la conception orientée Aspect

par Thomas Gil

Parrainé par Valtech Training

valtech training

DNG

Editions DotNetGuru

DotNetGuru
49, rue de Turenne
75003 PARIS

Site Web
http://www.dotnetguru.org

Edité le 1er octobre 2004, réédité le 1er mars 2005, puis le 21 janvier 2006

d'exemple et d'illustration.

Deux mots sur l'auteur

Thomas GIL est consultant-formateur indépendant, après avoir occupé ce poste dans la société *Valtech Training*. Il enseigne la conception Objet et la programmation avec C++, Java, C#, PHP et anime également des formations sur les plate-formes de développement J2EE et .NET et sur les outils issus du monde XML.

Thomas est aussi co-auteur du site *www.dotnetguru.org*, et gérant de la société DotNetGuru SARL associée au site. Cela l'amène donc à rédiger des articles techniques portant majoritairement sur la conception Objet et les architectures multi-couches, et à mener des projets de développement tels que le PetShopDNG et le tisseur AspectDNG.

Conception Orientée Aspects 2.0

Table des matières

Table des figures

Exemples de code

Remerciements

Ce livre n'aurait probablement jamais vu le jour sans le concours ou le soutien de certaines personnes de mon entourage, que je tiens à remercier vivement ici.

Tout d'abord, un grand merci à **Corinne Martinez**, directrice de *Valtech Training*. Grâce à elle et à sa confiance, Valtech Training a pris l'initiative de sponsoriser la rédaction de ce livre. Par cet investissement, la société s'engage fortement dans le domaine de la Conception et de la Programmation Orientées Aspects. Une autre preuve de cet engagement est l'apparition d'une formation sur l'AOP dans le catalogue Valtech Training, inspirée de ce livre.

Merci à **Patrick Smacchia**, auteur de Pratique de C#, qui a eu la gentillesse de me faire bénéficier de son expérience d'auteur d'ouvrages techniques et de réfléchir avec moi aux pistes à suivre dans la rédaction de ce livre, ainsi qu'à **Jean-Baptiste Evain**, concepteur et développeur Objet passionné, pour son intérêt dès le démarrage du projet AspectDNG, ses remarques pertinentes et son enthousiasme. Plus qu'une aide, Jean-Baptiste est devenu un développeur à part entière du projet AspectDNG, et à l'heure où sont écrites ces lignes, il est en passe d'en devenir le chef et de prendre ma place pour les versions à venir de cet outil.

Et bien entendu, il me faut rendre les honneurs à **Sami Jaber**, lui aussi consultant pour Valtech et auteur principal du site www.dotnetguru.org. Depuis deux ans et demi, grâce à lui, j'ai pu monter en compétence sur la plate-forme .NET et j'ai eu l'opportunité de rédiger des articles techniques sur cette plate-forme ainsi que sur J2EE et sur la conception Objet. En acceptant de faire de www.dotnetguru.org le vecteur de vente de cet ouvrage, Sami a lui aussi pris un risque important puisque cela associait l'image du site à cette publication.

Préface

Quel avenir pour la Programmation Orientée Aspect (AOP) ? L'adoption, si adoption il y a, sera-t-elle lente, progressive ou fulgurante ? Il est certain que de prime abord, cette nouvelle facette du développement est séduisante : elle présente une innovation technique intéressante et répond à une problématique récurrente. Mais est-ce pour autant l'assurance d'un bel avenir ? Nous gardons malheureusement tous en mémoire, le souvenir de bonnes idées, d'excellents produits retournés prématurément au stade du carton poussiéreux, détrônés par d'autres produits techniquement contestables, au prix d'une bataille commerciale et marketing déséquilibrée.

Alors, devant tant d'incertitudes, pourquoi parrainer des travaux sur l'AOP ? Tout simplement que c'est en explorant de nouvelles voies qu'on avance. Que ce soit sur OMT puis UML, Corba, Java ou XML en leur temps, le groupe Valtech s'est toujours voulu précurseur sur les technologies modernes de développement. Cette orientation stratégique implique nécessairement un certain niveau de prise de risque et donc d'investissement en recherche et développement. Valtech Training, filiale de Valtech, perpétue la tradition.

Au printemps 2004, à l'occasion d'une étude de marché, nous avons évalué l'intérêt porté par les équipes techniques de développement (consultants, développeurs, programmeurs...) à ce sujet. Nous avons été agréablement surpris de constater que la moitié des "développeurs" interrogés connaissaient ce sujet pourtant encore confidentiel. Mieux encore, le tiers de ces personnes voyait en AOP une technologie d'avenir alors que moins de 10% d'entre elles pensaient qu'elle n'émergerait pas.

Il y avait là de bonnes bases pour lancer des actions de R&D sur la programmation orientée Aspect : une solution technique intéressante et prometteuse, de futurs utilisateurs potentiels réceptifs voire déjà conquis, des forces certaines dans nos équipes techniques et une légitimité reconnue du groupe Valtech à s'attaquer à un sujet dans la droite ligne de ses compétences.

Et puis... et puis il y a l'homme : Thomas. Quand "Tom" vient vous voir

avec cette flamme dans les yeux qui dit "j'ai travaillé quelques dizaines de nuits sur un truc qui en vaut la peine, j'aimerais vous en parler..." Humainement, on n'a pas le droit de faire la sourde oreille. Parce que n'oublions pas que c'est par ces intuitions, ces coups de génie, que les tendances émergent. Il faut donner leur chance aux idées, les encourager et les soutenir... C'est la voie que nous suivons et, sur AOP aussi, nous serons prêts.

Jocelyn Thielois

Responsable Marketing Opérationnel de *Valtech Training*

Chapitre 1.

Introduction

Depuis quelques années, le marché de l'informatique se forme massivement aux technologies Orientées Objet. Entendez par là les langages Objet tels que C++, Eiffel, Python, Java, C#, mais aussi les techniques d'analyse et de conception Objet. Le fossé conceptuel a parfois été difficile à franchir pour certains: cela vient à la fois du besoin d'apprendre de nouveaux langages, mais surtout du fait que l'approche Objet offre (ou impose, selon votre perception) une nouvelle manière de penser les problèmes et leurs solutions.

Depuis longtemps déjà, certains précurseurs jonglent avec ces notions et ont poussé plus avant la réflexion sur l'Objet et en particulier sur la conception Orientée Objet. Ces champs d'investigation ont permis de formaliser de bonnes pratiques de conception, les fameux **Design Patterns**, que nous rappellerons brièvement au début de ce livre. Une excellente initiative a également été de formaliser un certain nombre de mauvaises pratiques de conception, connues sous le terme peu flatteur d'**Anti-Pattern**. Autant d'idées que nous aurions pu avoir pour résoudre nos problèmes de conception et que l'on nous déconseille, argumentation à l'appui, pour éviter de rendre nos logiciels trop couplés, difficiles à maintenir ou à faire évoluer.

Avec le décalage nécessaire à l'arrivée à maturité, à la démocratisation et à l'adoption de ces techniques, nous nous trouvons aujourd'hui dans une situation contrastée dans laquelle certains sont encore en train de "passer à l'objet" alors que d'autres connaissent et maîtrisent déjà les Design Patterns. C'est tout à fait normal et cet état de choses risque de durer encore quelques années. Certains domaines tels que les mondes scientifique et industriel, en particulier l'embarqué et le temps réel "dur" restent souvent récalcitrants et ne souhaitent pas nécessairement franchir le pas, du moins sur les projets les plus critiques.

Mais ceux qui ont fait le choix de l'objet et qui se sont approprié cette

manière de concevoir et de programmer ne se sont pas arrêtés en si bon chemin. Récemment, nous avons pu remarquer un certain renouveau de la recherche informatique: outre la conception Objet et les Design Patterns, nous parlons de plus en plus d'architecture technique (distribuée, multi-couches, orientée service...), d'architecture pilotée par les modèles (MDA), mais aussi de thèmes moins plébiscités tels que les Mixin Layers, la programmation orientée Policy, l'instrumentation de code, la séparation des responsabilités et des objectifs, l'inversion des dépendances (IOC) et enfin la programmation orientée Aspects (AOP).

Dans cet ouvrage, nous allons faire le point sur certaines bonnes pratiques de conception; ces techniques sont très puissantes, mais leur pouvoir d'expression est parfois limité par... les principes de base de l'Objet. En effet, et nous essaierons de vous en donner la preuve, les langages de programmation orientés objet actuels sont insuffisants pour répondre à toutes les exigences de qualité du développement logiciel.

En poussant cette réflexion, nous introduirons logiquement la notion d'**Aspect** non pas comme un concurrent, mais comme un outil complémentaire à la conception Orientée Objet. Dès lors, nous pourrons nous pencher sur l'outillage nécessaire et réfléchir à l'implémentation de ce que l'on appelle un Tisseur d'Aspects. Dans ce livre, nous citerons et nous utiliserons plusieurs tisseurs disponibles sur les plate-formes Java et .NET; mais quand il s'agira de dévoiler les détails d'implémentation interne d'un tisseur d'Aspects, nous nous baserons sur AspectDNG.

Après avoir passé en revue les Design Patterns et leurs limites, la notion d'Aspect et le fonctionnement d'un tisseur, nous parcourrons l'ensemble des applications intéressantes de la conception Orientée Aspects. Nous avons donc choisi un ensemble de situations dans lesquelles les Aspects jouent leur rôle de catalyseur de conception à la perfection: les traces (un grand classique des Aspects), la persistance automatique, le contrôle (et peut-être l'optimisation) des performances, la gestion du cycle de vie des objets et des ressources techniques, les tests unitaires, la sécurité...

Chaque lecteur étant différent, nous ne saurions trop vous recommander de lire ce livre comme bon vous semble: dans un ordre séquentiel, aléatoire, sélectif... Ceux qui maîtrisent bien les Design Patterns survoleront certainement le premier chapitre, d'autres tisseront

leurs aspects avec d'autres outils et seront peut-être moins intéressés par l'implémentation et les fonctionnalités d'AspecDNG...

L'essentiel est de vous faire plaisir, de découvrir ce qui est nouveau pour vous, en gardant toujours un œil critique sur les techniques présentées. Certaines vous plairont, d'autres vous sembleront inappropriées ou dangereuses. Et comme le domaine est encore jeune, nous comptons sur vous pour inventer de nombreux **Aspect Design Patterns** et pour participer activement à cette transformation passionnante que subit aujourd'hui le domaine de la conception logicielle.

Bonne lecture!

Chapitre 2.

Rappels de conception Orientée Objet

Dans cette section, nous allons revenir sur les notions fondamentales qui vous permettront à la fois de mieux appréhender les Design Patterns présentés dans la section suivante, mais aussi de comprendre que lorsque ces Patterns sont appliqués, ils introduisent parfois une complexité ou une dépendance technique qu'il serait bon d'éliminer (grâce à l'approche Orientée Aspects, bien sûr).

Les grands principes qui suivent peuvent être vus comme autant d'objectifs à atteindre au cours d'une phase de conception Objet mais aussi comme autant d'outils d'évaluation et de critique d'une conception déjà bien entamée. Nous vous invitons donc à essayer de prendre un exemple tiré de votre expérience et d'essayer d'y appliquer chacun de ces préceptes, soit pour aboutir à une solution élégante, soit pour critiquer les choix qui ont déjà été faits.

2.1. Faible couplage

L'un des objectifs majeurs de la phase de conception dans un projet est de faire en sorte que les évolutions futures du logiciel puissent être intégrées à la solution actuelle sans la remettre en cause de fond en comble. Une manière d'éviter un impact trop important de modifications à venir est de diminuer le couplage existant entre les composants constituant notre logiciel.

Le Faible Couplage est un principe d'évaluation, il permet de critiquer une conception Objet. Le mieux, pour l'introduire, est donc de donner un mauvais exemple de conception, et de vous expliquer en quoi il ne respecte pas le Faible Couplage.

Prenons donc l'exemple d'un système de gestion de compte en banque dans lequel il s'agit pour un utilisateur, client de la banque, de consulter son compte à travers une application graphique. Imaginons que le compte en banque soit représenté par la classe **CompteEnBanque**; sa métaphore graphique (disons une boîte de dialogue) pourrait être implémentée par une relation d'héritage entre la classe CompteEnBanque et une classe issue d'une bibliothèque de composants graphiques: **BoîteDeDialogue**.

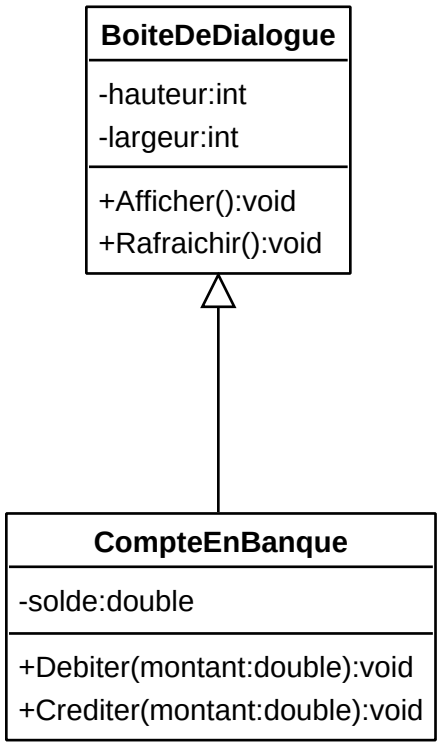


Figure 1. [Mauvaise pratique] CompteEnBanque est une BoîteDeDialogue

La relation d'héritage permet au CompteEnBanque de s'afficher à l'écran de l'utilisateur, nous avons donc atteint l'objectif principal et rendu le CompteEnBanque graphique. Pourtant, cette solution n'est pas acceptable en termes de conception: si l'on songe aux évolutions possibles du logiciel et que l'on imagine que, demain, il s'agira d'afficher les informations du compte en banque non plus dans une boîte de dialogue mais à travers une interface Web, vocale, braille, etc... il faudrait ajouter autant de relations d'héritage que de média de présentation. Or la relation d'héritage est généralement contrainte: une classe fille ne peut avoir qu'une seule classe mère (excepté dans les langages C++ et Eiffel).

On dit dans ce cas qu'il existe **un couplage trop important** entre la classe CompteEnBanque et la BoîteDeDialogue de la bibliothèque graphique. Nous verrons dans les sections suivantes comment limiter le couplage entre ces composants.

2.2. Forte cohésion

Les langages Orientés Objet nous 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 que nous rassemblons bien dans une classe 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.

Comme le Faible Couplage, la Forte Cohésion est un principe d'évaluation. On ressent naturellement le besoin de l'appliquer lorsqu'on croise une conception Objet qui ne le respecte pas.

Prenons donc à nouveau un mauvais exemple: dans la classe CompteEnBanque, il serait mal venu d'implémenter une méthode "Afficher()" ou "Tracer()" puisque l'objectif de cette classe est de représenter le modèle d'un compte en banque et non celui d'une fenêtre graphique ou d'un service de journalisation.

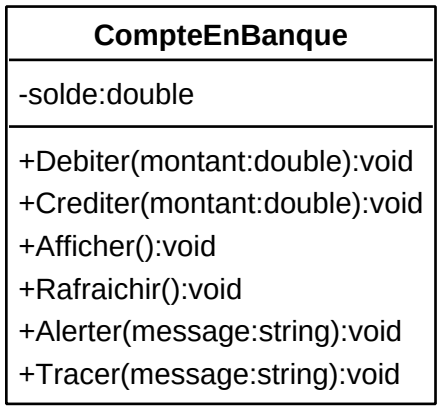


Figure 2. [Mauvaise pratique] CompteEnBanque affichable et journalisable

Au contraire, on conseille généralement de rassembler dans une autre classe (et ici, puisque ce sont des niveaux conceptuels différents, dans des espaces de nommage différents) toutes les méthodes de journalisation (Tracer(), Avertir()), et dans une troisième classe ce qui a trait à l'affichage d'une boîte de dialogue dans une interface graphique.

2.3. Inversion des dépendances

Ce troisième principe se base sur le faible couplage pour savoir quand il doit s'appliquer. Plus précisément, voici comment on peut évaluer le besoin d'inverser les dépendances: **une classe ne doit jamais dépendre d'une autre classe qui serait d'un niveau conceptuel plus bas**. Par exemple, une classe métier comme le CompteEnBanque ne doit jamais dépendre d'une classe technique comme la Journalisation ou graphique comme la BoîteDeDialogue.

Pour limiter un tel couplage, l'inversion des dépendances propose:

- D'inverser la relation à l'origine du couplage lorsque c'est possible: une classe graphique pourrait utiliser ou hériter du CompteEnBanque et enrichir son comportement par les fonctionnalités d'affichage requises.
- Lorsque c'est impossible, l'inversion des dépendances propose d'introduire un niveau d'abstraction plus élevé entre le CompteEnBanque et la BoîteDeDialogue et la Journalisation. Typiquement, il faudrait spécifier des interfaces qui exposent les services de ces deux dernières classes, et dont dépendrait le CompteEnBanque. Cela limiterait le couplage, sans l'éliminer totalement.

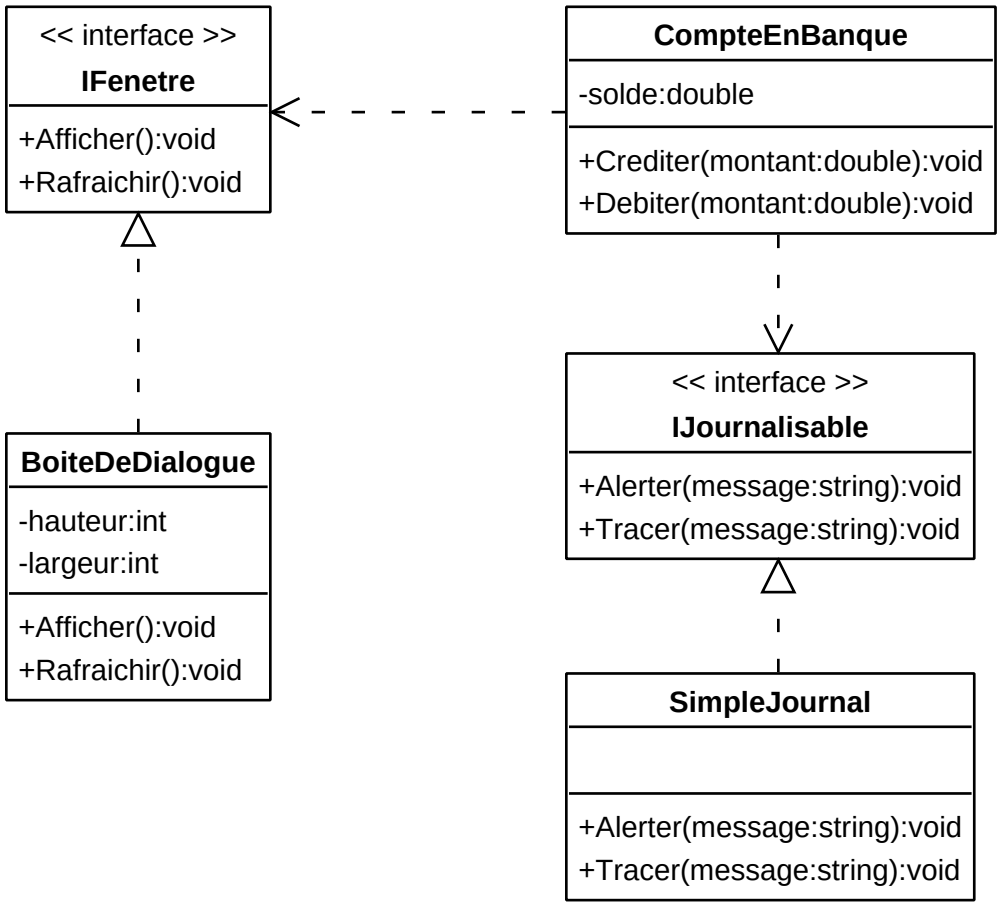


Figure 3. [Bonne pratique] CompteEnBanque utilisant IFenetre et IJournalisation

Cette manière de concevoir les applications et de limiter le couplage a le vent en poupe en ce moment. Nombreux sont les socles (ou frameworks) techniques d'**IOC** (Inversion Of Control) qui permettent aux objets du domaine de ne dépendre que d'interfaces techniques. Les objets techniques concrets qui implémentent ces interfaces sont instanciés par le framework à l'initialisation ou à la demande grâce aux renseignements trouvés dans un fichier de configuration centralisé. Nous reviendrons sur cette infrastructure d'IOC dans les chapitres suivants.

2.4. 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, disons par exemple entre un CompteChèque et un CompteEnBanque, il faudrait toujours que nous puissions:

1. Prononcer la phrase *"X est un Y"*, où X représente la classe fille et Y la mère.
2. Remplacer le CompteEnBanque par le CompteChèque dans toutes les situations sans que cela ne mène à une incohérence.

Dans notre exemple, *"un CompteChèque est un CompteEnBanque"*, la première assertion est donc bien vérifiée.

De même, nous pouvons questionner la relation d'héritage par la seconde assertion: peut-on débiter, créditer, demander le solde d'un CompteChèque? Est-ce que ces services ont bien la même sémantique que pour le CompteEnBanque? Oui, tout est cohérent, on peut dire que le CompteChèque respecte parfaitement le contrat du CompteEnBanque, et donc que le principe de substitution de Liskov est bien respecté.

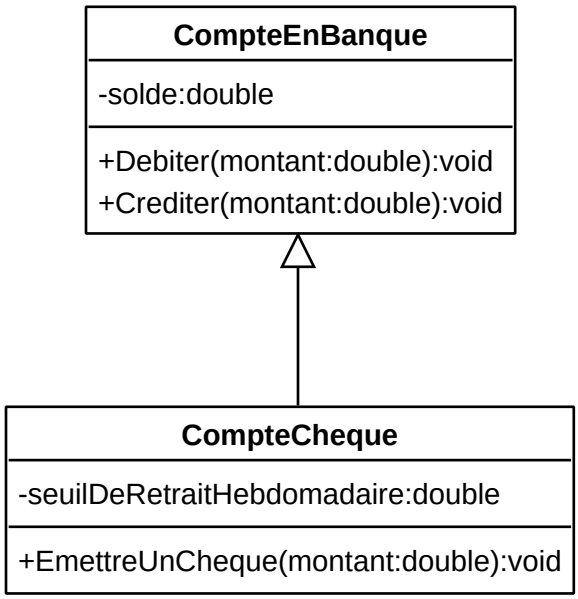


Figure 4. [Bonne pratique] CompteChèque hérite de CompteEnBanque

Par contre, si nous reprenons le mauvais exemple dans lequel le CompteEnBanque héritait de BoîteDeDialogue:

1. *"Un CompteEnBanque est une BoîteDeDialogue"*: cette phrase est fausse, mais on peut éventuellement la tolérer si l'on comprend "la représentation graphique du CompteEnBanque est une BoîteDeDialogue".

2. *"Chaque situation mettant en œuvre une BoîteDeDialogue pourrait se contenter d'un CompteEnBanque"*: cette fois, l'assertion est complètement erronée. Par exemple, cela n'a pas de sens de rendre un CompteEnBanque modal, de l'afficher, le rendre invisible. Pire encore: le service de "Fermer" un CompteEnBanque existe au même titre que "Fermer" une BoîteDeDialogue, mais la signification est toute autre. On dit que la sémantique du contrat est rompue.

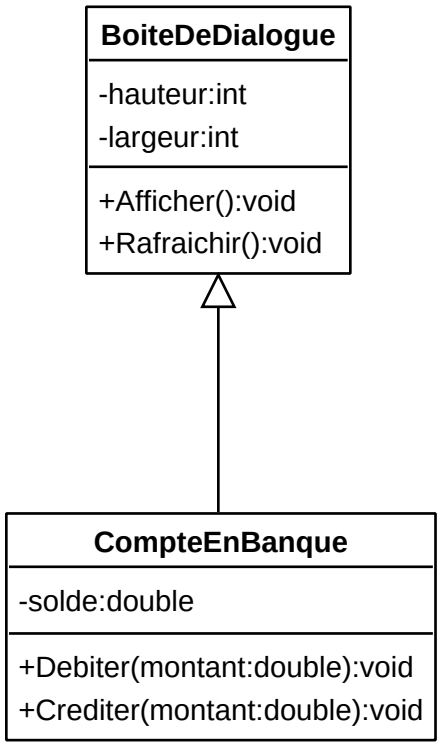


Figure 5. [Mauvaise pratique] CompteEnBanque est une BoîteDeDialogue

2.5. Ouvert fermé

Ce dernier principe de base de la conception Objet est peut-être l'un des plus délicats à mettre en œuvre. Il consiste à dire qu'**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**. La raison est simple: le code du logiciel actuel est identifié, testé, validé (et pour les projets les plus "carrés", référencé dans un mécanisme de gestion de la traçabilité des besoins utilisateur). Modifier ce code doit donc se faire en prenant de nombreuses précautions; en particulier, il faut mesurer l'impact de la modification avant de l'effectuer (ce qui n'est pas toujours simple). Alors qu'ajouter 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 se limitant aux techniques Orientées Objet, puisqu'on ne souhaite pas modifier le code existant mais que son comportement doit tout de même évoluer, il faut prévoir lors de la conception et de l'implémentation initiales des **points d'extensibilité**, c'est-à-dire des endroits où l'on pense qu'une évolution aura probablement lieu. A la manière d'un framework, les situations nouvelles pourront être implémentées par de nouvelles classes héritant de classes déjà existantes, et la sollicitation de ces nouvelles classes se fera dynamiquement par **polymorphisme**.

Grâce à l'approche Orientée Aspects, nous verrons que certaines évolutions pourront être intégrées non seulement sans avoir à changer le code existant, mais aussi sans avoir à prévoir tous les points d'extensibilité du logiciel à l'avance.

Chapitre 3.

Design Patterns

L'objectif de ce chapitre est de faire le point sur la notion de Design Pattern qui est un prérequis pour bien appréhender les chapitres suivants. L'approche choisie est de présenter de manière synthétique et pragmatique une petite dizaine de Design Patterns à travers un exemple simple qui servira de fil conducteur.

Nous n'avons toutefois pas la prétention d'éclipser les ouvrages entièrement dédiés aux Patterns et dont vous trouverez une référence dans la bibliographie. Leur lecture est passionnante et indispensable pour maîtriser les motivations et les implications des différents Patterns. Non, considérez plutôt ce chapitre comme un rappel ou une introduction, mais indispensable pour mesurer l'apport des Aspects face aux outils déjà existants dans le monde de l'Objet.

3.1. Définition et objectifs

Pour résumer l'essentiel, donnons sans détour une première définition: **un Design Pattern est une technique efficace et éprouvée de conception Orientée Objet**.

L'idée est la suivante: au cours de la réalisation de projets informatiques, il n'est pas rare que certaines situations soient récurrentes. Puisque les outils mis à notre disposition par les langages Orientés Objets sont limités (classes, interfaces, puis association, héritage), il est plus que probable que ces situations amènent à des solutions de conception similaires. Eh bien un Design Pattern correspond à l'une de ces solutions qui a été jugée appropriée pour de nombreux projets, par de nombreux concepteurs, dans une situation comparable.

En Français, Design Pattern est souvent traduit par:

- Patron de conception
- Bonne pratique de conception
- Modèle de conception

Le premier travail des concepteurs Objet est donc de repérer ces bonnes pratiques à travers les projets sur lesquels ils interviennent. Une fois repéré, un Pattern doit être formalisé, nommé, documenté... et déconseillé dans les cas où il ne serait pas approprié. Ce type de travail de recensement et de normalisation a été ponctué par la publication du livre mythique **"Design Patterns: Elements of Reusable Object-Oriented Software"** en 1995; depuis, de nombreux ouvrages sur le sujet réexpliquent les mêmes Patterns en utilisant un langage de modélisation ou de programmation différent, ou décrivent de nouveaux Patterns applicables à des typologies de projets différentes.

L'objectif d'un Pattern est multiple:

- **L'identification, la nomenclature:** afin de reconnaître les Patterns dans une conception ou de discuter des Patterns applicables, il faut établir un langage commun et donc nommer tous les Patterns et les définir de manière univoque.

- **L'efficacité, l'élévation du niveau de langage:** avoir un mot connu pour évoquer un principe de conception éventuellement complexe (qui met en œuvre plusieurs classes et associations) permet de communiquer plus vite, d'éviter les ambiguïtés et de gagner en clarté. La contre-partie, évidemment, est que chaque interlocuteur doit avoir appris ce nouveau **langage de Patterns**.
- **L'enseignement:** un novice en conception Orientée Objet apprend énormément en lisant la description et la motivation des Design Patterns et en essayant de les mettre en pratique sur ses projets. Par contre, comme certains de ces concepts sont assez abstraits, il ne faut pas s'effrayer de ne pas en percevoir toutes les implications à la première lecture: il faut avoir alterné lecture et mise en application plusieurs fois pour maîtriser tous les enjeux des Patterns. Personnellement, j'ai dû lire "Design Patterns, Elements of Reusable OO Software" 5 ou 6 fois, en ayant participé à des projets Java puis .NET entre-temps; je vous avoue que la première lecture m'a complètement désorienté, mais avec le recul et un peu d'expérience, je me suis aperçu que les principes présentés n'étaient pas aussi abstraits que la première lecture pouvait le laisser croire.

