RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ LARBI BEN MHIDI-OUM EL BOUAGHI

Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie

Département de Mathématiques et Informatique

Laboratoire de recherche sur les systèmes informatiques complexes ReLa(CS)^2

THÈSE

pour l'obtention du diplôme de Docteur 3ème cycle LMD en Informatique
Option : Ingénierie des Systèmes Distribuées

Thème :

VERS UN ENVIRONNEMENT DE SIMULATION À ÉVÈNEMENTS DISCRETS BASÉ SUR LE PARADIGME DE LA PROGRAMMATION ORIENTÉE ASPECTS

Présentée par :

Meriem CHIBANI

Soutenue publiquement le :

Dirigée par :

Pr. Belattar Brahim, Université de Batna.
Dr. Bourouis Abdelhabib, Université d'Oum El Bouaghi.

Devant le jury composé de :

Pr Bilami Azeddine	Professeur	Président	Université de Batna
Pr Chikhi Salim	Professeur	Examinateur	Université de Constantine 2
Dr Derdouri Lakhdar	MCA	Examinateur	Université d'Oum El Bouaghi
Dr Bourouis Abdelhabib	MCA	Rapporteur	Université d'Oum El Bouaghi

Université d'Oum El Bouaghi -2015-

Remerciements

Je voudrais en tout premier lieu remercier **Pr Brahim BELATTAR** mon ancien directeur de thèse, Professeur au département d'informatique à l'université de Batna, qui a suivi et encadré ce travail avec intérêt, disponibilité et compétence.

Je tiens à remercier chaleureusement **Dr Abdelhabib BOUROUIS** maître de conférences à l'université d'Oum El Bouaghi et mon encadreur. J'ai énormément apprécié les années de travail sous sa direction. Merci pour son soutien, ses conseils, son expérience et pour le temps qu'il m'a consacré.

Merci également aux personnes qui m'ont fait l'honneur d'accepter de participer à mon jury :

Pr. **Bilami azeddine**

Pr. **Chikhi Salim**

Dr. **Derdouri Lakhdar**

Enfin, j'adresse mes plus sincères remerciements à ma famille et tous mes proches qui m'ont toujours soutenu, encouragé et aidé au cours de la réalisation de ce travail.

Meriem Chibani

Dédicace

Je dédie ce travail
A mes très chers parents
A mon cher frère
A mes chères sœurs
A mon marie.

Meriem

Résumé

Résumé

La simulation orientée objet est actuellement très répandue et se base sur le paradigme orienté objet (OO). Les systèmes de simulation à évènements discrets (DES) mettent en œuvre plusieurs préoccupations transversales telles que la gestion des événements, la détection de la phase d'équilibre et le suivi de la trace d'une simulation. Ces préoccupations ont tendance à produire deux problèmes majeurs qui dépassent les capacités du paradigme OO utilisé en simulation : l'enchevêtrement et la diffusion de code de simulation. Cela augmente la complexité et réduit la maintenabilité qui exige une séparation spécifique des préoccupations (Separation of Concerns : SoC). La programmation orientée aspect (AOP) apprête une plus grande attention aux préoccupations transversales relativement à d'autres paradigmes en offrant des langages robustes tel qu'AspectJ.

L'application de l'AOP dans la simulation constitue un axe de recherche novateur et d'actualité. Dans cette thèse, nous proposons une architecture pour un environnement de modélisation et de simulation à évènements discrets basé sur l'AOP. Notre contribution comprend trois volets : le premier, représente une étude comparative entre les approches de la programmation orientée aspect tel que la programmation orientée sujet et Xerox Parc AOP. Le deuxième volet concerne l'identification des besoins non fonctionnels dans le domaine de la modélisation et de la simulation à évènements discrets. En plus, une architecture pour un environnement de modélisation et de simulation à évènements discrets basé sur l'AOP est proposée, en utilisant la bibliothèque Japrosim comme un noyau. Enfin, le dernier volet concerne la proposition d'un profile UML pour l'application de l'AOP au niveau de la conception. Ce profile est spécifique au langage AspectJ et utilise l'outil Xpand pour la génération automatique de code.

Mots-clefs

Programmation orientée aspect, simulation à évènements discrets, séparation des préoccupations (SoC), programmation orientée sujet, Japrosim, profile UML.

Abstract

Towards a discrete event simulation environment based on the aspect-oriented programming paradigm

Abstract

The object oriented simulation is currently widespread and is based on the object oriented (OO) paradigm. Discrete event simulation (DES) systems implement several crosscutting concerns such as event handling, steady state detection and keeping track of a simulation's state. These concerns tend to produce two major problems that exceed the capacity of the OO paradigm used in simulation : tangling and scattering simulation code. This increases the complexity and reduces the maintainability which requires specific separation of concerns (SoC). The aspect oriented programming (AOP) paradigm puts a greater focus on crosscutting concerns than other language paradigms by offering a strong language mechanisms such as AspectJ.

The application of the AOP in the simulation domain is a recent and innovative research field. In this thesis, we have proposed a discrete event simulation environment based on AOP paradigm. Our contribution is divided into three parts. The first one represents a comparative study between the aspect-oriented approaches such as subject-oriented programming and Xerox Parc AOP. The second part concerns the identification of the non-functional concerns of a discrete events simulation systems. Moreover, an architecture for discrete event simulation based on the AOP paradigm using Japrosim library is proposed. Finally, the last part concerns the proposition of a UML profile for the implementation of the AOP at the design level. This profile is specific to the AspectJ language and uses Xpand tool for automatic code generation.

Keywords

Aspect-oriented programming, discrete event simulation, separation of concerns (SoC), subject-oriented programming, Japrosim, UML profile.

Table des matières

Introduction générale **2**

I Etat de l'art **5**

1 Fondement de La programmation orientée aspect (AOP) **6**
1.1 Introduction 8
1.2 Motivations de l'AOP 8
1.3 Les approches de la programmation orientée aspect 11
1.4 La programmation orientée aspect : concepts de base 16
1.5 La démarche de mise en œuvre d'un programme orienté aspect 19
1.6 Les outils supportant le paradigme orienté aspect 20
1.7 Programmer avec AspectJ 23
1.8 La programmation orientée sujet 33
1.9 Conclusion 40

II Proposition **41**

2 La gestion des préoccupations transversales dans un système de simulation à évènements discrets **42**
2.1 Introduction 43
2.2 La simulation à évènements discrets 43
2.3 État de l'art des approches de modélisation et de simulation orientée aspect 44
2.4 Les principales préoccupations transversales d'un système à évènements discrets 48
2.5 Le processus de mise en œuvre d'un système de simulation orientée-aspect 50
2.6 Conclusion 50

3 Un nouveau profile UML pour la modélisation orientée aspect **52**
3.1 Introduction 53
3.2 La modélisation orientée-aspect 53
3.3 Les critères de classification des approches de modélisation orientée-aspect 54
3.4 Le processus de développement 55
3.5 UML pour la conception orientée aspect 57
3.6 Le profile AspectJ 58
3.7 Conclusion 68

III Implémentation 69

4 Mise en œuvre utilisant la bibliothèque Japrosim 70
4.1 Introduction 71
4.2 La bibliothèque Japrosim 71
4.3 La version orientée-aspect de Japrosim 72
4.4 La comparaison entre les deux version AO et OO de Japrosim 82
4.5 Conclusion 86

5 Profile AspectJ : Implémentation 87
5.1 Introduction 88
5.2 L'environnement de développement du profile 88
5.3 Génération de code 89
5.4 Application du profile 94
5.5 Conclusion 98

Conclusion 100

Conclusion générale 100

Bibliographie 102

Liste des tableaux

1.1 Une comparaison entre les approches de l'AOP [27]. 16
1.2 Les points de jonction exposés par AspectJ basé sur [61]. 25
1.3 Liste des Wildcard et leur signification [17]. 25
1.4 Les types de coupes offertes par AspectJ basé sur [61]. 27
1.5 Comparaison entre Hyper/J et AspectJ basé sur [21]. 36
1.6 Modèle de correspondance entre l'AOP et la SOP [28] 39

3.1 Comparison entre les langages de l'AOP [3]. 59

4.1 Comparaison entre les versions OO et AO de Japrosim [24]. 85

Table des figures

1.1 Les exigences non-fonctionnelles d'un système de gestion de base des données [8]. . . . 10
1.2 Localisation du code des fonctionnalités transversales dans des aspects séparés. . . . 11
1.3 CF-objet [12]. . . . 12
1.4 La *transversal strategy* [92]. . . . 13
1.5 Un sujet formé par la composition de deux sujets : *Shipping* et *Transportation* [72]. . . . 14
1.6 Modèle générique d'un système orienté aspect [61]. . . . 15
1.7 Le cycle d'évolution de l'AOP [61]. . . . 17
1.8 Le tisseur de l'AOP 18
1.9 Le processus de développement d'un programme orienté aspect [60]. . . . 19
1.10 La chaîne de tissage de l'outil PHPaspect [19]. . . . 21
1.11 La structure des aspects avec AspectS [48]. . . . 22
1.12 Un exemple d'un aspect avec AspectC++ [65]. . . . 22
1.13 DemoAspect.aj 24
1.14 Coupes nommé et anonyme sans paramètres. . . . 26
1.15 Les messages affichés par le compilateur lors de la détection des points de jonction. . . . 32
1.16 Les points de vues subjectifs de l'objet voiture. . . . 33
1.17 Composition des sujets basée sur [11]. . . . 34
1.18 Le mécanisme de composition dans l'AOP et la SOP [84]. . . . 39

2.1 Les domaines de recherche de l'AOP [88]. . . . 48
2.2 Le processus d'implémentation d'un système de simulation orienté aspect [27]. . . . 50

3.1 Le développement logiciel orienté-aspect. . . . 54
3.2 Catalogue des critères d'approches AOM [82]. . . . 55
3.3 Processus Aspectuel [54]. . . . 56
3.4 Processus Hybride [54]. . . . 57
3.5 L'architecture de metamodel selon l'OMG [43]. . . . 58
3.6 Le profile AspectJ. . . . 60
3.7 Le stereotype *Aspect*. . . . 61
3.8 Le stéréotype *Advice*. . . . 64
3.9 Le stéréotype *Pointcut*. . . . 65
3.10 Les stéréotypes : *StaticCrosscutting.static* et *StaticCrosscutting.dynamic* . . . 68

4.1 Diagramme de classes du paquetage *kernel* [27]. . . . 72
4.2 L'interaction entre les aspects et les paquetages de la version AO de Japrosim. 73

4.3 L'animation graphique basée sur le pattern Observateur [24]. 75
4.4 La détection de l'état stationnaire fondé sur le pattern Observateur [25]. . . 76
4.5 La méthode remove() de la préoccupation transversale trace de simulation en surbrillance [24]. 77
4.6 L'infection de classes de Japrosim avec la préoccupations transversale GUI [24]. 81
4.7 La relation entre la suite de métriques C & K et les propriétés qualitatives d'un système [89]. 83
4.8 Les résultats des mesures pour les paquetages de la version AO de Japrosim à l'aide de l'outil AOPMetrics [27]. 85
4.9 La différence des résultats de mesures des métriques des paquetages pour les deux versions AO et OO de Japrosim [27]. 86

5.1 L'architecture de papyrus [44]. 88
5.2 Fenêtre principale de papyrus. 89
5.3 Le principe de modèle templates et metamodel [91]. 90
5.4 Le principe de moteur workflow [81]. 90
5.5 Les aspects de Japrosim après l'application du profile AspectJ. 95
5.6 L'aspect Singleton. 96
5.7 La coupe CompositeP2NP1. 96
5.8 Les Tag values de Singleton stereotype. 96
5.9 L'aspect SteadyStateDetection. 97
5.10 La préoccupation transversal statique Declar1. 97
5.11 Les Tag values de SteadyStateDetection. 98
5.12 Code généré à partir du modèle de la figure 5.5. 99

Introduction générale

Contexte de recherche

Cette thèse s'inscrit dans le contexte du génie logiciel. Un environnement de modélisation et de simulation à évènements discrets possède des besoins fonctionnels et des besoins non fonctionnels. La programmation orientée aspect (AOP) en tant que technique de génie logiciel étend la programmation orientée objet avec la notion d'aspect, elle peut rendre modulaire des problèmes qui se recoupent tels que la distribution et la persistance. Nous nous intéresserons plus particulièrement a son apport dans le domaine du développement des environnements de modélisation et de simulation à évènements discrets.

Problématique

La programmation par objets constitue indéniablement une technologie importante pour aider à la construction d'applications complexes. En effet, elle favorise la réutilisation et permet ainsi la réduction des délais de développement et de maintenance. Cependant, cette réutilisation n'est pas toujours aisée. C'est notamment le cas pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de services donnés, mais nécessite également la prise en compte de différentes propriétés non-fonctionnelles telles que la distribution, la persistance ou encore le temps réel. Dans de telles applications, l'implémentation des services réalisés et celle des différentes propriétés non-fonctionnelles se trouvent intimement enchevêtrées. De ce fait, la réutilisation se trouve compromise. En effet, il n'est pas toujours possible de réutiliser les différentes propriétés non-fonctionnelles d'une application, indépendamment les uns des autres.

Dans le but de permettre une meilleure réutilisation, le paradigme de la programmation par aspects propose de structurer les applications sur la base du concept d'aspect. Un aspect est une brique de l'application représentant totalement et exclusivement une propriété donnée de l'application. La programmation par aspects consiste à réaliser une application en deux temps. Tout d'abord, les différents aspects sont définis séparément les uns des autres. Les définitions des aspects doivent êtres découpées de sorte qu'elles ne se référencent pas les unes les autres. Ensuite, l'application est produite par la composition des aspects ainsi définis.

Puisque le domaine de la modélisation et de la simulation permet d'étudier des phénomènes variés. La présence de diverses préoccupations non-fonctionnelles telles que l'animation graphique, la détection de l'état stable et la synchronisation assure la satisfaction des besoins fonctionnels des systèmes de simulation, alors que l'implémentation de ces derniers est dispersée dans les différents modules fonctionnels du système. En ef-

fet, leur implémentation dans le paradigme objet est souvent transversale à la hiérarchie des classes, et a tendance à produire des problèmes de dispersion et d'enchevêtrement de code ce qui diminue la qualité des systèmes et pénalise leurs capacités d'évolution. Il s'avère donc très intéressant de définir un environnement de simulation orienté aspect et d'exploiter ce paradigme dans le domaine de la modélisation et de la simulation.

Objectifs

L'objectif principale de ce manuscrit est d'étudier les principales techniques de la mise en oeuvre de l'AOP et de faire le point sur l'utilisabilité de ce paradigme dans le domaine de la modélisation et la simulation des systèmes à évènements discrets (DEVS) comme suit :

- Exploration des avantages de l'AOP.
- Exploration comparative des approches déjà proposées de l'AOP.
- Identifier les besoins non-fonctionnels dans la modélisation et la simulation à évènements discrets.
- La proposition d'une nouvelle architecture pour le domaine de la modélisation et de la simulation à évènements discrets basé sur l'AOP afin de séparer leur préoccupations.
- Proposer un nouveau support de l'AOP au niveau de la conception et l'utilisation de ce dernier dans le domaine de simulation.

Organisation de la thèse

CEtte thèse comprend trois parties essentielles qui sont organisés comme suit :

- **Introduction générale**
- **Partie I**
  - **Chapitre 1 :** présente une étude des fondements du paradigme de l'AOP et des approches d'implémentation, en particulier AspectJ qui constitue l'une des implémentations majeures de l'AOP. Il traite aussi les fondements de la programmation orienté sujet (SOP : Subject Oriented Programing) avec une étude comparative entre l'AOP et la SOP.
- **Partie II**
  - **Chapitre 2 :** ce chapitre identifie les différentes préoccupations transversales d'un système à évènements discrets et les perspectives d'utilisation du paradigme de l'AOP dans ce domaine.
  - **Chapitre 3 :** ce chapitre propose un nouveau profile UML pour la modélisation orientée aspect spécifique au langage AspectJ.
- **Partie III**
  - **Chapitre 4 :** ce chapitre propose une architecture pour un environnement de modélisation et de simulation à évènements discrets basé sur l'AOP et utilisant comme noyau la bibliothèque Japrosim. En plus, une comparaison entre les deux versions de Japrosim illustre l'effet de l'AOP en terme d'amélioration de la qualité du logiciel tel que la modularité, l'adaptabilité et la réutilisabilité.

- **Chapitre 5 :** présente le profile UML proposé, le processus de génération de code, et l'application de ce profile sur la bibliothèque Japrosim.

- **Conclusion générale**

Première partie

Etat de l'art

Chapitre 1

Fondement de La programmation orientée aspect (AOP)

« *Connaître, ce n'est point démontrer, ni expliquer. C'est accédé à la vision. Mais pourvoir, il convient d'abord de participer : cela est un dur apprentissage* »

Antoine de Saint-Exupéry

Sommaire

1.1 Introduction . . . 8
1.2 Motivations de l'AOP . . . 8
1.2.1 Les principes du paradigme orienté objet . . . 8
1.2.2 Les limites du paradigme orienté objet . . . 9
1.3 Les approches de la programmation orientée aspect . . . 11
1.3.1 Les filtres de composition (CF) . . . 11
1.3.2 La programmation adaptative (AP) . . . 12
1.3.3 La programmation orientée sujet . . . 13
1.3.4 XEROX Parc AOP . . . 14
1.4 La programmation orientée aspect : concepts de base . . . 16
1.4.1 Historique . . . 16
1.4.2 Principe . . . 17
1.4.3 Concepts de base . . . 18
1.5 La démarche de mise en œuvre d'un programme orienté aspect . . . 19
1.5.1 La décomposition aspectuelle . . . 19
1.5.2 L'implémentation de chaque préoccupation . . . 19
1.5.3 La recomposition aspectuelle . . . 19
1.6 Les outils supportant le paradigme orienté aspect . . . 20
1.7 Programmer avec AspectJ . . . 23
1.7.1 Aspects . . . 23
1.7.2 Points de jonction . . . 23
1.7.3 Coupes . . . 26
1.7.4 Advices . . . 28

1.7.5 Mécanisme d'introduction 30
1.7.6 Modifier le comportement du compilateur 31
1.7.7 Ramollissement des exceptions 32
1.8 La programmation orientée sujet 33
1.8.1 Principe et objectifs 33
1.8.2 Hyper/J 34
1.8.3 Comparaison entre Hyper/J et AspectJ 35
1.8.4 Relations entre AOP et SOP 37
1.9 Conclusion 40

1.1 Introduction

Un programme (logiciel) informatique est constitué de différentes préoccupations fonctionnelles qui répondent aux objectifs pour lesquels le système a été développé et des préoccupations non-fonctionnelles telles que la distribution, la concurrence des données, les interfaces homme-machine, la prise en compte du temps réel, la gestion de la persistance ou la tolérance aux pannes. Les préoccupations non-fonctionnelles assurent la satisfaction des besoins fonctionnels. Au même temps, l'implémentation de ces préoccupations est dispersée dans les différents modules fonctionnels du système. En effet, leur implémentation dans le paradigme objet est souvent transversale à la hiérarchie des classes, et a tendance à la *polluer*. Ce qui pénalise les capacités d'évolution de l'application et la laisse non conforme au critère de l'inversion des dépendances et le principe d'ouvert-fermé. L'AOP constitue un nouveau paradigme qui vient combler l'absence d'un support adéquat des besoins non-fonctionnels. Ce chapitre va jalonner les premiers pas dans le monde de la programmation orientée aspect. Il explique les principes de base des approches de la programmation orientée aspect. Ensuite, les principes et les concepts de l'approche XEROX Parc AOP seront abordés. Cette approche est la plus connue et prend actuellement le nom "programmation orientée aspect".

1.2 Motivations de l'AOP

1.2.1 Les principes du paradigme orienté objet

Les grands principes qui suivent peuvent être vus comme autant des objectifs ou comme autant des outils d'évaluation du paradigme orienté objet [41] :

Faible couplage

L'un des objectifs d'une application orientée objet est de pouvoir intégrer des évolutions futures du logiciel sans faire des changements radicales. Une manière d'éviter un impact trop important de modifications à venir et en diminuant le couplage existant entre les composants constituant un logiciel.

Forte cohésion

Les langages orientés objet offrent plusieurs outils de modularisation du code tels que les classes et les espaces de nommage. L'idée de ce second principe est de vérifier qu'une classe rassemble bien des méthodes cohérentes, qui visent à réaliser des objectifs similaires. De même, il ne faudrait rassembler dans un même espace de nommage que des classes répondant à un besoin précis.

Inversion des dépendances

Ce troisième principe se base sur le faible couplage. Une classe ne doit jamais dépendre d'une autre classe qui serait d'un niveau conceptuel plus bas. Pour limiter un tel couplage, l'inversion des dépendances propose :

- D'inverser la relation à l'origine du couplage lorsque c'est possible.
- Lorsque c'est impossible, l'inversion des dépendances propose d'introduire un niveau d'abstraction plus élevé entre les classes fonctionnelles et les classes techniques. Typiquement, il faudrait spécifier des interfaces qui exposent les services

des classes techniques, et dont dépendrait les classes fonctionnelles. Cela limiterait le couplage, sans l'éliminer totalement.

Principe de substitution de Liskov

Ce principe, également appelé "principe des 100 %", permet d'estimer la validité d'une relation d'héritage entre deux classes. En effet, pour chaque relation d'héritage, il faudrait toujours que nous puissions :

— Prononcer la phrase "X est un Y", où X représente la classe fille et Y la classe mère.

— Remplacer la classe Y par la classe X dans toutes les situations sans que cela ne mène à une incohérence.

Ouvert fermé

Pour ce principe, un logiciel orienté objet doit être ouvert aux modifications, aux évolutions futures, mais que ces modifications doivent pouvoir se faire sans changer la moindre ligne de code existante. Le code du logiciel actuel est identifié, testé, et validé, donc modifier ce code doit se faire en prenant de nombreuses précautions ; en particulier, il faut mesurer l'impact de la modification avant de l'effectuer. Grâce à ce principe l'ajout de nouvelles classes, accompagnées de leurs tests et à qui l'on affecterait les nouvelles responsabilités du logiciel, serait beaucoup plus léger, logique et maintenable.

En effet, il faut prévoir lors de la conception et de l'implémentation initiales du logiciel des points d'extensibilité, c'est-à-dire des endroits où l'on pense qu'une évolution aura probablement lieu. Les situations nouvelles pourront être implémentées par de nouvelles classes héritants de classes déjà existantes et la sollicitation de ces nouvelles classes se fera dynamiquement par polymorphisme.

1.2.2 Les limites du paradigme orienté objet

Même en utilisant toute la puissance des langages orientés objet et des patrons de conception les plus évolués, il reste dans le cœur des applications orientées objet des éléments de code qui sont impossibles à centraliser et à factoriser. Ces éléments sont en-dessous de la granularité d'une méthode, dès que l'on souhaite mettre un ensemble des instructions, les langages actuels nous obligent à réunir ces instructions en méthodes, ce qui n'est pas toujours possible ou élégant. Parmi les limitations de la programmation orientée objet, nous citons [41] :

L'héritage ne permet pas d'éviter complètement la duplication de code ni de garantir la cohérence globale de la gestion d'un problème précis, certains problèmes techniques que l'on aurait pu résoudre par une relation d'héritage deviennent délicats lorsque les langages nous restreignent à un héritage simple de classe. Ce que *C++* et *Eiffel* permettent, *Java* et *C♯* l'interdisent.

D'autre part, le bon usage du langage Java impose de créer des classes dont les attributs sont privés et d'offrir des méthodes *setXxx()* et *getXxx()* pour accéder à ces attributs. C'est une bonne pratique mais elle est un peu lourde à faire supporter par les développeurs. Les outils intégrés se sont donc empressés de fournir des assistants pour déduire automatiquement ces accesseurs de la liste des attributs présents dans une classe. C'est très pratique lorsqu'on les génère pour la première fois et qu'aucun de ces accesseurs n'a de comportement particulier, mais la maintenance de ce type de classes peut vite devenir lourde par rapport à sa valeur ajoutée.

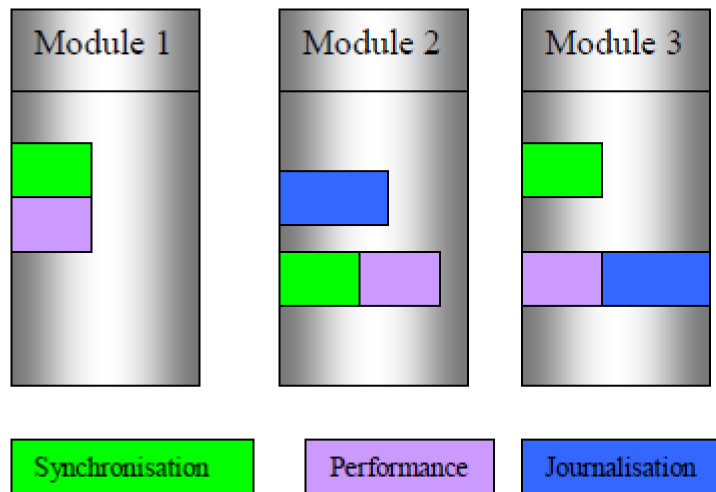


FIGURE 1.1 : Les exigences non-fonctionnelles d'un système de gestion de base des données [8].

En outre, la délégation est mal implémentée par de nombreux langages orientés objet (OO).

Les exigences non-fonctionnelles qui traversent les modules fonctionnels du système posent deux problèmes [8] :

— **La dispersion du code** traitant un aspect du système à travers différents modules. L'unité fonctionnelle de langage orienté objet est le paquetage et si nous raffinons encore la découpe, cette unité devient la classe. Il n'est pas rare de voir des méthodes traitants une exigence non-fonctionnelle se dispersent dans l'implémentation des différentes classes composants le cœur fonctionnel du système. Par exemple, dans un système de gestion de base des données illustré par la figure 1.1 les préoccupations : *performance, journalisation* et *synchronisation* concernent toutes les classes accédant à la base des données seront implémentées dans plusieurs modules sans être bien circonscrits. Il s'agit très souvent de portions de code très similaires à ajouter un peu partout dans les modules concernés.

— **L'enchevêtrement du code** de différentes problématiques. C'est la présence des éléments de plusieurs problématiques dans l'implémentation d'un seul et même module. Dans l'exemple précédent, le code qui traite les aspects : *performance, journalisation* et *synchronisation* s'enchevêtre dans les classes d'accès à la base des données.

L'AOP propose une organisation différente des éléments de code. Elle offre une bonne répartition des compétences sans induire trop de redondance, ainsi qu'une certaine robustesse et une bonne performance des applications. En termes de qualité de conception, elle permet de mieux respecter les principes *d'ouvert fermé* et *d'inversion des dépendances*. La programmation orientée-aspect permet d'implémenter chaque problématique non-fonctionnelles indépendamment du code de base dans des aspects, puis, de les assembler selon des règles de tissage bien définies. En effet, elle promet une meilleure productivité, une meilleure réutilisation du code et une meilleure adaptation du code aux changements comme illustre la figure 1.2.

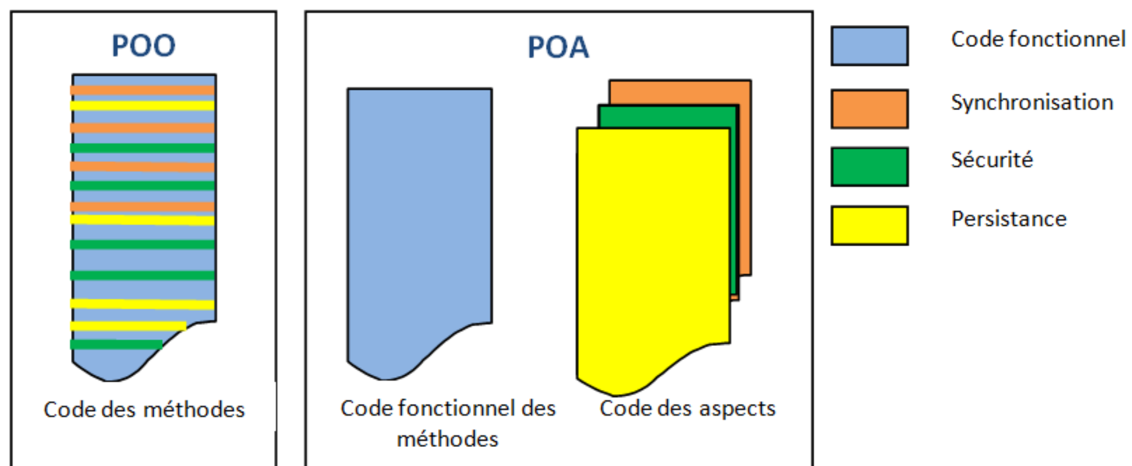


FIGURE 1.2 : Localisation du code des fonctionnalités transversales dans des aspects séparés.

1.3 Les approches de la programmation orientée aspect

La programmation orientée aspect gère les préoccupations transversales qui s'étendent sur plusieurs modules fonctionnels des systèmes menant à la dispersion et l'enchevêtrement du code. Elle a deux caractéristiques principales : la *quantification* et l'*inconscience* "AOP can be understood as the desire to make quantified statements about the behavior of programs and to have these quantifications hold over programs written by oblivious programmers" [40].

1.3.1 Les filtres de composition (CF)

Les filtres de composition est une approche de la programmation orientée aspect proposés par le groupe TRESE de l'université de Twente depuis 1988. Elle ajoute à un objet de base appelé noyau (Kernel) une ou plusieurs couches enveloppantes appelées interfaces qui interceptent les messages entrants et sortants. Un objet doté d'une interface est appelé CF-objet (voir figure 1.3). Le principe fondamental de cette approche est l'amélioration du modèle objet en interceptant et en manipulant tous les messages envoyés et reçus par cet objet. Cela permet d'exprimer plusieurs améliorations au niveau du comportement des objets, puisque dans les systèmes orienté objet le comportement extérieur visible d'un objet est exprimé par les messages qu'il envoie et qu'il reçoit. Un CF-objet est composé d'un ensemble d'éléments qui sont [6] :

1. Modules de Filtres : des sous-composants qui contiennent des filtres pour la manipulation des messages envoyés et reçus.
2. Filtres : les filtres définissent les améliorations possibles du comportement des objets. Chaque filtre spécifie une manipulation particulière des messages.
3. Objets internes : les méthodes de ces objets sont utilisées pour étendre le comportement d'un objet de base. Un message reçu par un CF-objet peut être délégué à un objet interne à la place de l'objet noyau auquel il était initialement adressé.
4. Objets externes : ils sont semblables aux objets internes. Cependant, ils peuvent exister indépendamment des CF-objets. Leurs références sont passées comme paramètres aux constructeurs des CF-objets au moment de leur instanciation.

En outre, l'objet noyau offre une interface d'accès aux méthodes disponibles.

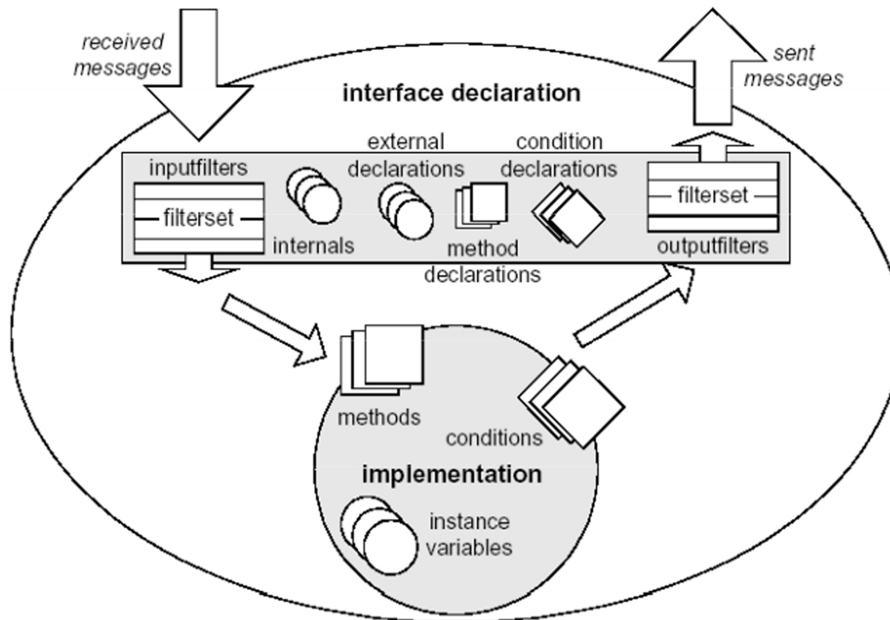


FIGURE 1.3 : CF-objet [12].

1.3.2 La programmation adaptative (AP)

La programmation adaptative a été introduite en 1991 par le groupe Demeter de l'université de Northeastern à Boston. Le groupe a utilisé les idées de XEROX AOP plusieurs années avant que le nom *programmation orientée aspect* soit inventé. Après que la collaboration avec le groupe Xerox PARC avait commencé, le groupe a redéfini AP et le terme AOP a été introduit. La programmation adaptative vise à séparer les préoccupations structurelles et comportementales du logiciel. Elle dresse la Loi de Demeter d'une manière que la composante comportementale a seulement une connaissance minimale de la structure du logiciel. La loi de Demeter, datant de 1987, s'annonce comme suit : " L'efficacité d'un projet informatique augmente si toutes les préoccupations de nature différentes sont bien modularisées et si un programmeur qui désire faire une modification ne doit parler qu'à ses voisins directs pour la faire tout en étant s*û*r de ne pas introduire de bugs ". La loi de Demeter est la pierre angulaire de la programmation adaptative. La méthode Demeter sépare efficacement les préoccupations structurelles et comportementales. Suite à cette loi de Demeter, la composante comportementale utilise seulement une quantité minimale d'informations sur la structure et elle est donc appelée *structure-shy*. La méthode Demeter se compose de deux concepts clés : la *transversal strategy* et le patron visiteur. La *transversal strategy* transmet l'information structurelle minimale nécessaire et le patron visiteur encapsule le comportement. Le parcours commence à partir d'un seul nœud dans un graphe d'objet. La *transversal strategy* définit les nœuds traversés qui seront visités et passés par l'objet visiteur. Le parcours est effectuée en-profondeur d'abord et le calcul est terminé lorsque tous les nœuds définis par la stratégie ont été visités [63]. On nomme la combinaison de la *transversal strategy* et l'objet visiteur *propagation pattern*.

Un exemple d'AP est présenté dans la figure 1.4. La partie gauche de la figure présente un diagramme de classe UML d'un système. Supposons que nous aimerions compter les personnes qui attendent à la station de bus au long de la route [92]. En programmation orientée objet ordinaire cela nécessite la mise en œuvre des petites méthodes dans toutes les classes concernées (en couleur gris). Si on utilise une *transversal strategy*, comme il est proposé dans l'AP, il n'est pas nécessaire de faire un changement dans les classes exis-

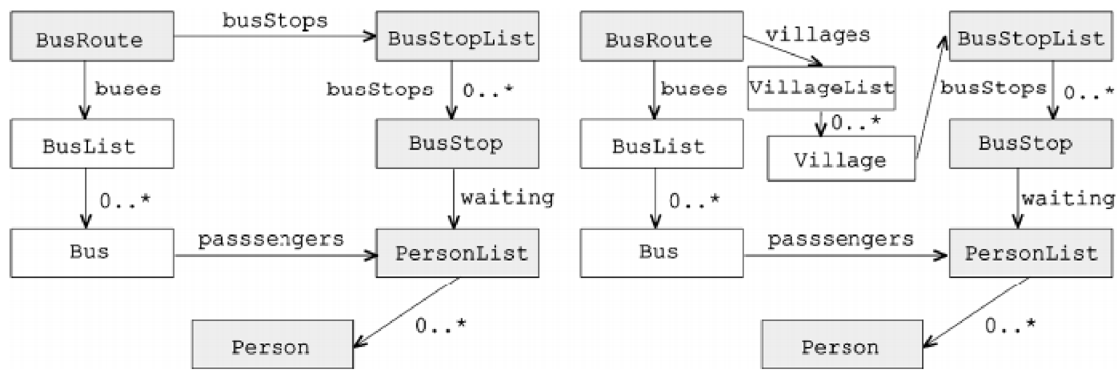


FIGURE 1.4 : La *transversal strategy* [92].

tantes. Dans ce cas, la *transversal strategy* : *from BusRoute through BusStop to Person* résout le problème de l'obtention des objets de la classe *Person* sur la ligne de bus, qui est suffisant de les compter. La partie droite de la figure démontre la robustesse de cette technique.

1.3.3 La programmation orientée sujet

La programmation orientée sujet (Subject Oriented Programming : SOP) est une extension du paradigme objet. Elle a été introduite en 1993 au centre de recherche d'IBM Thomas J. Watson. Un sujet est un programme ou un fragment du programme exprimé par le paradigme objet. La programmation par sujet propose de composer un ensemble de sujets afin de produire l'application finale. Ce processus de composition est nommé intégration. Les sujets ont tous le même niveau, aucun sujet est dominant par rapport aux autres. Les sujets peuvent être composés entre eux pour produire des sujets plus importants, qui peuvent être à leur tour composés entre eux. SOP est une extension de la programmation orientée objet qui prend en charge la construction des systèmes selon différents points de vue subjectifs. Elle est basée sur deux idées centrales : la division du système en différents sujets et les règles de composition. Un sujet est un modèle complet ou partiel d'objet. C'est une collection de classes connexes ou un fragment de classes pour un usage particulier tels que les sujets *Shipping* et *Transportation* qui sont illustrés par la figure 1.5. Un sujet peut être complet ou juste une partie d'une application. Ces différents sujets pourraient agir sur des objets partagés de manière indépendante [50].

L'intégration des sujets se fait en définissant des règles de composition [73]. Une règle implique au moins deux sujets, elle permet de définir par des opérations simples (fusion, redéfinition et séquencement) la composition. L'intégration est encapsulée par des modules de composition qui mettent en relation plusieurs sujets à intégrer ensemble. Chaque sujet est compilé séparément pour produire un sujet binaire. Le sujet binaire est constitué d'une étiquette qui contient les informations de ce sujet et un code binaire produit par le compilateur. Le compositeur orienté sujet utilise les informations offerts par les étiquettes pour lier les sujets. Il n'examine, ni modifie le code binaire des sujets individuels.

Parmi les objectifs de la programmation orientée sujet, selon [72] :

— Faciliter le développement et l'évolution des suites d'applications coopératives. Prendre en charge la décentralisation dans le temps et l'espace.

