type d'une classe du méta-modèle UML, est appelé rôle (classifier *role*). Le rôle spécifie les propriétés qu'un élément doit avoir pour être considéré comme un élément imité d'un patron. La classe concernée du méta-modèle est appelée la base du rôle et peut être un classifier ou une relation entre classes (association, généralisation/spécialisation, etc…).

La représentation d'un classifier « rôle » est illustrée figure 2.6. La partie haute indique le nom de la classe de base, le nom du rôle (précédé du caractère « | ») et une multiplicité (*realization multiplicity*) qui représente le nombre de classes pouvant jouer (i.e. réaliser) ce rôle dans un diagramme de classes conforme à la solution du patron. Une spécification structurelle (SPS) doit contenir au minimum un rôle à instancier (avec une multiplicité supérieure ou égale à un). La partie basse correspond aux propriétés structurelles du rôle (attributs). La troisième partie spécifie les propriétés comportementales associées au rôle (opérations).

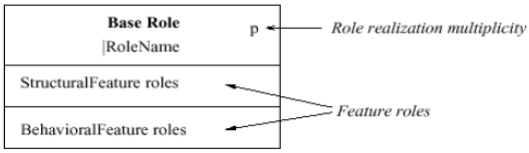


Figure 2.6 : Représentation de la Méta-entité « Rôle » (extrait de (France et al., 2004))

Un exemple partiel de SPS, appliqué au patron « Observateur », est montré figure 2.7. Il montre deux rôles de classes Subject et Observer et une association entre ces rôles appelée Observes (également un rôle). Il y est spécifié qu'il peut y avoir une ou plusieurs classes *Observer* et une ou plusieurs classes *Subject* et qu'une classe *Observer* doit avoir une seule association *Observes* avec une classe *Subject* et vice versa. De plus, un *Subject* doit contenir une et une seule propriété comportementale jouant le rôle de *Attach*. Ces rôles définissent des sous-types des classes du méta-modèle UML (version 1.5) comme montré dans la partie droite. *Subject* et *Observer* sont des sous-classes de la classe de base *Class, Observes* est une spécialisation de la classe de base *Association* et le rôle *Attach* est une spécialisation de *BehavioralFeature*.

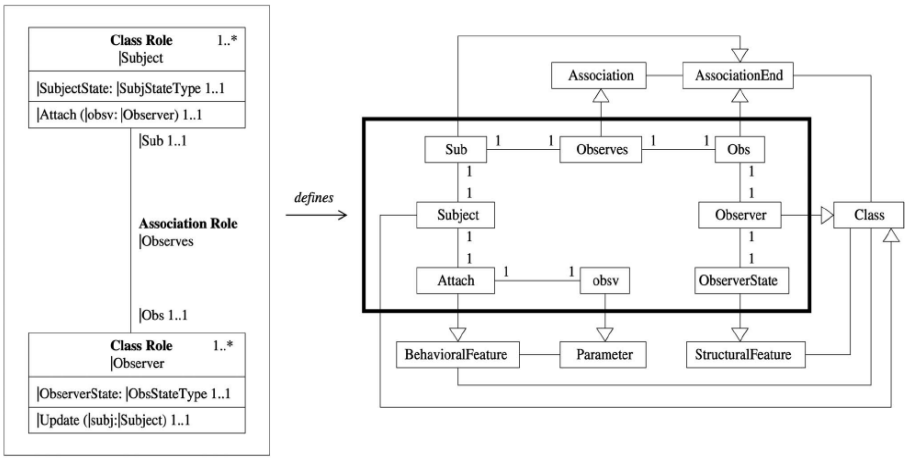


Figure 2.7 :SPS et méta-modèle partiel du patron « Observateur » (extrait de (France et al., 2004))

Certaines règles et contraintes additionnelles sur les éléments du modèle de la solution (les éléments qui jouent des rôles) sont exprimées sous forme de contraintes OCL. Elles interviennent à deux niveaux : le méta-modèle et la sémantique de la solution.

Au niveau du méta-modèle, ces propriétés servent à compléter la spécification en exprimant des règles garantissant les aspects structuraux du modèle. Afin de compléter le SPS de la figure 2.7, les contraintes suivantes peuvent être ajoutées :

- La classe *Subject* doit être concrète :
  ```
  context Subject inv : self.isAbstract = false
  ```
  - N.b. : isAbstract est une propriété de la classe *Classifier* du méta-modèle UML 1.5.
- La multiplicité de la terminaison d'association *Sub* doit être 1..1 :
  ```
  context Sub inv : self.lowerBound = 1 and self.upperBound = 1
  ```

- N.b. : lowerBound et upperBound sont des fonctions OCL applicables sur la classe *AssociationEnd* du méta-modèle UML 1.5.

Il faut bien distinguer la multiplicité des rôles et la multiplicité classique des terminaisons d'association. La contrainte ci-dessus impose que toute terminaison d'association jouant le rôle de *Sub* ait une multiplicité de 1..1 alors que la cardinalité 1..1 du rôle *Sub* (figure 2.7) implique qu'une classe jouant le rôle de *Subject* doit participer à une seule association *Observes*. Au contraire, la partie droite de la figure 2.8 montre un SPS dans lequel un *Subject* peut participer à plusieurs associations jouant le rôle d'*Observes*.

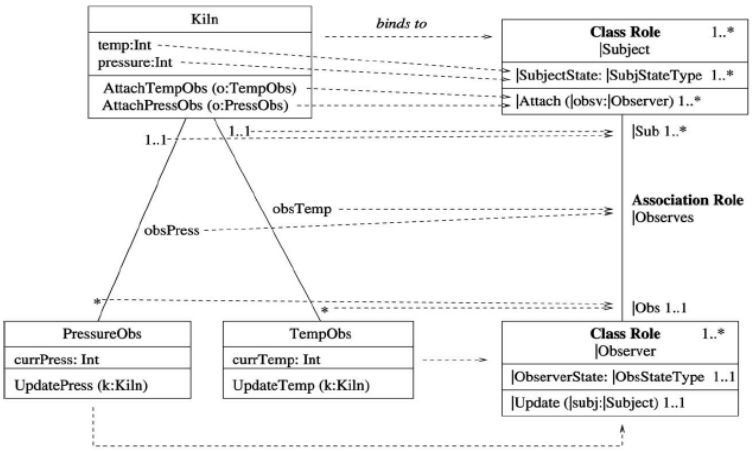


Figure 2.8 : Mise en correspondance d'un SPS et d'une imitation (extrait de (France et al., 2004))

En plus de ces aspects structurels, France et al. proposent également de spécifier des propriétés touchant à la sémantique particulière de la solution, à l'aide de « templates » de contraintes. Un « template » de contrainte est défini comme une contrainte exprimée en terme de rôle.

L'approche distingue, d'un côté, les «operation templates» qui expriment les propriétés sémantiques associées aux propriétés comportementales d'un rôle et, d'un autre côté, les «property templates» qui correspondent aux invariants du modèle. Ces « templates » seront par la suite instanciées en contraintes OCL classiques et appliquées sur les éléments jouant ce rôle. Nous pouvons compléter l'exemple précédent avec les contraintes sémantiques suivantes :

- Après exécution d'un *Attach* sur un *Subject,* l'observateur passé en paramètre fait partie des observateurs (terminaison d'association *Obs*) du *Subject* :

```
context |Subject :: |Attach(|obsv : |Observer)
  pre : true
  post : self:|Obs = self:|Obs@pre → including(|obsv)
```

  – N.b. : Ceci est une « operation template ».

- Dans un état stable du système, *ObserverState* est égal à *SubjectState*. (Nous supposons ici que ces deux attributs sont de même type) :

```
context |Subject
  |Obs → forAll(|ObserverState = |SubjectState)
```

  – N.b. : Ceci est une « property template ».

Un modèle de classes, pour être conforme à un patron, doit pouvoir être représenté à l'aide du méta-modèle du SPS et respecter ses contraintes structurelles. La figure 2.8 illustre la correspondance entre un modèle de classes imité (partie gauche) et un SPS (partie droite). La propriété ci-dessus, définissant la post-condition de la méthode *Attach,* est instanciée en :

```
context Kilm :: AttachTempObs (o: TempObs)
  pre : true
  post : self: Tobs = self:|Tobs@pre → including(o)
```

La spécification dynamique du patron est exprimée à l'aide des rôles définis dans le SPS. France et al. utilisent des diagrammes de séquence. Ils introduisent une spécification d'interaction du patron (IPS : interaction pattern specification) qui est à l'instar du SPS, un méta-modèle des modèles d'interaction conformes au patron.

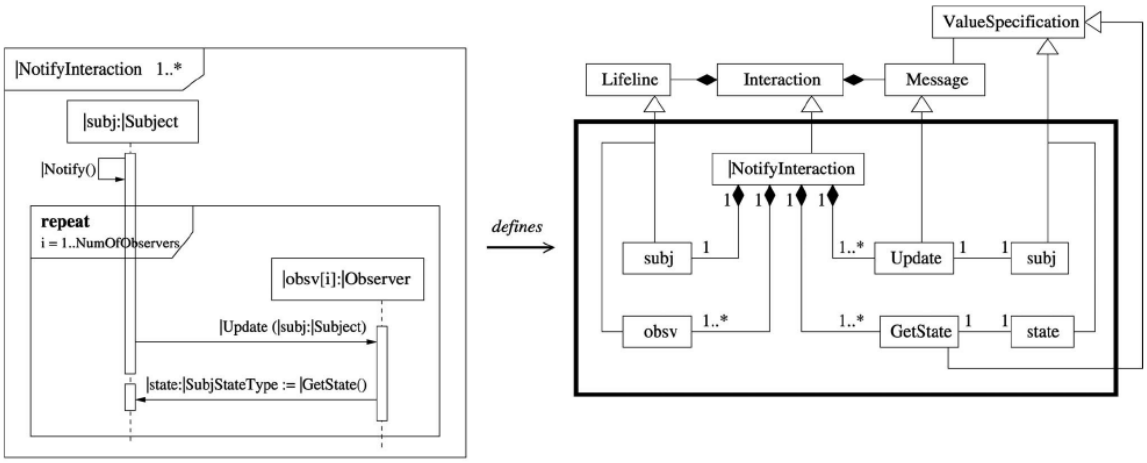


Figure 2.9 : IPS et méta-modèle partiels pour le patron « Observateur » (extrait de (France et al., 2004))

Tableau récapitulatif

Cible	Ces travaux traitent de la définition et de l'usage des patrons.	***usage & définition***

USAGE	**Type**	Il s'agit de générer des spécifications.	***génération-modèle***
	Processus	Une suite d'activités permet de définir si un modèle statique est conforme à un SPS donné.	***assisté (prototype)***
	Compétences	L'ingénieur d'applications peut produire son imitation et ensuite la comparer en associant chaque élément imité au rôle qu'il joue.	***comprendre meta-modèle (rôles) produire modèle***
DÉFINITION	**Processus**	Aucun processus de définition n'est proposé, même s'il est question d'un prototype d'aide à la création des SPS.	***non***
	Compétences	L'ingénieur de patrons doit produire une extension du méta-modèle UML ainsi qu'un modèle conforme à ce nouveau méta-modèle.	***produire-métamodèle, produire-modèle***
Complétude		SPS et IPS permettent de décrire les aspects statiques et dynamiques du patron.	***statique, dynamique***
Généricité		La notion de cardinalité de rôle permet une gestion assez fine du nombre d'imitations d'un élément. L'ajout de contraintes OCL sur le modèle statique affine également la définition de la validité des imitations.	***oui***
Variabilité		La gestion de la variabilité n'est pas traitée.	***non***
Nombre de Méta-Modèles		Il y a un méta-modèle (spécialisation d'UML) par patron.	***un par patron***
Dépendance		Chaque élément imité joue un rôle auquel il est relié par instanciation.	***instanciation***

Tableau 2.5 : Récapitulatif de (France et al., 2004)

4.5 DPML

Principes

David Maplesden et ses collègues (Maplesden et al., 2001) (Maplesden et al., 2002) proposent un langage visuel pour la modélisation et l'instanciation (imitation) des patrons (DPML : Design Patterns Modelling Language). La définition de DPML est la suivante : *« DPML définit un méta-modèle et une notation pour la modélisation des solutions de patrons de conception et la solution instance au sein de modèles à objets »*. L'objectif de ce travail est donc l'application de patrons au sein de modèles de classes UML et le maintien de leur validité. Pour cela, chaque DPML repose sur deux types de diagrammes : un diagramme de spécification et un diagramme d'instanciation du patron.

Le diagramme de spécification d'un patron est constitué d'une collection de participants, de dimensions associées aux participants (ensemble d'objets jouant un rôle dans le patron) et de relations binaires. Un diagramme d'instanciation représente les éléments imités, appelés instances de participants avec une syntaxe similaire au diagramme de spécification.

La notion de dimension introduite par cette approche est particulièrement intéressante car elle permet au concepteur de patrons de spécifier certaines corrélations entre les différents participants. Elle contraint le nombre d'occurrences d'imitation d'un élément avec une

combinaison du nombre d'imitations d'un ou plusieurs autres éléments. Par exemple, dans le patron « Fabrique Abstraite » du GoF, il doit y avoir autant de *ProduitConcret* que de couples *ProduitAbstrait/FabriqueConcrète*. Cette propriété est illustrée figure 2.10. Il faut donc définir une dimension pour *ProduitAbstrait* et une autre pour *FabriqueConcrète*, ces deux dimensions devant être appliquées à *ProduitConcret*.

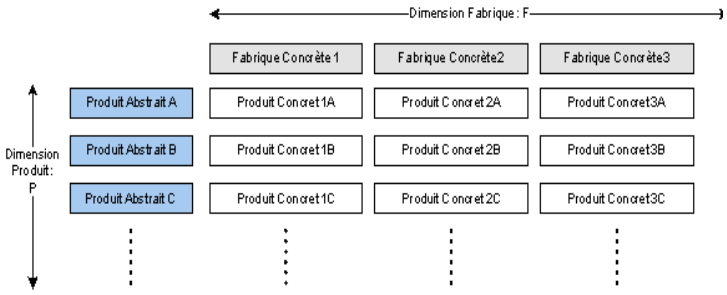


Figure 2.10 : Illustration du principe de dimension avec le patron « FabriqueAbstraite »

La figure 2.11 montre le modèle de classes UML donné par le GoF pour le patron « Fabrique Abstraite » et sa spécification à l'aide de DPML. On y retrouve les deux dimensions évoquées ci-dessus, s'appliquant à plusieurs des six participants. La partie inférieur de la figure illustre la syntaxe utilisée pour représenter les concepts objet considérés par l'approche : classe, interface, implémentation, méthode.

La figure 2.12 illustre une imitation du patron « Fabrique Abstraite ». La partie droite présente le diagramme d'instanciation DPML, où les noms des instances de participants sont écrits en italique et leurs types en pointillé.

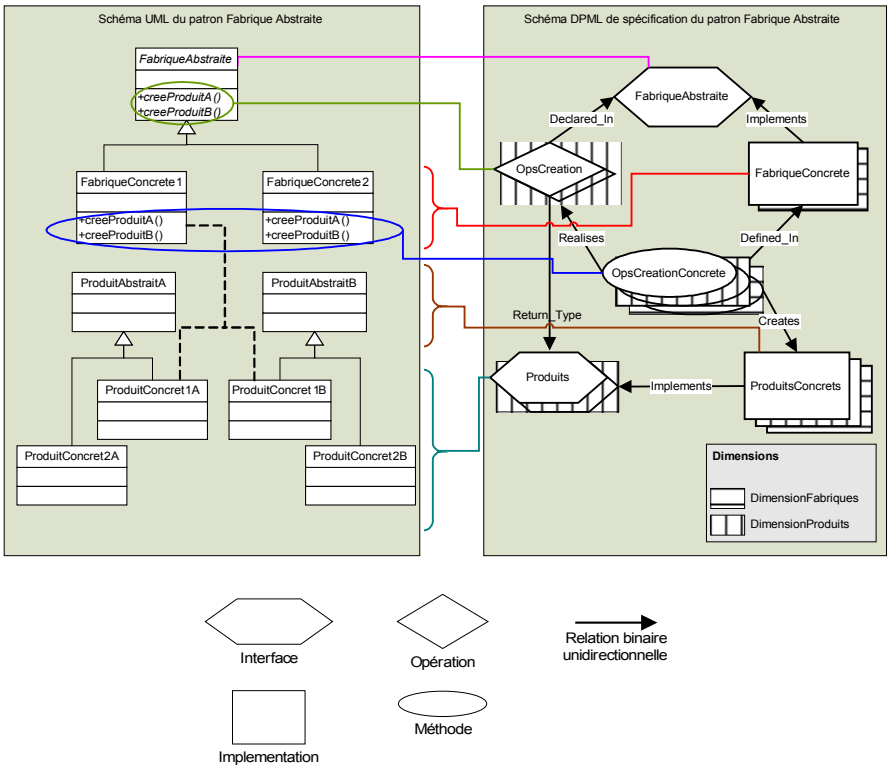


Figure 2.11 : Diagramme de spécification du patron « Fabrique Abstraite » avec DPML.

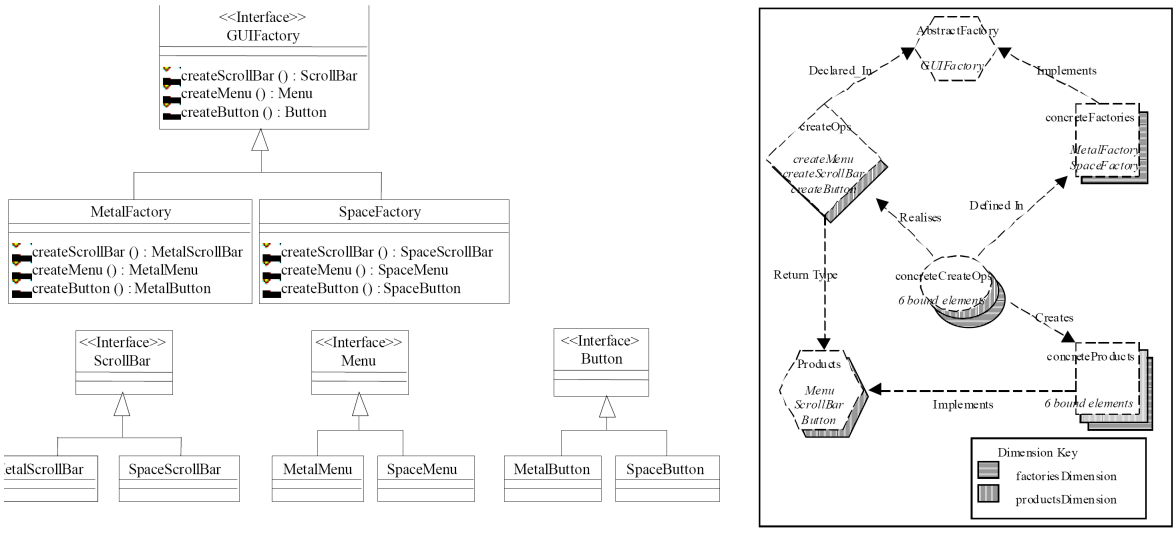


Figure 2.12 : Imitation du patron « Fabrique abstraite » avec DPML.
(extrait de (Maplesden et al., 2002))

L'approche proposée par les auteurs permet la représentation explicite des patrons de conception, et a été mise en œuvre dans un outil : *DPTool*. L'imitation y est aussi prise en charge mais il faut spécifier un diagramme d'instanciation pour chaque modèle ou fragment de modèle auquel on veut appliquer un patron.

Tableau récapitulatif

Cible		Ces travaux traitent de la définition et de l'usage des patrons.	***usage & définition***
USAGE	**Type**	L'objectif est de générer des imitations de patron au sein de spécifications préexistantes.	***composition-modèle***
	Processus	Même si un outil existe, le processus d'usage n'est pas formalisé.	***assisté (implicite)***
	Compétences	L'ingénieur d'applications doit spécifier le diagramme d'instanciation.	***produire-modèle***
DÉFINITION	**Processus**	Aucun processus de définition n'est proposé.	***non***
	Compétences	L'ingénieur de patrons doit produire une extension du méta-modèle UML ainsi qu'un modèle conforme à ce nouveau méta-modèle.	***produire-métamodèle, produire modèle***
Complétude		Seuls les aspects statiques des solutions sont traités.	***statique***
Généricité		Il est possible d'exprimer certaines propriétés génériques, notamment autour de la notion de dimension.	***oui***
Variabilité		La gestion de la variabilité n'est pas traitée.	***non***
Nombre de Méta-Modèles		Il y a un méta-modèle général.	***un pour tous les patrons***
Dépendance		Le lien de dépendance entre solution et imitation n'est pas explicité.	***?***

Tableau 2.6 : Récapitulatif de DPML.

4.6 Hook & template

Principes

L'approche de Pagel et Winter (Pagel et Winter, 1996) considère les patrons comme des primitives de conception au même titre que les classes, les attributs et les méthodes. Pour les auteurs, la plupart des patrons reposent sur le principe de réalisation d'interface.

En s'appuyant sur les méta-patrons de W. Pree (Pree, 1994), les auteurs proposent un méta-modèle (cf. figure 2.13) pour la description de tous les patrons de conception. On retrouve dans ce méta-modèle les éléments nécessaires à la spécification des aspects statiques du patron (*Pattern, PatternClass*, *Method*, *Association*,...) mais également une catégorisation des classes et méthodes selon deux types : *Hook* et *Template*. Cette distinction permet de mettre en avant les dépendances de type « réalisation » sensées constituer « l'essence de la plupart des patrons de conception ».

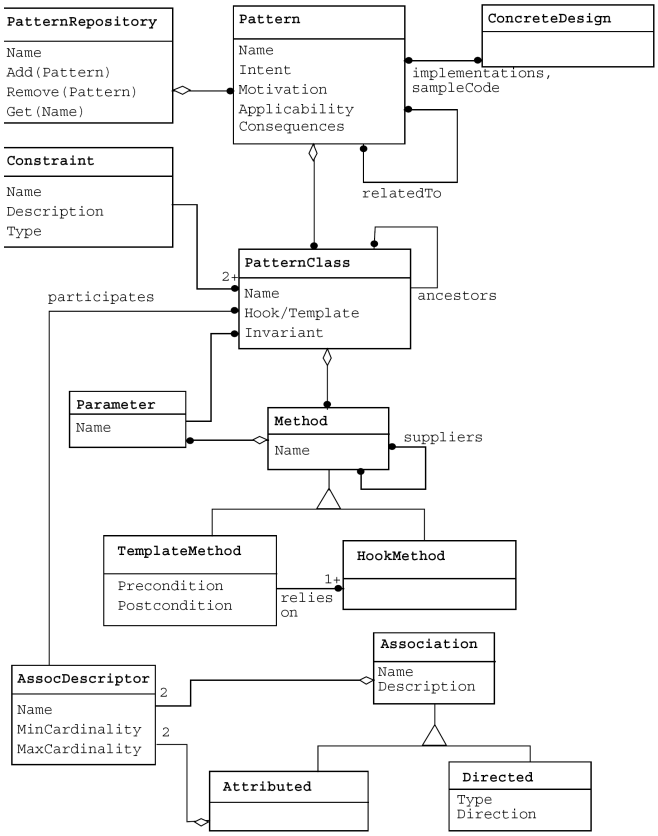


Figure 2.13 : Méta-modèle pour la définition des patrons de conception (extrait de(Pagel et Winter, 1996)).

En plus d'un méta-modèle pour la définition de patrons, Pagel et Winter proposent également le processus d'« instanciation » (i.e. imitation) des patrons. Ce processus permet à l'ingénieur d'applications de mettre en correspondance les différents constituants de la structure de la solution (classes, méthodes, etc.) avec leurs imitations respectives à travers quatre relations : *Classes* (classes), *Features* (attributs, méthodes), *Inheritance* (liens de généralisation/spécialisation) et *Associations* (associations). Ces relations peuvent être utilisées pour valider l'imitation par rapport à l'architecture abstraite du patron.

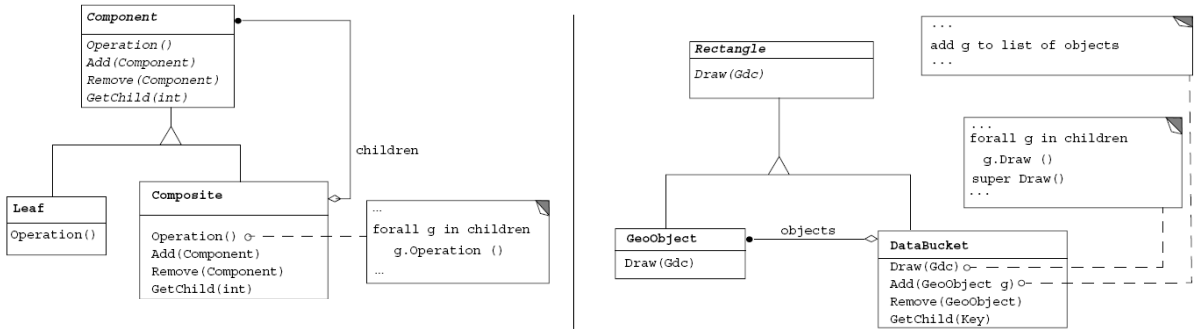


Figure 2.14 : Le patron « Composite » et une imitation (extrait de (Pagel et Winter, 1996))

L'imitation du patron « Observateur » présenté à droite de la figure 2.1 est instanciée comme suit :

```
Classes: {   (Rectangle, Component), (DataBucket, Composite), (GeoObject,
             Leaf) }

Features: {  (Rectangle.Draw(), Component.Operation()),
             (Rectangle.Add(Rectangle), Component.Add(Component)),
             (Rectangle.Remove(Rectangle), Component.Remove(Component)),
             (Rectangle.GetChild(), Component.GetChild()),
             (DataBucket.Draw(), Composite.Operation()),
             (DataBucket.Add(Rectangle), Composite.Add(Component)),
             (DataBucket.Remove(Rectangle), Composite.Remove(Component)),
             (DataBucket.GetChild(), Composite.GetChild()),
             (GeoObject.Draw(), Leaf.Operation()) }

Inheritance: { (Rectangle DataBucket, Component Composite),
             (Rectangle GeoObject, Component Leaf) }

Associations: { (objects(DataBucket, Rectangle, ε), children(Composite,
             Component, ε)) }
```

La relation *Associations* définit l'association *objects* comme une imitation de l'association *children*. En utilisant la relation *Classes* pour déterminer les antécédents des deux terminaisons de l'association *objects*, on obtient `objects(Composite,Leaf,ε)`, ce qui n'est pas cohérent avec `children(Composite, Component, ε)`. L'imitation n'est donc pas correcte.

Il est à noter qu'il n'est pas précisé quel méta-modèle utiliser pour les spécifications des imitations. En effet, la « trace » de l'imitation se faisant par l'intermédiaire des quatre ensembles, la seule contrainte sur ce méta-modèle est qu'il doit permettre de distinguer les classes, propriétés, liens de généralisation/spécialisation et associations pour pouvoir réaliser les correspondances.

Tableau récapitulatif

Cible		Ces travaux traitent de la définition et de l'usage des patrons.	***usage & définition***
USAGE	**Type**	L'ingénieur d'applications peut générer une imitation.	***génération-modèle***
	Processus	Le processus de réutilisation n'est pas détaillé.	***implicite***
	Compétences	L'ingénieur d'applications peut modéliser ses imitations dans la plupart des méta-modèles objets.	***produire modèle***
DÉFINITION	**Processus**	Aucun processus de définition n'est proposé.	***non***
	Compétences	L'ingénieur de patrons doit définir ses patrons dans le méta-modèle fournit.	***comprendre-méta-modèle, produire modèle***
Complétude		Cette approche traite des aspects statiques et, sans entrer dans les détails comportementaux, s'appuie sur les collaborations entre classes.	***statique, dynamique (minimale)***
Généricité		La structure de l'imitation est identique à la solution, à la différence près qu'il n'y a pas de limite sur le nombre d'imitations d'un élément. La notion de réalisation est la seule propriété générique réellement	***oui (minimale)***

	traitée.	
Variabilité	La gestion de la variabilité n'est pas traitée.	***non***
Nombre de Méta-Modèles	Un méta-modèle général est proposé.	***un pour tous les patrons***
Dépendance	Chaque élément imité est associé à son « antécédent » par le biais des relations.	***Délégation (externalisation)***

Tableau 2.7 : Récapitulatif de « Hook & Template ».

4.7 Travaux de Albin-Amiot et Guéhéneuc

Principes

Dans (Albin-Amiot et Guéhéneuc, 2001) (Albin-Amiot et al., 2002), les patrons sont manipulés à l'aide d'un méta-modèle dédié à la représentation des patrons (cf. figure 2.9). Ce méta-modèle est utilisé pour décrire la solution d'un patron, l'instancier, générer le code associé et détecter des patrons dans du code. Toute l'approche est orientée vers un environnement Java.

Dans ce méta-modèle, un patron est représenté par une instance d'une sous-classe de la classe Pattern. Cette dernière est une collection d'instances de la classe *Pentity* correspondant à un participant (classe ou interface) dans le patron. Chaque instance de *Pentity* contient un ensemble d'éléments instances de *Pelement* représentant ses attributs, ses méthodes et les relations d'association et de délégation (couple *Passoc-PDelegatingMethod*) qu'elle a avec d'autres instances de *Pentity*. En plus des aspects structurels, ce méta-modèle définit également des propriétés comportementales sur certaines méta-entités (i.e. la méthode *build* de *Pattern*).

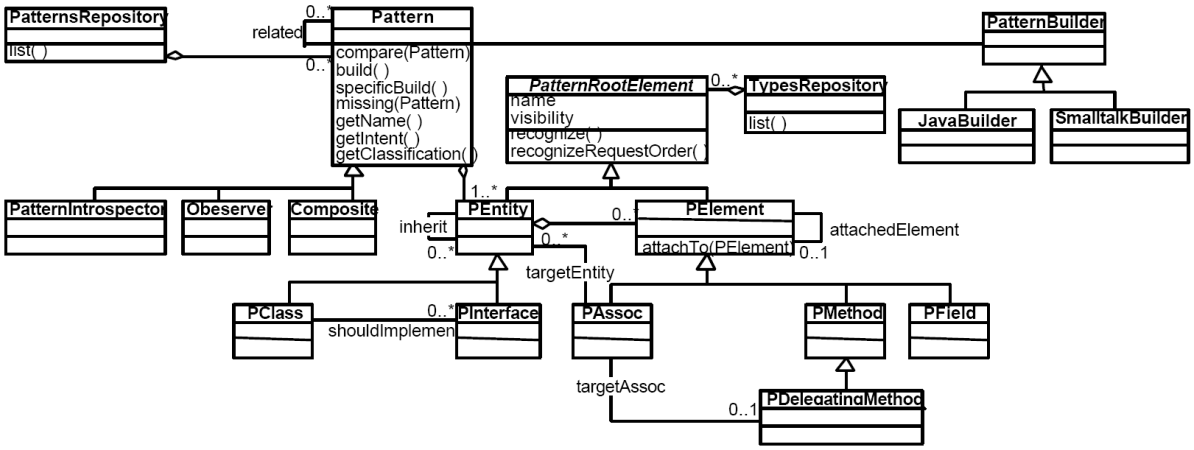


Figure 2.15: Méta-modèle pour les patrons, figure extraite de (Albin-Amiot et al., 2002)

Afin d'améliorer la manipulation des instances d'un tel méta-modèle, les méta-entités sont similaires à des JavaBeans, ce qui permet d'utiliser les méthodes de lecture-écriture d'attributs (préfixées *get* et *set*) et de collections (préfixés *add* et *remove*). Par exemple, on peut utiliser la

méthode *addshouldImplement* pour préciser, lors de la spécification d'un patron donné, qu'une des *Pclass* qui le composent doit, dans toute imitation implémenter une *PInterface* particulière.

Albin-Amiot et Guéhéneuc proposent également un processus d'« instanciation » de patrons composé de cinq étapes basées, similaire à celui décrit dans FACE (cf. 4.2), sur la manipulation du méta-modèle, d'un modèle abstrait et de modèles concrets. Les trois premières étapes permettent de spécifier une solution de patron tandis que les deux dernières sont destinées à l'imitation de celle-ci.

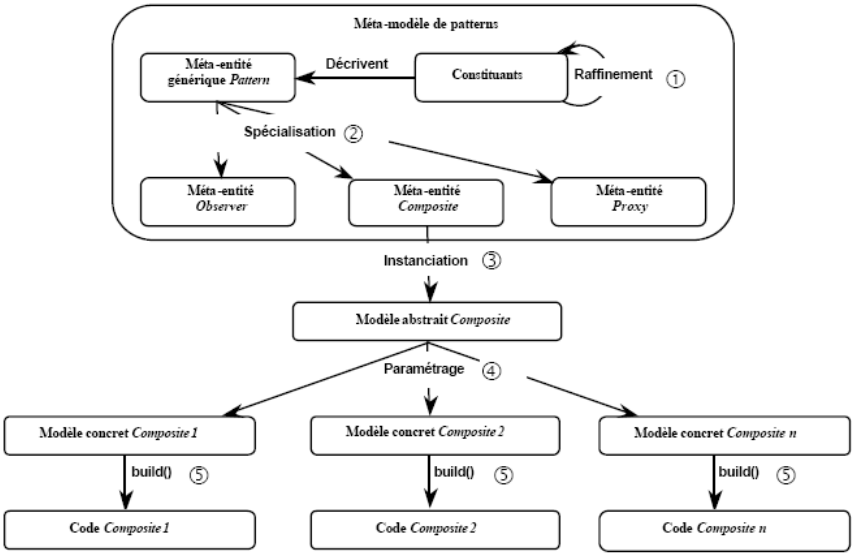


Figure 2.16 : Processus d'instanciation de patron (extrait de (Albin-Amiot et al., 2002))

La première étape (figure 2.14, point 1) consiste à améliorer le méta-modèle en spécialisant les méta-entités de manière à introduire de nouvelles propriétés. Pour le patron « Observateur », le méta-modèle présenté figure 2.9 est suffisant.

La seconde étape (figure 2.14, point 2) vise à ajouter un nouvelle entité spécialisant *Pattern* pour chaque patron et à en décrire la solution en explicitant les informations essentielles du patron. On exprime les propriétés structurelles à l'aide de primitives du méta-modèle précédemment défini tandis que du code Java permet d'exprimer la logique spécifique (appelée services propres du patron) d'adaptation au contexte d'imitation.

Cette description est réalisée de manière impérative, à l'aide du langage Java. La figure 2.14 présente un extrait de la déclaration de la méta-entité *Composite*, représentant le patron du GoF du même nom.

La troisième étape (figure 2.14, point 3) permet de créer un modèle abstrait de la solution par instanciation de la méta-entité spécifiée au point 2. Le modèle abstrait est ensuite stocké dans un entrepôt de patrons (entité *PatternsRepository*, figure 2.9) et sera le point de départ de toutes les imitations du patron.

Définition de la méta-entité *Composite*	Commentaires
`class Composite extends Pattern {`	Meta-entité *Composite*
`Composite(…) {` …	Déclaration du *leitmotiv* du pattern *Composite*, faite par le constructeur de la classe `Composite`, sous-classe de `Pattern`
`iComponent = new PInterface("Component")`	Déclaration de l'interface `Component`
`mOperation = new Pmethod("operation")` `iComponent.addPElement(mOperation)`	Déclaration de la méthode `operation()`, définie dans l'interface `Component`
`addPEntity(iComponent)`	Ajout de l'interface `Component` en tant qu'élément constitutif du pattern
`anAssoc = new PAssoc("children", iComponent, 2)`	Association ayant pour cible `Component` et de cardinalité > 1
`cComposite = new PClass("Composite")`	Déclaration de la classe `Composite`
`cComposite.addShouldImplement(iComponent)`	La classe `Composite` a pour interface `Component`
`cComposite.addPElement(anAssoc)`	L'association `children` lie `Composite` et `Component`

`aPDelegatingMethod = new PDelegatingMethod( "operation", anAssoc)`
`aPDelegatingMethod.attachTo(mOperation)`
`cComposite.addPElement(aPDelegatingMethod)`
	La méthode `operation()` définie dans la classe `Composite` implémente la méthode `operation()` de l'interface `Component` et est liée via le lien d'association `anAssoc` à cette même interface

`addPEntity(cComposite)`	Ajout de la classe `Composite` au pattern
`cLeaf = new PClass("Leaf")`	Déclaration de la classe `Leaf`
`cLeaf.addShouldImplement(iComponent)`	La classe `Leaf` a pour interface `Component`
`cLeaf.assumeAllInterfaces()`	L'interface publique de la classe `Leaf` est générée automatiquement (création d'une méthode `operation()`)
`addPEntity(cLeaf)`	Ajout de la classe `Leaf` au pattern
`}` …	Déclaration de services propres au pattern …

`void addLeaf(String leafName) {`
`PClass newPClass = new PClass(leafName)`
`newPClass.addShouldImplement(`
`(PInterface)getActor("Component"))`
`newPClass.assumeAllInterfaces()`
`newPClass.setName(leafName)`
`addPEntity(newPCclass)`
`}`
	Exemple de la méthode `addLeaf()`

… `}`	

Propriétés structurelles (dans le constructeur)

Propriétés dynamiques (services du patron)

Figure 2.17 : Extrait de la déclaration de la méta-entité « Composite » (extrait de (Albin-Amiot et al., 2002))

L'imitation d'un patron s'effectue en deux temps. Tout d'abord, la quatrième étape (figure 2.14, point 4) permet la « paramétrisation » du modèle abstrait en un modèle concret, adapté au contexte de réutilisation. Si la structure de base a été produite par le constructeur, il est toujours possible d'utiliser les services du méta-modèle pour modifier les propriétés de certaines entités (ie. leur nom). De plus, l'utilisation des services propres du patron permet d'affiner le modèle concret en utilisant la logique définie par et pour le patron. Ainsi, lors d'une imitation du patron « Composite » (en utilisant le modèle abstrait désigné par une variable *patComposite*), on pourra :

- changer le nom de la classe *Component* pour l'adapter au contexte :
  `patComposite.getActor("Component").setName("nouveauNom")`
- changer le nom de la méthode *operation* pour l'adapter au contexte :
  `patComposite.getActor("Component")`
  `.getActor("operation").setName("nouveauNom")`
- ajouter un type de feuille (*Leaf*) (service propre au patron « Composite ») :
  `patComposite.addLeaf("NomDeLaNouvelleFeuille")`

Finalement, l'appel de la méthode *build* (classe *Pattern*, figure 2.9) sur un modèle concret (figure 2.14, point 5) permet la génération du code Java décrivant l'imitation du patron précédemment paramétré. L'imitation (points 4 et 5) est assistée par un outil nommé *PatternBox*.