Pour clore cette introduction à la notion de Pattern, disons encore que le fait de documenter de bonnes pratiques n'est pas une chasse gardée du domaine de la conception Objet. Les cuisiniers proposent leurs recettes, les architectes du bâtiment ont leurs Patterns, les couturières leurs patrons... Plus proches de notre activité, il existe également des Patterns d'analyse, des Patterns d'architecture, des Patterns de codage (que l'on appelle plutôt des idiomes) et même des Anti-Patterns: des exemples de ce qu'il ne faudrait jamais faire...

Par abus de langage, dans ce livre qui ne parle que de conception, nous parlerons de Patterns en sous-entendant qu'il s'agit de Design Patterns.

(Re-)découvrons maintenant quelques Patterns incontournables.

3.2. Voyage au pays des Patterns

Pour bien appréhender l'intérêt des Patterns, le mieux est de réfléchir à la conception d'une application et de voir petit à petit quels besoins ou problèmes apparaissent. Pour répondre à chaque besoin, nous essaierons de piocher un Pattern standard de notre bibliothèque: celle constituée des **23 Patterns du GOF** (GOF signifie Gang Of Four, soit la bande des quatre auteurs qui ont écrit le livre phare mentionné précédemment).

L'application "cas d'école" que nous avons choisie est **un système de gestion d'articles et de news**. Nous nous proposons d'en concevoir à la fois le processus de publication et celui de lecture et de navigation à travers les articles. L'objectif n'est absolument pas d'être exhaustif ou très précis concernant les aspects fonctionnels de ce système, mais bel et bien d'introduire progressivement les Design Patterns.

3.2.1. Cahier des charges

Les besoins fonctionnels de notre application sont extrêmement simples: il s'agit pour les auteurs d'articles ou de news d'être capables de rédiger leurs écrits (cette partie n'est pas à intégrer dans le système), puis de les publier. Si d'aventure plusieurs auteurs tentaient de publier le même jour, un mécanisme de gestion des priorités devrait choisir l'élément à publier et mettre les autres en attente d'une date de publication ultérieure.

D'autre part, tant pour les auteurs que pour leurs lecteurs, il doit être possible de faire des recherches simples parmi tous ces articles et de lire ceux qui les intéressent.

Afin d'éviter les mauvaises surprises, un mécanisme de tolérance aux pannes devra être mis en œuvre. Il peut s'agir d'une sauvegarde périodique, à condition que la déperdition de performances ne se fasse pas sentir lorsque cette sauvegarde s'exécute.

Enfin, au cas où d'autres outils seraient utilisés à l'avenir pour gérer les articles et news, le système doit disposer d'un mécanisme d'export dans

un format neutre (un document XML semble tout indiqué).

3.2.2. Conception initiale

Afin d'être brefs sur cette partie, nous allons faire une grosse entorse au processus habituel de recueil des besoins, d'identification des cas d'utilisation, de découverte des objets métier, d'analyse statique et dynamique... en espérant que les experts de ces étapes (tels que Pascal Roques, cf Bibliographie) ne nous en voudront pas trop.

Supposons donc que nous ayons modélisé les objets métier à la va-vite et que nous soyons arrivés au résultat suivant.

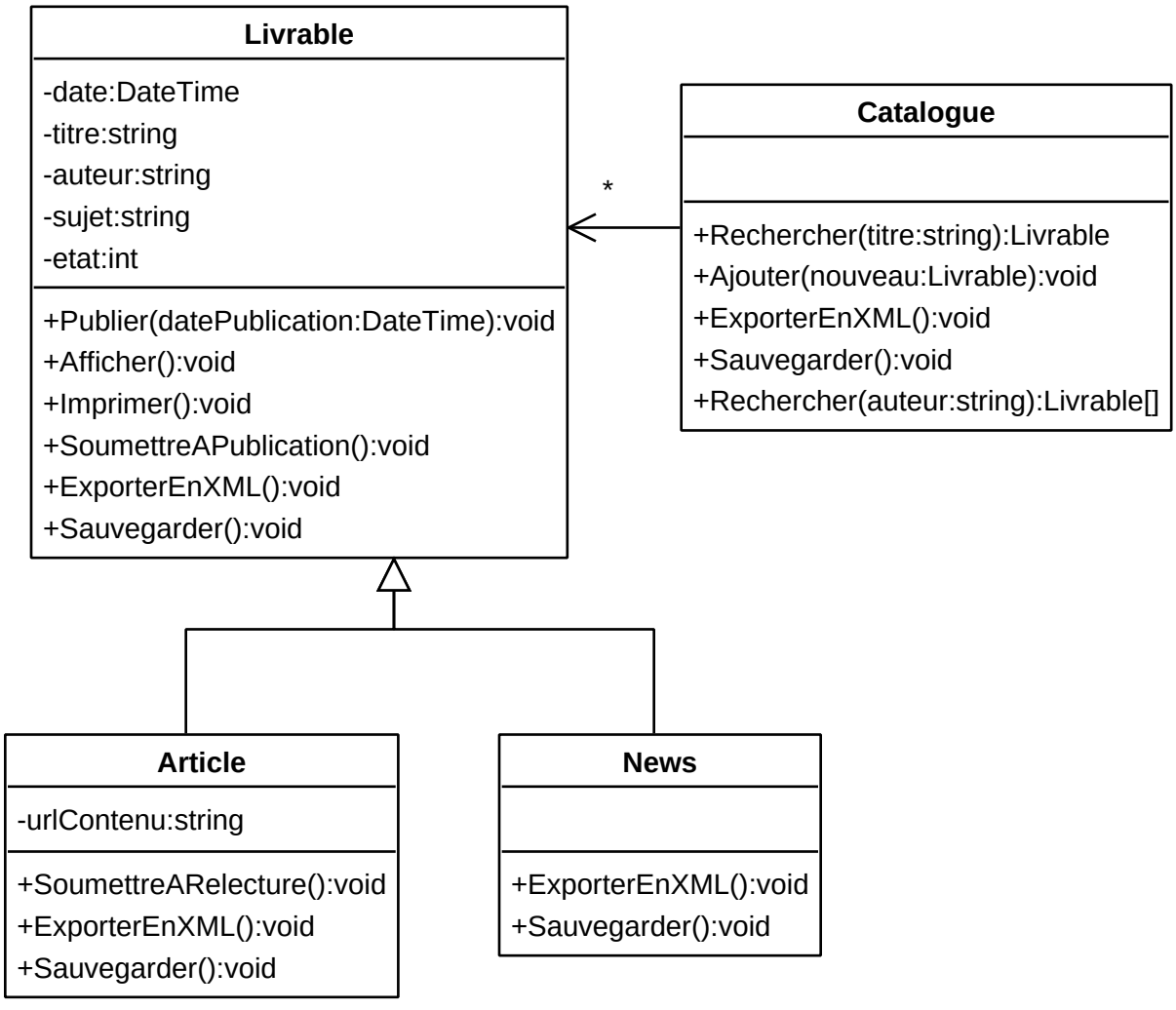


Figure 6. Conception initiale

Il est certain que ces quatre classes vont répondre aux besoins

exprimés dans le cahier des charges. Mais cette conception est-elle satisfaisante pour autant? Si l'on adopte le point de vue du concepteur Objet, la réponse est négative, et ce pour de nombreuses raisons. Prenons tout d'abord nos sacro-saints principes de conception:

- Les objets métier portent des responsabilités techniques. Par exemple, le Livrable implémente les services Afficher(), Imprimer(), Sauvegarder(), ExporterEnXML(), ce qui ne répond pas au critère de l'*Inversion des Dépendances*.

- Un corollaire du point précédent est que toutes les méthodes de la classe Livrable (il en va de même pour les autres classes, en réalité) ne visent pas le même objectif. Certaines sont là pour répondre à des besoins de sauvegarde, d'autres aux besoins fonctionnels de l'application... On sent entre elles une *faible cohésion*.

- Ces mêmes services techniques peuvent être amenés à évoluer. Le format d'export XML en particulier est susceptible de changer avec le temps et avec les besoins d'intégration aux applications connexes. Or si l'implémentation de ce service est du ressort du Livrable, nous ne pourrons pas faire évoluer le format XML sans toucher au code de cette classe. En conséquence, le critère de l'*Ouvert-Fermé* n'est pas respecté non plus.

- Pour reformuler le point précédent, il existe un *couplage fort* entre le format d'export XML et les classes Livrable, Article et News.

- En fait, le seul principe qui soit satisfait dans notre conception est celui de la *substitution de Liskov*, car les Articles et les News sont bien des livrables dans le domaine modélisé, et chaque cas d'utilisation d'un Livrable garde tout son sens si on l'applique à un Article ou à une News.

Ce n'est pas tout. D'autres critiques de conception peuvent être formulées du point de vue de la maintenabilité ou de l'évolutivité:

- La gestion de l'état du Livrable n'est pas claire; elle sera certes explicitée dans le code mais elle restera complètement opaque du point de vue de la conception.

- Il n'y a aucun découplage entre les processus métier, les interactions avec l'utilisateur et l'affichage ou l'impression des

Livrables. Le cycle de vie de ces différentes parties du logiciel est donc voué à être le même, alors que ce n'est pas forcément le cas dans le domaine métier modélisé; il est même probable que la charte graphique évolue, ou que le processus de sélection de la date de publication d'un article change, alors que les objets métier Livrables, Article et News, ont peu de chances de changer aussi vite.

- Il semble difficile de garder la trace de l'historique du déroulement des publications, et donc de faire des statistiques qui permettraient éventuellement d'améliorer leur processus.

Forts de toutes ces critiques, nous allons essayer d'améliorer la conception de ce petit système en appliquant progressivement les Design Patterns.

3.2.3. Différencier le contexte

Attardons-nous sur la publication d'un nouveau Livrable. Si aucune publication n'est planifiée, le processus est très simple: il suffit de rendre ce nouveau Livrable disponible dans la liste des publications, tous les lecteurs pourront en prendre connaissance immédiatement. Si par contre au moins une publication est déjà planifiée, il faut mettre le nouveau livrable en attente de la disponibilité d'une date.

Intuitivement, nous serions tentés de gérer le contexte dans l'implémentation de la méthode **Ajouter(nouveau:Livrable)** de la classe **Catalogue**. Mais tout changement fonctionnel aurait un impact sur cette implémentation. Il vaudrait donc mieux déléguer la responsabilité d'ajouter un livrable à un autre objet que le Catalogue, un objet qui pourrait être différent selon qu'une publication est ou non en attente.

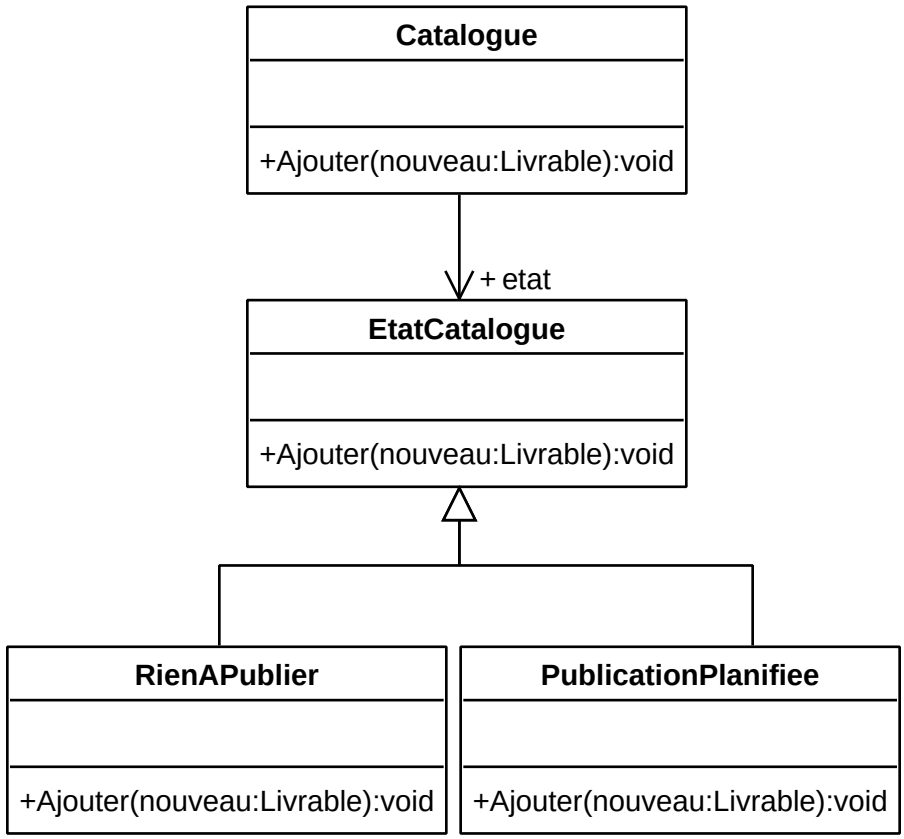


Figure 7. [Design Pattern] État

On appelle ce Pattern **État** car on délègue à la nouvelle classe **EtatCatalogue** la responsabilité de gérer le comportement du Catalogue dans un état donné. Chaque état spécifique est en réalité géré par une classe fille d'EtatCatalogue (nous n'avons ici que deux états différents: **RienAPublier** et **PublicationPlanifieee**), ce qui fait que chaque classe n'a qu'un objectif, et qu'un changement fonctionnel tel que l'ajout de nouveaux cas particuliers pourra se faire en ajoutant de nouvelles classes filles d'EtatCatalogue.

La méthode Ajouter(nouveau:Livrable) du Catalogue ne fait plus que déléguer à la méthode Ajouter(nouveau:Livrable) de l'état courant. Et c'est bien là que tout se joue: selon le type réel de l'état courant, le comportement sera différent grâce au polymorphisme.

Bien sûr, le diagramme de classes ne représente que la partie statique du Pattern; il faut également implémenter les transitions, qui correspondent simplement au "changement d'état", c'est-à-dire à l'affectation au Catalogue d'une référence vers le nouvel état. Cette affectation est souvent effectuée à l'initiative des états eux-mêmes, à qui

l'on fait porter la responsabilité de connaître (ou de choisir) l'état suivant.

Outre l'évolutivité, ce Pattern permet d'implémenter à la perfection les diagrammes d'états-transition UML et d'avoir une bijection, donc une traçabilité parfaite, entre le nombre d'états dans les diagrammes et le nombre de classes de conception.

3.2.4. Comment choisir la prochaine publication?

Finissons-en avec le processus de publication: vous vous êtes certainement dit que le choix de la prochaine publication pouvait se faire de diverses manières (la première soumise est la première publiée, ou les Articles avant les News, ou une alternance savante entre Articles et News...). Or celle pour laquelle nous allons opter aujourd'hui pourrait bien ne pas convenir, auquel cas il faudrait en changer demain.

Il faut donc trouver une technique élégante pour fixer un choix aujourd'hui et être capable d'en changer demain sans qu'il n'y ait d'impact sur le reste de l'application. La solution ressemble au Pattern précédent et tire elle-aussi profit de la délégation:

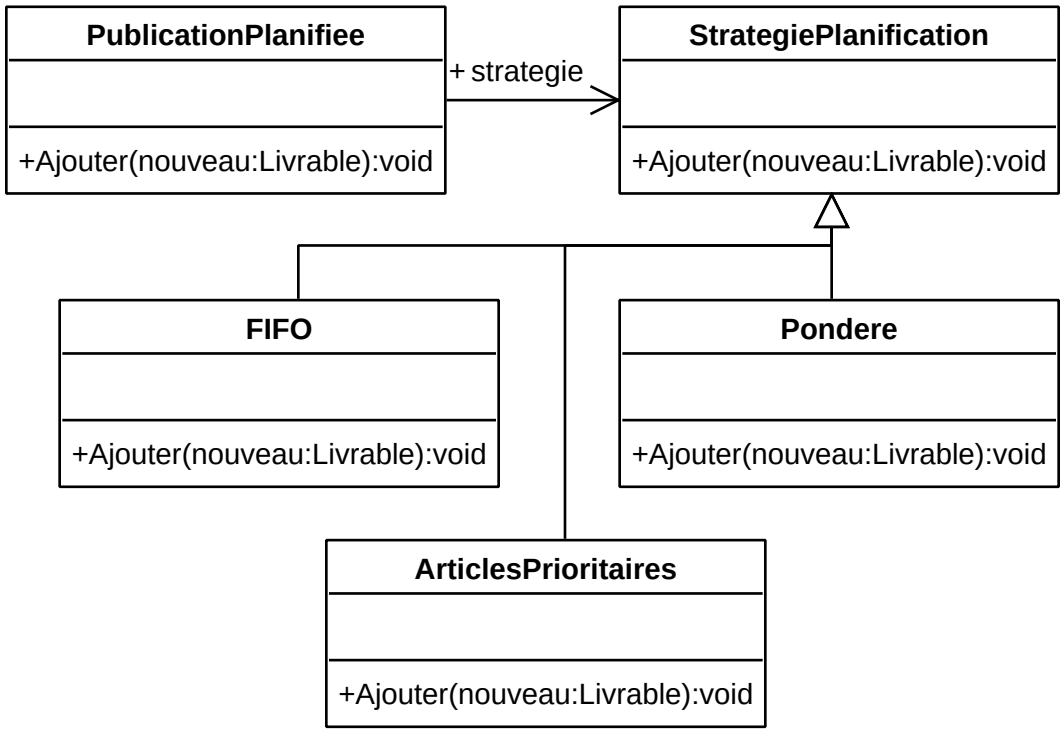


Figure 8. [Design Pattern] Stratégie

Stratégie et État se ressemblent donc énormément dans leur conception. La sémantique, elle, est bien différente car si la stratégie est choisie par paramétrage de l'application ou par le biais d'une interface utilisateur, les états quant à eux évoluent en autonomie au cours du fonctionnement de l'application.

3.2.5. Comment exporter les Livrables dans un format neutre?

Dans notre conception initiale, nous avons choisi d'ajouter la méthode ExporterEnXML() à chaque objet métier, leur faisant ainsi porter la responsabilité de "s'exporter eux-mêmes" dans ce format. En effet, qui mieux que l'objet Article pourrait savoir qu'il faut exporter l'attribut **urlContenu** sous forme d'une balise **<UrlContenu>** par exemple?

Mais critiquons ce choix. D'une part, il donne aux objets métier une responsabilité technique et l'inversion des dépendances nous rappelle qu'une telle conception n'est pas recommandable. D'autre part, imaginons que le format du document XML d'export change (son schéma

évolue): nous risquons alors d'avoir à apporter des modifications dans le code de plusieurs objets métier, au pire dans tous! C'est inacceptable. De plus, cette approche ne permet d'avoir qu'un seul format d'export. En supporter plusieurs requerrait d'ajouter autant de méthodes techniques, sur tous les objets métier, que de formats différents (binaire, CSV, un autre schéma XML...).

Pour toutes ces raisons, nous ne pouvons pas nous contenter de cette conception "brouillon". Pour rationaliser, il faudrait rassembler dans une nouvelle classe les spécificités du format d'export, tout en conservant la possibilité de choisir dynamiquement parmi plusieurs formats en fonction des besoins. Cerise sur le gâteau, il faudrait que les objets métier soient le moins couplé possible à ce mécanisme d'exportation, de même l'algorithme d'exportation à la structure des objets en mémoire, en particulier à leur organisation (en liste, en arborescence...). Que pensez-vous de cette nouvelle conception:

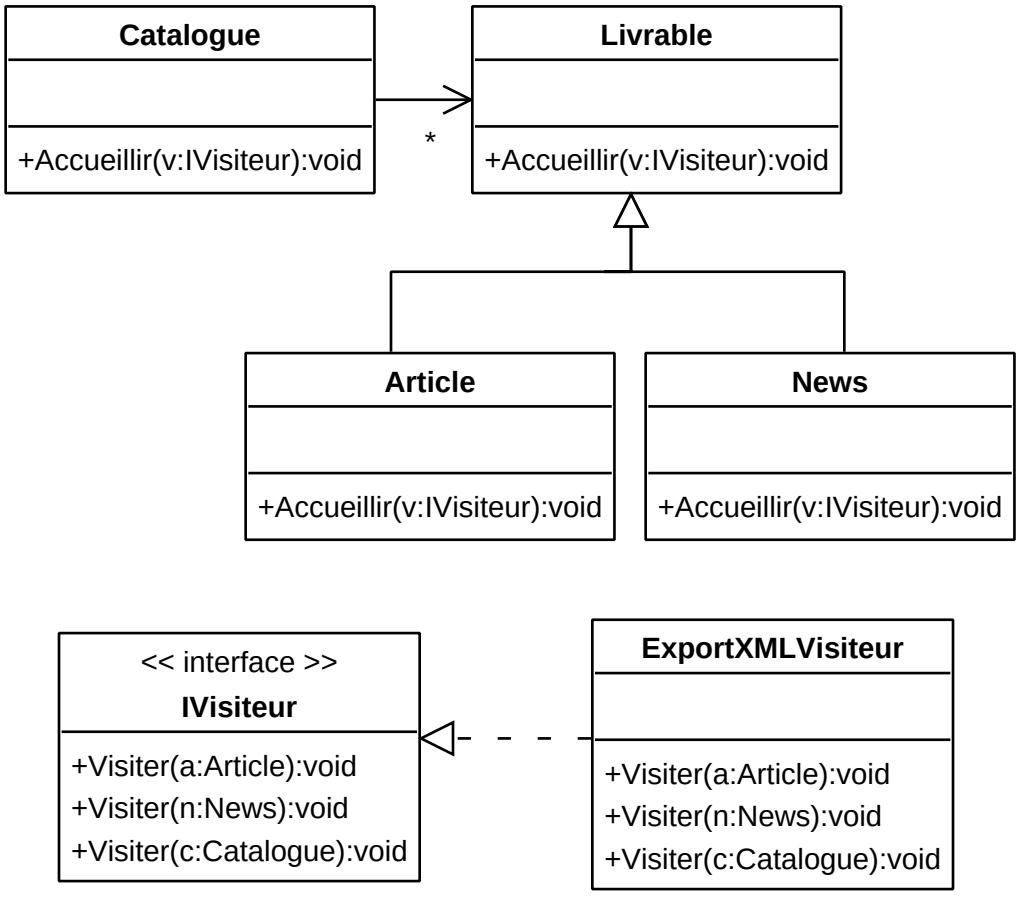


Figure 9. [Design Pattern] Visiteur

Vous l'aurez compris, la seule chose que l'on demande aux objets

métier est "d'accueillir" un **Visiteur** et de le faire progresser sur l'ensemble des objets dépendants: par exemple, le Catalogue invoquera la méthode **Visiter(this)** sur le Visiteur, puis bouclera sur les livrables dont il a la charge pour que le Visiteur ait l'occasion de tous les parcourir.

Nous avons ainsi découplé les objets métier de la technique d'export, ce qui nous permet d'imaginer autant de formats différents que cela est nécessaire. Chaque format donnera lieu à une nouvelle implémentation du IVisiteur.

3.2.6. Parcourir les livrables soi-même

Le Visiteur que nous avons mis en œuvre précédemment n'est pas le seul à devoir parcourir le graphe des objets métier. Certains algorithmes auront eux aussi ce besoin, mais tous ne pourront pas devenir des visiteurs, soit parce qu'ils voudront parcourir le graphe de diverses manières (parcourir les news d'abord, les articles ensuite, ou encore un parcours de livrables triés ou filtrés selon certains critères), soit parce qu'ils souhaiteront piloter le parcours au lieu de se laisser guider par le graphe d'objet lui-même.

Dans ce cas, il faudrait que les classes clientes des objets métier (les algorithmes qui utilisent ces objets) aient un accès en lecture à toutes les propriétés mono- ou multi-valuées. Or prenons l'exemple du Catalogue: s'il donnait à ses client la visibilité directe de sa collection de Livrables, rien n'empêcherait ceux-ci d'ajouter ou de supprimer des Livrables sans que le Catalogue ne puisse exercer de contrôle! **Nous venons de briser l'encapsulation**.

Pour contourner ce problème tout en permettant aux clients d'arpenter librement le modèle objet, il existe deux Design Patterns: le **Décorateur** et l'**Itérateur**. Le premier permet normalement d'enrichir le comportement nominal d'un objet, mais nous allons l'utiliser ici dans un mode dégradé: lorsqu'un client souhaitera parcourir la collection des Livrables, le Catalogue lui fournira bel et bien un objet de type Collection, mais dont seules les opérations de lecture seront autorisées.

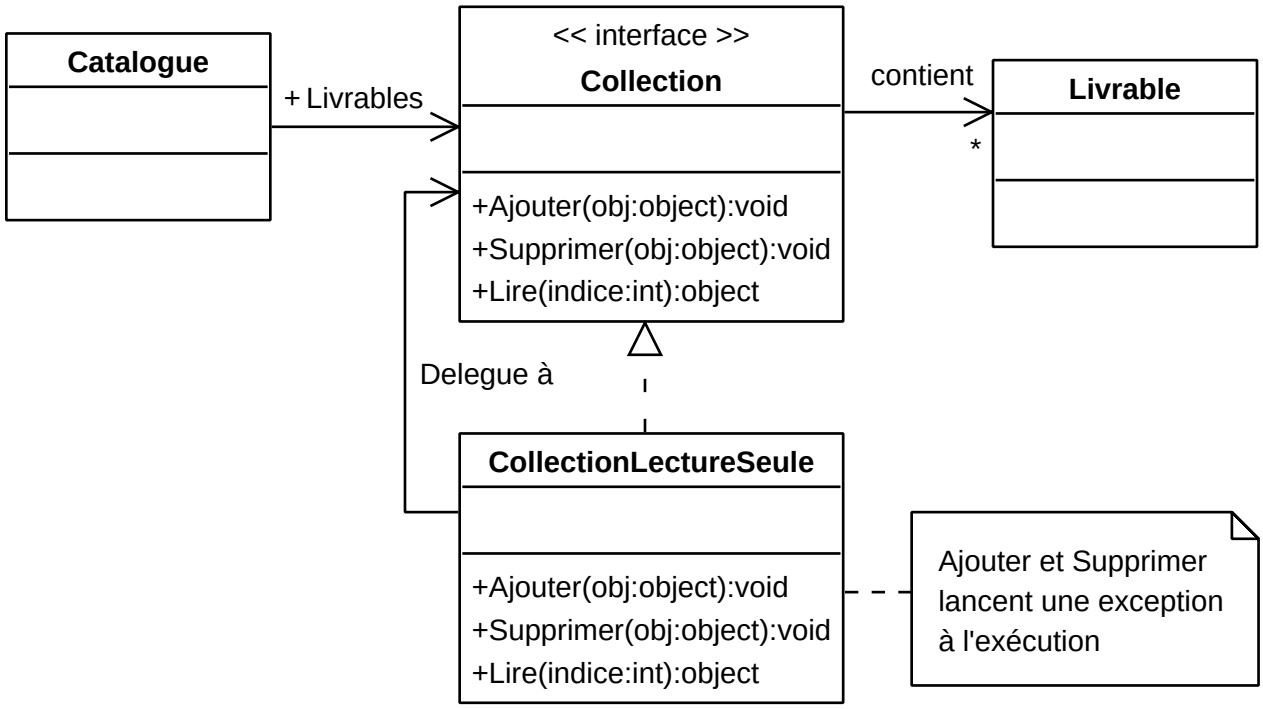


Figure 10. [Design Pattern] Décorateur

Cela peut paraître satisfaisant, mais en réalité, nos principes de conception ne sont pas tous satisfaits. En particulier, **notre décorateur ne répond pas au principe de substitution de Liskov** puisque l'utilisation d'une CollectionLectureSeule en lieu et place d'une Collection n'est pas neutre et ne peut pas être traitée sans précaution supplémentaire: l'utilisateur typique d'une Collection ne sera pas prêt à rattraper les exceptions que lèverait une CollectionLectureSeule! En décorant ainsi la Collection, **nous avons donc rompu le contrat** qui la lie à ses utilisateurs (celui de répondre aux services Ajouter() et Supprimer() sans lever d'exception). D'un autre coté, cette solution est très simple à mettre en œuvre et sa consommation en ressources supplémentaires est négligeable; elle ne requiert pas, par exemple, de dupliquer la Collection que les clients souhaitent parcourir.

Dans le cas où l'on souhaite maximiser la robustesse ou faire varier le mode de parcours d'un client à travers le graphe d'objets métier, notre second Pattern est plus adapté: l'**Itérateur**. A l'image de ce que proposent la plupart des bibliothèques d'accès aux données avec la notion de Curseur, l'Itérateur est un objet dédié à un seul client à la fois et qui permet de parcourir en lecture seule une collection d'objets. Outre le fait qu'il simplifie le parcours à travers une structure de données,

l'Itérateur est également un excellent outil de découplage entre la collection d'objet parcourus et celui qui l'arpente. En effet, quelle que soit l'organisation interne des objets, on utilise un Itérateur toujours de la même manière.