— Offrir la possibilité d'extension et de composition non prévue à priori, sans changer ou recompiler le code source existant. Pour cela, il suffit d'encapsuler dans un sujet

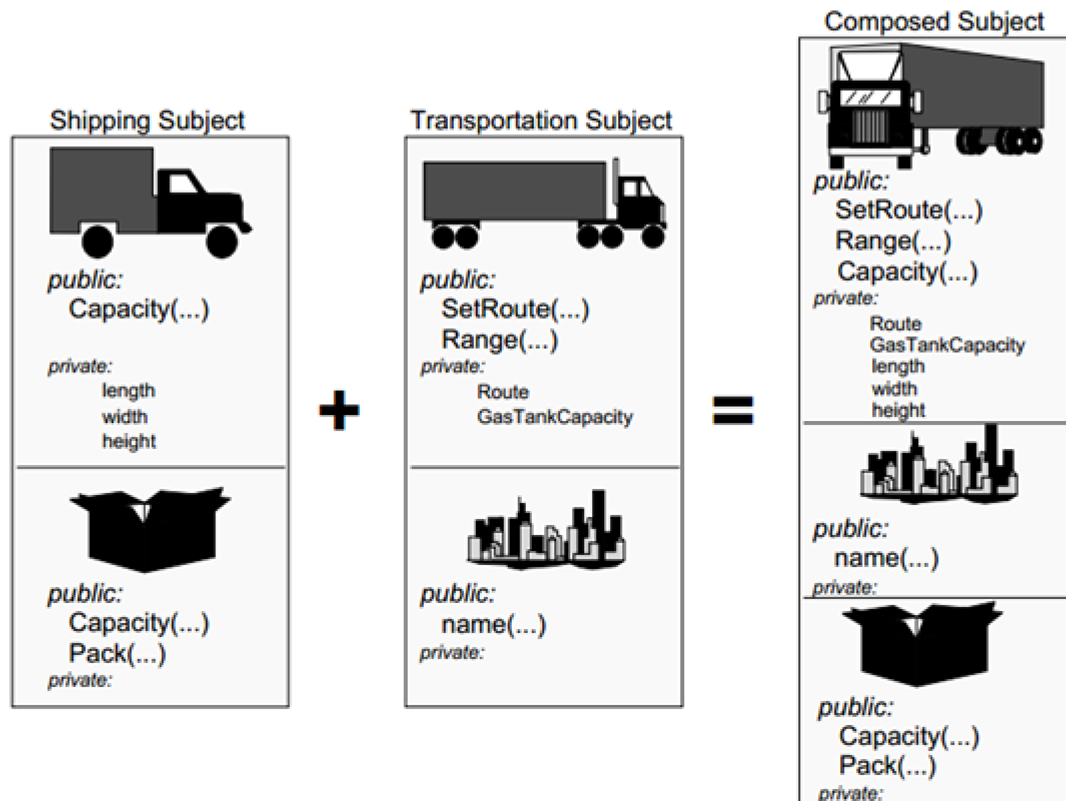


FIGURE 1.5 : Un sujet formé par la composition de deux sujets : *Shipping* et *Transportation* [72].

les extensions et de fournir les règles de composition pour l'intégrer avec les autres.

— Permettre le développement de classes décentralisées. Les auteurs de différentes applications qui partagent les mêmes objets peuvent avoir leurs propres définitions sur des objets partagés. Ces définitions peuvent ensuite être composées.

— Permettre le développement indépendant des préoccupations. L'implémentation des comportements peut être construit comme un sujet cohérent, plutôt que d'être entrelacé avec le reste du code.

1.3.4 XEROX Parc AOP

XEROX Parc AOP apporte une solution aux problèmes de dispersion et d'enchevêtrement de code par la séparation des préoccupations transversales dans de nouvelles unités de modularisation qui sont appelées *aspects*. C'est un nouveau paradigme de programmation qui trouve ses racines en 1996, suite aux travaux de Gregor Kiczales et de son équipe au centre de recherche Xerox à Palo Alto [57]. La XEROX Parc AOP est une technologie relativement jeune, puisque les premiers outils destinés à son utilisation ne sont apparus qu'à la fin des années 90. Cependant, l'adoption de cette approche est plutôt rapide à cause de la compatibilité entre ses concepts sous-jacents et ceux existants. Elle peut être intégrée à moindre co*û*t par les entreprises en étendant leurs outils. En Java elle est intégrée dans la plateforme Eclipse via un plugin qui permet l'utilisation du langage AspectJ. Ce langage étend le langage Java en introduisant de nouveaux mots-clés permettant de programmer des aspects mais l'AspectJ n'est pas l'unique langage orienté aspect. Même si celui-ci reste le plus utilisé, d'autres outils existent. On peut notamment citer JAC, JBoss AOP et AspectWerkz rien que pour le langage Java, mais il existe également des

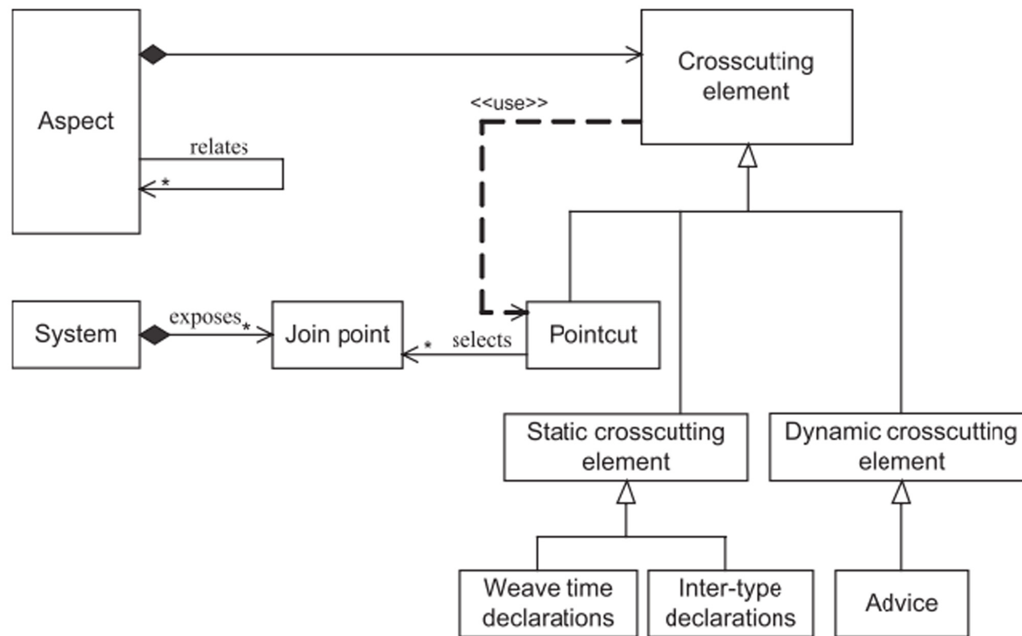


FIGURE 1.6 : Modèle générique d'un système orienté aspect [61].

outils pour les langages C, C++, C♯ou Smalltalk. La compatibilité entre les concepts utilisés par l'AOP et les outils existants est due au fait que l'AOP ne remet pas en cause les autres paradigmes de programmation comme l'approche procédurale et objet. Au contraire, elle les étend en proposant des mécanismes complémentaires afin d'améliorer la modularité d'une application et donc faciliter la réutilisation et la maintenance.

Un système de XEROX Parc AOP peut inclure de nombreux éléments. La figure 1.6 montre tous ces éléments et les relations des uns aux autres. Les points de jonction (join points) sont des endroits d'une application dans lesquels aura lieu le tissage. Les coupes (pointcuts) sont les mécanismes utilisés pour la sélection des points de jonction. Les advices sont attachées au points de jonction afin d'injecter les préoccupations transversales. Lorsque une advice est attachée à des points de jonction, elle est exécutée. En outre, elle a un modificateur qui précise son temps d'exécution par rapport aux points de jonctions : avant, après, autour, après exception, ou encore après le retour d'une valeur. L'advice est un élément transversal dynamique parce qu'il affecte l'exécution du système. En outre, l'implémentation d'AOP peut contenir des éléments transversaux statiques qui modifient la structure statique du système. En conclusion, XEROX Parc AOP est l'approche la plus mature et la plus utilisée [61]. Les membres d'IBM la considèrent comme une approche héritée du SOP par l'identification et la séparation des préoccupations non fonctionnelles, telles que la concurrence, la distribution, et la persistance. XEROX Parc AOP distingue la notion des *classes de base* qui encapsulent les exigences fonctionnelles d'un système et des *aspects* qui encapsulent les préoccupations non fonctionnelles (exigences transversales). Les aspects sont écrits selon les classes de base et chaque aspect contient des règles précisant comment cet aspect doit être tissé dans les classes de base [87]. Actuellement, XEROX Parc AOP prend le nom AOP [1]. Afin de bénéficier de techniques de l'AOP, une brève description des quatre approches de l'AOP est présentée sur le tableau 1.1. Cette description précise les principales constructions des approches et les relations

1. Ce terme sera utilisé dans le reste de cette thèse pour indiquer l'approche XEROX Parc.

TABLEAU 1.1 : Une comparaison entre les approches de l'AOP [27].

Critère \ Approche	**Xerox PARC AOP**	**SOP**	**AP**	**CF**
L'unité atomique	Aspect	Sujet	Propagation pattern	Filtre
La méthodologie d'accueil	orientée objet/composants/multiAgents	orientée objet	orientée objet	orientée objet
Le mécanisme de composition	le mécanisme de tissage	les règles de composition	le mécanisme de tissage	le mécanisme de superimposition
Le mécanisme utilisé pour la détection des zones de greffes des préoccupations transversales	coupes	étiquettes	transversal strategy	Type de filtre
Les zones d'injection des préoccupations transversales	les points de jonction	Les composants d'un sujet	le modèle d'objet transversal	La partie d'implémentation de CF-objet après l'interception de message
Langage	AspectJ	IBM VisualAge C++	DemeterJ	ComposeJ
Le moment de tissage	invoqué statiquement ou dynamiquement	Pendant la compilation	le temps d'exécution ou le temps de compilation	le temps d'exécution ou le temps de compilation

entre elles.

1.4 La programmation orientée aspect : concepts de base

1.4.1 Historique

Depuis 1997, la programmation orientée aspect provoque l'engouement de la communauté scientifique qui s'intéresse au génie logiciel. Gregor Kiczales est un des créateurs de ce concept et le fondateur d'AspectJ. Il dirige l'équipe du XEROX Parc qui a développé l'AOP et L'AspectJ. Cette technologie est passée par un cycle qui détermine sa maturité et son application dans le monde réel (voir la figure 1.12). Le cycle contient cinq phases. Dans la première (*technology trigger*), l'AOP a promis les développeurs des logiciels de donner des solutions pour plusieurs problèmes. Ensuite, dans la phase *peak of inflated expectations* plusieurs personnes sont attirées par la technologie mais seulement quelque

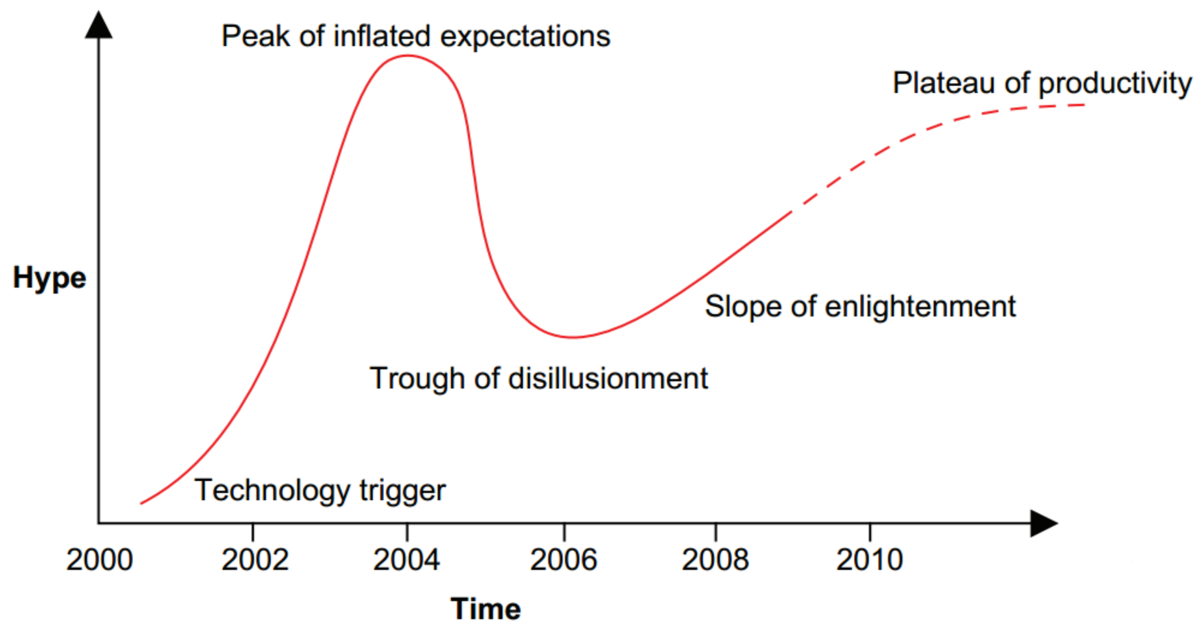


FIGURE 1.7 : Le cycle d'évolution de l'AOP [61].

développeurs l'essayaient. Puis, dans la phase de *disillusionment*, l'AOP a perdu beaucoup de sou éclat. Enfin, dans la phase de *slope of enlightenment* il y a un recul à cause de l'écart entre l'abstraction et l'application de l'AOP dans le monde réel. Actuellement l'AOP est dans la phase de productivité où elle apparait doucement dans les applications réelles [61]. On peut citer quelques dates importantes au cours d'évolution de ce paradigme [66] :

— En 1997, Gregor Kiczales a publié à propos de l'AOP.

— Ensuite, en 2001 la sortie d'AspectJ par Xerox PARC (trigger)

— En 2004 (peak) IBM travaille sur AJDT, AspectJ In Action 1

— En 2006, (desillusion) Pas d'adoption de masse

— Dans la phase d'enlightment, la fusion d'AspectJ et d'AspectWerkz, SpringSource prend le relai d'IBM, AspectJ in Action 2

1.4.2 Principe

L'AOP est une technique de programmation qui peut coexister avec l'approche orientée objet ou procédurale. Elle permet de factoriser et de rendre plus cohérentes certaines fonctionnalités, dont l'implémentation aurait nécessairement été répartie sur plusieurs classes et méthodes dans le monde objet ou sur plusieurs bibliothèques et fonctions dans le monde procédural. L'AOP n'est pas une technique autonome de conception ou de programmation. Mais inversement, les programmations orientée objet et procédurale ne sont pas complètes puisque inaptes factoriser ou à bien séparer certaines responsabilités des éléments logiciels [41].

L'implémentation d'AOP implique un langage de spécification ou une plateforme et un ensemble d'outils associés. On peut diviser ces exigences en deux parties [61] :

— Le langage de spécification : détermine le langage utilisé pour la programmation des préoccupations fonctionnelles et non-fonctionnelles des applications. On peut utiliser un langage standard comme java, c++, et c pour les deux types de préoccupations ou bien un langage différent.

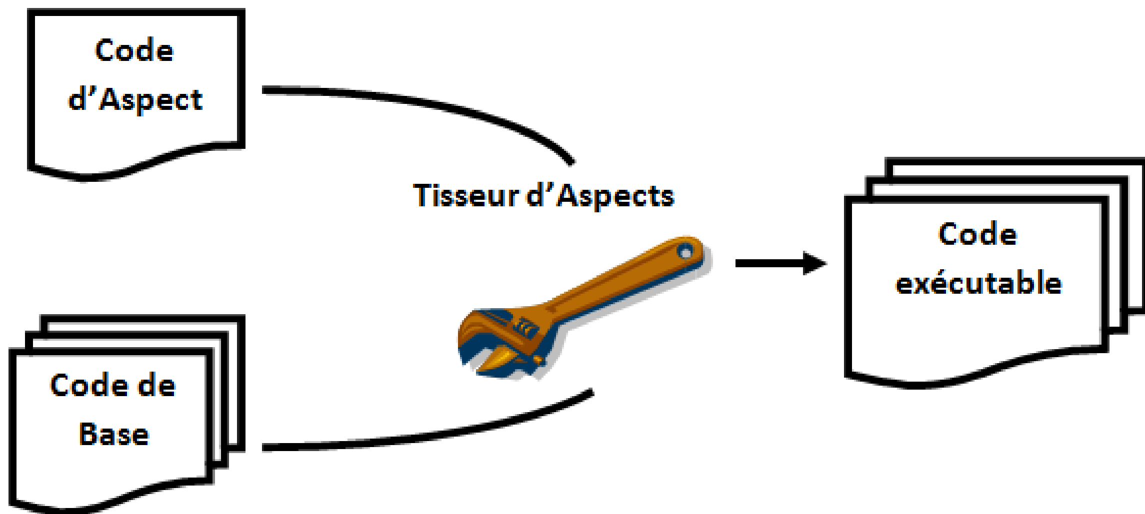


FIGURE 1.8 : Le tisseur de l'AOP

— Le langage d'implémentation : il procède logiquement en deux étapes. La première étape consiste à combiner le code transversal avec le code principal en utilisant les règles de tissage. Dans la deuxième, en convertissant le code obtenu vers une source exécutable. Les deux étapes résument le rôle d'un tisseur d'aspects (weaver) qui est illustré par la figure 1.8.

1.4.3 Concepts de base

Il y a des éléments importants pour un programme orienté aspect, on peut citer [41] :

1. Un aspect : est une abstraction d'une préoccupation. En effet, les aspects sont des modules à part entière, il faut pouvoir les composer pour construire une application. Selon les tisseurs, un aspect peut être une simple classe comme AspectDNG ou constituer un élément d'un langage spécifique comme AspectJ.
2. Les points de jonction : sont les endroits où les aspects interviennent avec le programme de base. Elles peuvent avoir une granularité variée. Chaque point de jonction possède une information contextuelle associée qui est utilisable par l'aspect pour savoir où ce dernier s'applique.
3. Les advices : sont les méthodes exécutés une fois que le flot d'exécution du programme atteint un point de jonction. Elles a un modificateur qui précise le moment de son exécution par rapport au point de jonction : avant, après, autour, après exception ou encore, après le retour d'une valeur. En plus, ces méthodes possèdent une variable d'instance supplémentaire nommée *thisJoinPoint* qui encapsule l'information contextuelle capturée depuis le point de jonction en cours : message intercepté,classe adressée par le message, paramètres, etc.
4. Code de base : ensemble de classes qui constituent une application ou une bibliothèque. Ces classes n'ont pas connaissance des aspects. Il n'y a aucune dépendance et elles ne savent pas qu'elles vont constituer la cible d'un tissage.
5. Tissage : est le processus qui consiste à ajouter le code des aspects sur les zones de greffe du code cible.

6. Langage de tissage : c'est le langage qui permet d'exprimer sur quelles zones de greffe doivent être tissés les aspects. Ce langage peut être intégré à celui des aspects (AspectJ) ou être séparé (AspectDNG qui utilise XPath comme langage de tissage).

1.5 La démarche de mise en œuvre d'un programme orienté aspect

Le cycle de développement d'un programme orienté aspect se divise en trois étapes (voir la figure 1.9) [60] :

1.5.1 La décomposition aspectuelle

Consiste à décomposer les besoins afin d'identifier et séparer les problématiques transversales et métiers. Cette phase est comparée au passage d'un rayon de lumière à travers un prisme afin de séparer ses différentes composantes chromatiques.

1.5.2 L'implémentation de chaque préoccupation

Consiste à implémenter chaque problématique séparément. Les problématiques métiers sont implémentées moyennant les techniques conventionnelles de programmation (orientée objet ou procédurale) alors que les problématiques transversales sont implémentée moyennant les techniques de l'AOP.

1.5.3 La recomposition aspectuelle

Consiste à construire le système final en intégrant les problématiques métiers avec les problématiques transversales. Nous pouvons comparer cette étape à un nouveau passage des composantes chromatiques dans un prisme qui les combine pour faire sortir un rayon de lumière unique. Cette phase est appelée *tissage*. Un tisseur utilise des règles spécifiées par le concepteur de l'application afin de combiner correctement les problématiques entre-elles. C'est un programme exécutable ou framework dynamique qui procède au tissage.

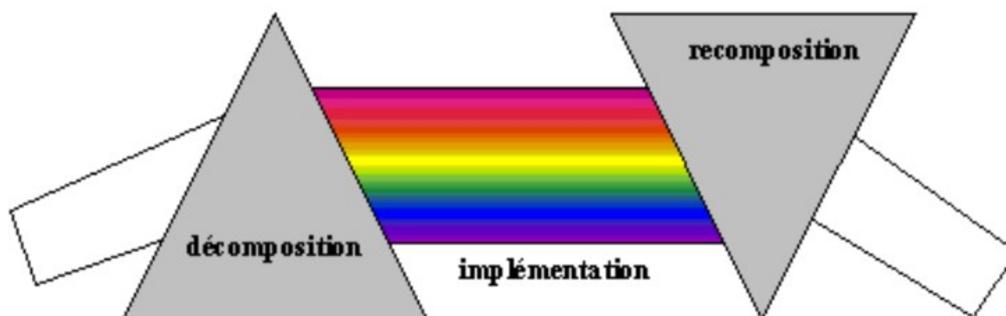


FIGURE 1.9 : Le processus de développement d'un programme orienté aspect [60].

1.6 Les outils supportant le paradigme orienté aspect

Nous citons dans cette section une liste non exhaustive des outils supportant le paradigme orienté aspect :

AspectJ

AspectJ offre un tisseur statique qui peut opérer sur le bytecode Java, mais il accepte également de travailler directement sur le code source. C'est le leader des tisseurs Java, il bénéficie d'une stabilité et d'une maturité importantes. Son origine remonte à la fin des années 90 quand Gregor Kiczales et son équipe au PARC Xerox travaillaient déjà sur les principes fondateurs de l'AOP. Le projet AspectJ est une partie de la communauté open source Eclipse. Le temps de tissage de cet outil est court et son langage de tissage est très puissant. Il ne filtre aucune possibilité du langage Java. Le résultat du tissage est du bytecode Java complètement standard [41].

Java Aspect Component (JAC)

JAC offre un tisseur dynamique dont les aspects sont définis en 100% Java. Il ne nécessite l'usage d'aucun langage propriétaire. Le code Java des aspects dépend de classes et d'interfaces de JAC, ce qui pourrait rendre moins immédiat un portage sur d'autres tisseurs.

JAC est un outil open source dont le développement est dirigé par Renaud Pawlak et qui a rejoint le consortium ObjectWeb. Son support actif est assuré par la société Aopsys. Le langage de tissage de JAC peut s'exprimer dans deux formats différents (ACC, une syntaxe concise, ou XML) et s'avère assez puissant de par l'usage des Expressions Régulières GNU, mais son comportement dynamique permet également de tisser ou d'annuler les greffes au cours de l'exécution d'une application. Il est donc particulièrement adapté aux environnements évolutifs dans lesquels les applications doivent parfois changer de stratégie d'implémentation. Mais cette souplesse se paie en termes de performances, car JAC fait un usage intensif de l'API de Réflexion Java, ainsi que sur certaines limitations telle que l'impossibilité d'insérer de nouveaux attributs sur une classe.

En outre, JAC fut le premier tisseur à offrir une bibliothèque d'aspects techniques réutilisables : de la présentation à la distribution en passant par la gestion transactionnelle et la supervision [41].

AspectWerkz

AspectWerks offre un tisseur que l'on peut utiliser statiquement, au chargement des classes Java, ou encore dynamiquement. Son langage de tissage est syntaxiquement proche de celui d'AspectJ, mais reste moins puissant : certaines zones de greffes lui sont encore inaccessibles. Piloté par Jonas Bonér, ce projet est open source et bénéficie du sponsoring de la société Bea Systems.

Contrairement au AspectJ, AspectWerkz permet de développer les aspects en Java standard. Afin de limiter l'adhérence entre les aspects et le tisseur, AspectWerkz permet de décrire les tissages d'aspects soit dans des fichiers externes au format XML, soit sous forme de commentaires spéciaux dans le code des aspects. Sous AspectWerks, le temps de tissage est court.

Les performances des applications dépendent du choix de la méthode de tissage : le tissage statique donne avec AspectWerkz des temps d'invocation de méthodes environ

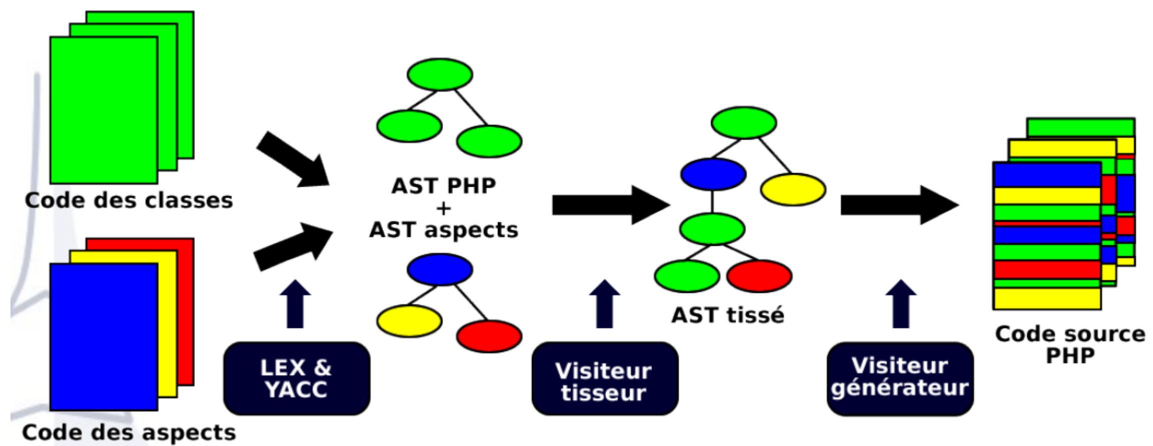


FIGURE 1.10 : La chaîne de tissage de l'outil PHPaspect [19].

deux fois plus rapides que le tissage dynamique. Dans les deux cas, les performances de l'application tissée sont moins bonnes que celles obtenues avec AspectJ [41].

PHPaspect

PHPaspect propose une solution pour supporter l'AOP en PHP5. Il offre une intégration forte avec le langage PHP. Son tissage est basé sur une analyse syntaxique (Lex & Yacc) et une analyse statique (AST) ce qui garantit de bonnes performances et une indépendance de la plate-forme (voir la figure 1.10). Il a 5 types de points de jonction : appel de méthode, exécution de méthode, construction d'objet, écriture et lecture d'attribut. Il offre des jokers et des opérateurs, réification des points de jonction et le mécanisme d'ordonnancement des aspects [19].

AspectS

AspectS est une implémentation de l'AOP en smalltalk. Il est basé sur la possibilité de smalltalk de développer des méthodes. Il est similaire au langage AspectJ et offre les différents concepts : advice, coupe, points de jonction. AspectS est un framework d'objet, permettant de tisser les aspects de manière dynamique. Tous les composants du langage AspectS sont des objects en smalltalk (voir la figure 1.11). A l'inverse d'AspectJ, les aspects sont instanciés explicitement [48].

AspectC++

AspectC++ est une implémentation de l'AOP en C++. Il a une syntaxe et une sémantique similaire à l'AspectJ (voir la figure 1.12) et son tissage a lieu au niveau du code source par un tisseur c++. AspectC++ combine la robustesse des aspects et des templates. Il offre un mécanisme très générique et efficace d'implémentations d'aspect et une injection transparente de *template-métaprograms* [65] .

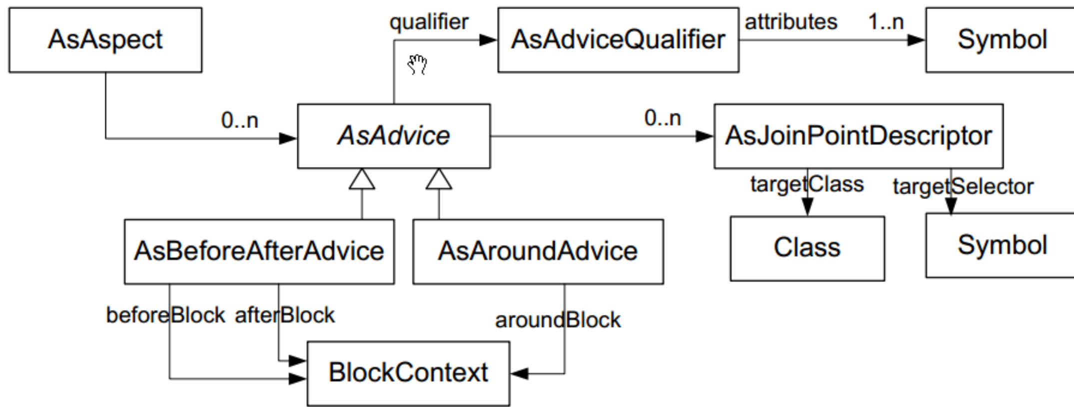


FIGURE 1.11 : La structure des aspects avec AspectS [48].

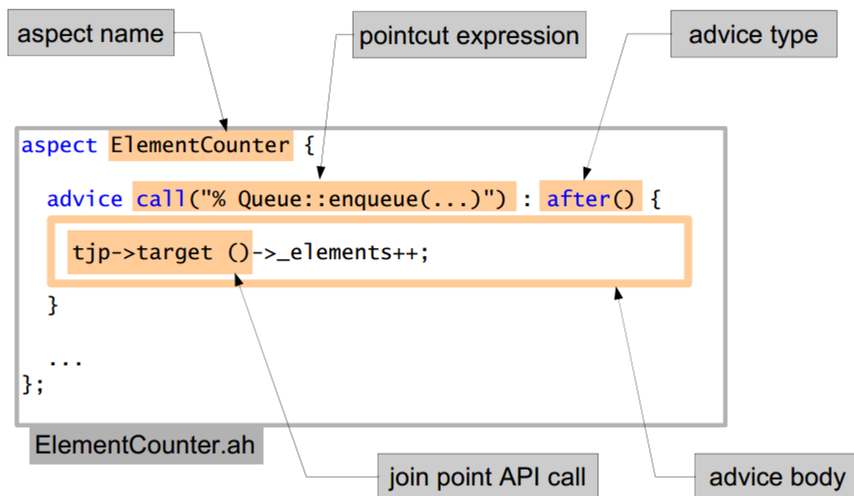


FIGURE 1.12 : Un exemple d'un aspect avec AspectC++ [65].

1.7 Programmer avec AspectJ

1.7.1 Aspects

Un aspect est une entité définissant un concept transversal aux objets métiers de l'application comme la persistance et la journalisation. Définir un nouveau aspect ressemble à la déclaration d'une nouvelle classe. Le mot-clé classe est seulement remplacé par le mot aspect. Un aspect peut inclure des coupes et des codes advice qui représentent les fonctionnalités liées à cet aspect. La figure 1.13 montre un aspect "DemoAspect" qui se charge d'afficher "getting data..." lorsque les fonctions qui débutent par "get" sont appelées.

1.7.2 Points de jonction

Les points de jonction sont les endroits du code de base de l'application où ils vont intégrer les fonctionnalités transversales. Elles se déterminent à l'aide de la spécification des coupes. Ces points peuvent être l'appel ou l'exécution d'une méthode, l'exécution d'un constructeur, la lecture ou l'écriture d'un attribut.

Plusieurs types d'évènements peuvent figure comme points de jonction :

1. *Méthodes* : puisque les méthodes forment l'outil principal de la programmation orienté objet, c'est autour des méthodes que les aspects se greffent le plus souvent. Les points de jonction liés aux méthodes sont : l'appel ou l'exécution de celle-ci. La spécification de ces points se fait dans l'aspect par **call(MethodSignature)** ou **execution(MethodSignature)** qui identifient tous les appels vers des méthodes dont le profil correspond à une description donnée par MethodSignature.
2. *Constructeurs.* Les constructeurs peuvent être considérés comme des méthodes particulières. AspectJ fournit quatre types de points de jonctions, pour les constructeurs : initialization, call, execution, et preinitialization.
   - **call(ConstructorSignature)** : on peut intercepter les appels des constructeurs avec le mot-clé **call**. Il suffit de fournir un profil correspondant à une description donnée par ConstructorSignature.
   - **execution(ConstructorSignature)** : l'exécution d'un constructeur constitue un point de jointure. Il suffit de fournir un profil correspondant à une description donnée par ConstructorSignature.
   - **initialization(ConstructorSignature)** : identifie toutes les exécutions d'un constructeur dont le profil vérifie ConstructorSignature.
   - **preinitialization(ConstructorSignature)** : identifie toutes les exécutions d'un constructeur hérité dont le profil correspond à l'expression ConstructorSignature. L'exécution d'un constructeur hérité se fait en Java avec l'aide du mot-clé **super(..)** où .. représente les paramètres fournis. Java impose que ce genre d'appel soit la première instruction d'un constructeur. Alors que *initialization* identifie l'exécution de constructeurs, *preinitialization* identifie l'exécution de constructeurs appelés via cette instruction.
3. *Exceptions* : les évènements de levée et de récupération d'exceptions peuvent aussi constituer des points de jonction. Le code à exécuter lors de la levée de cette exception sera défini une seule fois dans un aspect. La spécification de ces points se fait dans l'aspect par **handler (TypeSignature)** qui identifie toutes les récupérations d'exception dont le profil vérifie TypeSignature. Il s'agit donc, en Java de l'exécution

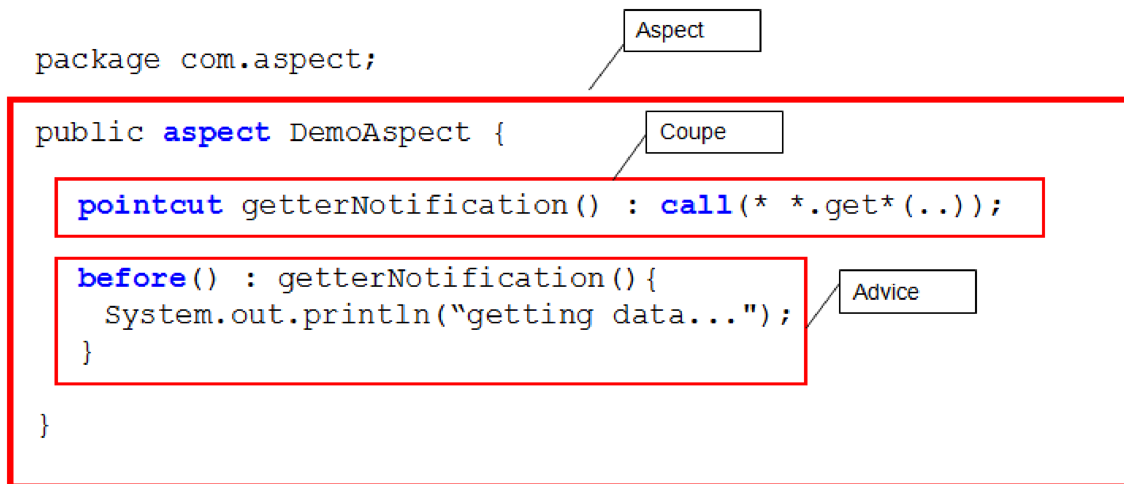


FIGURE 1.13 : DemoAspect.aj

des blocs catch. AspectJ ne permet pour le moment que l'utilisation de codes advice de type before sur les points de jonction de récupération d'exceptions.

4. *Attributs* : la lecture et la modification d'attributs constituent également des points de jonction. La spécification de ces points se fait dans l'aspect par **get(FieldSignature)** qui identifie tous les points de jonction représentant la lecture d'un attribut dont le profil vérifie FieldSignature et **set(FieldSignature)** qui identifie toutes les modifications d'un attribut dont le profil est compatible avec FieldSignature.
5. *Exécution de code advice* : identifie toutes les exécutions d'un code advice. Il ne requiert pas d'expression en paramètre, il identifie toute les exécutions de code advice. Ainsi, le code advice exécuté lors de l'exécution d'un code advice appartient, lui aussi, à l'ensemble défini par **adviceexecution**.
6. *Initialisation de classe* : **staticinitialization(TypeSignature)** Ce mot-clé identifie l'exécution des blocs statiques d'initialisation d'une classe correspondant à l'expression TypeSignature. Ces blocs sont exécutés automatiquement par la machine virtuelle lors du chargement de la classe.

Le tableau 1.2 donne un résumé des principaux points qui sont exposés par AspectJ.

A l'exception de **adviceexecution**, tous les mots-clés que nous avons vus requièrent un paramètre. Ce paramètre est une expression qui permet, en spécifiant un profil, de filtrer l'ensemble de points de jonction. L'expression précisant le profil des éléments qui nous intéresse peut faire usage de quantificateurs (appelés wildcards) afin d'introduire de la généricité dans les profils sélectionnés. Les wildcards offerts par AspectJ sont récapitulés dans le tableau 1.3.

On peut classer les profils en quatre types (patterns) :

— TypeSignature : Le terme type désigne collectivement les classes, les interfaces et les types primitifs. Dans AspectJ, le type se réfère également à des aspects. Ils peuvent utiliser des wildcards, des opérateurs binaires (|| et &) et l'opérateur unaire (!) e.x : *java..*Sun+* désigne Tous les types dans le java package et ses sous-paquetages directs et indirects qui ont un nom se terminant par Sun, et de leur sous-types.

— MethodSignature : précise le nom de la méthode, le type qui la déclare, le type de retour, le types d'arguments et modificateurs (ex : *public void Animation.get*(*)*).

— ConstructorSignature : il y a trois points différents entre le profil de méthode et celle de constructeur : premièrement, le constructeur ne permet pas la spécification de

TABLEAU 1.2 : Les points de jonction exposés par AspectJ basé sur [61].

Catégorie	**Point de jonction exposée**	**Description du point de jonction**
Method	Execution	Quand une méthode est exécutée
Method	Call	Quand une méthode est appelée
Constructor	Execution	Quand un constructeur est exécuté
Constructor	Call	Quand un constructeur est appelé
Field access	Read access	Quand un attribut non-constant d'une classe est référencé
Field access	Write access	Quand un attribut d'une classe est modifié
Exception processing	Handlers	Quand un traitement d'une exception est exécuté
Advice	Execution	Quand le code d'une advice est exécuté
Initialization	Class initialization	Quand l'initialisation statique d'une classe est exécutée
Initialization	Object initialization	Quand l'initialisation d'un objet est exécutée
Initialization	Object pre-initialization	Avant l'initialisation d'un objet

TABLEAU 1.3 : Liste des Wildcard et leur signification [17].

Wildcard	**Description**
*	Remplace un nom (de classe, de paquetage, de méthode, d'attribut, etc..) ou simplement une partie de nom. Il peut aussi remplacer un type de retour ou un paramètre. Il signifie " n'importe quel nom" ou " n'importe quel type".
..	Utilisé pour omettre les paramètres des méthodes ou le chemin complet des paquetages.
+	Permet de définir n'importe quel sous-type d'une classe ou d'une interface.