Tableau récapitulatif

Cible		L'approche traite de la définition et de l'usage des patrons.	***usage & définition***
USAGE	**Type**	L'ingénieur d'applications peut générer une imitation et détecter des patrons dans le code.	***génération-code détection-code***
	Processus	Le processus d'imitation, qui a pour but de générer du code, est assisté par un outil.	***assisté***
	Compétences	L'ingénieur d'applications manipule les patrons et les imitations uniquement à partir du code.	***produire code***
DÉFINITION	**Processus**	Le processus de définition de partons proposé inclut une amélioration par incrément du méta-modèle de définition.	***formalisé***
	Compétences	Codage et modélisation sont utiles au processus de définition de patron (et de raffinement du méta-modèle).	***comprendre-méta-modèle, produire méta-modèle produire-code***
Complétude		Puisque représentées sous la forme de code, les solutions de patrons comportent aspects statiques et dynamiques.	***statique, dynamique***
Généricité		Les opérations d'adaptation (autres que le renommage) sont implémentées dans le code du patron, par l'ingénieur de patrons. Quelques propriétés génériques sont proposées dans le méta-modèle de base.	***oui***
Variabilité		La gestion de la variabilité n'est pas traitée, bien qu'elle puisse être avec l'ajout de méthodes d'adaptation.	***non***
Nombre de Méta-Modèles		Un méta-modèle général est proposé. Il est amené à s'enrichir au fur et à mesure des ajouts de patrons.	***un pour tous les patrons***
Dépendance		L'imitation se faisant au final par codage, on peut parler d'instanciation des éléments constituant l'imitation.	***instanciation (codage)***

Tableau 2.8 : Récapitulatif de l'approche de (Albin-Amiot et al., 2002).

4.8 Travaux de Gerson Sunyé

Principes

Dans des travaux précédents la publication citée en 4.1, Gerson Sunyé a exploité la notion de variantes d'implémentation dans le cadre de la génération de code à partir de patrons (Sunye, 1999). Plus précisément, cette approche vise à expliciter les compromis d'implémentation de façon à permettre un choix de la part de l'ingénieur d'applications, avant de générer les codes adéquats. Pour ce faire, l'approche se base sur le langage MAC défini par un méta-modèle composé des trois principaux concepts utiles aux langages de programmation orientés-objet :

Méthode, Attribut et Classe. Chaque solution de patron sera exprimée à travers une sous-partie (appelée un plan) de ce méta-modèle.

Par exemple, pour le patron « Observateur », une méta-entité *Observateur* entretient quatre relations avec la méta-entité de base *Class*, de façon à représenter les quatre sortes de participants du patron : *Sujet*, *SujetConcret*, *Observateur* et *ObservateurConcret*. Les cardinalités appliquées à ces relations permettent d'exprimer le nombre possible d'imitations de chacun des participants. De plus, la méta-entité *Observateur* est dotée de rubriques exprimant les variantes du patron (cf. description du patron « Observateur », chapitre 1, section 2.2.b). Une rubrique *notification* explicite les deux alternatives *push* et *pull* de la notification décrite par les auteurs du patron. Lors de l'imitation du patron *Observateur*, l'ingénieur d'applications se verra proposer une liste de choix nommée *notification* et proposant donc deux choix : *push* et *pull*. (cf. figure 2.15).

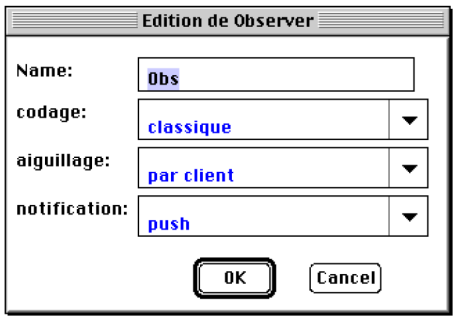


Figure 2.18 : Choix des compromis d'implémentation pour l'imitation du patron « Observateur » (extrait de (Sunye, 1999))

Si on replace cette utilisation de la variabilité d'implémentation dans la classification présentée dans la section 1, seule la notion de variantes alternatives est mise en œuvre dans cette approche. Cette approche traite également de la détection des imitations dans le code, par adjonction de règles de détection à chaque patron.

Tableau récapitulatif

Cible		L'approche traite de la définition et de l'usage des patrons.	***usage & définition***
USAGE	**Type**	L'ingénieur d'applications peut générer une imitation et détecter des patrons dans le code.	***génération-code détection-code***
	Processus	Le processus d'imitation, qui a pour but de générer du code, est assisté par un outil.	***assisté***
	Compétences	L'ingénieur d'applications se voit proposer des listes de choix quant aux variantes qu'il désire utiliser. Selon notre grille d'analyse, il n'y a pas vraiment de compétence associée.	***? (manipulation d'interface)***
DÉFINITION	**Processus**	Aucun processus de définition n'est présenté.	***non***
	Compétences	L'ingénieur de patrons doit savoir étendre le méta-modèle MAC.	***comprendre-méta-modèle, produire méta-modèle***

Complétude	Le code généré est fonctionnel. Néanmoins, la spécification dynamique du patron n'est pas apparente.	***Statique, dynamique (cachée)***
Généricité	Les cardinalités des participants de type méthodes, attributs et classes sont exprimables.	***oui***
Variabilité	Les compromis d'implémentation sont gérés sous la forme de variantes alternatives.	***oui (implémentation)***
Nombre de Méta-Modèles	Un méta-modèle générique, celui du langage MAC, est utilisé. Il est amené à s'enrichir au fur et à mesure des ajouts de patrons.	***un pour tous les patrons***
Dépendance	Il n'est pas fait mention de traçabilité entre le patron et le code généré.	***?***

Tableau 2.9 : Récapitulatif des travaux de Gerson Sunyé.

4.9 Autres approches

Il existe également d'autres approches centrées sur la réutilisation des patrons que nous n'avons pas détaillées dans cette étude. Dans (Maracano-Kamenoff et al., 2000) et (Blazy et al., 2003), la correction des imitations est garantie par l'utilisation du langage formel B et de la traduction (assimilable à une transformation) dans le langage UML.

Dans (Sunye et al., 2000), les auteurs proposent un outil destiné aux ingénieurs d'applications comprenant un cadriciel de transformation basé sur la méta-programmation. L'approche, à base de transformations, est similaire à celle proposée par (O`Cinneide, 2001) (cf. 4.3). L'imitation d'un patron est considérée comme une succession de transformations spécifiées, exprimées en langage OCL. Ce sont particulièrement les pré et postconditions qui garantissent la correction de l'imitation. Toutefois, les auteurs reconnaissent que l'approche transformationnelle, clairement assimilable à de la composition entre spécifications originales et imitation de patron, n'est valide que dans un objectif de « *réusinage* » de spécifications existantes. De plus, elle n'est applicable que pour des patrons pour lesquels on peut donner une description précise, sous forme de modèle, du contexte où ils s'appliquent.

Les travaux de thèse de Ouafa Hachani (Hachani, 2006) traitent de la spécification des patrons de conception à l'aide d'aspects et de leur réutilisation au sein des systèmes d'information. Les solutions sont composées de *préoccupations transversales* (*crosscuting concerns*) exprimant chacune une partie de la structure statique de la solution (classes, méthodes, etc..) ou même dynamique (sous la forme d'injection de code) qui sera tissé avec le système cible de la réutilisation. Si elle ne garantit pas la validité dans le temps de l'imitation, cette utilisation des aspects permet une première application précise de la solution à un contexte, que ce soit sous la forme d'un modèle statique ou de code.

5 SYNTHÈSE

Les approches présentées dans ce chapitre traitent toutes de la spécification des solutions de patrons en vue de l'imitation de ces derniers. A la lumière de notre cadre de comparaison, nous pouvons identifier quelques points communs et divergences significatifs.

Si tous ces travaux s'appuient sur les spécifications statiques des solutions, peu d'entre eux considèrent vraiment la dimension comportementale (France et al., 2004), à l'exception des approches orientées code (Albin-Amiot et Guéhéneuc, 2001), (Hachani, 2006).

La généricité des solutions est également très représentée, mais de manière hétérogène. En effet, bon nombre d'approches proposent des propriétés génériques qui leur sont propres, comme par exemple la notion de dimension de (Maplesden et al., 2001). Seuls les travaux de (France et al., 2004) semblent s'intéresser à la validité des imitations à plus long terme.

L'approche de (Sunye, 1999) exploite les variantes d'implémentation décrites dans les patrons du GoF et fournit un outil capable de générer le code correspondant aux choix faits par l'ingénieur d'applications. Toutefois, la généralisation de cette approche à d'autres patrons, ou à des variantes plus fonctionnelles ne nous a pas été possible, notamment en raison d'une description trop sommaire de la méthodologie de spécification des solutions.

Plus généralement, peu des approches étudiées donnent une description précise du processus de spécification de la solution. Les plus évolutives d'entre elles décrivent la manière d'ajouter des types de propriétés génériques (Albin-Amiot et Guéhéneuc, 2001). Seul le processus de définition de (Maplesden et al., 2001) est outillé. Le processus d'usage des patrons est quant à lui souvent décrit (Albin-Amiot et Guéhéneuc, 2001) mais parfois seulement outillé (Sunye, 1999), (Maplesden et al., 2001).

CHAPITRE 3 FORMALISATION DES PATRONS

Le premier chapitre a présenté les différents usages relatifs aux patrons en insistant sur la conception de systèmes à l'aide de patrons et sur les problèmes que cette activité soulève en terme de complétude, généricité, variabilité des solutions et de composition d'imitations. Le second chapitre a quant à lui décrit différentes approches traitant de la définition et/ou de l'usage des patrons. Ces approches ont notamment été comparées relativement aux aspects de complétude, variabilité et généricité des spécifications des solutions. Il ressort de ce comparatif que, dans le cadre des patrons, la variabilité a été peu étudiée et que, de manière générale ces trois aspects sont rarement combinés.

Ce constat, appuyé par les usages des patrons, nous amène dans ce troisième chapitre, à présenter nos propositions visant à permettre une spécification complète, variable et générique des solutions de patrons. Chacun de ces axes donne lieu à une section où nous décrivons les différents mécanismes proposés ainsi que le méta-modèle correspondant, en nous appuyant sur certaines remarques et patrons présentés dans le cas d'étude.

Afin de faciliter la production de telles spécifications, nous proposons dans la quatrième section un processus destiné à guider les ingénieurs de patrons pour une prise en compte de ces trois axes lors de la spécification de leurs solutions. Le patron « Composite » du GoF sert de démonstration finale à ce chapitre.

1 Complétude : la solution comme un mini-système

Le cas d'étude, présenté dans le premier chapitre, a mis en avant la nécessité de prendre en compte différentes facettes des solutions de patrons, et en particulier de ne pas se limiter aux aspects statiques. La première proposition de ce travail est donc d'intégrer des spécifications fonctionnelles et comportementales aux solutions de patrons de façon à en tirer partie lors de l'imitation.

Le domaine du développement des systèmes est riche de méthodes s'appuyant sur les différentes vues des systèmes (RUP, 2TUP, ...). Trois de ces vues complémentaires sont particulièrement récurrentes :

- la vue fonctionnelle. Dans la plupart des méthodes basées sur UML, les diagrammes de cas d'utilisation tiennent un rôle capital dans la description intentionnelle des systèmes.
- la vue dynamique. Beaucoup de méthodes déclinent le comportement du système sous la forme de scenarii de haut niveau d'abstraction qui décrivent le comportement attendu au sein des fonctionnalités. Ces scenarii sont par la suite décomposés et raffinés sous la forme, par exemple, de diagrammes de séquences plus concrets.
- la vue statique. Elle permet quant à elle de représenter les entités du système ainsi que les propriétés et contraintes jusqu'alors inexprimées à travers les autres vues, comme par exemple les cardinalités des associations entre classes. Dans les méthodes de développement objet, les diagrammes de classes permettent également de travailler sur la persistance des informations et ainsi de définir le modèle relationnel à mettre en œuvre.

Nous considérons la solution d'un patron comme un mini-système décrit par ces trois vues, avec le langage UML.

1.1 Vue des cas d'utilisation : les fonctionnalités d'un patron

Un modèle de cas d'utilisation présente les fonctionnalités du système en cours de construction, les dépendances qui les relient et les acteurs qui les déclenchent. Si la solution d'un patron apporte plusieurs fonctionnalités, nous proposons de les faire apparaître explicitement.

En ce qui concerne « Observateur », on peut déduire deux fonctionnalités : *modifier le sujet* (subject modification) et *gérer les observateurs* (observers management). La première consiste en la modification du sujet avec mise à jour des observateurs, la seconde traite de l'ajout et de la suppression des observateurs au sujet. La figure 3.1 illustre la vue des cas d'utilisation pour ce patron.

L'acteur de la vue des cas d'utilisation correspond au « client » des patrons du GoF. Ce dernier spécifie les points d'entrée qu'il utilise pour accéder à la solution. L'acteur *client* est donc l'entité qui déclenche les fonctionnalités.

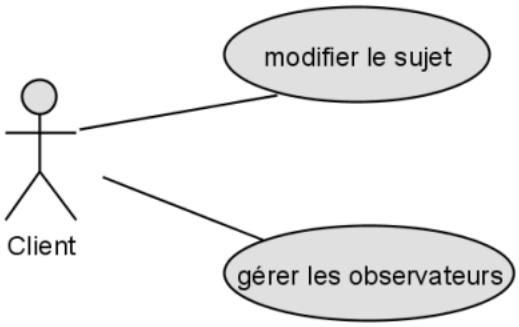


Figure 3.1 : Observateur : vue des cas d'utilisation.

1.2 Vue dynamique : diagrammes de séquence UML2

Dans la rubrique *Collaborations*, le GoF propose un diagramme de séquence décrivant le cas d'utilisation que nous avons appelé *modifier le sujet*. La figure 3.2 présente ce diagramme de séquence. On notera que la répétition de la sous-séquence *miseAJour/acqEtat* sur les deux observateurs cherche à illustrer l'algorithme itératif de la méthode *notifier*.

Dans ce diagramme, c'est un observateur qui modifie le sujet mais, a priori, n'importe quel objet pourrait en faire de même. C'est pourquoi la solution proposée en figure 3.3 nous semble plus générale car n'importe quelle entité pourra jouer le rôle du client, y compris un observateur concret.

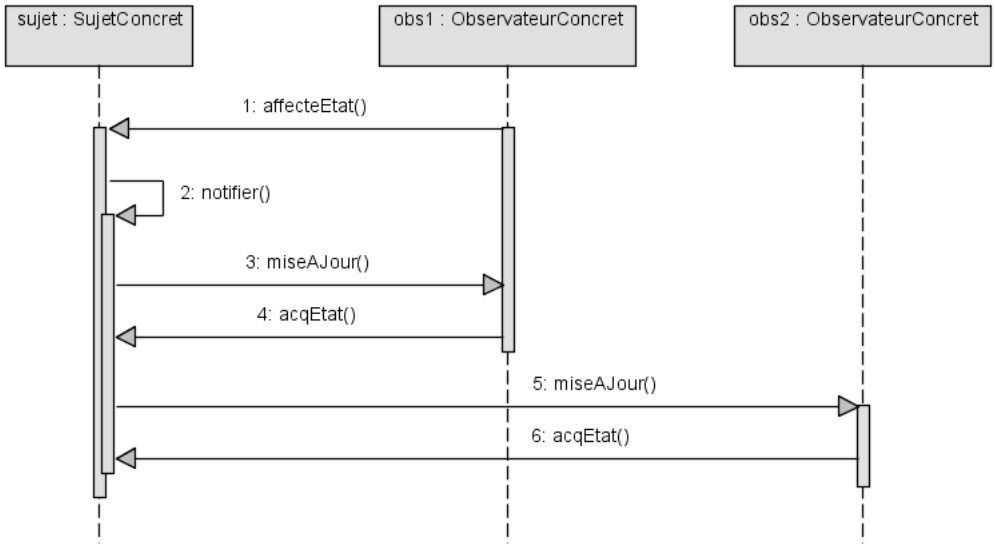


Figure 3.2 : Observateur : diagramme de séquence du GoF

Les diagrammes de séquence d'UML2 (OMGa, 2005) apportent beaucoup à la spécification du comportement. Ainsi, le parcours de tous les observateurs de l'algorithme implémentant la méthode *notifier* peut être modélisé à l'aide d'un fragment combiné muni d'un opérande de boucle (loop). Nous utilisons une « référence d'interaction » (interaction use) et le mécanisme des portes (gate) pour référencer le diagramme de séquence de la méthode *notifier*.

La figure 3.3 illustre notre adaptation des séquences *modifier le sujet* et *notifier* avec UML2. Nous complétons la solution originale avec la méthode *majEtat* afin de représenter le changement d'état du sujet. Ce que réalise précisément cette méthode n'est pas une préoccupation de ce patron. Le concepteur qui imitera ce patron aura à sa charge de définir cette méthode mais ne pourra remettre en cause son emplacement chronologique dans la séquence *affecteEtat*. Nous adoptons la même approche en ce qui concerne l'état de l'observateur, mis à jour dans la méthode *miseAJour* en utilisant sa méthode privée *majEtat*.

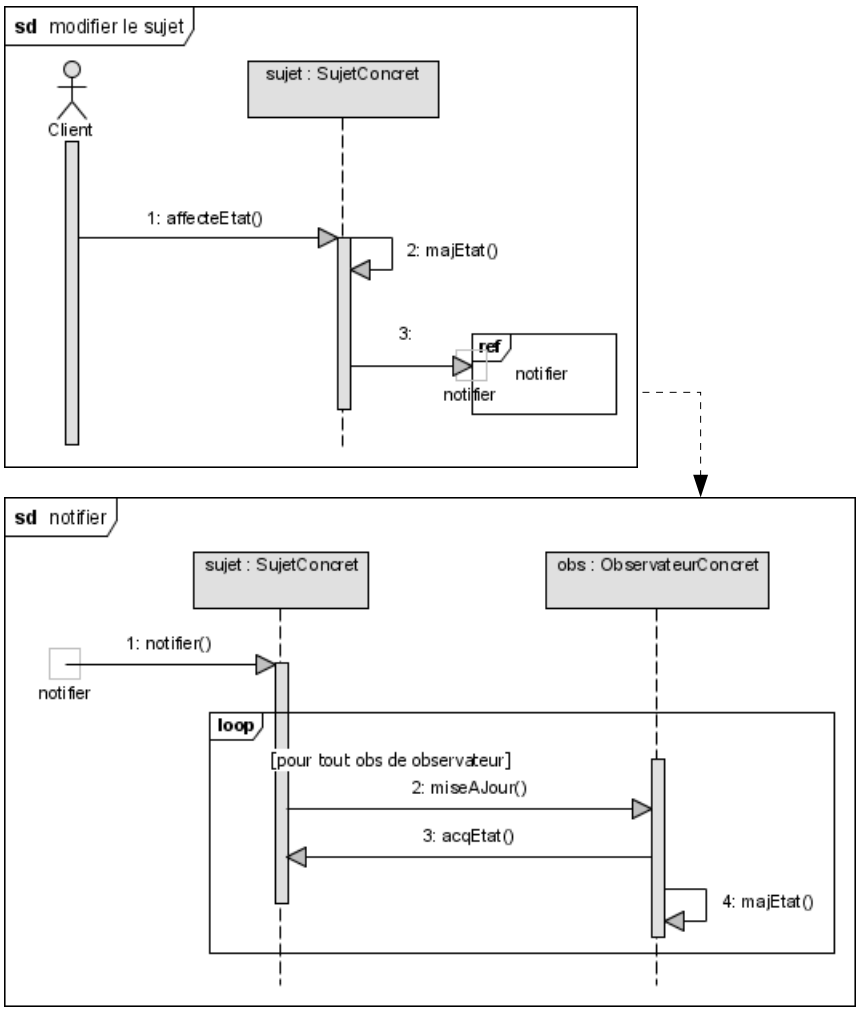


Figure 3.3 : Observateur : diagrammes de séquences avec UML2

1.3 Relations entre vue fonctionnelle et vue dynamique

L'expression du comportement défini au sein du patron *Observateur* a mis en avant l'importance de la notification ainsi que sa dépendance à la modification du sujet. Nous revenons maintenant sur la vue fonctionnelle du patron, afin de préciser l'inclusion du cas d'utilisation *notifier* à *modifier le sujet* (cf. figure 3.5).

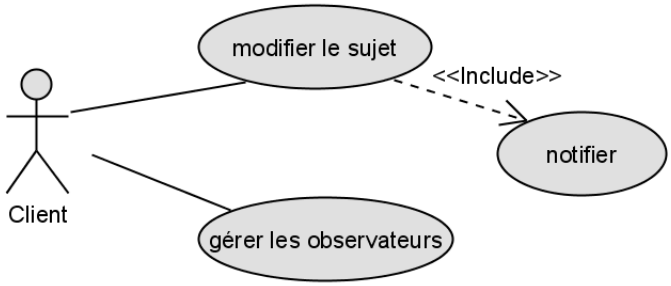


Figure 3.4 : Observateur : nouvelle vue fonctionnelle

Si la décomposition comportementale autorise l'ingénieur de patrons à revenir sur la spécification fonctionnelle de son patron, il est clair que cette dernière doit rester assez abstraite puisque les cas d'utilisation présentés dans la vue fonctionnelle doivent donner un aperçu global de la solution. Néanmoins, il faut que chaque cas d'utilisation soit affiné par un fragment d'interaction de la vue dynamique.

1.4 Vue statique : Diagrammes de classes

Les diagrammes de classes expriment la structure des entités du système ainsi que la réalisation de leurs relations. Cette structure est en partie déduite de la vue dynamique, mais le concepteur y précise des propriétés statiques, par exemple les cardinalités des associations.

La plupart des solutions de patrons se résument à une vue statique, en général un diagramme de classes. Ces structures sont souvent accompagnées de notes textuelles qui apportent quelques précisions. Cela peut aller du simple commentaire à l'algorithme. Dans le cas d' « Observateur », l'algorithme de la méthode *notifier* est décrit à l'aide d'un pseudo-code.

En considérant les nouveaux diagrammes de séquence de la vue dynamique, nous proposons une structure statique (cf. figure 3.5, à droite) de la solution du patron « Observateur », dont les différences avec la solution originelle du GoF (cf. figure 3.5, à gauche) sont présentées ci-dessous.

- Dans notre approche, un algorithme est représenté dans la vue dynamique. C'est le cas pour la méthode *notify* et sa note explicative devient donc redondante.
- Nous avons ajouté à la vue dynamique (cf. figure 3.3) une méthode (privée) de modification de l'état dont le contenu dépendra du contexte d'imitation, tout comme l'attribut *etatSujet* en dépend. C'est pourquoi cet attribut n'est plus présent dans la vue statique. Cependant, nous verrons dans la section 3 traitant de la généricité comment exprimer le fait que le sujet doit comporter un état.

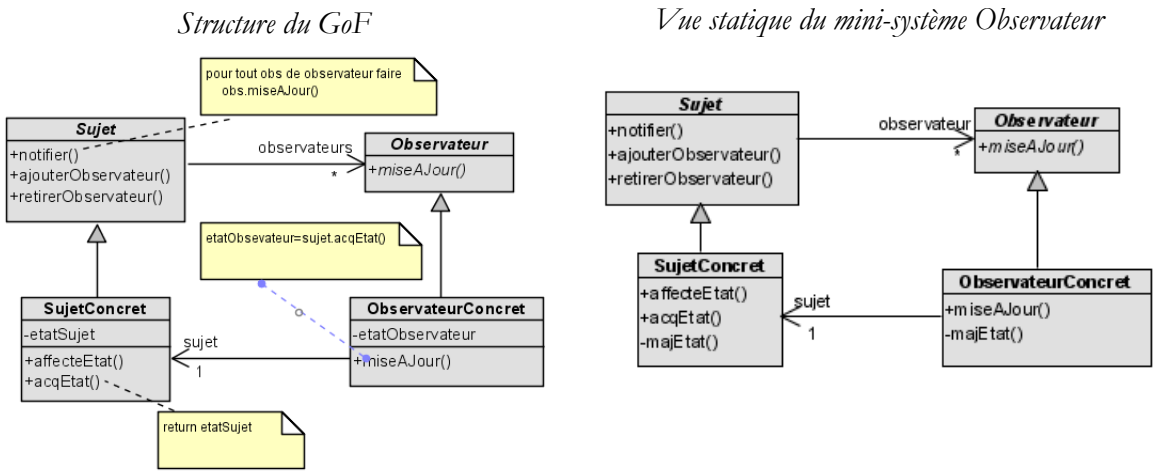


Figure 3.5: Observateur : vue statique.

1.5 Les limites du mini-système

Une approche à vues multiples permet d'exprimer plus complètement la solution d'un patron. Nous pourrions considérer que l'imitation de telles solutions apporterait une meilleure qualité de réutilisation. Pourtant, il est des informations que notre mini-système ne prend toujours pas en compte, par exemple le fait que les fonctionnalités offertes par la solution d'un patron soient essentielles ou facultatives, ainsi que les variantes données de manière textuelle par l'ingénieur de patrons. D'autre part, l'encapsulation de la solution dans un mini-système ne permet toujours pas d'exprimer les propriétés génériques de la solution, comme par exemple le fait qu'une classe peut, ou non, être imitée plusieurs fois.

Dans les sections suivantes, nous proposons d'identifier et d'exprimer la variabilité et la généricité des solutions de patrons, à partir de mini-systèmes.

2 VARIABILITÉ

Nous avons vu dans le premier chapitre qu'un patron peut proposer des variations de la solution nominale, ces dernières étant exprimées de manière textuelle. Pour illustrer l'intérêt de la variabilité au sein des patrons, nous nous appuyons sur deux thèmes évoqués dans la rubrique « Implémentation » du patron « Observateur » : le protocole de mise à jour et le déclenchement de cette dernière (cf. chapitre 1, section 2.2.c).

Les auteurs précisent que le protocole de mise à jour pourrait être différent de celui nominalement spécifié, en donnant directement, lors de l'appel à la méthode *miseAJour*, les informations nécessaires aux observateurs. Ce protocole implique une spécification différente de la solution, et en particulier au niveau de la vue dynamique, puisque l'appel à la méthode *acqEtat* n'aurait plus sens. D'ailleurs, ce protocole impacte également la vue statique, puisque qu'il n'y aurait aucun intérêt à conserver cette méthode. Cette variante propose en fait une variante d'implémentation du patron, puisque du point de vue du *Client*, rien n'est différent (cf. chapitre 1, section 5.3).

Les auteurs discutent également du déclenchement de la mise à jour. Dans la solution nominale du patron, la notification des observateurs est exécutée systématiquement après le changement d'état du *Sujet*. Ce déclenchement « implicite » (cohérent avec l'intention d'automatisation du patron) peut être remplacé par un déclenchement commandé par le *Client*. Cette variante prend tout son sens lorsque l'état du *Sujet* est complexe et que sa modification nécessite par exemple plusieurs appels de méthodes. Dans ce cas, cette approche plus « transactionnelle » est préférable, et les observateurs ne doivent être informés du changement d'état du sujet qu'une fois ce dernier complètement effectué. Cette variante, bien que décrite dans la rubrique « Implémentation », se situe à un niveau d'abstraction plus élevé, dans les fonctionnalités du patron.

Il peut également y avoir, au sein même de la solution d'un patron, des propriétés faiblement couplées aux objectifs « principaux » du patron. Ainsi, contrairement à l'intention précisée par le *GoF*, la solution du patron « Observateur » ne se limite pas à mettre en œuvre la notification des observateurs mais permet également d'attacher ou de détacher ces derniers à un sujet (cas d'utilisation *gérer les observateurs*). La solution devrait donc introduire le fait que la gestion des observateurs ne constitue qu'une fonctionnalité secondaire qui peut ne pas être retenue lors de l'imitation.

Conformément aux principes de la variabilité évoqués dans le chapitre 2, les trois paragraphes précédents décrivent en fait trois points de variations qu'il s'agit de définir au sein de la solution du patron. Toutefois, dans le cadre de cette thèse, nous nous intéressons uniquement aux points de variations « fonctionnels », et donc, pour le patron « Observateur » aux variations traitant du déclenchement de la mise à jour et à l'expression des fonctionnalités principales et secondaires.

2.1 Représenter la variabilité dans la vue fonctionnelle

La vue fonctionnelle semble être l'endroit idéal pour intégrer la définition de ces deux points de variation au sein de notre mini-système (i.e. la solution du patron). En repartant de la figure 3.4, nous devons plus précisément spécifier que le cas d'utilisation *notifier* est en fait un point de variation proposant deux variantes alternatives : *notifier explicitement* et *notifier implicitement*. Pour ce faire, nous utilisons une notation similaire à celle proposée par (Van der Maßen et Lichter, 2002), basée sur deux stéréotypes : « variation » et « variant » définissant respectivement un cas d'utilisation comme point de variation et variante. Nous utilisons une dépendance stéréotypée « alternative » pour exprimer le type de variation lié à ces deux variantes.

Tel que nous l'avons décrit, le cas d'utilisation *gérer les observateurs* s'apparente plutôt à une fonctionnalité secondaire du patron, facultative lors de l'imitation, par opposition à *modifier le sujet* fonctionnalité principale et obligatoire lors de l'imitation. Notre approche est particulièrement tournée vers l'utilisation des patrons, et donc l'ingénieur d'applications. C'est pourquoi nous préférons utiliser les termes « obligatoire » et « facultatif » et proposons à l'ingénieur de patrons de spécifier cette propriété à l'aide de deux stéréotypes « obligatoire » et « facultatif ». Cette propriété est uniquement applicable sur les fonctionnalités de premier niveau, c'est-à-dire celles qui sont directement associées au client.

La figure 3.6 ci-dessous illustre la vue fonctionnelle du patron « Observateur », en tenant compte des différents aspects variables évoqués ci-dessus.

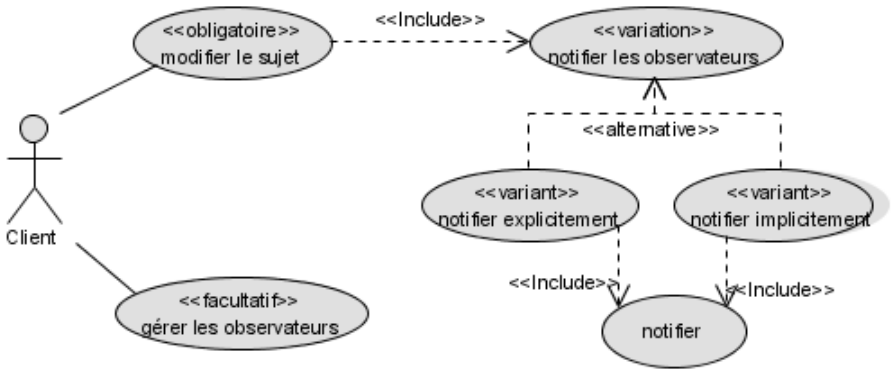


Figure 3.6 : Vue fonctionnelle à variantes du patron « Observateur »

2.1.a Un opérateur générique pour la variabilité

Le langage de spécification de la variabilité fonctionnelle proposé ci-dessus peut être généralisé sous la forme d'un opérateur générique. Cet opérateur est basé sur les concepts fondamentaux de la variabilité, c'est à dire un *point de variation* associé à plusieurs *variantes*. Toutes les variantes associées à un point de variation le sont selon un même type de variabilité, et par inclusion. Le type de variabilité (alternative, optionnel)[2] est exprimé à l'aide de deux cardinalités, minimale et maximale et doit être conforme aux définitions données au chapitre 2 :

- Si la cardinalité minimale est égale à 0, il s'agit alors de variantes optionnelles, et la cardinalité maximale indique le nombre d'options potentiellement imitables. Comme expliqué dans le chapitre 2, le regroupement des variantes optionnelles est possible, en utilisant une cardinalité maximale *, ce qui revient à dire *« au maximum, le nombre total de variantes »*.
- Si les cardinalités minimale et maximale sont égales à 1, il s'agit de variantes alternatives. Il faudra obligatoirement sélectionner l'une d'entre elles.

La figure 3.7 ci-dessous présente l'utilisation de l'opérateur générique sur une *fonctionnalité A*, admettant deux alternatives ainsi que deux options. Seules les combinaisons suivantes sont autorisées : *fonctionnalité A + alternative 1*, *fonctionnalité A + alternative 2*, *fonctionnalité A + alternative 1 + option 1*, *fonctionnalité A + alternative 1 + option 2*, *fonctionnalité A + alternative 2 + option 1*, *fonctionnalité A + alternative 2 + option 2*, *fonctionnalité A + alternative 1 + option1 + option 2*, *fonctionnalité A + alternative 2 + option 1 + option 2*.

Afin qu'un seul type de variabilité soit associé à chaque point de variation, les options sont extraites et définies au sein d'un cas d'utilisation abstrait dédié. Après la sélection des variantes, ce dernier n'aura plus d'intérêt.

La figure 3.8 illustre une syntaxe équivalente et plus accessible de cet opérateur, à l'aide des stéréotypes de dépendance. Cette notation, plus légère, sera préférée à la précédente, notamment lors des interactions avec les ingénieurs de patrons et d'applications.

2 Dans cette thèse, nous ne traitons que des variantes optionnelles ou alternatives. Les autres types de variabilité cités dans le chapitre 2 ne sont pas considérés.

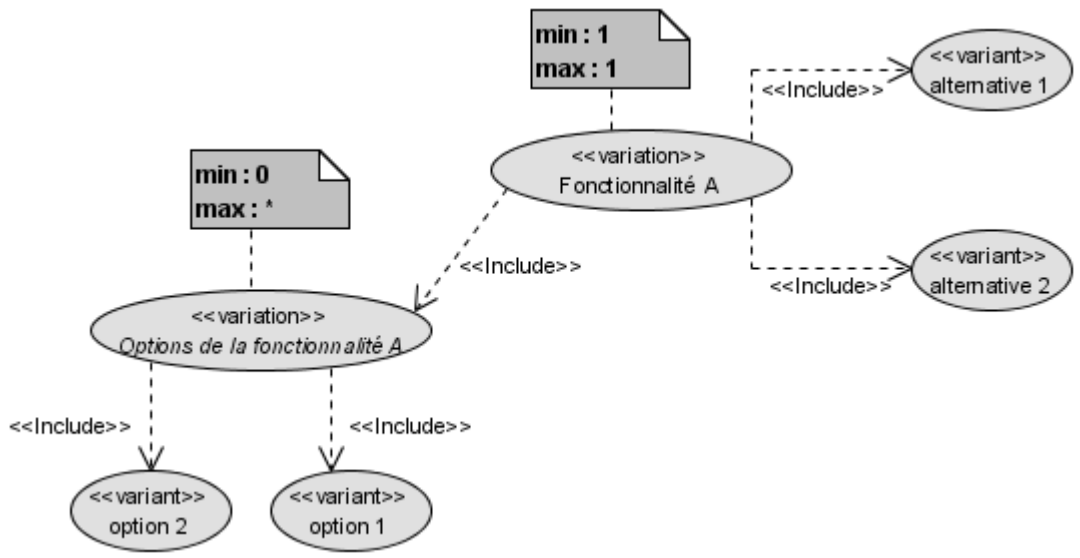


Figure 3.7 : Exemple d'application de l'opérateur générique pour la variabilité fonctionnelle.

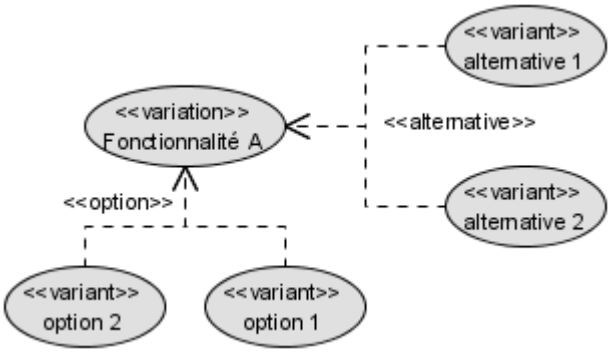


Figure 3.8 : Exemple d'application de l'opérateur générique pour la variabilité fonctionnelle en utilisant une syntaxe simplifiée.

2.1.b Application de l'opérateur générique à la propriété obligatoire/facultatif

Si l'utilisation de cet opérateur pour exprimer les variabilités de type optionnel et alternative semble évidente, son application à la notion obligatoire/facultatif nécessite une adaptation de la vue fonctionnelle. En effet, une fonctionnalité facultative peut être considérée comme une variante optionnelle du patron.

Nous complétons donc la vue fonctionnelle d'un patron par un cas d'utilisation *racine* qui inclura les fonctionnalités principales et sera un point de variation pour toutes les fonctionnalités facultatives. Plus précisément, la racine est un point de variation qui porte le nom du patron et qui est le seul cas d'utilisation en relation avec le *Client*. Une racine est toujours définie comme abstraite et ne peut admettre que des variantes de type optionnel (les fonctionnalités facultatives), les cardinalités sont donc toujours *0..**. La figure 3.9 ci-dessous montre l'application de l'opérateur générique à la vue fonctionnelle du patron « Observateur ».