L'Itérateur, quant à lui, doit avoir une relation privilégiée avec la collection dont il facilite la traversée. Or l'organisation des objets peut varier en fonction des types de collections (listes chaînées, listes basées sur des tableaux, arborescences triées...), donc il est nécessaire de disposer d'**un type d'itérateur par type de collection parcourue**. Enfin, toujours pour masquer l'implémentation à l'utilisateur, il faut faire porter à la Collection d'objets elle-même la responsabilité de créer le bon type d'Itérateur (en fonction de sa structure de données interne). On dit que la Collection se comporte comme une **Fabrique** (ou qu'elle porte une **méthode de fabrication**).

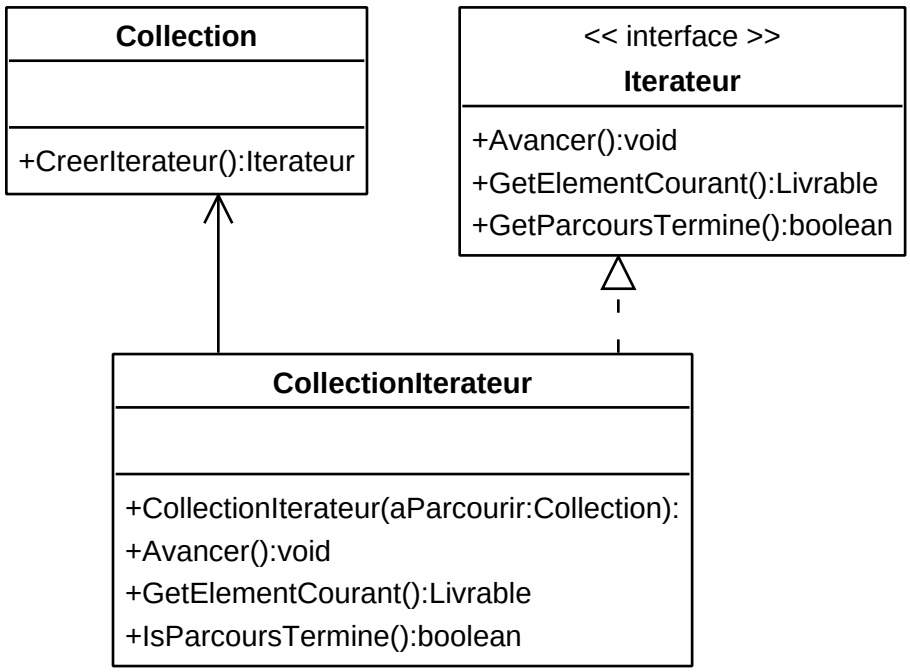


Figure 11. [Design Pattern] Itérateur

A noter que l'Itérateur que nous avons modélisé est très ancré dans le modèle des objets métier et qu'il ne pourrait pas être réutilisé dans d'autres situations. Cette contrainte a amené les concepteurs des frameworks Java et .NET, par exemple, à proposer des Itérateurs génériques qui parcourent et renvoient n'importe quel "Object". L'avantage induit est bien évidemment la souplesse, qui se paie par une

diminution de la robustesse de l'application car on peut se tromper lors du transtypage des objets parcourus et le compilateur n'est d'aucune aide puisque pour lui, l'Itérateur peut se promener sur n'importe quel Object. Cette gêne n'existe pas en C++ où les **Templates** permettent d'implémenter des Itérateurs paramétrés par le type d'objets qu'ils parcourent; et fort heureusement, les langages Java 1.5 et C# 2.0 nous proposent un équivalent aux Templates, les **Generics**.

D'autre part, on trouve également dans certains frameworks des Itérateurs "améliorés" qui permettent de supprimer l'objet courant lors d'une traversée de structure de données. Appelons ce type d'Itérateurs des "**Itérateurs modifiables**". Ceux-ci s'avèrent très pratiques comparés aux autres techniques de modification des Collections au gré d'un parcours. Mais à mieux y réfléchir, si nous renvoyons un Itérateur modifiable à un utilisateur de nos classes, par exemple à un utilisateur du Catalogue, rien ne l'empêcherait de parcourir les Livrables et d'en supprimer certains sans que le Catalogue ne puisse effectuer le moindre contrôle. Ce qui revient à nouveau à briser l'encapsulation: il est quasiment aussi grave de renvoyer un Itérateur modifiable qu'une référence sur la Collection d'objets elle-même.

3.2.7. Interactions avec le système

Avoir modélisé et conçu les objets métier d'un système est satisfaisant car on peut dès lors écrire du code qui se lit comme de la bonne prose; du moins, il parle à une personne du domaine que la syntaxe d'un langage de programmation ne fait pas fuir... Mais si l'on n'y prend garde, solliciter un modèle objet directement amène souvent à un problème de traçabilité. En effet, si n'importe quel objet (d'une couche de présentation par exemple) pouvait invoquer les méthodes des objets métier, non seulement nous aurions de nombreuses dépendances mais il deviendrait surtout très difficile de dire à quel besoin métier correspond chaque invocation ou à quelle motivation de l'utilisateur final cette invocation se rattache.

Afin de rationaliser l'interaction entre la **couche de présentation** et la **couche d'objets métier** , les architectures "multi-couches" nous recommandent d'insérer un intermédiaire, souvent appelé **couche de**

services, qui expose un ensemble de **Façades** fonctionnelles. Comme dans la vie courante, une Façade offre un point d'accès simplifié à un ensemble de services et permet par là-même de limiter le couplage entre deux couches logiques successives. Une autre manière de le dire: la façade masque les détails dont les utilisateurs ne doivent pas être dépendants.

Dans notre exemple, les utilisateurs finaux souhaitent gérer les publications ainsi que les thèmes qui les regroupent. Une seule Façade de **GestionDesPublications** devrait suffire à rassembler les principaux services du système; elle devient dès lors un point d'entrée obligatoire pour les utilisateurs (un contrôleur, souvent proche des cas d'utilisation déterminés en analyse).

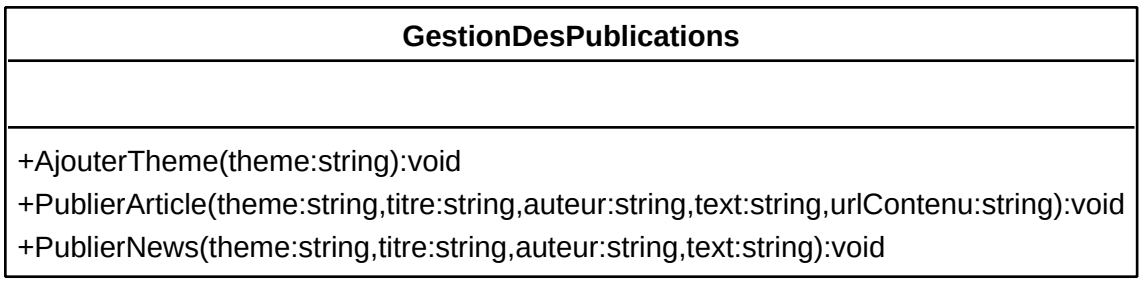


Figure 12. [Design Pattern] Façade

Comme nous pouvons le constater, la Façade est très simple à utiliser puisqu'elle se charge de tout:

- La création des objets nécessaires. La Façade peut s'appuyer sur une Fabrique (éventuellement abstraite) pour déléguer la responsabilité d'instancier les objets concrets.
- L'initialisation complémentaire de ces objets. Certains objets n'offrent qu'un constructeur sommaire (ou leur fabrique n'offre qu'une méthode acceptant peu de paramètres); il s'agit donc d'affecter les propriétés qui n'ont pas pu l'être par le biais de ce constructeur.
- L'invocation de méthodes, dans un ordre compatible avec les workflows applicatifs et métier.

La Façade GestionDesPublications est une classe sans attribut. Ses instances sont donc sans état, indistincts. Dans ce cas, il serait

dommage d'en créer plusieurs puisque chacune de ces instances est parfaitement identique à sa petite sœur.

Pour éviter ce gâchis, il convient d'interdire la création à tout va d'instances de GestionDesPublications. Comment? En rendant son constructeur inaccessible aux clients extérieurs (rendons-le **privé** par exemple). Mais de ce fait, il devient nécessaire d'offrir un autre point d'entrée à la GestionDesPublications, qui ne nécessite pas de l'instancier au préalable: seule une méthode statique peut faire l'affaire. Et donc sans surprise, cette méthode va permettre soit d'instancier la GestionDesPublications (la première fois qu'on l'invoque), soit de récupérer une référence à cette instance devenue unique (les fois suivantes).

Cette gymnastique de l'esprit, *qui part du besoin de partager des informations ou d'optimiser l'occupation mémoire par l'interdiction de créer plusieurs instances, et qui aboutit à une classe à l'instance unique accessible par le biais d'une méthode statique* porte un nom: le Design Pattern **Singleton**. Dans la phrase précédente, on comprend comment les Patterns permettent d'élever le niveau du langage et de limiter les ambiguïtés entre interlocuteurs avertis.

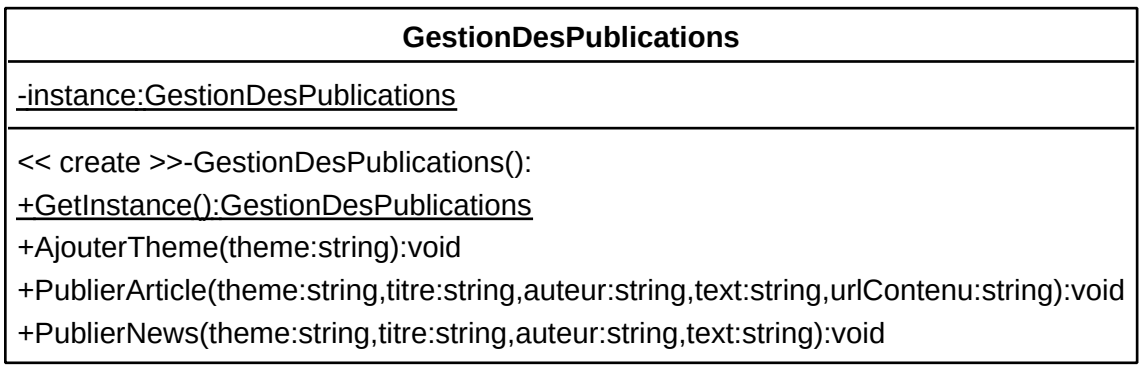


Figure 13. [Design Pattern] Singleton

3.2.8. Amélioration de la traçabilité

Introduire une Façade a certes simplifié l'accès aux fonctionnalités essentielles de notre système, mais cela n'a pas amélioré la traçabilité ni la gestion de l'historique des interactions entre l'utilisateur final et le

système. Mais nous sommes près du but: puisque nous ne pouvons cataloguer que des objets, et non des invocations de méthodes, il faudrait simplement que chaque sollicitation de la classe GestionDesPublications soit **réifiée en objet**, c'est-à-dire qu'il nous faudrait instancier un objet pour rendre compte de chaque invocation des méthodes AjouterThème(), PublierArticle() et PublierNews().

Afin de banaliser la gestion de toutes ces interactions, nous allons nous efforcer de respecter le même contrat dans chacune de ces nouvelles classes de conception. Typiquement, elles offriront toutes une méthode **Execute()** qui permettra de déclencher l'invocation du service associé. Seules varieront les méthodes qui permettront de fournir les informations spécifiques nécessaires à l'exécution de chaque commande (les "Setters"), ainsi que les méthodes de récupération des résultats éventuels (les "Getters"). Cette manière de faire s'appelle le Pattern **Commande**:

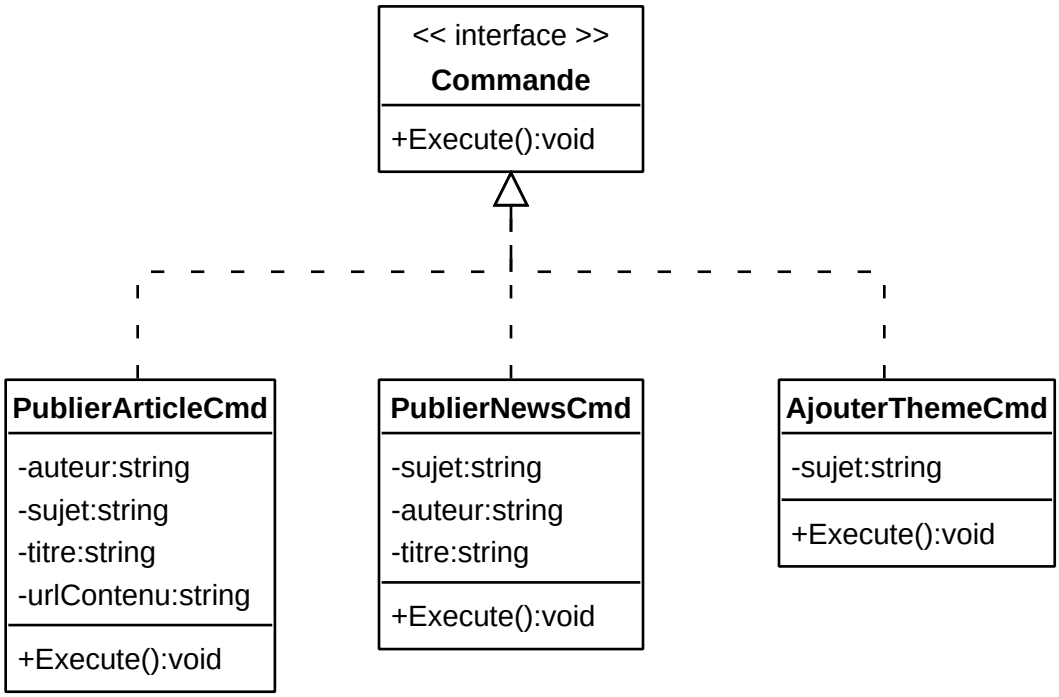


Figure 14. [Design Pattern] Commande

La Commande présente l'avantage de simplifier la traçabilité entre les besoins des utilisateurs finaux et leur implémentation dans le système. Mais tout de même, quelle lourdeur pour arriver à quelque chose qui ne semble pas très éloigné de la simple Façade précédente!

Non, en réalité, la valeur ajoutée de la Commande sur la Façade est ailleurs. Chaque invocation de service est maintenant un objet en mémoire, et il en découle de nombreuses (bonnes) propriétés:

- Il est maintenant possible de créer un **historique** de tous les services sollicités par les utilisateurs finaux.
- En cas de problème majeur, il sera donc toujours possible de **rejouer les dernières commandes** d'un utilisateur pour remettre le système dans un état cohérent.
- Certains domaines peuvent tirer parti de la connaissance des opérations déclenchées par un client. La gestion de la relation client en particulier est friande de ce type de renseignement. De même pour la gestion de la navigation des utilisateurs à travers un site Web.
- Comme nous disposons d'un historique chronologique de commandes, il devient possible de **défaire** ce que les commandes ont **fait**, dans le sens inverse, puis éventuellement de **rejouer** les commandes annulées.
- Les commandes seront exécutées soit par celui qui en fait la demande, soit par un intermédiaire que nous appellerons l'**ordonnanceur de commandes**. Ce dernier peut tout à fait gérer une notion de priorité entre commandes, traiter certaines de manière asynchrone ou même déléguer l'exécution de certaines commandes à d'autres nœuds de traitement du système (d'autres serveurs, dans le cas d'un cluster par exemple).

Remarque: ce Pattern a été poussé à l'extrême dans l'approche par **Prévalence**. Suivant les préceptes de Bertrand Meyer, les outils comme Prevayler et Bamboo.NET vont jusqu'à distinguer les commandes qui ont un effet sur l'état du système et celles qui ne font que le consulter. Seules les premières sont enregistrées dans un journal de commandes (typiquement sur disque dur, par sérialisation), ce qui permet, associé à un mécanisme de persistance périodique des objets métier, d'obtenir un système complètement fiable, persistant, orienté objet... sans utiliser la moindre base de données. Une initiative très intéressante et prometteuse...

3.2.9. Gérer les événements applicatifs

Imaginons que les administrateurs du système de publication souhaitent avoir sous les yeux un tableau de bord graphique qui leur permette de prendre connaissance en temps réel de l'état des demandes de publications de nouveaux Articles.

La première technique que l'on peut mettre en œuvre est simple: il suffit de scruter le Catalogue de Livrables et de réagir en déclenchant une alarme dès que l'état d'un Livrable change. Mais pour cela, il faudrait avoir fait une copie de l'état de chaque objet et comparer cet état "précédent" avec l'état "courant" de chaque objet lors de la scrutation périodique. C'est envisageable si le nombre d'objets à scruter n'est pas très important ou si la fréquence de scrutation est faible.

Dans le cas contraire, il serait plus judicieux d'inverser la dépendance. Après tout, celui qui paraît le plus à même de savoir quand change l'état d'un Livrable, c'est bien le Livrable lui-même. Il suffit donc de faire en sorte que ce dernier puisse informer, notifier les objets intéressés qu'il est en train de changer d'état. Pour cela, le Livrable doit offrir aux objets extérieurs un moyen de s'abonner auprès de lui et de se désabonner ultérieurement. Inversement, chaque objet extérieur doit convenir avec le Livrable de la méthode à déclencher pour notifier son changement d'état. Ce mécanisme d'**Abonnement-Publication**, que l'on aperçoit également dans les Middleware Orientés Message, porte le nom de "Pattern **Observateur-Observable**".

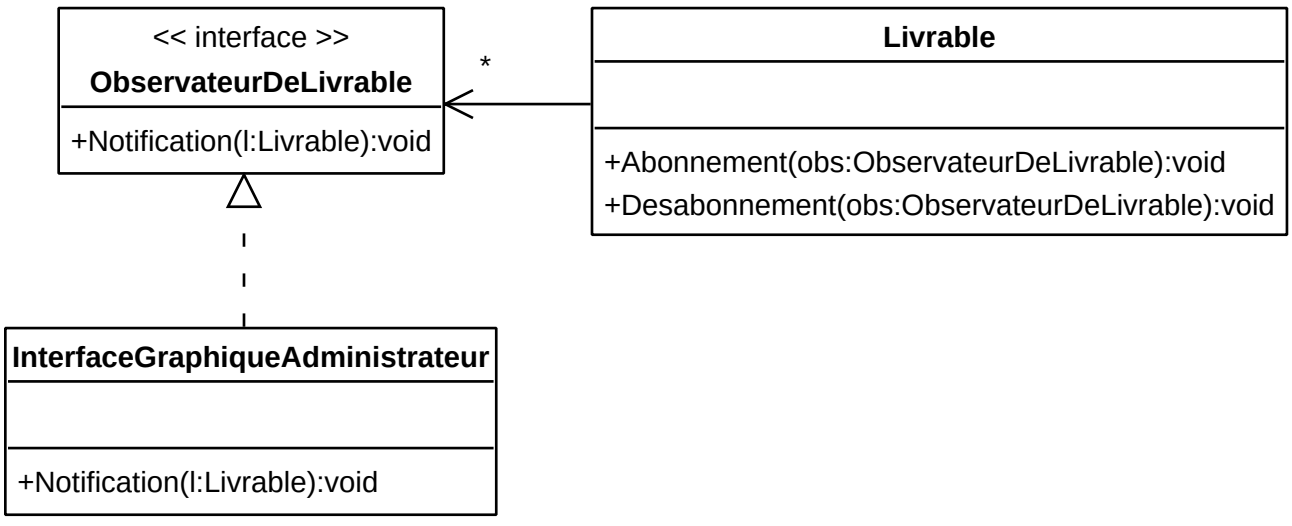


Figure 15. [Design Pattern] Observateur - Observable

La même remarque déjà formulée au sujet de l'Itérateur s'applique à nouveau ici: notre Observateur est ancré dans le modèle métier du système de publication puisqu'il ne peut être notifié que des modifications de l'état d'un Livrable. La généricité permet de généraliser ce Pattern tout en conservant un typage fort.

D'autre part, en particulier si l'on transpose l'Observateur-Observable dans une architecture distribuée, une question revient souvent: vaut-il mieux passer systématiquement l'objet dont l'état change en paramètre ou se limiter à sa référence, son identifiant? Même s'il est difficile de trancher dans le cas général, la première solution est souvent choisie:

- Elle n'implique que n invocations de méthodes à distance (autant que d'observateurs), alors que la seconde technique en implique jusqu'à 2*n voire même davantage (tous les Observateurs risquent d'invoquer une méthode sur l'Observable pour prendre connaissance de la modification qui a eu lieu, et éventuellement pour récupérer le nouvel état de l'objet).
- Elle garantit la cohérence des notifications. Dans la seconde approche, les Observateurs reviennent vers l'Observable pour récupérer les informations complémentaires; or ce dernier risque d'avoir encore changé d'état entre temps! Un état intermédiaire peut ainsi ne jamais être porté à la connaissance de certains Observateurs.

3.2.10.Améliorer les performances du système

La plupart des Design Patterns que nous avons abordés apportent à notre système plus de souplesse, d'évolutivité et parfois de maintenabilité. Mais rares sont ceux qui visent à une amélioration des performances. Terminons notre voyage dans le monde des Patterns avec quelques considérations sur ce thème.

Dans un langage à Objets, deux éléments techniques sont très consommateurs de temps: il s'agit de la construction et de la destruction des objets. Si nous pouvions trouver une technique pour les limiter au strict minimum, nous gagnerions certainement un temps considérable.

Remarque: le Singleton allait déjà dans ce sens, mais il ne s'applique

pas au cas général des classes instanciées de nombreuses fois.

Prenons l'exemple des Articles et des News dans notre application. De nombreuses instances de chacune de ces deux classes seront créées au cours de l'exécution de l'application, mais si nous utilisons un mécanisme de sauvegarde, il ne sera pas nécessaire de conserver constamment tous ces objets en mémoire. Donc certains objets, une fois sauvegardés, ne nous seront plus utiles et nous pourrons ainsi les libérer.

Au lieu de créer et de "jeter" nos objets après usage, ce qui donnera en Java et en .NET un gros travail au Ramasse-Miettes, nous ferions mieux de les recycler. Cette solution, plus écologique, est également plus efficace puisqu'il ne sera plus nécessaire d'allouer et de désallouer autant d'espaces mémoire qu'auparavant, ni de défragmenter la mémoire aussi souvent. Mais qui doit décider de créer, recycler ou détruire les objets? Sur quels critères?

Pour être certain de gérer ce problème de manière cohérente, il faut affecter **la responsabilité de gérer tout le cycle de vie** des Articles à la même classe, que nous appellerons **FabriqueArticles**. A travers notre application, dès que l'on aura besoin de créer un nouvel Article, il faudra s'adresser à elle. De même, après en avoir terminé avec un Article, il faudra veiller à "le lui rendre", de telle sorte que le recyclage puisse avoir lieu.

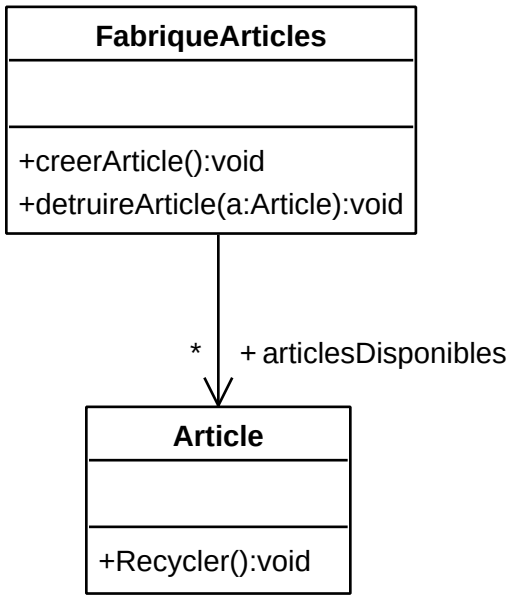


Figure 16. [Design Pattern] Fabrique et Pool d'objets

De son côté, la fabrique gère une collection d'Articles disponibles (c'est-à-dire existant en mémoire mais qui ne sont plus utilisés par personne). Lorsqu'une classe cliente lui demande de créer un nouvel Article:

- Si elle dispose d'Articles disponibles, elle en prend un, le recycle et le renvoie à l'appelant.
- Dans le cas contraire, elle peut prendre la décision d'instancier un nouvel Article, mais comme il est probable qu'on lui demande d'en créer encore un nouveau très peu de temps après, elle peut très bien prendre l'initiative d'instancier plusieurs Articles d'un coup. Ainsi elle se constitue un stock et répondra plus rapidement aux demandes suivantes.

Cette technique, appelée le **Pool d'objets**, suppose toutefois une chose: chaque Article doit savoir se recycler, c'est-à-dire se réinitialiser dans le même état que s'il venait d'être instancié. Ainsi les utilisateurs ne voient aucune différence entre un nouvel Article et un Article recyclé. D'aucuns diront que le principe d'inversion des dépendances n'est pas respecté à la lettre puisqu'un objet métier "s'abaisse à des considérations techniques"... Nous y reviendrons dans les chapitres suivants.

3.3. Fin du voyage

Nous avons utilisé un certain nombre de Design Patterns. Il en existe beaucoup d'autres, souvent associés à des domaines plus spécialisés tels que la distribution d'objets à travers un réseau, le mapping Objet/Relationnel, l'interaction utilisateur... Nous les introduirons au besoin, au fur et à mesure de notre progression dans le monde des Aspects.

Pour vous assurer de bien avoir intégré les notions de ce chapitre, amusez-vous à formuler des phrases au sujet de vos applications ou de situations de la vie courante, en usant (et en abusant) de Design Patterns. Par exemple:

- Les tour-operators sont des Façades qui agrègent plusieurs services (transport, hébergement, restauration, circuit touristique). Leurs agences sont des Observables auxquels les clients s'abonnent pour être notifiés en cas de changement d'horaire des transports.
- Ma femme est un singleton. Elle est l'instance unique d'une classe rare.
- Les contrôleurs de Gaz et d'Electricité possèdent un Itérateur sur une collection d'adresses dont ils doivent relever les compteurs.
- Les contrôleurs des Impôts, eux, sont des Visiteurs: un foyer fiscal les "accepte" et les fait parcourir spontanément tous les éléments constitutifs du capital ou du revenu imposable. Ils ont une méthode apte à gérer chaque type de revenu (immobilier, mobilier, salaires).
- Les sociétés de vente par correspondance implémentent le pattern Commande. Leurs clients peuvent passer Commande (par le biais de la Façade que représentent un site Web ou un télé-conseiller), mais sont également à même d'annuler leur commande avant son exécution ou de renvoyer le contenu de cette commande pour se faire dédommager après son exécution.
- Après avoir passé plus de 30 minutes dans les bouchons, un conducteur change d'Etat. Les mêmes événements issus du trafic routier auront sur lui un tout autre impact.

Finalement, nous pouvons dire que nous avons mis sur pied un **Langage de Patterns**. Plus notre entourage professionnel (analystes, concepteurs, développeurs, chefs de projet ou d'équipe et autres intervenants d'un projet) maîtrisera ce niveau de langage, et plus nous deviendrons précis et concis lorsque nous cernerons de nouveaux problèmes ou que nous en concevrons les solutions. Dans le reste de ce livre, qui se consacre entièrement à la Programmation Orientée Aspect, nous utiliserons sans cesse les Design Patterns comme base de solution des problèmes qui vont se présenter à nous.

Chapitre 4.

Dépasser les limitations de l'Objet

Ce chapitre établit le bilan des bienfaits de la conception Objet et de l'application de Design Patterns, mais établit également le constat des échecs ou des limitations de ces outils.

Dans un deuxième temps, il nous propose un ensemble de techniques complémentaires susceptibles de dépasser les limitations actuelles. Ces éléments nous mèneront en douceur à la Conception et à la Programmation Orientées Aspect.

4.1. Constats

4.1.1. Une conception bien conceptuelle

Le chapitre précédent s'appuyait très fortement sur les principes "sacrés" de la conception Objet pour introduire quelques exemples de conception réussis et réputés: les Design Patterns. Ces derniers offrent une solution très élégante aux problèmes récurrents, mais ils restent toutefois complexes à appréhender ou à mettre en œuvre pour certains profils.

Cela peut être dû à un problème de conceptualisation: les notions Objet telles que les classes et interfaces, l'héritage et le polymorphisme paraissent déjà un peu abstraits à certains. Il semble donc normal que de remonter d'un niveau d'abstraction avec la notion de Pattern ne simplifie pas forcément les choses. Mais que ce soit un problème d'acquisition de compétences ou d'un manque d'habitude de réflexion abstraite, il n'en reste pas moins que certains ne souhaitent pas s'investir trop avant dans ces domaines de conception avancée.

4.1.2. Limitations de l'approche Objet

Même en utilisant toute la puissance des langages Orientés Objet, même en usant des Design Patterns les plus évolués, il reste dans le cœur de nos applications des éléments de code qui sont impossibles à centraliser, à factoriser. Pire, 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.

Les exemples sont nombreux en particulier en-dessous de la granularité d'une méthode: dès que l'on souhaite mettre en facteur un ensemble d'instructions, les langages actuels nous obligent à réunir ces instructions en méthodes, ce qui n'est pas toujours possible ou élégant.

D'autre part, et nous y reviendrons dans la section suivante, il est amusant de noter que les principes de conception Objet interdisent

certaines relations, en particulier d'héritage, qui rendent donc assez complexe la résolution de problèmes simples.

Outre ces problèmes conceptuels, que nous rencontrons donc dans tous les langages à Objets, les langages eux-mêmes souffrent de limitations qui rendent notre tâche de développement logiciel parfois fastidieuse. Par exemple, 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++ ou Eiffel permettent encore, Java et C# l'interdisent en clamant que "ce n'est pas sain d'hériter de plusieurs classes".