Nom de coupe

```
pointcut somePointCut() :
    call(* *.*(..));

before() : somePointCut(){
    System.out.println("Injecting advice...");
}

after() : call(* *.*(..)){
    System.out.println("Advice injected");
}
```

Paramètres de coupe

Utilisation de la coupe définie

Coupe anonyme

FIGURE 1.14 : Coupes nommé et anonyme sans paramètres.

valeur de retour, deuxièmement, les constructeurs n'ont pas de noms donc le nom des constructeurs sera remplacé par **new**, Du fait que les constructeurs ne peuvent pas être déclarés static, il n'est pas possible d'utiliser le mot-clé static. Par exemple, le constructeur public de la classe Animation, ne prenant aucun argument, aura la signature *publique Animation.new ()*.

— FieldSignature : il permet la spécification des attributs. Il spécifie le nom des attributs, le type qui déclare ces attributs et leurs modificateurs (les spécifications d'accès, statique, et finale), (ex : *private double Animation.distance*).

1.7.3 Coupes

Une coupe est introduite grâce au mot-clé ***pointcut***. Elle spécifie une collection de points de jonction et, optionnellement, de quelques paramètres précisant le contexte d'exécution au niveau de ces points. Pour identifier ces points de jonction, une coupe utilise plusieurs désignations. La syntaxe de coupe est :

```
pointcut nomDeLaCoupe(paramètres) : TypeDeCoupe (Profil) ;
```

Nous distinguons, principalement, deux types de coupes : *coupe nommé* qui a un nom et des paramètres et *coupe anonyme* qui est sans paramètres et nom. Cette dernière est utilisée généralement au niveau du code advice. La figure 1.14 illustre les deux types de coupe.

AspectJ offre un ensemble de type de coupes primitifs et il est également possible de généraliser ces coupes via des opérateurs logiques, il s'agit de coupes composées (voir le tableau 1.4). En effet, il est possible de combiner des points de jonction avec les opérateurs logiques (!, ||, &&).

Par exemple la coupe permettant de joindre les points de jonction défini par les deux **call** est :

```
pointcut allGetSet(): call (void *..set*(..)) || call (**..get*());
```

TABLEAU 1.4 : Les types de coupes offertes par AspectJ basé sur [61].

Syntaxe	**Description**
execution(MethodSignature)	Exécution d'une méthode dont le nom vérifie MethodSignature
call(MethodSignature)	Appel d'une méthode dont le nom vérifie MethodSignature
execution(ConstructorSignature)	Exécution d'un constructeur dont le nom vérifie ConstructorSignature
call(ConstructorSignature)	Appel d'un constructeur dont le nom vérifie ConstructorSignature
get(FieldSignature)	Lecture d'un attribut dont le profil vérifie FieldSignature
set(FieldSignature)	Ecriture d'un attribut dont le nom vérifie FieldSignature
handler(TypeSignature)	Exécution d'un bloc de récupération d'une exception (catch) dont le nom vérifie TypeSignature
adviceexecution()	Exécution d'un code advice
staticinitialization(TypeSignature)	Exécution d'un bloc de code static dans une classe dont le nom ConstructorSignature vérifie TypeSignature
initialization(ConstructorSignature)	Exécution d'un constructeur de classe dont le nom vérifie
preinitialization(ConstructorSignature)	Exécution d'un constructeur hérité dont le nom vérifie ConstructorSignature
withincode(ConstructorSignature) || withincode(MethodSignature)	Tout point de jonction tel que le code en court d'exécution est défini dans la méthode ou constructeur dont la signature est ConstructorSignature ou MethodSignature respectivement
within(TypePattern)	Tout point de jonction tel que le code en court d'exécution est défini dans le type TypePattern
this(TypePattern)	Vrai lorsque l'évènement se passe à l'intérieur d'une instance de classe de type TypePattern
target(TypePattern)	Vrai lorsque le type de l'objet destination du point de jonction vérifie TypePattern
Argst(Type, ..)	Choisir les points de jonction selon le type des arguments de points de jonction. Le nombre des arguments détermine le type de point de jonction : méthode, constructeur, exception-handler ou attribut ex : args(Account, .., int) désigne toute méthode ou constructeur où le premier argument est de type Account et le dernier argument est de type int
cflow(Coupe)	Vrai pour tout point de jonction situé entre l'entrée dans la coupe et sa sortie (y compris l'entrée et la sortie)

TABLEAU 1.4 : AspectJ coupes (suite).

Syntaxe	Description
cflowbelow(Coupe)	Vrai pour tout point de jonction situé entre l'entrée dans la coupe et sa sortie (sauf pour l'entrée et la sortie)
if(ExpressionBooléenne)	Evaluation de l'expression booléenne. Exemple : pointcut test() : if (thisJoinPoint.getArgs().length == 1) && execution(* *.*(..)) ;
! Coupe	Vrai si Coupe n'est pas satisfait
Coupe1 & Coupe2	Vrai si Coupe1 et Coupe2 sont satisfaits
Coupe1 || Coupe2	Vrai si Coupe1 ou Coupe2 ou les deux sont satisfaits
(Coupe)	Vrai si Coupe est satisfait

1.7.4 Advices

Une fois la coupe définie, il faut écrire le code à exécuter au moment où l'évènement décrit par la coupe est levé. Avant d'écrire ce code advice, il faut décider à quel moment exécuter le code : avant l'évènement, après ou autour de l'évènement. Les types de code advice proposés par AspectJ sont :

1. **Le type before** : un code advice est associé à une coupe. Cette coupe peut être utilisée par plusieurs codes advices si celle-ci est nommée. L'exemple suivant présente un code advice de type before dont la coupe est nommée :

```
/* définition de la coupe */
pointcut coupe(): execution(* *..*(..));
/* définition du code advice */
before(): coupe() {
System.out.println("Avant l'exécution d'une méthode");
}
```

   Ce code advice écrit la chaine de caractères "Avant un appel de méthode" avant chaque appel de méthode de l'application.
   L'exemple suivant défini le même comportement mais sans faire usage d'une coupe nommée :

```
/* définition du code advice */
before(): execution(* *..*(..)) {
System.out.println("Avant l'exécution d'une méthode en utilisant une coupe
    anonyme");
}
```

   Nous avons vu qu'une coupe pouvait être paramétrée afin de transmettre des informations au code advice. Pour ce faire, le code advice doit également être paramétré.

```
/* définition de la coupe */
pointcut example(Rectangle rec, Point p, int x, int y):
execution(* *..*(..)) && this(rec) && target(p) && args(x,y);
/* définition du code advice */
```

```
after (Rectangle rec, Point p, int x, int y):
{dessin(rec, p, x, y);
System.out.println(" Dessiné un rectangle par une advice paramétrée");
}
```

2. **Le type after** : alors qu'un code advice de type **before** s'exécute avant les points de jonction de sa coupe, un code advice de type **after** s'exécute après ceux-ci. Les deux types ont une syntaxe identique : il suffit de remplacer le mot-clé **before** par **after**.

```
/* définition de la coupe */
pointcut coupe(): execution(* *..*(..));
/* définition du code advice */
after(): coupe() {
System.out.println("Aprés l'exécution d'une méthode");
}
```

3. **Le type around** : les codes advices de type **around** exécutent du code avant et après les points de jonction. Les deux parties sont séparées dans le code advice par le mot-clé **proceed**. Ce mot-clé a pour but d'exécuter le point de jonction courant. Il se peut que **proceed** ne soit pas appelé par le code advice, ceci impliquant que le point de jonction ne sera pas exécuté.

   Contrairement aux types **before** et **after**, le type **around** possède un type de retour qui correspond au type de retour des points de jonction. Par exemple, le type de retour d'un point de jonction appel de méthode correspond à celui de la méthode, le type de retour d'un point de jonction écriture d'attribut est void. Il se peut également que les points de jonction de la coupe ne soient pas tous du même type. Il faut alors renvoyer un type qui soit un sur-type de l'ensemble des types des points de jonction. Les exemples suivants permettent d'illustrer les concepts présentés.

   L'exemple suivant montre que l'exécution de *proceed()* génère une valeur, la valeur générée par le point de jonction.

```
Object around(): ... {
System.out.println("avant le point de jonction");
Object distance=proceed();
System.out.println("après le point de jonction");
return distance;
}
```

   L'exemple suivant illustre l'utilisation d'un code advice de type **around** dans le cas d'une écriture d'attribut. Cette écriture étant une instruction qui ne renvoie rien, le type de retour du code advice est void.

```
void around(): get(double Animation.distance) {
System.out.println("avant la lecture du champ distance");
proceed();
System.out.println("aprés la lecture du champ distance");
}
```

4. **Le type after returning** : Les codes advices de type **after returning** s'exécutent à chaque terminaison normale d'un des points de jonction de la coupe. Il est possible de récupérer la valeur retournée (si elle existe) en paramétrant le code advice. C'est ce qu'illustre le code suivant :

```
after() returning (double distance): call(double Animation.method1(..)) {
System.out.println("method1(..) a retourné : " + distance);
}
```

5. **Le type after throwing** : Les codes advices de type **after throwing** s'exécutent à chaque terminaison anormale d'un des points de jonction de la coupe. Il est possible de récupérer l'exception levée en paramétrant le code advice. C'est ce qu'illustre le code suivant :

```
after() throwing (Exception e): execution(double Animation.method1(..)) {
System.out.println("method1(..) a levé l'exception : " + e);
}
```

1.7.5 Mécanisme d'introduction

Il est possible d'étendre les classes, les interfaces, et les aspects d'une application en leur ajoutant divers éléments. L'introduction de ces éléments se déclare à l'intérieur d'un aspect. En effet, l'aspect déclare des éléments pour le compte d'autres type (classes, interfaces même des autres aspects). C'est ce que l'on dénomme une **déclaration inter-type**. On doit faire attention à ne pas introduire un élément déjà existant dans le type visé. AspectJ classe ces éléments en cinq types : attribut, méthode, interface héritée, classe héritée et interface implémentée.

Attribut

Cette déclaration se fait comme dans une classe si ce n'est qu'il faut préfixer le nom de l'attribut par le nom du type visé. Sans ce préfixe, l'attribut serait un champ de l'aspect. Par exemple, l'aspect suivant ajoute un attribut id à la classe Class :

```
public aspect AjoutChamp {
private int Class.id;
}
```

Méthode

L'introduction de méthode est semblable à l'introduction d'attribut. Elle se réalise comme la définition d'une méthode au sein d'une classe, sauf que l'on ajoute comme préfixe le nom du type cible.

```
public aspect AjoutMembre {
private int Class.id;
public int Class.getId() { return id; }
}
```

Classe héritée

AspectJ permet également de modifier la hiérarchie d'héritage des classes. A l'aide du mot-clé *déclare parents*. Il offre la possibilité de rendre une classe héritière d'une autre. Le nom de la classe à modifier peut contenir des wildcards, ce qui rend possible la modification de la hiérarchie d'un ensemble de classe. Dans l'exemple suivant, nous avons défini la Class1 hérite de la Class2.

```
public aspect heritage {
declare parents: Class1 extends Class2;
}
```

Interface héritée

AspectJ permet également de modifier la hiérarchie d'héritage des interfaces à l'aide du mot-clé *déclare parents* ex :

```
public aspect heritage {
declare parents: Interface1 extends Interface2;
}
```

Interface implémentée

Le mécanisme d'introduction d'interface implémentée se réalise à l'aide du même mot-clé *implements*. Il est toujours possible d'ajouter une interface à une classe. L'exemple suivant ajoute l'interface Interf à la classe Class et à toutes ses sous classes, mais également à toutes les classes dont le nom contient foo.

```
public aspect interface {
declare parents: Class+ implements Interf;
declare parents: *foo* implements Interf;
}
```

1.7.6 Modifier le comportement du compilateur

AspectJ offre des mécanismes afin de signaler des erreurs et des avertissements au cours du temps de tissage.

L'élément *declare error* est un moyen utilisé afin de signaler une erreur quand le compilateur détecte la présence des points de jonction correspondants à une coupe bien déterminée. Le compilateur affiche le message précisé par le programmeur et interrompt le processus de compilation. La syntaxe de cet élément est la suivante [61] :

```
public aspect EnforceLogging{
declare error : <pointcut> : <message>;
```

En outre, l'élément *declare warning* est un moyen pour signaler les avertissements de compilation, mais il n'interrompt pas le processus de compilation. La syntaxe de cet élément est la suivante :

Problems ✕ | Javadoc | Declaration | Console

0 errors, 16 warnings, 0 infos

Description	Resource	In Folder	Locati...
don't print, use the logger	AspectExample1.aj	HelloWorld/com/aspect	line 37
don't print, use the logger	AspectExample1.aj	HelloWorld/com/aspect	line 38
don't print, use the logger	DemoAspect.aj	HelloWorld/com/aspect	line 7
don't print, use the logger	DemoAspect.aj	HelloWorld/com/aspect	line 8
don't print, use the logger	GetSetAspect.aj	HelloWorld/com/aspect	line 7
don't print, use the logger	Example1.java	HelloWorld/com/example1	line 30

FIGURE 1.15 : Les messages affichés par le compilateur lors de la détection des points de jonction.

```
public aspect EnforceLogging{
declare warning : <pointcut> : <message>;
```

Lors d'exécution de code suivant, le compilateur affiche le message "Avertissement !!!" lors de la détection des points de jonctions spécifiés par la coupe scope(). La figure 1.15 montre le résultat affiché par AspectJ :

```
public aspect EnforceLogging{
  pointcut scope(): execution(communication.*);

  pointcut printing():
    get(* System.out) ||
    get(* System.err);
declare warning : scope() : "Avertissement!!! ";
```

1.7.7 Ramollissement des exceptions

Java spécifie deux catégories d'exceptions qu'une méthode peut lancer : *checked* et *unchecked*. Le ramollissement d'exceptions permet de traiter les *checked* exceptions lancées par des coupes spécifiées par le programmeur comme *unchecked*. Pour amollir des exceptions sous AspectJ, on utilise l'élément *declare soft*, qui prend la forme suivante [61] :

```
declare soft : <ExceptionTypePattern> : <pointcut>;
```

L'exemple suivant peut amollir tous les exceptions de type IOExceptions et ses sous-types qui sont projetés par la classe Animation :

```
public aspect ExceptionWrapper{

  declare soft: IOException+: call( * Animation .*(..) )

  }
```

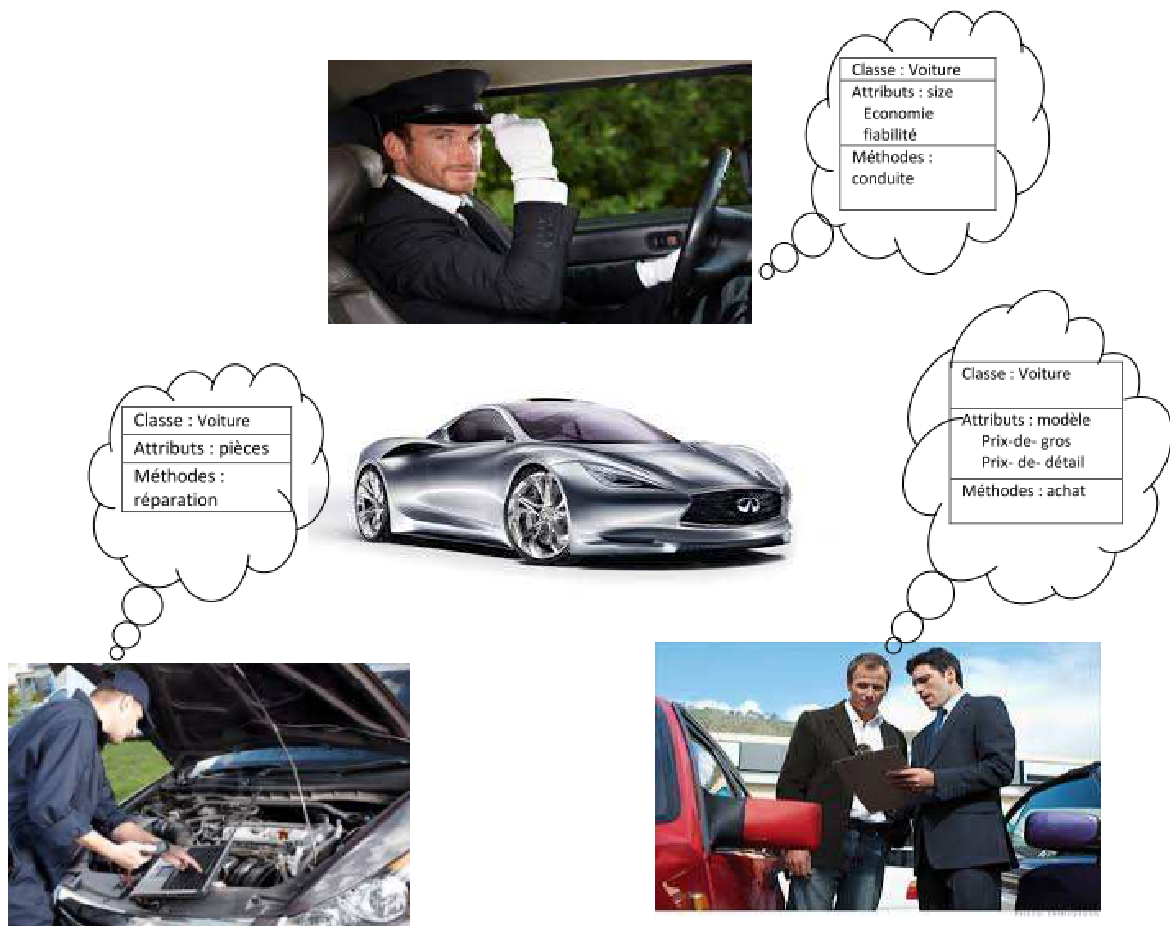


FIGURE 1.16 : Les points de vues subjectifs de l'objet voiture.

1.8 La programmation orientée sujet

1.8.1 Principe et objectifs

Le programmation orientée sujet a été introduite en 1993 au centre de recherche *Thomas J. Watson* d'IBM. C'est une extension du paradigme de la programmation orienté objet qui considère les systèmes comme un ensemble de sujets avec différents points de vue subjectifs. Un sujet modélise son domaine à partir de son propre point de vue particulier et il est mis en œuvre en utilisant des classes, variables d'instance et des opérations d'une manière orientée objet standard. La programmation orientée objet est bien adaptée à la construction des applications indépendantes mais moins bien adaptée pour la construction des suites ou familles d'applications intégrées [50]. Par exemple, une abstraction de voiture comme le montre la figure 1.16, peut être considérée selon le point de vue du conducteur, vendeur ou mécanicien. Chaque domaine a ses propres propriétés vitales de la voiture et a des exigences particulières sur le comportement qui peut affecter ces propriétés. Le conducteur peut classer les voitures en se basant sur la taille, l'économie, la fiabilité ce qui concerne principalement le comportement de conduite. Le vendeur peut classer les voitures en se basant sur la désignation du modèle ; le choix de voitures pour acheter et vendre dépend de la demande pour un modèle particulier, les prix de gros et de détail. Le mécanicien peut classer les véhicules en se basant sur les pièces et la disponibilité de l'outillage.

Plusieurs sujets peuvent être composés pour produire une suite complète d'applications. Aucun accès au code source, ni de recompilation n'est nécessaire pour effectuer la

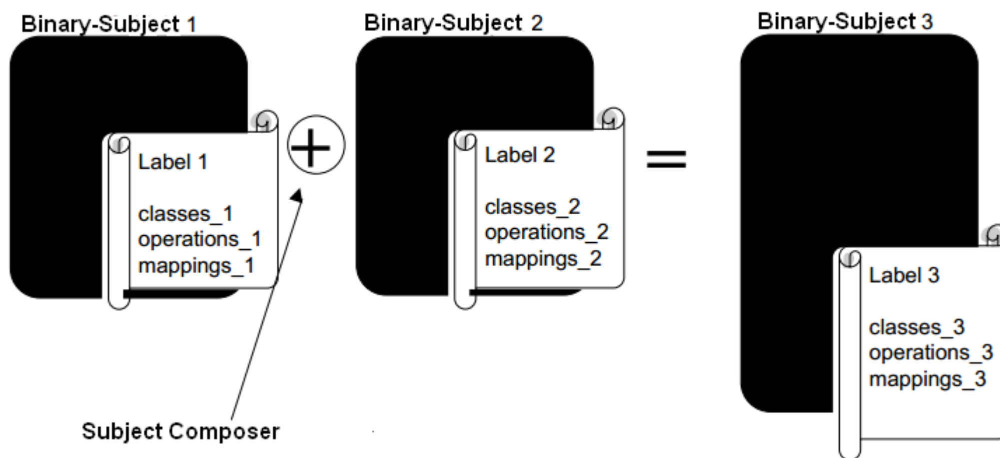


FIGURE 1.17 : Composition des sujets basée sur [11].

composition. Chaque sujet est compilé séparément pour produire un objet binaire qui comporte une étiquette fournissant des informations sur ce sujet, et d'un code binaire généré par le compilateur. Puis, le composeur orienté sujet utilise les informations qui se trouvent sur les étiquettes pour lier les sujets ensemble comme illustré sur la figure 1.17, sans aucune modification du code binaire des sujets individuels. Une grande variété de règles de composition est disponible pour déterminer la représentation et la sémantique d'une composition [72].

Ce paradigme offre des solutions adéquates à de nombreux problèmes dans le développement de logiciels orientés objet tel que [51] :

1. Développement de gros systèmes : concerne les applications interopérables ou intégrées qui évoluent de manière non planifiées sans avoir à modifier ou recompiler le code source existant.
2. Développement décentralisé : plusieurs équipes de différentes applications qui partagent des objets peuvent utiliser différentes vues de classes partagées séparément et ces vues subjectives peuvent ensuite être composées. Cela empêche les développeurs de travailler simultanément sur des parties imbriqués dans les mêmes classes.
3. Application et composition des patrons de conception. Dans la SOP, le code d'implémentation des patrons de conception pourrait être séparé du code auquel elles s'appliquent. En plus de cela, de nouvelles capacités de programmation sont ajoutées en tant qu'addition non-invasive de patrons de conception à tout moment et de la description des interactions de ces patrons à l'aide de règles de composition.
4. La décomposition du système sur la base des exigences : le code qui implémente une exigence ou fonctionnalité peut être construit comme un sujet cohérent, plutôt que d'être intercalé entre d'autres code de manière à le rendre difficile à identifier et à maintenir.

1.8.2 Hyper/J

Hyper/J est basé sur la programmation orientée sujet. À l'origine, il a été considéré comme un outil qui permet de spécifier un certain nombre de différents points de vue (ou dimensions et concerns selon la terminologie d'Hyper/J) d'un logiciel en correspondance avec les idées fondamentales de la programmation orientée sujet. Hyper/J est un

outil qui permet d'attribuer des classes et des méthodes spécifiées dans le langage de programmation Java à des **dimensions** et des **concerns** représentées par des identifiants correspondants, et de composer de telles **dimensions** et **concerns** à l'aide de quelques règles de composition prédéfinis. Ainsi, Hyper/J n'étend pas le langage de programmation Java et contrairement à AspectJ en fournissant de nouvelles constructions mais permet de préciser la composition en dehors des définitions de type habituelles dans des fichiers séparés. Ces fichiers sont utilisés pour composer l'application. Il y a trois types de fichiers de spécification différents dans Hyper/J [48] :

— **Hyperspace** : le fichier hyperespace contient toutes les classes qui sont considérés lors de l'exécution de la composition.

— **Concern Mapping** : le Concern Mapping attribue les éléments comme les classes, les méthodes et les variables d'instance aux dimensions.

— **Hypermodule** : le hypermodule spécifie les règles de composition qui sont appliquées à des classes dans le hyperespace.

1.8.3 Comparaison entre Hyper/J et AspectJ

Le modèle de programmation AspectJ soutient la dichotomie de base *aspect*. Les préoccupations transversales sont modularisées par aspects. La composition entre les fonctions de base et les aspects est définie en termes de points de jonction. Le comportement transversal peut être ajouté avant, après, autour, après throwing, après returning points de jonction. La composition est définie à l'intérieur des aspects. La précédence d'exécution entre les aspects est résolu implicitement (règles : avant, après, autour) ou explicitement (clause *dominates*).

Hyper/J supporte *hyperespaces*, une évolution des premiers travaux sur la programmation orientée sujet (SOP) qui ne distingue pas nécessairement entre la base et les préoccupations transversales. Les préoccupations sont modularisées utilisant *hyperslices*. Les règles de composition sont définies en termes d'unités correspondantes. Ces unités sont liées utilisant les relations *merge* et *override*. La composition est définie indépendamment des *hyperslices*. La précédence entre *hyperslices* est résolu explicitement (déclaration de *hyperslices* dans l'*hypermodule*, la clause de l'ordre). Le tableau 1.5 résume ces caractéristiques. Pour faire un passage des aspects à hyperlices, nous devons adopter un ensemble simple de transformation basée sur le travail de [21] :

1. Réécriture de chaque aspect sous forme d'une ou de plusieurs classes encapsulées par un hyperslice séparé.
2. Transformation du code advise en un code ordinaire (méthode).
3. Adoption des règles générales de composition *mergeByName*.

Un exemple simple présenté dans [74], considère un système de facturation (billing system) avec une classe centrale *Invoice*. Les factures peuvent être livrées en appelant la méthode *deliver()* dans la classe *Invoice*. Les instances de la classe *Invoice* sont créés dans plusieurs endroits du système et plusieurs sous-classes de *Invoice* existent. Nous allons prendre deux utilisateurs de *Invoice*, une sous-classe *SpecialInvoice* et *InvoiceClient* comme suit :

```
package billingsystem;
class Invoice {
void deliver() { ... }
...}
class SpecialInvoice extends Invoice {
```

TABLEAU 1.5 : Comparaison entre Hyper/J et AspectJ basé sur [21].

	AspectJ	Hyper/J
Préoccupation transversale	aspect	hyperslice
Endroit de composition	point de jonction	corresponding unit
Position relative de la composition	merge before, after, around, after throwing, after returning	merge ou override
La spécification de composition	à l'intérieur d'aspect	en dehors d'hyperslice
Priorité d'exécution	règles implicites et clause *dominates*	règle "order" et l'ordre de la déclaration des *hyperslices*
Moment de la composition	compilation ou exécution	compilation
Façons de mettre en oeuvre les changements	modification sur place	migration de client

```
...}
class InvoiceClient {
Invoice invoice = new Invoice();
void foo() {
invoice.deliver();}}
```

Supposons maintenant que, dans le cadre d'un passage à l'e-business, les factures devraient également être envoyés par e-mail avant la livraison normale. AspectJ et Hyper/J sont recommandés comme deux nouvelles approches prometteuses pour faire face à ce problème.

Nous pouvons aborder le problème avec AspectJ en définissant un aspect qui possède la méthode *sendMail()* et augmente la méthode *deliver()* de la classe *Invoice* par un appel à cette méthode :

```
package billingsystem;
class MailExtensionAspect {
introduction Invoice {
void sendMail() {
...
}
}
static advice(Invoice invoice): void print() & invoice {
before {
invoice.sendMail();
}
}
}
```

Le résultat d'une compilation de ces classes est une classe *Invoice* qui est équivalent à la rédaction des modifications appropriées directement dans la classe. Nous appelons ce type de modification d'une **modification sur place**.

Le même problème peut être traité avec Hyper/J en fournissant une classe indépendante nommée *extension. Invoice* pour ajouté l'option d'e-mail. L'ancienne classe *Invoice*

avec ses clients est ensuite fusionné avec l'extension dans un nouvel *hypermodule* (les spécifications des *concerns* et des *hyperslices* sont omis). Les définitions des *hyperslices* ci-dessous sont seulement des énumérations des classes intégrées :

```
package extension;
public class Invoice {
public void sendMail() {
...
}
public void deliver() {
sendMail();
}
}
hypermodule extendedbillingsystem
hyperslices:
Feature.billingsystem, // consists of billingsystem.*;
Feature.extension; // consists of extension.*
relationships:
mergeByName;
end hypermodule;
```

Après avoir effectué la composition, le package *extendedbillingsystem* contient les classes *Invoice*, *SpecialInvoice* et *InvoiceClient*. Le point intéressant dans ce cas est que *SpecialInvoice* et *InvoiceClient* utilisent *extendedbillingsystem.Invoice* là où était *billingsystem.Invoice* utilisé précédemment. Au lieu de changer la classe *Invoice* originale, de nouveaux clients sont créés qui se réfèrent à une nouvelle classe *Invoice*. Ce type de transformation du programme est appelé **migration de client** .

1.8.4 Relations entre AOP et SOP

Etat de l'art

Parce que l'AOP et la SOP sont de nouveaux paradigmes, par rapport à la programmation orientée objet, et comme ils sont en constante évolution, seule une quantité limitée de travaux de recherche est dédiée à l'évaluation et la comparaison des deux paradigmes. Dans ce qui suit, une brève description de la plupart des travaux existants est présentée [28] :

Dans le travail de [10], une comparaison entre l'AOP et les approches orientées objet connexes comme SOP, rôle, les techniques split et Crome a été réalisée. L'auteur a souligné le principe commun entre ces approches qui est la séparation des préoccupations transversales. D'un côté, dans la terminologie AOP, les membres transversaux sont des aspects et des composants ; les points d'interaction sont les points de jonction. De l'autre côté, dans le paradigme de SOP, les unités modulaires sont des sujets et des objets ; les points d'interaction sont des parties de description en sujets. Dans cette étude, la différence entre AOP et les autres approches est négligée.

Dans [84], l'auteur a révélé un aperçu sur les similitudes et les différences entre la SOP et l'AOP en termes de leurs hypothèses sur le processus de développement de logiciels grâce à une étude pratique. Les expériences réalisées dans cette étude n'utilisent que des petits programmes ce qui ne peut pas donner une idée claire sur les problèmes qui peuvent survenir lorsque les approches sont utilisées dans des projets de développement réels. En plus, les outils comme le compilateur de sujets, Watson (WSC) et AspectJ donnent une idée sur les avantages qui peuvent être obtenus en adoptant ces approches,

mais ne prennent pas en charge toutes les fonctionnalités proposées par leurs techniques respectives.

Le travail de [85] clarifie deux concepts centraux et communs entre AOP et SOP. Le premier est la correspondance qui spécifie les constructions qui pourraient être utilisés comme points de référence pour le processus de composition. Le deuxième l'unification est le processus par lequel les constructions correspondantes sont combinées. Dans cette prise de position, un modèle de composition formelle pour les deux approches, AOP et SOP a été proposé.

Dans le travail de [92], une étude sur les paradigmes de programmation récents tel que la programmation orientée aspect et la programmation générative a été présentée pour tenter de trouver le meilleur paradigme. L'auteur considère SOP, AOP, les filtres de composition, et la programmation adaptative comme des approches de la programmation orientée aspect sans aucune explication. En outre, il a présenté un exemple simple *tracking access* pour démontrer la correspondance entre AOP et SOP.

Par ailleurs, une brève étude sur les nouvelles méthodes de programmation a été présentée par [31] dans laquelle affirme que l'AOP sépare les préoccupations non fonctionnelles contrairement à la SOP qui sépare les deux exigences fonctionnelles et non fonctionnelles.

Enfin, [68] ont utilisé une étude de cas pour comparer trois approches : AOP, SOP et view-oriented programming (VOP). L'étude de cas consiste à prendre un exemple d'application, et de le soumettre à deux scénarios de maintenance, l'un ajoute une exigence fonctionnelle, et l'autre ajoute une nouvelle exigence architecturale, à savoir, la distribution (ou "accès distant") des objets. Ils ont déduit que laVOP gère la séparation des préoccupations fonctionnelles tandis que l'AOP supporte la séparation des préoccupations non-fonctionnelles. La SOP supporte les deux types des exigences fonctionnelles et non fonctionnelles.

Similarités

1. Les deux approches ont un objectif commun pour résoudre les problèmes dans le développement de logiciels relatifs à empêcher la dispersion et l'enchevêtrement du code, et en divisant le système dans des modules distincts pour atteindre la séparation des préoccupations (principe SoC).
2. Dans les deux méthodes, le système passe par le même cycle de vie. Tout d'abord, la décomposition du système en plusieurs modules (selon le principe SoC). Ensuite, l'utilisation de plusieurs mécanismes pour composer des modules du système.

Différences

1. Dans la SOP, un système comprend un certain nombre de modules (sujets) qui peuvent contenir des préoccupations fonctionnelles ou non fonctionnelles. D'un autre côté, dans l'AOP, un système en cours de développement comprend le code de base (module fonctionnel) et un certain nombre d'aspects qui contiennent du code non-fonctionnel. L'AOP est basé sur le principe de centralisation.
2. Dans la SOP, les informations sur les constructions qui sont impliquées dans la composition ; et la manière par laquelle ceux-ci devraient être combinées sont spécifiées hors des sujets. Elles sont séparées dans un fichier indépendant sous forme de règles de composition. Par ailleurs, dans l'AOP les places d'injections des préoccupations transversales sont spécifiées à l'intérieur des aspects grâce aux mécanismes des coupes et des modificateurs des advices comme le montre la figure 1.18.

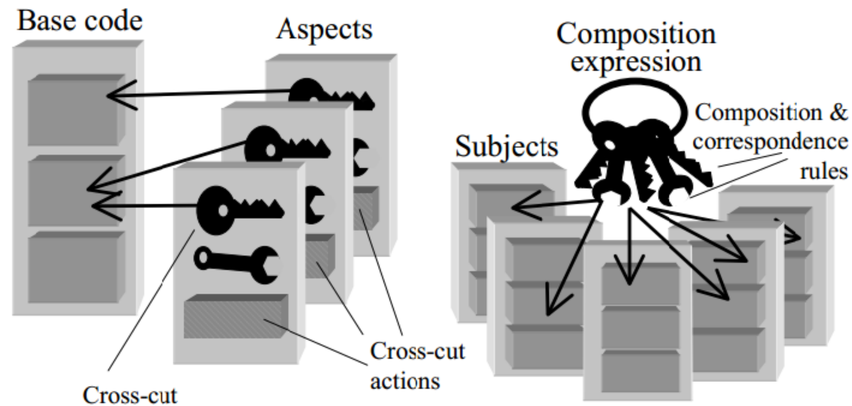


FIGURE 1.18 : Le mécanisme de composition dans l'AOP et la SOP [84].

TABLEAU 1.6 : Modèle de correspondance entre l'AOP et la SOP [28] .

AOP	SOP
Aspect	Sujet
code fonctionnel	Sujet
Le mécanisme de tissage	les règles de composition
la spécification des coupes	la spécification des étiquettes
points de jonctions	les parties d'un sujet

La mise en correspondance(mapping) entre l'AOP et la SOP

L'idée derrière le mapping consiste à répondre à la question : Pour chaque spécification donnée dans l'une des approches, quelle est la spécification correspondante dans l'autre ? Nous présentons ici, notre modèle de mapping entre AOP et SOP. Un système selon l'approche SOP correspond à un ensemble de sujets et de règles de composition pour les composer. On suppose que SU soit un système de la SOP :

$SU = \langle S, C \rangle$

S : est un ensemble de sujets, chacun représente un sous-système de SU.

C : ensemble de règles de composition utilisées pour combiner les sujets.

Si nous limitons SU à un seul système, l'ensemble S devient les exigences non fonctionnelles et fonctionnelles de ce dernier. La contrepartie de SU dans AOP est $\langle A, F \rangle$ où :

A : un ensemble d'aspects qui représentent la partie non fonctionnelle.

F : représente un ensemble de classes qui implémentent la partie fonctionnelle d'un système.

Le mapping est $S = A \cup F$.

C : est traduit en advices et coupes qui peuvent être ajoutées au code des aspects, A. De plus, le code des advices doit être omis du code fonctionnel, F.

Généralement, le mapping de la SOP vers l'AOP n'est possible que si la SOP représente une seule application. Cela signifie que l'AOP est un cas particulier de la SOP.

Le tableau 1.6, présente le résumé de notre modèle de mapping des deux approches fondées sur leurs principaux concepts.

1.9 Conclusion

Dans ce chapitre, nous avons présenté les approches du paradigme de la programmation orientée aspect. Nous avons expliqué les principes, les concepts et les avantages de l'AOP comparés au paradigme orienté objet. Il ressort de cette présentation que ce paradigme, relativement nouveau, représente une évolution certaine dans le domaine de la modélisation et de la séparation avancée des préoccupations en général. En plus, nous traitons un exemple d'implémentation de l'AOP avec AspectJ. En outre, nous avons présenté dans ce chapitre le paradigme de la programmation orienté sujet et nous avons passé en revue ses principes, ses concepts et ses avantages comparés au paradigme orienté objets. En outre, nous avons expliqué la relation entre celle-ci et la programmation orienté aspect du point de vue théorique et pratique par l'explication de la relation entre les deux leader Hyper/J et AspectJ. On peut conclure que la programmation orienté sujet est une collection de bonnes idées qui sont complexes à implémenter à cause de l'absence des supports d'implémentation matures, mais l'AOP hérite de cette théorie et peut être considérée comme un cas particulier. L'AOP a réussi à exploiter une portion des idées de la SOP en se focalisant seulement sur la séparation des préoccupations transversales des applications. Cela l'a permis de perfectionner ces propre outils tel qu'AspectJ. Le prochain chapitre dresse un panorama des approches de simulation orientées-aspects et met l'accent sur l'utilisation de ce paradigme dans le domaine simulation.

Deuxième partie

Proposition

Chapitre 2

La gestion des préoccupations transversales dans un système de simulation à évènements discrets

> *« Deux choses sont infinies : l'univers et la bêtise humaine ; en ce qui concerne l'univers, je n'en ai pas acquis la certitude absolue »*
>
> Albert Einstein

Sommaire

2.1 Introduction **43**

2.2 La simulation à évènements discrets **43**

2.3 État de l'art des approches de modélisation et de simulation orientée aspect **44**

2.4 Les principales préoccupations transversales d'un système à évènements discrets **48**

2.4.1 La détection de l'état d'équilibre 48

2.4.2 L'animation graphique 48

2.4.3 L'interface utilisateur graphique 49

2.4.4 Le traitement des exceptions 49

2.4.5 L'exactitude des calculs 49

2.4.6 La trace de simulation 49

2.4.7 La synchronisation 49

2.5 Le processus de mise en œuvre d'un système de simulation orientée-aspect **50**

2.6 Conclusion **50**

2.1 Introduction

Les projets de simulation à évènements discrets implémentent plusieurs préoccupations transversales, telles que l'ordonnancement d'évènements, la gestion d'évènements et l'enregistrement de la trace d'une simulation, qui ont tendance à produire un enchevêtrement et une dispersion de code. Cela augmente la complexité et réduit la maintenabilité qui exige une séparation spécifique des préoccupations (SoC). Le paradigme de la programmation orientée aspect met d'avantage l'accent sur les préoccupations transversales que les autres paradigmes classiques comme la programmation orientée objet. Il fournit des mécanismes qui capturent explicitement les préoccupations transversales de façon modulaire et d'atteindre ainsi les avantages qui résultent de l'amélioration de la modularité ; un code plus facile à concevoir, mettre en oeuvre, maintenir, réutiliser et faire évoluer. Dans ce chapitre, nous présentons la plupart des travaux qui utilisent ce paradigme de programmation dans le domaine de la modélisation et de la simulation. En outre, nous identifions les principales préoccupations transversales d'un simulateur à évènements discrets et le processus de mise en oeuvre de ce paradigme.