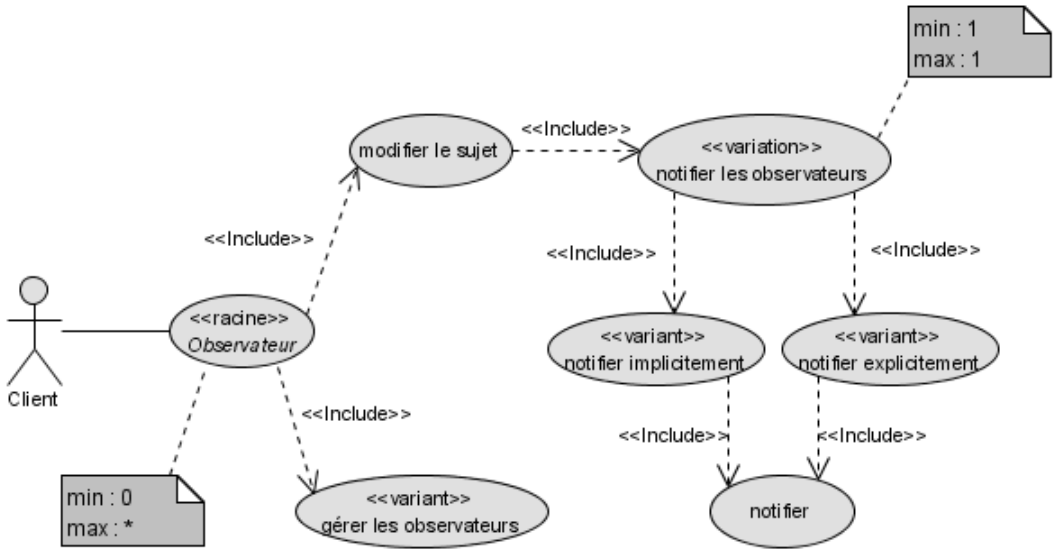


Figure 3.9 : Vue fonctionnelle à variantes du patron « Observateur ».

Puisque le point de variation abstrait sert à dénombrer les fonctionnalités facultatives et que les fonctionnalités incluses dans la racine sont principales, on obtient, en utilisant la syntaxe « intuitive », un diagramme équivalent à celui de la figure 3.6 : les fonctionnalités incluses par la racine porteront le stéréotype « obligatoire » tandis que les variantes de la racine porteront le stéréotype « facultatif ».

2.2 Le mini-système à variantes

Dans notre approche, les solutions de patrons sont exprimées sous forme de mini-systèmes (cf. section 1) et la définition de la variabilité ne doit pas se limiter à la vue fonctionnelle. Les variantes, sont réalisées à l'aide des propriétés dynamiques et statiques impossibles à exprimer dans un diagramme de cas d'utilisation. C'est pourquoi l'expression de la variabilité ne peut se faire qu'à travers une combinaison des trois vues du mini-système, guidée toutefois par la vue fonctionnelle. La logique de construction des mini-systèmes implique que la variabilité dans la vue statique soit traitée consécutivement à la variabilité dynamique.

Nous commençons par détailler les concepts nécessaires à l'expression de la variabilité dynamique et nous proposons une manière de les appliquer aux mini-systèmes exprimés avec UML2. Nous détaillons ensuite l'application de la variabilité sur la vue statique.

2.2.a Variabilité dynamique : concepts

Dans un mini-système, la vue dynamique a pour objectif de préciser la réalisation des éléments de la vue fonctionnelle au sein du déroulement des scénarii. Dans un contexte variable, la vue dynamique se doit de préciser comment les comportements associés aux variantes sont organisés chronologiquement et comment ils s'enchaînent. Au sein de la vue dynamique, la variabilité doit permettre :

- de définir le comportement des variantes ;
- d'exprimer une contrainte chronologique entre les variantes d'un même point de variation ;

- de définir l'impact du point de variation sur le scénario dans lequel il intervient.

Nous définissons un ensemble de règles (cf. *Variabilité dynamique : règles de traduction*) qui, à partir d'un modèle fonctionnel exprimé à l'aide de l'opérateur générique, permet de construire un cadre de définition du comportement. L'ingénieur de patrons devra par la suite compléter ce cadre en spécifiant le comportement qu'il attend.

Nous adaptons la règle énoncée en 1.3, imposant pour chaque fonctionnalité un bloc d'interaction dédié. En effet, l'utilisation de l'opérateur générique donne lieu, dans la vue fonctionnelle, à des cas d'utilisation abstraits tels que la racine et les points de variation abstraits, spécifiques aux options et fonctionnalités facultatives. Les cas d'utilisation abstraits ne donnent lieu à aucune description dynamique, contrairement à tous les autres cas d'utilisation.

La vue fonctionnelle ne donne aucune information sur la chronologie au sein d'une fonctionnalité. C'est pourquoi l'ingénieur de patrons doit préciser, en plus de leur contenu, l'ordonnancement des différents blocs d'interaction lorsqu'il y a inclusion de plusieurs cas d'utilisation.

Les propositions ci-dessus doivent être mises en œuvre dans la vue dynamique du mini-système, et donc, à l'aide d'UML2. Plus particulièrement, c'est autour du concept d'interaction (type *Interaction* d'UML2) que la variabilité dynamique va s'articuler.

Le concept d'interaction est défini comme suit dans la norme UML2 (OMGa, 2005) : « *Une Interaction est une unité de comportement qui décrit des échanges d'informations [...]* ». C'est également à partir des interactions qu'est traité, dans (Jézéquel et Ziadi, 2005), le problème de la spécification comportementale des lignes de produits logiciels. La figure 3.10 ci-dessous présente une partie du méta-modèle UML2 concernant la structure des interactions, éléments essentiels à la spécification du comportement sous la forme de diagrammes de séquences.

Notre approche s'appuie par la suite sur l'utilisation de fragments combinés (*CombinedFragment*, figure 3.10), dont nous précisons ici les principes. C'est un type de fragment d'interaction qui comporte un opérateur ainsi qu'une ou plusieurs opérandes. Nous détaillons quelques-uns des opérateurs disponibles (*InteractionOperator*, figure 3.10).

- L'opérateur *loop* associé à un seul opérande, permet d'exprimer une boucle (cf. interaction *notifier* section 1.2).
- L'opérateur *opt* permet de spécifier qu'un fragment d'interaction (opérande) est optionnelle (à l'aide d'une contrainte).
- L'opérateur *alt* est destiné à la définition de structures de choix conditionnels entre plusieurs fragments (opérandes).
- L'opérateur *seq* indique que différents opérandes doivent être exécutées en séquence, mais dans n'importe quel ordre.
- L'opérateur *strict* exprime lui aussi une séquence, mais selon l'ordonnancement donné.

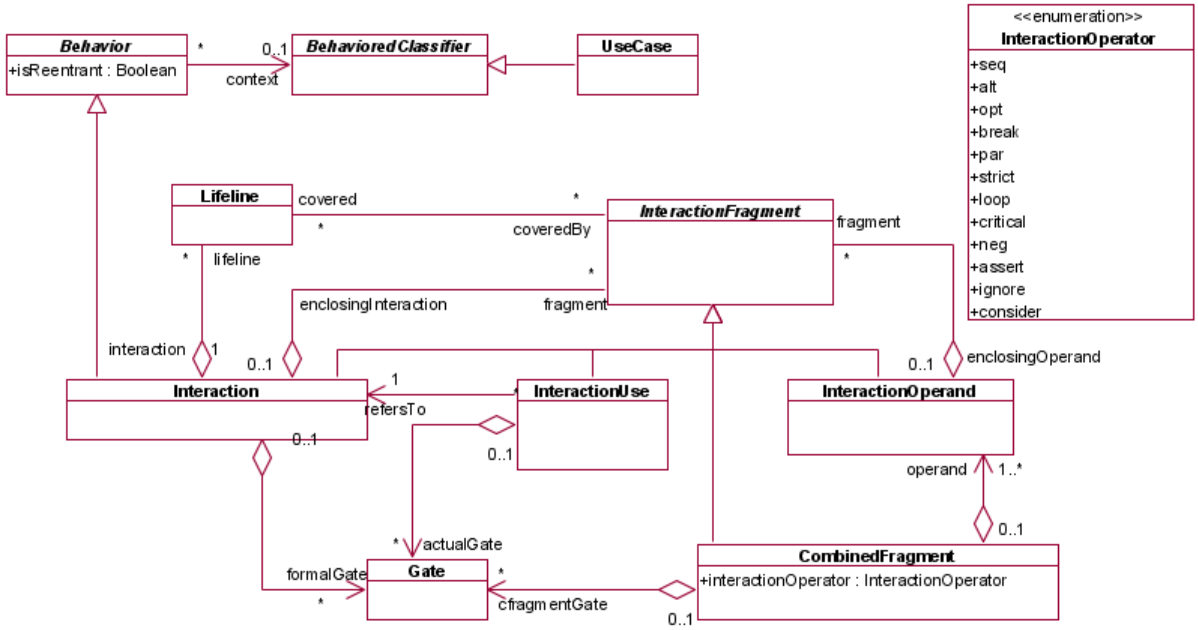


Figure 3.10 : Méta-modèle UML2 : Interactions.

La définition des règles de construction de la variabilité dynamique passe par la mise en correspondance des concepts. Le concept de bloc d'interaction est assimilé à celui d'interaction (*Interaction*) d'UML2. En tant que *Behavior*, une *Interaction* peut être associée à un cas d'utilisation, ce dernier devenant le contexte (rôle *context,* figure 3.10*)* de l'interaction. Il en résulte une règle : dans le mini-système, tout cas d'utilisation concret est associé à une interaction UML2, qui décrit sa réalisation comportementale.

2.2.b Variabilité dynamique : règle de traduction

L'opérateur générique de variabilité fonctionnelle est basé sur un seul type de dépendance entre les fonctionnalités : l'inclusion. Néanmoins la « traduction » dynamique de l'inclusion est traitée différemment selon le type du cas d'utilisation incluant. Nous dégageons trois cas distincts.

- Le cas d'utilisation incluant n'est pas un point de variation. Il s'agit alors d'une inclusion classique, que nous traduisons en une référence d'interaction UML2, comme dans la section 1.2. La figure 3.11 ci -dessous illustre cette traduction.

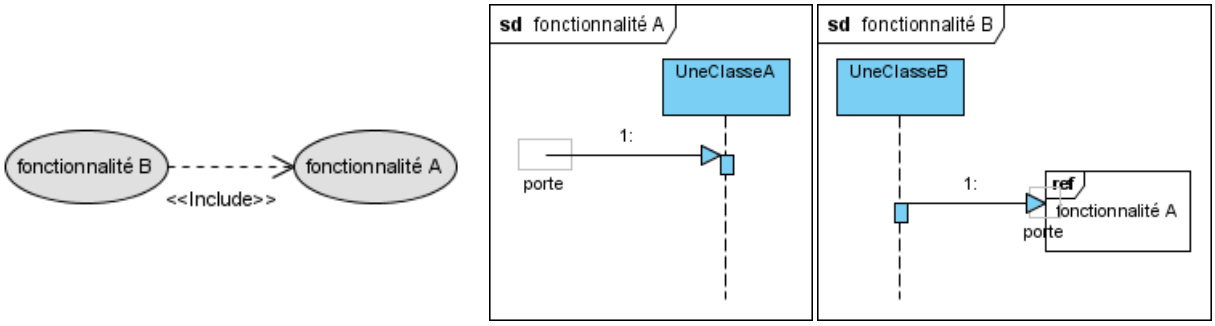


Figure 3.11 : Traduction dynamique d'une inclusion fonctionnelle classique.

- Le cas d'utilisation incluant est un point de variation concret. Selon le type de variabilité, exprimé par les cardinalités, l'inclusion prendra une forme différente.

– Toutes les variantes alternatives du point de variations sont regroupées au sein d'un fragment combiné de type *alt*. Plus précisément, chaque variante donne lieu à un opérande de ce fragment combiné qui contiendra une référence d'interaction vers l'interaction de la variante incluse. La figure 3.12 ci-dessous illustre cette traduction.

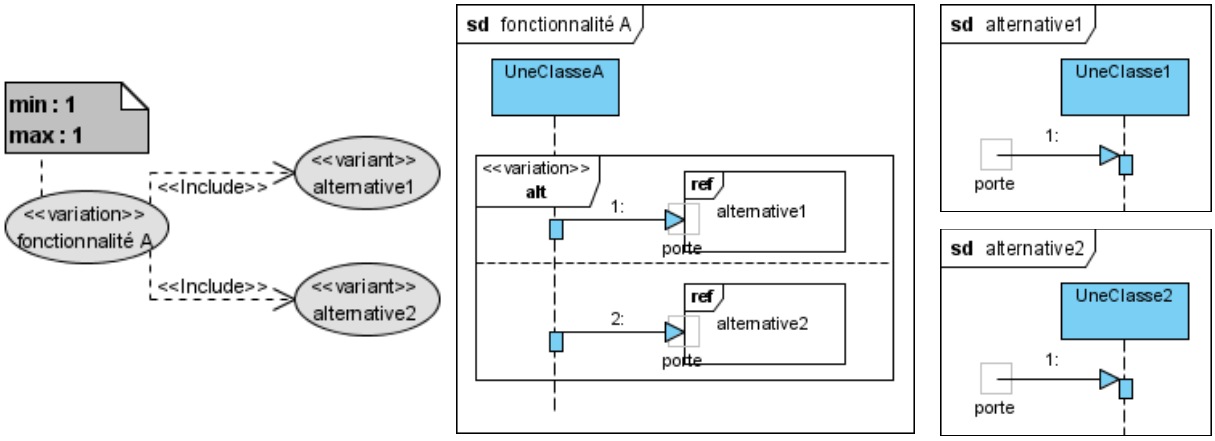


Figure 3.12 : Traduction dynamique d'un point de variation et ses variantes alternatives.

– Les variantes optionnelles sont traduites sous la forme d'un fragment combiné de type *strict*, comportant un seul opérande incluant une référence vers l'interaction de la variante incluse. Tous les fragments *strict* sont regroupés dans un fragment *seq,* correspondant au point de variation. La figure 3.13 montre la traduction d'un point de variation à deux options.

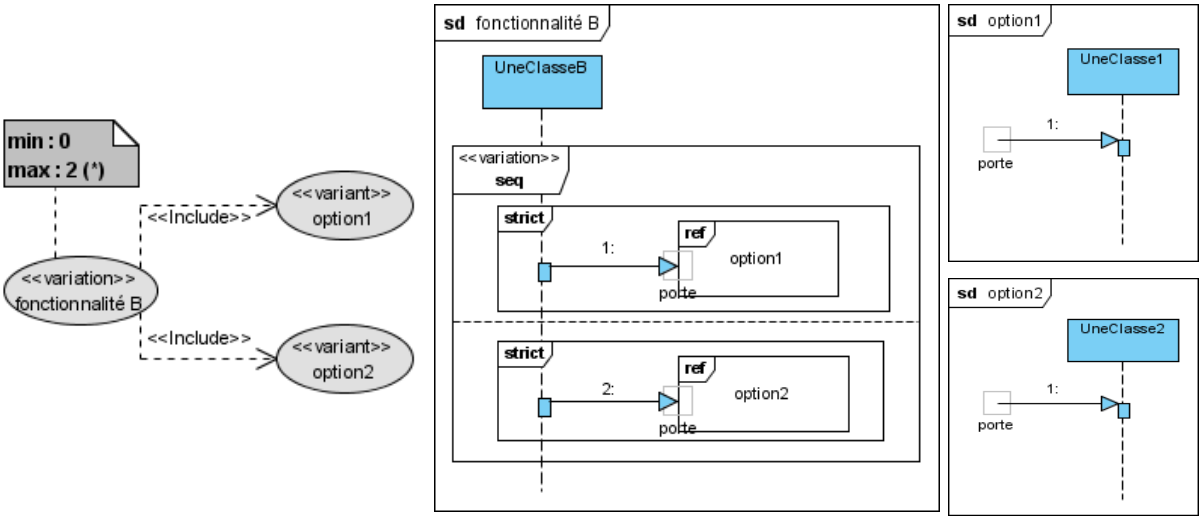


Figure 3.13 : Traduction dynamique d'un point de variation avec ses options.

- Le cas d'utilisation incluant est un point de variation abstrait. Nous rappelons que la cardinalité d'un point de variation abstrait est toujours *0..**. Il y a deux cas possibles :
  – Il s'agit de la racine. Dans ce cas, l'interaction de chaque fonctionnalité directement incluse (i.e. fonctionnalité obligatoire) ou variante (ie. fonctionnalité facultative) comporte une ligne de vie représentant le client, d'où part le premier message de sa séquence. Dans la figure 3.14 ci-dessous,

fonctionnalité B est obligatoire, ce qui ne l'empêche pas d'être aussi un point de variation, en l'occurrence celui traité dans l'exemple précédent (cf. figure 3.13). Par contre, *fonctionnalité C* est optionnelle (cf. cardinalités) et donc facultative.

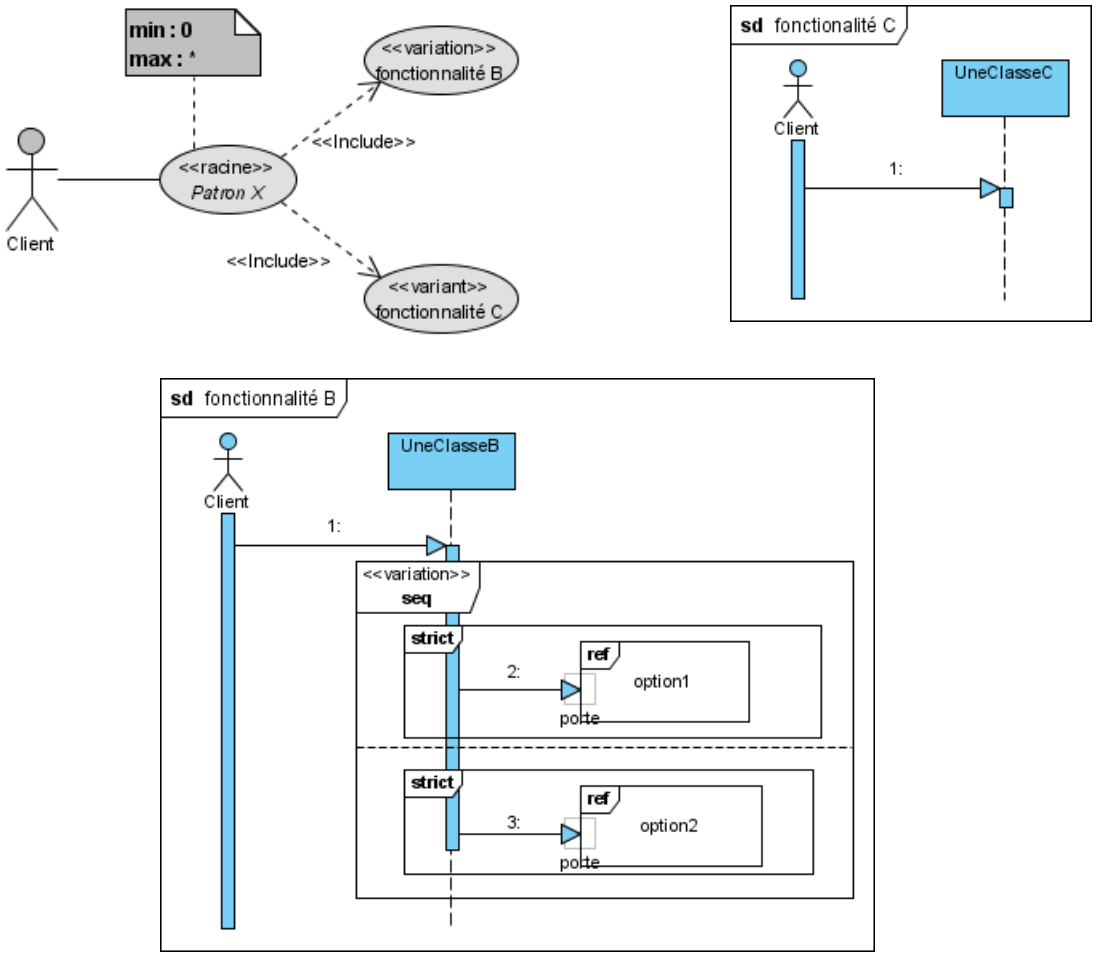


Figure 3.14 : Traduction dynamique d'une racine.

- Ce n'est pas la racine. Dans ce cas, il y a un point de variation concret qui inclut le cas d'utilisation considéré. Cela permet d'exprimer, sur le même point de variation, des variantes optionnelles en plus de variantes alternatives. De manière similaire aux variantes optionnelles d'un point de variation concret, un fragment combiné de type *seq* inclura, pour chaque variante, un fragment *strict* à un seul opérande, incluant une référence. Ce fragment *seq* sera contenu par le point de variation concret incluant. La figure 3.15 illustre le cas d'un point de variation comportant des alternatives et des options placées sur un point de variation abstrait.

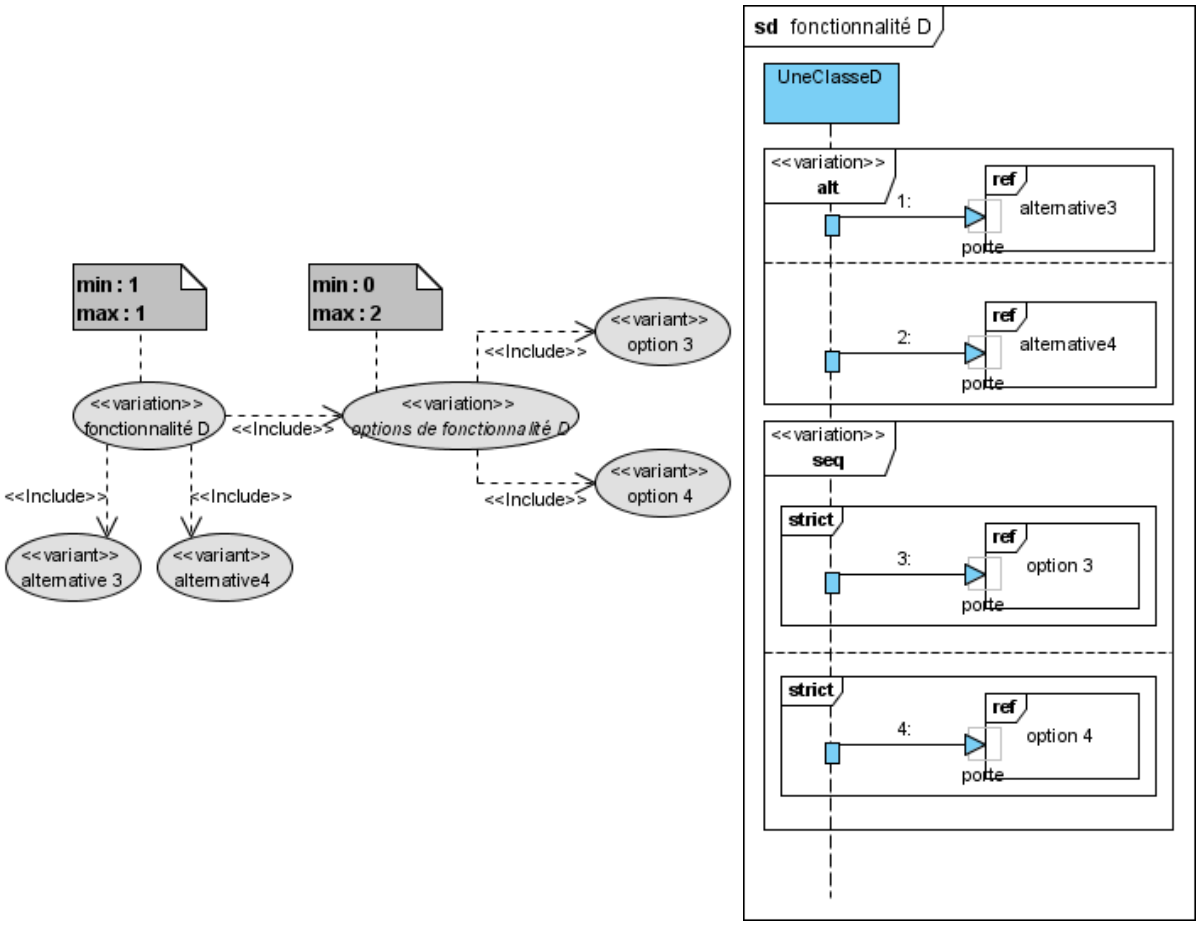


Figure 3.15 : Traduction dynamique d'un point de variation regroupant alternatives et options

Le figure 3.15 ci-dessus illustre néanmoins un problème majeur de la traduction systématique de la vue fonctionnelle en vue dynamique : la chronologie. En effet, rien n'est dit, dans la vue fonctionnelle, sur l'ordonnancement des variantes entre elles. Dans l'exemple ci-dessus, l'option *option 3* pourrait tout aussi bien se dérouler avant l'alternative ou après *option 4*. La chronologie des inclusions au sein du cas d'utilisation incluant reste donc à la charge de l'ingénieur de patrons. Au regard des règles définies ci-dessus, l'expression de cette chronologie se fera aisément par interclassement des fragments. Bien entendu, si les fragments *seq* englobant des variantes optionnelles sont divisibles et répartissables, ceux des fragments *alt* utilisés pour les variantes alternatives ne le sont pas, en raison de la notion même d'alternative.

La figure 3.16 montre l'application de ces règles au patron « Observateur », à partir de la vue fonctionnelle de la figure 3.9. On y retrouve uniquement cinq interactions, la racine étant abstraite et l'interaction de *notifier* étant la même que celle présentée en figure 3.3. La norme UML2 n'imposant pas d'utiliser systématiquement une porte pour « communiquer » avec un fragment référencé, les références vers les interactions des points de variation et variantes ne font que couvrir les lignes de vie qu'elle manipulent[3].

3 A l'instar de la référence à *notifier implicitement* dans *notifier les observateurs* qui ne couvre que la ligne de vie *Sujet*. En effet, la norme UML2 précise que : « *La référence d'interaction (InteractionUse) doit couvrir toutes les lignes de vie de l'interaction contenante qui apparaissent dans l'interaction référencée.* ».

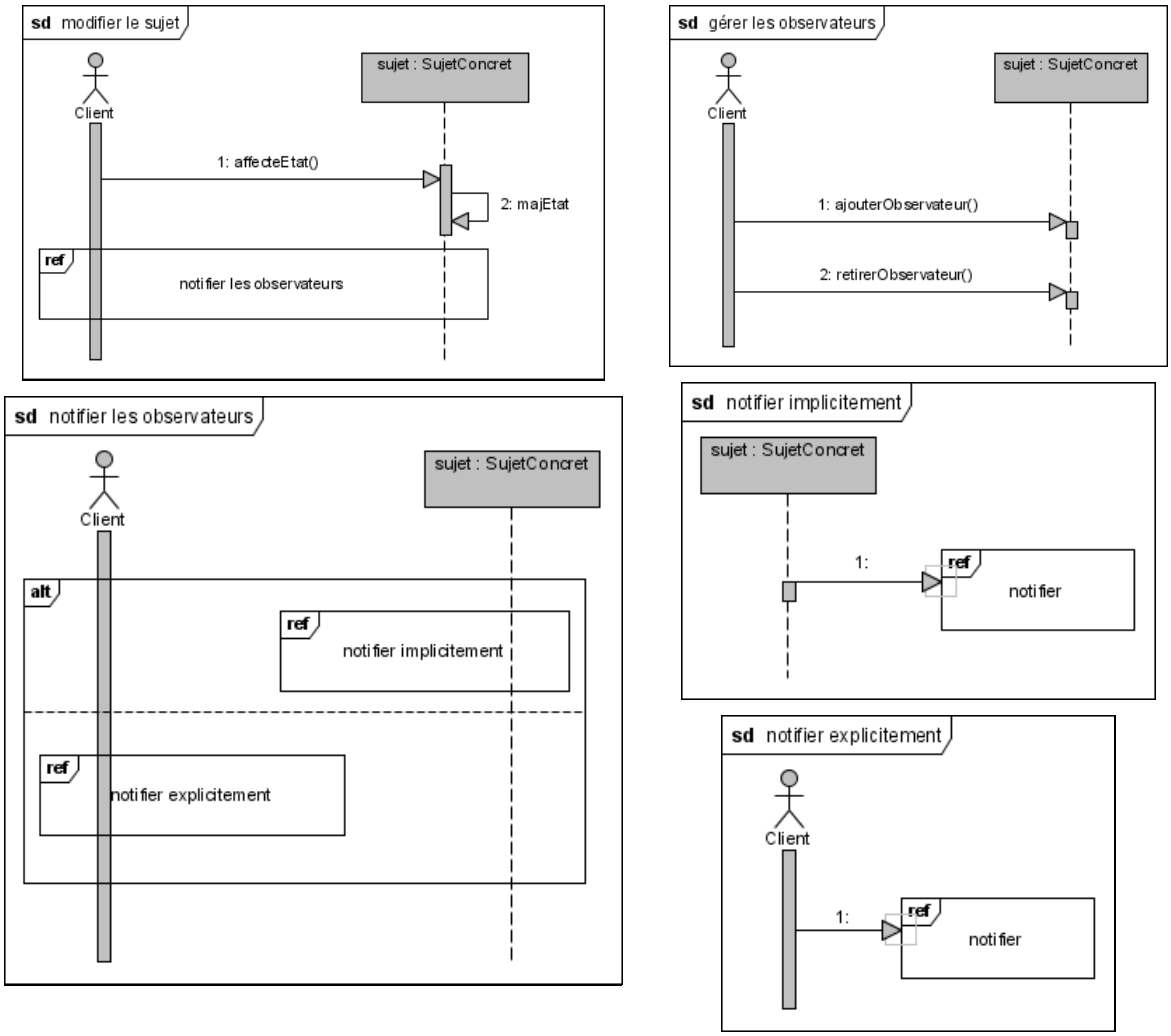


Figure 3.16 : Observateur : vue dynamique à variantes.

2.2.c Variabilité dynamique : une autre représentation avec UML2

De par la multiplication des fragments indépendants, la représentation graphique de la vue fonctionnelle montrée ci-dessus ne permet pas une lisibilité satisfaisante de la variabilité. De plus, les notions de point de variation et de variante n'apparaissent plus au niveau dynamique, ce qui peut poser des problèmes de spécification pour l'ingénieur de patrons, mais également de compréhension pour l'ingénieur d'applications.

C'est pourquoi nous proposons une notation plus concise, dans laquelle les contenus des interactions des points de variation et variantes sont directement reportés dans les interactions de haut niveau, correspondant aux cas d'utilisation obligatoires ou facultatifs. Cela se fait naturellement, grâce à la contrainte de couverture des lignes de vie évoquée plus haut. De plus, pour plus de lisibilité de la variabilité, nous utilisons à nouveau les stéréotypes « variation » et « variant », mais cette fois appliqués sur les interactions correspondantes. La figure 3.17 illustre l'utilisation d'une telle notation pour la représentation de l'interaction du cas d'utilisation *modifier le sujet*.

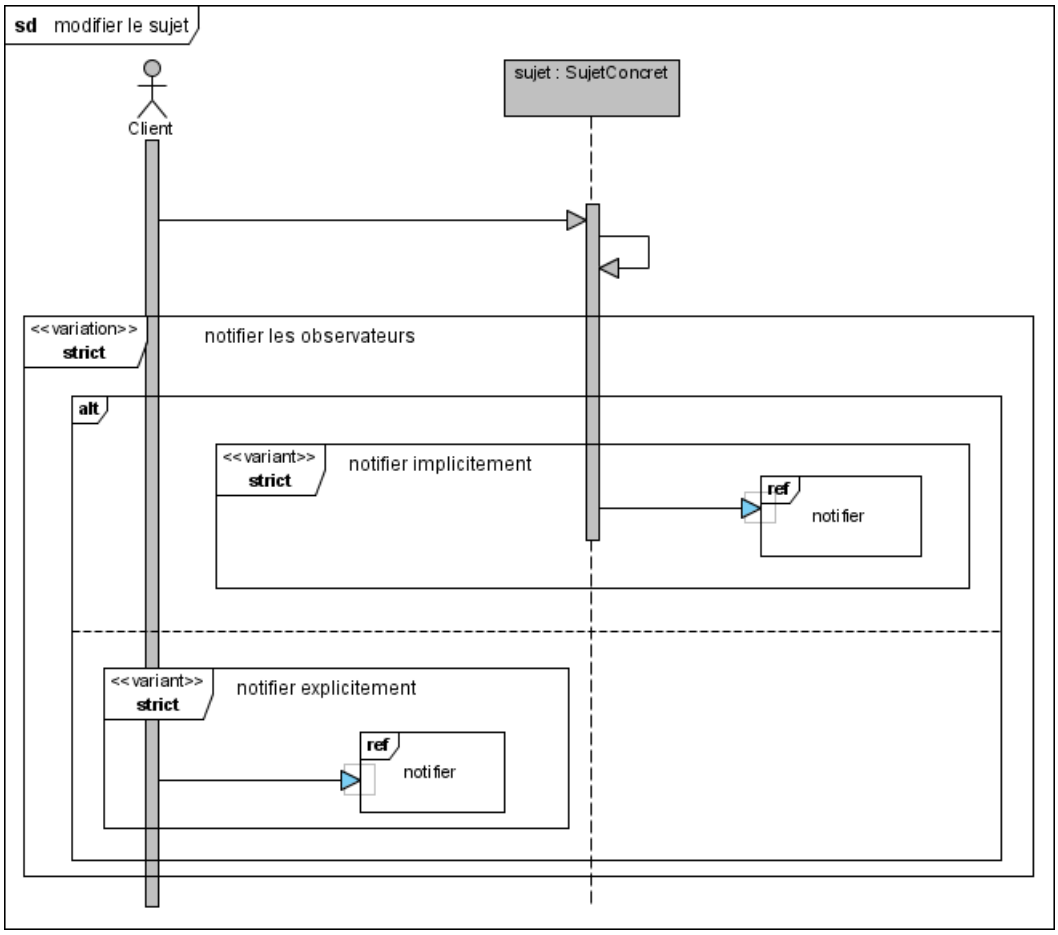


Figure 3.17 : Patron « Observateur » : vue dynamique, notation simplifiée.

2.2.d Variabilité statique

Il ne reste plus qu'à préciser les apports statiques de chaque fonctionnalité pour finaliser notre système. Nous l'avons déjà vu, la spécification statique est en grande partie déductible de la spécification dynamique. Puisque notre vue dynamique est décomposée selon les cas d'utilisation, il en sera de même pour la vue statique : chaque cas d'utilisation donne lieu à un « fragment » statique cohérent avec l'interaction définie auparavant. La vue statique variable sera composée de plusieurs fragments qui s'assembleront lors de l'imitation pour former une vue statique classique. Ces fragments sont des paquetages UML2 stéréotypés « fragment statique » et s'assembleront à l'aide de l'opérateur « merge » d'UML2.

Une grande partie des informations statiques est déductible de la vue dynamique. Par exemple, toutes les classes participant au fragment d'interaction sont présentes dans le fragment statique, et seulement celles-ci. Si un message de l'interaction « parcourt » une association, cette dernière sera présente dans le fragment statique. Il peut être également nécessaire d'introduire une super-classe d'une classe participante si une des propriétés de la super-classe (méthode, rôle d'association, ...) est explicitement utilisée dans l'interaction.

La figure 3.18 représente la vue statique du patron « Observateur ». Dans l'interaction *modifier le sujet*, seule la classe *SujetConcret* est « utilisée », par l'intermédiaire d'une ligne de vie et des méthodes *affectEtat* et *majEtat* (cf. figure 3.16) qu'elle comporte. En suivant les recommandations ci-dessus, le fragment statique de *notifier les observateurs* devrait être vide, puisque l'interaction ne

comporte aucun message propre. Toutefois, nous préconisons la généralisation des propriétés qui sont communes, y compris par inclusion, à toutes les variantes d'un même point de variation. D'un point de vue statique, la seule différence entre les deux variantes de notification est la visibilité de la méthode *notify*. Tous les autres éléments (associations, classe *Observateur,* classe *ObservateurConcret,* ...) sont communs, c'est pourquoi ils sont présents dans le fragment statique du point de variation. Dans cet exemple, le fragment statique *notifier les observateurs* inclut également les apports statiques de l'interaction *notifier* et le rôle *observateur* fait partie de ce fragment.