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. Une bonne pratique, oui, mais 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. N'aurait-il pas mieux valu comme en C# proposer une syntaxe uniforme entre propriétés (l'équivalent des accesseurs) et attributs, de telle sorte que nous puissions définir directement certains attributs ReadOnly sans devoir maintenir à la fois un attribut et une méthode getXxx()? La redondance est rarement signe d'intelligence...

Un autre exemple mal implémenté par de nombreux langages Objet: la délégation. Souvenez-vous de l'exemple du Catalogue de Livrables du chapitre précédent: ce Catalogue doit avoir le comportement d'une Collection de Livrables (et supporter l'ajout, la suppression, le parcours, etc...). Deux options s'offrent à nous:

- soit le Catalogue hérite de Collection et bénéficie de tout le comportement générique de sa classe de base,
- soit le Catalogue délègue à un autre objet de type Collection la gestion de la liste des Livrables dont il est le responsable.

Pour éviter qu'un objet métier ne dépende trop fortement d'un objet plus technique (ici la Collection), ou pour laisser la possibilité au

Catalogue d'hériter d'une autre classe de base, on choisit très souvent la seconde solution. Il faut donc exposer dans la classe Catalogue un ensemble de méthodes (pas nécessairement toutes) qui seront ensuite déléguées à l'objet Collection. Le code de ces méthodes de délégation est trivial, mais terriblement répétitif. Nous aurions pu attendre de langages aussi récents que Java et surtout C# qu'ils incluent un mécanisme de délégation intégré.

Un certain nombre de problèmes se posent donc au niveau des principes et des outils qu'offrent soit l'approche Objet elle-même, soit les langages qui la supportent.

4.1.3. Les Design Patterns sont contre-nature

Les meilleures techniques de conception Objet sont aujourd'hui bien connues et répertoriées sous forme de catalogues de Design Patterns. Grâce à cet effort de documentation de la part des experts et à la bonne adoption de ces Patterns par les concepteurs et les développeurs, un certain nombre de réflexes ont pu se mettre en place:

- De plus en plus de projets sont organisés en couches de responsabilité bien identifiées.
- Chaque couche est généralement exposée par le biais d'une ou plusieurs Façade.
- Les Façades sont souvent des Singletons.
- Les couches d'accès aux données "simples" emploient très souvent un objet intermédiaire pour limiter le couplage entre les objets métier et la structure de la base de données: un DAO (Data Access Object), parfois associé à une Fabrique de DAO.
- Les couches plus fines d'accès aux données, souvent intégrées aux frameworks de mapping Objet/Relationnel peuvent utiliser une Fabrique de Poids Mouches (Flyweight), ou du moins une fabrique de Proxies (l'objectif étant d'implémenter un cache intelligent des objets persistants et de ne cloner un objet que lorsqu'un utilisateur lui apporte une modification et que l'on ne souhaite pas la répercuter sur l'activité des autres utilisateurs avant une validation et un

enregistrement en base de données).

- Il est rare de voir des couches de présentation qui n'implémentent pas le pattern MVC (Modèle Vue Contrôleur).

Bref, les applications multi-couches modernes sont truffées de Design Patterns. D'ailleurs, les frameworks de développement tels que J2EE et .NET aussi: on y trouve les patterns Observer, Itérateur, Singleton et Fabrique à plusieurs reprises.

Ce n'est certainement pas moi qui vais militer contre les Design Patterns: je suis intimement convaincu de leur utilité, de la souplesse et parfois même des performances qu'ils amènent aux applications. Mais analysons froidement les applications actuelles:

- Elles regorgent de Patterns, ce qui nécessite de travailler avec des développeurs-concepteurs toujours plus compétents. Ces derniers doivent connaître leur(s) langage(s) de programmation, les responsabilités de chaque couche de l'application, ainsi qu'une trentaine de Patterns en moyenne.

- L'utilisation systématique de Patterns est à double tranchant: le projet peut gagner en lisibilité pour les nouveaux entrants qui connaissent déjà tous les Patterns utilisés, mais peut s'avérer impénétrable pour ceux qui n'ont pas cette connaissance.

- L'application de Design Patterns fait très souvent apparaître de nouvelles classes. Si cette approche amène à une évolutivité supérieure et donc une maintenance évolutive plus aisée, elle rend plus coûteuses les phases de refactoring et de maintenance corrective.

- Enfin, les développeurs d'objets métier ou de services applicatifs se doivent d'appliquer les Patterns adéquats.

Revenons sur ce dernier point: il signifie que le modèle des objets métier ou de la couche de service doit être conforme aux Design Patterns. De manière un peu raccourcie, nous pourrions reformuler cela en: *"il faut polluer le modèle métier de Patterns de conception"*. Il sera donc très difficile d'implémenter un modèle qui soit purement du niveau métier; généralement, la couche métier intègre des classes issues de l'analyse et de la conception (donc entre autres celles induites par

l'application de Design Patterns).

Pour les puristes, cela pose deux problèmes. Tout d'abord, un problème de traçabilité car les concepts métier se voient implémentés en plusieurs classes: si nous prenons l'exemple du Pattern Etat, et si un objet peut se trouver dans n états, il sera implémenté en au moins n+2 classes (la classe issue de l'analyse, un état abstrait et une classe concrète pour chaque état différent). Que ce soit dans les diagrammes UML ou dans le code du projet, il est difficile de séparer très clairement le métier des artifices techniques apportés par la conception (même si, dans les diagrammes, l'utilisation de couleurs ou de stéréotypes aide à la lisibilité).

Le deuxième problème est celui de l'affectation des responsabilités, à la fois aux composants et aux personnes qui les développent. Quelle classe porte les informations ou la compétence métier? Quelle classe offre un découplage par rapport aux couches connexes ou par rapport à un framework quelconque (de traces, de distribution, de présentation, de persistance...)? Il devient subtil de répondre à ces questions. Or dans l'idéal, il faudrait séparer la responsabilité de l'analyste de celle du concepteur. Allons plus loin, il faudrait que l'analyste puisse travailler sur des classes que le concepteur ne modifierait pas ou peu par la suite. Ainsi, les problèmes de "round-trip engineering" pourraient enfin se réduire voire disparaître puisque le couplage des itérations d'analyse et de conception s'affaiblirait.

Pour résumer, nous sommes tous convaincus de l'intérêt des Design Patterns. Mais ils représentent une compétence et une responsabilité techniques qui doivent être décorrélées de l'analyse (et de la programmation) d'objets métier. Le travail des concepteurs et des équipes responsables des frameworks doit pouvoir être découplé du travail d'analyse fonctionnelle! Dans la section suivante, nous allons passer en revue les diverses solutions ou tendances actuelles qui vont dans ce sens.

4.2. Solutions

4.2.1. Plus d'abstraction dans les frameworks

Il devient de plus en plus rare qu'une application n'utilise aucun framework technique, qu'il s'agisse d'un framework de traces, de persistance, de présentation, de tests unitaires... Mais nous perdons encore beaucoup de temps à assembler ces frameworks pour en faire un tout cohérent (quand la tâche est possible, car les dépendances entre frameworks interdisent certaines associations dans la même application).

De plus, utiliser directement un framework rend nos applications dépendantes de ce dernier; il est souvent coûteux de passer d'un framework à un autre alors que les principes sous-jacents sont souvent les mêmes. Par exemple, porter une application Java basée sur le framework de présentation Struts à IBatis Axle ou Spring MVC n'est ni neutre ni immédiat.

Nous avons donc besoin d'un niveau d'abstraction supérieur. En toute logique, nous pouvons nous attendre à voir apparaître de nouveaux outils dans les temps qui viennent:

- Des **méta-frameworks** devraient faire leur apparition. Ils se placeraient entre nos applications et les véritables frameworks techniques que nous souhaitons utiliser et transposeraient le paramétrage et le code générique en artéfacts nécessaires au bon fonctionnement de ces frameworks sous-jacents. Un exemple de ce type de méta-framework en Java: **commons-logging**, qui permet à une application d'utiliser le framework de traces du JDK1.4, ou Log4J, ou SimpleLog... sans avoir à changer la moindre ligne de code.

- D'un autre coté, l'initiative pourrait venir des frameworks actuels, qui pourraient gagner en abstraction pour imposer moins de dépendance aux applications qui les utilisent (cf. section suivante).

Outre le non-respect du principe d'inversion des dépendances, les frameworks actuels utilisent de manière intensive la réflexion (ou

introspection) ce qui mène à quelques problèmes de robustesse et de performance.

Rappel: [pour les lecteurs qui ne manipulent pas ce mécanisme très souvent] l'introspection est un mécanisme qui permet de découvrir dynamiquement (à l'exécution d'une application) le type d'un objet, ses attributs, ses propriétés et ses méthodes. Réflexion est un synonyme d'introspection. Mais ce n'est pas tout: l'introspection permet également d'invoquer une méthode, de lire un attribut ou d'instancier une classe de manière complètement dynamique, à la manière d'un langage de script dans lequel on ferait un **eval("new java.lang.String")**. Par exemple, il n'est pas rare de placer dans un fichier de configuration le nom d'une classe représentant un Driver d'accès aux bases de données et d'instancier cette classe par introspection. Ainsi, le programme n'a absolument aucune adhérence vis-à-vis de cette classe au moment de la compilation. Cette souplesse se paie bien sûr en termes de performances, même si les machines virtuelles récentes ont nettement optimisé ce mécanisme.

4.2.2. Inversion de contrôle (IOC)

Une nouvelle génération de frameworks fait son apparition depuis quelque temps pour tenter de répondre aux problèmes de dépendances entre nos applications et les frameworks techniques mentionnés dans la section précédente. Par la même occasion, ces nouveaux outils vont essayer de simplifier le code nécessaire à l'intégration d'une application sur son socle technique en limitant le luxe de Patterns employés. Par exemple, l'usage systématique de l'Abstract Factory disparaît dans une application basée sur un framework d'IOC tel que Spring ou Pico.

L'IOC est une intiative intéressante et novatrice, supportée par des acteurs importants du monde de la conception Objet tels que Martin Fowler. Si vous souhaitez en savoir plus que ce que vont résumer les quelques lignes qui suivent, n'hésitez pas à consulter entre autres son site Web: *http://www.martinfowler.com*.

Principe de l'IOC

L'idée est la suivante: au lieu d'invoquer le constructeur d'une classe

pour initialiser les objets dont nous avons besoin dans le corps d'une méthode, nous utilisons souvent des Fabriques, parfois Abstraites. Cette pratique limite le couplage entre les classes, mais le passage obligé par la Fabrique introduit une certaine lourdeur syntaxique. L'IOC propose simplement de définir comme propriétés de la classe utilisatrice tous les objets dont elle aura besoin. Afin de limiter le couplage, on sera bien avisé de choisir des interfaces ou des classes abstraites comme types apparents de ces propriétés.

Une fois que tout est déclaré, les méthodes de la classe utilisatrice peuvent considérer que les propriétés seront correctement initialisées, *comme par magie*. Cette magie est implémentée par le **Conteneur IOC**, qui se base sur un ou plusieurs fichiers de paramétrage XML pour savoir quand et comment initialiser les propriétés de toutes les classes utilisatrices de l'application. Ainsi, la complexité d'initialisation n'est plus située dans la classe utilisatrice (comme avec l'invocation directe du constructeur), ni dans une fabrique abstraite invoquée par cette même classe, mais dans un fichier de configuration complètement externe à cette classe. Le framework d'IOC dépend bel et bien de toutes les classes (utilisatrices et implémentatrices de services) mais les classes, elles, ne dépendent plus que d'interfaces. Nous avons inversé la dépendance, ou inversé le contrôle, d'où le nom de ce mécanisme.

Exemple de code d'application basée sur Spring

Prenons un petit exemple, adapté du tutoriel du framework Spring. Imaginons une classe métier Produit qui s'appuie sur ProduitDAO (DAO pour Data Access Object) pour implémenter sa persistance en base de données (pour cette adaptation, fermons les yeux sur le fait que notre objet métier dépende d'un objet technique, ce qui n'est tout de même pas très élégant...). A son tour, ProduitDAO utilise une Connection JDBC intanciée à partir d'une DataSource.

Voici le code de la classe Produit. Notez qu'elle ne se soucie pas de l'instanciation de son DAO.

```
01. package org.dotnetguru.bo;
02.
03. import org.dotnetguru.data;
04.
05. public class Produit {
```

```
06.     private ProductDAO dao;
07.     private int stock;
08.
09.     // Accesseur utilise par Spring
10.     public void setDAO(ProductDAO dao) {
11.        this.dao = dao;
12.     }
13.
14.     // Accesseur utilise par Spring
15.     public void setStock(int stock) {
16.        this.stock = stock;
17.     }
18.
19.     // Methode metier
20.     public void acheter() {
21.        dao.charger();
22.        // Comportement metier
23.        stock--;
24.        // ...
25.
26.              dao.sauvegarder();
27.     }
28. }
```

Produit.java

Puis le code de la classe ProduitDAO. A nouveau, notez qu'elle n'instancie pas elle-même la DataSource, et qu'elle n'utilise pas non plus JNDI pour l'initialiser.

```
01. package org.dotnetguru.data;
02.
03. import java.sql.Connection;
04. import javax.sql.DataSource;
05.
06. public class ProduitDAO {
07.     private DataSource dngDataSource;
08.     private int id;
09.
10.     // Accesseur utilise par Spring
```

```
11.     public void setDngDataSource(DataSource dngDataSource) {
12.         this.dngDataSource = dngDataSource;
13.     }
14.
15.     // Methode metier
16.     public void recharger() {
17.         Connection cnx = ds.getConnection();
18.
19.         // Utilisation de la connexion
20.     }
21. }
```

ProduitDAO.java

Enfin, voici comment décrire en XML l'initialisation des propriétés (ou la résolution des dépendances) que va implémenter le conteneur de Spring:

```
01. <beans>
02.     <bean id="dngDataSource"
03.         class="org.apache.commons.dbcp.BasicDataSource"
04.         destroy-method="close">
05.         <property name="driverClassName">
06.             <value>com.mysql.jdbc.Driver</value>
07.         </property>
08.         <property name="url">
09.             <value>jdbc:mysql://localhost/PetShopDNG</value>
10.         </property>
11.         <property name="username">
12.             <value>root</value>
13.         </property>
14.     </bean>
15.
16.     <bean id="productDAO"
17.         class="org.dotnetguru.data.ProductDAO">
18.         <property name="dataSource">
19.             <ref bean="dngDataSource"/>
20.         </property>
21.     </bean>
22.
```

```
23.     <bean id="product"
24.         class="org.dotnetguru.bo.Product">
25.         <property name="dao">
26.             <ref bean="productDAO"/>
27.         </property>
28.         <property name="stock">
29.             <value>50</value>
30.         </property>
31.     </bean>
32. </beans>
```

Spring-config.xml

Effectivement, disposer d'un conteneur IOC pour résoudre les dépendances entre classes, en particulier entre classes de niveaux d'abstraction différents, s'avère très pratique. Le code se simplifie mais le prix à payer est bien sûr de déplacer cette complexité dans les fichiers de configuration du framework.

Maintenant que nous avons une meilleure idée du mécanisme d'IOC, essayons d'en évaluer les problèmes résiduels:

- **Redondance:** les attributs positionnés par le framework (ou les accesseurs aux attributs), qui ont pourtant été déclarés dans le code des classes, doivent être mentionnés à nouveau dans les fichiers de configuration du framework. Cette redondance augmente le coût du développement et de la maintenance évolutive et ne favorise pas l'agilité dudit développement.
- **Robustesse:** les frameworks d'IOC fonctionnent à l'exécution; il faut donc lancer l'application pour en tester la robustesse, et il vaut mieux disposer de tests unitaires assez agressifs pour s'assurer qu'il n'existe aucune erreur de saisie entre le code de l'application et le paramétrage du framework. Aucune vérification de ce type n'est possible au moment de la compilation, ce qui peut s'avérer pénalisant.

En complément de ces nouveaux outils, il faudra donc disposer de logiciels statiques qui permettront de vérifier automatiquement l'adéquation entre code et paramétrage. Mieux, il faudrait qu'à partir du code, on puisse générer automatiquement les documents de

paramétrage "type", quitte à devoir les retoucher par la suite (dans le cas contraire, la saisie et la maintenance de ces fichiers sera certainement fastidieuse).

4.2.3. Enrichissement de code par les méta-données

L'idée de générer du code n'est vraiment pas nouvelle. Mais celle de placer dans le code des indications supplémentaires qui permettront de déduire le reste du code (souvent technique) est plutôt ingénieuse.

C'est l'approche qu'a suivie la **XDoclet** en Java ou les **Attributes** en .NET: il suffit d'ajouter des commentaires spéciaux (XDoclet) ou des méta-données (Attributes .NET) dans le code pour que le comportement de ce code soit "enrichi". Cet enrichissement peut se faire statiquement, donc par génération de code ou dynamiquement si le framework d'exécution comprend les méta-données qui ont été placées sur les éléments de code exécutés.

Ce mécanisme est intéressant car il concentre au même endroit, par exemple dans une classe, tout ce qui tourne autour de cette classe: persistance en XML ou en base de données, distribution, comportement transactionnel, sécurité, recyclage... On évite ainsi une redondance néfaste à l'évolutivité et à la productivité. Cette même approche peut également être utilisée pour générer les fichiers de paramétrage propriétaires que requièrent certains frameworks (en Java: Hibernate ou OJB, Struts...).

Mais d'un autre coté, **cette démarche trouve ses limites** dès que l'on souhaite attribuer à une classe un comportement différent selon le contexte d'exécution. Par exemple, elle ne tolère pas qu'une classe puisse être mappée en différents formats de documents XML (en différents XMLSchemas) ni vers plusieurs structures de bases de données, ce qui peut pourtant être intéressant lors d'une phase de migration de données, ou tout simplement pour les applications attaquant plusieurs bases à l'exécution. L'approche par méta-données semble donc n'être adaptée qu'à un seul contexte d'exécution.

Laissons agir nos réflexes et évaluons la viabilité de cette approche par

rapport aux principes de base de la conception objet. Nous nous apercevons bien vite que **deux de nos principes sont transgressés: l'inversion des dépendances et l'ouvert-fermé**. En effet, si nous plaçons les indications techniques dans les composants métier ou applicatif, nous rendons ces derniers dépendants des frameworks techniques sous-jacents. Si plus tard nous choisissons de changer de framework, il faudra modifier le code source des classes (et relancer la génération de code ou du moins re-compiler pour les frameworks évaluant les méta-données à l'exécution).

D'autre part, les méta-données rendent très rapidement le code **illisible**. Certaines classes finissent par avoir plus de méta-données techniques que de données ou de comportement métier ou applicatif. Dès lors, faut-il voir ces classes comme des classes métier ou comme un substitut aux fichiers de configuration des frameworks techniques?

Corollaire du point précédent, l'usage de ces méta-données est délicat dans le cadre d'un développement en équipe. En effet, le même code source contient des informations qui doivent être maintenues par les développeurs métier, par les responsables du mapping Objet/Relationnel ou du mapping Objet/XML, et par les experts de tous les autres services techniques. Donc nous avons deux options pour le développement d'une classe:

- soit la même personne doit être responsable de tous ces aspects fonctionnels et techniques à la fois,
- soit les membres du projet devront manipuler cette classe à leur tour, chacun ajoutant un élément technique ou fonctionnel.

La première alternative ne permet pas de séparer les responsabilités sur un projet (ni en termes de livrables, ni en termes d'organisation de projet), la seconde risque de nuire fortement à la parallélisation des tâches: les activités des différents membres d'une équipe seraient beaucoup trop couplées.

En conclusion, cette méthode de développement convient aux petites équipes où chacun est responsable de ses classes, dans des cas simples, en particulier non contextuels. Elle n'est malheureusement pas généralisable.

4.2.4. Model Driven Architecture (MDA)

Au lieu de privilégier le code et les développeurs (de composants métier ou de frameworks techniques), certains préfèrent centraliser l'information dans des diagrammes, exprimés en UML ou en toute autre notation graphique. Le gros avantage d'un diagramme est d'offrir une vision d'ensemble, de permettre de prendre du recul par rapport à l'activité de codage dans laquelle on ne voit en un coup d'œil qu'une parcelle très réduite de l'application.

Diagrammes statiques et diagrammes dynamiques permettent de décrire complètement un logiciel, ce qui fait qu'en théorie on pourrait tout indiquer dans les diagrammes et générer 100% du code des applications. En pratique, cela n'est pas encore faisable dans le cas général. Pire, nous pouvons nous demander si c'est réellement souhaitable; en effet, autant les diagrammes statiques sont très concis, autant il est souvent bien plus long de tracer un diagramme dynamique que de saisir le code correspondant.

Mais revenons au rapport qui lie nos applications aux frameworks techniques. Rien n'interdit de compléter la notation graphique des modèles (grâce aux stéréotypes ou aux tagged values UML par exemple) de manière à augmenter leur pouvoir d'expression. Une classe pourrait ainsi devenir persistante, ses méthodes transactionnelles et sécurisées, ses attributs exportables en base ou en XML... Il suffit pour cela de générer le code et les documents de configuration nécessaires à partir du modèle. Ce qui s'avère techniquement très simple depuis la généralisation de XMI (un dialecte d'XML permettant de représenter les diagrammes UML).

Figure 17. Approche MDA sur un outillage neutre

De nombreux éditeurs de logiciels ou initiatives OpenSource suivent cette vague technologique et offrent généralement des solutions mieux "packagées" que de passer par XMI et d'avoir à créer ses propres transformateurs. En particulier, certains outils commencent à générer, à partir du modèle objet, la couche d'accès aux données, les scripts de création de la base de données relationnelle, ainsi qu'une couche de présentation assez générique. Cela fonctionne très bien pour les besoins

de consultation de listes d'objets, de recherche, de pagination, de modification simple.

La question récurrente sur cette démarche est: **allons-nous parvenir à tout générer automatiquement?**. A nouveau, tout dépend de ce que nous allons accepter de décrire sous forme de diagrammes: si nous ne décrivons quasiment que la structure statique de notre modèle objet, son diagramme de classes en UML, le corps des méthodes et la cinématique de notre application ne pourront bien évidemment pas être produits. Ce n'est qu'en décrivant l'aspect dynamique, ce que permettent les diagrammes de séquence, de collaboration, d'activité ou encore d'états en UML, que nous pourrons espérer obtenir par génération de 60 à 85% du corps d'une application, voire 100% si l'on n'est pas trop regardant sur les finesses du mapping Objet/relationnel ou de la couche de présentation en particulier.

L'approche MDA semble donc tout à fait prometteuse. Essayons maintenant de faire le tour de ses faiblesses. Tout d'abord, parlons de **rétro-conception**: une fois que le code et autres documents auront été générés par la chaîne logicielle MDA, nous souhaiterons certainement apporter quelques modifications à la charte graphique, à l'implémentation de certaines classes ou encore aux réglages des frameworks techniques. Mais une fois ces modifications apportées, il faudrait qu'elles soient réinjectées dans les modèles d'origine, de manière à ne rien perdre lors des générations de code ultérieures. Et c'est bien là le problème: non seulement cela signifie qu'il faut disposer d'un outil capable de relire et de comprendre le code de chaque élément généré (code Java, C#, documents XML spécifiques, ASP.NET, JSP, syntaxes des moteurs de templates...), mais aussi que les diagrammes doivent être suffisamment complets pour porter tous les détails qui se retrouvent dans les fichiers générés. Pour tenter de répondre à ce double problème, MDA propose deux niveaux de modèles: le premier est complètement indépendant des frameworks techniques choisis (on parle alors de **Platform Independant Model** ou **PIM**) et le second inclue les éléments spécifiques à l'environnement technique cible (et s'intitule **Platform Specific Model** ou **PSM**).

Un autre élément à prendre en compte est la lourdeur de certains outils de modélisation actuels, que ce soit à l'usage (certaines opérations n'étant pas très intuitives) ou à l'exécution (en particulier pour les

modeleurs UML écrits en Java qui nécessitent des configurations matérielles assez "musclées"). Cette lourdeur peut avoir un impact très négatif sur l'analyste et le concepteur dont les rôles impliquent une utilisation intensive de ces outils. Heureusement, et "comme d'habitude", nous pouvons miser sur l'amélioration constante des capacités des machines et des systèmes sous-jacents à ces logiciels pour résoudre leur manque d'efficacité...

Plus inquiétant que les éventuels problèmes de performances: l'approche MDA semble pâtir des mêmes travers que celle basée sur les méta-données: elle est **non-contextuelle par défaut** (les stéréotypes techniques associés à une classe ou à une méthode conviennent très bien à certains cas mais risquent de ne pas s'appliquer au cas général), et **elle rassemble sur un même modèle de nombreuses informations fonctionnelles et techniques**. Or les outils de modélisation actuels sont peu adaptés au travail collaboratif. Ceux qui le permettent supposent généralement qu'un élément du modèle ne peut être manipulé que par une personne à la fois; cela reste tolérable quand l'élément en question est une méthode ou un attribut, mais cela devient très gênant quand une classe entière est verrouillée dès qu'une personne y apporte ses modifications, voire quand il s'agit de tout un diagramme et de son contenu! Heureusement, la démarcation PIM / PSM devrait limiter ce problème...

4.2.5. AOP

Sans en dévoiler les principes ni les outils, puisque c'est l'objet du chapitre suivant, disons que l'AOP propose une organisation différente tant pour les éléments de code que les équipes de développement. Elle devrait permettre 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 va nous permettre de mieux respecter l'Ouvert-Fermé et l'Inversion des Dépendances, principes les plus souvent transgressés par les approches que nous avons évaluées précédemment.

Mais pour le moment, tout ceci n'est que promesse. Afin de vous

permettre de juger sur pièce, découvrons ensemble les bases de l'AOP.

Chapitre 5.

Découverte du monde des Aspects

Ce chapitre va jalonner nos premiers pas dans le monde de la Programmation Orientée Aspect. Il explique les principes de base de l'AOP par la pratique, sur un exemple simple et concret, avant de définir plus formellement les concepts et les outils du monde des Aspects.

Puis nous recentrerons le débat sur les tisseurs d'Aspects et nous décrirons un ensemble de critères objectifs qui permettront ensuite d'évaluer plusieurs tisseurs du marché, disponibles sur les plate-formes Java et .NET.

Nous décrirons également le fonctionnement interne d'un tisseur d'Aspects statique, AspectDNG, avec l'objectif de vous donner tous les éléments pour bien délimiter le périmètre fonctionnel d'un tisseur et ainsi comprendre ce que l'on peut attendre d'un tel outil.

5.1. Principes de l'AOP

Tout d'abord, une précision: nous garderons dans ce livre le sigle anglais **AOP** pour signifier Programmation Orientée Aspects, et ce afin de ne pas troubler les lecteurs familiers de CORBA dans lequel **POA** revêt une toute autre signification.

Avant de définir formellement la notion d'aspect et les termes qui en dérivent, essayons tout d'abord de découvrir le besoin auquel les Aspects répondent, ainsi que leur principe de fonctionnement. Reprenons l'exemple de la classe Produit (ici implémentée en C#) et examinons le corps de quelques-unes de ses méthodes:

```
01. namespace Dotnetguru.BLL{
02.
03.     public class Produit {
04.         private const double Reduction = 0.1;
05.         private string m_nom;
06.         private int m_stock;
07.         private double m_prix;
08.         private bool m_enSolde;
09.
10.         public string Nom {
11.             get {
12.                 Console.WriteLine("Lecture du Nom");
13.                 return m_nom;
14.             }
15.             set {
16.                 Console.WriteLine("Ecriture du Nom");
17.                 m_nom = value;
18.             }
19.         }
20.
21.         public int Stock {
22.             get {
23.                 Console.WriteLine("Lecture du Stock");
24.                 return m_stock;
25.             }
```

```
26.         set {
27.             Console.WriteLine("Ecriture du Stock");
28.             m_stock = value;
29.          }
30.       }
31.
32.       public double Prix {
33.          get {
34.             Console.WriteLine("Lecture du Prix");
35.             double prixReel = m_prix;
36.             if (EnSolde){
37.                prixReel *= (1 - Reduction);
38.             }
39.             return prixReel;
40.          }
41.          set {
42.             Console.WriteLine("Ecriture du Prix");
43.             m_prix = value;
44.          }
45.       }
46.
47.       public bool EnSolde {
48.          get {
49.             Console.WriteLine("Lecture de l'etat EnSolde");
50.             return m_enSolde;
51.          }
52.          set {
53.             Console.WriteLine("Ecriture de l'etat EnSolde");
54.             m_enSolde = value;
55.          }
56.       }
57.
58.       public void Acheter(Client c) {
59.          Console.WriteLine("Achat du Produit");
60.          Stock--;
61.          c.Debiter(Prix);
62.       }
63.    }
```

```
64. }
```
Produit.cs

Certes, il s'agit d'une version plutôt simpliste d'un Produit. De plus, nous aurions pu utiliser un framework de traces au lieu d'imprimer les messages directement sur la sortie standard. Mais ceci mis à part, le code de cette classe n'est pas choquant en soi. Et c'est bien là le problème: nous acceptons de dupliquer le code de trace **Console.WriteLine()** bien qu'il soit très répétitif! Pire, comme nous allons utiliser la technique du "copier-coller" pour dupliquer ce code, nous risquons d'oublier de modifier le message affiché, et par là même d'induire en erreur ceux qui reliront les traces.