2.2 La simulation à évènements discrets

En considérant la dimension temporelle, trois types de simulation sont identifiés : la simulation Monté-Carlo pour les systèmes statiques, la simulation continue et la simulation à évènements discrets pour les systèmes dynamiques. La première pour les systèmes qui changent constamment et l'autre pour ceux qui changent de façon discrète. Un système à évènements discrets change d'état en l'occurrence d'évènements. Chaque simulateur à évènements discrets nécessite au moins une partie des composants suivants [64] :

- ▷ L'ordonnanceur d'évènement (Event sheduler) : il gère la liste de tous les évènements en attente en activant ou en suspendant les routines associés au moment approprié. En outre, il met à jour l'horloge de simulation.
- ▷ L'horloge de simulation : elle mémorise le temps de simulation courant. Elle pourrait être mis à jour par l'ordonnanceur en fonction d'un incrément progressif fixe ou à l'apparence des évènements.
- ▷ Traitement des évènements (Event processing) : Chaque évènement a ses propres routines de traitement qui représentent ce qui se passe lorsque l'évènement se produit. ces routines peuvent changer l'état global ou générer des évènements supplémentaires qui doivent être insérés dans la liste d'évènements de l'ordonnanceur.
- ▷ Les mécanismes de génération des évènements (Event generation mechanisms) : Il existe trois techniques pour générer des évènements : *execution driven*, *trace driven* et *distribution driven*.
- ▷ L'enregistrement des données et les routines de récapitulation (Data recording and summarization routines) : En plus de maintenir les variables d'état, le simulateur doit également tenir les mesures temporelles et les enregistrements d' évènements. Ces valeurs sont utilisées pour calculer les statistiques qui résument les résultats de la simulation.

Pour construire un modèle de simulation à évènements discrets, le concepteur doit choisir une approche de modélisation. Dans la pratique, il existe trois grandes approches pour modéliser un tel système [9] :

- ▷ L'approche par interaction de processus : c'est la plus intuitive. Elle se compose d'un ensemble de processus en interaction. Chaque processus modélise le cycle de vie d'un objet du système. Un processus est une séquence bien ordonnée d'activités qui sont logiquement liées.
- ▷ L'approche par évènements : elle définit tous les évènements pertinents et les modifications liées (actions) qui doivent avoir lieu pour chacun.
- ▷ L'approche par activités : elle définit toutes les activités qui peuvent être réalisées par les objets du système et les actions à exécuter au début et à la fin de chaque activité, ainsi que la durée de chacune.

2.3 État de l'art des approches de modélisation et de simulation orientée aspect

Un résumé des principaux travaux existants dans la littérature qui utilisent l'AOP dans le domaine de la simulation à évènements discrets est fourni par [1] :

Simkit est un outil open source basé sur le paradigme orienté-objet et le formalisme *event graph* pour la modélisation. Les auteurs dans [2] proposent une version orientée aspect basée sur AspectJ pour séparer les préoccupations transversales telles que les règles de suspension de la simulation et de restauration d'exécution d'une simulation. Cependant, ils n'arrivaient pas à séparer toutes les préoccupations transversales, notamment celles qui pourraient être trouvés dans n'importe quel framework mature tel que la détection de l'état d'équilibre et l'interface utilisateur. Le travail de [80] sur le framework OSIF qui est à base de composants utilise le paradigme AOP pour séparer les modèles DES du cadre expérimental afin de permettre la réutilisation des logiciels et faciliter leur évolution. En outre, l'AOP a été utilisé dans le développement de l'Open Simulation Architecture (OSA) [79] dans le même but que dans OSIF. Elle permet une meilleure réutilisation des composants des deux côtés, la réutilisation d'un modèle donné avec divers scénarios, ou la réutilisation d'un scénario donné avec divers modèles afin de gagner du temps, de l'argent, et d'effort humain. Un cas d'utilisation simple concernant l'étude de la sécurité d'un réseau a été utilisé pour illustrer les avantages de la technique précédente.

SimJ est un framework académique ne contenant que la préoccupation transversale log (la gestion des droits). Les auteurs dans [90] confirment leur hypothèse dans laquelle 20 patrons de conception AO diminuent la complexité du code, éliminent la dispersion de code, et permettent de concevoir des *hot-spots* AO supplémentaires dans les frameworks. La perte de performances est minime et tout à fait acceptable, ce qui est illustré par le framework SimJ et l'utilisation de la version AO de patron de conception *adaptateur*. Les auteurs considèrent que SimJ est une application ordinaire sans donner aucune importance à son domaine (simulation à évènements discrets). Dans [93], les auteurs ont présenté leur plateforme de simulation *Tortuga* basé sur Java. Ils ont utilisé l'AOP seulement pour mettre en œuvre l'aspect *synchronisation*.

Dans [46], un système de simulation multi-agents est discuté. Il se compose de deux types d'agents, un ensemble pour décrire le modèle de simulation et un autre pour les mécanismes d'observation. Cette collection d'agents indépendants interagit par des évènements discrets où chaque agent a un calendrier qui génère son plan d'activités. Le système est exécuté sur une plateforme qui utilise le paradigme OO pour définir ses modèles d'agents et de la technologie web pour interagir avec l'environnement de modélisation et de simulation. Le noyau de cette plateforme est le langage de programmation *MAML* (Multiagent Modeling Language) qui a la capacité de la dissociation entre le modèle et

les mécanismes d'observation grâce au paradigme orientée aspect à partir des phases de conception jusqu'à la phase de mise en oeuvre. Cela augmente la maintenabilité du système et diminue sa complexité. *MAML* doté du compilateur *xmc* qui génère du code *Objective-C* à partir du code source *MAML* après tissage de l'objet du modèle et l'objet de l'observation. Malgré la richesse de la syntaxe de *MAML* pour supporter l'AOP, il reste moins performant et moins riche par rapport à AspectJ.

Dans [13], le paradigme AO a été utilisé pour développer un système multi-agent dédié à simuler des phénomènes physiques. Le système *MAFES* (Multiagent Finite Environment System) se compose d'un environnement sous la forme d'une matrice de noeud et un ensemble d'agents opérant sur ces noeuds. Les aspects sont utilisés pour assigner des tâches à des agents en ajoutant des fonctionnalités appropriées pour accomplir leur tâche. *MAFES* contient trois autres types d'aspects pour le contrôle, la visualisation et le stockage des résultats de simulation. La mise en oeuvre de *MAFES* est basée sur le langage AspectJ et rend le système générique afin de construire des versions pour des conditions spécifiques (il suffit de tisser les aspects appropriés).

Dans [55], une nouvelle approche orientée aspect pour le système de simulation de la prévention des catastrophes (ABR) a été proposée. L'approche sépare la fonctionnalité de base de l'application de simulation des préoccupations transversales de simulation grâce à la méthode de décomposition horizontale (HD) qui repose sur le paradigme d'AOP. L'approche est mise en œuvre dans *AOSIF* (Aspect-Oriented Simulation Framework) qui est une extension du framework de la simulation distribuée (DiSiF) [77]. Il utilise la modélisation de workflow basé sur l'acteur, les services web et le grid computing comme technologie d'implémentation et les annotations Java pour la programmation déclarative, en plus d'Aspectj pour la mise en oeuvre orientée aspect. Pour démontrer l'applicabilité de l'approche, deux préoccupations transversales, à savoir, *intégration tool* et la *distribution*, sont implémentées. Malheureusement, les préoccupations ne sont pas spécifiques au domaine de la modélisation et la simulation.

Dans [18], les auteurs présentent un système de simulateur de conduite qui utilise le paradigme de l'AOP au niveau code. Le simulateur a des préoccupations transversales tel que la synchronisation et l'ordre d'exécution, interface utilisateur (IU), et la journalisation qui sont écrits dans différentes langages d'aspects spécifiques (ASLs). Il dispose d'un mécanisme de tisseur d'aspects modulaire qui offre la généralité d'un langage d'aspect à usage général sans perdre la capacité et les avantages de la définition d'aspects dans des langages spécifiques d'aspect.

Dans [89], un simulateur de trafic en temps réel AO a été développé. Il utilise l'AOP pour encapsuler sept préoccupations transversales en temps réel en utilisant AspectJ : ordonnancement des threads et routage, le partage des ressources et la synchronisation, terminaison asynchrone de thread, la gestion de la mémoire, l'accès à la mémoire physique, la gestion des évènements asynchrones, et le transfert asynchrone de contrôle. Une comparaison des deux systèmes, un simulateur de trafic en temps réel et son équivalent orientée aspect, est effectuée pour mettre en lumière les avantages et les inconvénients de l'utilisation de l'approche AO. Elle est basée sur la suite de métriques OO de Chidamber et Kemerer (C& K). Les préoccupations sus-mentionnées sont d'ordre général, elles pourraient être trouvés dans tous les systèmes temps réel. En outre, malgré le fait que la suite C& K est adaptée aux systèmes AO, une étude plus approfondie est nécessaire sur les métriques qui sont appropriées pour l'évaluation des applications AO.

Dans [75], une méthode d'analyse des performances au niveau de la conception basée sur la simulation a été proposée. La préoccupation 'performance' de l'application est séparée du modèle fonctionnel dès la phase de conception grâce à l'utilisation de la pro-

grammation orientée-aspect. Contrairement à la conception classique d'un système logiciel qui est modélisé en utilisant des diagrammes de classe UML et des diagrammes de séquence, le modèle de performance est une représentation basé sur XML provenant du profil de performance UML. Après la génération de code à partir du modèle de conception, le tisseur d'AspectJ est introduit afin de formuler le code de simulation. Les auteurs ont expérimenté leur approche en utilisant un système distribué. Dans [76], les auteurs affirment que leur approche est générique et peut être utilisée pour l'analyse des autres attributs de qualité d'un système tel que la fiabilité. Dans [33], une autre approche pour analyser la performance est proposée. Contrairement à l'approche précédente, on définit la performance comme une collection d'aspects qui comprend une longue liste de paramètres tels que le temps de réponse, la probabilité et le temps entre les erreurs sur le framework orientée-aspect d'analyse formel de conception (FDAF). Les auteurs se concentrent sur la modélisation de l'aspect "temps de réponse" de la performance en se basant sur real-time UML comme notation de base, traduite ensuite en *Architecture Description Language (ADL) Rapide*. Les auteurs utilisent la technique de simulation pour évaluer le temps de réponse pour le sous-système de traitement des requêtes DNS.

Dans [69], les auteurs proposent un framework orienté aspect pour les systèmes multi-agent qui sépare la préoccupation transversale *performance* au niveau de la conception en utilisant le langage aSideML, qui est une extension UML pour représenter les aspects à différents niveaux d'abstraction. Ce framework fournit la séparation de la préoccupation transversale performance parmi les différentes propriétés d'agents (mobilité, autonomie, adaptabilité, interaction et apprentissage) et les préoccupations spécifiques de scénarios de l'application. Le modèle de conception pour le scénario des préoccupations spécifiques de l'application et des agents ont leur propre code d'implémentation Java tandis que le modèle de performance a une implémentation AspectJ séparée. Plus tard, ils sont tous tissés ensemble en utilisant le compilateur AspectJ. L'architecture du framework est finalement constituée de la composante "performance", les préoccupations d'agents, le composant de la plate-forme agent, les préoccupations "*workload*" et "*resources*", l'interface *IInformatioGathering*, et l'interface *IperformanceReporting*.

Dans [83], les auteurs proposent une nouvelle approche qui étudie les effets de la performance des aspects transversaux comme la "sécurité", sur la performance globale du système. L'approche procède par l'ajout d'annotations de performance à la fois aux modèles primaires et au modèle d'aspects en utilisant le profil de performance d'UML. Ensuite, le modèle d'aspects générique est instancié en un contexte spécifique en suivant un ensemble de règles prévues par le concepteur, qui transforme les annotations paramétriques du modèle d'aspects générique en un modèle concret. Ce dernier est combiné avec le modèle primaire selon un ensemble de directives. Le résultat est un modèle UML annoté et composé qui peut être transformé automatiquement en un modèle de performance (Layered Queueing Networks (LQN) dans ce cas). Enfin, le modèle LQN est analysé avec les solveurs existants.

Dans [14], les auteurs discutent une nouvelle approche pour la séparation du comportement (qualitatif) fonctionnel des contraintes quantitatives de performance depuis la phase de spécification. Grâce à l'AOP, les aspects d'une spécification sont écrits dans différentes langages : l'algèbre de processus LOTOS pour une spécification abstraite du comportement fonctionnel et la logique temporelle probabiliste pour les aspects quantitatifs (contraintes de performance). Le tissage compose les deux spécifications d'aspects et le résultat est un modèle global de style automate qui peut être générée à partir de la composition d'un système de transition étiqueté (dérivé de LOTOS). Les planificateurs d'évènements sont dérivés des formules de la logique temporelle. Enfin, le modèle global

peut être utilisé pour l'analyse de la performance basée sur la simulation.

Comme un résumé de la discussion précédente, l'utilisation de l'AOP dans le domaine de la simulation à évènements discrets dépend de plusieurs considérations selon le paradigme orientée-aspect lui-même à savoir [23] :

1. Niveau d'application :
   - La phase de spécification.
   - La phase de conception.
   - Le niveau d'implémentation.
2. Technologie utilisée :
   - Logic Metaprogramming (LMP).
   - Model Driven Engineering (MDE).
3. Niveau de tissage d'aspects :
   - Niveau modèle.
   - Niveau code.
4. La méthodologie d'accueil :
   - L'AOP peut co-exister avec la méthodologie orientée objet.
   - Les systemes multi-agent (MAS).
   - La programmation par composants.

En outre, les systèmes de simulation orientée-aspect peuvent être classés d'un point de vue de la simulation, selon la nature des préoccupations transversales séparées. Ces dernières peuvent être soit spécifiques au domaine de la modélisation et de la simulation tel que la détection de l'état stationnaire, soit générales trouvées dans n' importe quelle application comme la gestion des exceptions.

Quelle que soit l'approche utilisée pour appliquer l'AOP dans le domaine de la simulation à évènements discrets, l'objectif principal est la conception de nouveaux systèmes basés sur le principe *Less Is More*. Maintenant, une méthode de séparation complète des préoccupations transversales de la fonctionnalité de base est primordiale. La fonctionnalité de base représente moins de code, mais plus de valeur ajoutée car elle est le produit réel des instituts de recherche [55]. Nous pourrions envisager que la simulation à évènements discrets constitue un domaine intéressant pour l'exploitation des avantages de l'AOP et ses thèmes de recherche récents qui sont illustrées par la Figure 2.1.

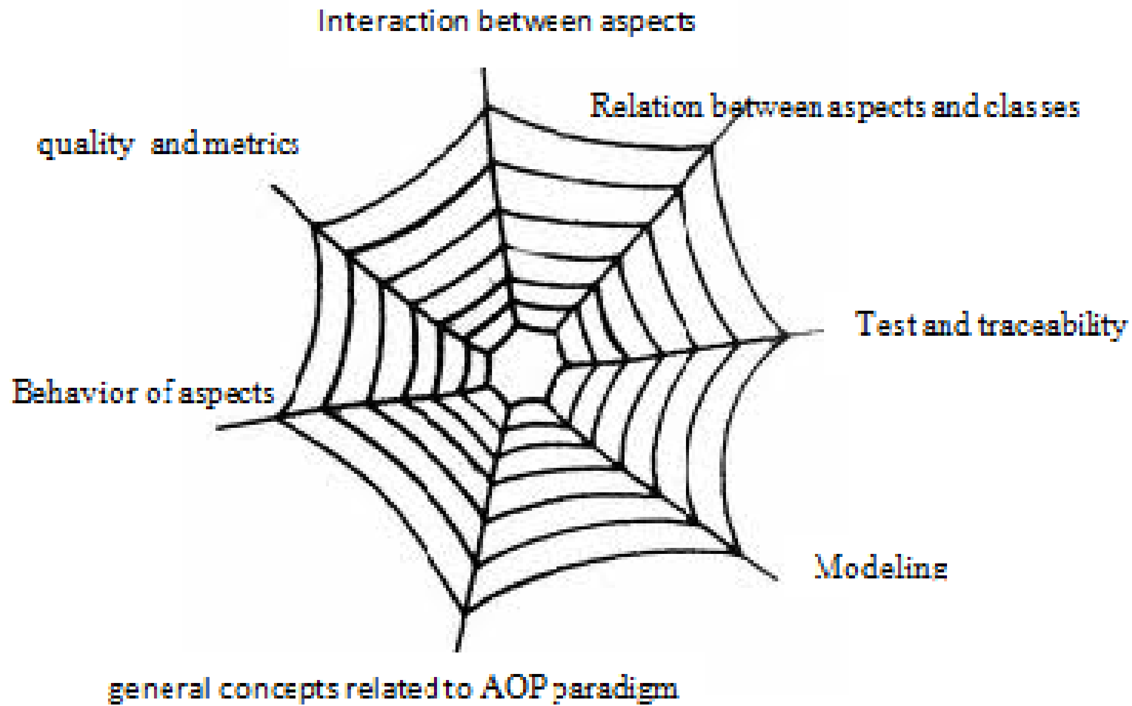


FIGURE 2.1 : Les domaines de recherche de l'AOP [88].

2.4 Les principales préoccupations transversales d'un système à évènements discrets

Après une étude minutieuse de plusieurs applications de simulation à évènements discrets, une liste de préoccupations transversales propre à ce domaine a été proposée dans [27], à savoir :

2.4.1 La détection de l'état d'équilibre

La préoccupation de la détection de l'état d'équilibre détermine la stabilisation du système dans lequel les données de sortie collectées pendant la période d'échauffement d'une simulation peuvent être trompeuses. Ainsi, l'élimination des effets de l'état transitoire du système sur les résultats est important pour obtenir des estimateurs de performance précis. Afin de déterminer le début de la phase d'équilibre, plusieurs méthodes ont été proposées. Ces méthodes d'estimation de la longueur de la phase transitoire peuvent être classées en cinq catégories principales : les méthodes graphiques, heuristiques, statistiques, tests de biais d'initialisation, et hybrides [53] .

2.4.2 L'animation graphique

L' animation graphique est une technique permettant de visualiser le comportement du système au cours de l'expérimentation. Cette visualisation est utile lors de la validation d'un modèle conceptuel et pour aider à l'interprétation des résultats d'une simulation. En outre, elle aide les utilisateurs à mieux comprendre la dynamique du système étudié et s'avère très utile pour l'enseignement et l'apprentissage.

2.4.3 L'interface utilisateur graphique

L'interface graphique est utilisée pour la présentation des résultats statistiques de la simulation et faciliter l'interaction avec les utilisateurs. Son code peut recouper divers modules fonctionnels rendant les interfaces graphiques difficiles à maintenir.

2.4.4 Le traitement des exceptions

La gestion des exceptions est une exigence non fonctionnelle pour n'importe quelle application pour gérer correctement toute condition erronée comme l'indisponibilité des ressources, entrée invalide, entrée nulle. Ainsi, les solutions proposées pour la gestion des exceptions polluent le code des systèmes de simulation et les rendent incohérents.

2.4.5 L'exactitude des calculs

Malheureusement, au cours des opérations arithmétiques (addition, multiplication, soustraction et division) les exceptions de dépassement de capacités ou de de soupassement ne sont pas signalés et passent inaperçus. C'est le cas de la plupart des langages de programmation, dont Java, où ces opérateurs arithmétiques ne signalent pas ces anomalies. Ils les remplacent tout simplement. Par conséquent, une telle préoccupation transversale s'assure de l'exactitude des résultats de calcul, ce qui est crucial dans les systèmes de simulation.

2.4.6 La trace de simulation

Dans les systèmes de simulation, il est important d'enregistrer les changements d'état de toutes les ressources, les entités passives et actives quand celle-ci se produisent afin de garder un historique. Ces enregistrements polluent le code fonctionnel de simulation de manière transversale.

2.4.7 La synchronisation

En plus de la synchronisation de l'exclusion mutuelle qui limite les activités concourantes sur des sections critiques pour les protéger contre l'incohérence des données en raison de l'accès simultané en écriture. Par exemple, dans la programmation multithread en langage Java, un objet pourrait être accessible par de nombreux threads simultanément. Par conséquence, les conflits d'accès aux données pourraient se produire si les applications ne sont pas préparés pour faire face à la concurrence. Ainsi, la synchronisation doit être mise en œuvre en utilisant le modificateur "synchronised" au niveau de la méthode ou de construire la synchronisation d'objet au niveau d'instruction ou de bloc [34]. Dans l'approche de modélisation par interaction de processus, qui est largement adoptée par la communauté de simulation à évènements discrets, le modèle se compose d'un ensemble de processus qui interagissent. Chaque processus est une séquence bien ordonnée d'activités qui sont logiquement liées. Donc, la synchronisation entre ces processus est une tâche essentielle au niveau d'un simulateur. Par conséquent, la synchronisation des processus de simulation et la synchronisation de l'exclusion mutuelle sont des préoccupations transversales dans le code fonctionnel de simulation.

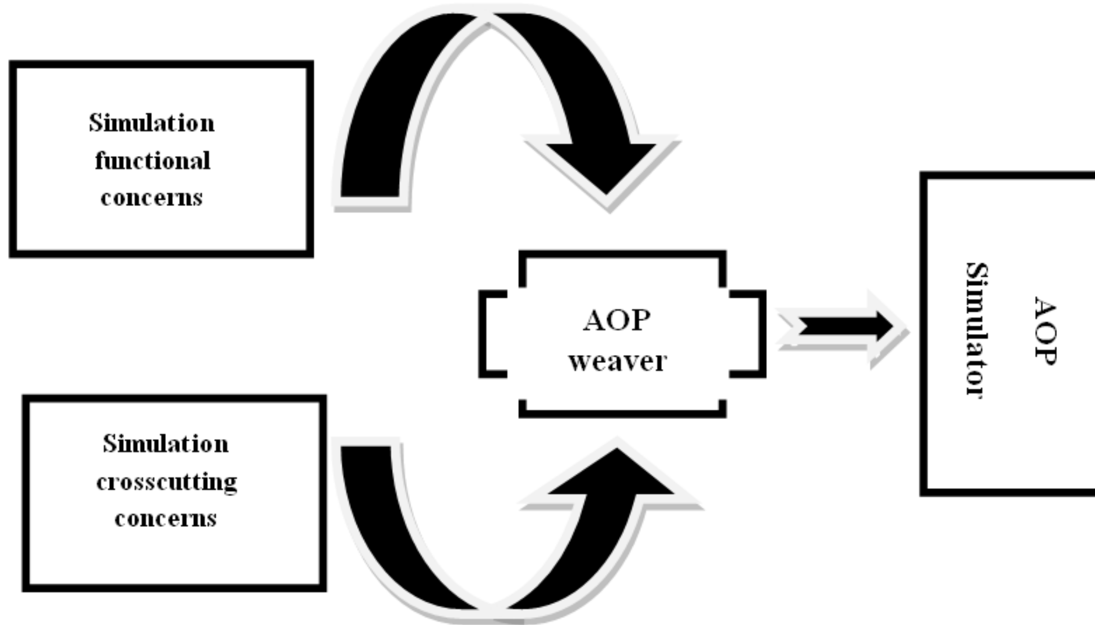


FIGURE 2.2 : Le processus d'implémentation d'un système de simulation orienté aspect [27].

2.5 Le processus de mise en œuvre d'un système de simulation orientée-aspect

En générale, le processus de mise en oeuvre d'un système de simulation orienté aspect se divise en trois phases, comme illustré dans la figure 2.2, l'identification des préoccupations du système, leur mise en oeuvre et le développement du système final par la composition des différentes préoccupations, de la façon suivante [27] :

1. Tout d'abord, les préoccupations transversales tel que la synchronisation et la détection de l'état stationnaire sont séparées des préoccupations fonctionnelles du noyau de simulation. Cette phase peut être comparée au passage d'un faisceau de lumière à travers un prisme pour séparer les différentes composantes de couleur.
2. Deuxièmement, chaque préoccupation transversale est mise en œuvre indépendamment en utilisant un langage orienté aspect comme AspectJ qui réduit la complexité globale de la conception et de l'implémentation. En plus, un des langages procéduraux ou orientés objet est utilisé pour l'implémentation des préoccupations fonctionnelles de la simulation.
3. Enfin, un tisseur d'aspect comme le compilateur AspectJ est utilisé pour composer toutes les préoccupations transversales et fonctionnelles de la simulation pour produire le système final.

2.6 Conclusion

Nous avons identifié dans ce chapitre les différentes préoccupations transversales dans le domaine de la modélisation et de la simulation. Une synthèse des différents travaux qui ont tenté d'exploiter l'AOP dans le domaine de la simulation a été présentée. Le couplage des domaines de l'AOP et de la simulation, en particulier celui des systèmes à évènements discrets, semble prometteur et présente des avantages certains comme :

- ♣ Réduction de la complexité des systèmes logiciels de simulation. En séparant le code dans diverses sections (modules), la complexité est ainsi réduite.
- ♣ Amélioration de la qualité des logiciels de simulation sur plusieurs plans (maintenance, test, et réutilisation).

Dans le prochain chapitre, nous allons à proposer un support de l'AOP pour la séparation des préoccupations transversales au niveau de la conception contrairement à la séparation classique au niveau code.

Chapitre 3

Un nouveau profile UML pour la modélisation orientée aspect

« *On devrait tout rendre aussi simple que possible, mais pas plus* »

Albert Einstein

Sommaire

3.1 Introduction **53**

3.2 La modélisation orientée-aspect **53**

3.3 Les critères de classification des approches de modélisation orientée-aspect **54**

3.4 Le processus de développement **55**

3.4.1 Processus Aspectuel 55

3.4.2 Processus Hybride 56

3.5 UML pour la conception orientée aspect **57**

3.5.1 L'extension générale d' UML 57

3.5.2 Le profile UML 58

3.6 Le profile AspectJ **58**

3.6.1 Travaux antérieures 58

3.6.2 Profile AspectJ proposé 59

3.7 Conclusion **68**

3.1 Introduction

Le paradigme orienté aspect est une technologie permettant d'améliorer la séparation des préoccupations des logiciels. Il a émergé au début au niveau code en utilisant des langages matures comme AspectJ. Actuellement, il devient nécessaire de pousser l'application de ce paradigme vers les étapes précoces du processus de développement logiciel. Cela est appelé le développement logiciel orienté-aspect (Aspect-Oriented Software Development-AOSD), qui est un sujet de recherche très actif en génie logiciel qui produit des systèmes plus fiables, réutilisables et maintenables. La modélisation orientée-aspect (MOA) consiste à séparer les préoccupation transversales au niveau de la phase de conception. Dans ce chapitre, nous présentons les concepts de base qui ont une relation avec la MOA. De plus, nous proposons un profile UML pour la modélisation des concepts d'AspectJ. Ce profile est un ensemble de mécanismes d'extensions UML. Nous essayons de discuter ses éléments par la suite.

3.2 La modélisation orientée-aspect

Le développement logiciel orienté-aspect (AOSD pour Aspect-Oriented Software Development) adopte les idées de la séparation des préoccupations dans le développement logiciel en les encapsulant dans des modules appropriés ou dans des parties spécifiques d'un logiciel, et a pour objectif supplémentaire de fournir un nouveau moyen d'encapsulation de préoccupations transversales, appelées aspects. Cette idée apparaît aujourd'hui comme une manière judicieuse de compléter la notion de modules disponible dans la plupart des langages de programmation. La séparation des préoccupations transversales permet au concepteur de logiciels d'avoir un meilleur contrôle sur les variations et les évolutions futurs du logiciel.

D'un point de vue développement logiciel, l'AOSD a émergé au niveau code, avec notamment Aspect-J qui a joué un rôle déterminant dans l'émancipation de cette approche. Pourtant, à travers l'importance croissante de l'IDM, le paradigme orienté-aspect ne s'est plus restreint au niveau de la programmation, et il s'étend maintenant aux phases amonts du développement logiciel, par exemple au niveau de la conception, de l'analyse ou encore de l'étude des exigences d'un système, comme illustre la figure 3.1. Dans ce contexte, la modélisation orientée-aspect étend le processus de développement de l'IDM (qui propose des transformations verticales), en découpant le modèle d'un système en plusieurs préoccupations à un même niveau d'abstraction, et en les composant à travers un processus de tissage qui peut être vu comme une transformation *horizontale* de modèles. Outre le fait de séparer les préoccupations transversales tôt dans le cycle de développement, couplé à des mécanismes automatiques de tissage d'aspects, la modélisation orientée-aspect pourrait permettre de cibler des plates-formes non orientées-aspect ou bien faciliter la mise en place de techniques de validation comme la simulation ou la génération de tests [58].

Pour une notation de modélisation orientée aspect qui fournit une base pour atteindre une meilleure séparation des préoccupations, un langage visuel de modélisation, basé sur UML à usage général présente plusieurs avantages par rapport aux alternatives textuelles. La notation doit être complète, ce qui implique un support d'abstraction pour chacun des concepts commun de l'AOSD (aspect, composants, coupe, advices, les mécanismes d'introduction, relations aspect-composants et relations aspect-aspect). En outre, différents concepts devraient être implicitement ou explicitement mis en correspondance avec les différents éléments UML existants ou nouveaux. La notation doit être indépendante du

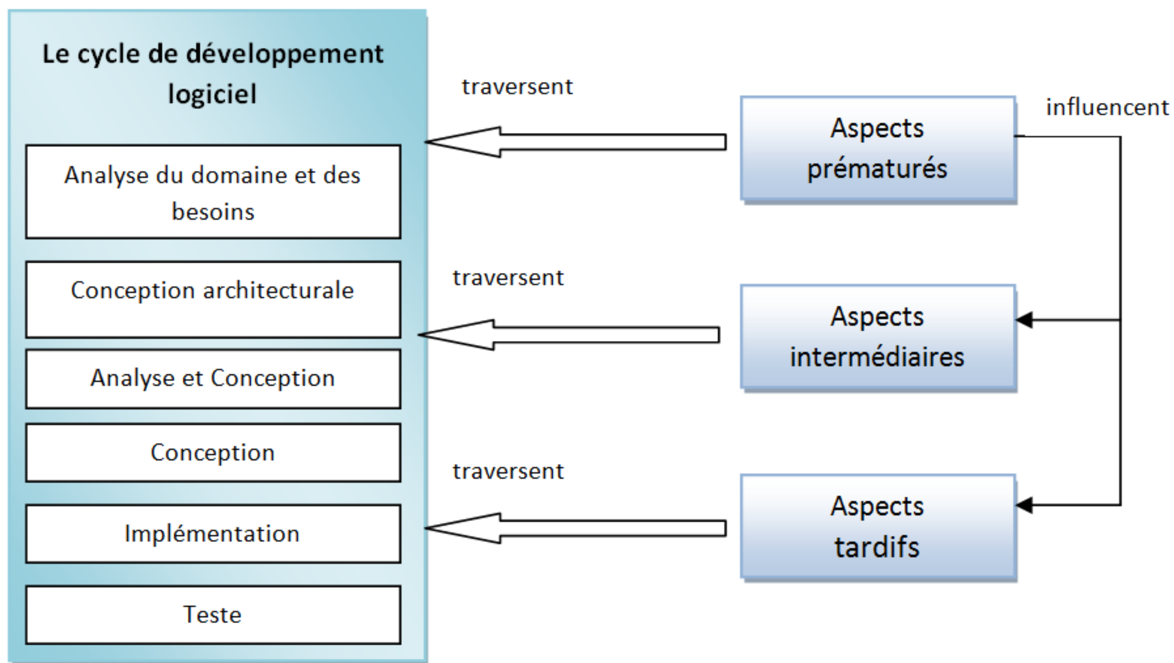


FIGURE 3.1 : Le développement logiciel orienté-aspect.

langage de mise en œuvre jusqu'à ce que le niveau de détail est fourni. De cette façon, les modèles architecturaux orientée aspect résultant pourraient être facilement convertis en éléments de langages/framework de programmation orientée aspect distincts et une notation de conception détaillée. Enfin, la notation intégrée basée sur UML devrait promouvoir la simplicité et éviter les extensions inutiles [59].

3.3 Les critères de classification des approches de modélisation orientée-aspect

Il ya un nombre considérables d'approches AOM (Aspect-Oriented Modeling) qui ont été proposées dans la littérature. Puisque l'orientation aspect est souvent considérée comme une extension de l'orientation objet, il semble presque naturel d'utiliser et / ou d'étendre le standard pour la modélisation orientée objet. Par exemple, le langage de modélisation unifié (UML) est utilisé pour les approches AOM où il ya seulement quelques propositions qui ne fondent pas leurs concepts sur UML. Dans ce qui suit, un catalogue des critères pour une évaluation structurée des approches AOM (voir la figure 3.2) [82] :

— *Concern Composition* : couvre les différents mécanismes de composition. Il traite d'abord, avec la modularisation et donc avec la séparation des préoccupations d'un système en unités principales et secondaires, avec leurs interactions, et par conséquent leur composition au moyen de règles appropriées.

— *Asymmetric Concern Composition* : cette catégorie englobe les critères d'évaluation des approches AOM qui ont une composition *asymétrique*. Elles sont classées en deux sous-catégories : *AspectualSubject* et *AspectualKind*. La première, fournit des critères pour évaluer les concepts utilisés pour décrire les points de greffe des modules non-fonctionnels ex : les points de jonction et ses sous-concepts. La seconde, contient les critères utilisés pour l'évaluation des concepts utilisés pour décrire comment intégrer les modules non-fonctionnels, par exemple, les advices et

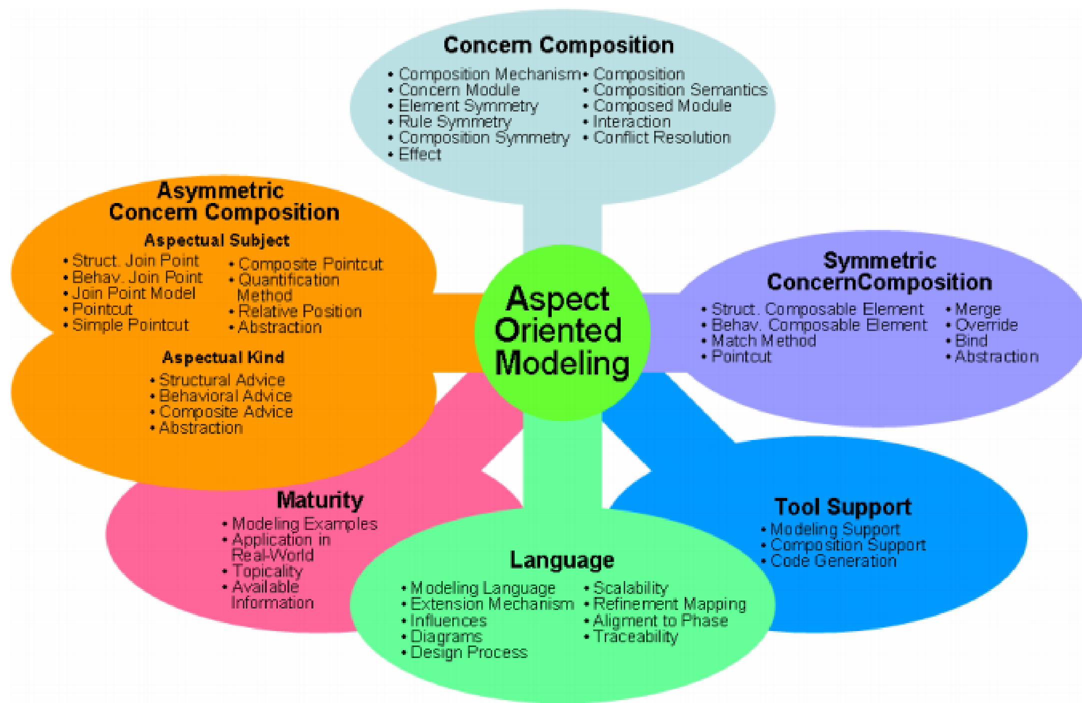


FIGURE 3.2 : Catalogue des critères d'approches AOM [82].

le niveau d'abstraction à laquelle la modélisation des advices est possible.

— *Symmetric Concern Composition :* contient des critères pour l'évaluation des approches qui ont une composition *symétrique*.

— *Language* : cette catégorie contient des critères généraux décrivant le langage de modélisation et le processus de conception.

— *Maturity* : évalue la maturité de l'approche en général. Il sert pour évaluer si une approche a été appliquée sur des exemples du monde réel. L'évaluation se fait avec un ensemble de sous-critères. (*Modelling Examples*, *Application in Real-World Projects*, *Topicality* and *Available Information*).

— *Tool Support* : améliore l'adoption d'une approche et s'assure de l'exactitude syntaxique du modèle. Ce critère distingue entre le support de modélisation, la composition et le support utilisé pour la génération de code.

3.4 Le processus de développement

Deux méthodes alternatives peuvent être adoptée lors d'une conception orientée-aspect [54] :

3.4.1 Processus Aspectuel

Il permet la séparation des préoccupations du niveau conception jusqu'au niveau code. Il comprend trois étapes illustrées par la figure 3.3. La première consiste en la conception des préoccupations séparées et la précision de leur composition, la deuxième génère le code, par exemple, le code AspectJ en utilisant des outils de transformation modèle à code et la dernière complète le code AO partiellement généré.

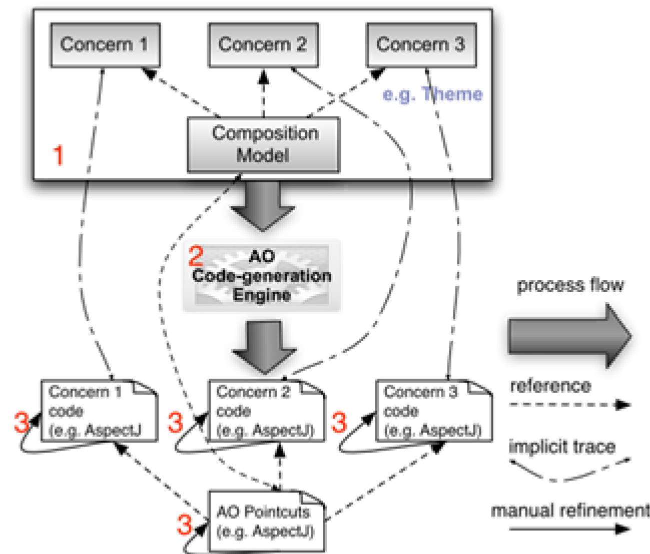


FIGURE 3.3 : Processus Aspectuel [54].