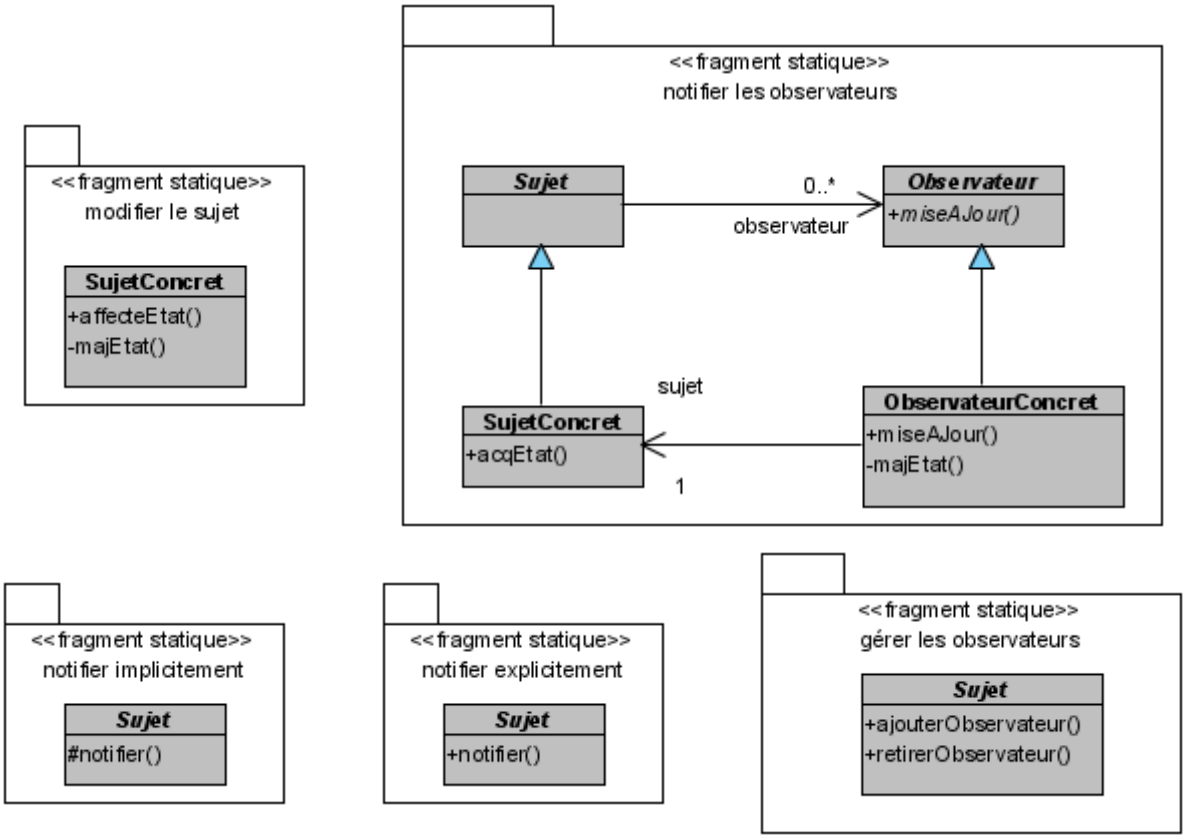


Figure 3.18 : Fragments statiques pour le patron « Observateur »

2.3 Variabilité : extensions d'UML

La figure 3.19 ci-dessous illustre les extensions d'UML dédiées à l'expression de la variabilité au sein des trois vues (les éléments en gris sont les ajouts) :

- Une *Solution* de patron est composée de fonctionnalités (i.e. *CasUtilisationImitable*).
- Un *CasUtilisationImitable* est un *UseCase* classique qui comporte un F*ragmentStatique* et dont le comportement est une seule *Interaction* (*ownedBehavior* spécialisé en *fragmentDynamique*)
- La racine (*Racine*) est un point de variation de type « options » de cardinalité 0..* (cf. 2.1.b).
- Les points de variation et variantes sont respectivement représentés par les méta-entités *VariationPoint* et *Variant*. Les cardinalités d'un point de variation, qui

définissent le type de variabilité, sont représentées comme des attributs de *VariationPoint*.

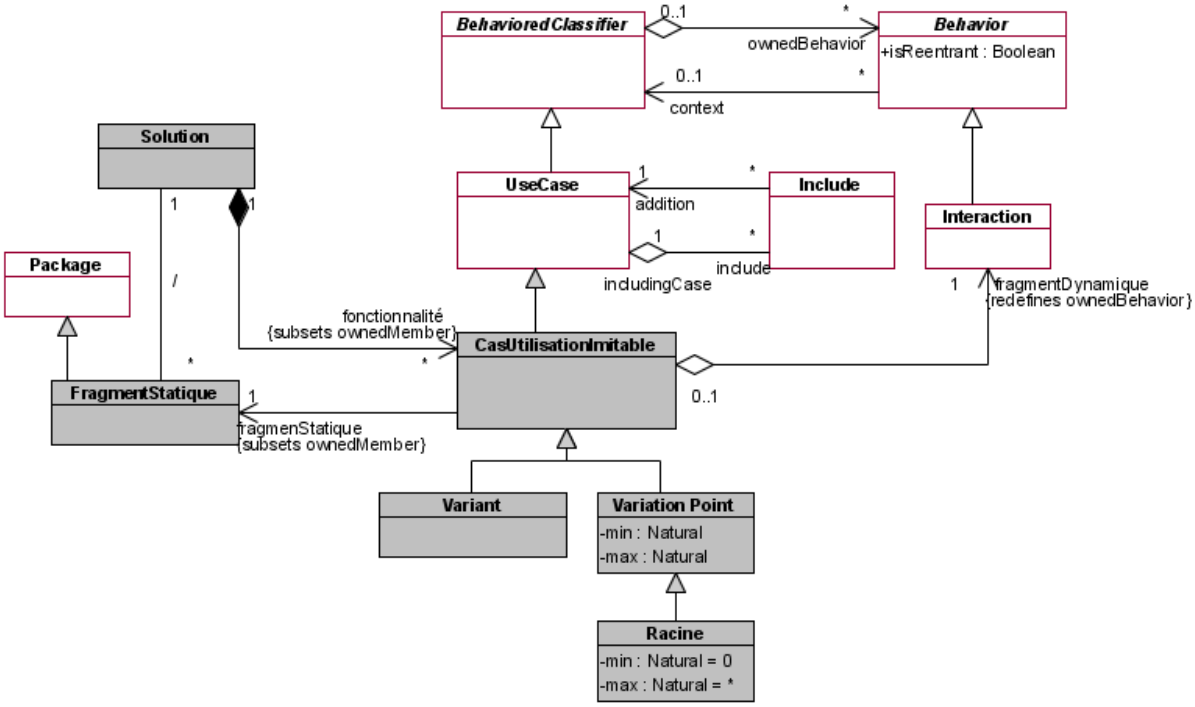


Figure 3.19 : Variabilité : extensions d'UML.

Si les propositions ci-dessus répondent à la problématique d'expression de la variabilité des solutions de patrons, restent à représenter les aspects génériques de telles solutions, indispensables à leur adaptabilité au contexte d'imitation. La section suivante présente nos propositions en terme d'expression des propriétés génériques au sein du mini-système variable.

3 GÉNÉRICITÉ

Nous l'avons vu dans les deux chapitres précédents, la correction d'une imitation est un problème capital lorsque l'on veut construire un système à l'aide de patrons. De manière consensuelle, il est établi que la validité d'une imitation s'appuie sur le respect de propriétés inhérentes à la solution, exprimant le « leitmotiv » (Eden, 2000), ou encore « l'essence » du patron (Pagel et Winter, 1996), (Arnaud et al., 2004).

Une partie de ces propriétés existe déjà au sein de tout patron, à travers les spécifications objet de la solution. Mais, ces spécifications sont destinées à être adaptées au contexte d'imitation et, de fait, l'essence du patron est également constituée des propriétés définissant les limites de cette adaptation. Ces propriétés sont dites génériques et donneront lieu à des contraintes à respecter lors de l'imitation du patron.

3.1 Précision sur le nommage des éléments

La plupart des éléments de modélisation porte un nom. Lors de la spécification, les noms sont choisis avec attention afin d'être clairs et de correspondre le mieux possible au contexte. Contrairement aux noms utilisés dans un patron de couverture *générique*, (cf. chapitre 1, section 2), il est probable que ceux choisis par l'ingénieur de patrons dans un patron de couverture *entreprise* correspondent au mieux aux différents contextes d'imitation. Autrement dit, une imitation de la classe *Sujet* du patron « Observateur » portera probablement un nom plus spécifique que « Sujet », comme c'est le cas dans le cas d'étude, où elle est nommée *Détecteur*. Par conséquent, nous considérons que tous les noms utilisés dans une solution de patron sont adaptables, sans restriction, lors de l'imitation.

3.2 Cadre de définition

Avant de se concentrer sur l'expression des différentes propriétés génériques, nous donnons un cadre général à leur définition ainsi qu'à leur mise en œuvre. L'objectif étant au final de vérifier la validité de l'imitation par rapport à la solution, ce lien de dépendance, qui sera le support effectif de cette validation, doit être précisé.

Pour ce faire, nous nous basons sur deux postulats.

- A l'instar du langage MAC dans (Sunye, 1999), l'expression de propriétés génériques est particulièrement pertinente sur une partie seulement des types d'éléments de modélisation objet. Notre approche considérera uniquement les aspects génériques des classes, associations, attributs et méthodes. Toutefois, puisque notre approche est basée sur l'expression de la variabilité sur les fonctionnalités, il serait possible d'intégrer la généricité au niveau des cas d'utilisation (*CasUtilisationImitable*)
- Chacun des éléments de modélisation cités ci-dessus participant à la spécification de la solution est potentiellement imitable et servira de référence à (aux) l'élément(s) de l'imitation qu'il a engendrés. Autrement dit, il doit être possible, pour tout élément imité (ie. appartement à l'imitation), de retrouver l'élément imitable (ie. appartenant à la solution) dont il est « issu ». Cette traçabilité de bas niveau sera essentielle à la vérification globale de l'imitation, et donc au maintien de la cohérence entre patrons et imitations.

En accord avec l'approche présentée dans (Albin-Amiot et Guéhéneuc, 2001), nous considérons que les propriétés génériques sont communes à tous les patrons et qu'il est donc préférable de définir un référentiel commun à ces propriétés. Notre approche étant jusque-là associée à UML, cela implique de concevoir ce référentiel autour de ce langage, et plus particulièrement sous la forme d'une extension. Cela revient à dire que nous utilisons un méta-modèle commun pour tous les patrons.

Notre approche réalise le lien solution-imitation par délégation en mettant en relation un *ElementImitable*, via le rôle *élémentImité* de l'association Imitation, avec un *Element*, (racine des types définis par UML). La qualification de cette association (qualificateur *imitation*) permet de retrouver uniquement les imitations de l'élément imitable qui appartiennent à une imitation de patron donnée. *ElementImitable* est une classe abstraite et sera spécialisée pour chaque type d'éléments de

modélisation par une méta-entité qui spécialisera également le type de base (classe, association, etc.) qu'elle traite.

Une solution de patron peut être imitée plusieurs fois, mais ses propriétés génériques portent, de manière indépendante, sur chacune de ses imitations. La question de l'encapsulation des imitations, comme des solutions, se pose donc. Nous proposons de définir deux méta-entités *Solution* et *Imitation*. Ces entités ont pour rôle de regrouper les différents participants d'une solution (ses éléments imitable) et d'une imitation (ses éléments imités).

Bien qu'une *Solution* soit un *ElementImitable*, l'association *Imitation* ne sera jamais utilisée pour ce type d'entités. Toutefois, une imitation étant un *NamedElement*, et donc un *Element*, cette dernière possède un et un seul élément imitable : la solution dont elle est issue.(cf. contraintes OCL de la figure 3.20).

La figure 3.20 illustre ce cadre d'application et donne l'exemple de l'extension pour le concept de classe et d'opération imitable. La contrainte OCL appliquée sur *ClasseImitable* (resp. *OpérationImitable*) garantit la cohérence de l'élément imité avec l'élément imitable dont il est issu. La contrainte OCL appliquée sur *Solution* interdit à une solution (qui est un *Package* et donc un *Element*) d'être un élément imité.

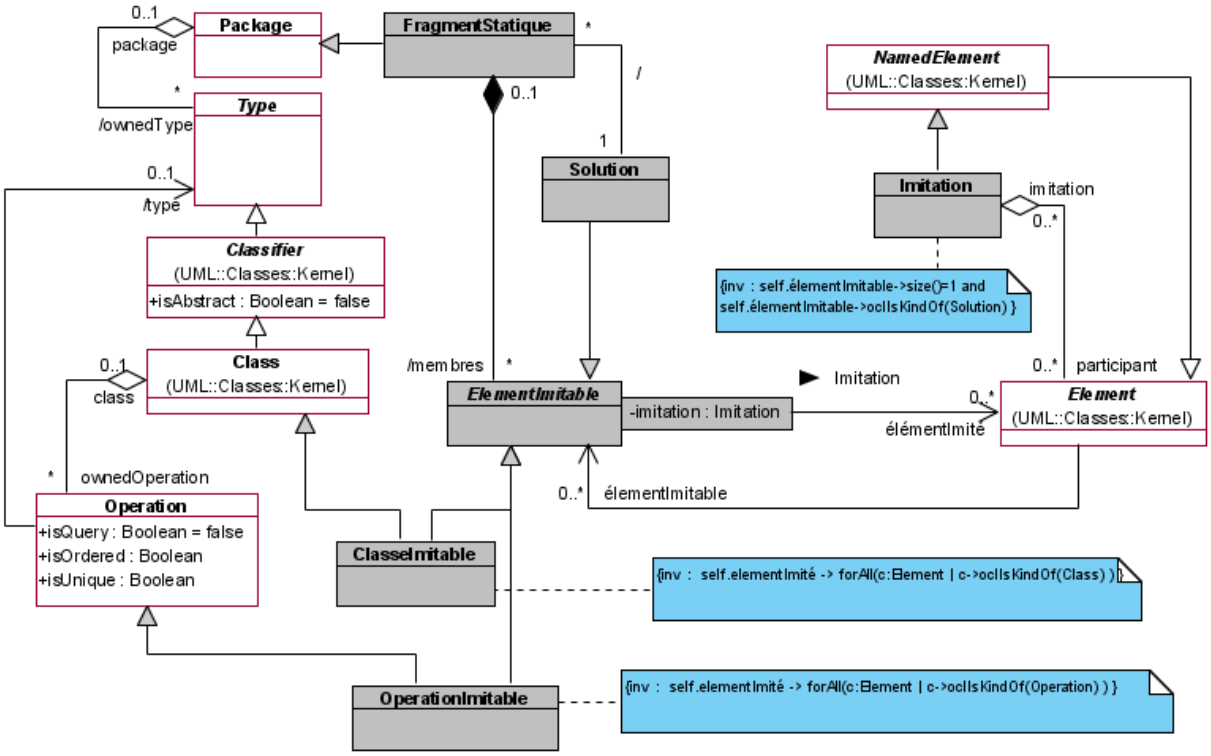


Figure 3.20 : Cadre de définition des propriétés génériques.

3.2.a Relation entre généricité et variabilité

Les propriétés génériques ne s'appliquent que sur des éléments de modélisation statiques (classes, associations, etc.). C'est pourquoi l'intégration de nos propositions en termes de variabilité et de généricité s'opère au sein de la vue statique. En fait, c'est au sein de chaque fragment statique que ces propriétés seront exprimées, et au moment de l'imitation, la fusion des fragments statiques

correspondant aux variantes sélectionnées donnera lieu à une union de ces propriétés. Un élément imitable appartient donc forcément à un fragment statique, faisant lui-même partie de la solution.

3.2.b Avantages du cadre de définition

Le cadre d'application ci-dessus présente un triple avantage. Premièrement, il permet une démarche incrémentale pour la définition des catégories de propriétés génériques, puisqu'elle sera centrée sur le type de base relatif à chaque propriété. Par exemple, pour toutes les propriétés génériques applicables sur les classes et les méthodes, il faudra respectivement agir (ajout de méta-attibuts, d'associations, ...) sur la méta-entité *ClasseImitable* ou *OperationImitable*. Potentiellement, d'autres types d'éléments peuvent aisément être introduits dans notre approche.

Le second avantage est que les élément imités peuvent être considérés comme des entités UML au sens strict du terme et que l'intelligence relative aux patrons reste confinée à l'écart du modèle du système cible, qui y fait seulement référence par le rôle *élémentImitable* de la méta-association *Imitation*.

Le troisième avantage est induit par les cardinalités du rôle *élémentImitable*. En effet, la multiplicité *0..** autorise un élément du système cible à participer à plusieurs imitations. Cela permettra, nous le verrons par la suite, de gérer plus aisément la composition d'imitations.

3.3 Principes et Exemples

3.3.a Propriétés génériques

Dans une imitation du patron « Observateur », les imitations des classes *Sujet* et *Observateur* n'ont aucune raison d'être multiples, et d'ailleurs, cela poserait des problèmes pour les liens de généralisation-spécialisation entretenus avec les autres classes de la solution. Par contre, les autres classes, comme par exemple *ObservateurConcret*, ont vocation, en fonction du contexte, à être imitées plusieurs fois (cf. *AfficheurGraphique* et *AfficheurSimple* du cas d'étude). Nous devons donc permettre à l'ingénieur de patrons de préciser, pour chaque classe et dans le cadre de l'essence de la solution du patron, si elle peut être imitée plus d'une fois.

Dans la plupart des approches abordant le problème du nombre d'occurrences des éléments imités (France et al., 2003), (Sunye, 1999), le nombre nul d'occurrences est traité comme les autres. Dans notre approche, un élément de modélisation tel qu'une classe, un attribut ou une méthode, s'il est présent dans la vue statique, a forcément un rôle à jouer dans la ou les fonctionnalités dont il dépend, et doit donc être imité au moins une fois. La « disparition » d'un élément à l'imitation ne peut être que consécutive à la non sélection de la variante dont cet élément dépend.

Comme l'ont montré David Mapelsden et ses collègues dans (Maplesden et al., 2001), le nombre d'occurrences d'imitation d'un élément peut être en relation avec le nombre d'occurrences des imitations d'autres éléments (cf. patron « Fabrique Abstraite »). Les auteurs résolvent ce problème à l'aide de la notion de dimension présentée chapitre 2, section 4.5.

Notre approche traite ces deux aspects de généricité autour du nombre d'occurrences d'imitation d'un élément. L'ingénieur de patrons se verra proposer deux propriétés à renseigner, décrites ci-dessous. Pour un même élément imitable, ces propriétés s'excluent mutuellement.

Nombre d'occurrences d'imitation.
Applicable sur des éléments de type *classe*, *association, méthode* et *attribut*, cette propriété définit le nombre d'imitations autorisées de l'élément, sous la forme d'une cardinalité minimale et maximale, similaire à une multiplicité classique. Toutefois, cette multiplicité est à moduler en fonction des relations conteneur/contenu que certains types d'éléments entretiennent. Par exemple, la méthode *update* de la classe *ObservateurConcret* doit être imitée une fois par occurrence d'imitation d'*ObservateurConcret*.

Dimension
Cette propriété est applicable sur des éléments de type *classe*, *association, méthode* et *attribut*. Elle définit une relation complexe entre les nombres d'occurrences d'imitation de plusieurs éléments. Les dimensions peuvent être associées entre elles pour former des produits cartésiens (cf. « Fabrique Abstraite »).

3.3.b Mise en œuvre dans le cadre de définition

Les deux propriétés ci-dessus peuvent s'appliquer sur tous les types d'éléments imitables considérés dans le deuxième postulat du cadre de définition. Ainsi, nous généralisons le support de ces deux propriétés au niveau de l'*ElementImitable*. La figure 3.21 présente l'extension de cette méta-entité par le biais de trois rôles d'association :

- Le rôle *nbOccurAutorisée* référence une entité de type « multiplicité » afin d'exprimer la propriété « Nombre d'occurrences d'imitation ». S'il n'y a pas de contrainte à définir, la valeur par défaut est *0..**. Ce rôle est vide uniquement si l'élément est associé à une ou plusieurs dimensions.
- Le rôle *définit* permet de définir une dimension à partir d'un élément.
- Le rôle *associéA* référence un ensemble de *Dimension* dont l'élément dépend, et que ses imitations doivent respecter.

Un élément imitable ne peut pas définir une dimension alors qu'il est associé à une autre. La contrainte OCL appliquée à *ElementImitable* le garantit.

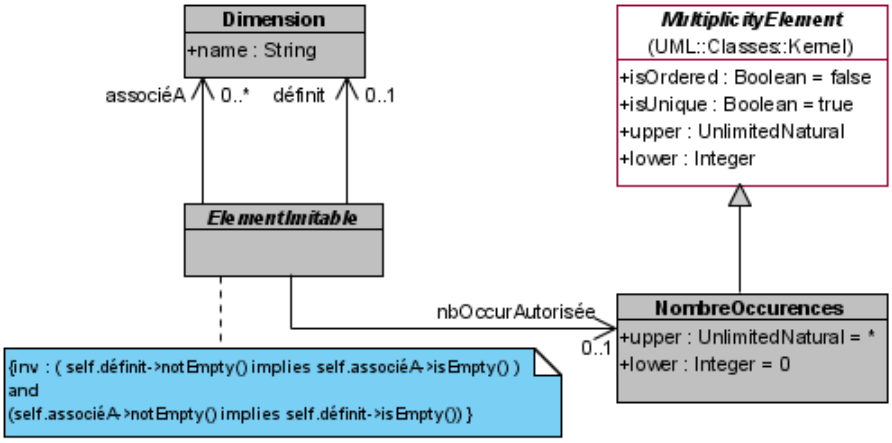


Figure 3.21 : Extension de la méta-entité ElementImitable pour prendre en compte le nombre d'occurrences et les dimensions.

Nous donnons ici une partie de l'essence du patron « Fabrique abstraite ».

- Il ne peut y avoir qu'une seule imitation de *FabriqueAbstraite*.
- Les fabriques doivent comporter autant d'opérations de création de produit qu'il y a de familles de produits et donc d'imitations de *ProduitAbstrait*.
- Il doit également y avoir autant d'imitations de *ProduitConcret* que de paires d'imitations *ProduitAbstrait*/*FabriqueConcrete*.

La figure 3.22 montre l'instanciation de propriétés génériques pour spécifier l'essence du patron « Fabrique abstraite ». Le nombre d'occurrences pour la classe *FabriqueAbstraite* est *1..1* tandis que pour *FabriqueConcrete* et *ProduitAbstrait* il n'y a pas de limitation (*0..**). Ces deux classes définissent également deux dimensions auxquelles *ProduitConcret* est associée. L'opération *créer* est associée à la dimension de *ProduitAbstrait*.

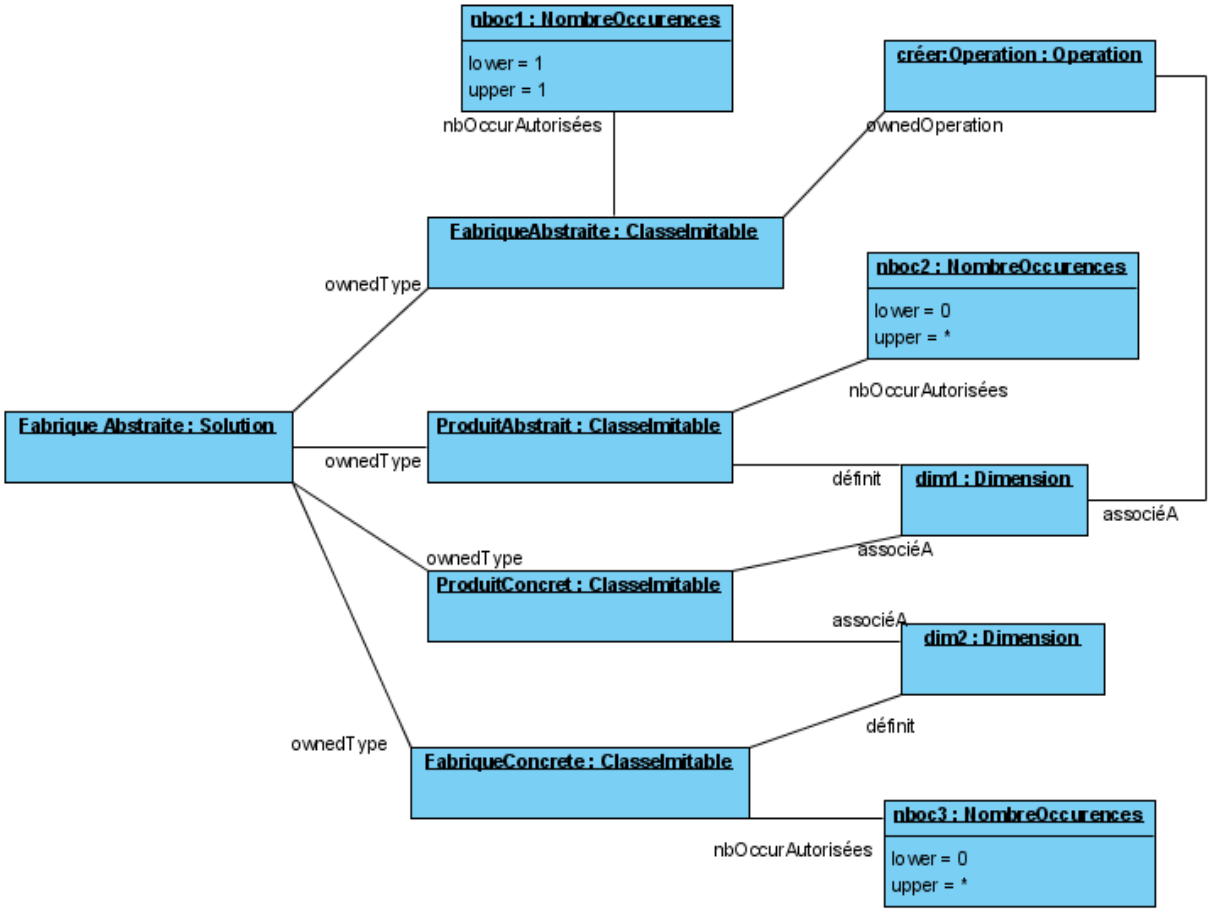


Figure 3.22 : Patron « Fabrique abstraite » : dimension et nombre d'occurrences.

3.3.c Contraintes lors de l'imitation

Toute utilisation de propriété générique sur un élément imitable implique des contraintes qui garantissent la validité de ses imitations. Ces contraintes portent sur les éléments de modélisation des imitations et non sur les instances de ces derniers puisque solution et imitation se trouvent au même niveau de modélisation (cf. chapitre 2, section 2). Par exemple, il s'agit de vérifier que, dans le modèle d'une imitation du patron « Observateur », il n'y a qu'une seule classe occurrence d'imitation de *Sujet* et non pas de s'assurer qu'il n'y aura qu'une seule instance de cette occurrence lors de l'exécution.

Une contrainte d'imitation n'a de sens qu'au sein d'une seule imitation de patron. Cela signifie qu'elle doit être vérifiée pour chaque imitation de patron. Pour exprimer ces contraintes, nous tirons partie du rôle *élémentImité* qui permet, pour tout élément imitable, d'atteindre ses éléments imités au sein d'une imitation donnée (grâce au qualificateur).

La contrainte OCL ci-dessous permet, à partir d'une classe imitable, de vérifier le nombre d'occurrences de ses imitations au sein d'une imitation donnée (*IMIT* dans la contrainte).

```
context ClasseImitable
inv :
  let
  min = self.nbOccurAutorisée.lower
  max = self.nbOccurAutorisée.upper
  elems = self.élémentImité[IMIT]
  in
elems->size() >= min and elems->size() <= max
```

Si l'expression de la contrainte de nombre d'occurrences sur les classes est aisée, il en est différemment pour les éléments de modélisation qui composent les classes. Par exemple, le nombre d'imitations d'une méthode n'a un sens qu'au sein de la classe qui la comporte. En effet, il doit y avoir une et une seule imitation de la méthode *miseAJour* pour chaque imitation de la classe *ObservateurConcret*, et donc, une contrainte similaire à celle donnée pour les classes imitables ne serait pas correcte. La contrainte OCL ci-dessous permet, au sein d'une imitation donnée (*IMIT* dans la contrainte), de vérifier que le nombre d'imitations de la méthode est correct dans chaque classe.

```
context OperationImitable
inv :
  let
  minocc = self.nbOccurAutorisée.lower
  maxocc = self.nbOccurAutorisée.upper
  classeImitable = self.class
  classeImitées = classeImitable.élémentImité[IMIT]
   in
  classeImitées->forAll(c|c.ownedOperation->select(m|
                          m.élémentImitable=self)->count()>= minocc)
and
  classeImitées->forAll(c|c.ownedOperation->select(m|
                          m.élémentImitable=self)->count()<= maxocc)
```

Lors de la définition d'une propriété générique s'appliquant sur plusieurs types d'éléments, il faut s'attendre à définir plusieurs contraintes, en fonction des types de base des spécialisations d'éléments imitables.

3.4 Abstraction et Visibilité

Notre approche permettant aisément l'ajout de nouveaux types de propriétés génériques, nous ne pouvons donc donner une liste exhaustive de propriétés. Néanmoins, dans cette section, nous en citons deux autres, accompagnées des contraintes qu'elles engendrent.

3.4.a Abstraction

De nombreux patrons mettent en œuvre des mécanismes d'abstraction au sein de leur solution. Cela permet notamment d'en séparer certaines parties fixes d'autres plus « adaptables » au contexte. C'est le cas dans le patron « Observateur », où l'architecture du protocole de mise à jour est défini dans les classes abstraites *Sujet* et *Observateur*. Le caractère abstrait d'une classe, défini dans UML par un booléen de la méta-entité *Classifier*, peut être une propriété forte de la solution, et mérite donc qu'on lui consacre une propriété générique. Cette propriété ne doit pas définir si la classe est abstraite (puisque cela est déjà fait par UML), mais si sa qualité d'abstraction peut être modifiée par l'ingénieur d'applications. Nous considérons, par exemple, que les classes *Sujet* et *Observateur* du patron « Observateur » doivent rester abstraites.

Nous exprimons cette propriété générique au sein de la méta-entité *ClasseImitable*, par l'ajout d'un méta-attribut booléen nommé *gardeAbstraction*.. Par défaut sa valeur est « faux ». La figure 3.23 illustre cet ajout.

La contrainte OCL associée peut être exprimée comme suit :

```
context ClasseImitable
inv :
  let
  classeImitées = self.élémentImité[IMIT]
  in
  classeImitées->forAll(self.gardeAbstraction implies
                          isAbstract=self.isAbstract)
```

3.4.b Visibilité

Dans une spécification orientée-objet, la notion de visibilité d'un attribut ou d'une méthode est une information importante. Nous l'avons vu, dans le patron « Observateur », la visibilité de la méthode *notifier* dépend du type de notification (implicite ou explicite) que l'ingénieur d'applications choisit. De manière analogue au problème de l'abstraction évoqué ci-dessus, nous proposons une propriété générique permettant à l'ingénieur de patrons de préciser si la visibilité qu'il a donnée à une méthode peut être modifiée dans une imitation.

Dans le cas d'une notification explicite, la méthode *notifier* est définie comme publique mais doit absolument le rester, au risque d'empêcher tout appel extérieur de cette méthode et de rendre la vue dynamique de la notification explicite incohérente. Pour la notification implicite, la visibilité originelle est « protégée » et ne peut être réduite à « privée » car elle rendrait elle aussi la vue dynamique incohérente, et donc le patron inconsistant.

Il faut aussi s'interroger sur l'ouverture d'une visibilité. Le rôle *observateur* de la classe *Sujet* est défini comme privé, ce qui valide l'idée selon laquelle le protocole de mise à jour est spécifique aux classes abstraites *Sujet* et *Observateur*. L'ouverture de cette visibilité permettrait donc aux imitations de la classe *SujetConcret* de manipuler directement leurs observateurs.

Même si le problème de cohérence inter vues est normalement traité par le langage de modélisation (UML dans notre cas), nous considérons que la précision, par l'ingénieur de patrons, d'une limitation dans l'ouverture ou la fermeture d'une visibilité ajoute du sens à sa solution. Nous proposons donc deux propriétés génériques traitant de l'ouverture et de la fermeture.

Dans UML, la majorité des éléments ont potentiellement une visibilité (attribut *visibility* de la méta-entité abstraite *NamedElement*). Il existe quatre types de visibilité : publique, paquetage, protégée et privée. Pour pouvoir les comparer, nous utilisons une relation d'ordre telle que : publique > paquetage > protégée > privée. Nous ajoutons à la méta-entité *ElementImitable,* un méta-attribut *fermetureVis* (respectivement *ouvertureVis*) de type booléen, qui, s'il vaut «vrai », indique qu'une fermeture (resp. une ouverture) sera possible, dans la limite, toutefois, du maintien de la cohérence inter-vues. Ces ajouts sont illustrés figure 3.23.

En utilisant la relation d'ordre donnée plus haut, la contrainte OCL associée à la propriété générique de fermeture est la suivante :

```
context ElementImitable
inv :
  let
  élémentImitées = self.élémentImité[IMIT]
```

```
in
élémentImitées->forAll(self.fermetureVis implies
                        visibility<=self.visibility)
```

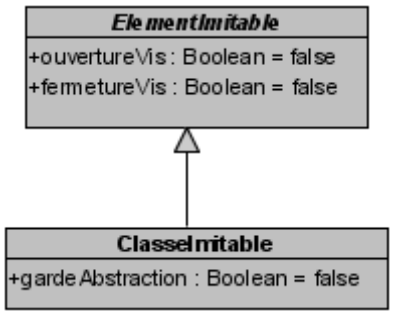


Figure 3.23 : Nouvelles propriétés génériques : abstraction et visibilité.

3.5 Une autre forme de généricité : les notes « A Faire »

Les propriétés génériques ci-dessus ont un caractère assez formel, qui permet d'ailleurs d'exprimer des contraintes qui garantissent leur application. Dans beaucoup de solutions de patrons il existe des propriétés auxquelles ni la variabilité, ni la généricité telles que nous les avons vues ne peuvent apporter de solutions, en raison de leur aspect très générique et surtout extrêmement dépendant du contexte d'imitation.

Par exemple, dans la solution originelle du patron « Observateur », les auteurs représentent l'état du sujet concret (tout comme celui de l'observateur concret) par un attribut de classe. Nous avons d'ailleurs déjà fait évoluer cette solution dans la section 1.2, afin d'encapsuler strictement la modification de cet état dans une méthode privée (*majEtat*). Néanmoins, rien n'a encore été dit sur l'adaptation de l'état au contexte d'imitation et il est raisonnable de penser que l'état du sujet ne puisse s'exprimer à travers un simple attribut. Les auteurs mettent d'ailleurs ce point en avant dans la description textuelle du patron. L'état d'un sujet peut être une agrégation de nombreuses valeurs telles celles de plusieurs attributs et/ou d'associations. Ainsi, si le sujet a bien besoin d'un état, il n'est pas possible a priori, pour l'ingénieur de patrons, de prévoir (et donc de contraindre) la forme que cet état prendra lors de chaque imitation. Par conséquent, il ne peut pas non plus prévoir le contenu de la méthode *majEtat*. La seule chose que l'ingénieur de patrons puisse faire, c'est d'attirer l'attention de l'ingénieur d'applications sur le fait qu'il devra constituer un état, et réaliser la méthode de modification.

Nous proposons d'exprimer ce type de propriétés sous la forme de notes « A Faire » que l'ingénieur de patrons pourra apposer sur n'importe quel élément imitable et dont le contenu est un texte libre. Ces notes ont pour vocation d'être dupliquées pour chaque élément imité, puis validées par suppression lorsque l'ingénieur d'applications estimera qu'il les aura prises en compte.

Les notes « A Faire » sont présentées dans le fragment statique qui, d'après l'ingénieur de patrons, leur correspond le mieux. Il faut néanmoins que l'élément imitable soit présent dans ce fragment. La figure 3.24 illustre l'utilisation des notes « A Faire » sur les fragments statiques *modifier le sujet* et *notifier les observateurs* du patron « Observateur ». En effet, le problème de l'état et de sa méthode de mise à jour se retrouve sur *SujetConcret* et *ObservateurConcret*.

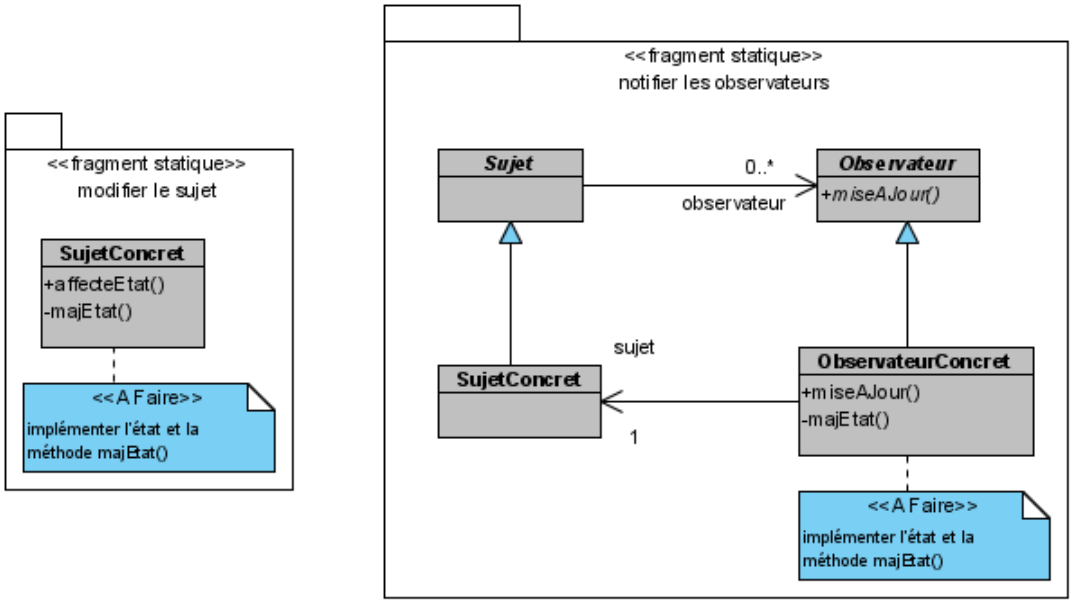


Figure 3.24 : Utilisation des notes "A Faire" dans le patron « Observateur ».

4 Processus de spécification

Dans le chapitre deux, nous avons reproché à certaines approches de ne pas donner aux ingénieurs de patrons une description précise de la manière de spécifier la solution d'un patron. Nous proposons ici un guide méthodologique permettant de construire une solution de patron conforme à nos propositions en terme de complétude, variabilité et généricité. La figure 3.25 ci-dessous illustre ce processus de spécification. Les activités sont détaillées ci-dessous.