Heureusement, le Framework .NET nous offre de bons outils pour simplifier et rationaliser ce code. En particulier, la classe **System.Diagnostics.StackFrame** permet de savoir quelle méthode vient d'être appelée. Nous nous proposons donc de créer une classe utilitaire nommée **Traces**, dans laquelle les méthodes (Info(), Warn()...) permettront d'afficher au moins le nom de la méthode courante:

```
01. namespace Dotnetguru.Util {
02.     using System;
03.     using System.Diagnostics;
04.
05.         public class Traces{
06.         [Conditional("TRACE")]
07.         public static void Info(){
08.             StackFrame frame = new StackFrame(1, true);
09.             string methodName = frame.GetMethod().Name;
10.             if (methodName.StartsWith("get_")) {
11.                 Console.WriteLine("Lecture de " +
12.                     methodName.Substring("get_".Length));
13.             }
14.             else if (methodName.StartsWith("set_")) {
15.                 Console.WriteLine("Ecriture de " +
16.                     methodName.Substring("set_".Length));
17.             }
18.             else {
19.                 Console.WriteLine("Invocation de " + methodName);
```

```
20.         }
21.       }
22.    }
23. }
```
Traces.cs

Vous aurez noté au passage l'Attribute **Conditional** qui permet au compilateur C# d'invoquer ou non la méthode Info() selon la présence ou l'absence de la variable de précompilation "TRACE". Comme nous l'avons mentionné précédemment, la programmation par méta-données est une technique complémentaire pour découpler les éléments techniques du code applicatif; c'est en tous cas celle que privilégie Microsoft dans le framework .NET.

Grâce à l'externalisation des traces, la classe Produit se simplifie un peu; du moins, nous ne pouvons plus nous tromper sur le contenu des messages d'erreur affichés:

```
01. namespace Dotnetguru.BLL{
02.     using Dotnetguru.Util;
03.
04.     public class Produit {
05.         private const double Reduction = 0.1;
06.         private string m_nom;
07.         private int m_stock;
08.         private double m_prix;
09.         private bool m_enSolde;
10.
11.         public string Nom {
12.             get {
13.                 Traces.Info();
14.                return m_nom;
15.             }
16.             set {
17.                 Traces.Info();
18.                m_nom = value;
19.             }
20.         }
21.
```

```
22.         public int Stock {
23.             get {
24.                 Traces.Info();
25.                 return m_stock;
26.              }
27.             set {
28.                 Traces.Info();
29.                 m_stock = value;
30.              }
31.         }
32.
33.         public double Prix {
34.             get {
35.                 Traces.Info();
36.                 double prixReel = m_prix;
37.                 if (EnSolde){
38.                     prixReel *= (1 - Reduction);
39.                 }
40.                 return prixReel;
41.              }
42.             set {
43.                 Traces.Info();
44.                 m_prix = value;
45.             }
46.         }
47.
48.         public bool EnSolde {
49.             get {
50.                 Traces.Info();
51.                 return m_enSolde;
52.              }
53.             set {
54.                 Traces.Info();
55.                 m_enSolde = value;
56.             }
57.         }
58.
59.         public void Acheter(Client c) {
```

```
60.             Traces.Info();
61.             Stock--;
62.             c.Debiter(Prix);
63.         }
64.     }
65. }
```
ProduitBis.cs

En exécutant une classe utilisatrice du Produit, nous pouvons donc obtenir la sortie suivante:

Figure 18. Utilisation de Produit sans Aspects

Mais malgré tout, nous ne pouvons pas nous débarrasser complètement de la dépendance entre les classes Produit et Traces. Or Produit représente bien un objet métier et Traces un utilitaire technique. L'importation de l'espace de noms **Util** dans une classe **Business** (ou "BLL") nous montre bien que nous ne respectons pas le principe d'inversion des dépendances.

Dans une telle situation, bien que fréquente, les outils offerts par l'approche Objet s'avouent vaincus: aucun d'entre eux ne permet d'invoquer automatiquement **Traces.Info()** au début de chaque méthode de la classe **Produit**. Eh bien c'est exactement le type de mécanisme que propose la Programmation Orientée Aspect: ajouter du code aux endroits souhaités dans une classe existante, sans avoir à éditer cette classe manuellement bien entendu.

Dans notre exemple, il faudrait définir **un Aspect qui tisse l'invocation de Traces.Info() en début de chaque méthode de la classe Produit**. Si nous suivons cette voie:

- La classe Produit ne dépend plus de Trace: non seulement elle n'importe plus l'espace de nommage Dotnetguru.Util, mais de plus elle n'invoque plus Trace.Info().
- La classe Trace ne change pas.
- Un Aspect apparaît pour faire le lien entre Produit et Trace. Il va décrire à quels endroits il faut tisser du code (ici: à chaque début de corps de méthode de la classe Produit) et quel est le comportement

de ce code (ici: invoquer la méthode Trace.Info()).

Avec AspectDNG, un Aspect se composera d'une classe (C#, VB.NET ou autre) et d'un fichier XML décrivant le tissage. Sans entrer dans les détails de l'outil pour le moment, voici un avant-goût. Tout d'abord la classe:

```
01. namespace Dotnetguru.Aspects {
02.     public class AspectTrace {
03.         public void MethodeATisser(){
04.             Dotnetguru.Util.Traces.Info();
05.         }
06.     }
07. }
```

AspectTrace.cs

Comme vous pouvez le constater, elle ne présente aucune complexité. Passons au descripteur de tissage:

```
01. <Advice>
02.     <InlineAtStart
03.         targetCondition="self::Method[../@name='Produit']"
04.         aspectXPath="//Method[@name='MethodeATisser']"/>
05. </Advice>
```

Advice-Traces.cs

Dans ce document, c'est le langage XPath qui permet de décrire **où tisser** et **quoi tisser**. Et sans surprise, la classe utilisatrice de Produit donne exactement le même résultat que précédemment:

Figure 19. Utilisation de Produit avec Aspects

Maintenant que nous savons de quoi nous parlons, nous pouvons avancer quelques définitions pour mettre en mots les concepts précédents.

5.2. Définitions

5.2.1. Définition formelle de l'AOP

L'AOP est une technique de conception et de programmation qui vient en complément de l'approche Orientée Objet ou procédurale. Elle permet de factoriser (et donc 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: sans code procédural ou objet, la notion d'Aspect perd tout son sens. Mais inversement, on pourrait dire que les programmations Orientée Objet ou procédurale ne sont pas complètes puisque inaptes à mettre en facteur ou à bien séparer certaines responsabilités des éléments logiciels.

Remarque: nous ne traiterons dans ce livre que d'AOP associée à la conception et à la programmation Orientées Objet.

5.2.2. Définition informelle de l'AOP

L'AOP est une utilisation particulière de l'instrumentation de code qui s'attache essentiellement à résoudre des problèmes de conception. L'AOP ne vise donc pas, a priori, à satisfaire les besoins de génération automatique de code ou de documentation.

5.2.3. Définition des mots-clé

Code cible (ou socle ou encore 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 donc aucune dépendance), elles ne se "doutent" pas qu'elles vont constituer la cible d'un tissage.

Aspect

Il s'agit du code que l'on souhaite tisser. Selon les tisseurs, un aspect peut être une simple classe (cf. AspectDNG) ou constituer un élément d'un langage spécifique (cf. AspectJ).

Zone de greffe

Ce sont les "endroits" dans le code cible dans lesquels aura lieu le tissage. Par exemple: "au début du corps de la méthode hello()" est une zone de greffe.

Tissage (ou greffe ou encore injection de code)

Processus qui consiste à ajouter (tisser, greffer, injecter) le code des Aspects sur les zones de greffe du code cible.

Insertion (ou introduction ou injection)

Tissage spécial consistant à rajouter de nouvelles méthodes ou de nouveaux attributs à une classe cible, ou encore à rendre disponibles de nouveaux types (interfaces, classes, enums...) dans un projet cible (i.e. dans un module .NET ou une archive Java).

Langage de tissage

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

Tisseur

Programme exécutable ou framework dynamique qui procède au tissage. Le tisseur peut être invoqué:

- statiquement avant la compilation du code cible (le tisseur travaille sur le code source),
- statiquement pendant la compilation du code cible (le tisseur intègre un compilateur du langage de programmation),

- statiquement après la compilation du code cible (le tisseur travaille sur le code précompilé: bytecode, CIL),
- dynamiquement avant exécution du code (au moment de la compilation Just-In-Time),
- dynamiquement pendant l'exécution du code (par la technique d'interception d'invocation de méthode, utilisée intensivement par les frameworks d'inversion de contrôle).

Puisque les notions de classe, de méthode, d'attribut... peuvent apparaître à la fois dans le contexte du code cible et des aspects, nous lèverons l'ambiguïté en parlant par exemple de **méthode cible** ou d'**attribut d'Aspect**.

Pour reformuler ce que nous avons fait dans la section précédente de découverte du principe des Aspects, disons que **nous avons tissé la méthode d'aspect AspectTrace.MethodeATisser() au début de toutes les méthodes de la classe cible Produit**.

5.3. Tisseurs d'Aspects

Un tisseur d'Aspect (ou **Aspect Weaver** en anglais) est donc l'outil responsable de la greffe des Aspects sur le code cible. Il existe aujourd'hui de nombreux projets de développement de tisseurs qui varient par leur approche, par le(s) langage(s) qu'ils visent ou la plate-forme sur laquelle ils s'exécutent. Un bon point de départ pour se faire une idée sur les tisseurs existants ou en cours de développement est le site *http://www.aosd.net*.

Comme vous pourrez le constater, il en existe vraiment beaucoup. Et ce n'est probablement pas fini: comme l'AOP semble devenir à la mode, il est probable que de nouveaux projets démarrent dans les prochains mois! Nous n'allons pas nous en plaindre: que les meilleurs gagnent, et que les utilisateurs aient une belle palette d'outils matures le plus vite possible!

Mais par conséquent, sur vos projets ou globalement dans votre entreprise, vous aurez probablement à faire un choix. Or, contrairement à ce que l'on pourrait croire, tous ces tisseurs sont loin d'être équivalents. Sans même parler de leur maturité, nous vous proposons ici un certain nombre de critères techniques objectifs qui vous permettront de mieux cerner un tisseur d'Aspects et de faire un choix argumenté en fonction de vos contraintes de projet.

Le tisseur est-il mono- ou multi-langage cible?

Cette question est particulièrement importante dans le monde .NET où le multi-langage est chose courante (C#, VB.NET, Managed C++, Eiffel Envison, J#...). Dès lors, choisir un tisseur mono-langage (par exemple: C#) est très contraignant car cela obligerait tous les développeurs de composants métier comme de frameworks techniques à faire le choix du même langage de programmation. Vous me direz, peu d'entreprises ont choisi plusieurs langages .NET... A supposer que cela soit vrai, n'oublions pas toutes les bibliothèques de classes ou de composants réutilisables qui peuvent être développées par des prestataires de service ou des éditeurs de logiciel: ceux-ci risquent de ne pas avoir fait le même choix de langage.

Dans le monde Java, bien sûr, ce critère n'existe pas.

Le tissage est-il statique ou dynamique?

On dit qu'un tisseur est statique lorsqu'il greffe réellement du code avant ou après la compilation. Avant la compilation, nous avons affaire à un tisseur qui modifie directement le code des classes cibles; après la compilation, le tisseur doit faire son travail sur le bytecode Java ou le CIL .NET.

Les tisseurs dynamiques quant à eux peuvent s'exécuter au chargement des classes en mémoire lors de la compilation Just-In-Time ou à l'exécution en interceptant les invocations de méthodes.

Les avis des experts sont partagés sur ce sujet. D'aucuns préféreront la robustesse et la performance des tisseurs statiques, d'autres la souplesse des tisseurs dynamiques. En réalité, tout dépend de votre besoin: si vous souhaitez pouvoir tisser un aspect à la volée, en cours d'exécution de vos applications, seul un tisseur dynamique pourra vous satisfaire. Si vos applications sont déjà un peu juste en termes de performances, ou si vous avez besoin de "tisser très fin" (avant ou après l'accès aux attributs, qu'il existe ou non des accesseurs pour ces derniers), alors il vous faudra vous orienter vers les tisseurs statiques. Nous reviendrons plus en détail sur ce point après avoir pratiqué quelques tissages concrets.

Quelle est la puissance d'expression du langage de tissage?

Le langage de tissage est un bon point de départ pour découvrir les fonctionnalités implémentées par un tisseur. Un tisseur "sérieux" doit au minimum être capable de greffer du code:

- au début du corps des méthodes,
- avant l'invocation d'une méthode,
- avant qu'une méthode ne renvoie la main à son appelant,
- autour d'une méthode (éventuellement en escamotant la méthode originale),
- avant ou après l'accès à un attribut.

Outre les zones sur lesquelles une greffe peut avoir lieu, il est

important que le langage de tissage soit suffisamment puissant pour éviter les redondances. Par exemple, il faudrait pouvoir exprimer que l'on souhaite tisser du code *"sur tous les débuts de méthodes de toutes les classes d'un package"* ou *"autour de tous les getters et setters d'une classe et de ses filles"*.

En effet, le langage de tissage fait le lien ou la jointure entre le code de base et celui des Aspects et doit donc permettre le plus de tissages possibles. Voire même certains dont nous ne connaissons pas encore l'usage.

Dans quel langage doivent être développés les Aspects?

Pour le développeur d'un tisseur, il est tentant de définir un nouveau langage doté de nouveaux mots-clé plus adaptés aux concepts de l'AOP que ne le sont les classes, attributs, méthodes et autres outils déjà disponibles dans les langages objet. De plus, cela permettra certainement aux utilisateurs du tisseur de ne pas faire l'amalgame entre classes de base et Aspects.

Mais la décision de créer un langage propriétaire, spécifique à un tisseur, est à double tranchant. En contre partie, les utilisateurs devront encore apprendre un nouveau langage et disposer d'outils complémentaires pour développer et maintenir leurs Aspects. Plus grave encore, leurs Aspects seront complètement dépendants du tisseur en question. Il sera dès lors difficile voire coûteux (en adaptations à apporter au code des Aspects) de réutiliser ces Aspects sur d'autres projets qui feraient le choix d'un autre tisseur.

Dans la première partie de ce livre, nous avons passé en revue les principes de base de la conception Objet, nous avons insisté sur les risques d'un trop fort couplage et sur l'importance de respecter l'ouvert-fermé et l'inversion des dépendances. Si nous voulons être cohérents, il nous faut donc condamner les tisseurs d'Aspects qui ne respectent pas ces principes et qui rendent leurs utilisateurs captifs.

Plus généralement, existe-t-il une adhérence quelconque entre les Aspects et le tisseur utilisé?

Sans revenir sur le langage de développement des Aspects, il faut se demander si nos Aspects dépendent d'une interface de programmation

spécifique au tisseur utilisé. Typiquement, certains tisseurs proposent de manipuler des objets techniques afin de pouvoir prendre une décision contextuelle dans le corps de l'aspect (un aspect peut ainsi se dire: "si j'ai été tissé dans telle méthode, alors je me comporte de telle manière, sinon autrement...").

S'ils offrent cette possibilité, c'est généralement à travers une API propriétaire. Mais si l'on souhaite rendre nos aspects portables sur tous les tisseurs du marché, il faudrait soit ne pas utiliser ce type de mécanisme, soit reposer sur une API commune à tous les tisseurs d'Aspects; or un consortium œuvre dans ce sens et semble recueillir les suffrages des principaux groupes de développements de tisseurs du monde Java: l'AOP Alliance (*http://aopalliance.sourceforge.net/*). L'avenir nous dira si son API sera supportée par tous les tisseurs Java du marché; il faudra également qu'une API standard soit supportée (celle-ci de l'AOP Alliance ou une API équivalente) par les tisseurs du monde .NET.

Sur quelle machine virtuelle fonctionne le tisseur?

Cette question prend tout son sens lorsque le tisseur fonctionne au chargement des classes car cela nécessite d'être très proche de la machine virtuelle utilisée. Les tisseurs de ce type offrent en général plusieurs implémentations de leur couche basse, de manière à être portables sur plusieurs environnements. Il faut donc vérifier que votre machine virtuelle cible en fasse partie.

Combien de temps prend le tissage?

Le processus de tissage peut intervenir à plusieurs moment, comme nous l'avons vu dans la définition d'un tisseur. Dans tous les cas, il vaudrait mieux que le temps d'exécution de la phase de tissage ne soit pas trop long.

Un tisseur statique, par exemple, sera intégré dans la phase de compilation (certains tisseurs comme AspectJ prennent même la place de cette phase de compilation) et sera donc déclenché assez souvent. Il ne faudrait pas que la productivité des développeurs soit limitée par un tissage trop long. Difficile de chiffrer précisément le temps que devrait prendre un tissage (puisqu'il dépend du code de base, du nombre

d'aspects à tisser, du nombre de zones de greffe visées et bien entendu de la puissance du matériel sur lequel s'effectue le tissage) mais un ordre de grandeur de quelques secondes reste acceptable pour des projets de taille moyenne. En fait, il faudrait que le temps de compilation et de tissage (notons-le TCT pour Temps de Compilation et de Tissage) soit du même ordre de grandeur que le temps de compilation du même code dans lequel les instructions ou les classes tissées auraient été développées à la main (notons-le TC pour Temps de Compilation). Si TCT = TC, le tisseur est très efficace. Si TCT = 5 * TC, le tisseur est un peu lent. Et si TCT > 10 * TC, le tisseur est très lent, voire trop lent selon de la taille de vos projets.

Un tisseur qui ne s'exécutera qu'au chargement des classes (généralement à la compilation à la volée des classes en Java ou en .NET) doit lui aussi être le plus rapide possible, sinon le temps de chargement de nos applications va s'allonger. Ceci peut s'avérer gênant, surtout pour les classes chargées à la demande, en cours d'exécution du programme: des contraintes fortes sur le temps de réponse du système pourraient ne plus être respectées. Il n'est pas rare de voir le temps de chargement des classes à tisser s'allonger d'un facteur 5.

Enfin, **les tisseurs dynamiques** ont un temps d'exécution qui est probablement encore plus critique que les précédents puisqu'ils vont être sollicités lors d'un certain nombre d'invocations de méthodes (le nombre varie avec la stratégie de tissage). Selon leur implémentation, il se peut que le surcoût induit à l'exécution ne soit pas le même lors de la première invocation d'une méthode tissée que lors des invocations ultérieures. Mais quoi qu'il en soit, il faut que vous mesuriez le surcoût "moyen" d'invocation des méthodes si vous utilisez de tels tisseurs et que vous voyiez si les performances globales restent compatibles avec vos contraintes d'exécution. Si vos aspects portent sur peu de méthodes cibles, ou si les performances globales avant tissage de vos applications sont bien en-deçà de leurs maxima imposés, alors un tisseur dynamique peut parfaitement faire l'affaire. Dans le cas contraire, évaluez les tisseurs statiques (qu'ils fonctionnent au cours de la phase de développement ou au chargement des classes): le code de l'application tissée sera généralement bien plus performant, ce qui est normal, mais vérifiez également que vos autres critères d'évaluation

restent respectés (couplage faible, dynamisme suffisant...).

Quelle est le niveau d'intégration du tisseur dans un environnement de développement?

Certains tisseurs sont dotés de plug-in adaptés aux principaux environnements de développement. L'intégration permet non seulement de simplifier le démarrage de projets contenant des Aspects, mais dans certains cas elle va jusqu'à nous montrer en temps réel sur quelles zones sont greffés les Aspects du projet! Voici par exemple une copie d'écran du plug-in AJDT qui intègre parfaitement le tisseur AspectJ à l'environnement Eclipse (notez les petites icônes caractéristiques dans la marge gauche).

Figure 20. Le plug-in AJDT dans Eclipse

Est-il possible de tester unitairement les Aspects?

Nous reprenons aujourd'hui l'habitude de tester unitairement tous les éléments cruciaux de nos applications. Les Aspects plus encore que les classes cibles nécessitent d'être testés car ils risquent d'être tissés sur plusieurs zones de greffe; une erreur qui se glisse dans le code d'un Aspect risque donc d'introduire de nombreux dysfonctionnement dans l'application finale.

Mais encore faut-il que les tisseurs (ou plutôt les langages de développement des Aspects) permettent d'effectuer des tests unitaires sur les Aspects. Pour cela, faut-il être capable d'instancier un aspect, d'invoquer la plupart de ses méthodes? Ou devons-nous passer par un "Mock Object", une souche neutre sur laquelle nous grefferions nos aspects avant de pouvoir les tester? Nous reviendrons sur la mise en œuvre de tests sur les Aspects dans un chapitre ultérieur, mais gardez ce besoin à l'esprit et informez-vous, dans la documentation du tisseur que vous évaluez, de la possibilité de réaliser ce type de tests indispensable à l'assurance de qualité de nos logiciels.

Limitations potentielles: le tisseur est-il un filtre des langages de programmation?

Ou encore, est-ce que le résultat du tissage peut être aussi riche que les classes cibles?

Cette question est particulièrement importante pour les tisseurs statiques qui fonctionnent sur le bytecode des classes cibles. Dans ce contexte, nous pourrions la reformuler de la manière suivante: le tisseur est-il capable de re-générer toute bibliothèque de classes (assembly .NET ou tout fichier JAR Java) à l'identique? En effet, comme la lecture et l'écriture du bytecode Java ou du CIL .NET reposent sur des bibliothèques non standard, il n'est pas garanti que tous les éléments des langages intermédiaires soient bel et bien pris en compte. En particulier, la notion de méta-donnée risque de ne pas être prise en charge par certains tisseurs. C'est très grave car cela signifierait par exemple qu'une classe C# définie comme étant Sérialisable ne le serait plus après tissage!

Limitations potentielles: le tisseur permet-il l'insertion de code?

L'insertion est le mécanisme qui permet par exemple d'ajouter de nouvelles classes à une application ou de nouvelles méthodes et de nouveaux attributs à une classe. Elle peut s'avérer très utile pour implémenter certains Design Patterns ou pour rendre une classe automatiquement persistante par exemple.

5.4. Évaluons quelques tisseurs

Nous n'avons pas l'ambition de dresser ici une liste exhaustive ni des tisseurs du marché (d'avance, donc, pardon aux oubliés), ni de toutes les fonctionnalités des tisseurs que nous avons retenus. Essayons tout de même d'évaluer les principaux tisseurs grâce aux critères présentés à la section précédente.

5.4.1. Tisseurs Java

Commençons par le monde Java qui s'est montré précurseur dans le domaine, surtout grâce au projet qui a tout déclenché: AspectJ.

AspectJ

AspectJ est un tisseur statique qui peut opérer sur le bytecode Java, mais il accepte également de travailler directement sur du code source. Leader des tisseurs Java, il bénéficie d'une stabilité, d'une maturité et d'une image de qualité 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 fait maintenant partie de la communauté Open Source Eclipse.

Le temps de tissage de cet outil est court et se réduit encore dans les dernières versions. Son langage de tissage est très puissant et AspectJ ne filtre aucune possibilité du langage Java (même les méta-données devraient être intégrées dès que Java 1.5 sera livré en version finale par Sun Microsystems). Le mécanisme d'introduction est supporté quasiment sous toutes ses formes. Enfin, le résultat du tissage est du bytecode Java complètement standard et ne nécessite que la présence d'une (toute petite) bibliothèque de classes.

La seule véritable critique que l'on puisse faire à ce bel outil est d'utiliser un langage propriétaire pour décrire ses Aspects. Certes, celui-ci est proche de Java, mais cela signifie que nous devons être certain de faire le bon choix en utilisant AspectJ, sans quoi le portage des Aspects pourra s'avérer coûteux.

AspectWerkz

AspectWerks est 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 lui-aussi OpenSource et bénéficie du sponsoring de la société Bea Systems.

Contrairement à AspectJ, AspectWerkz permet de développer les Aspects en Java standard. Et pour 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 (pour simplifier la tâche au développeur dans le cas où un aspect n'est utilisé que sur un projet). Le temps de tissage, quant à lui, est court.

Les performances des applications dépendent bien entendu du choix de la méthode de tissage: le tissage statique donne avec AspectWerkz des temps d'invocation de méthodes environ deux fois plus rapides que le tissage dynamique. A noter que dans les deux cas, les performances de l'application tissée sont moins bonnes que celles obtenues avec AspectJ.

JAC

JAC est un tisseur dynamique dont les aspects sont définis en 100% Java; comme AspectWerkz, il ne nécessite l'usage d'aucun langage propriétaire. Certes, le code Java des Aspects dépend encore de classes et d'interfaces de JAC, ce qui pourrait rendre moins immédiat un portage sur d'autres tisseurs, mais comme JAC est un membre actif de l'AOP Alliance, il en implémente l'API et limite ainsi l'adhérence de nos Aspects.

JAC est un outil OpenSource dont le développement est dirigé par Renaud Pawlak et qui a aujourd'hui rejoint le consortium ObjectWeb. Son support actif est assuré par la société Aopsys, également créée par Renaud Pawlak.

Non seulement 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 nos applications doivent parfois changer de stratégie d'implémentation. Par exemple, nous pourrions parfaitement imaginer un mécanisme de cache d'objets en mémoire qui serait implémenté sous la forme d'un Aspect; si l'occupation mémoire devenait critique, notre application pourrait prendre la décision de se passer temporairement de ce cache en procédant à "l'ablation de l'Aspect de cache".

La contrepartie de cette souplesse se paie bien sûr 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.

Par contre, il est très important de souligner cette initiative, 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, JAC nous offre en tout 19 Aspects de qualité professionnelle. Un exemple à suivre pour les autres tisseurs du marché.

SpringAOP

SpringAOP est un constituant du framework d'Inversion de Contrôle *Spring*. Également OpenSource et membre de l'AOP Alliance, son développement est mené par Rod Johnson. Fidèle à la philosophie de l'IOC, SpringAOP impose peu de contraintes sur les Aspects qui s'écrivent bien entendu en 100% Java. Son langage de tissage est complètement intégré au langage XML de configuration de Spring et permet plusieurs axes de définition: soit on liste les intercepteurs qui doivent être placés devant un objet, formant ainsi une chaîne de responsabilités, soit on peut utiliser les expressions régulières pour décrire ce sur quoi porte un Aspect.

L'implémentation de SpringAOP repose sur le mécanisme des dynamic proxies ou intercepteur d'invocation de méthode. Ce tisseur souffre donc des mêmes limitations que JAC: l'introduction ou l'interception

d'accès aux attributs est impossible et le surcoût occasionné par l'ajout d'intercepteurs pénalise les performances.

Le critère le plus intéressant de ce framework est sa disponibilité au sein d'un framework plus générique. Il y a donc fort à parier que ceux qui feront le choix de Spring tireront parti de SpringAOP pour tisser leurs Aspects afin de ne pas ajouter à la complexité de l'outillage de leurs projets.

JBossAOP

Comme AspectWerkz, *JBossAOP* est un tisseur dual que l'on peut utiliser pour tisser soit statiquement à la AspectJ, soit dynamiquement à la JAC. Piloté par Bill Burke, le développement de cet outil OpenSource s'inscrit dans la lignée des produits du JBoss Group et sa meilleure publicité est d'être utilisé dans une application de qualité: le serveur d'applications JBoss lui-même. Ce qui ne veut pas dire que les deux produits soient intimement liés: JBossAOP peut très bien s'utiliser comme un simple tisseur d'Aspects, indépendamment du serveur d'applications JBoss.

Dans cet outil, les aspects s'écrivent en 100% Java mais dépendent d'une API propre à JBossAOP. Apparemment, au moment où nous écrivons ces lignes, il ne semble pas y avoir de rapprochement prévu entre JBossAOP et l'AOPAlliance, ce qui aurait permis de limiter le couplage entre nos aspects et le tisseur.

Ceci mis à part, le langage de tissage est très puissant et s'exprime dans un fichier de configuration XML. Il permet de tirer profit soit des zones de greffe banalisées offertes par le langage Java (début et fin de méthodes, lecture et écriture d'attributs...), soit des annotations que l'on pourrait placer dans le code cible. Ce dernier mécanisme se base sur les méta-données introduites par Java 1.5 mais peut également se contenter de commentaires spéciaux lorsqu'on utilise encore Java 1.4.

La version statique du tisseur JBossAOP a un temps de tissage court et son résultat offre des performances du même ordre de grandeur que AspectJ (sur les tests que nous avons pu mener jusqu'ici). Mais contrairement à ce dernier dont le résultat ne dépend que d'une toute petite bibliothèque de classes (35 Ko), le code produit par JBossAOP nécessite de placer sur le classpath de l'application un certain nombre

de bibliothèques qui pèsent 1,5 Mo en tout. Cette contrainte n'en est pas une dans le contexte d'un serveur (Web, Ejb) ou d'une application déjà volumineuse, mais elle risque d'être bloquante dans le contexte d'une application embarquée ou s'exécutant sur un équipement à ressources limitées.

Et les autres...