3.4.2 Processus Hybride

Il commence comme le modèle aspectuel par la séparation des préoccupations au niveau de la conception en spécifiant des règles de tissage (étape 1 comme le montre la figure 3.4). Ensuite, la composition des préoccupations modélisées est réalisée au niveau modèle (étape 2). Ce modèle composé est traduit par un moteur de type transformation modèle-à-code qui produit du code source orienté objet (dans notre cas, Java), enfin ce code est affiné manuellement.

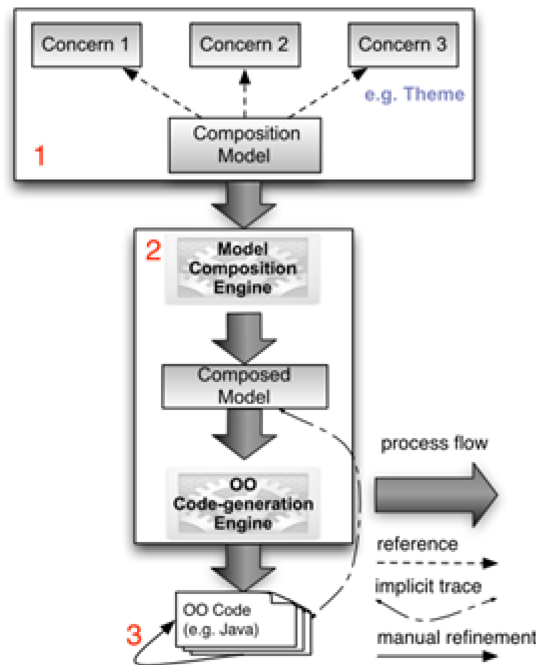


FIGURE 3.4 : Processus Hybride [54].

3.5 UML pour la conception orientée aspect

Le langage UML (*unified modeling language*) est un langage standard orienté objet de modélisation pour spécifier, visualiser, construire et documenter les artefacts d'un système. UML a été initialement conçu par Rational Software Corporation et trois des méthodologistes spécialisés dans les systèmes d'information : GradyBooch, James Rumbaugh et Ivar Jacobson. Le langage a acquis un appui significatif de l'industrie et de différentes organisations via UML Partners Consortium et a été approuvé par l'Object Management Group (OMG) comme un standard en Novembre 1997 [94]. La version 2.0 d'UML a été finalisée par l'OMG en Juillet 2005, en ajoutant de nouvelles fonctionnalités à la version 1.x d'UML. La dernière version officielle selon l'OMG est UML 2.4.1 (08/2011). UML 2.4 est composé de 14 diagrammes.

Le métamodèle UML spécifie le langage de modélisation. Il est défini et normalisé au niveau 2 au dessus de la couche 3 *UML Meta Object Facility* (MOF) (voir la figure 3.5). Le niveau 1 définit la couche modèle. La couche d'objets utilisateur définit les instances du modèle au niveau 0 [5]. Il existe différents langages de conception orientée aspect utilisés pour spécifier les préoccupations transversales basées sur UML tels que AODM, Thème/UML, SUP, UFA et AML [30]. Pour activer UML afin de représenter les concepts AO au niveau de la conception, deux alternatives sont disponibles :

3.5.1 L'extension générale d' UML

Elle modifie le méta modèle UML pour inclure à ce niveau, les concepts liés au paradigme. Ainsi, UML sera en mesure de modéliser tous les types d'aspects dans n'importe quelle situation et dans n'importe quel contexte. L'inconvénient de cette approche est que pour être si général, il se peut que certains concepts liés aux aspects soient perdus lors de la modélisation [7].

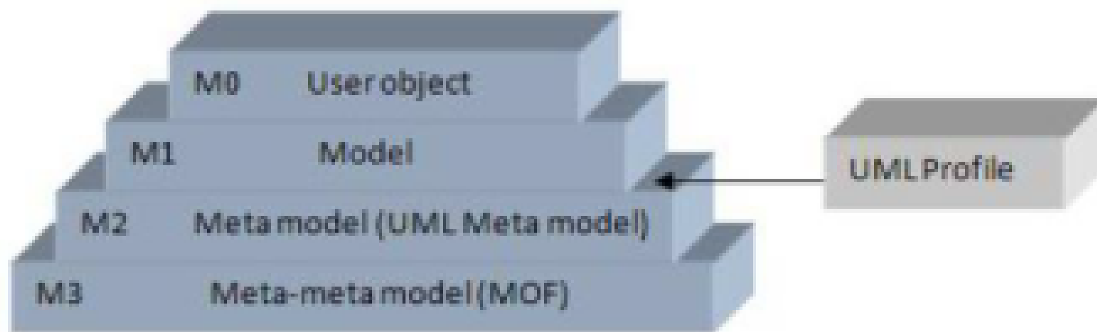


FIGURE 3.5 : L'architecture de metamodel selon l'OMG [43].

3.5.2 Le profile UML

L'un des points forts d'UML, c'est qu'il fournit des mécanismes d'extension qui permettent au langage de s'adapter à différents types d'aspects. Les mécanismes d'extension sont les moyens de personnaliser et d'étendre UML. Les mécanismes d'extension UML sont basées sur les stéréotypes (*Stereotypes*), les valeurs marquées (*Tagged Values*), et les *Contraintes* :

1. Les *Stéréotypes* : sont des moyens d'étendre le vocabulaire d'UML. [70] ". Le stéréotype fournit une manière de classification des éléments afin qu'ils se comportent comme s'ils étaient des instances virtuelles de nouveaux métamodèles . Les instances ont la même structure (attributs, associations, opérations) comme une instance non-stéréotypée du même type. Le stéréotype peut spécifier des contraintes et valeurs marquées supplémentaires applicables aux instances. En outre, un stéréotype peut être utilisé pour indiquer une différence de sémantique ou d'usage entre deux éléments ayant une structure identique."
2. Les *valeurs marquées* : sont des propriétés pour la spécification des caractéristiques ou des attributs pour les éléments du modèle. [70] " Une valeur marquée est une paire (nom, valeur) qui permet l'ajout d'une nouvelle propriété à n'importe quel élément de modèle"
3. Les *contraintes* : sont utilisées pour détailler la façon dont un élément d'UML peut être traité. Elles peuvent être spécifiées de manière informelle. [70] " Le concept de contrainte permet de spécifier linguistiquement une sémantique supplémentaire pour un élément du modèle. La spécification est exprimée en langage de contrainte. Le langage peut être spécialement conçu pour l'écriture des contraintes (comme OCL), un langage de programmation, la notation mathématique, ou un langage naturel."

3.6 Le profile AspectJ

3.6.1 Travaux antérieures

La première discussion sur le profile UML a été présentée dans [4] qui proposait la spécification des aspects comme des stéréotypes sur des classes et des aspects et le comportement comme une relation d'association utilisant le diagramme de collaboration. Le profile est incomplet et spécifique à l'aspect "synchronisation". Il n'aborde pas les

TABLEAU 3.1 : Comparison entre les langages de l'AOP [3].

	AspectJ	**AspectS**	**AspectML**
Aspects can be instantiated	⊠	✓	AspectML does not have an aspect construct
Aspect inheritance	⊠	✓	
Nested aspects	✓	⊠	
Privileged aspects	✓	⊠	
Polymorphic pointcuts	⊠	⊠	✓
Polymorphic advices	⊠	⊠	✓
Advice on field access	✓	⊠	NA

concepts clés comme : points de jonction, advice et coupe. Il a été ensuite étendu pour inclure la spécification d'advice et de coupe dans [5].

Contrairement aux travaux précédents, un nouveau profile d'AspectJ sans indication textuelle est discuté dans [37]. Il a été développé en utilisant l'outil commercial *MagicDraw* avec le format *XMI* (*XML Metadata Interchange*) qui permet facilement la génération de code. Cependant, il a des incohérences par rapport à ce qui est requis par le paradigme et la preuve est apportée par un processus pour la vérification des profiles orientées aspect [42]. Dans [56], Le profile d' Evermann a été étendu pour supporter les *frameworks* orientés aspect en tenant compte de certains idiomes AspectJ, les patrons et les stéréotypes d'un profile de *framework* orienté objet appelé *UML-F*.

Selon la terminologie *Model Driven Architecture (MDA)* et contrairement aux travaux précédents, qui ne permettent que la création de modèles spécifiques à la plateforme *PSM*, un profile qui vise la création de *PIM* a été développée dans [38]. Après l'identification de points communs et les différences entre les deux implémentations de l'AOP comme le montre le tableau 3.1. Les différences significatives entre les langages de mise en œuvre, c'est à dire, *AspectJ* et *AspectS*, rendent le profile résultant complexe à appliquer à des modèles. Ainsi, un profile dédié à une plateforme spécifique constitue la meilleure solution pour réduire la complexité [3].

Récemment, Gowri [43] a modélisé les points de jonction comme des diagrammes de séquence en adoptant *XMI* pour le déploiement du profile en utilisant les outils CASE disponibles. C'est un profil générique qui ne capture que quelques mécanismes d'AspectJ.

3.6.2 Profile AspectJ proposé

Le profile AspectJ proposé est basé sur l'extension *lightweight* de UML, voir la figure 3.6. Dans le contexte MDA, ce profile permet le développement d'un modèle spécifique à la plateforme (PSM) avec la possibilité de génération de code, basé sur [22].

Le stéréotype *Aspect*

Le concept *aspect* représente une unité primordiale dans les systèmes orientés aspect. Il comprend des membres statiques, des advices et des coupes. D'une part, il est similaire à une classe. Il a des attributs et des opérations. Il pourrait hériter d'autres classes. Il peut réaliser des interfaces. En outre, il peut être abstrait [61]. Ainsi, l'aspect est modélisé au moyen du stéréotype 'Aspect ' qui est une extension de la métaclasse 'Class ' comme indiqué sur la figure 3.7. D'autre part, plusieurs caractéristiques distinguent un aspect d'une classe. Afin de surmonter ce problème, des attributs et des contraintes supplémentaires sur la métaclasse 'Aspect' sont ajoutés comme suit :

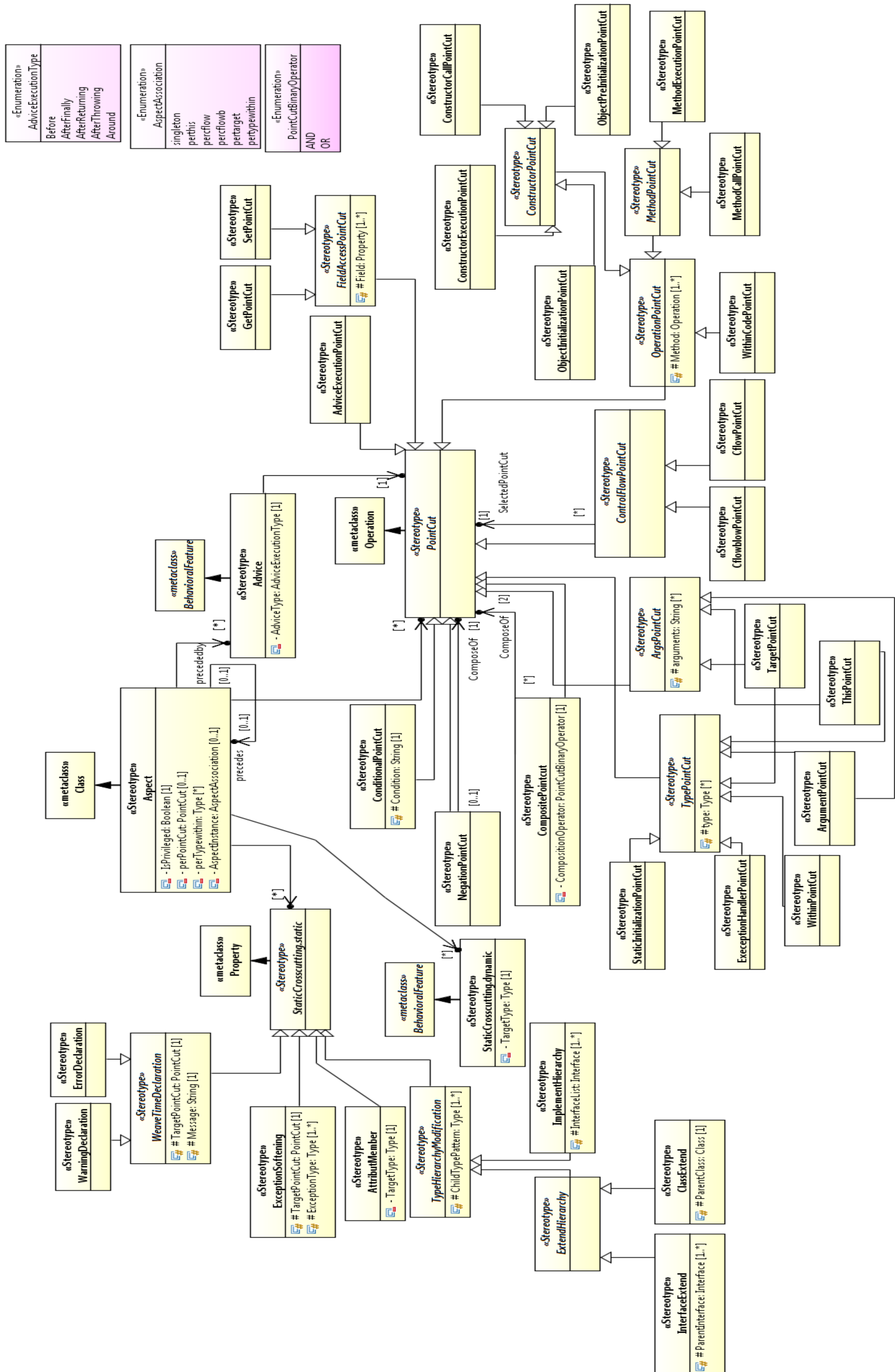


FIGURE 3.6 : Le profile AspectJ.

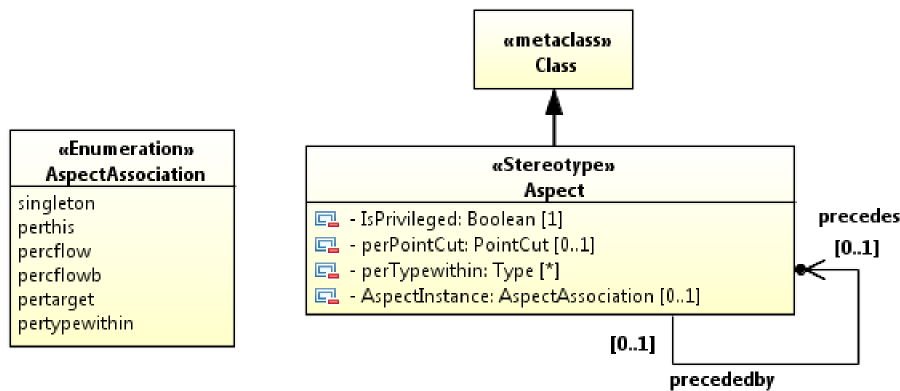


FIGURE 3.7 : Le stereotype *Aspect*.

1. Les attributs

— IsPriviliged : est un attribut booléen qui indique la capacité d'accès privilégié de l'aspect. Si sa valeur est *true*, l'aspect peut accéder aux membres privés de toutes les classes de l'application.

— AspectInstance : AspectInstance spécifie le modèle d'instanciation d'aspect. Ses valeurs possibles sont "singleton ""perthis ", "pertarget ", "percflow ", "singleton ", "pertypewithin "et "percflowb ".

— perPointCut : cet attribut contient la coupe associée.

— perTypewithin : cet attribut comprend le type associé quand l'instanciation d'aspect prend la valeur 'pertypewithin ' .

— Precedence : il est modélisé à l'aide d'une association (réflexive) récursive et il détermine l'ordre d'exécution des aspects ayant le même point de jonction.

2. Les contraintes

— Contrairement à une classe, un aspect concret ne peut pas déclarer des paramètres génériques [61]. Par conséquent, seuls les aspects abstraits peuvent déclarer des paramètres de type générique. Tout aspect concret qui hérite des aspects abstraits doit lier les paramètres génériques dans sa déclaration.

```
context AspectJ::Aspect ERROR
" A concret aspect must have not generic parameters ":
    !isTemplate() || isAbstract;
```

— Un aspect concret ne peut être hérité [61].

```
context  AspectJ::Aspect ERROR
"The concrete aspect is not available for inheritance":
this.generalization.forAll(e|(e.general.getAppliedStereotypes().name.
    contains('Aspect')
  && e.general.isAbstract)
  ||(e.general.isAbstract  ||(!e.general.getAppliedStereotypes().name.
    contains('Aspect')
  && !e.general.isAbstract)));
```

— Contrairement à une classe, un aspect qui n'est imbriqué dans aucun autre aspect ne peut avoir qu'un spécificateur d'accès *public* ou *package*. Un aspect interne, comme une classe interne, pourrait avoir un spécificateur d'accès *public*, *private*, *protected*, ou *package* [61].

```
context AspectJ::Aspect ERROR
" Aspect visibility kind must be public or package unless nested aspect "
    :
(visibility.toString()=='public' || visibility.toString()=='package')||!
    this.isActive;
```

— Un aspect interne (imbriqué dans un autre) doit être statique [61].

```
context AspectJ::Aspect ERROR
" A nested aspect must be declared as static aspect ":
this.nestedClassifier.typeSelect(AspectJ::Aspect).notExists(e|e.isActive)
    ;
```

— Un aspect abstrait contient au moins une coupe abstraite ou une méthode abstraite.

```
context AspectJ::Aspect ERROR
" An abstract aspect has at least one abstact method or pointcut ":
!( this.allOwnedElements().typeSelect(uml::Operation).exists(e|e.
    isAbstract))
  || this.isAbstract ;
```

— L'instanciation d'aspects est indiquée par un des deux attributs "perTypewithin" ou "perPointCut", mais pas par les deux en même temps.

```
context AspectJ::Aspect ERROR
" The aspect instantiation is indicated either by perTypewithin or by
    perPointCut attributes, but not for both":
 (AspectInstance.name=='pertypewithin' && !perTypewithin.isEmpty &&
    perPointCut==null)
     || ((AspectInstance.name=='perthis' || AspectInstance.name=='
    pertarget' || AspectInstance.name=='percflow'
     || AspectInstance.name=='percflowb') && perPointCut!=null &&
    perTypewithin.isEmpty)
     || (AspectInstance.name=='singleton'&& perTypewithin.isEmpty&&
    perPointCut==null);
```

Le stéréotype *Advice*

C'est une construction dynamique dans AspectJ. Elle modifie le comportement du système visé aux points choisis par les coupes. Elle exprime un comportement. En plus, elle a un nom, un corps, des arguments et pourrait renvoyer des exceptions. Elle est modélisée en utilisant la métaclasse 'Advice' qui étend la métaclasse 'BehavioralFeature ' comme indiqué sur la figure 3.8.

1. Les attributs

— L'attribut AdviceType est de type AdviceExecutionType lui même de type énumération. Il détermine le type d'advice. Ses valeurs possibles sont : “Before”, “AfterReturning”, “AfterThrowing”, “AfterFinally”et “Around”.

2. Les contraintes

— Une advice n'a pas un type de retour sauf si son type est “around” [61] comme suit :

```
context  AspectJ::Advice ERROR
" An advice has not a return type unless the -around- advice ":
 (AdviceType.name=='Around') ||(this.ownedParameter.notExists(e|e.
    direction.toString()=='return'));
```

— Une advice peut être juste un membre des aspects [52]. Pour assurer cette propriété une contrainte est ajoutée.

```
context AspectJ::Advice ERROR
"An advice may be just a member of aspects":
this.featuringClassifier.getAppliedStereotypes().name.contains('Aspect');
```

— Une advice ne peut pas être abstrait.

```
 context  AspectJ::Advice ERROR
 "An advice may not be abstract":
 !this.isAbstract==true;
```

— Une advice ne peut pas être *statique*.

```
context  AspectJ::Advice ERROR
"An advice may not be static":
!this.isStatic==true;
```

— Une advice ne peut pas être *finale*.

```
context  AspectJ::Advice ERROR
"An advice may not be final":
!this.isLeaf==true;
```

Le stéréotype *Pointcut*

Une coupe sélectionne les points de jonction avec une description structurelle. Elle n'a aucune relation avec le comportement dynamique et elle a des paramètres. Par conséquent, elle ressemble à une opération sans la mise en œuvre du corps d'implémentation. Une opération UML est une déclaration avec un nom et des paramètres [43]. Pour cette raison, elle est modélisée en utilisant la métaclasse "Pointcut " qui étend la métaclasse "Operation" comme indiqué sur la figure 3.9. Plusieurs contraintes sont ajoutées sur la métaclasse PointCut afin de surmonter la différence entre l'opération d'UML et la coupe comme suit :

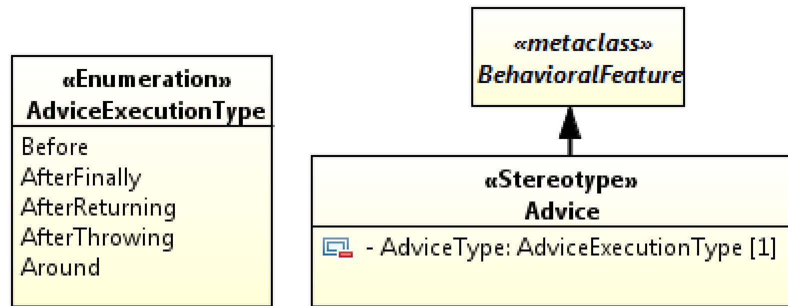


FIGURE 3.8 : Le stéréotype *Advice*.

— Une coupe n'a pas un type de retour [52] .

```
context AspectJ::PointCut ERROR
" The PointCut has not a return result ":
this.ownedParameter.notExists(e|e.direction.toString()=='return');
```

— Une coupe ne peut pas être *statique*.

```
context AspectJ::PointCut ERROR
"A pointCut may not be static":
this.isStatic==false;
```

— Une coupe ne peut pas être *finale*.

```
context AspectJ::PointCut ERROR
"A pointCut may not be final":
this.isLeaf==false;
```

PointCut est une méta-classe abstraite qui ne peut être appliquée directement aux opérations d'un aspect, qu'à travers ses sous-classes abstraites, comme MethodCallPointCut et MethodExecutionPointCut. Plutôt que de préciser le type et la déclaration des coupes d'AspectJ comme des attributs sur *PointCut*, ce profile propose des sous-classes de la métaclasse PointCut pour permettre la modélisation des différents attributs d'une coupe.

OperationPointCut est une super-classe pour décrire les coupes qui sélectionnent les points de jonction liées à des opérations. Par conséquent, cette métaclasse a l'attribut ordonné *Method* qui peut prendre différentes valeurs dont le type de données est la métaclasse UML "Operation". Les sous-classes de ce stéréotype reflètent directement les types de coupes d'AspectJ. Le stéréotype *OperationPointCut* a deux sous-catégories : *MethodPointCut* et *ConstructorPointCut*. Parce que *UML* ne fait pas de distinction entre les opérations et les constructeurs, deux contraintes sont ajoutées :

```
context AspectJ::ConstructorPointCut ERROR
"ConstructorPointCut is target constructors":
Method.forAll(e|e.name==e.class.name);
```

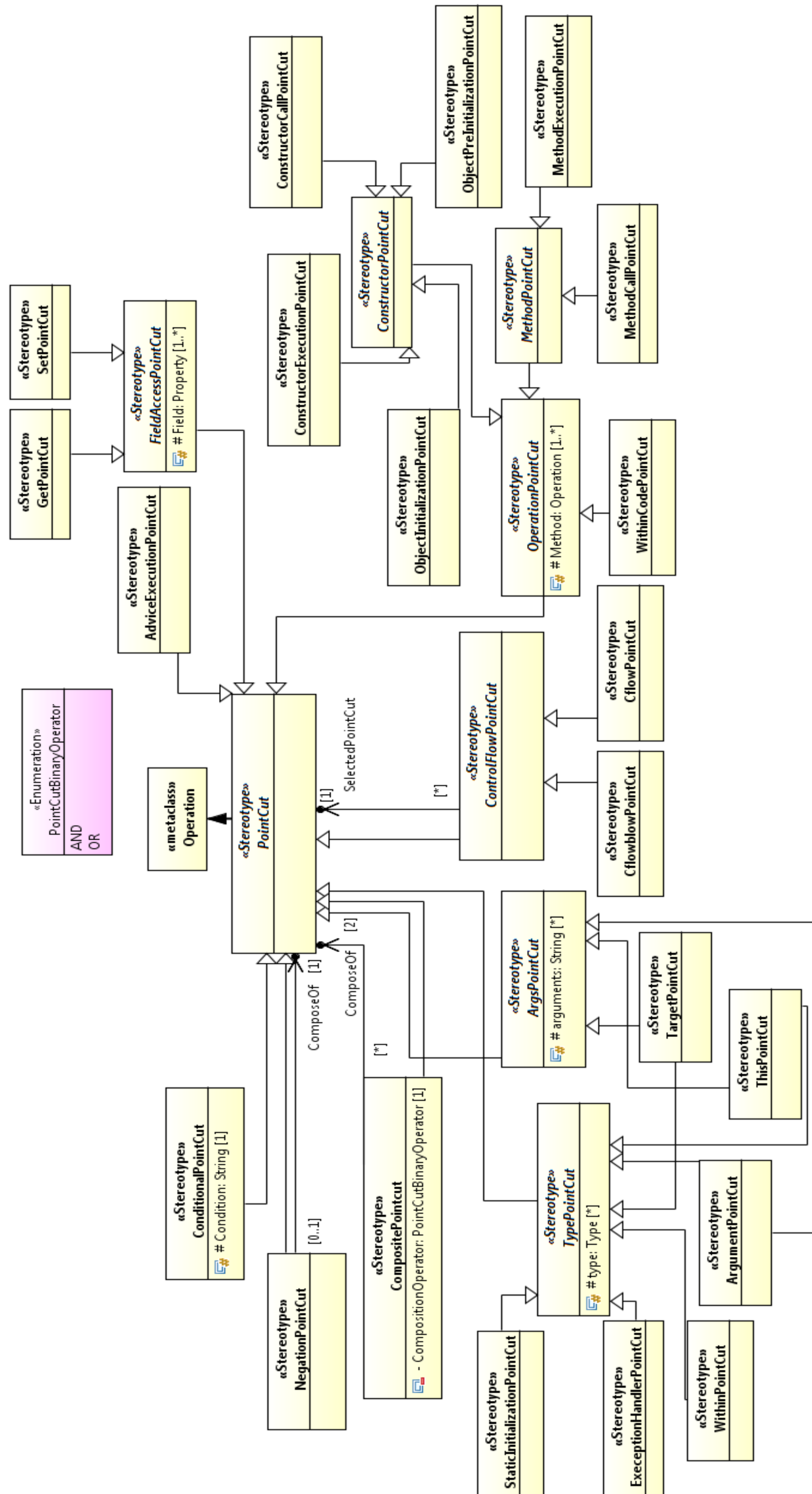


FIGURE 3.9 : Le stéréotype *Pointcut*.

```
context  AspectJ::MethodPointCut ERROR
"MethodPointCut is target methods":
Method.forAll(e|e.name!=e.class.name);
```

Les sous-classes de *ConstructorPointCut* et qui héritent l'attribut *Method* sont : *ObjectInitializationPointCut*, *ObjectPreInitializationPointCut*, *ConstructorExecutionPointCut* et *ConstructorCallPointCut*. *MethodCallPointCut* et *MethodExecutionPointCut* sont des sous-classes de *MethodPointCut*.

TypePointCut est une super-classe pour décrire les coupes qui sélectionnent les points de jonction liée à des types. Ainsi, elle contient un attribut ordonné *type* de valeurs multiples dont le type de données est la métaclasse *Type* d'UML. Les sous-classes de *TypePointCut* reflètent directement les modèles de l'élément coupe en *AspectJ*. Pour les sous-classes *WithinPointCut*, *ExeceptionHandlerPointCut* et *StaticInitializationPointCut*, l'attribut *type* doit avoir au moins une valeur. Les sous-classes *ThisPointCut*, *TargetPointCut* et *ArgumentPointCut* peuvent être utilisées pour capturer le contexte des points de jonction et l'attribut *type* peut être nul.

```
context  AspectJ::WithinPointCut ERROR
"WithinPointCut is indicated at least by a one type value":
!type.isEmpty;
```

```
context  AspectJ::ExeceptionHandlerPointCut ERROR
"ExeceptionHandlerPointCut is indicated at least by a one type value":
!type.isEmpty;
```

```
context  AspectJ::StaticInitializationPointCut ERROR
"StaticInitializationPointCut is indicated at least by a one type value":
!this.type.isEmpty;
```

ArgsPointCut est une super-classe abstraite qui peut exposer un contexte sur une advice. Elle contient l'attribut ordonné *arguments* de valeurs multiples. Cet attribut désigne les noms des arguments exposés. Cette collection est ordonnée, de sorte que le type correspondant peut être déterminé d'après le type de collection ordonnée spécifiée pour la métaclasse *TypePointCut*. *ArgsPointCut* possède trois sous-classes : *ThisPointCut*, *TargetPointCut* et *ArgumentPointCut* qui héritent l'attribut *arguments*. Ainsi, des contraintes sont ajoutées pour s'assurer que ceux-ci sont indiqués soit par l'attribut *arguments* ou *type*, mais pas les deux à la fois de la façon suivante :

```
context  AspectJ::ThisPointCut ERROR
"ThisPointCut is indicated either per arguments or type attributes, but not for
    both":
(arguments.size==1 && type.size==0) || (arguments.size==0 && type.size==1);
```

```
context  AspectJ::TargetPointCut ERROR
"TargetPointCut is indicated either per arguments or type attributes, but not for
    both":
```

```
3 (arguments.size==1 && type.size==0) || (arguments.size==0 && type.size==1);
```

```
1 context  AspectJ::ArgumentPointCut ERROR
"ArgumentPointCut is indicated either per arguments or type attributes, but not for
    both":
3 (arguments.isEmpty && !type.isEmpty) || (!arguments.isEmpty && type.isEmpty);
```

AdviceExecutionPointCut décrit une coupe qui sélectionne toutes les exécutions des advices. *ControlFlowPointCut* est une super-classe abstraite pour les types de coupe qui choisissent une autre coupe. Par conséquent, elle est associée à la métaclasse *PointCut* pour spécifier les coupes sélectionnées. *FieldAccessPointCut* est une super-classe de sous-classes *GetPointCut* et *SetPointCut* qui décrit le type des coupes qui sélectionnent les champs. Par conséquent, elle possède l'attribut *Field* qui a des valeurs multiples et de type *Property* qui est une méta-classe UML.

ConditionalPointCut décrit une coupe qui sélectionne les points de jonction basées sur un certain contrôle conditionnel au point de jonction. Elle possède l'attribut *Condition* qui est utilisé pour spécifier la condition.

En *AspectJ*, les coupes peuvent être composées de coupes primitives. Par conséquent deux types de coupes sont introduites : *CompositePointCut* et *NegationPointCut*. Celles-ci sont modélisées comme des sous-catégories distinctes parce que l'opération de négation n'accepte qu'un seul opérande, alors que la conjonction et la disjonction nécessitent au moins deux opérandes. Aucun ordre des opérandes pour la conjonction ou la disjonction n'est nécessaire car ces opérations sont associatives et commutatives. La métaclasse *CompositePointCut* possède un attribut supplémentaire *composition _operator* avec l'énumération type de données *BinaryOperator*.

Pour faire toutes les références aux points de jonction choisies par les coupes, différentes valeurs pour les attributs *Method*, *Field*, *arguments* et *type* sont utilisés sur *OperationPointCut*, *FieldAccessPointcut*, *ArgsPointcut* et *TypePointCut* respectivement pour réduire la complexité du modèle résultant. L'alternative utilisée pour forcer le modélisateur est l'utilisation de la composition de coupes basée sur la disjonction dans la spécification de la coupe. Lorsque plusieurs caractéristiques sont précisées pour les coupes, par exemple les valeurs multiples de l'attribut *Method* d'une instance de *MethodExecutionPointCut*, l'hypothèse lors de la génération de code, c'est de considérer qu'elles sont composées à l'aide d'une disjonction logique.

Les stéréotypes : *StaticCrosscutting.static* et *StaticCrosscutting.dynamic*

La construction *static crosscutting* modifie la structure statique des classes d'une manière transversale. Puisque ces fonctionnalités transversales peuvent être statiques ou dynamiques. Elle est modélisée utilisant les métaclasses 'StaticCrosscutting.static' et 'StaticCrosscutting.dynamic' respectivement comme le montre la figure 3.10. Deux contraintes sont ajoutées pour s'assurer que ces dernières ne sont appliquées que sur les opérations et les attributs de classes qui sont stéréotypés *Aspect* comme suit :

```
1 context  AspectJ::StaticCrosscutting_static ERROR
"The Static Crosscutting stereotype (static) may not be a member of classes that
    aren't stereotyped Aspect ":
3 this.featuringClassifier.getAppliedStereotypes().name.contains('Aspect');
```

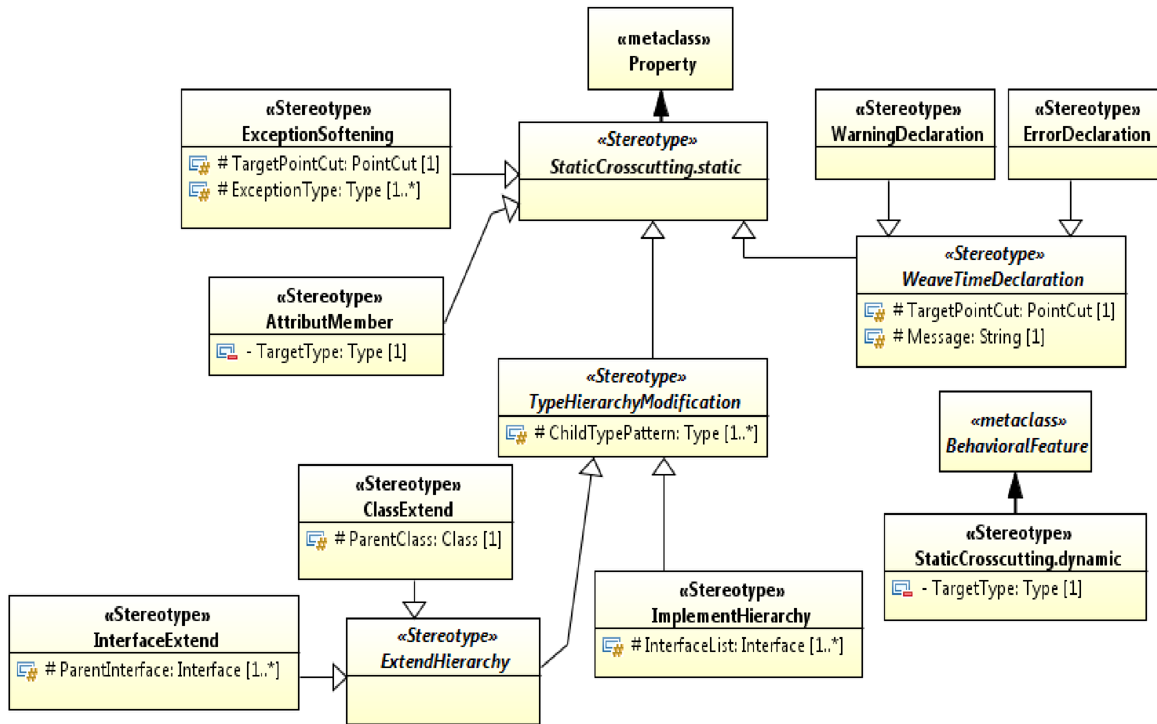


FIGURE 3.10 : Les stéréotypes : *StaticCrosscutting.static* et *StaticCrosscutting.dynamic*

```
context  AspectJ::StaticCrosscutting_dynamic ERROR
"The Static Crosscutting stereotype(dynamic) may not be a member of classes that
   aren't stereotyped Aspect ":
this.featuringClassifier.getAppliedStereotypes().name.contains('Aspect');
```

Le stéréotype *StaticCrosscutting.static* est modélisé en utilisant la métaclasse Property. C'est une métaclasse abstraite qui ne peut être appliquée aux attributs d'un aspect qu'à travers ses sous-classes non abstraites comme *ExceptionSoftening, ErrorDeclaration, ClassExtend* et *WarningDeclaration.*

Le stéréotype *StaticCrosscutting.dynamic* ajoute des méthodes à des classes, interfaces, ou aspects. Ainsi, il est modélisé utilisant la métaclasse 'BehavioralFeature'.

3.7 Conclusion

Dans ce chapitre, nous avons conçu et développé un profile UML pour la modélisation orientée aspect. Ce profile adopte la terminologie du langage *AspectJ*. Il s'agit d'une spécification complète du langage *AspectJ* du point de vue statique en termes d'*aspect, advice, pointcut* et *static crosscutting* utilisant les mécanismes d'extension du méta-modèle d'UML. En plus, Il est compatible avec le format XMI ce qui signifie qu'il est possible de manipuler et d'échanger le profile entre les différents outils CASE basés sur UML. Dans les prochains chapitres, nous visons à proposer des solutions selon la POA pour séparer les préoccupations transversales telles que l'interface graphique, l'ordonnancement des threads, et la synchronisation des événements en utilisant Japrosim comme noyau fonctionnel pour la simulation à événements discrets.

Troisième partie

Implémentation

Chapitre 4

Mise en œuvre utilisant la bibliothèque Japrosim

« *La théorie, c'est quand on sait tout et que rien ne fonctionne. La pratique, c'est quand tout fonctionne et que personne ne sait pourquoi. Si la pratique et la théorie sont réunies, rien ne fonctionne et on ne sait pas pourquoi* »

Albert Einstein

Sommaire

4.1 Introduction **71**

4.2 La bibliothèque Japrosim **71**

4.3 La version orientée-aspect de Japrosim **72**

4.3.1 La version orientée aspect des patrons de conception 74

4.3.2 L'aspect *SimulationTrace* 76

4.3.3 L'aspect *Synchronization* 77

4.3.4 L'aspect *ExceptionHandling* 79

4.3.5 L'aspect *GraphicalUserInterface* 80

4.3.6 L'aspect *CalculationAccuracy* 80

4.4 La comparaison entre les deux version AO et OO de Japrosim **82**

4.4.1 Les métriques d'évaluation du logiciel 82

4.4.2 L'outil : AOPmetrics 82

4.4.3 Évaluation 84

4.5 Conclusion **86**

4.1 Introduction

JAPROSIM (JAva PRocess Oriented SIMulation) est une librairie, libre et open source. Elle est mise en œuvre utilisant le langage de programmation Java pour construire des modèles de simulation à évènements discrets. Cette bibliothèque souffre de quelques préoccupations transversales que nous avons mis en évidence dans les chapitres précédents, telles que la détection de la phase d'équilibre et l'animation graphique. Ces préoccupations traversent ses modules et ont tendance à diminuer des propriétés de qualité comme la modularité, la compréhensibilité, la maintenabilité, la réutilisabilité, et la testabilité. Dans ce chapitre, nous proposons des solutions visant à séparer les principales préoccupations transversales de Japrosim en utilisant AspectJ. Une évaluation empirique de la version orientée aspect de Japrosim est effectuée en utilisant l'outil AOPmetrics.