- *Identifier les fonctionnalités obligatoires et facultatives.* Il s'agit d'identifier les fonctionnalités de premier niveau, c'est-à-dire celles directement accessibles par le client. Cela donne une première ébauche de la vue fonctionnelle où les fonctionnalités sont stéréotypées « obligatoire » ou « facultatif ».
- Pour chaque cas d'utilisation, *détailler la fonctionnalité.*
  - *Détailler la séquence.* Il s'agit de donner le fragment d'interaction associé à la fonctionnalité. Il est probable que des inclusions de fonctionnalités (i.e. références d'interactions) soient nécessaires. Dans ce cas, la fonctionnalité incluse doit également être détaillée.
  - Il faut également préciser le fragment statique comportant les éléments utilisés dans le fragment dynamique. L'ingénieur de patrons peut, si besoin, *identifier les propriétés génériques* et *ajouter les notes « A Faire »* sur ce fragment statique.

Une fois toutes les fonctionnalités détaillées, l'ingénieur de patrons doit identifier les points de variation à partir de la vue fonctionnelle. Pour chaque point de variation identifié, l'ingénieur doit :

- *Déclarer le point de variation.* Il s'agit de marquer ce cas d'utilisation comme étant un point de variation et de déterminer le type de variabilité.

- *Ajouter une variante* dans la vue fonctionnelle. Chacune de ces variantes doit être détaillée de la même manière que précédemment (*détailler la fonctionnalité*). Cette activité est répétée tant que toutes les variantes ne sont pas spécifiées.
- Une fois toutes les variantes spécifiées, l'ingénieur de patrons peut *factoriser* les éléments statiques communs aux fragments statiques de toutes les variantes au sein du fragment statique du point de variation.

Le processus se termine lorsqu'il n'est plus possible à l'ingénieur de patrons d'identifier de nouveaux points de variations.

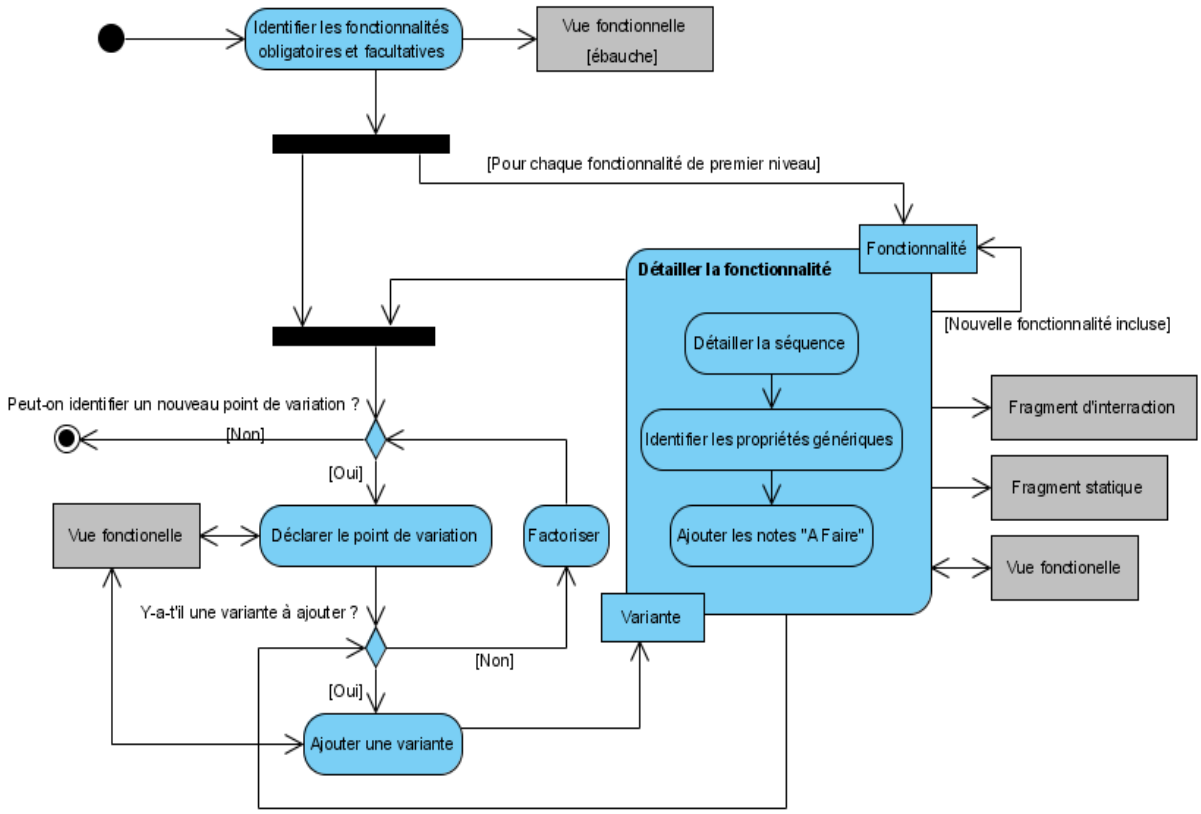


Figure 3.25 : Processus de spécification de la solution d'un patron.

5 Application : Patron Composite

Dans cette section, nous appliquons le processus de spécification défini ci-dessus à la spécification de la solution du patron « Composite » du GoF (cf. Annexes). Nous rappelons que l'intention de ce patron est de « *composer des objets dans des structures d'arbres pour représenter des hiérarchies tout-partie* ». La figure 3.26 ci-dessous illustre le diagramme de classes UML de la solution initialement proposée (Gamma et al., 1995).

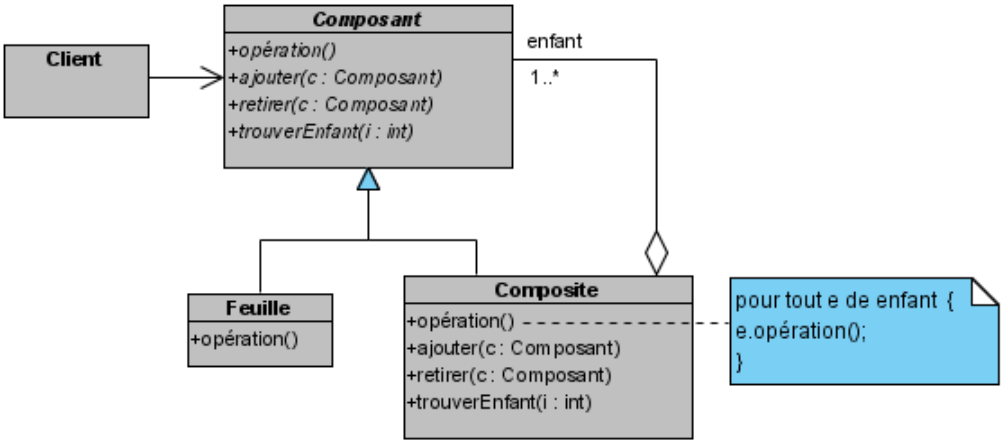


Figure 3.26: Diagramme de classes original du patron « Composite ».

La première étape du processus de spécification consiste à identifier les fonctionnalités obligatoires et facultatives du patron. Quatre fonctionnalités de premier niveau se dégagent : *Effectuer une opération récursive*, *Ajouter un composant, Retirer un composant*, *Trouver un enfant*. Elles sont facilement identifiables sur le diagramme de classes original car bien visibles dans l'interface de la classe *Composant*.

Trouver un enfant est une fonctionnalité de second ordre et donc facultative. La construction de la structure arborescente (*Ajouter* et *Retirer un composant*) ainsi que son exploitation (*Effectuer une opération récursive*) sont quant à elles des fonctionnalités obligatoires fortement liées à l'intention du patron. La figure 3.27 présente l'ébauche de vue fonctionnelle issue de ce constat, dans la notation simplifiée.

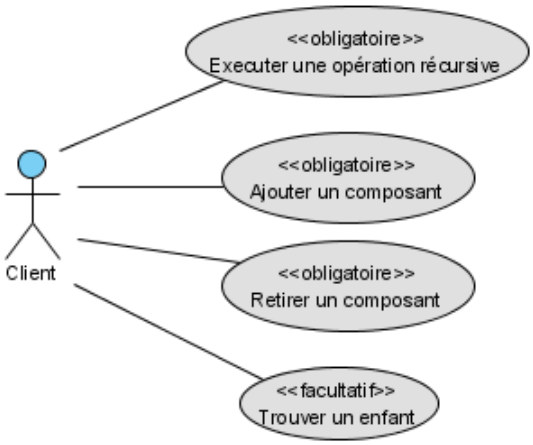


Figure 3.27 : Patron « Composite » : ébauche de vue fonctionnelle.

L'étape suivante consiste à détailler chaque fonctionnalité identifiée à l'aide d'un fragment dynamique (interaction) et d'un fragment statique. L'étude de *Exécuter une opération récursive* nous amène à considérer une nouvelle fonctionnalité incluse : *opération*.

Les figures 3.28 et 3.29 présentent respectivement une partie de la nouvelle vue fonctionnelle et les fragments dynamiques et statiques des fonctionnalités *Exécuter une opération récursive* et *opération*. Nous rappelons que le fragment statique comporte exclusivement des éléments utilisés dans le

fragment dynamique, et peut donc être construit simultanément. A partir de ce fragment statique, l'ingénieur de patrons peut par exemple préciser qu'il ne doit y avoir qu'une seule imitation de *Composant* et que cette classe doit rester abstraite (cf. section 3. Dans la figure 3.29, ces propriétés génériques sont signifiées à l'aide de notes).

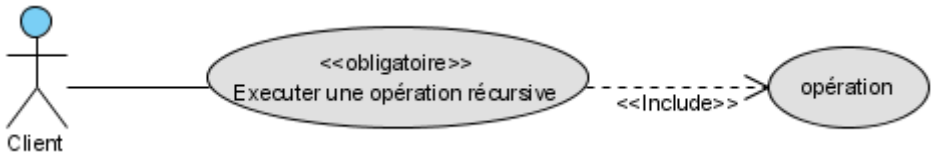


Figure 3.28 : Patron « Composite » : augmentation de la vue fonctionnelle.

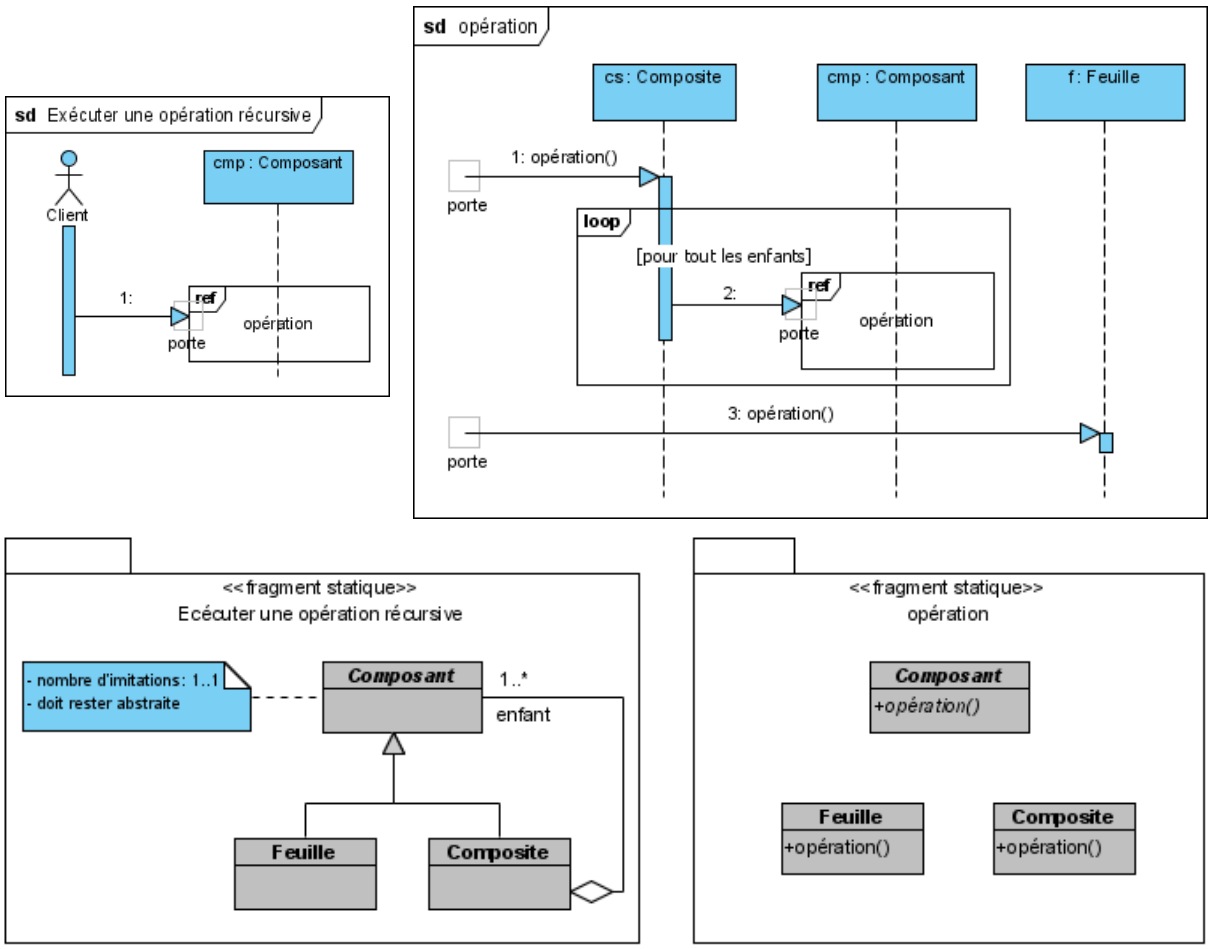


Figure 3.29 : Patron « Composite » : quelques fragments dynamiques et statiques.

Dans la description originale de ce patron, beaucoup de variantes d'implémentation sont données ainsi que quelques variantes plus fonctionnelles. En particulier une réflexion est entamée sur l'interface de *Composant* et sur la déclaration des opérations de gestion des enfants. La question principale est la suivante : Est-ce une bonne chose que de proposer une interface de gestion des enfants pour les feuilles, qui n'ont jamais d'enfant ?.

Les auteurs décrivent deux philosophies : celle dite de transparence, et celle dite de sécurité.

- Dans la philosophie de transparence, *Composant* définit un comportement nul par défaut des opérations de gestion des enfants. Cela permet au client d'utiliser uniformément feuilles et composites. Cette solution autorise donc au client d'exécuter des opérations qui n'ont pas de sens (i.e. ajouter un enfant à une feuille), avec l'avantage que de telles requêtes ne déstabilisent pas le système.
- La philosophie de sécurité implique la déclaration des opérations de gestion dans le composite. Les composants ne peuvent donc plus être traités uniformément. Dans ce cas, les auteurs conseillent d'utiliser un mécanisme de détection de type pour être sûr de pouvoir utiliser une opération de gestion des enfants. Même si la plupart des langages orientés-objet ont des opérateurs d'élicitation de type dynamique (*instanceof* en Java), les auteurs proposent de définir dans *Composant* une opération retournant un booléen nommée *estComposite* retournant la valeur « faux ». Cette opération est spécialisée dans *Composite* pour retourner la valeur « vrai ».

Ces deux philosophies apparaissent clairement comme des alternatives à la fonctionnalité *Ajouter un composant*, qui est donc un point de variation. Les figures 3.30 et 3.31 montrent respectivement l'évolution de la vue fonctionnelle et les fragments dynamiques et statiques des fonctionnalités impactées. Les fragments statiques ont été factorisés, ainsi l'opération *ajouter* de *Composite* fait partie du point de variation. Nous n'avons pas dégagé d'autres propriétés génériques à appliquer.

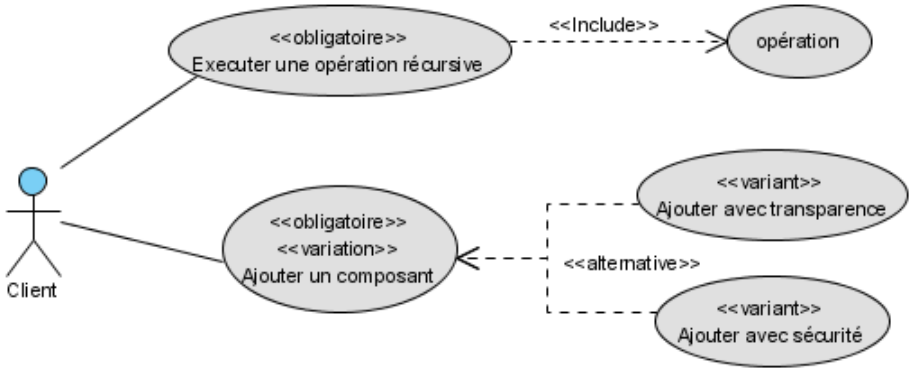


Figure 3.30 : Patron « Composite » : alternatives « transparence » et « sécurité ».

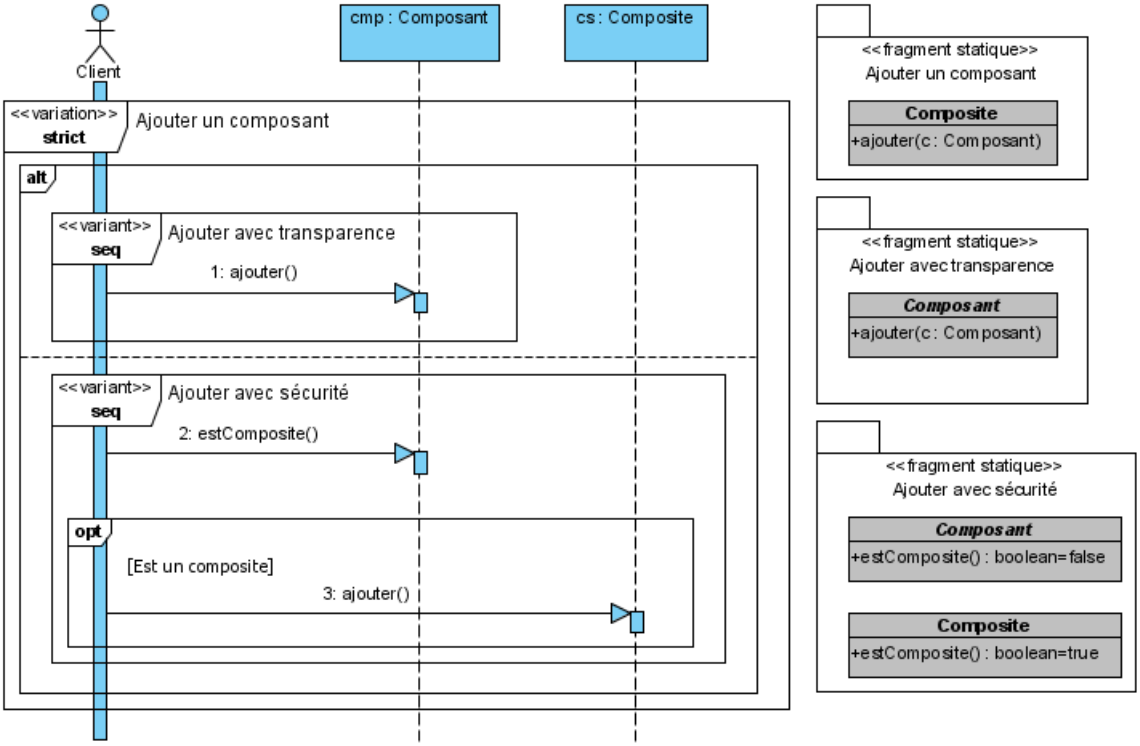


Figure 3.31 : Patron « Composite » : Ajouter un composant, vue dynamique et statique.

La même attention pouvant être apportée à la suppression des enfants, la méthode *estComposite* serait donc présente dans deux fragments statiques « indépendants » et utilisée dans plusieurs variantes de points de variations différents. Ici, ces deux points de variation sont issus d'une réflexion générale qui englobe la fonctionnalité d'ajout et de suppression (gestion des enfants). Notre approche ne comporte pas de mécanisme pour gérer spécifiquement ce type de cas, où l'on peut distinguer une dépendance entre certaines variantes.

6 Synthèse

Dans ce chapitre, nous avons présenté nos propositions en terme de spécification des solutions de patrons. Ces dernières comportent plusieurs vues et sont organisées autour de l'expression de la variabilité fonctionnelle. Elles expriment également des propriétés génériques qui donnent lieu à des contraintes que toute imitation devra respecter.

Ces contraintes d'imitation, telles que nous les définissons, ne s'appliquent qu'à une partie du système cible, puisqu'elles n'ont un sens que pour les participants d'une imitation de patron. En effet, si un même patron est imité plusieurs fois au sein d'un système, il en découlera deux jeux distincts de contraintes, spécifiques à chacune des deux imitations. Cela justifie d'autant l'encapsulation des imitations de patrons dans un conteneur (objet *Imitation*). C'est notre approche par délégation qui est à l'origine du problème, mais, au final, la traçabilité des imitations en est facilitée. Cette encapsulation permet de traiter les imitations indépendamment les unes des autres sans toutefois apporter de solution au problème de l'incompatibilité des imitations entre elles.

Nous gageons qu'il sera possible, au sein d'une mise en œuvre technique, de détecter que deux contraintes issues de deux imitations de patrons sont incompatibles.

Notre objectif principal est d'aider les ingénieurs d'applications lors de l'imitation des patrons. Même s'ils ont maintenant à leur disposition des spécifications de solutions complètes, variables et génériques, il nous faut désormais proposer un processus d'imitation capable de tirer partie de toutes ces informations et définir les grandes lignes des outils qui pourront le supporter. Ceci fait l'objet du chapitre suivant.

CHAPITRE 4 PROCESSUS D'IMITATION DE PATRONS

Le chapitre précédent a décrit nos propositions permettant aux ingénieurs de patrons de spécifier les solutions de leurs patrons de manière complète, variable et générique. De telles spécifications n'ont d'intérêt que si les ingénieurs d'applications peuvent tirer partie de toutes ces informations lors de l'imitation. C'est dans ce but que nous proposons, dans ce chapitre, un processus d'imitation destiné à assister les ingénieurs d'applications lors de la réutilisation des patrons.

La première partie de ce chapitre présente le processus et détaille les deux sous-processus qui le compose en illustrant les différentes activités sur une imitation du patron « Observateur ».

La deuxième section traite de la mise en œuvre technique de ces deux sous-processus en utilisant les principes ainsi que certains outils du domaine de l'ingénierie dirigée par les modèles. Les deux outils sont illustrés à l'aide des patrons utilisés dans le cas d'étude.

1 Le processus d'imitation

1.1 Terminologie et définitions

La figure 4.1 présente le processus d'imitation sous la forme d'un diagramme d'activités. Toutes les activités de ce processus sont réalisées par l'ingénieur d'applications désireux d'imiter un patron, du choix de ce dernier jusqu'à la composition de l'imitation dans le système cible. Ce processus est décomposé en deux sous-processus de *réduction* et *d'application*, respectivement détaillés dans les sections 1.2 et 1.3.

Si un ingénieur d'applications veut réutiliser un patron, c'est qu'il estime que ce dernier résout le problème qu'il rencontre. Avant de réaliser l'imitation à proprement parler, l'ingénieur d'applications effectue donc une recherche du patron qui correspond le mieux à son problème. Pour les patrons du GoF, c'est notamment en s'appuyant sur les rubriques « Intention » et « Motivation » que le concepteur va arrêter son choix. Cette phase décisionnelle, représentée dans la figure 4.1 par l'activité c*hoisir un patron*, n'est pas détaillée dans notre processus.

Le choix du patron permet d'identifier la solution à réutiliser et, dans notre approche, cette solution est complète, variable et générique. Dans le cadre du processus d'imitation, la solution à imiter est représentée par un modèle conforme au méta-modèle présenté chapitre 3 et nommée *Modèle imitable*. C'est ce dernier qui, en fonction de son contenu, va caractériser l'exécution du processus.

Ce sont tout d'abord les aspects variables du modèle imitable qui sont exploités par le sous-processus de *réduction*. L'objectif de ce sous-processus est de permettre à l'ingénieur d'applications de préciser ses besoins (par rapport au patron) en limitant (i.e. réduisant) la solution aux variantes qu'il désire. Le concepteur obtient, en sortie de ce sous-processus, un *modèle imité* qui, s'il n'est plus porteur d'information sur la variabilité, comporte toujours trois vues ainsi que des informations traitant des propriétés génériques. Ce modèle se trouve dans l'état *applicable* car il doit maintenant être appliqué au contexte d'imitation. C'est en fait une première version de l'imitation, qui sera ensuite modifiée par le sous-processus *d'application*.

Le sous-processus *d'application* réalise la mise en œuvre de l'imitation au sein de son contexte : le *système cible*. Ce processus gère l'adaptation du *modèle imité applicable* et sa composition dans le système cible, tout en garantissant le respect des propriétés génériques. Le *système cible* est impacté, tout comme le *modèle imité* qui passe dans l'état *appliqué*.

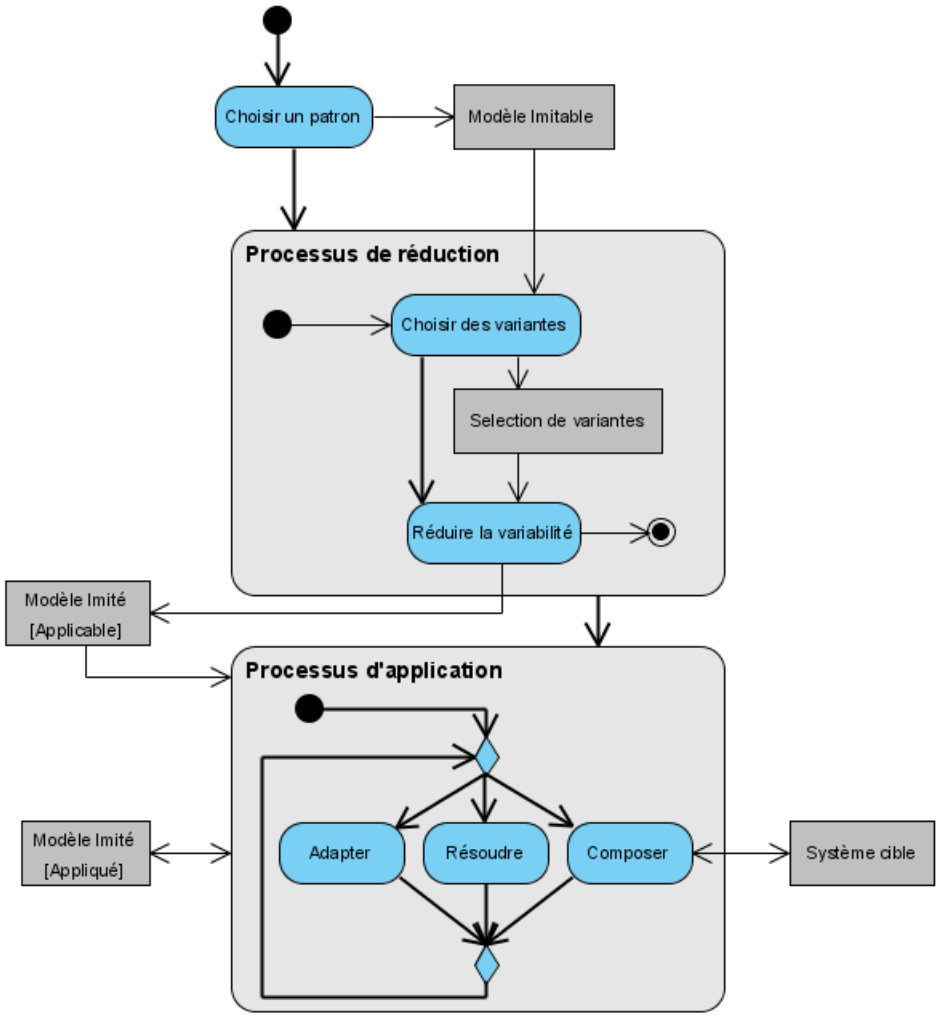


Figure 4.1 : Processus d'imitation

1.2 Processus de réduction

Le processus de réduction permet à l'ingénieur d'applications, à partir d'un modèle imitable comportant des informations de variabilité, d'obtenir un modèle plus spécifique à ses besoins, qui constitue l'imitation de base qu'il devra appliquer à son contexte. Il lui faut tout d'abord sélectionner les variantes et fonctionnalités facultatives qu'il désire imiter, tout en respectant les types de variabilité associés à chaque point de variation (cf. cardinalités, chapitre 3, section 2.1).

Nous souhaitons que ce processus soit le plus automatisable possible. C'est pour cette raison que la sélection des variantes est faite à partir de la vue fonctionnelle qui, rappelons-le, pilote les deux autres vues. La réduction de la vue fonctionnelle pilote également la réduction des vues dynamique et statique.

1.2.a Choix des variantes et réduction fonctionnelle

La réduction de la vue fonctionnelle consiste en la conservation des fonctionnalités qui ne correspondent pas au « vrai » problème de l'ingénieur d'applications. Pour chaque point de variation, l'ingénieur d'applications doit sélectionner une ou plusieurs variantes (en fonction du type de variabilité). Afin d'être la plus intuitive possible pour l'ingénieur d'applications, la sélection

s'opère sur une vue fonctionnelle représentée avec la notation simplifiée. Cette sélection (objet *Sélection de variantes* du processus d'imitation, cf. figure 4.1) servira de paramètre à la réduction.

Néanmoins, les variantes ainsi sélectionnées ne seront pas les seules à faire partie du modèle imité. En effet, certaines fonctionnalités, comme par exemple les fonctionnalités obligatoires, doivent y figurer ainsi que les fonctionnalités incluses par ces variantes. Nous détaillons ci-dessous les règles de réduction fonctionnelle.

- Les fonctionnalités « obligatoires » du modèle imitable font forcément partie de la réduction.
- Les fonctionnalités « facultatives » sélectionnées sont directement associées au client.
- Si un point de variation admet des alternatives, il sera remplacé par la variante sélectionnée.
- Si un point de variation admet des options, il sera conservé et seules les variantes sélectionnées lui seront incluses.
- Toute fonctionnalité qui n'est ni un point de variation, ni une variante, mais qui est incluse à une fonctionnalité « obligatoire » ou sélectionnée, fera partie de la réduction, tout comme l'inclusion elle-même.

La figure 4.2 représente la vue fonctionnelle (dans la notation simplifiée) du patron « Observateur », ainsi que les choix d'un ingénieur d'applications en termes de variantes (gros pointillés). Les éléments qui feront automatiquement partie de la réduction sont encadrés avec un trait plein. Conformément à la cinquième des règles énoncées ci-dessus, le cas d'utilisation *notifier* fera partie de la réduction (petits pointillés).

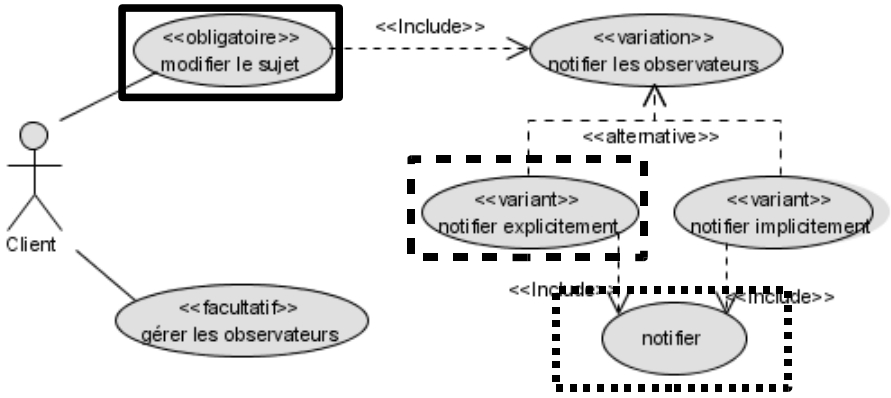


Figure 4.2 : Sélection des variantes pour le patron « Observateur ».

La figure 4.3 ci-dessous illustre la vue fonctionnelle issue de la réduction. On remarque que les stéréotypes relatifs à la variabilité ont disparu. En effet, cette vue fait partie d'un modèle imité où les informations de variabilité n'ont plus lieu d'être. Cette vue est représentée uniquement avec des concepts UML2.

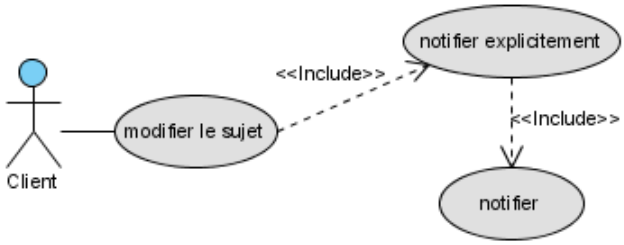


Figure 4.3 : Patron « Observateur » : Une réduction fonctionnelle.

1.2.b Réduction des vues dynamique et statique

A l'instar de la construction du modèle imitable (cf. chapitre 3), la réduction des vues dynamique et statique découle de celle de la vue fonctionnelle. Nous rappelons ici que la vue dynamique d'un modèle imitable est constituée d'un ensemble organisé d'interactions, dont la structure reflète les dépendances entre fonctionnalités. La vue statique d'un modèle imitable est quant à elle constituée d'autant de fragments statiques que de fonctionnalités.

Vue dynamique

La réduction dynamique s'opère en écartant les interactions des fonctionnalités ne faisant pas partie de la réduction fonctionnelle :

- Tous les fragments qui correspondent à des fonctionnalités non sélectionnées, ainsi que leur contenu, sont supprimés.
- Le ou les fragments combinés des variantes sélectionnées (*alt* pour les alternatives, *seq* et *strict* pour les options) sont supprimés, mais pas leur contenu.

La figure 4.4 illustre la vue dynamique du modèle imité issu de la réduction fonctionnelle du patron « Observateur » présentée plus haut. On ne retrouve que les interactions relatives aux cas d'utilisation de la vue fonctionnelle réduite. Tout comme pour la vue fonctionnelle, les informations de variabilité sont supprimées. Cette vue s'exprime donc exclusivement avec des concepts du méta-modèle UML 2.

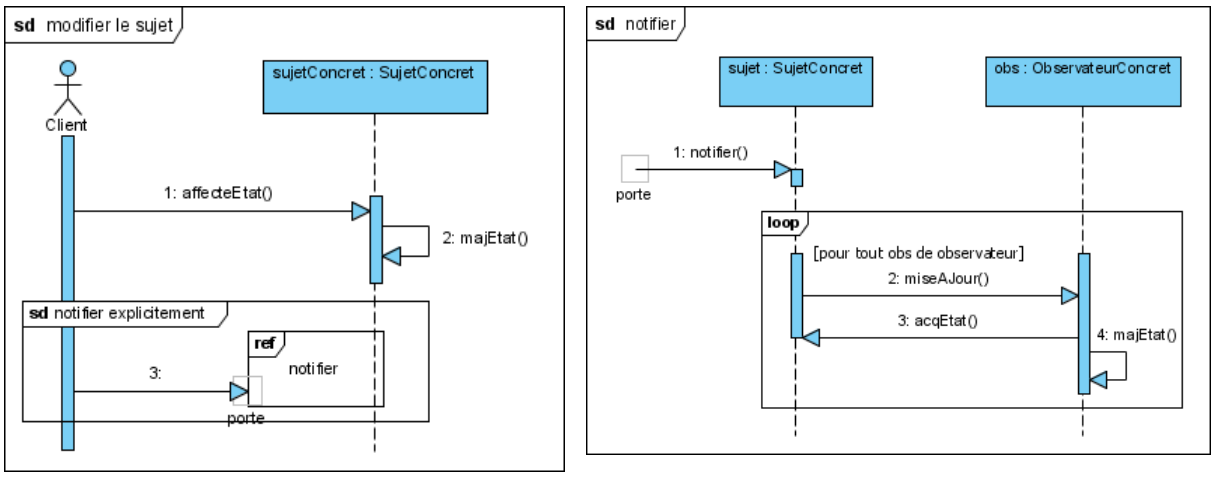


Figure 4.4: Patron « Observateur » : Une réduction dynamique.

Vue statique

A l'instar de la réduction dynamique, la réduction statique consiste en l'élimination des éléments statiques ne faisant pas partie des fonctionnalités réduites. Nous rappelons que chaque fonctionnalité du modèle imitable a donné lieu à un fragment statique comportant uniquement les éléments nécessaires à sa réalisation. Les fragments statiques des fonctionnalités réduites doivent être fusionnés au sein de la vue statique de l'imitation.

Puisque le modèle imité produit par la réduction est en fait une première version de l'imitation, les éléments statiques qui le composent sont des éléments imités. Conformément au méta-modèle présenté en chapitre 3, ces derniers sont donc associés à l'élément imitable dont ils sont issus.

Cependant, au sein d'une même imitation, certains éléments du modèle imité peuvent être associés à plusieurs éléments imitables. En effet, certains éléments de la solution peuvent être représentés dans plusieurs fragments statiques de fonctionnalités réduites (principalement des classes).

La figure 4.5 montre le résultat de la réduction statique issue de la réduction fonctionnelle du patron « Observateur » présentée plus haut. Cette figure insiste sur le fait que les classes communes aux fragments *modifier le sujet, notifier les observateurs*, *notifier explicitement* (qui incluent les éléments de *notifier)* ont été fusionnées.