A travers ces évaluations sommaires, nous espérons que vous aurez acquis les réflexes nécessaires pour suivre vous-même l'actualité du monde des tisseurs Java. Déjà plusieurs outsiders se mettent sur les rangs tels que *Prose*, *DynAOP* ou *Nanning*... ce marché est en pleine ébullition et ne se rationalisera probablement pas avant un an ou deux. Dans cette période, avant de débuter un nouveau projet, il faudra systématiquement s'assurer d'utiliser le tisseur le plus adapté aux contraintes du cycle de développement et d'exécution du logiciel à réaliser.

5.4.2. Tisseurs .NET

Malheureusement pour ses développeurs, le monde .NET est beaucoup moins bien loti en termes de tisseurs. Il n'en existe aucun qui soit du niveau de qualité ou de maturité d'un AspectJ, JAC ou JBossAOP.

Sans vouloir jeter la pierre, l'éditeur de la plate-forme .NET, Microsoft, a sa part de responsabilité dans cette situation: bien sûr parce qu'il ne propose aucun tisseur d'Aspects lui-même, mais aussi parce qu'en présentant initialement la programmation orientée méta-donnée comme étant de l'AOP, Microsoft a créé la confusion (au moins dans l'esprit des utilisateurs de la plate-forme) et le besoin d'un tisseur d'Aspects .NET ne s'est pas exprimé comme il aurait dû.

Heureusement, grâce au recul que nous ont donné ces dernières années, nombreux sont les développeurs et concepteurs qui se rendent compte du besoin d'un tisseur d'Aspects .NET solide et puissant; aussi voit-on se développer quelques nouveaux projets qui n'attendent que leurs utilisateurs pour devenir matures et s'enrichir de nouvelles fonctionnalités.

Loom.Net et Rapier-Loom.Net

Ces deux outils, gratuits pour le moment et gérés par le Hasso-Plattner-Institute de l'Université de Potsdam, sont en réalité deux facettes différentes du même tisseur. *Loom.NET* procède aux greffes sur le code CIL des assemblies .NET, alors que Rapier-Loom.Net opère lors de la compilation à la volée des classes (au **JIT** ou Just-In-Time compiler).

Dans les deux cas, un Aspect peut s'écrire dans n'importe quel langage supporté par la plate-forme .NET (C#, VB.NET, J#, etc...) et s'appliquer à des classes cibles éventuellement développées dans un autre langage .NET. Cela n'impose donc aucune contrainte aux développeurs d'Aspects ou d'objets métier quant au choix de leur langage de programmation.

Il est par contre nécessaire, comme avec JBossAOP et JAC, de faire en sorte que nos Aspects héritent de Loom.Aspect, une classe spécifique au tisseur.

Le langage de tissage de Loom n'est pas aussi puissant que ses collègues Java, et surtout, sa spécification ne peut être exprimée que dans les méta-données des aspects (sous forme d'Attributes de méthodes). Cela signifie que nous ne parviendrons pas à créer des Aspects réutilisables avec cet outil puisqu'il faudra éditer leur code et les re-compiler pour modifier l'ensemble des zones sur lesquelles doivent être greffés les Aspects.

Aspect#

Aspect# est un tisseur dynamique dont l'architecture ressemble à celle de SpringAOP en ce sens que les Aspects sont en réalité des objets intercalés entre les objets cibles et leurs utilisateurs (ce type d'objet est souvent appelé un dynamic proxy, un décorateur ou un intercepteur, selon les auteurs). Aspect# ne modifie donc pas le code intermédiaire CIL mais greffe ses Aspects uniquement en mémoire, à l'exécution. Cet outil est OpenSource et son développement est piloté par Henry Conceição et Rafael Steil.

Comme pour Loom.Net, on peut choisir avec Aspect# de développer à la fois le code cible et celui des Aspects dans n'importe quel langage

.NET. Les Aspects sont donc de simples classes, qui doivent toutefois implémenter une interface. Mais il faut souligner ici une initiative très intelligente d'Aspect#: il se conforme aux interfaces de l'AOP Alliance. A terme, les Aspects développés pour Aspect# seront donc portables sur la majorité des nouveaux tisseurs .NET qui suivront certainement son exemple.

Les avantages et inconvénients de ce tisseur sont globalement les mêmes que ceux de SpringAOP: le dynamisme d'Aspect# permet aux applications tissées de greffer et d'éliminer les Aspects à volonté, même au cours de l'exécution. La contre-partie est à chercher au niveau des performances globales que cette architecture induit: la présence d'un intercepteur devant un objet cible implique une invocation de méthode supplémentaire par rapport au code cible initial et le chargement des classes au sein du CLR se trouve allongé. Mais ne dramatisons pas: nombreuses seront les applications qui se contenteront de ce niveau de performances et pour qui le surcoût induit pourrait même être complètement indolore par rapport aux temps d'exécution des autres éléments de code (entrées sorties, communications réseau...).

AspectDNG

AspectDNG est un tisseur statique "à la AspectJ": il modifie le code intermédiaire CIL des assemblies cibles, qui peuvent donc avoir été développées avec n'importe quel langage .NET; de même, le code des Aspect peut être rédigé en C#, VB.NET ou tout autre langage pour peu qu'il soit compilé en une assembly standard.

AspectDNG est un logiciel libre que j'ai développé dans le cadre des activités de la communauté www.DotNetGuru.org. Un second développeur, Jean-Baptiste Evain, m'a rejoint pour améliorer la complétude de notre tisseur vis-à-vis des éléments du langage CIL.

Outre le fait de permettre le développement d'Aspects dans n'importe quel langage, AspectDNG n'impose l'usage d'aucune interface de programmation particulière. Les Aspects développés pour ce tisseur sont donc de simples classes, complètement indépendantes d'AspectDNG, et par là même portables sur tout autre tisseur qui offrirait la même souplesse.

Le langage de tissage utilisé par cet outil est XPath, pour une raison que vous comprendrez mieux dans le chapitre suivant. Mais disons que ce langage a un pouvoir d'expression extraordinaire, à défaut de disposer d'une syntaxe triviale... Avec AspectDNG, les expressions de tissage sont placées dans des fichiers XML que l'on fournit en paramètre du tisseur.

Ses points forts sont les performances du code résultant, grâce à la stratégie de tissage qu'il a suivie, et l'indépendance vis-à-vis de toute bibliothèque dynamique: à l'exécution des applications tissées, aucune bibliothèque de classe fournie par AspectDNG n'est requise, l'application est autonome, comme elle l'était avant tissage. Ses points faibles sont tout d'abord son manque de maturité: les projets .NET faisant usage d'AOP ne sont pas encore légion, nous n'avons donc pas encore eu beaucoup de retour d'expérience.

Et les autres...

Le monde .NET est resté longtemps silencieux au sujet de l'AOP. L'apparition de nouveaux tisseurs est un très bon signe du dynamisme naissant de ce domaine; le portage de frameworks d'IOC tels que Spring sur la plate-forme .NET nous offrira certainement un SpringAOP.NET, et comme la manipulation de code intermédiaire CIL devient plus outillée, de nouveaux tisseurs statiques devraient également voir le jour. Pour le plus grand bonheur des développeurs et des concepteurs .NET.

5.5. Fonctionnement interne d'un tisseur d'Aspects

Le mécanisme de tissage peut paraître magique à ceux qui découvrent l'instrumentation de code pour la première fois. Nous nous proposons donc ici de dédramatiser ce processus en dévoilant les détails d'implémentation du tisseur que nous connaissons le mieux: AspectDNG.

5.5.1. Objectifs et choix de conception

Voici les objectifs visés par AspectDNG, et qui ont donc guidé sa conception:

- maximiser la robustesse des applications,
- obtenir les meilleures performances possibles dans le code généré,
- être multi-langage, tant pour le code cible que pour les Aspects,
- interdire toute dépendance entre le code cible et le tisseur d'Aspects,
- éviter que les applications tissées ne nécessitent de bibliothèque propres au tisseur à l'exécution,
- limiter au maximum les adhérences entre le code des Aspects et le tisseur (donc ne pas créer de langage propriétaire de programmation d'Aspects et, si possible, faire en sorte que les Aspect ne dépendent d'aucune API. De même que pour le code cible, permettre aux développeurs d'Aspects de choisir leur langage de développement),
- séparer clairement les trois éléments: code cible, Aspects et description des greffes par le langage de tissage.

Pour garantir la meilleure robustesse au tissage et les meilleures performances à l'exécution, nous avons donc décidé qu'AspectDNG serait un tisseur statique. Et pour que les développeurs à la fois du code cible et de celui des Aspects ne soient pas liés à un langage de programmation particulier, AspectDNG travaille sur le résultat de la

compilation .NET: les assemblies (qui renferment code intermédiaire CIL et méta-données).

Pour être fidèles aux principes d'ouvert-fermé et d'inversion des dépendances, la notion de méta-donnée qui aurait permis de marquer les zones de greffe dans le code cible a été proscrite. Les Aspects, quant à eux, sont de simples classes développées en un langage .NET pur (C#, VB.NET...) qui ne dépendent ni d'une interface de programmation, ni de méta-données spécifiques au tisseur.

5.5.2. Manipuler des assemblies binaires

L'activité de tissage statique revient à manipuler du code avant ou après compilation. Sur la plate-forme .NET, de par son caractère multi-langage, il vaut mieux procéder à la manipulation après compilation sur le code CIL. Pour ce faire, plusieurs stratégies sont envisageables. La plus simple semble être d'exporter le CIL en une représentation textuelle, ce que les outils fournis par le SDK .NET permettent en standard, puis de travailler sur le texte (qui ressemble à un langage d'assembleur de très haut niveau) et enfin de reconstruire le CIL binaire après modification.

Pour vous faire une idée, examinons un extrait de texte représentant le code CIL du projet "SimpleTracesSansAspects" (contenant la classe Produit, Client) aperçu précédemment. Cette représentation peut être obtenue par la commande suivante:

ILDASM /ALL /TEXT /OUT=Produit.il SimpleTracesSansAspects.dll.

Et voici son résultat textuel:

```
01. .assembly extern mscorlib
02. {
03.   .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
04.   .ver 1:0:5000:0
05. }
06.
07. .assembly SimpleTracesSansAspects
08. {
09.   .hash algorithm 0x00008004
```

```
10.   .ver 0:0:0:0
11. }
12.
13. .module SimpleTracesSansAspects.exe
14. .imagebase 0x00400000
15. .subsystem 0x00000003
16. .file alignment 4096
17. .corflags 0x00000001
18.
19. .namespace Dotnetguru.BLL
20. {
21.   .class public auto ansi beforefieldinit Client
22.         extends [mscorlib]System.Object
23.   {
24.     .method public hidebysig
25.           instance void   Debiter(float64 prix) cil managed
26.     {
27.       .maxstack   0
28.       IL_0000:   nop
29.       IL_0001:    ret
30.     }
31.
32.     .method public hidebysig specialname rtspecialname
33.           instance void   .ctor() cil managed
34.     {
35.       .maxstack   1
36.       IL_0000:    ldarg.0
37.       IL_0001:   call instance void [mscorlib]System.Object::.ctor()
38.       IL_0006:   ret
39.     }
40.
41.   }
42.
43.   .class public auto ansi beforefieldinit PointEntree
44.         extends [mscorlib]System.Object
45.   {
46.     .method public hidebysig static
47.           void   Main() cil managed
```

```
48.     {
49.       .entrypoint
50.       .maxstack  2
51.       .locals init ([0] class Dotnetguru.BLL.Client c,
52.                     [1] class Dotnetguru.BLL.Produit p)
53.       IL_0000:  newobj instance void Dotnetguru.BLL.Client::.ctor()
54.       IL_0005:  stloc.0
55.       IL_0006:  newobj instance void Dotnetguru.BLL.Produit::.ctor()
56.       IL_000b:  stloc.1
57.       IL_000c:  ldloc.1
58.       IL_000d:  ldc.r8      10.
59.       IL_0016:  callvirt instance
60.                 void Dotnetguru.BLL.Produit::set_Prix(float64)
61.       IL_001b:  ldloc.1
62.       IL_001c:  ldc.i4.1
63.       IL_001d:  callvirt instance
64.                 void Dotnetguru.BLL.Produit::set_EnSolde(bool)
65.       IL_0022:  ldloc.1
66.       IL_0023:  ldloc.0
67.       IL_0024:  callvirt instance
68.                 void Dotnetguru.BLL.Produit::Acheter
69.                (class Dotnetguru.BLL.Client)
70.       IL_0029:  ret
71.     }
72.
73.     // ...
74.   }
75.
76.   // ...
77.
78. }
```

SimpleTracesSansAspects.il

Il est donc tout à fait concevable de travailler sur cette représentation pour implémenter le tissage d'Aspects:

Figure 21. Tissage sur la représentation textuelle du CIL

Mais cette approche souffre de plusieurs problèmes. Tout d'abord, elle dépend de l'outil ILDASM qui n'est disponible que sur Windows, alors que le framework .NET peut également fonctionner sous Linux grâce au projet Mono. D'autre part, il faudrait ré-implémenter un parseur et un générateur de ce format textuel, qui n'est pas standardisé contrairement au code binaire CIL et qui est donc susceptible de changer au gré des versions du SDK .NET. Une alternative à l'implémentation d'un parseur et d'un générateur aurait également pu être d'utiliser des expressions régulières pour modifier directement le texte, mais cette approche n'aurait été ni modulaire ni évolutive.

Nous avons donc choisi de baser le tissage sur un langage du même niveau que cette représentation textuelle, mais que l'on pourrait produire à la fois sur les CLR Windows et Linux. Ce langage, également textuel, devait être simple à parser, à produire et à requêter ou à transformer. Nous avons donc logiquement opté pour une représentation XML du code CIL que nous avons nommée **ILML**; AspectDNG dispose donc de deux outils, utilisés en bout de chaîne de tissage, qui permettent le passage d'une assembly binaire en ILML, et réciproquement. Voici un extrait de document ILML, correspondant au texte IL précédent:

```
01. <?xml version="1.0" encoding="utf-8" ?>
02. <Assembly
03.       fullName="SimpleTracesSansAspects, Version=0.0.0.0,
Culture=neutral"
04.       entryPointSignature="Dotnetguru.BLL.PointEntree Main()">
05.      <AssemblyName name="SimpleTracesSansAspects" >
06.         <Version major="0" minor="0" build="0" revision="0" />
07.      </AssemblyName>
08.      <ReferencedAssembly>
09.         mscorlib, Version=1.0.5000.0, Culture=neutral,
10.          PublicKeyToken=08b77a5c561934e0
11.      </ReferencedAssembly>
12.      <Attribute
13.          type="System.Diagnostics.DebuggableAttribute"
14.
constructorSignature="System.Diagnostics.DebuggableAttribute
15.              .ctor(System.Boolean,System.Boolean)"
16.         data="AQABAQAA" />
```

```
17.     <Module
18.         name="SimpleTracesSansAspects.exe"
19.         fullName="SimpleTracesSansAspects.exe">
20.         <Type
21.             baseType="System.Object"
fullName="Dotnetguru.BLL.Client"
22.             name="Client" namespace="Dotnetguru.BLL"
23.             isNotPublic="False" isPublic="True"
24.             isNestedPublic="False" isNestedPrivate="False"
25.             isNestedFamily="False" isNestedFamANDAssem="False"
26.             isNestedFamORAssem="False" isAutoLayout="True"
27.             isSequencialLayout="False" isExplicitLayout="False"
28.             isClass="True" isInterface="False"
29.             isAbstract="False" isSealed="False" isSpecialName="False"
30.             isImport="False" isSerializable="False"
31.             isAnsiClass="True" isUnicodeClass="False"
32.             isAutoClass="False" isBeforeFieldInit="True"
33.             isRTSpecialName="False" isHasSecurity="False">
34.             <Constructor
35.                 name=".ctor" signature="Dotnetguru.BLL.Client .ctor()"
36.                 isStandard="False" isVarArgs="False"
37.                 isAny="False" isHasThis="True" isExplicitThis="False"
38.                 isPrivateScope="False" isPrivate="False"
39.                 isFamANDAssem="False" isAssembly="False"
40.                 isFamily="False" isFamORAssem="False"
isPublic="True"
41.                 isStatic="False" isFinal="False" isVirtual="False"
42.                 isHideBySig="True" isReuseSlot="True"
43.                 isNewSlot="False" isAbstract="False"
isSpecialName="True"
44.                 isPinvokeImpl="False" isUnmanagedExport="False"
45.                 isRTSpecialName="True" isHasSecurity="False"
46.                 isRequireSecObject="False" isIL="True"
isNative="False"
47.                 isOPTIL="False" isRuntime="False"
isUnmanaged="False"
48.                 isManaged="True" isForwardRef="False"
49.                 isPreserveSig="False" isInternalCall="False"
```

```
50.                isSynchronized="False" isNoInlining="False">
51.              <Instruction offset="0" code="ldarg.0" />
52.              <Instruction offset="1" code="call"
53.                  operand="System.Object .ctor()" />
54.              <Instruction offset="2" code="ret" />
55.            </Constructor>
56.            <Method
57.                name="Debiter"
58.                signature="Dotnetguru.BLL.Client
Debiter(System.Double)"
59.                returnType="System.Void"     isStandard="False"
60.                isVarArgs="False" isAny="False" isHasThis="True"
61.                isExplicitThis="False" isPrivateScope="False"
62.                isPrivate="False" isFamANDAssem="False"
63.                isAssembly="False" isFamily="False"
64.              isFamORAssem="False" isPublic="True" isStatic="False"
65.                isFinal="False" isVirtual="False"
66.              isHideBySig="True" isReuseSlot="True"
isNewSlot="False"
67.                isAbstract="False" isSpecialName="False"
68.              isPinvokeImpl="False" isUnmanagedExport="False"
69.                isRTSpecialName="False" isHasSecurity="False"
70.              isRequireSecObject="False" isIL="True" isNative="False"
71.                isOPTIL="False" isRuntime="False"
72.              isUnmanaged="False" isManaged="True"
isForwardRef="False"
73.                isPreserveSig="False" isInternalCall="False"
74.              isSynchronized="False" isNoInlining="False">
75.              <Parameter
76.                  name="prix" type="System.Double" index="1"
77.                  isIn="False" isOut="False" isLcid="False"
78.                isRetval="False" isOptional="False" isNone="True"
79.                  isHasDefault="False" isHasFieldMarshal="False"
80.                isReserved3="False" isReserved4="False" />
81.              <Instruction offset="0" code="nop" />
82.              <Instruction offset="1" code="ret" />
83.            </Method>
84.          </Type>
```

```
85.         <Type
86.             baseType="System.Object"
87.             fullName="Dotnetguru.BLL.PointEntree"
88.             name="PointEntree" namespace="Dotnetguru.BLL"
89.             isNotPublic="False" isPublic="True" isNestedPublic="False"
90.            isNestedPrivate="False" isNestedFamily="False"
91.             isNestedFamANDAssem="False"
isNestedFamORAssem="False"
92.            isAutoLayout="True" isSequencialLayout="False"
93.             isExplicitLayout="False" isClass="True"
94.            isInterface="False" isAbstract="False" isSealed="False"
95.             isSpecialName="False" isImport="False"
96.            isSerializable="False" isAnsiClass="True"
97.             isUnicodeClass="False" isAutoClass="False"
98.            isBeforeFieldInit="True" isRTSpecialName="False"
99.             isHasSecurity="False">
100.             <Method
101.                 name="Main"
102.                 signature="Dotnetguru.BLL.PointEntree Main()"
103.                 returnType="System.Void" isStandard="False"
104.                 isVarArgs="False" isAny="False"
105.                 isHasThis="False" isExplicitThis="False"
106.                isPrivateScope="False" isPrivate="False"
107.                 isFamANDAssem="False" isAssembly="False"
108.                 isFamily="False" isFamORAssem="False"
109.                 isPublic="True" isStatic="True"
110.                 isFinal="False" isVirtual="False"
111.                isHideBySig="True" isReuseSlot="True"
isNewSlot="False"
112.                 isAbstract="False" isSpecialName="False"
113.                isPinvokeImpl="False" isUnmanagedExport="False"
114.                 isRTSpecialName="False" isHasSecurity="False"
115.                isRequireSecObject="False" isIL="True"
isNative="False"
116.                 isOPTIL="False" isRuntime="False"
117.                isUnmanaged="False" isManaged="True"
118.                 isForwardRef="False" isPreserveSig="False"
119.                 isInternalCall="False" isSynchronized="False"
```

```
120.              isNoInlining="False">
121.             <Local index="0" type="Dotnetguru.BLL.Client" />
122.             <Local index="1" type="Dotnetguru.BLL.Produit" />
123.             <Instruction offset="0" code="newobj"
124.                 operand="Dotnetguru.BLL.Client .ctor()" />
125.             <Instruction offset="1" code="stloc.0" />
126.             <Instruction offset="2" code="newobj"
127.                 operand="Dotnetguru.BLL.Produit .ctor()" />
128.             <Instruction offset="3" code="stloc.1" />
129.             <Instruction offset="4" code="ldloc.1" />
130.             <Instruction offset="5" code="ldc.r8" operand="10" />
131.             <Instruction offset="6" code="callvirt"
132.                 operand="Dotnetguru.BLL.Produit
133.                 set_Prix(System.Double)" />
134.             <Instruction offset="7" code="ldloc.1" />
135.             <Instruction offset="8" code="ldc.i4.1" />
136.             <Instruction offset="9" code="callvirt"
137.                 operand="Dotnetguru.BLL.Produit
138.                 set_EnSolde(System.Boolean)" />
139.             <Instruction offset="10" code="ldloc.1" />
140.             <Instruction offset="11" code="ldloc.0" />
141.             <Instruction offset="12" code="callvirt"
142.                 operand="Dotnetguru.BLL.Produit
143.                 Acheter(Dotnetguru.BLL.Client)" />
144.             <Instruction offset="13" code="ret" />
145.         </Method>
146.       </Type>
147.     </Module>
148. </Assembly>
```

Produit.ilml

Remarque: l'équivalence entre assembly et document ILML offre un intérêt qui dépasse de loin la notion de tissage d'Aspect. Elle permet de simplifier la création d'outils de génération de code automatique, de moteurs de statistiques ou de critiques de conception, et pourra même être utilisée pour assurer la synchronisation de modèles (UML, WhiteHorse...) avec le code (C#, VB.NET...). La bibliothèque **ILML.dll**, tout comme AspectDNG, est OpenSource et peut être utilisée sans

contrainte dans toutes vos applications.

5.5.3. Tissage par transformation XSLT

Avec AspectDNG, les applications cibles doivent être compilées en assemblies (.exe ou .dll). Il en est de même pour les bibliothèques d'Aspects. Comme nous l'avons vu dans la section précédente, cela revient à dire que nous disposons de documents XML qui représentent respectivement le code cible (nommons-le **"code-cible.ilml"**) et celui des aspects (**"code-aspects.ilml"**).

Reste à spécifier précisément **où** nous souhaitons greffer du code et **quel code** doit être greffé. Il suffit pour cela de disposer d'un langage d'expression qui permette de sélectionner des fragments de nos deux documents XML; logiquement, AspectDNG utilise **XPath** pour cela. Par exemple, dire que nous voulons appliquer un aspect à toutes les méthodes dont le nom commence par "Test" dans le package "Business" peut s'exprimer de la façon suivante:

//Method [starts-with(@name, 'Test')] [../@namespace = 'Business']

De même, pour spécifier que nous souhaitons greffer dans le code cible la méthode d'Aspect "TracerDansLaConsole":

//Method [@name = 'TracerDansLaConsole')]

Ces expressions sont rassemblées dans des fichiers XML qui configurent le tissage (nommons ce fichier "Advice-Trace.xml" dans notre exemple).

Tisser des Aspects se résume donc à appliquer une transformation qui travaille sur les documents XML code-cible.ilml et code-aspects.ilml, en respectant les directives spécifiées dans le fichier de configuration Advice-Trace.xml. Le document résultant est lui aussi au format ILML et peut dès lors être re-transcrit en une assembly .NET standard.

Figure 22. Tissage XML - XSLT

Comme vous le voyez, l'activité de tissage n'est donc pas particulièrement complexe. La véritable complexité sera de bien utiliser

les tisseurs du marché, et surtout, de concevoir nos applications en suivant de bonnes pratiques adaptées à l'AOP. C'est précisément l'objet des chapitres suivants.

[Addendum] Dans les versions les plus récentes d'AspectDNG, nous avons dû renoncer à utiliser XSLT pour implémenter le tissage. Le principe reste exactement le même mais le mécanisme a été ré-implémenté en C# (qui utilise massivement le langage XPath) de manière à gagner en performances (d'un facteur 6 !). Les fichiers de configuration, de même que les expressions de tissage, sont restés inchangés.

Chapitre 6.

Simples Aspects

Dans ce chapitre, nous allons tisser des fonctionnalités assez simples. Cela nous permettra de mettre en œuvre l'AOP sans trop avoir à insister sur la conception d'applications orientées Objet et Aspects, sachant que nous y reviendrons amplement dans les chapitres ultérieurs.

Les Aspects que nous allons présenter ici sont ceux qui viennent naturellement à l'esprit quand on présente les principes de l'AOP: ils sont assez intuitifs et vous les rencontrerez dans d'autres livres ou articles. Donc, outre leur présentation, nous allons mettre ce chapitre à profit pour comparer l'usage des trois tisseurs que nous avons retenus pour le reste de ce livre: AspectJ, JBossAOP et AspectDNG. Trois tisseurs statiques pour disposer de toute la puissance de l'AOP (dont les introductions de types et d'attributs) et pour obtenir de nos applications les meilleures performances possibles.

Contrairement aux chapitres précédents, les exemples de code seront ici rédigés en anglais, et ce pour deux raisons: d'une part certains exemples sont tirés d'une bibliothèque d'Aspects réutilisables et se trouvent donc être écrits en anglais; d'autre part, disposer du code en anglais vous simplifiera la tâche si vous souhaitez l'incorporer à vos propres projets.

6.1. Gestion des traces avec JBossAOP et AspectJ

Commençons avec le "Hello World" de la Programmation Orientée Aspect: la gestion des traces, ou logs, dans une application Orientée Objet. Comme nous l'avons vu dans le chapitre de découverte des principes de l'AOP, ce besoin se prête particulièrement bien au tissage de code, du moins tant que le code est répétitif et non contextuel.

Par contre, il ne faut surtout pas que l'avènement de l'AOP nous fasse oublier nos bonnes pratiques de conception Orientée Objet. Il ne faut donc surtout pas nous empresser de tisser des traces brutales (qui produiraient des lignes de texte en direction de la sortie standard ou d'un fichier) pour la seule raison qu'il sera plus simple et moins coûteux de modifier ce comportement grâce aux Aspects. Au contraire, et ce sera notre leitmotiv, il faudrait que les applications tissées soient aussi bien conçues que si nous les avions écrites nous-mêmes complètement, sans disposer de tisseur d'Aspects.

Reformulons l'idée sous-jacente aux phrases précédentes: **l'AOP est un outil d'industrialisation, mais les applications tissées doivent absolument respecter les principes de conception Objet** que nous avons rappelés au début de ce livre. Après tout, une fois les Aspects tissés, l'application résultante n'est "que" Orientée Objet.

Ainsi, à moins d'avoir une excellente raison, nous devons nous garder de ré-implémenter nous-mêmes un mécanisme spécifique de gestion des traces. Au contraire, utilisons un framework générique existant dans le monde Java. Mais lequel? En réalité, choisir un framework est lourd de conséquences car certaines applications ou modules qui choisiraient Log4J risqueraient de ne pas pouvoir être intégrées facilement avec d'autres modules basés sur les logs du JDK1.4 par exemple.

Pour résoudre ce problème d'hétérogénéité des frameworks disponibles, et pour éviter d'imposer aux applications de faire un choix de conception lourd de sens, il vaut mieux utiliser un **méta-framework**, ou moins prosaïquement une sur-couche générique de framework, qui permettrait de choisir plus tard sur quel framework concret nous

souhaitons nous reposer. Un tel méta-framework existe en Java, il s'agit de **commons-logging**, un sous projet mené au sein de la communauté Jakarta-Apache.

Figure 23. Le méta-framework commons-logging

6.1.1. Développement artisanal

Prenons l'exemple d'une application très simple, qui met en œuvre deux classes métier (Client et Product), manipulées par une classe EntryPoint qui se contente de simuler l'achat de produits par un certain nombre de clients.

Comme elle est minimaliste, nous pouvons nous permettre de développer complètement cette application en utilisant commons-logging manuellement dans nos trois classes. Il suffit pour cela:

- de déclarer un attribut de type Log dans chaque classe,
- de l'initialiser par le biais d'une Fabrique Abstraite de Log (fournie par commons-logging),
- puis d'invoquer les méthodes (warn(), info(), debug()) de l'objet ainsi créé dans le corps de nos méthodes.