4.2 La bibliothèque Japrosim

Japrosim est une librairie libre et open source qui adopte l'approche de modélisation la plus utilisée à savoir : l'approche par interaction de processus. Sa conception est intuitive et facile à appréhender. Elle facilite la construction de modèles de simulation à évènements discrets pour des utilisateurs qui maitrisent Java ou même pour les experts en simulation n'ayant que des connaissances élémentaires en programmation. Il faut noter aussi que Japrosim n'est pas une version Java d'une quelconque bibliothèque, comme c'est le cas de SimJava et JavaSim.

La bibliothèque Japrosim possède un mécanisme caché pour la collecte automatique des statistiques. Cette technique permet une nette séparation entre l'implémentation de la dynamique du modèle et la collecte des statistiques sur celui-ci. Ainsi, les mesures de performances traditionnelles sont automatiquement calculées. Le modèle est développé sans se soucier des statistiques. Ainsi, aucun code pour la collecte des statistiques n'apparait dans le modèle de simulation. L'utilisateur possède la possibilité, si nécessaire, d'implémenter des statistiques spécifiques utilisant les mécanismes offerts par cette bibliothèque dans le package *statistics*, ce qui distingue Japrosim des autres bibliothèques de simulation à évènements discrets écrites en Java. A l'exception de SimKit qui offre déjà un mécanisme similaire bien que celle-ci utilise une approche basée sur les graphes d'évènements [15].

La bibliothèque est divisée en paquetages pour organiser les classes en groupes fonctionnels. La bibliothèque est divisée en huit principaux paquetages [16] :

- kernel : Le paquetage kernel est au cœur de Japrosim. Il est constitué de classes traitant des entités actives, ordonnanceur, queues et des ressources comme le montre la figure 4.1.
- random : contient des classes pour la génération de flux de nombres aléatoire uniforme.
- random.distributions : contient un riche ensemble de classes pour les distributions de probabilité utiles.
- statistics : contient des classes représentant des variables statistiques intelligents.
- gui : un ensemble de classes d'interface d'utilisateur graphiques à utiliser pour le paramétrage du projet, visualiser la trace d'exécution et la présentation des résultats de la simulation.
- Utilities : un ensemble de classes qui facilitent le développement rapide des modèles de simulation.

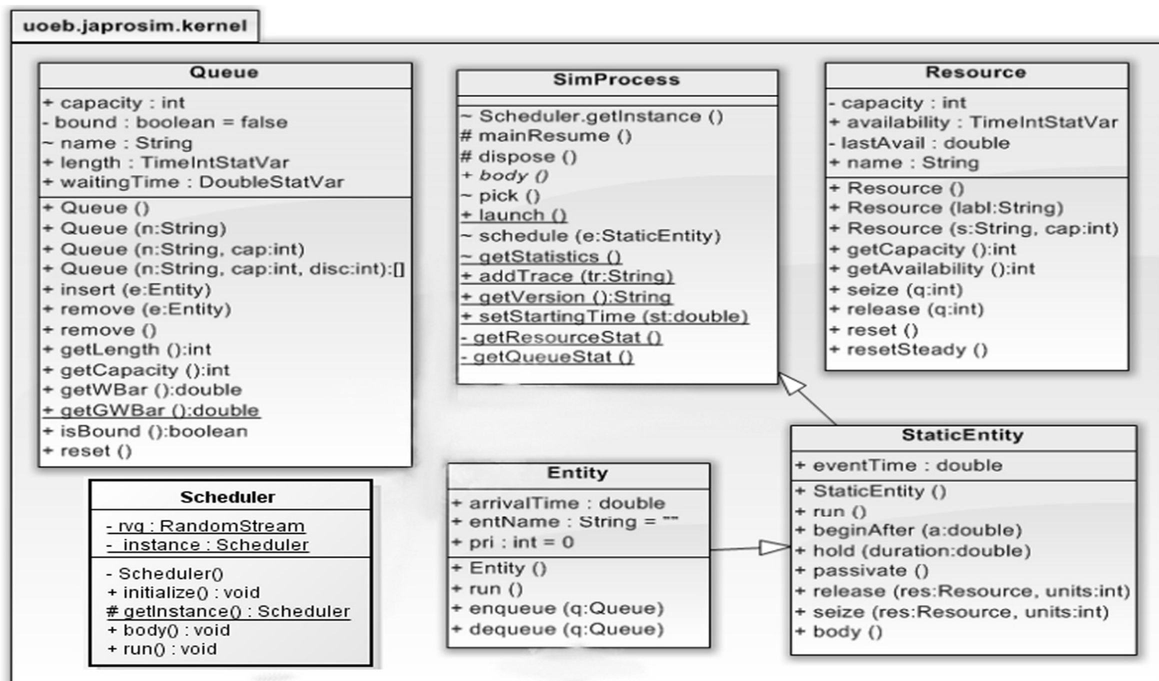


FIGURE 4.1 : Diagramme de classes du paquetage *kernel* [27].

— statistics.steady : utile pour détecter l'état d'équilibre.

— animation : un ensemble de classes utilisées pour fournir une animation en temps réel des modèles de simulation.

Le mécanisme de coroutines est mis en œuvre à travers les classes *SimProcess, Scheduler, StaticEntity* et *Entity*. Un programme de coroutine est une collection de coroutines qui s'exécutent. Chaque coroutine est un objet avec son propre état d'exécution, de sorte qu'il peut être suspendu et repris. Japrosim met une grande importance à s'adhérer à la sémantique de SIMULA mais la conception elle-même n'est pas une copie. L'avantage de cette approche est que la conception est simple, ne nécessite pas de classe coroutine explicite et la sémantique des concepts et mécanismes qui sont bien connus et testés à fond grâce à de nombreuses années d'utilisation de SIMULA sont complètement pris en charge. La prise en charge native de l'exécution multithread est un aspect fondamental pour la mise en œuvre d'une capacité de modélisation orientée processus naturelle en Java. Le cycle de vie de chaque entité active est exécuté en un seul thread séparé [16].

4.3 La version orientée-aspect de Japrosim

Les principales préoccupations transversales, décrites dans les chapitres précédents, sont identifiées et séparées dans le code de Japrosim à l'aide de l'outil AJDT [32]. La version orientée-aspect de Japrosim est restructurée en rajoutant le paquetage *uoeb.Japrosim.crosscutting.conc* Ce dernier se compose de huit aspects : *SingletonConcern, Animation, ExceptionHandling, GraphicalUserInterface, SimulationTrace, SteadyStateDetection, Synchronization* et *CalculationAccuracy*. Ainsi, chaque aspect donne une solution séparée à une préoccupation transversale qui pollue les paquetages fonctionnels de Japrosim comme le montre la figure 4.2.

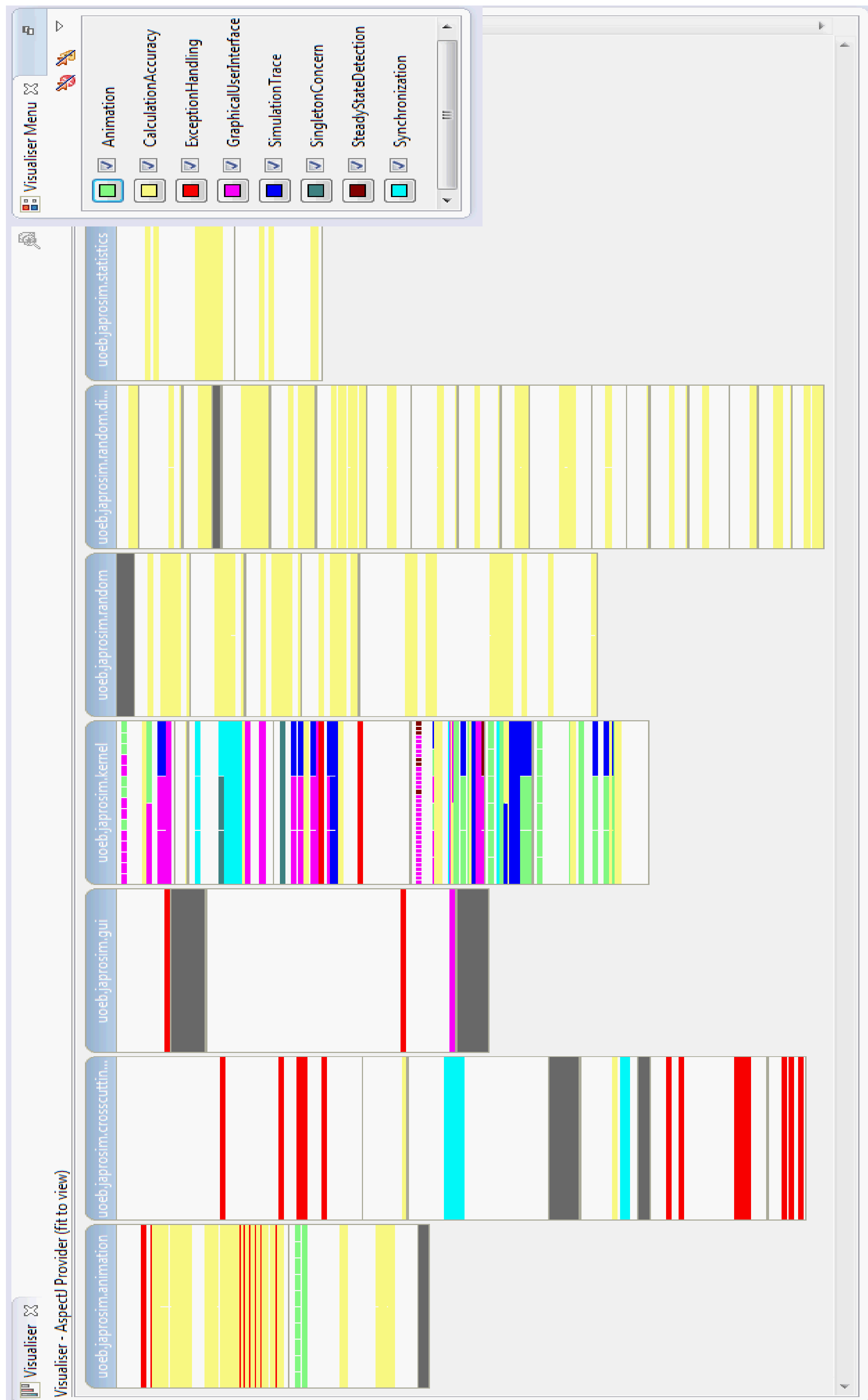


FIGURE 4.2 : L'interaction entre les aspects et les paquetages de la version AO de Japrosim.

4.3.1 La version orientée aspect des patrons de conception

Trois modèles de patrons de conception sont actuellement utilisés par la bibliothèque Japrosim. Les *creational patterns* sont *Singleton* et *Factory* tandis que le *behavioral pattern* est *Observer*. Les design patterns sont utiles, mais leur utilisation peut néanmoins diminuer la cohésion à l'intérieur des classes fonctionnelles. Ainsi, les implémentations des patrons de conception du Gang Of Four (GoF) utilisant AspectJ, augmente la modularité dans 17 des 23 cas en termes de meilleure séparation de code et réutilisation [49].

Singleton

Le patron *Singleton* empêche l'utilisation de deux objets de la même classe. La classe *Scheduler* en fait usage pour éviter plusieurs instances et s'assurer que la gestion de la liste d'évènements se fait exclusivement par un thread unique. Dans la version OO classique, Japrosim assure l'unicité de l'ordonnanceur en déclarant son constructeur comme privé. Elle fournit une méthode publique nommée getInstance() qui renvoie l'unique instance existante, et la sauvegarde dans une variable statique. L'aspect *SingletonConcern* est la solution proposée comme le montre le listing 4.1. L'aspect comprend une "around advice" qui s'applique au moment de l'appel du constructeur. L'advice a un rôle similaire à la méthode *getInstance()* sans la nécessité de déclarer le constructeur comme privé. En outre, l'instance de l'ordonnanceur est enregistrée dans une variable statique déclarée à l'intérieur de l'aspect en utilisant le mécanisme d'introduction d'AspectJ.

```
1  public aspect SingletonConcern {
2
3    /** precedence declaration */
4    declare precedence : Synchronization, SingletonConcern;
5
6    /** Initializes the Scheduler instance */
7    private static volatile Scheduler Scheduler.instance = null;
8
9    /**
10    * provides Scheduler instance
11    * @return
12    */
13   Scheduler around() : (call(public Scheduler.new())) && !this(SingletonConcern) {
14     if (Scheduler.instance == null) {
15       Scheduler.instance = new Scheduler();
16     }
17
18     return Scheduler.instance;
19   }
20
21 }
```

Listing 4.1 : L'aspect Singleton.

Observateur

Le modèle *Observer* crée une relation entre un sujet et un observateur. Un objet est appelé un sujet, lorsque des changements dans son état sont intéressants pour des objets observateurs. Pour mettre en œuvre le modèle d'observateur, les constructions suivantes sont utilisées [71] :

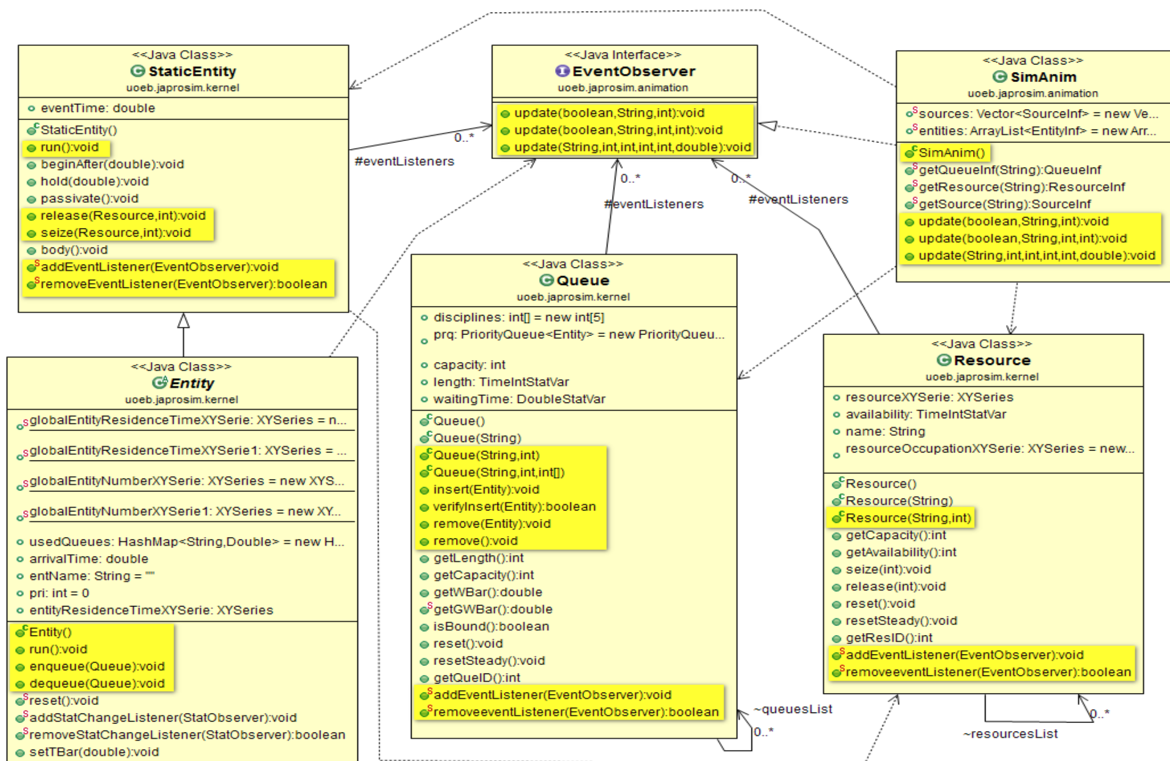


FIGURE 4.3 : L'animation graphique basée sur le pattern Observateur [24].

- Une interface d'observateur définit une méthode *update ()*. Celle-ci est appelée pour chaque observateur concret dans le cas d'un changement d'état du sujet observé.
- Une classe sujet définit des méthodes afin d'enregistrer les observateurs.
- Un sujet concret appelle, pour chaque modification de son état une méthode *fireUpdate ()*. Par la suite, tous les observateurs enregistrés pourraient réagir au changement et modifier leur propre état.
- Un observateur concret implémente l'interface d'observateur et s'enregistre auprès de tous les sujets qui l'intéressent.

Ces éléments du patron observateur sont dispersés dans les classes de Japrosim, ce qui diminuent leur cohésion et leur maintenabilité. Ainsi l'AOP propose l'encapsulation de ces éléments d'observateur dans un aspect et les classes de Japrosim restent sans infection. Ainsi, la détection de l'état d'équilibre et de l'animation graphique dans Japrosim font usage de ce modèle comme suit :

L'animation graphique La classe *SimAnim* fait partie du paquetage d'animation utilisée pour fournir une animation en temps réel. Elle récupère des données utiles au moyen de l'interface *EventObserver*. De plus, chacune des classes *Queue*, *Resource* et *StaticEntity* enregistre des écouteurs de type *EventObserver* pour les informer en cas d'apparition d'un évènement. Ce mécanisme est assuré par le patron "Observateur" comme le montre la figure 4.3. En surbrillance, les méthodes qui contiennent le code nécessaire pour mettre en œuvre cette instance du modèle d'observateur. L'aspect *Animation* est proposé en vue de séparer ces éléments en fournissant le lien entre le sujet et l'observateur en utilisant le mécanisme d'introduction pour déclarer l'interface *EventObserver* et la modification de type hiérarchique, qui touchent des sujets et la classe *SimAnim* respectivement .

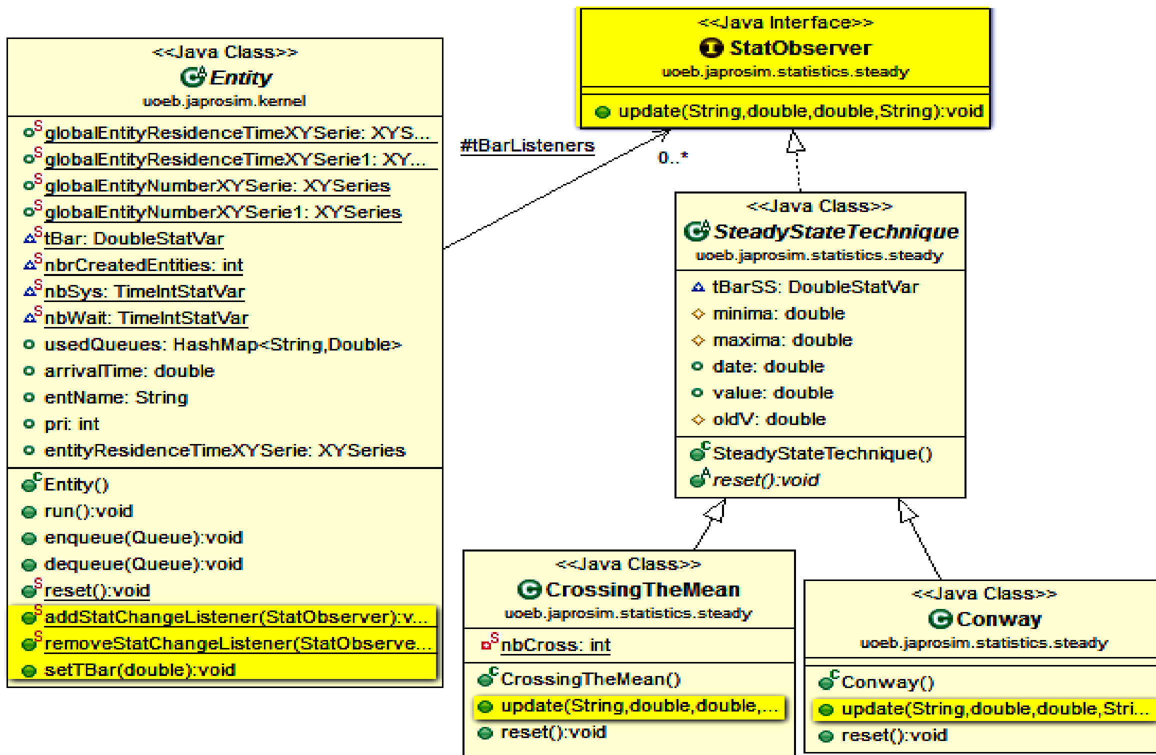


FIGURE 4.4 : La détection de l'état stationnaire fondé sur le pattern Observateur [25].

La détection de la phase d'équilibre : la version actuelle de Japrosim offre deux méthodes pour la détection de l'état stationnaire, *Conway* et *Crossing the Means* qui sont mises en œuvre en utilisant les modèles de conception Factory et Observateur. Les éléments d'observateur sont emmêlés dans les classes *Entity*, *SteadyStateTechnique*, *Conway* et *CrossingTheMean* (voir la figure 4.4). Comme dans le cas de l'animation, le même principe est appliqué pour résoudre le problème en utilisant l'aspect S*teadyStateDetection*.

4.3.2 L'aspect *SimulationTrace*

Pour enregistrer la trace de simulation, Japrosim génère trois fichiers. L'un est au format texte (trace.txt) et les deux autres sont conformes au format XML (trace.xml et resource.xml). Chaque évènement exécuté lors de la simulation implique la mise à jour de ces fichiers ce qui rend cette préoccupation transversale par rapport aux modules fonctionnels de Japrosim. Par exemple, la méthode *public void remove(Entity)* supprime l'entité spécifiée de la file d'attente ensuite appelle la méthode *addTrace (String)* et *TraceWriter* pour enregistrer cet évènement dans les deux fichiers trace.txt et trace.xml comme illustré dans la figure 4.5. L'aspect *SimulationTrace* fournit la solution illustrée par l'advice de type "after" comme représenté sur le listing 4.2.

```
1 after (Queue q, Entity e) : (this(q) && get (public TimeIntStatVar length)) && (
      cflowbelow(p22(e))
        && withincode(public void remove(Entity))) {
3
      addTrace("        " + e.getName() + " leaves queue "
5         + q.name + "\n");
      TraceWriter.println("<Event " + " Date = \""
7         + SimProcess.time + "\" Subject = \"" + e.getName()
          + "\" Identifier = \"" + e.getId() + "\" Action = \""
9         + "leaves" + "\" Object = \"" + q.name + "\"/>");
```

```
public void remove(Entity e) {
    if (this.prq.remove(e)) {
        this.waitingTime.update(SimProcess.time
                - e.usedQueues.get(this.toString()).doubleValue());
        Queue.gwaitingTime.update(SimProcess.time
                - e.usedQueues.get(this.toString()).doubleValue());
        Entity.nbWait.decrement();
        length.decrement();
        SimProcess.addTrace("      "+e.getName()+" leaves queue "
                + this.name + "\n");
        SimProcess.TraceWriter.println("<Event "+" Date = \""
                +SimProcess.time+"\" Subject = \""+e.getName()
                +"\" Identifier = \""+e.getId()+"\" Action = \""
                +"leaves"+"\" Object = \""+this.name + "\"/>");
        if (Scheduler.animation) {
            int p = Integer.parseInt(e.getName().substring(
                    e.getName().lastIndexOf(" ") + 1));
            ListIterator<EventObserver>it=eventListeners.listIterator();
            while (it.hasNext()) {
                it.next().update(e.getClass().getSimpleName(), p,
                        this.getQueID(), 0, 2, SimProcess.time);
            }
        }
    }
}
```

FIGURE 4.5 : La méthode remove() de la préoccupation transversale trace de simulation en surbrillance [24].

```
}
```

Listing 4.2 : Arouund advice à l'intérieur de l'aspect *SimulationTrace*

4.3.3 L'aspect *Synchronization*

La synchronisation de l'exclusion mutuelle

Les méthodes de la version OO de Japrosim qui manipulent des variables partagées en exclusion mutuelle sont améliorées avec le mot-clé *synchronized* de la méthode *getInstance ()* de la classe *Scheduler*. Cela tend à polluer le code fonctionnel de Japrosim. Afin de surmonter ce problème, une part à l'intérieur de l'aspect *Synchronization* est conçue pour assurer la synchronisation des méthodes, comme dans le listing 4.3 à l'aide d'un verrou commun de l'aspect.

```
Object around() : (execution(protected static void SimProcess.schedule(StaticEntity
    ))
            || execution(protected  static StaticEntity SimProcess.pick()))
            || (((execution(* *.add*Listener(..)) || execution(* *.remove*Listener
    (..)))
                || (call (public Scheduler.new()) && !within(SingletonConcern)))) {
    synchronized(this) {
      return proceed();
    }
```

Listing 4.3 : Un fragment du code de l'aspect Synchronization pour la synchronisation de l'exclusion mutuelle.

La synchronisation des coroutines

Japrosim implémente le mécanisme de coroutine. Chaque coroutine est un objet ayant son propre état d'exécution, de sorte qu'elle peut être suspendue et reprise. A tout moment du temps réel une seule coroutine est activée. La méthode *processResume (Entity e)* est appelée par l'ordonnanceur afin de réactiver un processus de simulation et *mainResume ()* est appelée par un processus de simulation afin de réactiver l'ordonnanceur. Chaque processus de simulation a son propre objet de verrouillage. L'ordonnanceur possède l'objet *mainLock*. Les verrous sont utilisés en combinaison avec *wait ()* et *notify ()* pour synchroniser la mise en œuvre des threads. Un thread qui appelle une des méthodes précédentes se bloquer sur son propre verrou après en avoir avisé le thread approprié. À la fin de son cycle de vie, le processus de simulation appelle automatiquement la méthode *dispose ()* pour réactiver l'ordonnanceur sans se bloquer. Ainsi, le thread correspondant pourrait être résilié [16]. Les éléments qui assurent le mécanisme de la coroutine sont *processResume (Entity e)*, *mainResume ()* et *dispose ()*, en plus des objets *mainLock* et *lock*. Ces éléments sont respectivement séparés dans un aspect de synchronisation unique qui rend la conception claire et augmente la compréhensibilité. Une illustration du listing 4.4 contient deux "around advices" pour remplacer respectivement les méthodes *processResume(Entity e)* et *mainResume()*.

```
1  void around(StaticEntity e) : execution(protected static void processResume(
       StaticEntity)) && args(e) {
       while (!e.getState().toString().equals(String.valueOf("WAITING"))) {
3        SimProcess.yield();
       }
5
       synchronized (e.lock) {
7        e.lock.notify();
       }
9
       synchronized (SimProcess.mainLock) {
11       try {
           SimProcess.mainLock.wait();
13       } catch (Exception excep) {}
       }
15   }

17   /**
      * advice executed  around mainResume()
19    * @param ob
      */
21   void around(SimProcess ob) : execution(protected void mainResume()) && this(ob) {
       while (!String.valueOf("WAITING").equals((SimProcess.sched.getState()).toString
       ())) {
23       SimProcess.yield();
       }
25
       synchronized (SimProcess.mainLock) {
27       SimProcess.mainLock.notify();
       }
29
       synchronized (ob.lock) {
31       try {
           ob.lock.wait();
33       } catch (Exception e) {}
       }
35   }
```

```
37    /**
       * advice executed  around dispose()
39     */
      void around() : execution(protected void dispose()) {
41      while (!String.valueOf("WAITING").equals((SimProcess.sched.getState()).toString
        ())) {
          SimProcess.yield();
43      }
        synchronized (SimProcess.mainLock) {
45        SimProcess.mainLock.notify();
        }
47    }
```

Listing 4.4 : Un fragment de code de l'aspect Synchronization.

4.3.4 L'aspect *ExceptionHandling*

L' aspect *ExceptionHandling* contient cinq pointcuts et cinq advices qui traitent tous les types d'exceptions à l'intérieur de Japrosim comme illustré sur le listing 4.5.

```
1   privileged public aspect ExceptionHandling {

3     /** pointcut1 for capturing fnf*/
      pointcut pointcut1(FileNotFoundException fnf) : handler(FileNotFoundException+)
        && args(fnf);
5
      /** pointcut2 for capturing ex */
7     pointcut pointcut2(Exception ex) : (handler(Exception+) && args(ex)) && within(
        MainFrame);

9     /** pointcut3 for capturing uee*/
      pointcut pointcut3(UnsupportedEncodingException uee) :  handler(
        UnsupportedEncodingException+) && args(uee);
11
      /** pointcut4 for capturing ie*/
13    pointcut pointcut4(Exception ie) : handler(Exception+) && (within(TraceFrame) &&
        args(ie));

15    /** pointcut5 for capturing e*/
      pointcut pointcut5(Exception e) : (handler(Exception+) && args(e)) && (!(
        pointcut4(Exception))
17                                      && !pointcut3(UnsupportedEncodingException)) &&
        (!(pointcut2(Exception))
                                            && !(pointcut1(FileNotFoundException)));
19
      /**
21     * FileNotFoundException advice
       * @param fnf
23     */
      before(FileNotFoundException fnf) : pointcut1(fnf) {
25      System.err.println("Resources XML File couldn't be created");
        System.err.println(fnf);
27      System.exit(1);
      }
29
      /**
31     * Exception advice with MainFrame
```

```
   * @param ex
   */
  before(Exception ex) : pointcut2(ex) {
    System.err.println("Cannot install " + MainFrame.PREFERRED_LOOK_AND_FEEL
        + " on this platform:" + ex.getMessage());
  }

  /**
   * UnsupportedEncodingException advice
   * @param uee
   */
  before(UnsupportedEncodingException uee) : pointcut3(uee) {
    System.err.println("utf-8 not recognized nor suported !");
    System.err.println(uee);
    System.exit(1);
  }

  /**
   * Exception advice with TraceFrame
   * @param ie
   */
  before(Exception ie) : pointcut4(ie) {
  System.err.println("Error closing file.");
  ie.printStackTrace();
  }

  /**
   * Exception advice excluding the precedence advices
   * @param e
   */
  before(Exception e) : pointcut5(e) {
    e.printStackTrace();
  }

}
```

Listing 4.5 : L'aspect Exceptionhandling.

4.3.5 L'aspect *GraphicalUserInterface*

Le paquetage *gui* de Japrosim comprend un ensemble de classes (TraceFrame, PresentationFrame, Mainframe, et GraphicFrame) qui facilite la manipulation des paramètres des modèles de simulation, comme le nombre de réplications, la durée de l'expérience, et le type de générateur de nombres aléatoires. Malgré les avantages et la flexibilité offertes par la préoccupation interface utilisateur graphique, sa nature transversale conduit à la pollution de code comme illustré sur la figure 4.6. Il diminue la cohésion, en particulier à l'intérieur des classes du paquetage kernel de Japrosim. Comme solution basée sur l'AOP, l'aspect *GraphicalUserInterface* est proposé.

4.3.6 L'aspect *CalculationAccuracy*

Cet aspect gère le soupassement et le dépassement de capacité lors des opérations sur les types primitifs : int, float, double, et long. Les opérateurs arithmétiques impliquées sont : addition, soustraction, multiplication et division comme le montre le listing 4.6.

```
Object around (Object aa, Object bb) :
```

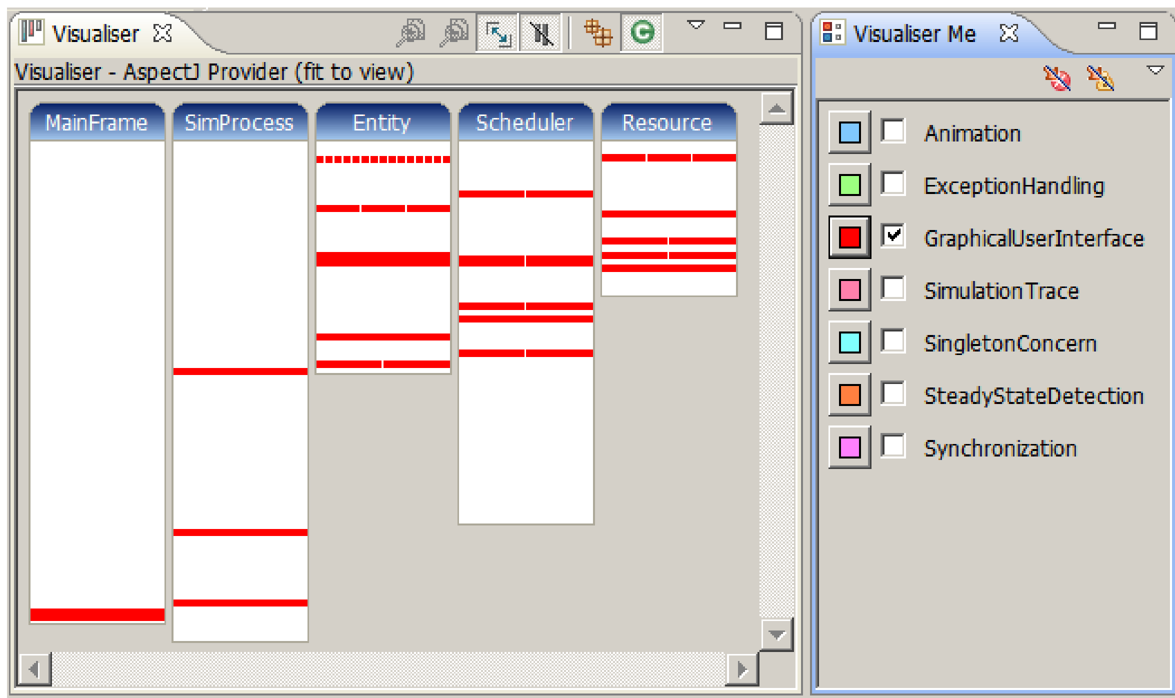


FIGURE 4.6 : L'infection de classes de Japrosim avec la préoccupations transversale GUI [24].

```
2   call (static public Object add(Object, Object)) && args(aa,bb)
    {
4     String ca = aa.getClass().getSimpleName();
      String cb = bb.getClass().getSimpleName();
6
      if(ca.equals(cb)&&ca.equals("Integer")) {
8       int a=((Integer)aa).intValue();
      int  b=((Integer)bb).intValue();
10    long c=(long)a+b;
      if(c <Integer.MIN_VALUE )
12      throw new ArithmeticException("int underflow");
      else if(c> Integer.MAX_VALUE)
14      throw new ArithmeticException("int overflow");
      return (new Integer((int)c));}
16    if(ca.equals(cb)&&ca.equals("Float")) {
        float a=((Float)aa).floatValue();
18    float  b=((Float)bb).floatValue();
      float c=a+b;
20    if(c==0 && (a!=-b))
        throw new ArithmeticException("float underflow");
22    else if(c==Float.POSITIVE_INFINITY||c==Float.NEGATIVE_INFINITY)
        throw new ArithmeticException("float overflow ");
24    return (new Float(c));
      }
26    if(ca.equals(cb)&&ca.equals("Double")) {
        Double a=((Double) aa).doubleValue();
28      Double  b=((Double)bb).doubleValue();
        double c=a+b;
30      if(c==0 && (a!=-b))
          throw new ArithmeticException("double underflow");
32      else if(c==Double.POSITIVE_INFINITY||c==Double.NEGATIVE_INFINITY)
        throw new ArithmeticException("double overflow ");
34      return (new Double(c));}
      if (ca.equals(cb)&&ca.equals("Long")) {
```

```
      long a=((Long)aa).longValue();
      long  b=((Long)bb).longValue();
      long c=a+b;
      if  (a > 0 && b > 0 && (c < a || c < b))
        throw new ArithmeticException("long Overflow");
    else if  (a < 0 && b < 0 && (c > a || c > b))
        throw new ArithmeticException("long underflow");
    return (new Long(c));
    }
    return null;
}
```

Listing 4.6 : Fragment de code de l'aspect *CalculationAccuracy* pour capturer le soupassement et le dépassement de capacité d'opération d'addition.

4.4 La comparaison entre les deux version AO et OO de Japrosim

4.4.1 Les métriques d'évaluation du logiciel

De nombreux travaux ont été menés afin de mieux comprendre et de définir la qualité d'un logiciel. Par exemple, l'approche GQM (Goal-Question-Metrics) [39] permet de définir des métriques pour l'évaluation systématique de la qualité d'un logiciel. Le standard ISO 9126, quant à lui, définit un modèle ayant pour but l'évaluation de la qualité. Ce modèle est divisé en six caractéristiques principales : fonctionnalité, fiabilité, utilisabilité, efficacité, maintenabilité, portabilité. Chacune d'entre elles se décompose à son tour en sous-caractéristiques. Une métrique est formellement définie comme une association entre le monde empirique et le monde numérique. C'est un nombre ou un symbole associé à une entité qui permet de caractériser ses attributs. Selon Fenton , nous cherchons à formaliser et à mieux comprendre le monde qui nous entoure grâce à des séries de métriques. Ces métriques nous permettent de valider nos intuitions par rapport à ce monde. Les mesures que nous obtenons doivent donc représenter fidèlement les entités observées et préserver les relations qui les lient les unes aux autres [47]. Par exemple, la figure 4.7 montre la relation entre certaines métriques de la suite de Chidamber et Kemerer (les mesures de CK) : Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Objects (CBO), Response For a Class (RFC), Lack of Cohesion of Methods (LCOM) et les propriété qualitatives d'un logiciel.

4.4.2 L'outil : AOPmetrics

AOPMetrics [86] est un outil de mesures communes pour la programmation orientéobjet et la programmation orientée-aspect. Cet outil fournit des extensions orientée aspect de la suite des métriques de Chidamber et Kemerer (les mesures de CK) [29] et la suite de Robert Martin (les métriques de dépendances du paquetage) [67].

Mesure LOCC

Lines of Class Code (LOCC) est une mesure simple qui compte le nombre de ligne de code des classes.

	WMC	DIT	NOC	CBO	RFC	LCOM
Understandability	X	X		X	X	
Maintainability	X			X	X	
Reusability	X	X	X	X		X
Testability		X	X	X	X	

FIGURE 4.7 : La relation entre la suite de métriques C & K et les propriétés qualitatives d'un système [89].