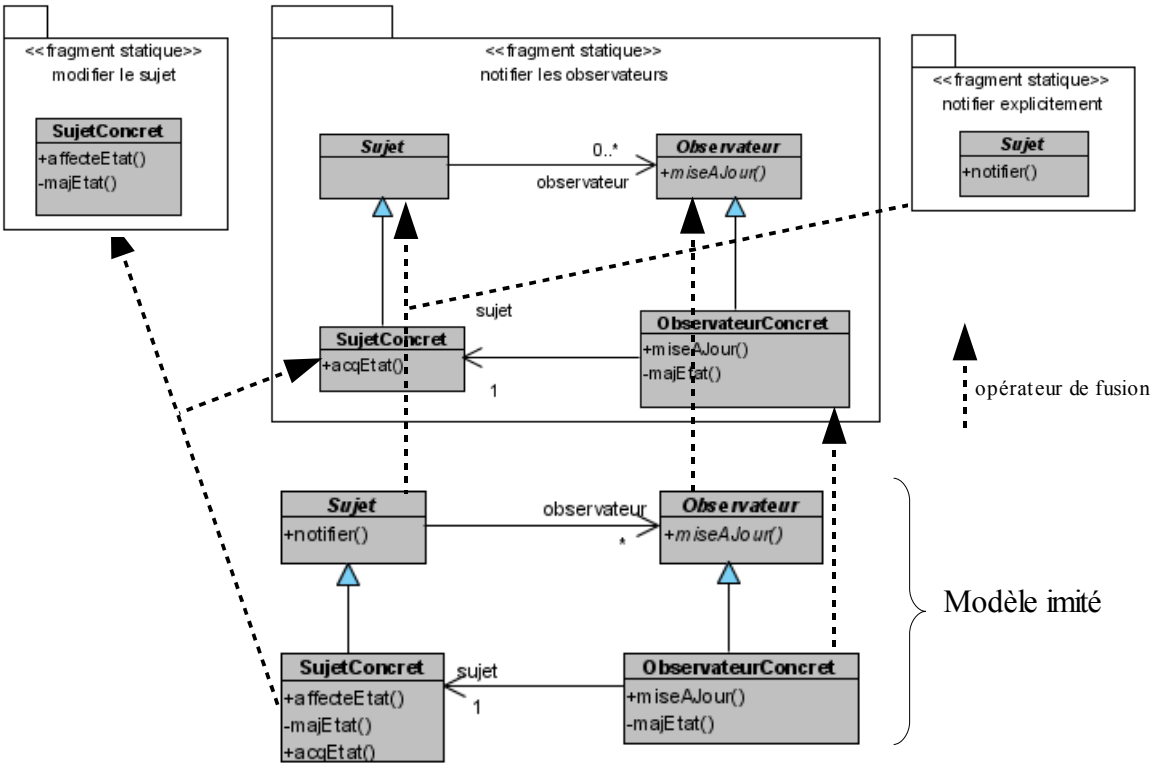


Figure 4.5: Patron « Observateur » : Une réduction statique.

La classe *SujetConcret* du modèle imité correspond donc à deux éléments imitables (de type *ClasseImitable*) provenant des fragments *modifier le sujet* et *notifier les observateurs*. La figure 4.6 illustre, par un diagramme d'objets, les liens d'imitation de cette classe avec ses éléments imitables. Les objets à fond clair font partie de l'imitation, les autres de la solution.

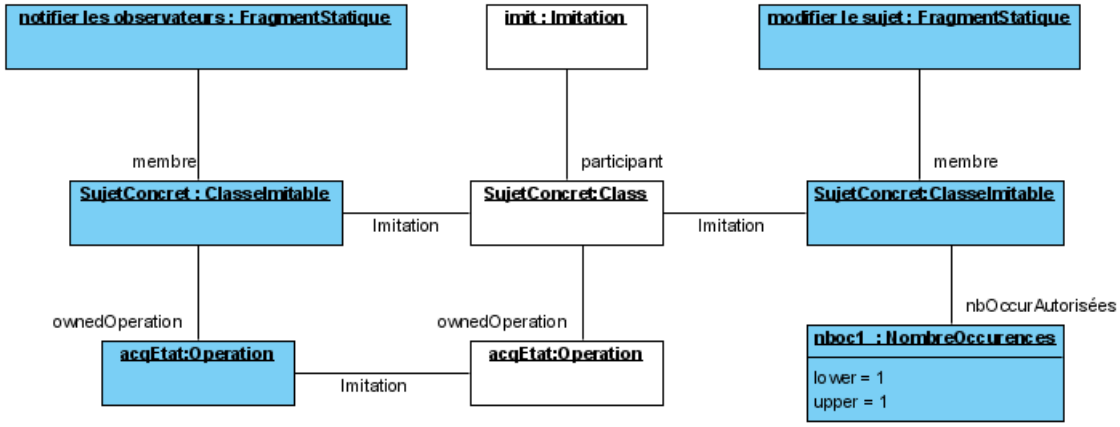


Figure 4.6 : Patron « Observateur » : Réduction statique, diagramme d'objets.

1.3 Processus d'application

Le processus d'application consiste en la prise en compte du contexte d'imitation dans le modèle imité. Comme montré ci-dessus, dans sa première version (adaptable), le modèle imité est composé d'éléments de modélisation UML2 standards associés à des éléments imitables de la solution. Il y a trois types d'activités qu'un ingénieur d'applications peut opérer sur un modèle imité : l'adaptation, la résolution et la composition. Ces trois activités sont exécutables autant de fois que nécessaire, sans ordre prédéfini.

1.3.a Adaptation

L'adaptation permet au concepteur de renommer les éléments de modélisation (nous rappelons qu'il n'y a aucune contrainte à ce sujet) et de faire évoluer le modèle imité tout en restant conforme aux propriétés génériques exprimées dans les éléments imitables associés. Toute modification affectant un élément de l'imitation ne doit pas violer les contraintes OCL induites par les propriétés génériques. Dans le cas contraire, la modification rendrait l'imitation invalide.

Il est également possible d'ajouter des imitations d'éléments imitables, à condition bien sûr que les propriétés génériques traitant des occurrences d'imitation soient respectées. Par exemple, lors de l'adaptation d'une imitation du patron « Observateur », un ingénieur d'applications pourra ajouter autant d'occurrences d'imitation de la classe *ObservateurConcret* qu'il le souhaite. Bien entendu, les éléments ainsi ajoutés à l'imitation seront associés à leurs éléments imitables. De manière similaire, la suppression d'un élément du modèle imité n'est autorisée que si elle n'invalide pas l'imitation.

Une imitation (méta-entité *Imitation)* n'est composée que d'éléments issus d'éléments imitables de la solution. Si, toujours dans le cadre d'une imitation du patron « Observateur », l'ingénieur d'applications voulait ajouter un attribut à l'imitation de la classe *Sujet*, cet attribut serait forcément membre de la classe imitée mais pas de l'imitation.

1.3.b Résolution

La résolution permet à l'ingénieur d'applications de déclarer qu'il a « résolu » une note « A Faire », c'est-à-dire qu'il a effectué les modifications demandées par la note. Bien souvent, ces modifications consistent en des ajouts qui, conformément au paragraphe précédent, ne feront pas

partie de l'imitation à proprement parler. La seule conséquence de cette activité est la disparition de la note.

1.3.c Composition

La composition des imitations permet d'intégrer une imitation dans le système cible, c'est à dire mettre en relation les éléments qui la composent avec d'autres éléments (participant ou non à une imitation) de façon à résoudre tout ou partie d'un problème plus large. La figure 4.7 ci-dessous rappelle la structure finale du cas d'étude, où cinq patrons ont été composés.

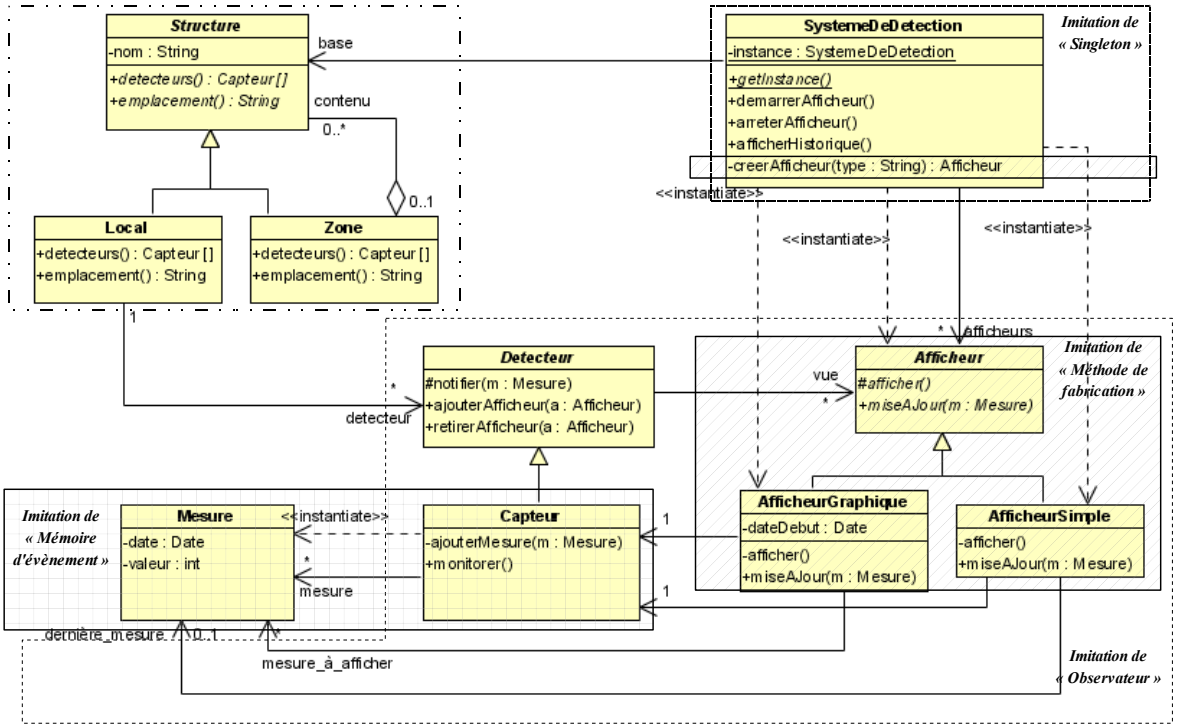


Figure 4.7 : Récapitulatif des patrons imités (tiré du cas d'étude)

La composition est une activité complexe pour laquelle nous ne faisons pas de proposition détaillée, même si une première étude est présentée chapitre cinq. Nous pouvons néanmoins décrire les propriétés et règles fondamentales que doit respecter un mécanisme de composition s'appliquant à notre approche.

- La composition d'une imitation peut se faire selon trois mécanismes :
  - l'ajout d'un ou plusieurs éléments de modélisation, qui n'appartiendront à aucune des imitations mais qui serviront à leur mise en relation (cf. figure 4.7, association entre *Local* et *Détecteur*);
  - l'identification d'une correspondance entre deux éléments dont l'un au moins participe à une imitation. C'est par exemple le cas dans la figure 4.7 où la classe *Capteur* participe aux imitations de « Mémoire d'évènement » et d' « Observateur ». Dans ce cas, la relation entre les imitations est représentée par l'élément lui même;
  - La composition peut également s'exécuter *a priori*, c'est à dire lors de la création du modèle imité. Dans ce cas, il faut identifier la correspondance (au sens du paragraphe ci-dessus) dès la fin de la réduction de manière à utiliser

les éléments déjà membres du système pour la construction de la première version du modèle imité.

- La composition conserve les propriétés génériques (et donc les contraintes) de toutes les imitations concernées. Une composition ne peut rendre une imitation invalide. Toutefois, dans le cas où deux propriétés génériques de même type portent sur un même élément et sont issues de deux imitations à composer, deux cas se présentent :
  - Les propriétés sont incohérentes. Par exemple si une classe doit rester concrète dans une imitation mais doit rester abstraite dans une autre. La composition est alors interdite.
  - Les propriétés sont cohérentes, mais l'une est probablement plus restrictive que l'autre. Dans ce cas la composition est autorisée, mais, de fait, c'est la plus restrictive qui régira les adaptations de l'élément concerné, puisque toutes les contraintes doivent être vérifiées en permanence. Si elles sont identiques, il n'y a pas de problème particulier.

2 Mise en œuvre

Dans cette section nous présentons des éléments de mise en œuvre de nos propositions. Un premier prototype est principalement destiné à la validation des imitations. Un second prototype, plus récent, permet de supporter le processus d'imitation, c'est-à-dire l'automatisation des deux sous-processus de réduction et d'application ainsi que la validation des imitations.

2.1 Un premier prototype d'imitation/validation

Dans de précédents travaux sur la généricité des solutions de patrons (Arnaud et al., 2004) nous avons réalisé un prototype sous la forme d'une extension de l'AGL MagicDraw UML (MagicDraw, 2008). Ce premier prototype, basé sur la gestion des profils et de l'API fournie par l'AGL, permet de définir des solutions de patrons (vue statique uniquement) ainsi que les propriétés génériques associées à certains éléments (nombre d'occurrences d'imitation, abstraction, etc..). Il est également possible d'imiter un patron à l'aide d'une interface graphique assurant aussi la vérification de la validité de l'imitation.

La figure 4.8 illustre la saisie des propriétés générique par l'ingénieur de patrons, une fois le modèle statique du patron « Observateur » terminé. Par exemple, il doit y avoir une et une seule imitation de la classe *Sujet* et elle doit être abstraite.

La figure 4.9 illustre l'imitation/adaptation d'un patron par un ingénieur d'applications. Les boutons *Imite* et *Elit* permettent respectivement d'ajouter un élément imité associé à un élément imitable ou de réaliser une composition. Dans cet exemple, l'opération *miseAJour* de la classe imitable *Observateur* n'est pas présente dans la classe *Afficheur* (imitation de *Observateur*), ce qui entraine une erreur de validation.

Figure 4.8: Saisie des propriétés génériques du patron « Observateur »

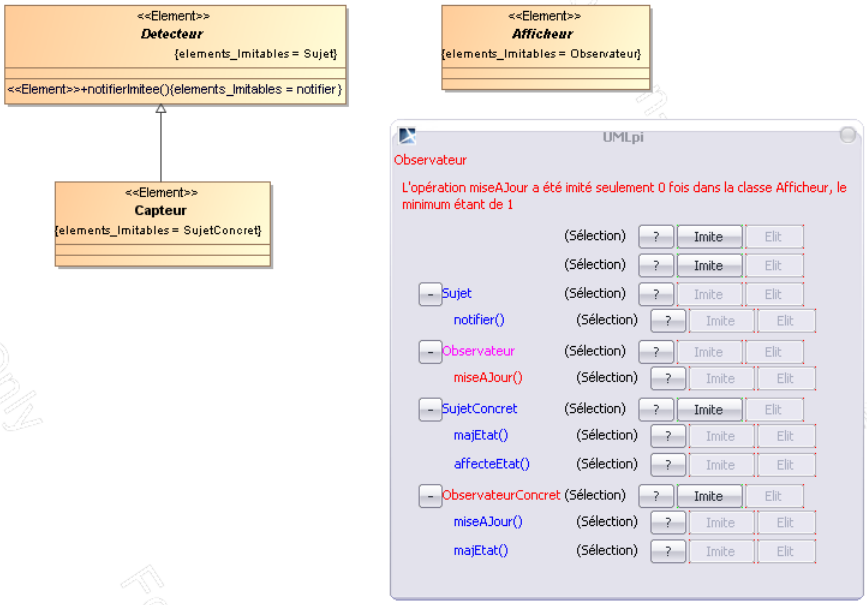


Figure 4.9 : Adaptation et validation d'une imitation du patron « Observateur » à l'aide du premier prototype

Si ce prototype réalise la majeure partie des fonctionnalités nécessaire à l'accompagnement de l'ingénieur d'applications lors du sous-processus d'application, il ne traite pas le sous-processus de réduction. De plus, l'impact sur le modèle cible (stéréotypes, valeurs marquées) nous semble trop important et nuit à sa lisibilité alors qu'un de nos objectifs est de limiter l'impact de notre approche sur les habitudes des ingénieurs d'applications. Enfin, les techniques utilisées pour la réalisation du prototype, fortement liées à MagicDraw UML, ne sont pas aisément réutilisables pour d'autres plateformes.

Aussi, nous proposons dans la suite un nouveau prototype, basé sur l'approche IDM, couvrant l'ensemble du processus d'imitation.

2.2 Ingénierie Dirigée par les Modèles : quelques outils

Bien avant la formalisation des concepts de l'IDM (cf. chapitre 2, section 2), la plupart des Ateliers de Génie Logiciel (AGL), outre modélisation sous forme graphique, étaient également capables de générer du code source ou de la documentation, voire d'autres spécifications. Certains AGL comme les outils *IBM/Rational* (Rational, 2008) disposaient même d'interfaces de programmation (API, *Application Programming Interface*) permettant de définir des extensions de l'atelier dédiées à la manipulation des éléments de modélisation. Ces outils ont par exemple permis aux ingénieurs de rendre opérationnels les profils UML au sein même des environnements de modélisation.

Plus récemment des travaux industriels et de recherche ont donné lieu à des propositions outillées qui facilitent la mise en œuvre des différents concepts de l'IDM. Ainsi, la plateforme de développement Eclipse (Eclipse, 2008), initialisée par la société IBM, s'est dotée d'une architecture cadre générique pour la gestion et la définition de modèles : EMF (*Eclipse Modeling Framework*) (EMF, 2008). Cette architecture, qui s'appuie sur le méta-modèle *Ecore* (similaire à MOF), est la base d'une grande partie des projets actuels d'outillage pour l'IDM. Sous la forme d'une extension (plugin) de la plateforme Eclipse, EMF permet de définir un méta-modèle (conforme à Ecore) à partir duquel est généré le code de manipulation et d'édition[4] des modèles conformes à ce méta-modèle. Le projet *MDT-UML2* (MDT-UML2, 2008) s'appuie sur EMF et propose une extension comportant un méta-modèle ainsi qu'une API d'exploitation de modèles UML2.

La similarité entre MOF et Ecore permet également à différent outils basés sur Eclipse d'implémenter les spécifications de MOF-QVT pour les méta-modèle EMF. C'est notamment le cas de *Kermeta* (Kermeta, 2008) et du projet *M2M* (M2M, 2008) de la fondation Eclipse. Le premier s'appuie sur un langage impératif proche de Java permettant d'ajouter du comportement aux méta-modèles et par extension d'exprimer des transformations de modèles. Le second propose un langage de définition de règles de transformation (ATL) alliant forme déclarative et impérative. Il faut également noter que le code de manipulation d'un méta-modèle EMF permet d'écrire des transformations en Java standard.

2.3 Nouveau prototype : architecture générale

La mise en œuvre de nos propositions dans le contexte de l'IDM implique certains ajustements des solutions conceptuelles que nous avons apportées dans les chapitres précédents. En particulier, le méta-modèle (cf. chapitre 3, figures 3.19 et 3.20) que nous avons proposé doit être revu afin de correspondre à certains besoins plus techniques que nous présentons ci-dessous.

- Dans nos démonstrations précédentes, les modèles de solutions et d'imitations de patrons étaient conformes à un même méta-modèle, extension d'UML. Dans cette réalisation, les imitations doivent être conformes à UML, de façon à faciliter

4 L'outil GMF (*Graphical Modeling Framework*) (GMF, 2008) permet aussi de générer un éditeur graphique à partir d'un méta-modèle EMF.

l'intégration de nos propositions dans les environnements de développement des ingénieurs d'applications. Notre objectif n'est donc pas de développer un nouvel AGL.

- Le méta-modèle que nous avons proposé étend UML en le modifiant certains éléments. C'est notamment le cas de l'association *Imitation* qui impacte l'entité *Element* (cf. chapitre 3, section 3). Le mécanisme d'extension proposé dans la spécification d'UML, basé sur les stéréotypes interdit la modification d'entités nativement présentes dans le langage et nous devons donc considérer le méta-modèle d'UML comme inaliénable. Il nous faut donc trouver une solution pour réaliser cette association tout en conservant la nature des modèles UML que doivent être les imitations.
- Notre méta-modèle suggère également que toutes les imitations d'une solution sont associées au même « objet » solution. Cette propriété impliquerait une centralisation de tous les liens solution/imitation qui n'aurait guère d'intérêt. Il est néanmoins capital qu'au sein de son environnement de développement, chaque ingénieur d'applications ait accès à cette traçabilité.

Afin de pallier à ces problèmes, nous proposons une architecture réaliste pour la mise en œuvre de nos propositions qui s'appuie principalement sur EMF et ATL. Les modèles conformes à un méta-modèle EMF peuvent être stockés sous la forme d'un fichier et, grâce à un système de référencement que nous ne détaillerons pas ici, les liens entre les éléments de différents modèles (et donc différents fichiers) peuvent être spécifiés.

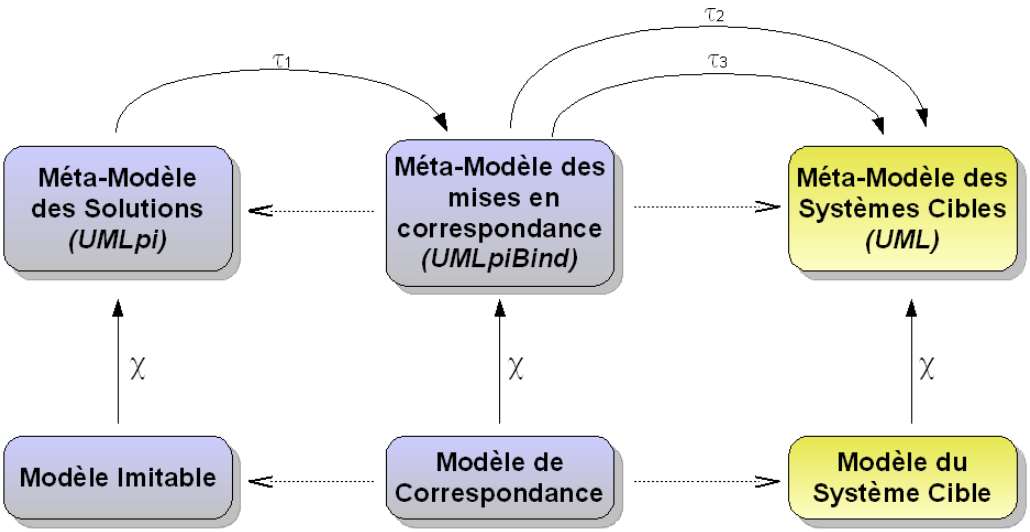


Figure 4.10 : architecture de mise en œuvre.

En se basant sur ces propriétés, nous proposons un environnement architecturé en trois parties qui supporte les deux sous-processus d'imitation (cf. figure 4.10) :

- La partie « Solution » comporte un méta-modèle EMF extension d'UML2 (fourni par *MDT-UML2*) spécifique aux solutions (complétude, variabilité et généricité) qui ne s'occupe pas de la relation d'imitation ni des imitations. Toutes les solutions,

stockées sous la forme d'un fichier XML, seront conformes (χ) à ce méta-modèle (nommé *UMLpi*, pour *UML Pattern Imitation,* donné en annexe).

- La partie « Cible » n'est autre que les systèmes cibles et leur méta-modèle : UML2. Les éléments issus des imitations y sont adaptés et composés.
- La partie « Correspondance » permet de faire le lien entre les éléments imitables de la partie « Solution » et leurs éléments imités de la partie « Cible ». Un modèle de correspondance a pour élément racine un élément de type *Imitation*. Chaque imitation de solution donnera lieu à un modèle de correspondance conforme au méta-modèle *UMLpiBind*, réalisé à l'aide d'EMF, illustré avec la figure 4.11. C'est ce modèle qui sera responsable de la validité des imitations, puisqu'il a accès d'une part aux éléments imitables et d'autre part aux éléments imités.

La figure 4.11 ci-dessous représente le méta-modèle *UMLpiBind* destiné à mettre en relation les éléments des systèmes cibles (*NamedElement* issu d'UML) avec les éléments imitables dont ils sont des imitations (*ElementImitable* issu d'*UMLpi*) au sein d'un *ElementBinder*. On retrouve la structure d'un modèle à objets classique (type MAC, cf. chapitre 2 section 4.8). Une imitation se compose de participants (relation *participant*) qui sont en fait des *ClassBinder* ou des *AssociationBinder* (cf. contrainte de l'élément *Imitation*). Les élément imités d'un *ElementBinder* doivent obligatoirement respecter les contraintes induites par les propriétés génériques de ses sources.

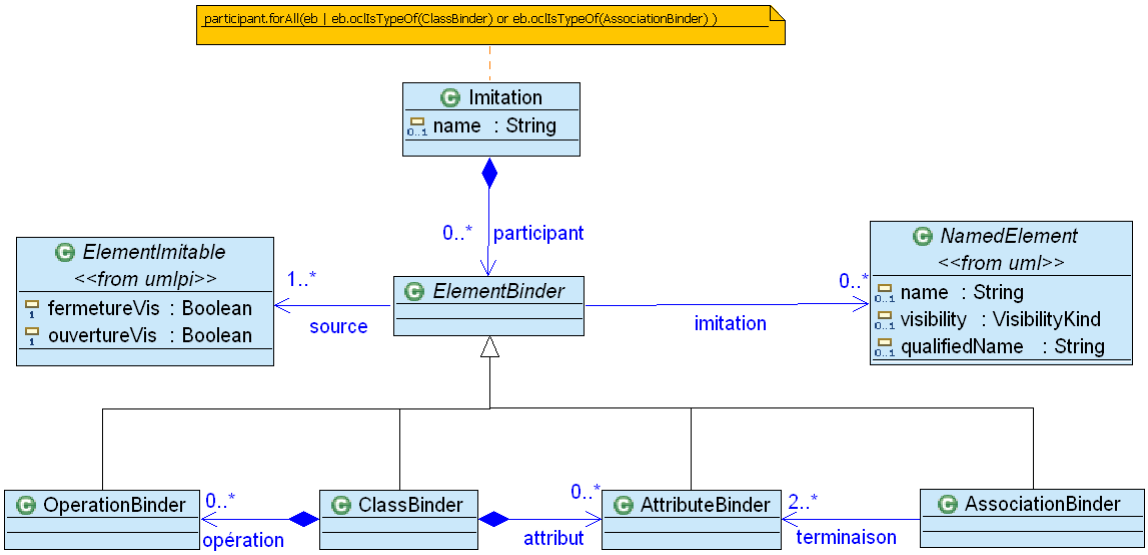


Figure 4.11 : Méta-modèle UMLpiBind.

Les transformations τ_1 et τ_2 illustrées figure 4.10 permettent de réaliser l'activité de réduction.

La transformation τ_1 construit un premier modèle de correspondance (sans éléments imités), à partir d'un modèle de solution et d'une sélection de cas d'utilisation conforme aux règles de variabilité. A partir des fragments statiques sélectionnés, un *ElementBinder* est créé pour chaque ensemble d'éléments imitables (relation *source*) représentant un élément de la solution. Autrement dit, un *ElementBinder* permet d'accéder à tous les éléments imitables d'un même concept de la solution (issu d'un cas d'utilisation sélectionné) et ainsi aux propriétés génériques qui leurs sont associées. Par exemple, dans le patron « Observateur », la classe *SujetConcret* du fragment statique

de *modifier le sujet* et la classe *SujetConcret* du fragment statique de *notifier les observateurs* représente le même concept.

La transformation τ_2 vise à peupler le modèle cible avec la première version de l'imitation issue de la réduction. A chaque *ElementBinder* du modèle de correspondance est associé un élément UML (relation *Imitation*). Cet élément imité est créé dans le modèle du système cible, sauf dans le cas d'une composition *a priori* (cf. 1.3.c) ou seule la relation *imitation* est modifiée.

La figure 4.12 ci-dessous présente partiellement le résultat de la réduction statique (transformation τ_1 puis τ_2) similaire à la figure 4.5 où la classe *Capteur* du système cible est une imitation issue des « deux » classes imitables *SujetConcret*.

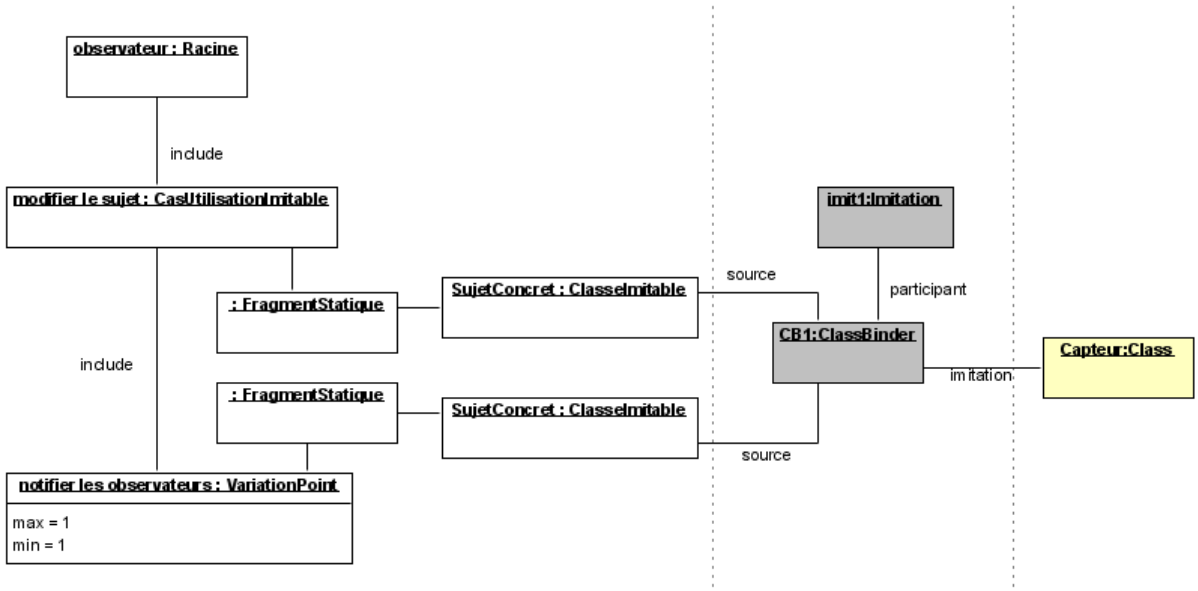


Figure 4.12: Modèle imitable, correspondance et système cible : un exemple.

La transformation τ_3 est en fait un ensemble de transformations réalisant l'ajout *a posteriori* d'éléments imités, dans le cadre de l'adaptation. Il y a autant de transformations τ_3 que d'*ElementBinder* dans le modèle de correspondance.

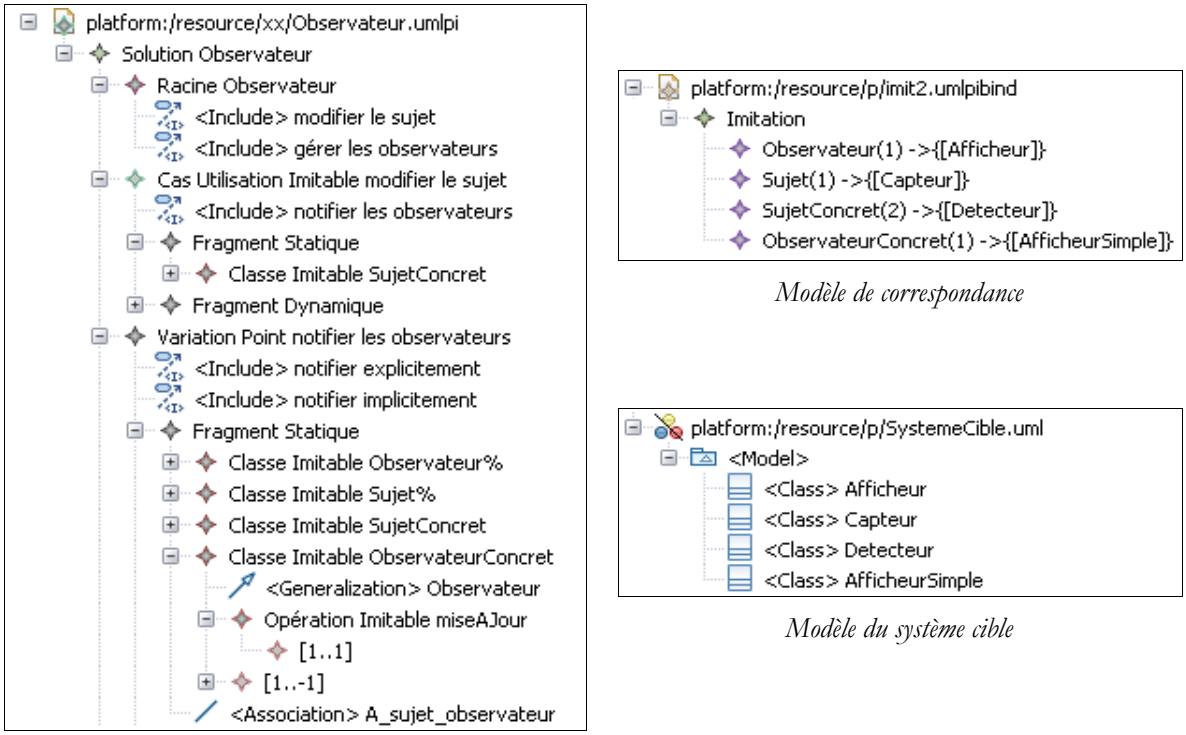
La transformation τ_1 est implémentée à l'aide de code Java exploitant l'API EMF des méta-modèles *UMLpi* et *UMLpiBind*. La transformation τ_2 est quant à elle réalisée à l'aide d'un ensemble de règles ATL. Les transformations de type τ_3, destinées à l'ajout ou à l'attachement d'éléments imités, peuvent être générées lors de la transformation τ_1.

2.4 Utilisation du nouveau prototype

La partie gauche de la figure 4.13 illustre, par l'intermédiaire de l'éditeur arborescent commun à tout méta-modèle EMF, une partie du modèle imitable du patron « Observateur ». On y retrouve les cas d'utilisation imitables, leurs fragments statiques ainsi que les éléments statiques et leurs propriétés génériques. Par exemple, dans *modifier le sujet*, la classe *SujetConcret* comporte un élément nommé *[1..-1]*, c'est en fait le nombre d'occurrences d'imitation autorisées (-1 représente la

valeurs infinie *). Le caractère % à droite du nom d'une classe signifie que la propriété générique « doit rester abstraite » est activée.

La partie droite de la figure 4.13 montre un modèle de correspondance et le modèle cible auquel il est associé. Ces deux modèles sont en fait le résultat d'une réduction du patron « Observateur », c'est-à-dire de l'application des transformations τ_1 et τ_2 sur le modèle imitable présenté en partie gauche. Tous les participants de l'imitation sont des *ClassBinder*. Le nombre entre parenthèse à gauche de la flêche représente le nombre de classes imitables c'est à dire la cardinalité de la relation *source* (2 pour *SujetConcret*). A droite de la flèche sont représentés les élément imités (relation *imitation*), appartenant au système cible.



Modèle de correspondance

Modèle du système cible

Modèle imitable

Figure 4.13 : Modèle imitable du patron « Observateur », modèle de correspondance et modèle du système cible.

La figure 4.14 illustre, à partir de l'exemple précédent, le résultat de la transformation de type τ_3 permettant d'ajouter des imitations de la classe *ObservateurConcret* lors de l'adaptation. Ici, la classe imitée (*AfficheurGraphique*) n'existait pas dans le modèle cible, et a donc été créée.

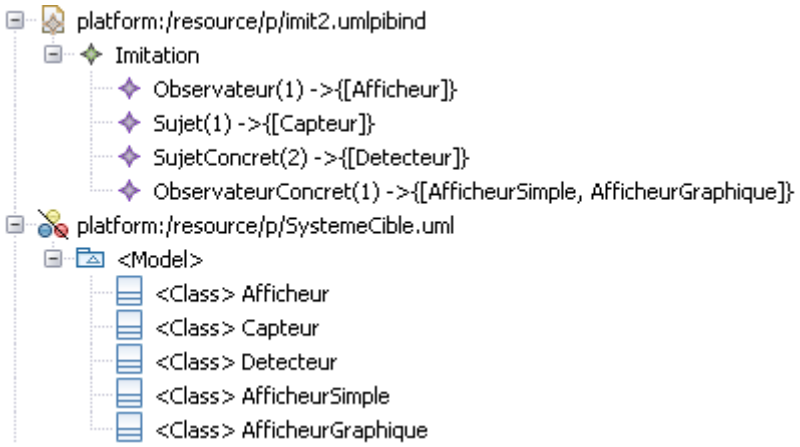


Figure 4.14 : Résultat d'une transformation de type τ_3.

Grâce au module de validation inclus à EMF, la validation des imitations peut se faire à l'aide de code Java mais également par l'ajout de contraintes OCL, similaires à celles présentées chapitre 3. Puisque la définition des propriétés génériques implique de fournir une contrainte OCL, nous utilisons ces contraintes en les appliquant à l'architecture générale. En effet, le point d'entrée de toute validation est l'objet *Imitation* qui englobe toutes les mises en correspondance entre éléments imitables et imités.

3 Synthèse

Dans ce chapitre nous avons présenté un processus d'imitation tirant partie des spécifications de solutions décrites dans le chapitre trois. Les sous-processus de réduction et d'application qui le composent permettent à l'ingénieur d'applications d'exploiter les informations de variabilité et de généricité.

Ce processus est en partie automatisé par un prototype basé sur une approche IDM et s'appuyant principalement sur les outils EMF et ATL. Ce prototype adopte une architecture tri-partie permettant de découpler les imitations des solutions dont elles sont issues par l'intermédiaire d'un modèle de correspondance.

Cet outil est en cours d'amélioration pour supporter l'ensemble des éléments de modélisation des solutions de patrons (fragments dynamiques en particulier). Il est pour l'instant basé sur un éditeur arborescent généré automatiquement par EMF mais le développement d'un éditeur graphique est nécessaire pour en faciliter l'utilisation. L'utilisation de GMF est une piste idéale pour le développement de cet éditeur.

CHAPITRE 5 TRAVAUX CONNEXES : RÉFLEXIONS SUR LA COMPOSITION

La composition n'est pas une problématique centrale de cette thèse. Dans le cadre d'une approche globale de l'imitation, cet aspect doit être considéré comme une problématique à part entière.