Voici à quoi pourrait ressembler la classe Client:

```
01. package org.dotnetguru.stockmanagement.bll;
02.
03. import org.apache.commons.logging.Log;
04. import org.apache.commons.logging.LogFactory;
05.
06. public class Client {
07.     private static Log log = LogFactory.getLog(Client.class);
08.
09.     // Fields
10.     private String m_name;
11.
12.     private double m_balance;
13.
```

```
14.     // Getters and setters
15.     public double getBalance() {
16.         log.info("Reading Client.balance [" + m_balance + "]");
17.         return m_balance;
18.     }
19.
20.     protected void setBalance(double newAmount) {
21.         log.info("Writing Client.balance [" + newAmount + "]");
22.         m_balance = newAmount;
23.     }
24.
25.     public String getName() {
26.         log.info("Reading Client.name [" + m_name + "]");
27.         return m_name;
28.     }
29.
30.     protected void setName(String newName) {
31.         log.info("Writing Client.name [" + newName + "]");
32.         m_name = newName;
33.     }
34.
35.     // Constructors
36.     public Client(String name, double initialBalance) {
37.         log.info("Creating a new Client [" + name + ", " +
initialBalance + " ]");
38.         setName(name);
39.         setBalance(initialBalance);
40.     }
41.
42.     // Methods
43.     public void credit(double amount) {
44.         log.info("Invoking Client.credit [" + amount + "]");
45.         setBalance(getBalance() + amount);
46.     }
47.
48.     public void debit(double amount) {
49.         log.info("Invoking Client.debit [" + amount + "]");
50.         setBalance(getBalance() - amount);
```

```
51.     }
52. }
```

Client.java

Intentionellement, nous n'avons placé dans le corps du Client que des traces techniques et systématiques. La classe EntryPoint, quant à elle, trace des messages plus ancrés dans le comportement applicatif:

```
01. package org.dotnetguru.stockmanagement;
02.
03. import java.util.ArrayList;
04. import java.util.List;
05.
06. import org.apache.commons.logging.Log;
07. import org.apache.commons.logging.LogFactory;
08. import org.dotnetguru.stockmanagement.bll.Client;
09. import org.dotnetguru.stockmanagement.bll.Product;
10.
11. public class EntryPoint {
12.         private static Log log = LogFactory.getLog(EntryPoint.class);
13.
14.     public static void main(String argv[]) {
15.         // 100 Clients buy Products among 50 available in the system
16.
17.         log.info("Creating clients");
18.         List clients = new ArrayList();
19.         for (int i = 0; i < 100; i++) {
20.             clients.add(new Client("client-" + i, 1000 + 10 * i));
21.         }
22.
23.         log.info("Creating products");
24.         ArrayList products = new ArrayList();
25.         for (int i = 0; i < 50; i++) {
26.             products.add(new Product("product-" + i, 100, 200 + 10 *
i));
27.         }
28.
29.         log.info("Having clients buy products");
30.         for (int i = 0; i < clients.size(); i++) {
```

```
31.             Client c = (Client) clients.get(i);
32.             Product p = (Product) products.get(i % products.size());
33.             p.boughtBy(c);
34.         }
35.     }
36. }
```
EntryPoint.java

Cette fois, les messages de traces ne sont pas déductibles du seul comportement technique de notre classe (invocation de méthodes, accès aux attributs) mais ils renferment une valeur ajoutée applicative que seul un développeur peut apporter.

Le résultat (partiel) de l'exécution de ce petit programme est le suivant:

```
01. 12:47:34 [INFO] EntryPoint - -Creating clients
02. 12:47:34 [INFO] Client - -Creating a new Client [client-0, 1000.0 ]
03. 12:47:34 [INFO] Client - -Writing Client.name [client-0]
04. 12:47:34 [INFO] Client - -Writing Client.balance [1000.0]
05. 12:47:34 [INFO] Client - -Creating a new Client [client-1, 1010.0 ]
06. 12:47:34 [INFO] Client - -Writing Client.name [client-1]
07. 12:47:34 [INFO] Client - -Writing Client.balance [1010.0]
08. 12:47:34 [INFO] Client - -Creating a new Client [client-2, 1020.0 ]
09. 12:47:34 [INFO] Client - -Writing Client.name [client-2]
10. 12:47:34 [INFO] Client - -Writing Client.balance [1020.0]
11. [...]
12. 12:47:34 [INFO] EntryPoint - -Creating products
13. 12:47:34 [INFO] Product - -Creating a new Product [product-0, 100,
200.0 ]
14. 12:47:34 [INFO] Product - -Writing Product.name [product-0]
15. 12:47:34 [INFO] Product - -Writing Product.stock [100]
16. 12:47:34 [INFO] Product - -Writing Product.price [200.0]
17. 12:47:34 [INFO] Product - -Creating a new Product [product-1, 100,
210.0 ]
18. 12:47:34 [INFO] Product - -Writing Product.name [product-1]
19. 12:47:34 [INFO] Product - -Writing Product.stock [100]
20. 12:47:34 [INFO] Product - -Writing Product.price [210.0]
21. [...]
```

```
12:47:34 [INFO] EntryPoint - -Having clients buy products
12:47:34 [INFO] Product - -Invoking Product.boughtBy
    [org.dotnetguru.stockmanagement.bll.Client@1754ad2]
12:47:34 [INFO] Product - -Reading Product.stock [100]
12:47:34 [INFO] Product - -Writing Product.stock [99]
12:47:34 [INFO] Product - -Reading Product.price [200.0]
12:47:34 [INFO] Client - -Invoking Client.debit [200.0]
12:47:34 [INFO] Client - -Reading Client.balance [1000.0]
12:47:34 [INFO] Client - -Writing Client.balance [800.0]
12:47:34 [INFO] Product - -Invoking Product.boughtBy
    [org.dotnetguru.stockmanagement.bll.Client@1833955]
12:47:34 [INFO] Product - -Reading Product.stock [100]
12:47:34 [INFO] Product - -Writing Product.stock [99]
12:47:34 [INFO] Product - -Reading Product.price [210.0]
12:47:34 [INFO] Client - -Invoking Client.debit [210.0]
12:47:34 [INFO] Client - -Reading Client.balance [1010.0]
12:47:34 [INFO] Client - -Writing Client.balance [800.0]
[...]
```

Output.txt

Maintenant que nous connaissons les moindres détails du comportement et de l'implémentation de l'application "Client-Product", essayons d'extirper la gestion des traces et d'en faire un Aspect.

Remarque: la démarche que nous avons suivie ici (le fait de développer les classes comme si nous n'allions pas utiliser l'AOP) ne doit pas être appliquée à toutes les classes d'un véritable projet, mais il s'avère utile de disposer du code source d'une ou deux classes représentatives des besoins techniques de nos applications avant de concevoir les Aspects, puis de procéder à un *"refactoring orienté Aspect"* de notre code.

6.1.2. Tissage avec JBossAOP

Le code de la classe Client est le plus simple à re-façonner par le biais de l'AOP: à chaque début de méthode, il imprime une trace technique contenant le nom de la méthode courante et la valeur de ses paramètres.

Dans un premier temps, nous allons simplifier le problème et nous passer de l'attribut de type Log qui devrait se trouver dans chaque classe imprimant des traces. Nous ferons la recherche de l'objet de Log systématiquement, à l'exécution de chaque méthode. Nous procéderons à cette simplification afin de nous concentrer sur le tissage de code, avant d'aller plus loin avec le mécanisme d'introduction d'attribut.

Dans ces conditions, il suffit de développer le code qui sera greffée au début de chaque méthode du Client et du Produit. Avec JBossAOP, il faut pour cela créer une classe qui implémente l'interface **Interceptor**; puis tout se joue dans la méthode **invoke()** qui sera sollicitée au début de chaque méthode métier. Rien ne vaut un bon exemple, aussi empressons-nous d'examiner le code de notre premier Aspect, **MethodTraceAspect**:

```
01. package org.dotnetguru.stockmanagement.aspects;
02.
03. import java.lang.reflect.Method;
04.
05. import org.apache.commons.logging.Log;
06. import org.apache.commons.logging.LogFactory;
07. import org.jboss.aop.advice.Interceptor;
08. import org.jboss.aop.joinpoint.Invocation;
09. import org.jboss.aop.joinpoint.MethodInvocation;
10.
11. public class MethodTraceAspect implements Interceptor {
12.     public String getName() {
13.         return getClass().getName();
14.     }
15.
16.     public Object invoke(Invocation invoc) throws Throwable {
17.         MethodInvocation mInvoc = (MethodInvocation) invoc;
18.
19.         Method targetMethod = mInvoc.getMethod();
20.         String targetMethodName = targetMethod.getName();
21.         Class targetClass = mInvoc.getTargetObject().getClass();
22.         Object[] targetMethodArgs = mInvoc.getArguments();
23.
24.         Log log = LogFactory.getLog(targetClass);
```

```
25.
26.         if (targetMethodName.startsWith("set")) {
27.             traceSetter(targetMethod, targetClass, targetMethodArgs,
28.                     log);
29.         } else if (targetMethodName.startsWith("get")) {
30.             traceGetter(targetMethod, targetClass, log);
31.         } else {
32.             traceMethod(targetMethod, targetClass, targetMethodArgs,
33.                     log);
34.         }
35.
36.         return mInvoc.invokeNext();
37.     }
38.
39.     private void traceMethod(
40.             Method targetMethod,
41.             Class targetClass,
42.             Object[] targetMethodArgs,
43.             Log log) {
44.         StringBuffer buf = new StringBuffer("Invoking ");
45.         traceFullName(targetMethod, targetClass, buf, 0);
46.         traceArguments(targetMethodArgs, buf);
47.         log.info(buf);
48.     }
49.
50.     private void traceGetter(
51.             Method targetMethod,
52.             Class targetClass,
53.             Log log) {
54.         StringBuffer buf = new StringBuffer("Reading ");
55.         traceFullName(targetMethod, targetClass, buf, 3);
56.         log.info(buf);
57.     }
58.
59.     private void traceSetter(
60.             Method targetMethod,
61.             Class targetClass,
62.             Object[] targetMethodArgs,
```

```
63.             Log log) {
64.         StringBuffer buf = new StringBuffer("Writing ");
65.         traceFullName(targetMethod, targetClass, buf, 3);
66.         traceArguments(targetMethodArgs, buf);
67.         log.info(buf);
68.     }
69.
70.     private void traceFullName(
71.             Method targetMethod,
72.             Class targetClass,
73.             StringBuffer buf,
74.             int charsToSkip) {
75.         String className = targetClass.getName();
76.         int packageNameLength =
targetClass.getPackage().getName()
77.                 .length();
78.
79.         if (packageNameLength > 0) {
80.             className = className.substring(packageNameLength +
1);
81.         }
82.         buf.append(className);
83.         buf.append(".");
84.
buf.append(targetMethod.getName().substring(charsToSkip));
85.     }
86.
87.     private void traceArguments(
88.             Object[] targetMethodArgs,
89.             StringBuffer buf) {
90.         buf.append(" [");
91.         if (targetMethodArgs != null) {
92.             for (int i = 0; i < targetMethodArgs.length; i++) {
93.                 buf.append(targetMethodArgs[i]);
94.                 if (i != targetMethodArgs.length - 1) {
95.                     buf.append(", ");
96.                 }
97.             }
```

```
98.         }
99.         buf.append("]");
100.     }
101. }
```
MethodTraceAspect.java

Vous l'aurez remarqué, à l'exception de "invoke()", toutes les méthodes sont privées et ne servent qu'à structurer le code de l'Aspect lui-même. Comme un Aspect est une simple classe Java avec JBossAOP, nous pouvons laisser libre cours à nos réflexes habituels et créer des méthodes utilitaires (comme dans notre exemple) ou encore des classes annexes sur lesquelles les Aspects pourront se reposer. C'est particulièrement intéressant dès lors que le comportement d'un Aspect est complexe ou que le même besoin se retrouve dans plusieurs Aspects. Nous verrons un exemple d'Aspect complexe avec le moteur de statistiques dans une prochaine section.

Il ne reste plus qu'à indiquer au tisseur JBossAOP où il doit greffer notre Aspect. Cela se fait dans le document **jboss-aop.xml**:

```
01. <aop>
02.     <bind pointcut="execution
03.         (* org.dotnetguru.stockmanagement.bll.*->*(..))">
04.         <interceptor class=
05.             "org.dotnetguru.stockmanagement.
06.                 aspects.MethodTraceAspect"/>
07.     </bind>
08. </aop>
```
jboss-aop.xml

De leur côté, les classes Client et Product voient leur syntaxe considérablement s'alléger et leur dépendance vis-à-vis du méta-framework commons-logging disparaît. Elles redeviennent des classes purement métier:

```
01. package org.dotnetguru.stockmanagement.bll;
02.
03. public class Client {
04.     // Fields
05.     private String m_name;
```

```
06.
07.     private double m_balance;
08.
09.     // Getters and setters
10.     public double getBalance() {
11.         return m_balance;
12.     }
13.
14.     protected void setBalance(double newAmount) {
15.         m_balance = newAmount;
16.     }
17.
18.     public String getName() {
19.         return m_name;
20.     }
21.
22.     protected void setName(String newName) {
23.         m_name = newName;
24.     }
25.
26.     // Constructors
27.     public Client(String name, double initialBalance) {
28.         setName(name);
29.         setBalance(initialBalance);
30.     }
31.
32.     // Methods
33.     public void credit(double amount) {
34.         setBalance(getBalance() + amount);
35.     }
36.
37.     public void debit(double amount) {
38.         setBalance(getBalance() - amount);
39.     }
40. }
```

Client.java

Nous avons donc atteint une partie de notre objectif: la gestion des

traces techniques et systématiques a complètement disparu des classes métier. Mais il reste à régler le problème celles qui ont une valeur ajoutée applicative ou métier et que seul un développeur peut produire. Il faudrait simplement rendre transparente à ce dernier l'utilisation d'un framework technique tel que commons-logging.

A nouveau, si les outils Objet sont impuissants dans cette situation, il n'en va pas de même pour l'AOP: il suffirait de remplacer l'usage de commons-logging dans le reste du code "fait main" par l'utilisation d'un mécanisme tout à fait standard pour produire des messages textuels. Par exemple en Java, il suffirait de remplacer toutes les invocations de **log.info(..)** par un simple **System.out.println(..)**.

Jusque là, rien de très novateur; au contraire, nous avons remplacé l'usage d'un meta-framework par l'invocation d'une trace concrète sur la sortie standard... Nous avons donc visiblement perdu la possibilité d'opter pour une redirection de cette trace dans un framework de logs concret tel que log4j ou simplelog...

Attendez, voici la deuxième étape: grâce à un second Aspect, nous allons **remplacer toutes les invocations directes de System.out.println(..) par log.info(..)**. Ainsi, l'application cible ne dépendra plus d'aucun framework, seuls nos Aspects seront tributaires de commons-logging.

Cette fois, commençons par examiner le descripteur de tissage jboss-aop.xml:

```
01. <aop>
02.    <bind pointcut="execution
03.        (* org.dotnetguru.stockmanagement.bll.*->*(..))">
04.       <interceptor class=
05.          "org.dotnetguru.stockmanagement.
06.              aspects.MethodTraceAspect"/>
07.    </bind>
08.
09.    <bind pointcut="
10.        call
11.        (* java.io.PrintStream->println(..)) AND
12.         withincode
13.        (* org.dotnetguru.stockmanagement.*.*->main(*))">
```

```
14.       <interceptor class=
15.           "org.dotnetguru.stockmanagement.
16.              aspects.ConsoleInterceptorTraceAspect"/>
17.    </bind>
18. </aop>
```

jboss-aop.xml

Dans son langage un peu particulier, ce fichier exprime que:

- MethodTraceAspect doit toujours être tissé au début de chaque méthode de chaque classe métier (ie du package "..stockmanagement.bll").
- ConsoleInterceptorTraceAspect doit être tissé de manière à intercepter les invocations de la méthode println(..) sur un flux de sortie PrintWriter, à condition que ces invocations soient réalisées à partir des classes de notre projet (ie dans le package "..stockmanagement").

Il ne reste plus qu'à implémenter ce deuxième intercepteur:

```
01. package org.dotnetguru.stockmanagement.aspects;
02.
03. import org.apache.commons.logging.LogFactory;
04. import org.jboss.aop.advice.Interceptor;
05. import org.jboss.aop.joinpoint.Invocation;
06. import org.jboss.aop.joinpoint.MethodCalledByMethodInvocation;
07.
08. public class ConsoleInterceptorTraceAspect implements Interceptor {
09.     public String getName() {
10.         return getClass().getName();
11.     }
12.
13.     public Object invoke(Invocation invoc) throws Throwable {
14.         MethodCalledByMethodInvocation mInvoc =
15.             (MethodCalledByMethodInvocation) invoc;
16.
17.         Class callingClass = mInvoc.getCallingClass();
18.         Object[] targetMethodArgs = mInvoc.getArguments();
19.
```

```
20.         if (targetMethodArgs != null  targetMethodArgs.length > 0) {
21.
LogFactory.getLog(callingClass).info(targetMethodArgs[0]);
22.         }
23.         return null;
24.     }
25. }
```

ConsoleInterceptorTraceAspect.java

Grâce à ce second Aspect, la classe EntryPoint (ainsi que toute autre classe souhaitant imprimer des traces applicatives) est effectivement rendue indépendante de tout framework technique:

```
01. package org.dotnetguru.stockmanagement.ui;
02.
03. import java.util.ArrayList;
04. import java.util.List;
05.
06. import org.dotnetguru.stockmanagement.bll.Client;
07. import org.dotnetguru.stockmanagement.bll.Product;
08.
09. public class EntryPoint {
10.         public static void main(String argv[]) {
11.         // 100 Clients buy Products among 50 available in the system
12.         System.out.println("Creating clients");
13.         List clients = new ArrayList();
14.         for (int i = 0; i < 100; i++) {
15.             clients.add(new Client("client-" + i, 1000 + 10 * i));
16.         }
17.
18.         System.out.println("Creating products");
19.         ArrayList products = new ArrayList();
20.         for (int i = 0; i < 50; i++) {
21.             products.add(new Product("product-" + i, 100, 200 + 10 *
i));
22.         }
23.
24.         System.out.println("Having clients buy products");
25.         for (int i = 0; i < clients.size(); i++) {
```

```
26.             Client c = (Client) clients.get(i);
27.             Product p = (Product) products.get(i % products.size());
28.             p.boughtBy(c);
29.         }
30.     }
31. }
```
EntryPoint.java

Ce deuxième Aspect est plus critiquable que le premier:

- Comment faire pour différencier les traces d'une véritable interaction avec un utilisateur en ligne de commande (même si ce type d'application se fait rare)?
- System.out.println() est une expression idiomatique Java, mais elle fait appel à une couche technique (le package "java.io") et sa présence dans un objet métier n'est donc pas beaucoup moins indésirable que celle de commons-logging.

Pour répondre à la première critique, nous pourrions utiliser la même technique utilisant le framework de logs standard du JDK1.4 plutôt que System.out.println(..) puis intercepter les invocations de méthodes vers ce framework et les remplacer par l'utilisation de commons-logging (ou directement d'un autre framework concret pour des raisons de performances).

Malheureusement, il n'existe pas de parade à la seconde remarque: pour produire des traces applicatives, nos classes métier devront toujours accepter d'avoir une dépendance, fût-elle très légère, vis-à-vis d'une couche plus technique.

6.1.3. Faisons le point

Le travail accompli dans la section précédente a permis de faire en sorte que les classes EntryPoint, Client et Product n'aient plus du tout conscience de produire des traces puisque cette intelligence technique a été externalisée sous forme d'Aspect.

Toutefois, la conception retenue n'est pas idéale, en particulier en termes de performances à l'exécution. Or il est crucial de prêter une

attention plus grande encore que d'habitude à ce point lors de l'implémentation d'Aspects, surtout si ceux-ci sont voués à être tissés sur un très grand nombre de zones de greffe.

Un premier problème relève du choix de conception que nous avons fait: nous avons tissé le même intercepteur "MethodTraceAspect" en début de toutes les méthodes. Dès lors, il devient indispensable de tester le nom de la méthode interceptée afin de déterminer s'il s'agit d'un getter, un setter ou d'une véritable méthode. C'est très simple: un test sur les trois premiers caractères de son nom semble suffire...

Or ce n'est pas le cas: tout d'abord, notre approche était naïve car d'autres méthodes peuvent tout à fait s'appeler "settle()" ou "setUp()" par exemple, sans que cela signifie qu'il s'agisse de modificateurs de propriétés. Et d'autre part, au vu des outils dont nous disposons, il est dommage d'analyser systématiquement les noms des méthodes à l'exécution du programme, alors que cela pourrait être fait statiquement par le tisseur d'Aspects lui-même. Il suffit pour cela de dissocier les Aspects en:

- FieldWriteTraceAspect (tissé après l'écriture d'un champ),
- FieldReadTraceAspect (tissé après l'écriture d'un champ),
- MethodTraceAspect (allégé, donc, et tissé au début de l'invocation d'une méthode),
- ConstructorTraceAspect (tissé au début de l'invocation d'un constructeur).

En spécialisant ainsi les Aspects, nous gagnons en sémantique ainsi qu'en performances car chacun ne sera tissé qu'aux endroits judicieux, et n'aura donc plus à se soucier du contexte. Essayons d'en déduire notre premier principe de conception Orientée Aspect: **en règle générale, tout comme il est néfaste de devoir tester le type d'un Objet en POO, un Aspect ne devrait pas avoir à tester le type d'élément sur lequel il est greffé en AOP**.

Le deuxième problème posé par la conception que nous avons choisie est de devoir récupérer dynamiquement l'objet de type Log, à chaque sollicitation d'un constructeur, d'une méthode ou d'un attribut. A nouveau, ce ne serait pas choquant si nos Aspects n'étaient pas voués à

être tissés à de nombreux endroits, et donc si leur code ne risquait pas de s'exécuter très souvent.

Avant de réfléchir à l'utilisation de l'AOP, les classes Client, Product et EntryPoint étaient optimisées pour ne pas subir ce problème: chacune d'entre elles déclarait un attribut statique de type Log et se contentaient de l'utiliser au moment opportun dans le corps de leurs méthodes. Il suffirait donc de revenir à cette conception plutôt que d'invoquer *LogFactory.getLog(nomDeClasse)* systématiquement.

Pour cela, il va falloir introduire un attribut statique de type Log dans chaque classe et l'initialiser en fonction du nom de celle-ci. Nous allons le faire avec l'autre tisseur statique que nous avons retenu: AspectJ.

6.1.4. Tissage avec AspectJ

AspectJ supporte le mécanisme d'introduction (de méthode et d'attribut) mais il souffre d'un certain nombre de limitations à ce sujet.

Tout d'abord, AspectJ ne permet pas de tisser du code dans le bloc statique des classes Java, ce qui interdit d'initialiser notre attribut statique "log" à une valeur calculée en fonction du nom de la classe cible (Client, Product ou EntryPoint).

D'autre part, la syntaxe actuelle (AspectJ version 1.2) ne permet d'introduire des attributs ou des méthodes que sur une classe cible à la fois. Dans notre exemple, aucun problème, mais dans un véritable projet, cela n'est pas acceptable. Heureusement, AspectJ propose un moyen de contournement. Il "suffit" de:

- Déclarer une interface, disons **LogConsumer** dans notre cas. Nous allons laisser cette interface vide, elle ne sera qu'un marqueur de classes (vous allez comprendre).
- Introduire une relation d'implémentation entre cette interface et toutes les classes sur lesquelles nous souhaitons introduire l'attribut "log". Typiquement, toutes les classes métier et applicatives de nos systèmes.
- Enfin, introduire l'attribut "log" sur l'interface "LogConsumer". Cela aura pour effet réel de procéder à l'introduction sur toutes les classes

qui implémentent cette interface.

Il semblerait donc que nous soyons sur la bonne piste... Mais comme nous l'annonce le compilateur d'AspectJ, l'introduction d'attributs statiques sur une interface n'est pas une fonctionnalité supportée!

Nous avons donc deux options: soit nous tissons notre attribut et son utilisation autant de fois que de classes cibles, soit nous acceptons que cet attribut ne soit pas statique. Gardant en vue la maintenabilité et la souplesse de nos applications, seule la seconde option est tolérable; mais il est évident que nous touchons ici à une limitation importante d'AspectJ qui n'existe pas dans d'autres tisseurs comme AspectDNG. Voici le code de l'Aspect correspondant:

```
01. package org.dotnetguru.stockmanagement.aspects;
02. import org.apache.commons.logging.*;
03.
04. public aspect TraceAspect {
05.     private interface LogConsumer {}
06.
07.     declare parents:
08.         org.dotnetguru.stockmanagement.bll.*
09.         implements LogConsumer;
10.
11.        declare parents:
12.         org.dotnetguru.stockmanagement.ui.*
13.         implements LogConsumer;
14.
15.        private Log LogConsumer.log =
16.         LogFactory.getLog(getClass());
17.
18.     void around(Object o, LogConsumer consumer):
19.         this(consumer)
20.         call(void java.io.PrintStream.println(..))
21.         args(o){
22.        consumer.log.info(o);
23.     }
24.
25.       before(LogConsumer consumer):
26.         this(consumer)
```

```
27.         execution(* *(..)){
28.        StringBuffer buf = new StringBuffer("Invoking ");
29.                traceFullName
30.           (thisJoinPoint.getSignature().getName(),
31.            thisJoinPoint.getThis().getClass(), buf, 0);
32.        traceArguments(thisJoinPoint.getArgs(), buf);
33.        consumer.log.info(buf);
34.    }
35.
36.        before(LogConsumer consumer):
37.         this(consumer)
38.         get(* *){
39.        StringBuffer buf = new StringBuffer("Reading ");
40.                 traceFullName
41.           (thisJoinPoint.getSignature().getName(),
42.            thisJoinPoint.getThis().getClass(), buf, 0);
43.        consumer.log.info(buf);
44.    }
45.
46.    before(LogConsumer consumer):
47.         this(consumer)
48.         set(* *){
49.        StringBuffer buf = new StringBuffer("Writing ");
50.        traceFullName
51.           (thisJoinPoint.getSignature().getName(),
52.            thisJoinPoint.getThis().getClass(), buf, 0);
53.        traceArguments(thisJoinPoint.getArgs(), buf);
54.        consumer.log.info(buf);
55.    }
56.
57.    // Utility methods
58.
59.        private void traceFullName(
60.           String targetMethodName,
61.           Class targetClass,
62.           StringBuffer buf,
63.           int charsToSkip) {
64.        String className = targetClass.getName();
```

```
65.         int packageNameLength =
66.             targetClass.getPackage().
67.             getName().length();
68.
69.         if (packageNameLength > 0) {
70.             className = className.substring
71.                 (packageNameLength + 1);
72.         }
73.         buf.append(className);
74.         buf.append(".");
75.         buf.append(targetMethodName.substring(charsToSkip));
76.     }
77.
78.     private void traceArguments(
79.             Object[] targetMethodArgs,
80.             StringBuffer buf) {
81.         buf.append(" [");
82.         if (targetMethodArgs != null) {
83.             for (int i = 0; i < targetMethodArgs.length; i++) {
84.                 buf.append(targetMethodArgs[i]);
85.                 if (i != targetMethodArgs.length - 1) {
86.                     buf.append(", ");
87.                 }
88.             }
89.         }
90.         buf.append("]");
91.     }
92. }
```

TraceAspect.java

Cette syntaxe un peu biscornue mérite quelques explications:

- *declare parents ... implements LogConsumer* permet de tisser la relation d'implémentation de l'interface LogConsumer sur un ensemble de classes.
- *private Log LogConsumer.log = ...;* introduit un attribut "log" dans toutes les classes qui implémentent l'interface LogConsumer, c'est-à-dire toutes les classes visées par le tissage précédent.

- *void around(Object o, LogConsumer consumer)* permet de tisser un Aspect littéralement "autour" de l'invocation de méthodes, nous laissant la possibilité d'invoquer la méthode cible ou non. Couplé à la zone de greffe *call(void java.io.PrintStream.println(..))*, cela permet de remplacer les invocations de la méthode *println(...)* par l'invocation de la méthode *info(..)* sur l'attribut *log* précédemment introduit.

- Enfin, les autres tissages permettent d'insérer du code au début de chaque méthode, avant chaque accès en lecture et en écriture des attributs. Dans ces trois cas, l'aspect peut accéder au contexte sur lequel il a été tissé grâce à un objet spécifique: *thisJoinPoint*. Par exemple, *thisJoinPoint.getSignature()* permet de savoir quelle est la méthode cible et *thisJoinPoint.getThis()* l'objet cible.

Si l'on fait abstraction de cette syntaxe particulière avec laquelle il faut se familiariser, cet exemple ne fait que mettre en application directe des principes de l'AOP: introduction d'attribut, tissage au début du corps des méthodes et tissage avant l'accès en lecture/écriture aux attributs. Notre seul regret est de ne pas avoir pu rendre statique l'attribut introduit sur les classes cibles. Gageons que cette limitation sera dépassée dans les versions suivantes d'AspectJ.

6.1.5. Tissages comparés

Un exercice intéressant, à la fois pour évaluer un tisseur d'Aspect et pour mieux comprendre la mécanique interne de l'AOP, consiste à tisser des Aspects et à analyser le code résultant. C'est ce que nous nous proposons de faire ici sur la classe Product que nous avons décompilée avec l'outil **Jad** (Java Decompiler).