La suite de mesures de CK pour l'AOP

Avec AOPMetrics, le "module" a été utilisé comme un terme commun pour les classes et les aspects. De même, les méthodes, les advices et les introductions ont été désignées par le terme "opération" comme suit :

- Weighted Methods per Module (WOM) : mesure le nombre d'opérations mises en œuvre dans un module.
- Depth of Inheritance Tree (DIT) : est la longueur du chemin le plus long à partir d'un module donné dans la hiérarchie classe/aspect.
- Number of Children (NOC) : le nombre de classes/aspects héritiers d'un module.
- Coupling on Field Access (CFA) : est le nombre de modules ou interfaces déclarant des variables accessibles par un module donné.
- Coupling on Method Call (CMC) : est le nombre de modules ou interfaces déclarant des méthodes qui sont éventuellement appelées par un module donné.
- Coupling between Modules (CBM) : est le nombre de modules ou interfaces déclarant des méthodes ou des variables qui sont éventuellement appelées ou accessibles par un module donné.
- Crosscutting Degree of an Aspect (CDA) : est un nombre de modules affectés par les points de coupure et par les introductions dans un aspect donné.
- Coupling on Advice Execution (CAE) : est le nombre d'aspects contenant des advices éventuellement provoquées par l'exécution des opérations dans un module donné.
- Response For a Module (RFM) : le nombre de méthodes qui peuvent potentiellement être exécutées en réponse à un message reçu par un module donné.
- Lack of Cohesion in Operations (LCO) : est le nombre de paires d'opérations travaillant sur des champs de classes différentes moins le nombre d'opérations travaillant sur des champs communs (zéro si négatif). Lorsque toutes les méthodes n'ont pas accès à un champ, LCO prend la valeur zero.

Métriques du paquetage

AOPMetrics fournit deux versions des métriques de dépendances d'un paquetage. Les deux sont basées sur la gamme de Robert Martin.

— Number of Types (*NOT*) : est le nombre de types au sein d'un paquetage donné. C'est un indicateur de la capacité d'extension du paquetage.

— Abstractness (*A*) : est le rapport entre le nombre de modules abstraits et le nombre total de modules dans le paquetage.

— Afferent Couplings (*Ca*) : est le nombre de modules à l'extérieur du paquetage qui dépendent de modules à l'intérieur de celui-ci. *Ca* est un indicateur de la responsabilité de paquetage .

— Efferent Couplings (*Ce*) : est le nombre de modules à l'intérieur du paquetage qui dépendent de modules à l'extérieur de paquetage. *Ce* est un indicateur de l'indépendance de paquetage.

— Instability (*I*) : I est le rapport du couplage efférent (Ce) sur le couplage total (Ce + Ca) de telle sorte que I = Ce / (Ce + Ca). Cette mesure est un indicateur de la capacité de résistance du paquetage au changement. Une valeur de zéro indique un paquetage complètement stable et une valeur de un indique un paquetage complètement instable.

— Normalized Distance from Main Sequence (*Dn*) : est la distance perpendiculaire normalisée du paquetage de la ligne idéalisée A + I = 1. Cette mesure est un indicateur de la balance du paquetage entre l'abstraction et la stabilité.

4.4.3 Évaluation

Pour mesurer l'impact de l'application du paradigme de l'AOP sur la qualité de la conception de Japrosim tel que la flexibilité, maintenabilité, et de réutilisabilité, une évaluation automatique des métriques a été réalisée pour les deux versions OO et AO de Japrosim à l'aide de l'outil AOPMetrics. Nous avons enregistré la différence entre toutes les valeurs des mesures de CK pour les deux versions de Japrosim comme rapporté dans le tableau 4.1. Une amélioration notable est observée à peu près sur toutes les mesures individuelles, sauf pour la métrique *NOC* qui n'est pas affectée car il n'existe aucun cas dans la version OO de Japrosim où les sous-classes sont définies uniquement dans le but d'appliquer leur propre mise en œuvre du comportement aspectuel. En outre, la version AO de Japrosim n'offre aucune diminution de *WOM* après la mise en œuvre des aspects parce que le paradigme AO ne réduit pas nécessairement le nombre d'opérations dans le corps fonctionnel de l'application, mais ajoute au moins une méthode dans le code des aspects comme une advice. D'autre part, l'augmentation de *CDA* et *CAE* se réfère à la présence d'aspects : *CDA* donne une idée de l'impact global d'un aspect sur les autres modules et une valeur élevée de *CDA* est généralement souhaitable. La métrique CAE indique le degré de couplage entre le module donné et l'aspect contenant des advices. Ce type de couplage est absent dans le système orienté-objet [20].

Les métriques des logiciels sont des indicateurs des attributs externes de la qualité de logiciels orientés-aspect comme la maintenabilité, l'intelligibilité, la réutilisabilité, manque de prédisposition, et la stabilité [36]. Par exemple, dans notre étude de cas, la réduction du nombre de lignes de code conduit à l'amélioration de la maintenabilité, la compréhensibilité et de stabilité de Japrosim. En outre, la diminution de *LOC* et *CBM* permet d'améliorer la modularité [78] .

Les résultats présentés sur la figure 4.8 ont été obtenus à partir de la mesure des paquetages de la version AO de Japrosim. Le paquetage qui contient les aspects est caractérisé par la haute valeur de la métrique *Ce*, car il met en œuvre toutes les préoccupations transversales d'autres paquetages et la valeur nulle de *Ca* qui exprime l'inconscience du

TABLEAU 4.1 : Comparaison entre les versions OO et AO de Japrosim [24].

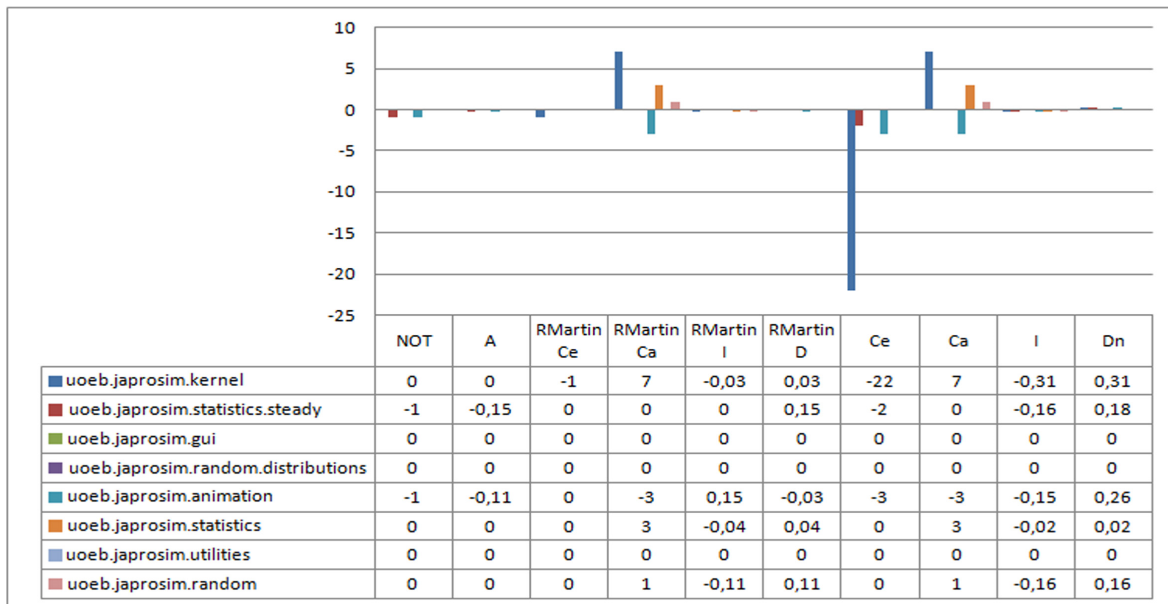
Métriques	**La version orientée objet**	**Corps**	**Les aspects**	**globale**	**Différence**	**gagnant**
LOCC	4438	3499	770	4269	-169	AO
WOM	417	383	87	470	53	OO
DIT	52	48	0	48	-4	AO
NOC	13	13	0	13	0	NONE
CFA	60	33	26	59	-1	AO
CMC	114	68	42	110	-4	AO
CBM	149	90	53	143	-6	AO
CDA	0	0	34	34	34	OO
CAE	0	0	3	3	3	OO
RFM	510	409	28	437	-73	AO
LCO	1864	1643	0	1643	-221	AO

Package name	NOT	A	RMartin Ce	RMartin Ca	RMartin I	RMartin D	Ce	Ca	I	Dn
uoeb.japrosim.crosscutting.concerns	10	0,2	9	0	1	0,2	43	0	1	0,2
uoeb.japrosim.kernel	7	0,285714	5	42	0,106383	0,607903	7	42	0,142857	0,571429
uoeb.japrosim.statistics.steady	4	0,25	3	2	0,6	0,15	2	2	0,5	0,25
uoeb.japrosim.gui	9	0	3	2	0,6	0,4	2	2	0,5	0,5
uoeb.japrosim.random.distributions	19	0,052632	18	2	0,9	0,047368	2	2	0,5	0,447368
uoeb.japrosim.animation	8	0	2	3	0,4	0,6	1	3	0,25	0,75
uoeb.japrosim.statistics	2	0	2	11	0,153846	0,846154	1	11	0,083333	0,916667
uoeb.japrosim.utilities	3	0	3	0	1	0	5	0	1	0
uoeb.japrosim.random	6	0,166667	5	2	0,714286	0,119048	1	2	0,333333	0,5

FIGURE 4.8 : Les résultats des mesures pour les paquetages de la version AO de Japrosim à l'aide de l'outil AOPMetrics [27].

paradigme AO. Le paquetage *crosscutting* est complètement instable ce qui confirme son caractère transversal sans aucune préoccupation fonctionnel.

Les différences entre toutes les valeurs des métriques de dépendances des paquetages des deux versions orienté-objet et orienté-aspect de Japrosim sont enregistrées tel que présenté dans la figure 4.9. Un changement remarquable équivalent à environ 62 % dans des paquetages de Japrosim est obtenu à l'exception des paquetages :
uoeb.Japrosim.gui, *uoeb.Japrosim.utilities*, et *uoeb.Japrosim.random.distributions* qui ne sont pas touchés, car ils ont peu d'interaction avec d'autres paquetages dans la version OO de Japrosim. En outre, la version AO de Japrosim offre une diminution totale de la valeur de la métrique I après la mise en œuvre des aspects en raison des paquetages qui sont suffisamment souples pour être étendu sans nécessité de modification. Le principe "Ouvert/Fermé" [67] est mis en œuvre de manière idéale grâce à la spécification du *uoeb.Japrosim.crosscutting.concerns* qui a une résilience totale et une valeur élevée de la métrique *Ce*. Par ailleurs, l'augmentation globale de *Ce* donne une idée de l'amélioration de l'indépendance des paquetages, ce qui augmente leur réutilisation. Une diminution de *NOT* et *A* à la fois pour les paquetages *uoeb.Japrosim.animation* et *uoeb.Japrosim.statistics.steady* est d*û*e au transfert des interfaces *EventObserver* et *StatObserver* aux aspects : *Animation* et *SteadyStateDetection* respectivement. En outre, l'augmentation de la métrique Ca se réfère à la présence de paquetage supplémentaire qui est *uoeb.Japrosim.crosscutting.concerns*. Enfin, l'augmentation de la valeur *Dn* se rapporte à la variation de la valeur de I .



	NOT	A	RMartin Ce	RMartin Ca	RMartin I	RMartin D	Ce	Ca	I	Dn
uoeb.japrosim.kernel	0	0	-1	7	-0,03	0,03	-22	7	-0,31	0,31
uoeb.japrosim.statistics.steady	-1	-0,15	0	0	0	0,15	-2	0	-0,16	0,18
uoeb.japrosim.gui	0	0	0	0	0	0	0	0	0	0
uoeb.japrosim.random.distributions	0	0	0	0	0	0	0	0	0	0
uoeb.japrosim.animation	-1	-0,11	0	-3	0,15	-0,03	-3	-3	-0,15	0,26
uoeb.japrosim.statistics	0	0	0	3	-0,04	0,04	0	3	-0,02	0,02
uoeb.japrosim.utilities	0	0	0	0	0	0	0	0	0	0
uoeb.japrosim.random	0	0	0	1	-0,11	0,11	0	1	-0,16	0,16

FIGURE 4.9 : La différence des résultats de mesures des métriques des paquetages pour les deux versions AO et OO de Japrosim [27].

4.5 Conclusion

Dans ce chapitre, nous avons traité les principales préoccupations transversales de Japrosim. Ainsi, nous avons proposé des aspects utilisant le langage AspectJ pour les séparation de ses préoccupations fonctionnelles. Pour évaluer l'impact du paradigme orienté aspect sur ce cardiciel en terme de qualités telles que la maintenabilité, la réutilisabilité, et testabilité ; une évaluation automatique des métriques les plus importantes pour les deux versions OO et AO de Japrosim a été réalisée en utilisant l'outil *AOPMetrics*. Les résultats obtenus montrent que la séparation des préoccupations transversales de Japrosim améliore ses attributs de qualité. Dans le prochain chapitre, on va exploiter les constats discutés dans ce chapitre au niveau implémentation. Ainsi, le profile UML proposé et discuté précédemment sera appliqué pour les modéliser.

Chapitre 5

Profile AspectJ : Implémentation

> « *Trois idéaux ont éclairé ma route et m'ont souvent redonné le courage d'affronter la vie avec optimisme :la bonté ,la beauté et la vérité.* »
>
> Albert Einstein

Sommaire

5.1 Introduction **88**

5.2 L'environnement de développement du profile **88**

5.3 Génération de code **89**

5.3.1 Le modèle templates et metamodel 89

5.3.2 Xpand 89

5.3.3 Spécification de règles de transformation 91

5.4 Application du profile **94**

5.5 Conclusion **98**

5.1 Introduction

Dans les chapitres précédents, nous avons proposé un profile UML pour la spécification des concepts du langage AspectJ au niveau de la conception. Ce profile est développé en utilisant l'outil *Papyrus*. Le pluging Papyrus fournit un outil de modélisation UML de haut niveau en s'appuyant sur *Eclipse UML2*. Cet outil offre la possibilité d'utiliser les différents diagrammes d'UML afin de manipuler des modèles UML2. En plus, il offre un support complet pour le développement de profiles UML. La génération automatique de code offre de nombreux avantages comme le développement rapide d'un code de haute qualité et la réduction des erreurs de programmation accidentelles. Dans ce chapitre, nous présentons les détails d'implémentation du profile AspectJ. En outre, un mécanisme est développé spécialement pour la génération de code adéquat aux modèles UML en terme de codes Java et AspectJ. Enfin, nous discutons l'application de ce profile brièvement sur le modèle UML de la bibliothèque Japrosim.

5.2 L'environnement de développement du profile

Le profile AspectJ est basé sur l'extension d'UML par stéréotypage. Il est développé en utilisant la dernière version de papyrus. Papyrus est un outil open source pour la modélisation UML2 [62]. En plus, il adopte XMI (XML Metadata Interchange) comme format d'échange persistant pour déployer le profile UML dans les outils disponibles. XMI utilise la syntaxe XML et donc il a tous les avantages de XML. En plus, il permet de capturer le métamodèle proposé sous une forme spécifique, formelle et persistante pour définir les transformations modèle à modèle (M2M) et modèle à texte (M2T). La figure 5.1 illustre son architecture .

Papyrus est développé par le CEA LIST depuis Février 2007. Il vise à fournir un environnement intégré facile à utiliser pour éditer les modèles de type EMF (Eclipse Modeling Framework), il soutient en particulier UML et les langages de modélisation connexes tels que SysML et MARTE. Il est intégré comme pluging au projet Eclipse depuis octobre 2008. La figure 5.2 illustre sa fenêtre principale. Papyrus offre également un support très avancé pour les profils UML qui permet aux utilisateurs de définir des éditeurs pour les DSL (Domain Specific Language) basés sur le standard UML 2 [62].

FIGURE 5.1 : L'architecture de papyrus [44].

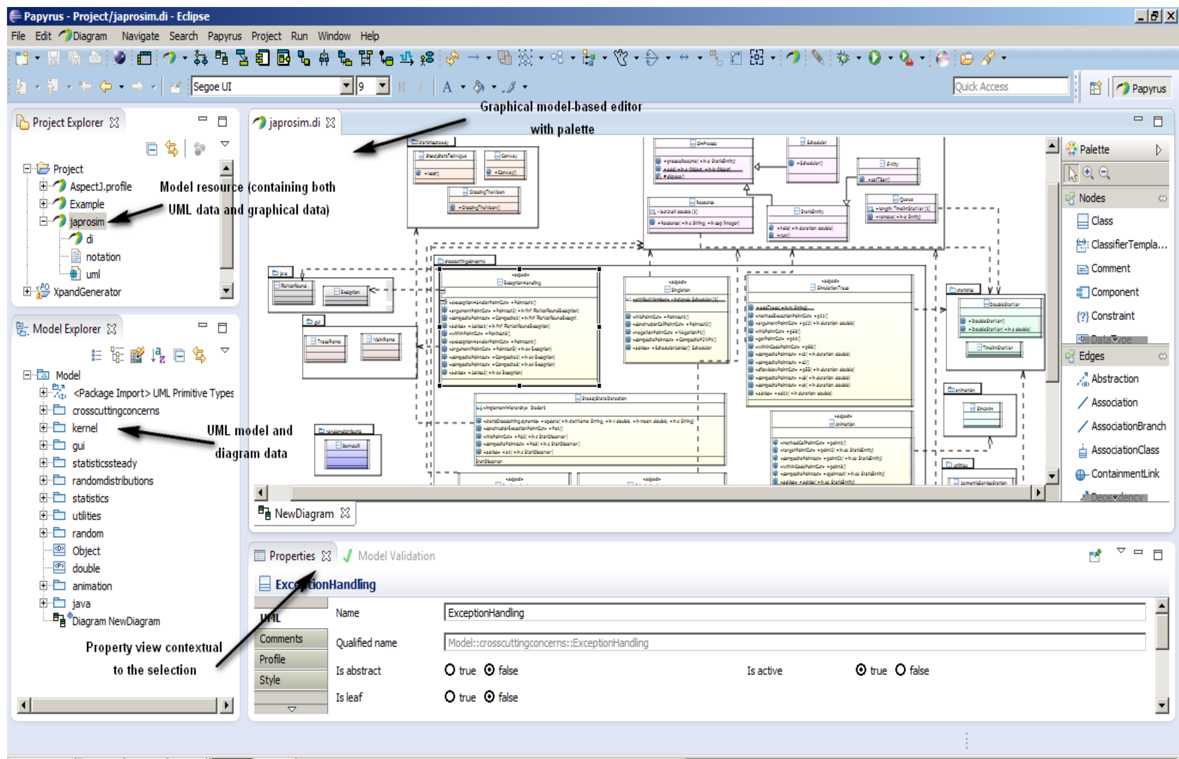


FIGURE 5.2 : Fenêtre principale de papyrus.

5.3 Génération de code

5.3.1 Le modèle templates et metamodel

La génération de code est considérée comme une transformation *model-to-text* dans le cadre de l'élaboration de l'IDM (Ingénierie Dirigée par les modèles). Il ya plusieurs raisons qui laissent cette phase primordiale tels que l'augmentation de la flexibilité du système pour des raisons de performance, la minimisation de la taille du programme à exécuter dans le cas d'une quantité limitée de mémoire, l'évitement de l'allocation dynamique de la mémoire lors de l'exécution, le développement des applications à un niveau d'abstraction plus élevé que le niveau prévu par l'utilisation du langage de programmation, et la détection précoce des erreurs de conception ou de mise en œuvre avant l'exécution. Il existe sept modèles tés répandus pour la génération de code : *templates et filtering, templates et metamodel, frame processing, API-based generators, inline code generation, code attributes*, et *code weaving*. Le modèle templates et metamodel fournit un métamodèle personnalisable explicite pour la génération de code. Il se compose de deux étapes. En premier temps, le modèle est instancié basé sur le métamodèle. Ensuite, le processus de génération de code écrit les templates en termes de ce métamodèle. Le métamodèle peut être adapté pour inclure des concepts spécifiques à un domaine particulier, c-à-d, les profiles UML [91].

5.3.2 Xpand

Le modèle "templates & metamodel" est utilisé à l'aide du langage de template Xpand qui est un moteur de template très répandu. Ce modèle a un grand avantage où les modèles d'instanciation sont complètement séparés des templates qui contrôlent la génération de code comme illustré à la figure 5.3. Il est donc plus facile de changer le format du

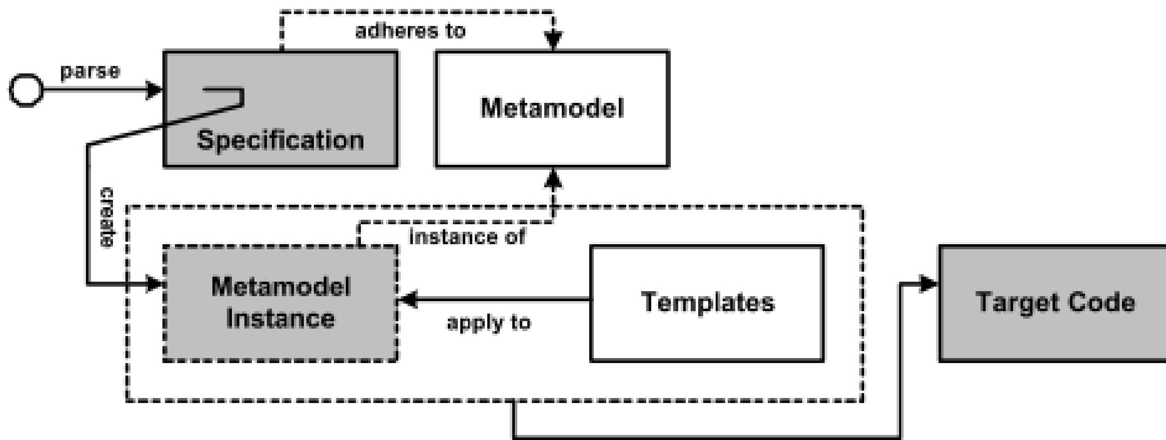


FIGURE 5.3 : Le principe de modèle templates et metamodel [91].

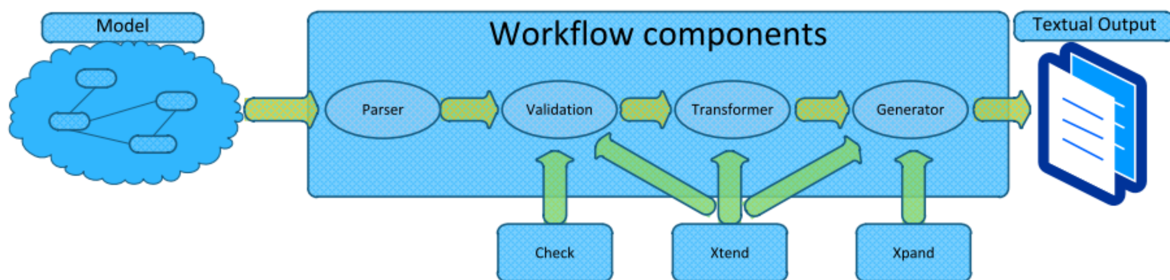


FIGURE 5.4 : Le principe de moteur workflow [81].

modèle sans avoir à adapter les templates. Il est également possible d'écrire les templates en termes de métamodèle et non en termes de détails de bas niveau de la représentation du modèle. Par conséquent, les modèles deviennent plus lisible et plus facile à entretenir. En outre, il est possible d'intégrer des attributs et des comportements spécifiques à un domaine particulier dans le métamodèle. Le métamodèle est également un endroit idéal pour inclure des contraintes de modélisation. Celles-ci seront évaluées par le générateur lorsque le métamodèle est instancié lors de la génération de code. Puisque la dernière version de papyrus n'a pas un support pour la validation des contraintes OCL, le langage *check* est la solution adoptée pour exprimer les contraintes dans notre cas. Ce langage est très facile à comprendre et à utiliser. Fondamentalement, il est construit autour de la syntaxe du langage *expression* qui est un mélange syntaxique de Java et OCL. Les contraintes spécifiées dans ce langue doivent être stockées dans un fichier indépendant avec l'extension (.chk). De ce fait, le langage de template est simple et facile à comprendre [45]. Xpand est un composant dans le projet M2T pour la génération de code à partir de modèles. C'est le moteur de template qui est utilisée avec Eclipse Modelling Framework (EMF). Il a lui-même une syntaxe de base, mais utilise un langage d'expression sous-jacent *Xtend* pour fournir de puissantes capacités M2T (et même M2M). En plus, il offre des capacités orientée-aspect qui prête à ses fonctionnalités d'extensibilité. Xpand est utilisé au sein de EMFT workflow comme le montre la figure 5.4. Les templates Xpand sont invoquées conformément au modèle de workflow (*.mwe). Le composant workflow se trouve dans le projet EMFT et est livré avec son propre Wizard et éditeur [35].

5.3.3 Spécification de règles de transformation

Un ensemble de templates Xpand sont proposées pour générer les paquetages, les classes et les interfaces Java du modèle UML de base et un autre ensemble pour la génération du code AspectJ, comme une interprétation des stéréotypes du profile AspectJ. L'algorithme 1 représente le générateur de code (code Xpand).

Algorithm 1 : The generator

Input : UML model, Xpand templates
Output : Java and AspectJ code

```
1  for all packages of UML model do
2      for all classes of package do
3          generate Java code according to class template
4      for all interfaces of package do
5          generate Java code according to interface template
6      for all classes stereotyped aspect of package do
7          for all operations of aspect do
8              generate Java code according to UML Operation Define block
9          for all attributes of aspect do
10             generate Java code according to UML Property Define block
11         for all operations stereotyped pointcut of aspect do
12             generate AspectJ code according to pointcut Define blocks
13         for all operations stereotyped advice of aspect do
14             generate AspectJ code according to advice Define blocks
15         for all attributes stereotyped StaticCrosscutting.static of aspect do
16             generate AspectJ code according to Crosscutting.static Define blocks
17         for all operations stereotyped StaticCrosscutting.dynamic of aspect do
18             generate AspectJ code according to Crosscutting.dynamic Define blocks
19         for all nested classes of aspect do
20             generate Java code according to class template
21         for all nested interfaces of aspect do
22             generate Java code according to interface template
23         for all nested classes of aspect stereotyped aspect do
24             generate AspectJ code according to aspect template
25     for all nested packages of package do
26         generate AspectJ and Java code according to package template
```

A titre d'illustration un fragment de code du template *Aspect* qui génère les différents types de coupes du langage AspectJ à partir de modèles UML est présenté sur le listing 5.1.

```
«DEFINE OperationRoot FOR MethodCallPointCut»
2 «this.visibility» «IF this.isAbstract-»abstract pointcut «name»(); «ELSE» pointcut
    «name»():(«FOREACH Method AS m SEPARATOR ' || '»call(«m.visibility» «IF m.
    getReturnResult().type.name==null-»void«ELSE-»«m.getReturnResult().type.name»«
    ENDIF-» «m.class.name».«m.name»(«FOREACH m.ownedParameter AS p SEPARATOR ','-»«
    p.type.name»«ENDFOREACH-»))«ENDFOREACH»);
```

```
«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR MethodExecutionPointCut»
«visibility» «IF this.isAbstract−»
abstract pointcut «name»();«ELSE»pointcut «name»():(«FOREACH Method AS m SEPARATOR
    ' || '»execution(«m.visibility» «IF m.getReturnResult().type.name==null−»void«
    ELSE−»«m.getReturnResult().type.name»«ENDIF−» «m.name»(«FOREACH m.
    ownedParameter AS p SEPARATOR ','−»«p.type.name»«ENDFOREACH−»))«ENDFOREACH»);

«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR GetPointCut»
«visibility» «IF this.isAbstract−»
abstract pointcut «name»();«ELSE−»pointcut «name»():(«FOREACH Field AS f SEPARATOR
    ' || '»get(«f.visibility» «f.type.name» «f.name»)«ENDFOREACH»);
«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR SetPointCut»
«visibility» «IF this.isAbstract−»
abstract pointcut «name»();«ELSE»pointcut «name»() : («FOREACH Field AS f SEPARATOR
    ' || '»set(«f.visibility» «f.type.name» «f.name»)«ENDFOREACH»);
«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR AdviceExecutionPointCut»
«visibility» «IF this.isAbstract−»
abstract pointcut «name»();«ELSE»pointcut «name»() : adviceexecution();
«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR ExeceptionHandlerPointCut»
«visibility» «IF this.isAbstract−»abstract pointcut «name»();«ELSE»pointcut «name»
    ():(«FOREACH type AS t»handler(«t.name»)«ENDFOREACH»);
«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR WithinPointCut»
«visibility» «IF this.isAbstract−»abstract pointcut «name»();«ELSE−»pointcut «name»
    ():(«FOREACH type AS t»within(«t.name»)«ENDFOREACH»);
«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR WithinCodePointCut»
«visibility» «IF this.isAbstract−»
abstract pointcut «name»();«ELSE»pointcut «name»():(«FOREACH Method AS m SEPARATOR
    ' || '»withincode(«m.visibility»«IF m.getReturnResult().type.name==null−» void«
    ELSE−»«m.getReturnResult().type.name»«ENDIF−» «m.name»(«FOREACH m.
    ownedParameter AS p SEPARATOR ','−»«p.type.name»«ENDFOREACH−»))«ENDFOREACH»);
«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR CflowPointCut»
«visibility» «IF this.isAbstract−»abstract pointcut «name»(«FOREACH ownedParameter
    AS p SEPARATOR ','−»«p.type.name» «p.name»«ENDFOREACH−»);«ELSE»pointcut «name»(
    «FOREACH ownedParameter AS p SEPARATOR ','−»«p.type.name» «p.name»«ENDFOREACH−»
    ):cflow(«SelectedPointCut.base_Operation.name»(«FOREACH SelectedPointCut.
    base_Operation.ownedParameter AS p SEPARATOR ','−»«p.name»«ENDFOREACH−»));
«ENDIF−»
«ENDDEFINE»
«DEFINE OperationRoot FOR CflowblowPointCut»
«visibility» «IF this.isAbstract−»abstract pointcut «name»(«FOREACH ownedParameter
    AS p SEPARATOR ','−»«p.type.name» «p.name»«ENDFOREACH−»);«ELSE−»pointcut «name»
    («FOREACH ownedParameter AS p SEPARATOR ','−»«p.type.name» «p.name»«ENDFOREACH−
    »):cflowbelow(«SelectedPointCut.base_Operation.name»(«FOREACH SelectedPointCut.
```

```
    base_Operation.ownedParameter AS p SEPARATOR ','-»«p.name»«ENDFOREACH-»));
«ENDIF-»
«ENDDEFINE»
«DEFINE OperationRoot FOR TargetPointCut»
«visibility» «IF this.isAbstract-»
abstract «ENDIF-»pointcut «name»(«FOREACH ownedParameter AS p SEPARATOR ','-»«p.
    type.name» «p.name»«ENDFOREACH-»)«IF this.isAbstract-»
;«ELSE» «IF arguments.isEmpty»target(«type.toList().get(0).name»); «ELSE»: target(«
    arguments.toList().get(0).toString()»); «ENDIF»«ENDIF»
«ENDDEFINE»
«DEFINE OperationRoot FOR ThisPointCut»
«visibility» «IF this.isAbstract-»
abstract «ENDIF-»pointcut «name»(«FOREACH ownedParameter AS p SEPARATOR ','-»«p.
    type.name» «p.name»«ENDFOREACH-»)«IF this.isAbstract-»
;«ELSE» «IF arguments.isEmpty»: this(«type.toList().get(0).name»); «ELSE»: this(«
    arguments.toList().get(0).toString()»); «ENDIF»«ENDIF»
«ENDDEFINE»
«DEFINE OperationRoot FOR ArgumentPointCut»
«visibility» «IF this.isAbstract-»abstract «ENDIF-»pointcut «name»(«FOREACH
    ownedParameter AS p SEPARATOR ','-»«p.type.name» «p.name»«ENDFOREACH-»)«IF this
    .isAbstract-»
; «ELSE» «IF arguments.isEmpty»: args(«FOREACH type AS t SEPARATOR ','»«t.name» «
    ENDFOREACH»); «ELSE»: args(«FOREACH arguments AS a SEPARATOR ','»«a.toString()»
    «ENDFOREACH»);«ENDIF»
«ENDIF»
«ENDDEFINE»
«DEFINE OperationRoot FOR ConditionalPointCut»
«visibility» «IF this.isAbstract-»abstract pointcut «name»();«ELSE-»pointcut «name»
    (): if(«Condition»);«ENDIF-»
«ENDDEFINE»
«DEFINE OperationRoot FOR StaticInitializationPointCut»
«visibility» «IF this.isAbstract-»abstract pointcut «name»();«ELSE-»pointcut «name»
    () : («FOREACH type AS a SEPARATOR ' || '»staticinitialization(«a.name»)«
    ENDFOREACH»);
«ENDIF-»
«ENDDEFINE»
«DEFINE OperationRoot FOR ObjectInitializationPointCut»
«this.visibility» «IF this.isAbstract-»abstract  pointcut «name»();«ELSE-»pointcut
    «name»():(«FOREACH Method AS m SEPARATOR ' || '»initialization(«m.visibility» «
    m.name».new(«FOREACH m.ownedParameter AS p SEPARATOR ','-»«p.type.name»«
    ENDFOREACH-»))«ENDFOREACH»);
«ENDIF-»
«ENDDEFINE»
«DEFINE OperationRoot FOR ObjectPreInitializationPointCut»
«this.visibility» «IF this.isAbstract-»abstract pointcut «name»();«ELSE»pointcut «
    name»():(«FOREACH Method AS m SEPARATOR ' || '»preinitialization(«m.visibility»
     «m.name».new(«FOREACH m.ownedParameter AS p SEPARATOR ','-»«p.type.name»«
    ENDFOREACH-»))«ENDFOREACH»);
«ENDIF-»
«ENDDEFINE»
«DEFINE OperationRoot FOR ConstructorCallPointCut»
«this.visibility» «IF this.isAbstract-»abstract pointcut «name»();«ELSE»pointcut «
    name»():(«FOREACH Method AS m SEPARATOR ' || '»call(«m.visibility» «m.name».new
    («FOREACH m.ownedParameter AS p SEPARATOR ','-»«p.type.name»«ENDFOREACH-»))«
    ENDFOREACH»);
«ENDIF-»
«ENDDEFINE»
«DEFINE OperationRoot FOR ConstructorExecutionPointCut»
«this.visibility» «IF this.isAbstract-»abstract pointcut «name»();«ELSE» pointcut «
```

```
    name»():(«FOREACH Method AS m SEPARATOR ' || '»execution(«m.visibility» «m.name
    ».new(«FOREACH m.ownedParameter AS p SEPARATOR ','-»«p.type.name»«ENDFOREACH-»)
    )«ENDFOREACH»);
«ENDIF-»
84 «ENDDEFINE»
«DEFINE OperationRoot FOR NegationPointCut»
86 «visibility» pointcut «name»(«FOREACH ownedParameter AS p SEPARATOR ','-»«p.type.
    name» «p.name»«ENDFOREACH-»): !«ComposeOf.base_Operation.name»(«FOREACH
    ownedParameter AS p SEPARATOR ','-»«p.type.name» «p.name»«ENDFOREACH-»);
«ENDDEFINE»
88 «DEFINE OperationRoot FOR CompositePointcut»
«visibility» pointcut «name»(«FOREACH ownedParameter AS p SEPARATOR ','-»«p.type.
    name» «p.name»«ENDFOREACH-») : «ComposeOf.toList().get(0).base_Operation.name»(
    «FOREACH ComposeOf.toList().get(0).base_Operation.ownedParameter AS p1
    SEPARATOR ','-»«p1.name»«ENDFOREACH-»)«IF CompositionOperator.name=='AND'» && «
    ELSE» || «ENDIF»«ComposeOf.toList().get(1).base_Operation.name»(«FOREACH
    ComposeOf.toList().get(1).base_Operation.ownedParameter AS p2 SEPARATOR ','-»«
    p2.name»«ENDFOREACH-»);
90 «ENDDEFINE»
```

Listing 5.1 : Fragment de la template *Aspect*

5.4 Application du profile

Afin de valider le profil proposé, il a été appliqué dans le domaine de la modélisation de simulation. La bibliothèque Japrosim est l'étude de cas sélectionnée, notre première contribution est détaillée en [26]. Le paquetage *crosscutting.concerne* discuté dans le chapitre précédent contient tous les aspects de la version AO de Japrosim : Animation, Singleton, SimulationTrace, Sychronization, GraphicalUserInterface, CalculationAccuracy and SteadyStateDetection. Ainsi, l'utilisation de cette étude de cas, nous donne la possibilité de tester la plupart des éléments du profile. La figure 5.5 illustre une partie du modèle japrosim et spécifiquement le paquetage *crosscutting.concern* après l'application du profil proposé.

Par exemple, l'aspect *Singleton*, comme le montre la figure 5.6, possède trois coupes primitives : PointCut1 stéréotypé constructorCallPointCut, Pointcut2 stéréotypé thisPointCut et NegationP1stereotyped negationPointCut. En plus de cela, il possède une coupe composite qui est CompositeP2NP1. Elle se compose de deux coupes qui sont NegationP1et Pointcut2 utilisant respectivement l'opérateur binaire *AND* comme illustré sur la figure 5.7. En outre, Singleton comporte un élément statique qui est l'attribut d'instance. Cet attribut est stéréotypé attributMember. En plus, un around advice qui est déclarée à l'intérieur de l'aspect *Singleton* et qu'il a "Scheduler" comme un type de retour. Rappelons que dans le langage UML, les métaclasses qui étendent d'autres métaclasses existantes deviennent des stéréotypes et leurs attributs deviennent des tags values, après l'application du profile comme le montre la figure 5.8. L'aspect *Singleton* est une classe stéréotypée aspect. L'attribut IsPrivileged est devenu le tag IsPrivileged du stéréotype *Singleton* qui a une valeur *true*. La méta-classe d'aspect est associée avec les métaclasses pointcut, advices, StaticCrosscutting.dynamic et StaticCrosscutting.static. Par conséquent, chaque stéréotype aspect a les tags : pointcut, des advice, et StaticCrosscutting.dynamic StaticCrosscutting.static respectivement, par exemple, Advice tag a *SchedulerAdvice* valeur.

Si le tag AspectInstance possède la valeur *singleton*, les tags perPointCut et pertypewithin auront une valeur nulle.

FIGURE 5.5 : Les aspects de Japrosim après l'application du profile AspectJ.

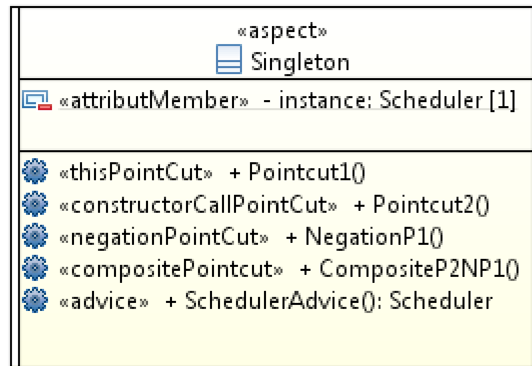


FIGURE 5.6 : L'aspect Singleton.