Néanmoins, lors de la description du processus d'imitation, une définition intuitive de la composition a été donnée. De plus, dans le cadre de précédents travaux (Arnaud et al., 2005), nous proposions une première caractérisation de la composition selon deux opérateurs : délégation et fusion. Ces travaux, que nous présentons dans ce chapitre avec l'éclairage des propositions précédentes, ne traite que de la composition de la vue statique, et mettent en jeu des éléments exclusivement issus d'imitations.

1 Composition d'imitations : deux opérateurs

La composition d'imitations consiste en l'intégration de leur contenu dans le cadre du système cible pour lequel elle a été produite. En effet, il est rare que les éléments participant à une imitation soient isolés dans le système cible. Ils doivent donc collaborer avec les éléments présents au préalables, selon les besoins de l'ingénieur d'applications.

Dans la description intuitive de la composition faite chapitre quatre, nous avons distingué deux mécanismes permettant d'intégrer les éléments statiques d'une imitation que nous pouvons généraliser sous le forme de deux opérateurs de composition.

- Le premier mécanisme consiste en l'identification d'éléments communs (au sens du contexte d'application) entre l'imitation et le système cible. L'objectif est alors de superposer les deux structures que proposent ces deux fragments. Dans le cas d'imitations de parons orientés-objet, cette unification se fait principalement par classes, autrement dit de concepts du domaine modélisé. Ce type de composition est réalisé par un opérateur dit de « fusion ». Cet opérateur doit conserver les propriétés génériques de l'imitation ainsi que celles des imitations déjà présentes dans le système.
- Le second mécanisme de composition met en relation les éléments de l'imitation avec ceux du système cible par le biais de nouveaux éléments. En fait, l'objectif est de permettre à un des deux fragments d' « utiliser » certaines fonctionnalités du second. Dans un contexte orienté-objet, cette mise en relation consiste en l'ajout d'une association entre deux classes (l'une issue de l'imitation, l'autre du système). La composition est alors réalisée par un opérateur dit de « délégation ».

Dans les chapitres précédents, et afin de faciliter la traçabilité d'une imitation, les éléments qui la composent ont été regroupés au sein d'un conteneur. Cette encapsulation, associée à la problématique de délégation, fait fortement penser aux problèmes de l'assemblage des composants, et plus particulièrement dans le domaine des composants métier. Nous proposons par la suite d'utiliser le modèle de composants métier de la démarche Symphony (Hassine et al., 2002) (Hassine, 2005) afin de supporter la composition d'imitations.

2 Composants et opérateurs

2.1 Le modèle de composants Symphony

Le modèle de composants proposé par la démarche Symphony est inspiré de la technique CRC (Classe-Responsabilité-Collaboration) (Wirfs-Brock et al., 1990). Un composant métier est modélisé par un paquetage composé de trois parties (cf. figure 5.1) :

- une partie contrat avec l'extérieur (*ce que je sais faire*),
- une partie structurelle (*ce que je suis*),

- une partie collaboratrice (*ce que j'utilise*).

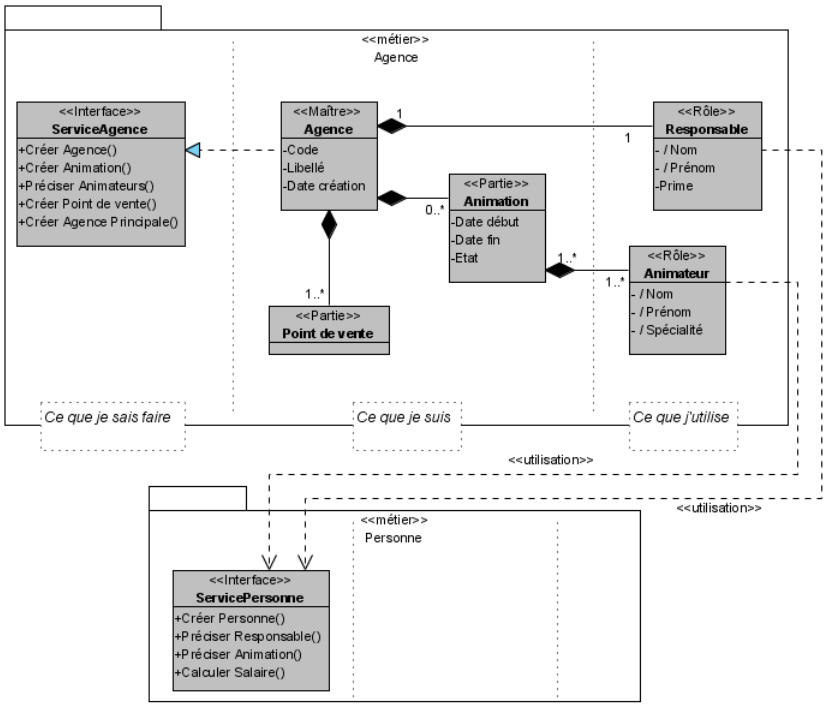


Figure 5.1: Illustration du modèle à composant Symphony.

Les trois parties du composant sont respectivement représentées par quatre types d'objets stéréotypés par :

- *Maître* : c'est l'objet principal du composant pour lequel les services sont réalisés. Il est qualifié de hiérarchiquement supérieur et est identifiable par tout acteur externe (cf. *Agence*).
- *Partie* : c'est un objet complémentaire à l'objet « Maître ». Il est identifié par son attachement à l'objet « Maître » auquel il est relié par une relation de composition (cf. *Point de vente*).
- *Rôle* : il représente un objet fournisseur de l'objet « Maître ». Tous les services des autres composants sont appelés au travers d'un objet « Rôle ». Un objet « Rôle » est relié à son maître (ou à une partie) par une relation de composition. Cette notion est proche de celle des services attendus (cf. *Responsable*). Un objet « Rôle » est un objet du niveau conceptuel, il peut être implémenté comme une classe ou comme une simple association (s'il ne comporte pas de propriétés).
- *Interface* : c'est l'objet représentant les services offerts par le composant. Il supporte les opérations représentant d'une part, les cas d'utilisation du composant et d'autre part, les services attendus par les rôles qu'il joue dans les autres composants (cf. *ServiceAgence*).

Le modèle de composants Symphony propose également plusieurs relations inter-composants. Seule la relation d'utilisation, utile pour notre étude, est décrite ici :

- *Utilisation* : cette relation permet de décrire le rôle joué par un composant pour un autre en termes des services attendus. Elle met en œuvre une classe *Rôle* dans le composant client relié au composant fournisseur (via son interface) par une

dépendance stéréotypée « utilisation ». La classe *Rôle* peut définir en son sein un ensemble de propriétés dérivées, images de propriétés de la classe *Maître* cible ainsi que des propriétés intrinsèques. Dans la figure 5.1, le composant *Agence* utilise le composant *Personne* avec les rôles *Animateur* et *Responsable*.

2.2 Encapsuler une imitation dans un composant

Dans un premier temps, l'encapsulation d'une imitation dans un composant métier consiste à construire la partie contrat avec l'extérieur et la partie structurelle. Pour ce faire, il faut donc tout d'abord identifier la classe « Maître » puis définir l'interface qu'elle réalise.

Le patron « Singleton » du GoF (cf. Annexes) a pour but de garantir qu'une classe n'a qu'une seule instance et fournit un point d'accès de type global à cette classe. Nous présentons figure 5.2 la structure originale de la solution de ce patron ainsi qu'une imitation encapsulée (conforme au cas d'étude du chapitre 1). On remarque que, hormis l'attribut d'instance et la méthode de récupération de l'instance, qui sont considérées comme obligatoires (nombre d'occurrences d'imitation 1..1), l'adaptation n'a pas encore pris en compte les autres propriétés.

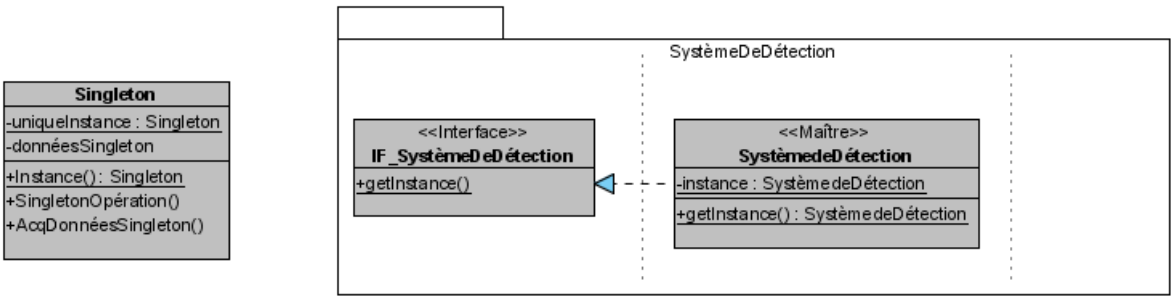


Figure 5.2 : Structure du patron « Singleton » et imitation encapsulée : le composant SystèmeDeDétection.

L'imitation du patron « Observateur » évoquée dans le cas d'étude donne lieu à l'encapsulation présentée figure 5.3. Les classes « Maître » sont les classes *Détecteur* et *Capteur*(imitation de *Sujet et SujetConcret)* et les observateurs sont des classes « Partie » puisque toutes les fonctionnalités de premier niveau (cf. chapitre 3) sont effectuées par l'intermédiaire des sujets. D'ailleurs, l'interface d'une classe « Maitre » comportera toujours les opérations réalisant les cas d'utilisation de premier niveau. Dans cet exemple, c'est la variante « notifier implicitement » qui a été choisie et, ainsi, la méthode *notifier* n'apparait pas dans l'interface. Par contre, les fonctionnalités facultatives d'ajout et de suppression d'observateurs ont été imitées.

La figure 5.4 illustre l'encapsulation de l'imitation du patron « Composite » nécessaire au cas d'étude pour représenter la structure des bâtiments. Ici, une première opération spécifique consiste en la méthode *emplacement*, appartenant à l'interface, localisant la structure (cf. chapitre 1, section 4.2.c).

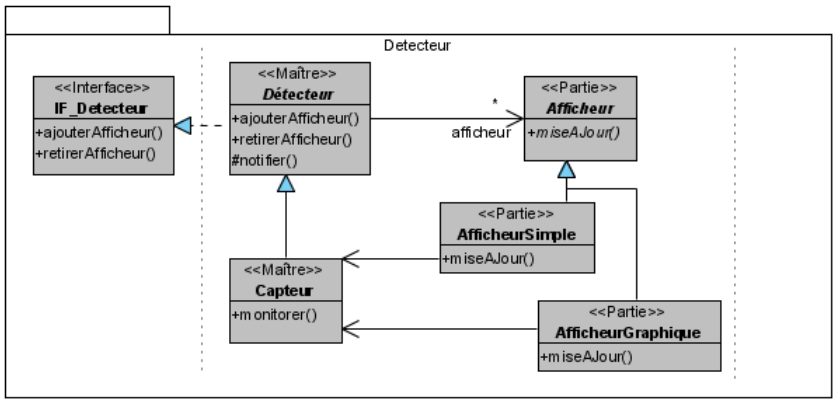


Figure 5.3 : Encapsulation d'une imitation du patron « Observateur » : le composant Détecteur.

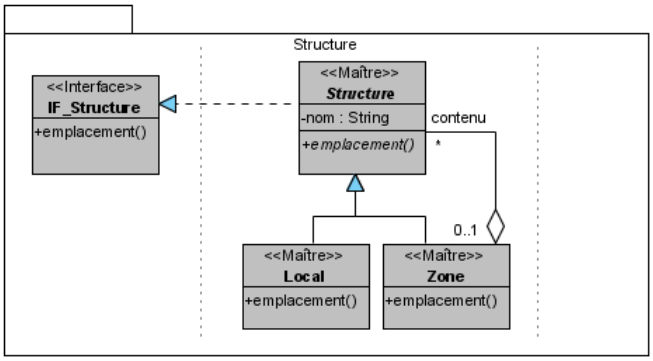


Figure 5.4 : Encapsulation d'une imitation du patron « Composite » : le composant Structure.

2.3 Composition par délégation

Symphony définit la manière de créer des liens d'assemblage de type utilisation entre les composants, par l'intermédiaire de classes « Rôle » et de dépendances de type « Utilisation » entre ces dernières et les interfaces des composants. Il s'agit en fait de la mise en place d'une structure de délégation. Nous pouvons donc utiliser ce mécanisme pour réaliser les composition d'imitation par délégation.

Dans notre exemple, le contexte d'imitation nous impose de préciser qu'un *Local* possède un ensemble de capteurs pour lesquels il pourra être demandé d'afficher des *Afficheur*. Pour réaliser ce lien, nous allons composer, par délégation, le composant *Structure* au composant *Détecteur*. Le résultat de cette composition est présenté figure 5.5.

- Une classe « Rôle » *DétecteurLocal* a été créée dans la partie collaboratrice du composant *Structure* qu'elle utilise le composant *Détecteur*. Cette assemblage permet d'adapter le composant *Structure* avec une nouvelle imitation de *opération* rendant la liste des détecteurs d'une structure (méthode *détecteurs*). De plus la classe « Maître » *Local* est dotée d'une interface plus spécifique, permettant la création et la suppression des afficheurs à travers ce rôle.

- Le composant *SystèmeDeDétection* a également été assemblé (rôle *Base)* avec le composant *Structure* afin de permettre, la gestion des structures et de leurs capteurs par le système (cf. figure 5.5). Si elle n'est pas reliée directement au système de détection, l'interface d'un *Local* est néanmoins accessible et, en parcourant l'arbre des structures pointée par *Base*, on pourra retrouver les locaux et utiliser leur interface spécifique pour ajouter des afficheurs. *SystèmedeDétection est donc* adapté pour réaliser les fonctionnalités *démarrerAfficheur* et *arrêterAfficheur.*

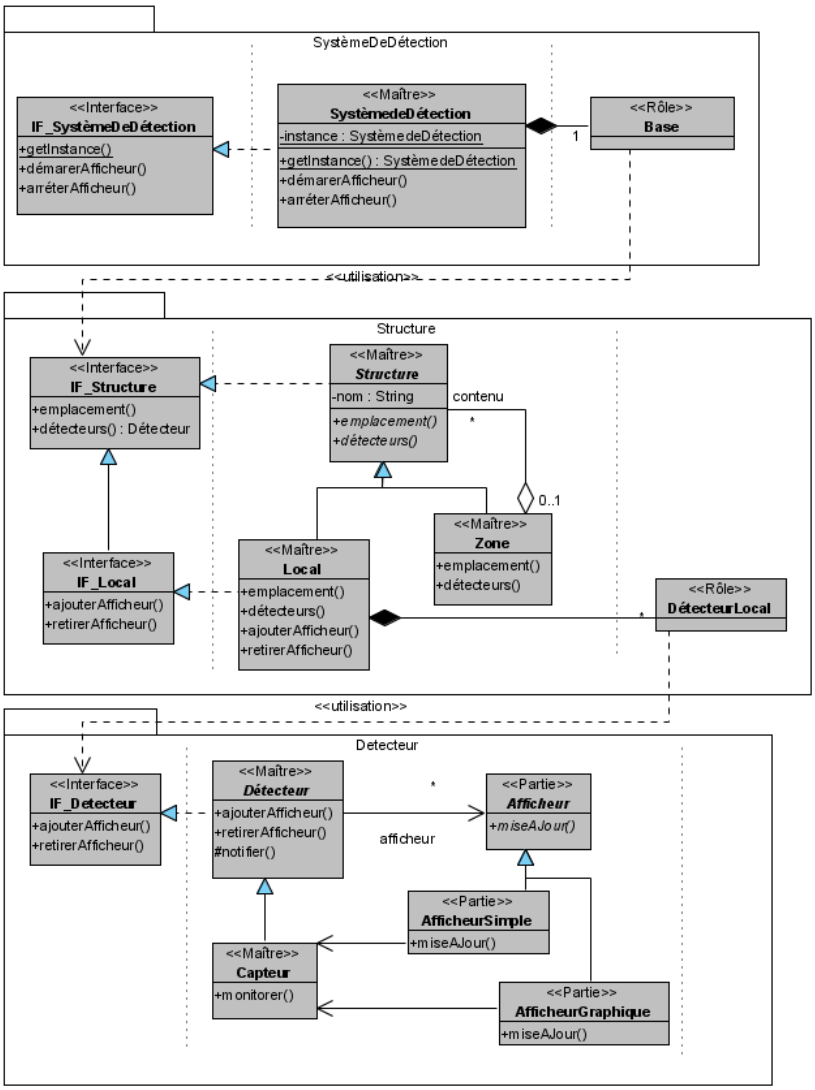


Figure 5.5 : Application de l'opérateur de délégation au cas d'étude.

2.4 Composition par fusion

2.4.a Principes

L'ingénieur d'applications a recours à la composition par fusion lorsqu'il veut réellement mettre en commun deux fragments de spécification, et plus particulièrement ici, deux classes de composants d'imitations. Ce ne sont pas les interfaces de manipulation et d'utilisation de services qui intéressent le concepteur, mais plutôt une mise en commun de la structure de deux composants.

Cette approche se focalise uniquement sur la composition des éléments statiques d'une imitation. Ainsi, comme dans le monde orienté-objet, l'unité de fusion de base est la classe. Nous proposons donc un opérateur de fusion de classes, similaire à la dépendance « merge » proposée par l'OMG (OMGa, 2005), mais adaptée aux composants que nous manipulons. La fusion de deux imitations, dont le résultat est un nouveau composant, est réalisée par un ensemble de fusions de classes.

La figure 5.6 illustre l'opérateur de fusion. Dans cet exemple les classes *A* et *B* sont fusionnées dans un nouveau composant. L'opérateur est représenté graphiquement par un lien de généralisation UML stéréotypé « Fusion ». Les règles de fusion sont les suivantes :

- La classe fusionnée comporte l'union des propriétés de ses deux sources. Ainsi la classe *AB* comporte les attributs, les méthodes, les classes « Partie » et les classes « Rôle » de *A* et de *B*. Par simplification, deux propriétés homonymes sont considérées comme synonymes, donc unique dans le résultat. Les associations, et notamment leurs terminaisons, sont considérées comme des propriétés.
- Les interfaces des classes « Maître » fusionnées sont également fusionnées.
- L'arbre de spécialisation des classes « Maître » du résultat est équivalent, aux fusions près, à l'union des arbres de spécialisation des classes « Maître » des composants sources.

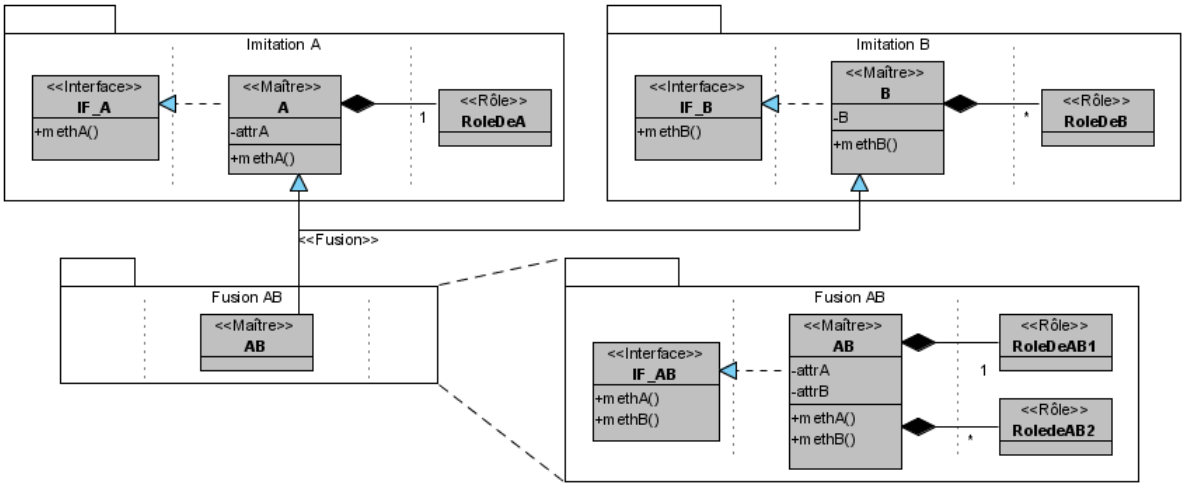


Figure 5.6 : Principes d'utilisation de l'opérateur de fusion.

2.4.b Application au cas d'étude

Dans le cas d'étude, une imitation du patron « Mémoire d'évènement » (composant *Capteur*) est composée avec l'imitation du patron « Observateur » (composant *Détecteur*). Cette composition, qui consiste en fait en la fusion des deux classes *Capteur,* est présentée dans la figure 5.7. Lors de cette fusion, les méthodes *monitorer* et *mesurer* sont apparues comme assimilables, c'est pourquoi seule *monitorer* a été conservée. Par contre *ajouterMesure* a été conservée puisqu'elle devrait être utilisée au sein de *monitorer*.

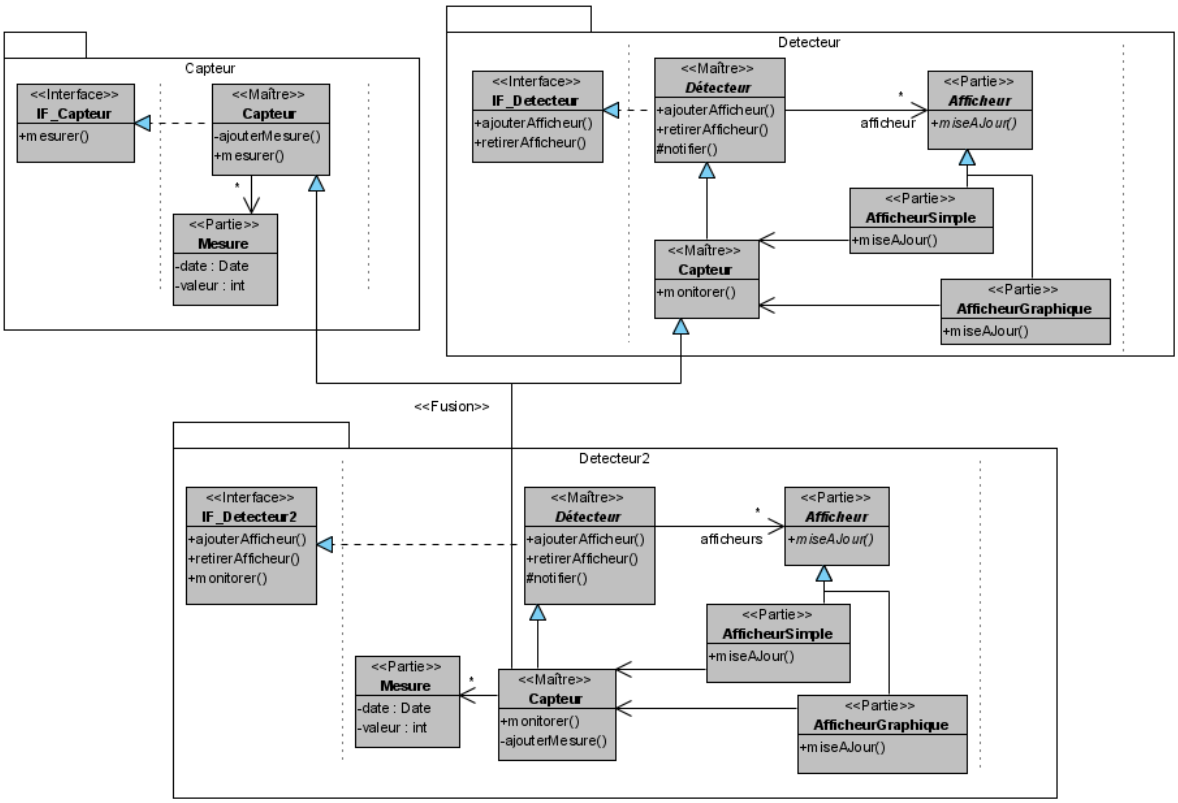


Figure 5.7: Fusion entre les imitations des patrons « Mémoire d'événement » et « Observateur ».

2.5 Vues d'un système

La composition des imitations doit conserver les propriétés génériques et plus généralement permettre la traçabilité de la construction du système cible. Pour ce faire nous proposons de garder les composants opérandes de chaque composition. Ainsi, les imitations et fusions intermédiaires sont conservées. La figure 5.8 donne un aperçu du système cible sous la forme d'un diagramme de composants, illustrant toutes les compositions effectuées dans les exemples ci-dessus. Les imitations originales sont grisées.

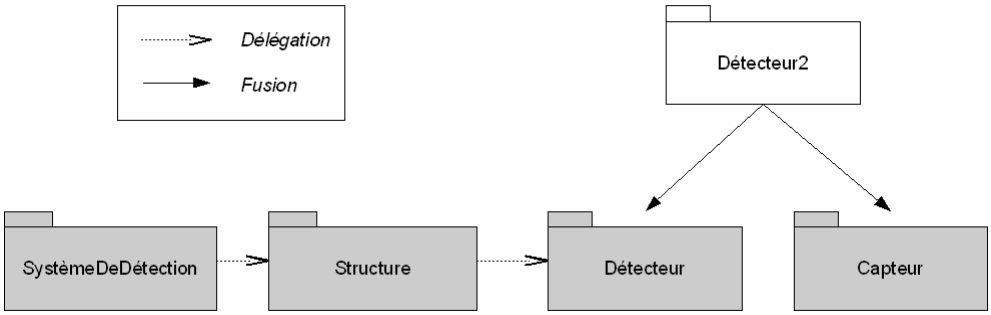


Figure 5.8 : Vue globale d'un système composé.

3 SYNTHÈSE

Ce chapitre a présenté une technique de compositions d'imitations basée sur l'utilisation de conteneurs de type « composant » où deux opérateurs sont dégagés. L'opérateur de délégation s'appuie sur le concept de classes « Rôle » de la méthode Symphony tandis que l'opérateur de fusion est spécifiquement introduit.

Dans le modèle de composants Symphony, une classe « Rôle » ne peut utiliser que l'interface d'un composant, c'est à dire d'une classe « Maître ». Or, certaines compositions par délégation nécessite de relier une classe « Rôle » à une classe « Partie ». Dans le cas d'étude, par exemple, les afficheurs entretiennent une relation avec la classe *Mesure* qui, dans le composant *Capteur,* est une classe « Partie ». De nouveaux mécanismes sont donc à prévoir pour permettre ce type de délégation.

Conclusion générale et Perspectives

Les patrons d'ingénierie ont été introduits afin de capitaliser des savoirs et savoir-faire de façon à en faciliter la réutilisation. Un patron est représenté sous la forme d'un ensemble de rubriques qui décrivent un problème identifié dans un contexte, ainsi qu'une solution pour le résoudre. Dans l'ingénierie logicielle, et plus particulièrement dans le développement des systèmes d'information, l'utilisation de patrons d'analyse et de conception est reconnue comme un gage de qualité des spécifications. Nous appelons *imitation* le fait qu'un ingénieur d'applications réutilise un patron, c'est également le nom de l'artefact résultant. Cet artefact est un fragment de spécification généralement orientée-objet, que l'ingénieur d'applications adapte et intègre à la description de son système.

Cette adaptation de la solution proposée peut être source de problèmes ou d'interrogations. Nous avons dégagé, à partir du cas d'étude, trois axes sur lesquels les solutions de patrons orientés-objet peuvent être améliorées de façon à faciliter leur réutilisation : l'axe de la complétude, l'axe de la variabilité et celui de la généricité. La problématique de la composition des imitations, également illustrée dans le cas d'étude, ne fait pas partie des thématiques principales de ce travail, mais des premières réflexions sont néanmoins abordées.

Si le problème de généricité a trouvé un écho dans différents travaux existant traitant de la conception à l'aide de patrons, la complétude et surtout la variabilité sont nettement plus en retrait. L'approche que nous proposons dans cette thèse permet aux ingénieurs de patrons d'augmenter les spécifications de patrons selon les trois axes ci-dessus. D'autre part elle permet aux ingénieurs d'applications de tirer partie de ses spécifications lors de l'imitation.

1 Rappel des propositions

La solution d'un patron orienté-objet se limite souvent à un diagramme de classes plus ou moins bien commenté par d'autres rubriques. La complétude des solutions orientées-objet est, pour

nous, un facteur déterminant pour la compréhension et l'intérêt de l'imitation. En effet, l'utilisation de plusieurs vues (fonctionnelle, dynamique et statique) sur la solution permet 1) une meilleure appréhension de l'essence de la solution ; 2) d'imiter, en plus de la structure statique, le comportement qu'elle réalise. Nous proposons de décrire la solution d'un patron avec UML 2, sous la forme d'un mini-système composé d'une vue fonctionnelle (diagramme de cas d'utilisation), d'une vue dynamique (diagramme de séquences) et d'une vue statique (diagramme de classes).

La plupart des patrons comporte des variantes que l'ingénieur d'applications pourrait vouloir mettre en œuvre, comme par exemple la notification implicite ou explicite du patron « Observateur ». Malheureusement, ces variantes sont décrites textuellement et leur impact sur la solution proposée peut rendre leur application difficile à appréhender. La variabilité est une problématique transversale en génie logiciel et la littérature nous fournit une taxonomie riche sur les types de variabilité. Même si seules les notions de variante alternative et optionnelle sont considérées dans ce travail sur les patrons, nous proposons un opérateur générique de variabilité (permettant de prendre en compte tout type de variabilité), que nous utilisons pour représenter la variabilité au sein de la vue fonctionnelle du mini-système.

C'est à partir de cette vue fonctionnelle « variable » que l'ingénieur de patrons organise les impacts des variantes dans les deux autres vues. Pour ce faire, nous établissons un ensemble de règles de construction qui régit, pour tout cas d'utilisation de la vue fonctionnelle, la définition d'un fragment dynamique et d'un fragment statique regroupant les informations nécessaires à sa mise en œuvre. Les ingénieurs de patrons peuvent ainsi construire des mini-systèmes « variables » grâce à une extension du méta-modèle d'UML.

De nombreuses approches existantes permettent d'exprimer des propriétés génériques qui représentent *l'essence* ou encore le *leitmotiv* du patron, garantes par exemple de la relation d'égalité (nommée *dimension)* entre nombre de produits et nombre de méthodes de fabrication du patron « Fabrique Abstraite », que toute imitation de ce patron doit respecter. Nous proposons dans cette thèse que le méta-modèle définit pour la variabilité prenne également en compte ces propriétés au sein de la vue statique et des éléments imitables (classes, associations, méthodes, attributs) qui la composent. Ces propriétés donnent lieu à des contraintes OCL, interprétées via le méta-modèle, afin de garantir la validité des imitations.

Notre approche donne également la possibilité aux ingénieurs de patrons d'apposer des notes « A Faire » comprenant des recommandations textuelles. Ces notes servent à exprimer des propriétés très génériques, souvent plus en relation avec une chose à faire (implémentation d'opérations, ajout d'attributs,...) qu'avec une propriété essentielle du patron.

Afin de formaliser la mise en œuvre de ces propositions par les ingénieurs de patrons, un processus de spécification des solutions de patrons a été proposé et finalise les propositions faites dans la première partie de cette thèse, propositions destinées aux ingénieurs de patrons.

La deuxième partie de ce travail est destinée aux ingénieurs d'applications et consiste en un processus de réutilisation des spécifications produites par les ingénieurs de patrons. Ce processus

est composé de deux sous-processus. Le premier sous-processus, nommé processus de réduction; débute par le choix des variantes sur la vue fonctionnelle suivi de la réduction des deux autres vues, pour laquelle nous donnons des règles de réalisation. Le second sous-processus, appelé processus d'application, exploite les propriétés génériques lors de l'adaptation et la composition de l'imitation dans le système cible. Enfin, nous définissons l'architecture cadre d'un prototype automatisant les sous-processus de réduction et d'application.

La composition des imitations entre elles ou avec des spécifications originales ne fait pas partie des problématiques principales de cette thèse. Toutefois nous avons dégagé des règles générales et une première étude propose qu'après avoir été encapsulées dans des composants métiers, les imitations soient composées grâce à deux opérateurs : délégation et fusion.

2 PERSPECTIVES

Nos propositions pour la spécification des solutions de patrons peuvent être améliorées à plusieurs niveaux.

Dans notre approche, la vue fonctionnelle sert à guider la spécification de la variabilité et à l'initialisation du processus d'imitation. Cependant, la vue fonctionnelle, ne fait pas l'objet d'une attention particulière après la réduction. Nous estimons qu'il serait pertinent de définir un vrai statut pour la vue fonctionnelle imitée, afin de rendre plus lisible encore l'imitation. Il s'agit principalement de savoir si on ne peut pas également définir des propriétés génériques au niveau des cas d'utilisation, notamment en termes d'occurrences d'imitations.

Les notes « A Faire » permettent d'attirer l'attention de l'ingénieur d'applications sur des aspects qui sont trop dépendants du contexte d'imitation pour figurer plus formellement dans la solution. La sémantique de ces notes est assez pauvre, mais il est probable que beaucoup de solutions comportent des notes similaires (comme par exemple « implémenter la méthode X »). Il serait alors intéressant d'élaborer une taxonomie des notes « A Faire » de manière à mieux encadrer le contenu et l'utilisation de ces notes.

Le mécanisme qui gère les propriétés génériques a été pensé pour être évolutif et complété par de nouvelles propriétés. Dans (O`Cinneide, 2001) (cf. chapitre 2, section 4.4), le processus de définition permet aux ingénieurs de patrons d'ajouter des mini-transformations si aucune existante ne satisfaisait leur besoin. Un concept similaire d'enrichissement de la base des propriétés génériques doit être envisagé afin que les ingénieurs de patrons puissent eux-mêmes proposer sinon créer leurs propres propriétés ainsi que les contraintes d'imitation qu'elles impliquent. Cette perspective rendrait le méta-modèle évolutif et inscrirait encore plus l'approche dans le monde de l'ingénierie dirigée par les modèles.

Le processus d'imitation peut également bénéficier d'améliorations en particulier en ce qui concerne le sous-processus d'application qui ne se focalise pour l'instant que sur les aspects

statiques de l'imitation. S'il est aisé de générer la vue dynamique de la première version du modèle imité, les conséquences de l'adaptation et de la composition sur ses aspects comportementaux n'ont pas encore été étudiées.

Nous l'avons vu, la vue fonctionnelle à variantes pilote tout le processus de spécification d'un nouveau patron. Il est d'ailleurs possible, à partir de cette vue, de générer le squelette des deux autres. De plus, la notation simplifiée que nous avons présentée pour cette vue est très aisément réalisable à partir d'un profil UML comportant les stéréotypes « variation », « variante » pour les cas d'utilisation et « alternative », « option » pour les dépendances. Un modèle UML exprimé à l'aide de ce profil pourrait aisément être transformé en un modèle UML-pi incomplet mais prêt à recevoir les informations dynamiques et statiques manquantes. Le support du processus de spécification par notre prototype est d'ores et déjà à l'étude.

Nous dégageons également des perspectives plus générales, pouvant donner lieu à de nouveaux travaux de recherche.

Comme nous l'avons vu lors de l'application de nos propositions concernant la spécification du patron « Composite » (cf. chapitre 3, section 5), la question de la dépendance entre variantes mériterait d'être approfondie. La plupart des approches pour les lignes de produits incluent la notion d'exclusion entre variantes, mais d'autres cas sont envisageables (implication, préférence,...). Ces aspects sont d'ailleurs à l'étude dans des travaux de thèse menés au sein de l'équipe SIGMA, autour de la variabilité dans les composants métier. Un premier travail a d'ailleurs montré l'applicabilité de nos propositions en termes de variabilité et de généricité à ce domaine (Saidi et al., 2007) et démontre, outre la nécessité d'affiner les relations de variabilité, des besoins forts en termes de formalisation de la composition.

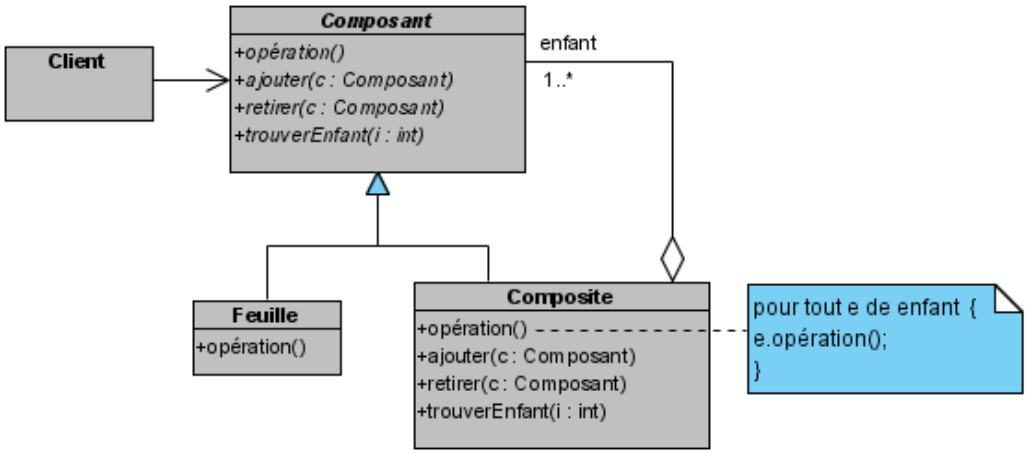
L'expression de la variabilité fonctionnelle dans les patrons présente des avantages indéniables. Cependant, elle est à utiliser avec parcimonie, s'agissant notamment des variations de type « alternative ». En effet, avec notre approche, un ingénieur de patrons pourrait être tenté de regrouper dans le même patron deux solutions très distinctes, résolvant des problèmes qui auraient habituellement donné lieu à deux patrons aux intentions et motivations bien différentes. Ainsi, une étude approfondie sur la granularité des patrons et sur leur organisation doit être menée afin d'appréhender l'impact de la variabilité sur la définition des patrons et sur leurs usages.