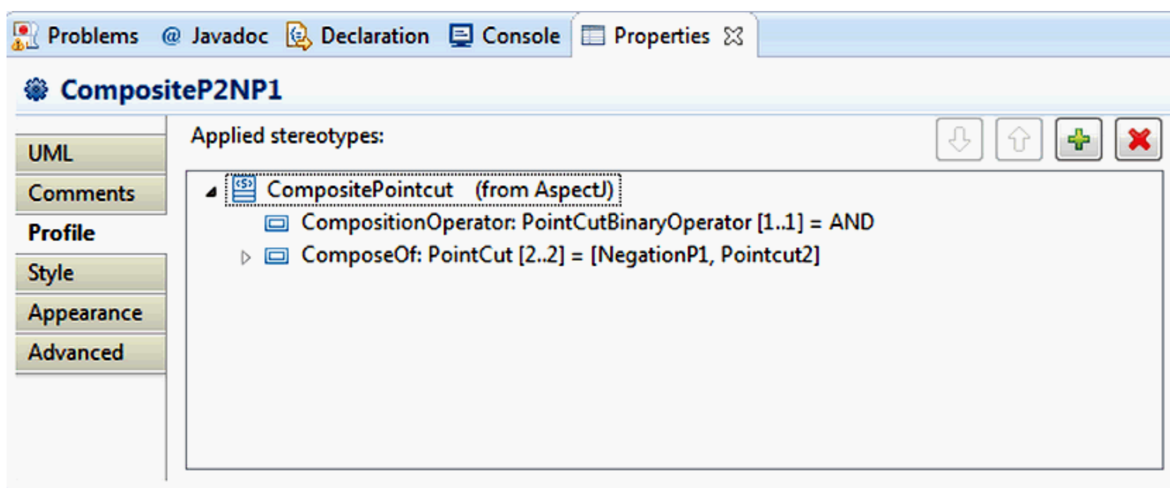


FIGURE 5.7 : La coupe CompositeP2NP1.

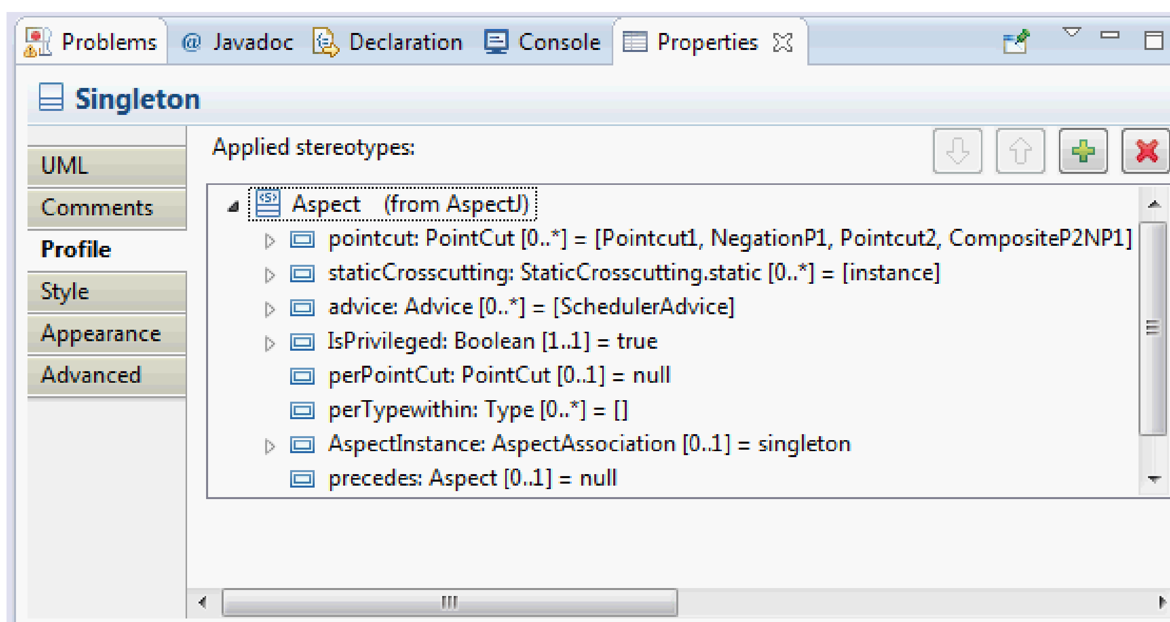


FIGURE 5.8 : Les Tag values de Singleton stereotype.

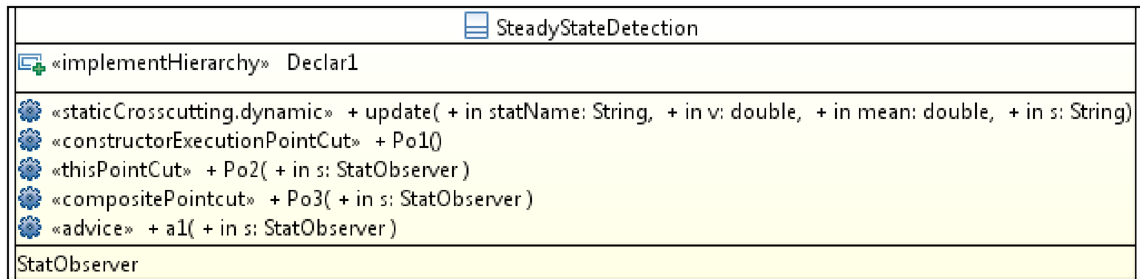


FIGURE 5.9 : L'aspect SteadyStateDetection.

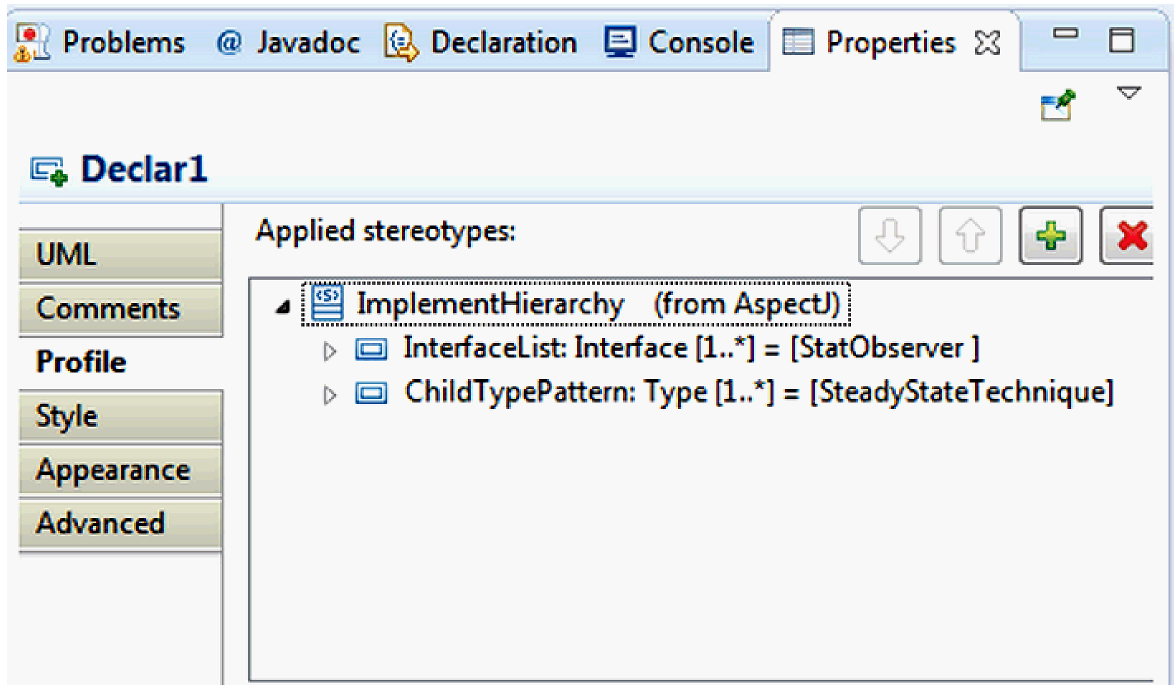


FIGURE 5.10 : La préoccupation transversal statique Declar1.

Par ailleurs, l'aspect *SteadyStateDetection* comme montre la figure 5.9 possède deux coupes primitives : Po1 stéréotypée constructorExecutionPointCut et Po2 stéréotypée thisPointCut. En plus de cela, il a une coupe composite Po3 qui est composée des deux coupes primitives citées avant en utilisant l'opérateur binaire *AND*. *SteadyStateDetection* possède deux membres transversaux statiques qui sont : update stéréotypé StaticCrosscutting.dynamic et Declar1 stéréotypés implementHierarchy. Ce dernier modifie la hiérarchie de la classe SteadyStateTechnique par la mise en œuvre de l'interface nested StatObserver comme représenté sur la figure 5.10. En fin, une advice AfterFinally est déclaré à l'intérieur de l'aspect SteadyStateDetection.

Après l'application du profile AspectJ comme montre la figure 5.11. La classe *SteadyStateDetection* est stéréotypée Aspect. L'attribut IsPrivileged est devenu le tag IsPrivileged du stéréotype *SteadyStateDetection* qui a une valeur *true*. Le stéréotype *SteadyStateDetection* a les tags pointcut, advice, StaticCrosscutting.dynamic et StaticCrosscutting.static respectivement, par exemple, le tag StaticCrosscutting.dynamic a la méthode *update* comme valeur.

Le projet Xpand importe le profile AspectJ et le modèle de Japrosim sous forme de fichiers UML2 et utilise le fichier chek de validation qui contient les métaclasses de profile avec les contraintes associées afin de valider les modèles UML (le modèle Japrosim

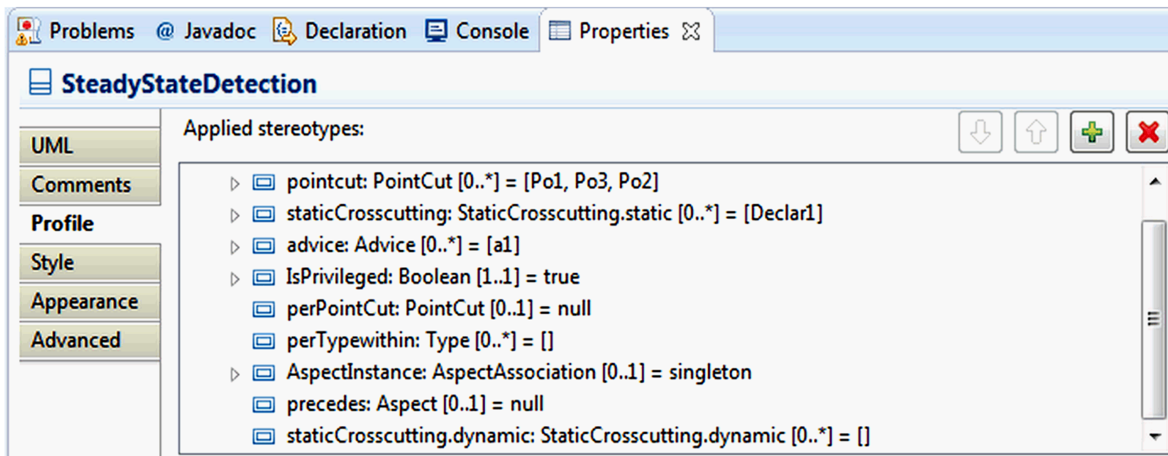


FIGURE 5.11 : Les Tag values de SteadyStateDetection.

dans notre cas). Les Templates “Aspect”, “Class”, “Interface ”, “main” et “package”sont utilisés pour la génération de code Java et AspectJ. Enfin, le fichier de workflow “WorkflowFile.mwe ”est utilisé pour invoquer l’exécution des templates de Xpand. Le résultat est les huit paquetages de la bibliothèque Japrosim en plus du paquetage des préoccupations transversales qui comprend les sept aspects comme montre la figure 5.12.

5.5 Conclusion

Nous avons présenté dans ce chapitre les différents détails d’implémentation du profile AspectJ. L’environnement de modélisation graphique a été mis en œuvre utilisant l’outil Papyrus. A partir du modèle UML et les contraintes exprimées en langage *check*, une génération automatique de code est faite conformément au profile (métamodèle) et en respectant les contraintes préétablies. Enfin, une illustration de l’application du profile est réalisé avec succès en utilisant la bibliothèque Japrosim. Cette étude de cas nous a permis de tester la plupart des concepts mais reste insuffisante et d’autres études de cas sont nécessaires pour valider davantage ce profile.

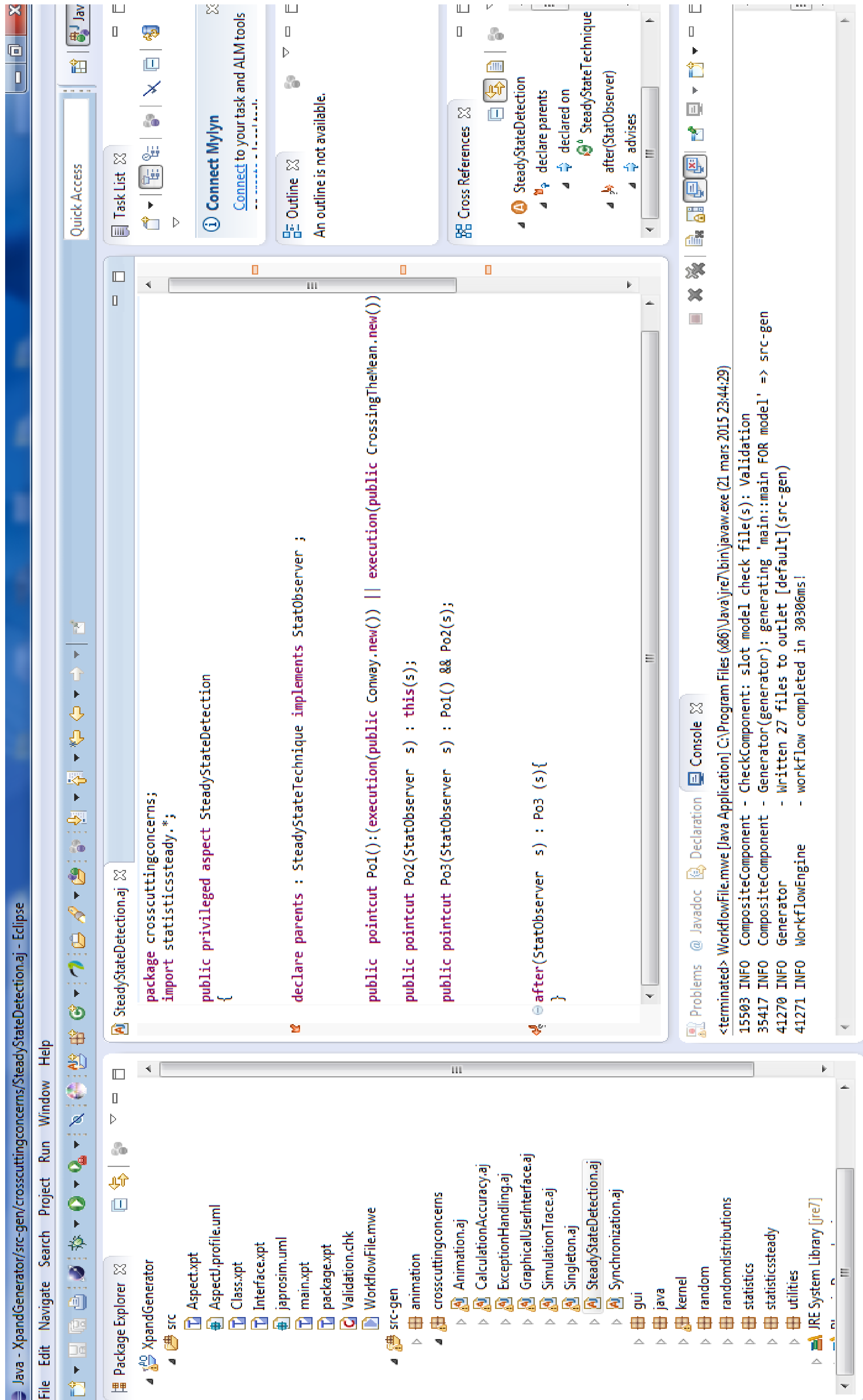


FIGURE 5.12 : Code généré à partir du modèle de la figure 5.5.

Conclusion générale et perspectives

Ce manuscrit discute les possibilités d'utilisation de la programmation orientée aspect dans le domaine de la modélisation et de la simulation à évènements discrets. L'AOP offre des nouvelles perspectives en se basant sur un principe bien connu en génie logiciel : la séparation des préoccupations. Le domaine de la modélisation et de la simulation est pluridisciplinaire. Il permet d'étudier des phénomènes variés et des systèmes existants ou à concevoir. Les applications issues de ce domaine méritent d'être au coeur des évolutions du génie logiciel. L'application de l'AOP dans ce domaine constitue un axe de recherche novateur et d'actualité.

La simulation orientée aspect est un domaine de recherche prometteur qui offre une solution au problème principal posé par la simulation orientée objet à savoir : la séparation des préoccupations transversales de la simulation de façon modulaire. Les systèmes de simulation orientée aspect ont plusieurs caractéristiques comme la haute modularité, la réutilisabilité, la maintenabilité et la visibilité. Les résultats des travaux que nous avons entrepris dans le cadre de cette thèse peuvent être résumés comme suit :

L'identification des principales préoccupations transversales de simulation telles que la détection de l'état stationnaire, l'animation graphique, l'interface utilisateur graphique, la gestion des exceptions, la précision et l'exactitude des calculs, la trace de simulation, et la synchronisation qui dégradent la qualité des logiciels de simulation.

Au niveau expérimental, la version orientée objet de la bibliothèque *Japrosim* a été choisie comme un exemple pratique. Les principales préoccupations transversales de ce framework ont été identifiées et séparées. La nouvelle version orientée aspect de cette bibliothèque a été obtenue avec succès et elle est disponible à l'adresse :
http ://sourceforge.net/projects/Japrosim/files/AOP-Japrosim.

En outre, afin de prouver l'impact du paradigme orienté aspect (AO) sur *Japrosim* en terme de qualité de la conception et qui affecte les propriétés telles que : la flexibilité, la maintenabilité, et la réutilisation, une évaluation automatique des métriques logicielles pour les deux versions *OO* et *AO* de Japrosim a été réalisée en utilisant l'outil *AOPMetrics* qui est un outil de métriques commun pour les programmes *Java* et *AspectJ*.

Récemment, les idées de la programmation orientée aspect sont utilisées dans les étapes précoces du processus de développement de logiciels (*AOSD*) à l'aide d'un ensemble de techniques. Un domaine de recherche future serait d'évaluer les avantages de l'intégration des techniques de l'*AOSD* dans le domaine de la simulation. Notre contribution est la proposition d'un profile *UML* qui utilise la terminologie du langage *AspectJ*. C'est une description complète de la vision statique du langage AspectJ en termes d'aspects, advices, pointcut et *static crosscutting* utilisant le métamodèle d'*UML*. En plus, il est conforme au format XMI qui signifie qu'il est possible de manipuler et d'échanger le profile entre les *Case tools* d'UML. En plus, il permet la génération automatique de code qui a de nombreux avantages comme le développement rapide d'un code de haute qualité et l'élimination des erreurs de programmation accidentelles, et conserve les avantages

du paradigme AOP jusqu'au niveau de l'implémentation.

Tout au long de ce travail de thèse, des perspectives de recherches se sont dégagées :Il serait intéressant d'approfondir les études comparatives entre les approches de l'AOP et de leur applicabilité dans le domaine de la DEVS qui permettrait d'anticiper sur les orientations futures dans le domaine des nouveaux paradigmes de programmation. La programmation orienté sujet est une collection de bonnes idées qui sont complexe à implémenter à cause de l'absence des supports d'implémentation. La proposition d'une plateforme de simulation à évènement discrets basé sur la *SOP* serait une approche très intéressante. Aussi, l'utilisation de *XEROX Parc AOP* comme une approche de pivotage à laquelle on compare les autres approches de séparation des préoccupation(CF, AP, SOP...) grâce à des concepts de mapping. Ce choix est dû au fait que cette approche a maintenant atteint une certaine maturité et beaucoup de développeurs et programmeurs l'utilisent.

Un autre défi majeur concerne l'utilisation des autres langages de l'AOP comme *AspectS* qui offre un tissage dynamique et de comparer ses résultats avec ceux obtenus avec *AspectJ*. Cela nous aidera à choisir le meilleur langage de mise en œuvre pour la version orientée aspect de Japrosim.

Il serait, également, intéressant de mener plusieurs études de cas et faire des comparaisons pour confirmer l'utilité de l'AOP. Le même travail que nous avons fait avec Japrosim, pourrait aisément être entrepris pour les deux bibliothèques bien connues, open source et ayant des principes différents : *SSJ* : *http ://simul.iro.umontreal.ca/ssj/indexf.html* et *Desmo-J* : *http ://desmoj.sourceforge.net/home.html*

Il serait aussi intéressant d'étudier comment le profile UML proposé pourrait être développé pour d'autres langages orientés aspect tel que *AspectS* pour permettre le développement des modèles plateforme spécifique (*PSM*). En outre, le profile proposé pourrait être abstrait en un profile *UML* générique utilisé pour le développement des modèles plateforme indépendant (Pim dans la terminologie *MDA*). Des transformations pourraient ensuite être faites pour transformer les modèles orientée aspect (*PIM*) dans des modèles orientée aspect spécifiques (*PSM*) et à la fois au code.

Bibliographie

[2] A. Aksu, F. Belet, and B. Zdemir. Developing aspects for a discrete event simulation system. In *Proceedings of the Third Turkish Aspect-Oriented Software Development Workshop*, pages 84–93, Bilkent University, Ankara, Turkey, 2008.

[3] F. E. Alam, J. Evermann, and A. Fiech. Modelling for dynamic aspect-oriented development. In *Proceedings of the Int. Conf. The 2nd Canadian Conference on Computer Science and Software Engineering (C3S2E-09)*, pages 109–113, Montreal, Canada, 2009.

[4] O. Aldawud, T. Elrad, and A. Bader. A uml profile for aspect oriented modeling. In *Proceedings of Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA2001 (2001)*, 2001.

[5] O. Aldawud, T. Elrad, and A. Bader. Uml profile for aspect-oriented software development. In *Proceedings of of the 3rd Int. Workshop on Aspect Oriented Modeling at AOSD 2003*, Boston, Massachusetts, 2003.

[6] A. Amirat. *Une Approche Hybride pour la Séparation des Préoccupations avec Résolution de Conflits durant l'Ingénierie des Besoins.* PhD thesis, Faculté des Sciences de l'Ingénieur, Département d'Informatique, univérsité de Annaba, 2007.

[7] F. Asteasuain, B. Contreras, E. Estéevez, and P. R. Fillottrani. Evaluation of uml extensions for aspect oriented design. In *The IV JIISIC (Iberoamerican Conference on Software Engineering and Knowledge Engineering)*, Madrid, Spain, November 2004.

[8] J. Baltus. La programmation orientée aspect et aspectj : Présentation et application dans un système distribué. In *Mini-Workshop : Systèmes CoopératifsInstituts*, 2001.

[9] J. Banks, editor. *Handbook of Simulation. Principles, Methodology, Advances, Applications, and Practice.* Wiley-Interscience, 1998.

[10] D. Bardou. Roles, subjects and aspects : How do they relate ? In *Workshop on the aspect-oriented programming, ecoop98*, pages 8–15, 1998.

[11] R. Basekow. Subject oriented programming, 2000. Software engineering Seminar.

[12] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters. TRESE group, Department of Computer Science, University of Twente, 2001.

[13] S. Bieniasz, S. Ciszewski, and S. BŚnieżykiś. Multi-agent simulation of physical phenomena by means of aspect programming. In *Proceedings of The 6th international conference on Computational Science. Vassil N. Alexandrov (eds.) ICCS 2006*, volume 3993 of *LNCS*, pages 759–766, Heidelberg, 2006. Springer-Verlag.

[14] L. Blair, G. Blair, and A. Andersen. Separating functional behaviour and performance constraints : aspect oriented specification. Technical report, 1998.

[15] A. Bourouis. *une approche de modélisationSystèmes à Évènements Discrets utilisant les Concept-map*. PhD thesis, University of Batna, 2011.

[16] A. Bourouis and B. Belattar. Japrosim : A java framework for discrete event simulation. *Journal of Object Technology*, 7(1) :103–119, 2008.

[17] R. Boutaghane. Test automatique des préoccupations dans les langages a aspects. Master's thesis, UNIVERSITÉ 20 AOUT 1955-SKIKDA, 2010.

[18] J. Brichau, K. Mens, and K. Volder. Building composable aspect-specific languages with logic metaprogramming. In *Generative Programming and Component Engineering : Proceedings of the ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 Pittsburgh, PA, USA*, volume 2487, page 110127, Berlin, Germany, 2002. Springer-Verlag.

[19] W. Candillon and G. Vanwormhoudt. Programmation orientée aspects en php. Telecom LILLE1, 2005.

[20] M. Ceccato and P. Tonella. Measuring the effects of software aspectization. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004). (Tom tourwé, Magiel Bruntink, Marius Marin, David Shepherd,eds)*, Delft, The Netherlands, 2004.

[21] C. Chavez, A. Garcia, and C. Lucena. Some insights on the use of aspectj and hyper/j. In *Tutorial and Workshop on AspectOriented Programming and Separation of Concerns*, 2001.

[22] M. Chibani, B. Belattar, and A. Bourouis. Towards a uml meta model extension for aspect oriented modeling. In *Proceedings of the eighth Int. Conference on Software Engineering Advances ICSEA 2013*, pages 591–596, Venice, Italy, 2013.

[23] M. Chibani, B. Belattar, and A. Bourouis. The use of the aspect oriented programming (aop) paradigm in discrete event simulation domain : overview and perspectives. In *Proceedings of the Third International Conference on Digital Information Processing and Communications (ICDIPC2013)*, pages 635–660, Islamic Azad University (IAU), UAE, 2013. sdiwc digital-library. URL `http://sdiwc.net/digital-library/web-admin/upload-pdf/00000468.pdf`. ISBN : 978-0-9853483-3-5.

[1] M. Chibani, B. Belattar, and A. Bourouis. Toward an aspect-oriented simulation. *International Journal of New Computer Architectures and their Applications (IJNCAA)*, vol. 3 :1–10 pages, 2013.

[24] M. Chibani, B. Belattar, and A. Bourouis. Using aop in discrete event simulation : A case study with japrosim. In *Proceedings of the 2013 International Conference on Systems, Control and Informatics (SCI 2013)*, pages 164–170, Venice, Italy, 2013.

[25] M. Chibani, B. Belattar, and A. Bourouis. Aspect oriented simulation : A case study with the japrosim framework. In *Proceedings of the 2013 EUROPEAN SIMULATION AND MODELLING CONFERENCE ESM2013*, pages 91–100, Lancaster University, Lancaster, United Kingdom, 2013.

[26] M. Chibani, B. Belattar, and A. Bourouis. Towards a new aspect-oriented modelling approach. In *Proceedings of The 2 nd Conference on Theoretical and Applicative Aspects of Computer Science (CTAACS13)*, 2013.

[27] M. Chibani, B. Belattar, and A. Bourouis. Practical benefits of aspect-oriented programming paradigm in discrete event simulation. *Modelling and Simulation in Engineering*, ISSN 1687-5605 (Online), 1687-5591 (Print), . vol. 2014, Article ID 736359 : 1–16 pages, 2014.

[28] M. Chibani, B. Belattar, and A. Bourouis. Aspect and subject oriented programming paradigms : A comparative study. In *The 3th international Conference on complex System (CISC'2014), December 09 10, 2014, Jijel, Algeria*, 2014.

[29] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *EEE Transactions on Software Engineering*, 20(6) :476–493, 1994.

[30] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design approaches ? In *Report of the EU Network of Excellence on AOSD*, pages 1–82, Venice, Italy, May 2005.

[31] K. R. Christiansen and N. Oost. A look at programming methods for solving problems of current software development. In *StudColl*, pages 8–15, University of Groningen, 2005.

[32] A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with ajdt. In *Workshop on Analysis of Aspect-Oriented Software held in conjunction with the European Conference on Object-Oriented Programming (ECOOP)*, Darmstadt, Germany, 2003.

[33] K. Cooper, L. Dai, and Y. Deng. Modeling performance as an aspect : a uml based approach. In *Proceedings of the 4th AOSD Modeling with UML Workshop*, 2003.

[34] C. A. S. D. Cunha. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. Master's thesis, University of Minho, Braga, 2006.

[35] S. Efftinge, P. Friese, and A. H. et al. Xpand documentation, 2004.

[36] M. O. Elish, M. Al-Khiaty, and M. Alshayeb. Investigation of aspect-oriented metrics for stability assessment : A case study. *Journal of Software*, 6 (12) :2508–2514, 2011.

[37] J. Evermann. A meta-level specification and profile for aspectj in uml. *International Journal of Object Technology*, 6 (7) :27–49, 2007.

[38] J. Evermann, A. Fiech, and F. E. Alam. A platform-independent uml profile for aspect-oriented development. In *Proceedings of the Int. Conf. The Fourth International Conference on Computer Science and Software Engineering (C3S2E'11)*, pages 25–34, Montreal, Canada, 2011.

[39] N. Fenton and S. L. Pfleeger. Software metrics - a rigorous and practical approach. *International Thomson Computer Press*, 1996.

[40] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced separation of Concerns, OOPSLA*, 2000.

[41] T. Gil. *Conception Orientée Aspects 2.0.* DotNetGuru, 49, rue de Turenne 75003 PARIS, 2004.

[42] T. Gottardi, R. Aparecida, and V. Camargo. A process for aspect-oriented platform-specific profile checking. In *Proceedings of the Int. Workshop on Early aspects (EA'11)*, pages 1–5, Porto de Galinhas, Brazil, 2011.

[43] V. Gowri. Extending the uml meta-model to grant prop up for crosscutting concern. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 1 (7) :193–198, 2012.

[44] S. Gérard. On the papyrus use : Usage, specialization and extension, september 2010. Laboratory of model driven engineering for embedded systems (LISE).

[45] R. C. Gronback. *Eclipse Modelling Project A Domain-Specific Language Toolkit.* First Edition, (Addison-Wesley Professional, 2009.

[46] L. Gulyás and T. Kozsik. The use of aspect-oriented programming in scientific simulations. In *Proceedings of Software Technology, Fenno-Ugric Symposium FUSST'99 (Jaan Penjam ed.), Technical Report CS 104/99*, pages 17–28, Tallinn, Estonia, August 1999.

[47] J.-Y. Guyomarch. *Une architecture pour lt'evaluation qualitative de limpact de la programmation orientt'ee aspect.* PhD thesis, Département dinformatique et de recherche opérationnelle Faculté des arts et des sciences,Universitt'e de Montréal, 2006.

[48] S. Hanenberg. *Design Dimensions of Aspect-Oriented Systems.* PhD thesis, University Duisburg-Essen, 2005.

[49] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, pages 161–173, Seattle, Washington, USA, 2002.

[50] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming : Systems, Languages, and Applications*, pages 411–428, Washington, D.C., 1993.

[51] W. Harrison and P. Tarr. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In *ICSE '99 Proceedings of the 21st international conference on Software engineering*, pages 687–688, 1999.

[52] L. Hendren, O. D. Moor, A. Simon, Christensen, and the abc team. The abc scanner and parser, including an lalr(1) grammar for aspectj, 2004.

[53] K. Hoad, S. Robinson, and R. Davies. Automating warm-up length estimation. In *Proceedings of the 2008 Winter Simulation Conference, S. J. Mason, R. R. Hill, L. (Monch, O. Rose, T. Jefferson, J. W. Fowler eds)*, pages 532–540, 2008.

[54] A. Hovsepyan, R. Scandariato, S. V. Baelen, Y. Berbers, and W. Joosen. From aspect-oriented models to aspect-oriented code ? the maintenance perspective. In *AOSD' 10 Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, 2010.

[55] T. Ionescu, W. S. A Piater, and E. Laurien. An aspect-oriented approach for the development of complex simulation software. *Journal of Object Technology*, 9(1) :161–181, 2010.

[56] J. U. Júnior, V. V. Camargo, and C. V. Flach. Uml-aof, a profile for modeling aspect-oriented frameworks. In *Proceedings of the 13 th Int. Workshop on Aspect-Oriented Modeling (AOM'09)*, pages 1–6, Charlottesville, Virginia, USA, 2009.

[57] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.

[58] J. Klein. *Aspects Comportementaux et Tissage.* PhD thesis, University devant lUniversité de Rennes 1, 4 décembre 2006.

[59] I. Krechetov, B. Tekinerdogan, A. Garcia, C. Chavez, and U. Kulesza. Towards an integrated aspect-oriented modeling approach for software architecture design. software architecture design. In *8th Workshop on Aspect-Oriented Modelling (AOM.06)*, 2006.

[60] R. Laddad. *AspectJ in Action Practical Aspect-Oriented Programming.* First Edition, Manning Publications, 2003.

[61] R. Laddad. *Aspectj in Action : Enterprise AOP with Spring Applications.* Second Edition, Manning Publications, Greenwich, USA, 2009.

[62] C. Letavernier. Papyrus user guide. Series CEA, LIST, Laboratory of model driven engineering for embedded systems, 2012.

[63] K. Lieberherr and D. Lorenz. Coupling aspect-oriented and adaptive programming. In *In Robert E. Filman and Tzilla Elrad and Siobhn Clarke and Mehmet Akşit, editors, Aspect-Oriented Software Development*, pages 145–164, Addison-Wesley, Boston, 2005.

[64] J. D. Liljat. Measuring computer performance : A practitioner's guide. Cambridge University Press, Cambridge, 2000.

[65] D. Lohmann and O. Spinczyk. Aspect-oriented programming with c++ and aspectc++. AOSD 2007 Tutorial, University of Erlangen-Nuremberg Computer Science 4, 2007.

[66] G. Lours, S. Lecacheur, Y. Petit, and R. Verdier. Programmation orientée aspects. Copyright NormandyJUG-License Creative Commons 2.0 France-Patérnité-Partage des conditions initiales à l'identique, 2009.

[67] R. Martin. Oo design quality metrics, an analysis of dependencies. In *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*, 1994.

[68] H. Mcheick, H. Mili, A. El-Kharraz, and S. Sadou. comparaison of aspect-oriented software development techniques for distributed applications. In *International Conference Applied Computing*, pages 324–333, San Sebastian, Spains, 2006.

[69] T. Mehmood, N. Ashraf, K. Rasheed, and S. T. ur Rehman. Framework for modeling performance in multi agent systems (mas) using aspect oriented programming (aop). In *Workshop on The Sixth Australasian Software and System Architectures (AWSA 2005)*, 2005.

[70] OMG. Unified modeling language specification, version 1. Object Management Group (OMG), 2000.

[71] A. Oprisan. Aspect oriented implementation of design patterns using metadata. Master's thesis, Department of Computer Science and Statistics, University of Joensuu., 2008.

[72] H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds. Subject-oriented programming : Supporting decentralized development of objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, pages 1–13, Santa Clara, CA, 1994.

[73] H. Ossher, M. Kaplan, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA'95*, volume 2(3), pages 235–250, Austin, TX, USA. TAPOS, 1995.

[74] K. Ostermann and G. Kniesel. Independent extensibility an open challenge for aspectj and hyper/j. In *Intl Work. on Aspects and Dimensional Computing at ECOOP00*, 2000.

[75] D. Park and S. Kang. Design phase analysis of software performance using aspect-oriented programming. In *the 5th International Workshop on Aspect-Oriented Modeling*, Lisbon, Portugal, 2004.

[76] D. Park, S. Kang, and J. Lee. Design phase analysis of software qualities using aspect-oriented programming. In *the 7th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 06),Y.-T.Song,C.Lu, and R.Lee,Eds.*, page 2934. IEEE Computer Society, Las Vegas, Nev, USA, 2006.

[77] A. Piater, T. B. Ionescu, and W. Scheuermann. A distributed simulation framework for mission critical systems in nuclear engineering and radiological protection. *International Journal of Computers, Communications & Control*, 3 :448453, 2008.

[78] A. Przybyek. An empirical assessment of the impact of aop on software modularity. In *Proceedings of the 5th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2010)*, Athens, Greece, 2010.

[79] J. Ribault and O. Dalle. Enabling advanced simulation scenarios with new software engineering techniques. In *Proceedings of the the 20th European Modeling and Simulation Symposium (EMSS08)*, page 515520, Briatico,Italy, 2008.

[80] J. Ribault, O. Dalle, D. Conan, and S. Lerichel. Osif : A framework to instrument, validate, and analyze simulations. In *Proceedings of 3rd International Conference on Simulation Tools and Techniques SIMUTools'10*, pages 56–60, Malaga, Spain, 2010.

[81] A. Sandven. Metamodel based code generation in dpf editor, 2012. Master's Thesis in Informatics-Program Development, Department of Computer Engineering Bergen University College.

[82] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel. A survey on aspect-oriented modeling approaches. Publication database administration program of the Vienna University of Technology, 2007.

[83] H. Shen and D. C. Petriu. Performance analysis of uml models using aspect-oriented modeling techniques. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems MoDELS' 05*, pages 156–170, Berlin, 2005. Springer-Verlag.

[84] M. Skipper. The watson subject compiler and aspectj (a critique of practical objects, in workshop on multi-dimensional separation of concerns. In *OOPSLA Workshops*, 1999.

[85] M. Skipper and S. Drossopoulou. Formalising composition-oriented programming. In *ECOOP Workshop*, volume 1743, pages 307–308, 1999.

[86] M. Stochmialek. Aopmetrics project, 2005. URL `http://aopmetrics.tigris.org`.

[87] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation : Multidimensional separation of concerns. In *Proceedings of ICSE'99*, pages 107–119, 1999.

[88] F. Tessier. quality assurance and development of aspect oriented software development : a model based detection of conflicts between aspects. Quebec University, 2005.

[89] S. Tsang. An evaluation of aop for java-based real-time systems development. Master's thesis, University of Dublin, 2004.

[90] Z. Vaira and A. Caplinskas. Application of pure aspect-oriented design patterns in the development of ao frameworks : a case study. *Informacijos Mokslai/Information Sciences*, 56 :146–155, 2011.

[91] M. Voelter. A catalog of patterns for program generation, 2003.

[92] V. Vranié. Towards multi-paradigm software development. *Journal of Computing and Information Technology - CIT*, 10(2) :133–147, 2002.

[93] W. Weiland, R. Weatherly, K. Ring, E. Page, R. Mikula, and F. Kuhl. Simplified concurrency : A java simulation framework. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications Conference*, 2005.

[94] A. A. Zakaria, H. Hosny, and A. Zeid. A uml extension for modeling aspect-oriented systems. In *2nd Workshop on Aspect-Oriented Modeling with UML*, Dresden, Germany, September 2002.