Annexes

1 Patron Composite....140
2 Patron Méthode de Fabrication....142
3 Patron Singleton....144
4 Patron Fabrique Abstraite....145
5 Méta-modèle UMLpi....148

1 Patron Composite

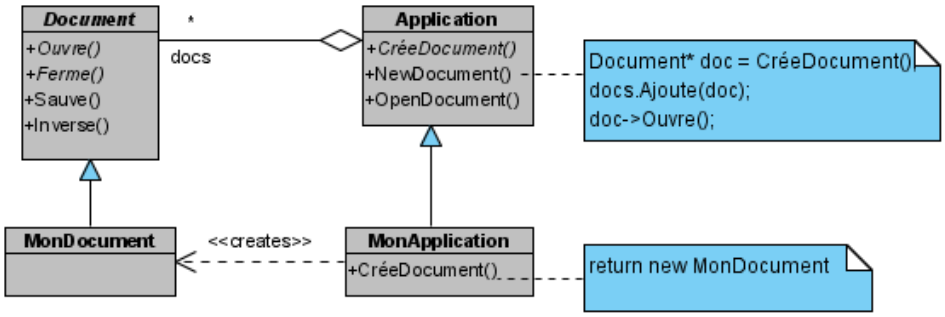
Extrait du patron « Composite » (Gamma et al., 1995)

INTENTION	Le modèle Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même et unique façon les objets individuels et les combinaisons de ceux-ci.
MOTIVATION	Les applications graphiques telles que éditeurs de dessin et systèmes de représentations schématiques, permettent à l'utilisateur de construire des diagrammes complexes à partir de composants simples. L'utilisateur peut groupes les composants pour en former de plus grands, qui, à leur tour, pourront être groupés pour former des composants encore plus grands. Une implémentation simple peut définir des classes de primitives graphiques telles que Textes et Lignes, assorties, de plus, de quelques autres classes, jouant le rôle de container pour ces primitives. Il y a cependant une difficulté avec cette approche : le code qui manipule ces classes doit traiter différemment les objets primitives et ceux containers, même si, la plupart du temps, les utilisateurs les traitent de la même façon. Le fait d'avoir à faire cette distinction entre objet complique l'application. Le modèle Composite décrit l'utilisation de la composition récursive, qui dispense les clients d'avoir à faire cette distinction. [...]
INDICATION D'UTILISATION	On utilisera le modèle Composite lorsque • On souhaite représenter des hiérarchies de l'individu à l'ensemble. • On souhaite que le client n'ait pas à se préoccuper de la différence entre combinaisons d'objets et objets individuels. Les clients pourront traiter de façon uniforme tous les objets de la structure composite.
STRUCTURE	
CONSTITUANTS	• **Composant** – Le Composant déclare l'interface des objets entrant dans la composition. – Il implémente le comportement par défaut qui convient pour l'interface commune à toutes les classes. – Il déclare une interface pour accéder à ses composants enfants et les gérer.

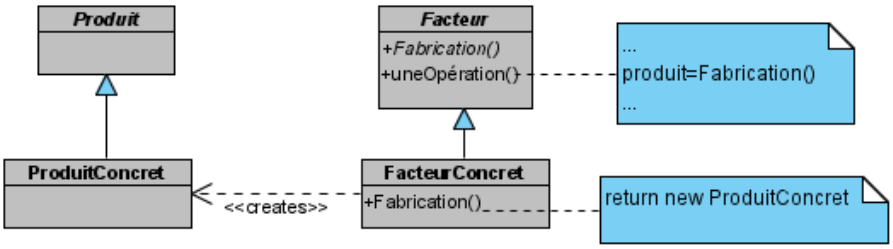
	– Éventuellement, il définit une interface pour accéder à un parent de composant dans une structure récursive, et l'implémente si besoin est. • **Feuille** – La feuille représente des objets feuille dans la composition. Une feuille n'a pas d'enfants. – Elle définit le comportement d'objets primitives dans la composition. • **Composite** – Le Composite définit le comportement des composants dotés d'enfants. – Il stocke les composants enfants. – Il implémente les opérations liées aux enfants dans l'interface Composant. • **Client** – Le Client manipule les objets de la composition à l'aide de l'interface Composant.
COLLABORATION	Les clients utilisent l'interface de la classe Composant pour manipuler les objets de la structure composite. Si l'objet manipulé est une feuille, la requête est traitée directement. Si c'est un Composite, il transfère généralement cette requête à ses composants, en effectuant éventuellement des opérations supplémentaires avant et/ou après ce transfert.
IMPLÉMENTATION	[...] *Partage des composant*s. Il est souvent utile de partager les composants, par exemple pour réduire les besoins de stockage mémoire. Mais si un composant ne peut avoir plus d'un parent, ce partage de composants devient difficile. Une solution possible consiste, pour un enfant, à stoker les références de plusieurs parents. Mais cela peut conduire à des ambiguïtés lorsqu'une requête remonte dans la structure. [...] *Déclaration des opération de gestions des enfants*. Bien que la classe Composite implémente les opérations Ajouter et Supprimer de gestion des enfants, un point important du modèle Composite est la détermination de celles des classes qui déclarent ces opérations dans la hiérarchie de la classe Composite. Faut-il déclarer ces opérations dans Composant, et leur donner une signification pour les classes Feuille, ou bien faut-il les déclarer et les définir seulement dans Composite et ses sous-classes. La décision implique un compromis entre sécurité et transparence : • Définir l'interface de gestion des enfants dans la racine de la hierarchie de classes procure de la transparence car on peut alors traiter uniformément tous les composants. Cela pénalise cependant la sécurité, car les clients peuvent tenter de faire des actions sans signification telles que Ajouter des objets à des feuilles ou leur en supprimer. • Définir la gestion des enfants dans la classe Composite procure de la sécurité, puisque toute tentative d'ajout ou de retrait d'objets des feuilles sera réfutée à la compilation. Mais on perd en transparence, du fait que les feuilles et composites ont des interfaces différentes. Dans ce modèle, la transparence a été favorisée par rapport à la sécurité. [...]

2 Patron Méthode de Fabrication

Ce patron est aussi appelé « Fabrication ».
Extrait du patron «Fabrication» (Gamma et al., 1995)

INTENTION	Définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier. La Fabrication permet à une classe de déléguer l'instanciation à des sous-classes.
MOTIVATION	Les Frameworks utilisent des classes abstraites pour définir et gérer les relations entre les objets. Un framework est souvent également chargé des la création de ces objets. Considérons un framework conçu pour des applications susceptibles de présenter plusieurs documents à l'utilisateur. Ce framework possède deux entités fondamentales qui sont les classes Application et Document. Ces deux classes sont abstraites, et les clients doivent en dérive des sous-classes pour réaliser l'implémentation spécifique de leur application. Pour créer, par exemple, une application de dessin, on définira les classes ApplicationDessin et DocumentDessin. La classe Application a la charge de gérer les Documents, et les créera selon le besoin – lorsque l'utilisateur sélectionne Open ou New dans un menu, par exemple. Toute sous-classe particulière de Document destinée à être instanciée, est spécifique de l'application. De ce fait, la classe Application ne peut prévoir la sous-classe Document qu'il cnvient d'instancier – la classe Application sait seulement qu'un nouveau document doit être créé, mais ignore le type de ce dernier. Ily a donc dilemme : le framework doit instancier des classes, mais il ne connaît que des classes abstraites, qui ne peuvent être instanciées. Le modèle de Fabrication offre une solution. Il encapsule la connaissance du type de la sous-classe Document à créer, et diffuse cette information hors du framework. 
INDICATION D'UTILISATION	La Fabrication doit être utilisée lorsque : • Une classe ne peut prévoir la classe des objets qu'elle aura à créer. • Une classe attend de ses sous-classes qu'elles spécifient les objets qu'elles créent. • Les classent délèguent les responsabilités à une de leurs nombreuses sous-classes assistantes, et l'on veut disposer localement de l'information permettant de connaître la sous-classe assistante qui a reçu cette délégation.

STRUCTURE



CONSTITUANTS

- **Produits** (Document)
  - Le Produit définit l'interface des objets créés par la fabrication.
- **ProduitConcret** (MonDocument)
  - Le ProduitConcret implémente l'interface du Produit.
- **Facteur** (Application)
  - Le Facteur déclare la fabrication ; celle-ci renvoie un objet de type Produit. Le Facteur peut également définir une implémentation par défaut de la fabrication, qui renvoie un objet ProduitConcret par défaut.
  - Il peut appeler la fabrication pour créer un objet Produit.
- **FacteurConcret** (MonApplication)
  - Le FacteurConcret surcharge la fabrication pour renvoyer une instance d'un ProduitConcret.

COLLABORATION

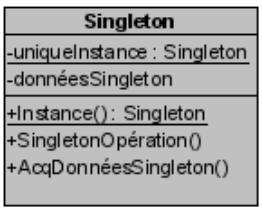
Le Facteur confie à ses sous-classes la définition de la fabrication de sorte qu'il renvoie une instance du ProduitConcret approprié

IMPLÉMENTATION

[...]
Fabrication paramétrée. Une autre variante du modèle permet à la fabrication de créer de nombreuses espèces de produits. La fabrication utilise un paramètre qui identifie la variété d'objets à créer. Tous les objets créés par la fabrication partageront l'interface Produit. [...]
[...]

3 PATRON SINGLETON

Extrait du patron «Singleton» (Gamma et al., 1995)

INTENTION	Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès de type global à cette classe.
MOTIVATION	Il est important pour certaines classes de n'avoir qu'une seule instance. Alors qu'il peut y avoir de nombreuses imprimantes dans un système, il ne peut y avoir qu'un serveur d'impression. Il ne doit y avoir qu'un fichier système et qu'un seul gestionnaire de fenêtres. Un filtre binaire ne peut avoir qu'un seul convertisseur A/D. Un système de comptabilité sera dédié au service d'une seule entreprise. Comment assurer qu'une classe n'a qu'une instance, et que cette dernière est facilement accessible ? Une variable globale permet d'accéder à un objet, mais elle n'empêche pas des instanciations multiples de cet objet. Une meilleure solution consiste à confier à la classe elle-même la responsabilité d'assurer l'unicité de son instance. La classe peut assurer qu'aucune autre instance ne sera créée (en interceptant les requêtes demandant à créer de nouveaux objets), et elle peut fournir un moyen pour accéder à cette instance. C'est le Singleton.
INDICATION D'UTILISATION	On utilisera le modèle Singleton : • S'il doit n'y avoir exactement qu'une instance d'une classe, qui, de plus, doit être accessible aux clients en un point bien déterminé. • Si l'instance unique doit être extensible par dérivation en sous-classe, et si l'utilisation d'une instance étendue doit être permise aux clients, sans qu'ils aient à modifier leur code.
STRUCTURE	
CONSTITUANTS	• **Singleton** – Le Singleton définit une opération Instance qui donne au client l'accès à son unique instance. Instance est une opération (c'est-à-dire une méthode de classe en Smalltalk, et une fonction membre statique en C++). – Il peut avoir la charge de créer sa propre instance unique.
COLLABORATION	Les clients accèdent à l'instance d'un Singleton par le seul intermédiaire de l'opération Instance de Singleton.

4 Patron Fabrique Abstraite

Extrait du patron « Fabrique Abstraite » (Gamma et al., 1995)

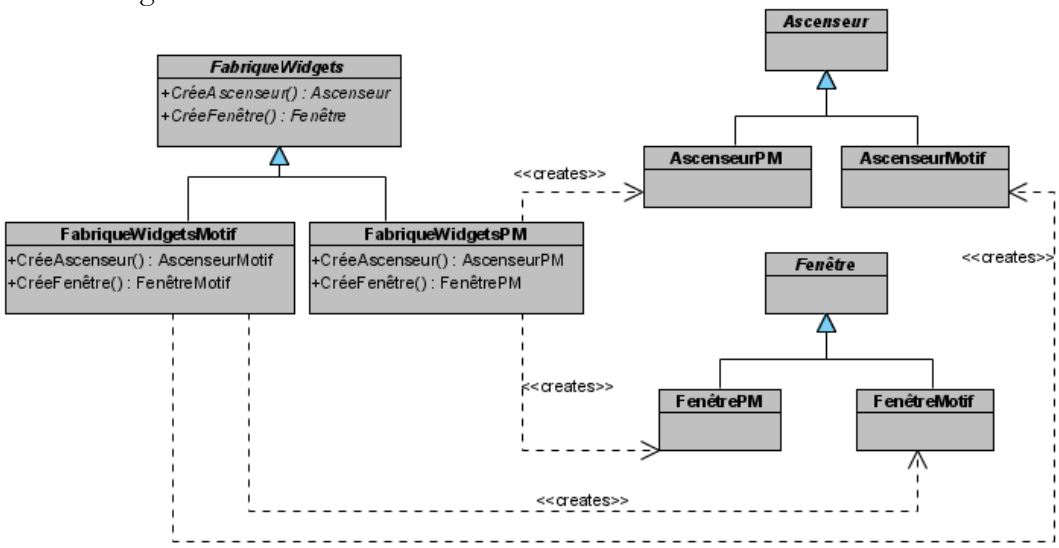
INTENTION

La Fabrique Abstraite fournit une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes.

MOTIVATION

Considérons une boite à outils d'interfaces utilisateurs, appareillant plusieurs « look-and-feel » (décors) standards, tels que Motifs et Presentation Manager (PM). Des décors différents instaurent des aspects et des comportements spécifiques des « widgets » de l'interface utilisateur, que sont les Ascenseurs de défilement, les fenêtres et les boutons. Pour qu'une application soit portable d'un décor standard à l'autre, il ne faut pas que ses widgets soient « codés en dur » pour un décor donné. Instancier des classes de widgets spécifiques d'un décor dans toute une application, rend difficile un changement ultérieur de décor.

Cette difficulté peut être levée par la création d'une classe de base abstraite FabriqueWidgets qui définit une interface pour créer chaque variété de widget. Il faut également, pour chaque variété de widget, créer une classe abstraite dont les sous-classes concrètes implémenteront les versions de ces widgets conformément aux standards de décor supportés. L'interface de classe FabriqueWidgets possède, pour chaque classe abstraite de widgets, une opération qui renvoie un nouvel objet widget. Les clients appelleront ces opérations pour obtenir des instances de widgets, mais sans savoir quelles classes concrètes ils utilisent. Ainsi, le client reste-t-il indépendant du décor en vigueur.



Il y a une sous-classe concrète de FabriqueWidgets pour chaque décor standard. Chaque sous-classe implémente les opérations de création du widget dans la présentation correspondant au décor considéré. Par exemple, l'opération CreeAscenseur et FabricationWidgetsMotif instancie et renvoie Ascenseur de défilement Motif, tandis que la même opération de FabriqueWidgetPM instancie et renvoie un Ascenseur de défilement pour Presentaion Manager. Les clients créent les widgets exclusivement par l'interface FabriqueWidgets, et ignorent tout des classes qui implémentent les widgets pour un décor particulier. Autrement dit, les clients n'ont qu'à se fier à une interface définie par une classe abstraite, et non par une classe concrète particulière.

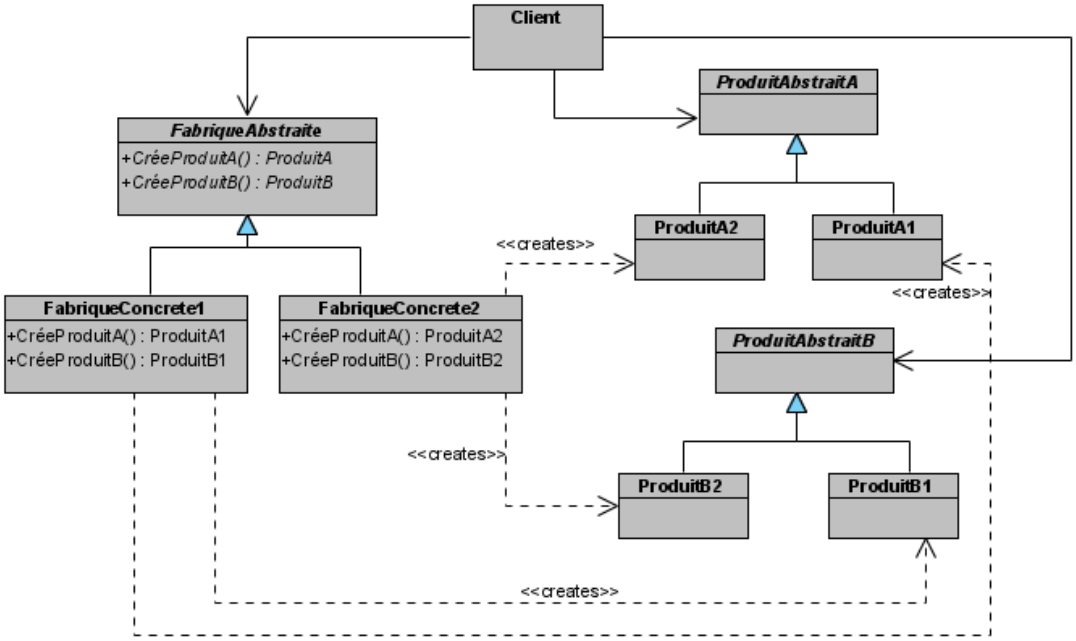
Une FabriqueWidgets renforce également l'interdépendance des classes widgets concrètes. Un Ascenseur Motif doit être utilisé avec un bouton Motif et un éditeur de texte Motif. Cette contrainte est automatiquement satisfaite du fait de l'utilisation de FabriqueWidgetsMotif.

INDICATION D'UTILISATION

L'utilisation du modèle Fabrique Abstraite est recommandée lorsque :

- Un système doit être indépendant de la façon dont ses produits ont été crées, combinés et représentés.
- Un système doit être constitué à partir d'une famille de produit, parmi plusieurs.
- On souhaite renforcer le caractère de communauté d'une famille d'objets produits conçus pour être utilisés ensemble.
- On souhaite fabriquer une bibliothèque de classes produits, en n'en révélant que l'interface et non l'implémentation.

STRUCTURE

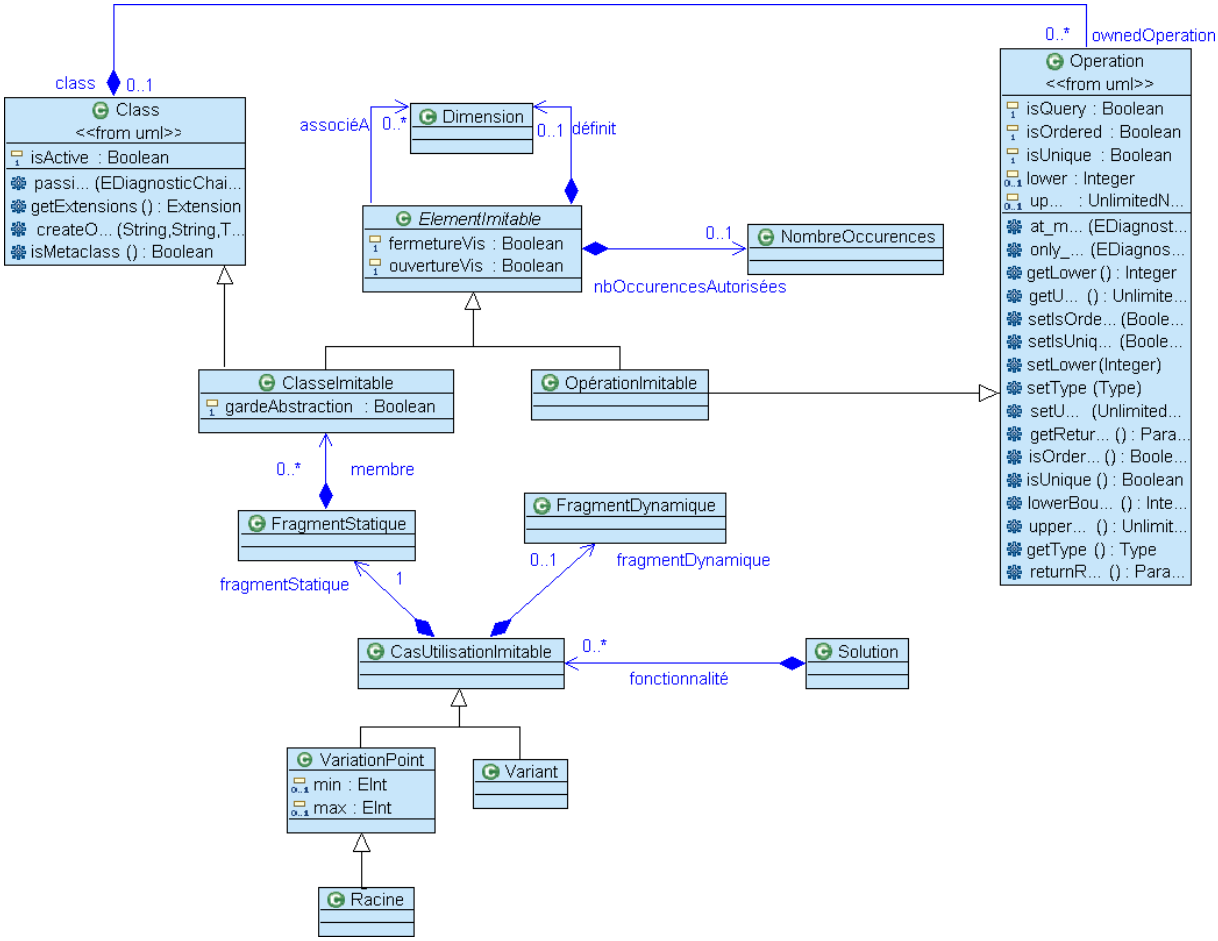


CONSTITUANTS

- **FabriqueAbstraite** (FabriqueWidgets)
  - FabriqueAbstraite déclare une interface contenant les opérations de création d'objets produits abstraits.
- **FabriqueConcrete** (FabriqueWidgetsMotif, FabriqueWidgetsPM)
  - FabriqueConcrete implémente les opérations de création d'objets produits concrets.
- **ProduitAbstrait** (Fenetre, Ascenseur)
  - ProduitAbstrait déclare une interface pour un type d'objet produit.
- **ProduitConcret** (Fenetremotif, AscenseurMotif)
  - ProduitConcret définit un objet produit qui doit être créé par la fabrique concrète correspondante.
  - Implémente l'interface de ProduitAbstrait.
- **Client**
  - Il n'utilisera que les interfaces déclarées par les classes FabriqueAbstraite et ProduitAbstrait.

COLLABORATION	Normalement, une instance unique de la classe FabriqueConcrete est créée à l'exécution. Cette fabrique concrète crée des objets produits, dotés d'une implémentation spécifique. Pour créer des objets produits différents, les clients doivent utiliser une fabrique concrète différente. Une FabriqueAbstraite délègue la création des objets produits à sa sous-classe FabriqueConcrete.

5 Méta-modèle UMLpi



BIBLIOGRAPHIE

ACTOLL, 2008 SOCIÉTÉ ACTOLL : *Site web*, http://www.actoll.com. Dernière visite : Juillet 2008.

AJAXPATTERNS, 2007 AJAX PATTERNS : *Main Page*, http://ajaxpatterns.org. Dernière visite : Octobre 2007.

ALBIN-AMIOT ET AL., 2002 H. ALBIN-AMIOT, P. COINTE, Y. GUÉHÉNEUC (2002) : *Un méta-modèle pour coupler application de patrons et detection de design patterns*.

ALBIN-AMIOT ET GUÉHÉNEUC, 2001 H. ALBIN-AMIOT, Y. GUÉHÉNEUC (2001) : *Meta-modeling DesignPatterns: application to pattern detection and code synthesis*. Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods.

ALEXANDER ET AL., 1977 C. ALEXANDER, S. ISHIKAWA, M. SILVERSTEIN (1977) : *A Pattern Language : Towns, Buildings, Construction*. Oxford University Press.

ALEXANDER, 1979 C. ALEXANDER (1979) : *The Timeless Way of Building*. Oxford University Press.

AMBLER, 1998 S. W. AMBLER (1998) : *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press.

AMBLER, 1999 S. W. AMBLER (1999) : *More Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press.

ARNAUD ET AL., 2004 N. ARNAUD, A. FRONT, D. RIEU (2004) : *Une approche par méta-modélisation pour l'imitation*. INFORSID'04, p479-494.

ARNAUD ET AL., 2005 N. ARNAUD, D. RIEU, A. FRONT (2005) : *Deux opérations pour l'intégration d'imitations de patrons*. INFORSID'05.

BACHMANN ET BASS, 2001 F. BACHMANN, L. BASS (2001) : *Managing variability in software architecture* . Software Engineering Notes, Vol.*26*, n°3, p.126-132.

BECK ET CUNNINGHAM, 1987 K. BECK, W. CUNNINGHAM (1987) : *Using Pattern Languages for Object-Oriented Program*.

BLAZY ET AL., 2003 S. BLAZY, F. GERVAIS, R. LALEAU (2003) : *Reuse of Specification Patterns with the B Method*. ZB 2003 : Formal Specification and Development in Z and B, p40-57.

BOOCH ET AL., 2005 G. BOOCH, J. RUMBAUGH, I. JACOBSON (2005) : *The Unified Modeling Language Reference Manual (2nd Edition)* .

BORCHERS, 2000 J. O. BORCHERS (2000) : *A Pattern Approach to Interaction Design*. Symposium on Designing Interactive Systems, p369-378.

BORNE ET REVAULT, 1999 I. BORNE, N. REVAULT (1999) : *Comparaison d'outils d e mise en œuvre de design patterns*. L'objet, Vol.*5*, n°2/1999, p.243 à 266.

Buschmann et al., 1996 F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal (1996) : *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley & Sons.

Clauss, 2001 M. Clauss (2001) : *Generic Modeling using UML extensions for variability*. Workshop on Domain-specific Visual Languages, OOPSLA 2001, p11-18.

Coad, 1992 P. Coad (1992) : *Object-oriented patterns*. Communication of the ACM, Vol.*35*, n°9, p.152-159.

Coad, 1995 P. Coad (1995) : *Object Models : Strategies, Patterns and Applications*. Yourdon Press Computing Series.

Conte et al., 2001 A. Conte, M. Fredj, J. Giraudin, D. Rieu (2001) : *P-SIGMA : un formalisme pour une représentation unifiée de patrons*. INFORSID'01, p67-86.

Conte et al., 2001b A. Conte, I. Hassine, J. Giraudin, D. Rieu (2001) : *AGAP : un Atelier de Gestion et d'Application de Patrons.*. INFORSID'01, p142-159.

Coplien et al., 1995 J. O. Coplien, D. C. Schmidt (1996) : *Pattern Languages of Program Design*. Addison-Wesley.

Coplien et al., 1996 J. O. Coplien, J. M. Vlissides, N. L. Kerth (1996) : *Pattern Languages of Program Design 2*. Addison-Wesley.

Coplien, 1992 J. O. Coplien (1992) : *Advanced C++ Programming Styles and idioms*. Addison-Wesley.

Czarnecki et Eisenecker, 2000 K. Czarnecki, U. W. Eisenecker (2000) : *Generative Programming – Methods, Tools and Applications*.

Eclipse, 2008 The Eclipse Foundation : *Site web de la plateforme Eclipse*, http://www.eclipse.org/. Dernière visite : Août 2008.

Eden, 2000 A. H. Eden (2000) : *Precise Specification of Design Patterns and Tool Support in Their Application*. Thèse de doctorat, Tel Aviv University.

EMF, 2008 The Eclipse Foundation : *Site web du projet EMF*, http://www.eclipse.org/modeling/emf/. Dernière visite : Août 2008.

Favre et al., 2006 Ouvrage collectif sous la direction de J. M. Favre, J. Estublier, M. Blay-Fornarino (2006) : *L'ingénierie dirigée par les modèles, au-delà du MDA.*. Lavoisier.

Florijn et al., 1997 G. Florijn, M. Meijers, van Winsen Pieter (1997) : *Tool support for objectoriented patterns*. Lecture Notes in Computer Science, Vol.*1241*, n°, p.472-495.

Fowler, 1997 M. Fowler (1997) : *Analysis Patterns: Reusable Object Models*.

France et al., 2003 R. B. France, S. Ghosh, E. Song, D. Kim (2003) : *A Metamodeling Approachto Pattern-based Model Refactoring*. Special Issue on Model DrivenDevelopment.

France et al., 2004 R. B. France, D. Kim, S. Ghosh, E. Song (2004) : *A UML-Based Pattern Specification Technique*. IEEE Transactions on Software Engineering, Vol.30, n °3, p.193-206.

Gamma et al., 1995 E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995) : *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GMF, 2008 The Eclipse Foundation : *Site web du projet GMF*, http://www.eclipse.org/modeling/gmf/. Dernière visite : Aoüt 2008.

Gzara, 2000 L. Gzara, D. Rieu, M. Tollenaere (2000) : *Patterns Approach to Product Information Systems Engineering*. Requierement Engineering, Vol., n°3, p.157-179.

Hachani, 2006 O. Hachani (2006) : *Patrons de conception à base d'aspects pour l'ingénierie des systèmes d'information par réutilisation*.

Hassine et al., 2002 I. Hassine, D. Rieu, F. Bounaas, O. Seghrnouchni (2002) : *Symphony : Un modèle conceptuel de composants métier*. INFORSID'02.

Hassine, 2005 I. Hassine (2005) : *Spécification et formalisation des démarches de développement à base de composants métier : la démarche Symphony*.

Jézéquel et Ziadi, 2005 J. M. Jézéquel, T. Ziadi (2005) : *Manipulation de lignes de produits logiciels : une approche dirigée par les modèles*.. Ingénierie Dirigée par les Modèles (IDM'05).

Kang et al., 1990 K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson (1990) : *Feature-Oriented Domain Analysis (FODA) feasibility study, Technical report CMU/SEI-90-TR-21*.

Kermeta, 2008 Equipe Triskell : *Si web du projet Kermeta*, http://www.kermeta.org/. Dernière visite : Août 2008.

Konrad et Cheng, 2005 S. Konrad, B. H. C. Cheng (2005) : *Real-time specification patterns*. ICSE, p372-381.

LePUS, 2008 *LePUS3 and Class-Z*, http://lepus.org.uk/. Dernière visite : Juillet 2008.

M2M, 2008 The Eclipse Foundation : *Site du projet M2M*, http://www.eclipse.org/m2m/. Dernière visite : Août 2008.

MagicDraw, 2008 NoMagic : *Site web de MagicDraw*, http://www.magicdraw.com/. Dernière visite : Août 2008.

Maplesden et al., 2001 D. Maplesden, J. Hosking, J. Grundy (2001) : *A Visual Language for DesignPattern Modelling and Instantiation*. Proceedings of the IEEE Symposia on Human-CentricComputing Languages and Environments.

Maplesden et al., 2002 D. Maplesden, J. Hosking, J. Grundy (2002) : *Design Pattern Modelling andInstantiation using DPML*. Proceedings of 14th International Conference on Technology ofObject-Oriented Languages and Systems.

Maracano-Kamenoff et al., 2000
R. Marcano-Kamenoff, Lévy Nicole, Francisca Losavio (2000) : *Spécification et spécialisation de Patterns en UML et B*.. LMO'00, p245-260.

MDT-UML2, 2008 The Eclipse Foundation : *Wiki du projet MDT-UML2*, http://www.eclipse.org/modeling/mdt/?project=uml2. Dernière visite : Août 2008.

Meijler et al, 1997 T. D. Meijler, S. Demeyer, R. Engel (1997) : *Making design patterns explicit in Face*. European Software Engineering Conference.

Michael Mahemoff, 2006 M. Mahemoff (2006) : *Ajax Design Patterns* . O'Reilly.

**O`Cinneide, 2001** M. O`Cinneide (2001) : *Automated Application of Design Patterns: a Refactoring Approach*.

OMGa, 2005 Object Management Group (2005) : *Unified Modeling Language v2.0 : Superstructure*.

OMGb, 2005 Object Management Group (2005) : *Meta Object Facility (MOF) Spécification*.

OMGc, 2005 Object Management Group (2005) : *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*.

Pagel et Winter, 1996 B. Pagel, M. Winter (1996) : *Towards Pattern-Based Tools*. EuroPLoP'96, p.

Portland, 2007 Portland Pattern Repository : *About the Portland Form*, http://c2.com/ppr/about/portland.html. Dernière visite : Juillet 2007.

Pree, 1994 W. Pree (1994) : *Meta Patterns - A Means for Capturing the Essentials of Reusable Object-Oriented Design*. Proceedings of ECOOP 94, Vol., n°, p.150-162.

Rational, 2008 IBM: *Site web d'IBM Rational*, http://www-01.ibm.com/software/fr/rational/. Dernière visite : Août 2008.

Saidi et al., 2007 R. Saidi, N. Arnaud, D. Rieu, M. Fredjm (2007) : *Multi-View Variability Modelling for Business Component Reuse*. First IEEE international workshop on Methods and Design models for Health Information Systems (MeDeHIS'07).

Sunye et al., 2000 G. Sunyé, A. Le Guennec, Jézéquel Jean-Marc (2000) : *Design Patterns Application in UML*. ECOOP'00, p44-62.

Sunye, 1999 G. Sunyé (1999) : *Génération de code à l'aide de patrons de conception.*. LMO, p163-178.

Tarpin-Bernard et al., 1998 F. Tarpin-Bernard, B. T. David, Primet P. (1998) : *Frameworks and Patterns for Synchronous Groupware: AMf-C Approach.*. EHCI, p225-241.

The Hillside Group, 2007 The Hillside Group : *A pattern definition*, http://hillside.net/patterns/definition.html. Dernière visite : Juillet 2007.

TRAFIC, 2008 TRAFIC, Equipe projet : *Projet TRAFIC : "Transports : Réutilisation, Adaptation, Fiabilité, Intermodalité et Coopération inter-systèmes"*, http://projet-trafic.imag.fr. Dernière visite : Juillet 2008.

Van der Massen et Lichter, 2002 T. Van der Massen, H. Lichter (2002) : *Modeling variability by UML use case diagrams*. International Workshop on Requirements Engineering for Product Lines (REPL'02), p19-25.

VanGrup et al., 2001 J. Van Gurp, J. Bosch, M. Svahnberg (2001) : *On the Notion of Variability in Software Product Lines*. , p45-54.

WELIE ET GERRIR, 2003 M. VAN WELIE, G. C. VAN DER VEER (2003) : *Pattern Languages in Interaction Design*. INTERACT'03, p527-534.

WIRFS-BROCK ET AL., 1990 R. WIRFS-BROCK, R. BROCK, B. WILKERSON (1990) : *Designing Object-Oriented Software*. International Ed.

ZIADI ET AL., 2003 T. ZIADI, L. HÉLOUËT, J. JÉZÉQUEL (2003) : *Towards a UML Profile for Software Product Lines*. 5th workshop on Software Product-Family Engineering, p129-139.

Résumé

Les patrons d'ingénierie ont été introduits afin de capitaliser et de réutiliser des savoirs et savoir-faire. Dans l'ingénierie logicielle, leur usage est aujourd'hui reconnu, à tous les niveaux (analyse, conception, ...), comme un gage de qualité. Outre une solution, un patron comporte également de nombreuses informations, en langage naturel, décrivant des contraintes et/ou variantes. Dans cette thèse, nous nous intéressons à l'activité de réutilisation des patrons (que nous appelons « imitation ») d'analyse ou de conception pour lesquels la solution est donnée sous la forme de spécifications orientées objet. L'imitation consiste en une adaptation et une intégration de cette solution par l'ingénieur d'applications qui tiennent compte des autres informations contenues dans le patron. Ainsi, nous dégageons trois axes que nous considérons comme les piliers d'une bonne imitation : la complétude, la variabilité et la généricité des solutions.

Nous proposons une nouvelle forme de définition des solutions qui s'appuie sur l'utilisation de plusieurs vues (fonctionnelle, dynamique et statique) ainsi que sur l'utilisation d'un méta-modèle permettant d'une part d'exprimer à partir de la vue fonctionnelle, la variabilité de la solution (fonctionnalités obligatoires, facultatives, optionnelles ou alternatives) et d'autre part d'exprimer l' « essence » de la solution sous la forme de propriétés génériques définissant les bornes des adaptations permises lors de l'imitation. Un processus d'imitation dédié ainsi qu'un premier outillage basé sur l'approche IDM (Ingénierie Dirigée par les Modèles) sont également proposés aux ingénieurs d'applications.

Mots-clefs

Patrons orientés-objet, Réutilisation, Complétude, Variabilité, Généricité, Modélisation, Méta-modélisation.

Title

Reliability of pattern reuse through completeness, variability and genericity of specifications.

Abstract

Engineering patterns have been introduced in order to capitalize and reuse knowledge and know-how. Nowadays, in software engineering, their use is acknowledged at each step of the process (analysis, design…) as a guarantee of quality. In addition to a solution, a pattern consists also of a lot of information, in natural language, describing constraints and/or variants. In this thesis, we focus on the reuse (that we name "imitation") of analysis or design patterns, which solution is given as object oriented specifications. An imitation consists of an adaptation and an integration of this solution by the application engineer considering the other information provided by the pattern. Thus, we draw three axis that we consider as pillars for a good imitation: completeness, variability and genericity of solutions.

We propose a new way of defining solutions that leans on the use of many views (functional, dynamic and static) along with the use of a metamodel. On the one hand, the metamodel allows expressing, from the functional view, the solution variability (mandatory, facultative, optional or alternative functionalities). On the other hand, it permits expressing the nature of the solution using generic properties defining the bounds of the adaptation admitted for the imitation. A dedicated imitation process and a first tool based on MDE (Model Driven Engineering) are also offered to applications engineers.

Keywords

Object-oriented patterns, Reuse, Completeness, Variability, Genericity, Modelling, Metamodelling.