Voici tout d'abord le code de la méthode **Product.setName()** après tissage par JBossAOP (greffe "au début de la méthode setName()"):

```
01. package org.dotnetguru.stockmanagement.bll;
02.
03. import org.jboss.aop.*;
04. import org.jboss.aop.joinpoint.InvocationBase;
05.
```

```
06. // [Instructions decoupees sur plusieurs lignes]
07.
08. public class Product implements Advised{
09.     public void setName(String s){
10.         if(((Advisor) (aop$classAdvisor$aop)).doesHaveAspects
11.          || _instanceAdvisor != null
12.          _instanceAdvisor.hasInstanceAspects){
13.           org.jboss.aop.advice.Interceptor ainterceptor[] =
14.            aop$MethodInfo_setName1344297395548290975.
15.           interceptors;
16.           if(_instanceAdvisor != null)
17.              ainterceptor = _instanceAdvisor.getInterceptors
18.              (aop$MethodInfo
19.                 _setName1344297395548290975.interceptors);
20.           Product
21.              _setName_1344297395548290975
22.                 _OptimizedMethodInvocationproduct
23.                 _setname_1344297395548290975
24.                 _optimizedmethodinvocation =
25.               new Product
26.                 _setName_1344297395548290975
27.                 _OptimizedMethodInvocation
28.                 (aop$MethodInfo
29.                 _setName1344297395548290975,
30.                  ainterceptor);
31.           product_setname_1344297395548290975
32.              _optimizedmethodinvocation.arg0 = s;
33.           product_setname_1344297395548290975
34.              _optimizedmethodinvocation.
35.                 setTargetObject(this);
36.           product_setname_1344297395548290975
37.              _optimizedmethodinvocation.typedTargetObject
38.                  = this;
39.           product_setname_1344297395548290975
40.              _optimizedmethodinvocation.
41.                 setAdvisor(aop$classAdvisor$aop);
42.           product_setname_1344297395548290975
43.              _optimizedmethodinvocation.invokeNext();
```

```
44.         } else{
45.             org$dotnetguru$stockmanagement
46.                 $bll$Product$setName$aop(s);
47.         }
48.     }
49. }
```

Product-JBossAOP.jad

Comparez le code précédent avec celui de la même méthode Product.setName() mais tissée par AspectJ (greffes "au début de chaque méthode" et "avant l'accès en écriture aux attributs"):

```
01. package org.dotnetguru.stockmanagement.bll;
02.
03. import org.apache.commons.logging.Log;
04. import org.aspectj.runtime.internal.Conversions;
05. import org.aspectj.runtime.reflect.Factory;
06. import org.dotnetguru.stockmanagement.aspects.TraceAspect;
07.
08. public class Product implements
09.
org.dotnetguru.stockmanagement.aspects.TraceAspect.LogConsumer{
10.
11.     public void setName(String newName)
12.     {
13.         String s1 = newName;
14.         org.aspectj.lang.JoinPoint joinpoint1 =
15.             Factory.makeJP(ajc$tjp_3, this, this, s1);
16.         TraceAspect.aspectOf().ajc$before$org_dotnetguru_
17.             stockmanagement_aspects_TraceAspect$2$a10b154
18.                 (this, joinpoint1);
19.         this;
20.         String s = newName;
21.         Product product;
22.         product;
23.         org.aspectj.lang.JoinPoint joinpoint =
24.             Factory.makeJP(ajc$tjp_2, this, product, s);
25.         TraceAspect.aspectOf().ajc$before$org_dotnetguru_
26.             stockmanagement_aspects_TraceAspect$4$903e36dd
```

```
27.                 (this, joinpoint);
28.         product.m_name = s;
29.         return;
30.     }
31. }
```
Product-AspectJ.jad

Bien sûr, ce code tissé n'est pas destiné à être relu. Sa syntaxe est ardue, en particulier à cause des noms très longs de variables représentant les Aspect. Mais on arrive à reconnaître l'origine des éléments de code entre ce que nous avions développé dans la classe Product et le travail de greffe du tisseur.

Il est difficile, sur un exemple aussi simple, de dire quel est le meilleur tisseur entre AspectJ et JBossAOP en termes de qualité du code tissé. D'un côté, on peut simplement regretter la lourdeur de l'instanciation de l'objet représentant le contexte de tissage avec JBossAOP, alors que AspectJ passe par une Fabrique pour simplifier le code. Cela peut avoir son importance si l'on greffe un Aspect sur de nombreuses zones: la taille du bytecode résultant sera plus importante avec JBossAOP.

D'un autre côté, si le code produit par AspectJ est plus propre, il n'est pas exempt de lourdeurs inutiles telles que les instructions inopérantes **this;** et **product;**. Fort heureusement, cela ne devrait pas porter à conséquence car les compilateurs JIT, qui analyseront ce code à leur tour, devraient élaguer ce type d'instructions.

6.2. Moteur de statistiques avec AspectDNG

6.2.1. Implémentation

Changeons de plate-forme et d'exemple. Nous allons maintenant nous attacher à comptabiliser le nombre d'accès en lecture/écriture de tous les attributs, le nombre d'invocations de toutes les méthodes ainsi que le temps passé à exécuter chacune d'entre elles au cours du déroulement d'une application .NET.

Remarque: l'objectif n'est pas de créer ici un logiciel complet et prêt à l'emploi dans le cas général, mais plutôt de réfléchir à la conception par Aspects et de comparer les différents choix d'implémentation d'un moteur de statistiques.

La question cruciale dans notre problème est la suivante: où allons-nous comptabiliser les statistiques relatives à chaque élément de code? Examinons les différentes solutions:

- Nous pourrions introduire dans chaque classe de nouveaux attributs statiques: par exemple, une Hashtable permettrait aisément de stocker les accès à tous les attributs et méthodes d'une classe.

  Mais cette stratégie pose deux problèmes: tout d'abord, elle ne fonctionne pas dans un environnement multi-thread car plusieurs tâches risquent d'essayer de mettre à jour simultanément les statistiques, qui pourraient s'en trouver faussées (il pourrait arriver que le nombre d'accès comptabilisés soit inférieur au nombre d'accès réel). D'autre part, il semble difficile de faire ensuite un recensement de toutes les statistiques, classe par classe: l'information est beaucoup trop disséminée à travers notre programme.

- Pour éliminer les incohérences dues au multi-threading tout en évitant d'entrer trop souvent en section d'exclusion mutuelle, nous pourrions nous reposer sur un environnement de stockage qui est par définition multi-thread safe: le **Thread Local Storage** (TLS). Cette

zone mémoire peut être allouée dans le contexte de n'importe quelle méthode et se trouve automatiquement attachée au Thread courant.

Mais quelle organisation de données choisir? Un "slot" de données par méthode et par Thread qui l'exécute (dans ce cas, il faut bien faire attention à ne pas allouer deux fois le même slot dans le cas d'invocation de méthodes récursives ou plus généralement ré-entrantes)? Ou un slot global?

Et comme précédemment, il faudrait centraliser les statistiques à la fin de l'exécution de chaque méthode, de façon à ne pas avoir à faire un recensement global Thread par Thread.

- Une dernière solution serait de considérer que le comportement des Aspects de statistiques est sans état. Pour cela, il suffirait que le moteur de statistiques ne soit pas vraiment implémenté dans le corps des méthodes, mais dans une classe externe, technique, qui centraliserait toutes les informations et gèrerait les problèmes d'accès concurrents. La seule chose à tisser dans les objets observés serait la sollicitation de cette classe technique.

Pour plusieurs raisons, dont la simplicité et la réutilisabilité, nous avons choisi la dernière option. Concrètement, elle se décompose en trois classes techniques et un Aspect:

- Les classes *FieldStatistic* et *MethodStatistic* permettent de comptabiliser les informations liées respectivement aux manipulations d'attributs et de méthodes ou de constructeurs.

- La classe *StatisticManagement* constitue la façade du mécanisme de statistiques. Elle permet d'incrémenter le nombre d'accès en lecture/écriture d'un champ et de comptabiliser le temps passé à exécuter chaque méthode. Elle est également capable d'externaliser ses statistiques au format XML à tout moment.

- Enfin l'Aspect, qui se contente d'invoquer les services de StatisticManagement aux bons moments: comptabilisation des statistiques après l'accès aux attributs, avant et après l'invocation de méthodes ou de constructeurs, et externalisation des statistiques globales en XML juste avant la fin de l'exécution de la méthode Main de l'application.

Le code de la classe MethodStatistic est assez simple (celui de la classe FieldStatistic est du même acabit):

```
01. using System.Reflection;
02. using AspectDNG;
03.
04. namespace DotNetGuru.Aspects.Statistic {
05.     [Insert("")]
06.     public class MethodStatistic {
07.         // Fields
08.         private MethodBase m_metadata;
09.         private int m_invocations;
10.         private long m_totalExecutionTime;
11.
12.             // Getters and Setters
13.         public MethodBase Metadata{
14.             get { return m_metadata; }
15.         }
16.
17.             public int Invocations{
18.             get { return m_invocations; }
19.         }
20.         protected void SetInvocations(int newInvocations){
21.             m_invocations = newInvocations;
22.         }
23.
24.                 public long TotalExecutionTime{
25.             get { return m_totalExecutionTime; }
26.         }
27.         protected void SetTotalExecutionTime
28.             (long newTotalExecutionTime){
29.             m_totalExecutionTime = newTotalExecutionTime;
30.         }
31.
32.             // Constructors
33.         public MethodStatistic(MethodBase metadata){
34.             m_metadata = metadata;
35.         }
36.
```

```
37.             public void IncrementInvocations(){
38.             SetInvocations(Invocations + 1);
39.                   }
40.
41.             public void AddToTotalExecutionTime (long additionalTime){
42.             SetTotalExecutionTime(TotalExecutionTime +
additionalTime);
43.                 }
44.         public void RemoveFromTotalExecutionTime(long time){
45.             SetTotalExecutionTime(TotalExecutionTime - time);
46.                   }
47.     }
48. }
```

MethodStatistic.cs

Grâce à l'attribute [Insert("")], cette classe sera insérée automatiquement dans l'assembly cible. Cela peut également être spécifié dans un fichier XML externe si l'on souhaite éliminer tout couplage entre la classe MethodStatistic et le tisseur AspectDNG.

La classe StatisticManagement s'appuie sur la précédente pour centraliser toutes les informations au fur et à mesure de l'exécution du programme. Notez que ses méthodes ont été rendues multi-thread safe par apposition d'une méta-donnée standard de la plate-forme .NET (**MethodImpl**):

```
01. using System;
02. using System.IO;
03. using System.Diagnostics;
04. using System.Reflection;
05. using System.Xml;
06. using System.Xml.Serialization;
07. using System.Collections;
08. using System.Runtime.CompilerServices;
09. using AspectDNG;
10.
11. namespace DotNetGuru.Aspects.Statistic {
12.     [Insert("")]
13.     public class StatisticManagement {
```

```
14.         // Singleton implementation
15.         private static StatisticManagement m_Instance;
16.         public static StatisticManagement Instance{
17.             get{
18.                 return m_Instance == null ?
19.                     m_Instance = new StatisticManagement() :
20.                     m_Instance;
21.             }
22.         }
23.
24.                 // Fields
25.         private IDictionary m_fieldStats = new Hashtable();
26.         private IDictionary m_methodStats = new Hashtable();
27.
28.             // Getters and Setters
29.         private IDictionary FieldStats{
30.              get { return m_fieldStats; }
31.         }
32.          private IDictionary MethodStats{
33.             get { return m_methodStats; }
34.         }
35.
36.              // Methods
37.         [MethodImpl(MethodImplOptions.Synchronized)]
38.         public void IncrementReadAccess(FieldInfo metadata){
39.             FieldStatistic fs = (FieldStatistic) m_fieldStats[metadata];
40.             if (fs == null){
41.                 fs = new FieldStatistic(metadata);
42.                 m_fieldStats[metadata] = fs;
43.             }
44.             fs.IncrementReadAccess();
45.         }
46.
47.             [MethodImpl(MethodImplOptions.Synchronized)]
48.         public void IncrementWriteAccess(FieldInfo metadata){
49.             FieldStatistic fs = (FieldStatistic) m_fieldStats[metadata];
50.             if (fs == null){
51.                 fs = new FieldStatistic(metadata);
```

```
52.                 m_fieldStats[metadata] = fs;
53.             }
54.             fs.IncrementWriteAccess();
55.         }
56.
57.             [MethodImpl(MethodImplOptions.Synchronized)]
58.         public void
StartIncrementingMethodInvocationAndTime(MethodBase mb){
59.             string signature = GetSignature(mb);
60.             MethodStatistic ms = m_methodStats[signature] as
MethodStatistic;
61.             if (ms == null){
62.                 ms = new MethodStatistic(mb);
63.                 m_methodStats[signature] = ms;
64.             }
65.             ms.IncrementInvocations();
66.             ms.RemoveFromTotalExecutionTime (DateTime.Now.Ticks);
67.         }
68.
69.                 [MethodImpl(MethodImplOptions.Synchronized)]
70.         public void
StopIncrementingMethodInvocationAndTime(MethodBase mb){
71.             string signature = GetSignature(mb);
72.             MethodStatistic ms = (MethodStatistic)
m_methodStats[signature];
73.             ms.AddToTotalExecutionTime (DateTime.Now.Ticks);
74.         }
75.
76.             [MethodImpl(MethodImplOptions.Synchronized)]
77.         public void WriteXml(){
78.             XmlTextWriter writer  = new XmlTextWriter
79.                 ("stats.xml", System.Text.Encoding.Unicode);
80.             writer.Formatting = Formatting.Indented;
81.
82.                 foreach (string name in
GetType().Assembly.GetManifestResourceNames()) {
83.                 if (name.EndsWith("stats.xsl")) {
84.                     StreamReader sreader =   new
```

```
GetType().Assembly.GetManifestResourceStream(name));
85.                                 string stylesheetContents =
sreader.ReadToEnd();
86.                 sreader.Close();
87.
88.                     StreamWriter swriter = new
StreamWriter("stats.xsl");
89.                 swriter.WriteLine(stylesheetContents);
90.                 swriter.Close();
91.                 break;
92.               }
93.             }
94.             writer.WriteProcessingInstruction("xml-stylesheet",
"type='text/xsl' href='stats.xsl'");
95.             writer.WriteStartElement("Stats");
96.             writer.WriteAttributeString("assembly",
GetType().Assembly.GetName().FullName);
97.
98.               ArrayList allTypes = new ArrayList();
99.
100.               writer.WriteStartElement("Fields");
101.             foreach( FieldStatistic fs in FieldStats.Values ){
102.               writer.WriteStartElement("Field");
103.               Type type = fs.MetaData.DeclaringType;
104.               if (! allTypes.Contains(type)){
105.                   allTypes.Add(type);
106.               }
107.               writer.WriteAttributeString("declaringType",
type.FullName);
108.               writer.WriteAttributeString("name",
fs.MetaData.Name);
109.
110.                   writer.WriteAttributeString("readAccess",
fs.ReadAccess.ToString());
111.               writer.WriteAttributeString("writeAccess",
fs.WriteAccess.ToString());
112.               writer.WriteEndElement();
113.             }
```

```
114.            writer.WriteEndElement();
115.
116.               writer.WriteStartElement("Methods");
117.            foreach( MethodStatistic ms in MethodStats.Values ){
118.               writer.WriteStartElement("Method");
119.               Type type = ms.Metadata.DeclaringType;
120.               if (! allTypes.Contains(type)){
121.                  allTypes.Add(type);
122.               }
123.               writer.WriteAttributeString("declaringType",
type.FullName);
124.               writer.WriteAttributeString("signature",
GetSignature(ms.Metadata));
125.               writer.WriteAttributeString("invocations",
ms.Invocations.ToString());
126.               writer.WriteAttributeString("totalExecutionTimeMillis",
(ms.TotalExecutionTime / 10000).ToString());
127.               writer.WriteEndElement();
128.            }
129.            writer.WriteEndElement();
130.
131.               writer.WriteStartElement("Types");
132.            foreach(Type type in allTypes ){
133.               writer.WriteElementString("Type", type.FullName);
134.            }
135.            writer.WriteEndElement();
136.
137.               writer.WriteEndElement();
138.            writer.Close();
139.         }
140.
141.            private string GetSignature(MethodBase mb){
142.            string parametersList = "";
143.            foreach(ParameterInfo pi in mb.GetParameters()){
144.               parametersList += pi.ParameterType;
145.            }
146.
147.               return string.Format("{0}::{1}({2})",
```

```
DeclaringType.FullName, mb.Name, parametersList);
148.                 }
149.     }
150. }
```

StatisticManagement.cs

Le code de la classe d'Aspect se contente d'invoquer les services de la précédente:

```
01. using System;
02. using System.Reflection;
03. using AspectDNG;
04.
05. namespace DotNetGuru.Aspects.Statistic {
06.     public class Aspect {
07.         private const string TargetFields = "* *Base.*::*";
08.         private const string TargetMethods = "* *Base.*::*(*)";
09.
10.             [InlineAfterFieldReadAccess(TargetFields)]
11.         public void AfterFieldRead(){
12.             FieldInfo fi = null;
13.             StatisticManagement.Instance.IncrementReadAccess(fi);
14.         }
15.
16.             [InlineAfterFieldWriteAccess(TargetFields)]
17.         public void AfterFieldWrite(){
18.             FieldInfo fi = null;
19.             StatisticManagement.Instance.IncrementWriteAccess(fi);
20.         }
21.
22.                 [InlineBeforeConstructorCall(TargetMethods)]
23.         [InlineBeforeMethodCall(TargetMethods)]
24.         public void BeforeCall(){
25.             MethodBase mb = null;
26.
StatisticManagement.Instance.StartIncrementingMethodInvocationAndTime(mb);
27.         }
28.
29.             [InlineAfterConstructorCall(TargetMethods)]
```

```
30.         [InlineAfterMethodCall(TargetMethods)]
31.         public void AfterCall(){
32.             MethodBase mb = null;
33.
StatisticManagement.Instance.StopIncrementingMethodInvocationAndTime(mb);
34.         }
35.
36.             [InlineBeforeReturn("*Base.*::Main(*)")]
37.         public void BeforeTheEnd(){
38.             StatisticManagement.Instance.WriteXml();
39.         }
40.     }
41. }
```

Aspect.cs

Les attributes permettent de spécifier le tissage:

- Les invocations des méthodes de StatisticManagement doivent être tissées aux emplacements judicieux (au moment de la lecture/écriture des attributs, avant/après l'invocation de méthodes ou de constructeurs).
- L'écriture du document XML représentant les statistiques globales doit être tissée à la fin de la méthode Main du code cible.

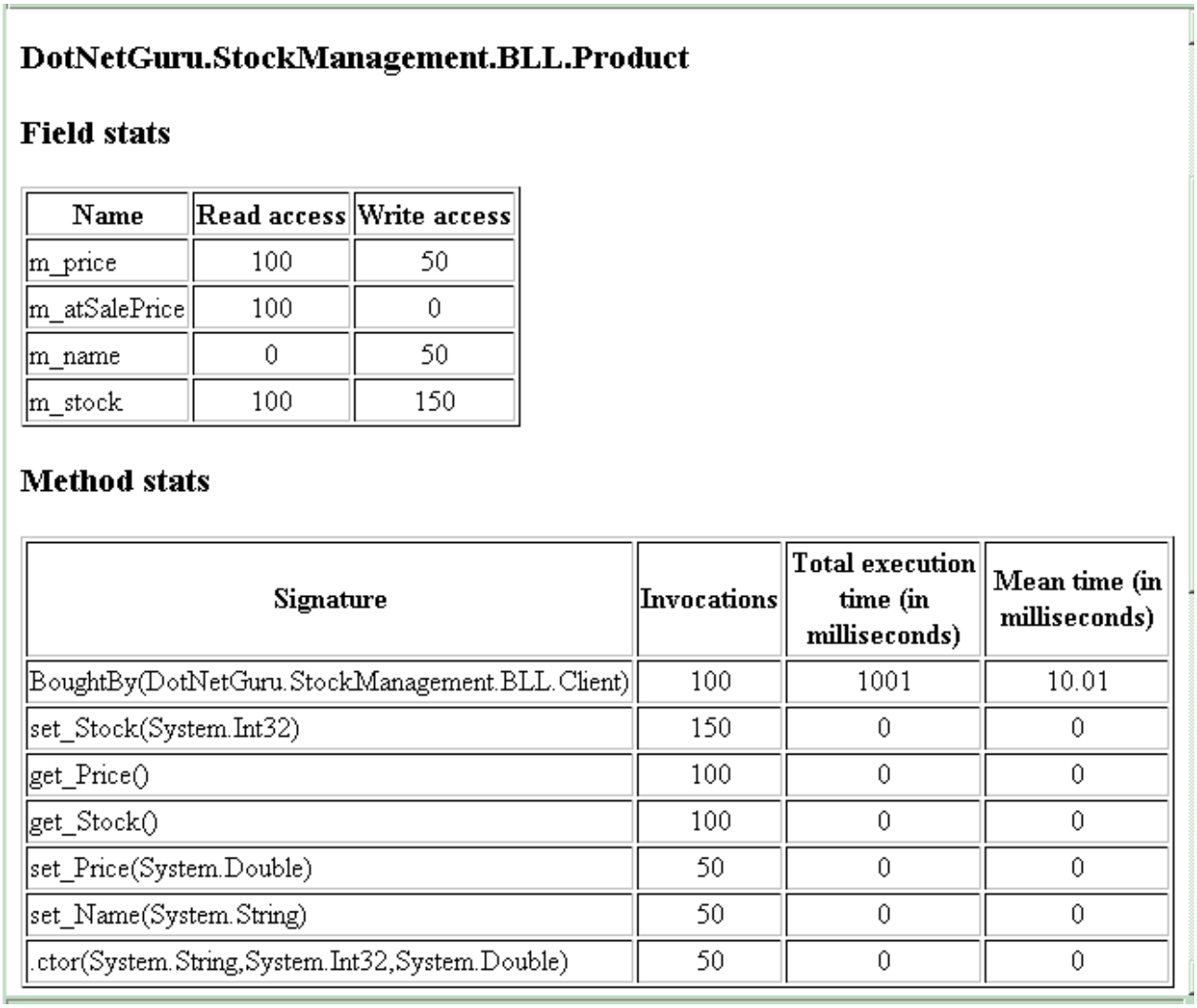
Voici, dans la syntaxe d'AspectDNG, comment décrire ces greffes:

```
01. <?xml version="1.0" encoding="utf-8"?>
02. <!DOCTYPE Advice [
03.     <!ENTITY target-instructions "self::Instruction
04.         [not(starts-with(@operand, 'DotNetGuru.Aspects'))]
05.         [not(starts-with(@operand, 'System'))]">
06. ]>
07. <Advice xmlns=
08.     "http://www.dotnetguru.org/AspectDNG/Advice.xsd">
09.
10.     <Insert
11.         targetCondition="self::Module"
12.         aspectXPath="//Type
13.         [@namespace='DotNetGuru.Aspects.Statistic']
```

```
14.         [not(@name = 'Aspect')]"/>
15.
16.     <InlineAfterFieldReadAccess
17.         targetCondition="target-instructions;"
18.         aspectXPath="//Method[@name='AfterFieldRead']"/>
19.
20.     <InlineAfterFieldWriteAccess
21.         targetCondition="target-instructions;"
22.         aspectXPath="//Method[@name='AfterFieldWrite']"/>
23.
24.           <InlineBeforeConstructorCall
25.         targetCondition="target-instructions;"
26.         aspectXPath="//Method
27.         [@name='BeforeMethodOrConstructorCall']"/>
28.     <InlineAfterConstructorCall
29.         targetCondition="target-instructions;"
30.         aspectXPath="//Method
31.         [@name='AfterMethodOrConstructorCall']"/>
32.
33.           <InlineBeforeMethodCall
34.         targetCondition="target-instructions;"
35.         aspectXPath="//Method
36.         [@name='BeforeMethodOrConstructorCall']"/>
37.     <InlineAfterMethodCall
38.         targetCondition="target-instructions;"
39.         aspectXPath="//Method
40.         [@name='AfterMethodOrConstructorCall']"/>
41.
42.           <InlineBeforeReturn
43.         targetCondition="self::Method[@name='Main']"
44.         aspectXPath="//Method[@name='BeforeTheEnd']"/>
45. </Advice>
```
Advice.xml

Finalement, nous disposons d'un Aspect et de ses classes utilitaires et nous pouvons les greffer sur n'importe quelle application .NET. Si nous réalisons ce tissage sur l'application cible de gestion des stocks (Client - Product), nous obtenons après exécution un fichier de statistiques

stats.xml qu'une feuille de styles XSLT permet de présenter en HTML sous la forme suivante:



DotNetGuru.StockManagement.BLL.Product

Field stats

Name	Read access	Write access
m_price	100	50
m_atSalePrice	100	0
m_name	0	50
m_stock	100	150

Method stats

Signature	Invocations	Total execution time (in milliseconds)	Mean time (in milliseconds)
BoughtBy(DotNetGuru.StockManagement.BLL.Client)	100	1001	10.01
set_Stock(System.Int32)	150	0	0
get_Price()	100	0	0
get_Stock()	100	0	0
set_Price(System.Double)	50	0	0
set_Name(System.String)	50	0	0
.ctor(System.String,System.Int32,System.Double)	50	0	0

Figure 24. Statistiques HTML

Ce petit Aspect peut s'avérer bien utile dans de nombreux contextes et fait donc partie de la bibliothèque d'Aspects réutilisables en cours d'élaboration au sein du projet AspectDNG.

6.2.2. Analyse du code CIL

Par curiosité, nous pouvons également analyser le code du setter de la propriété **Product.Name** après tissage avec AspectDNG:

```
01. namespace DotNetGuru.StockManagement.BLL{
02.     using DotNetGuru.Aspects.Statistic;
```

```
03.
04.         public class Product {
05.         public string Name {
06.             set {
07.                 m_name = value;
08.                 StatisticManagement.IncrementWriteAccess
09.                   ("DotNetGuru.StockManagement.BLL.Product
m_name");
10.                         }
11.         }
12.
13. }
```
Product.cs

Si ce code est nettement plus simple que celui produit par JBossAOP et AspectJ, c'est qu'AspectDNG a fait des choix d'implémentation très différents. Le plus radical est le suivant: au lieu d'instancier un objet représentant l'Aspect greffé sur une zone (ce que font nos deux tisseurs Java), AspectDNG prend le parti de greffer les instructions de l'Aspect directement au sein de la méthode cible!

Cela nous fait gagner une invocation de méthode d'Aspect dans chaque méthode tissée, donc les performances seront meilleures avec ce tisseur .NET, mais c'est au prix d'une limitation importante sur le code des Aspects: aucune méthode d'Aspect ne peut avoir de variable locale sur sa pile! Si l'on a besoin de faire des manipulations complexes dans un Aspect, AspectDNG nous invite plutôt à greffer des méthodes utilitaires dans les classes cibles et à nous appuyer dessus dans les instructions greffées directement dans les méthodes cibles.

Dans le chapitre précédent, nous avions déjà évoqué les critères de choix d'un tisseur d'Aspects. Mais à travers ces détails techniques d'implémentation, vous comprenez certainement mieux ce qui distingue ces outils. Et comme toujours, vous vous rendez certainement compte qu'il n'existe pas d'outil idéal, mais que chacun d'entre eux a établi un ordre de priorité dans les caractéristiques qu'il voulait mettre en avant. A terme, quand nous arriverons en phase de maturité du domaine de la Programmation Orientée Aspects, certains tisseurs proposeront peut-être plusieurs stratégies de tissage: efficace avec limitations dans les

Aspects, ou générique sans limitation...

Chapitre 7.

Conception Orientée Aspects

Depuis le début de ce livre, nous nous sommes laissés guider par la mode et avons évoqué le terme de Programmation Orientée Aspects. Certes les tisseurs d'Aspect travaillent sur le code issu de la programmation (orientée objet ou non, d'ailleurs), mais il est dommage aujourd'hui de ne parler que de programmation sans insister sur le besoin de conception. Parlerions-nous encore de Programmation Orientée Objet sans évoquer la conception et sans nous appuyer sur un langage de modélisation graphique?

Dans ce chapitre-charnière, nous allons prendre un peu de recul par rapport à l'AOP pour examiner les changements impliqués par cette technologie et les besoins qu'elle occasionne. Cela va nous amener à parler de Conception Orientée Aspects et à réfléchir à une syntaxe graphique apte à exprimer nos idées de conception.

7.1. L'AOP crée de nouveaux besoins

Depuis quelques années, la phase de conception prend de plus en plus d'importance dans les projets objet. Adossée à un langage graphique tel que UML, la conception s'avère être:

- Un excellent moyen d'échanger des idées et de discuter des différentes options possibles sans avoir à les implémenter directement dans un langage de programmation, donc d'éviter de perdre un temps précieux.
- Un niveau d'abstraction intéressant pour documenter les choix retenus par un projet.
- Et parfois un moyen de générer une partie du code (cette facette n'est pas toujours utilisée sur les projets).

Il serait dommage que la Programmation Orientée Aspects ne s'appuie pas sur ces acquis dans le domaine de la Programmation Orientée Objet. Autrement dit, il faut sans plus attendre mettre l'accent sur la phase de **Conception Orientée Aspects** ou **AOD** (Aspect Oriented Design) au lieu de n'insister que sur l'AOP. Automatiquement, cela fait naître le besoin d'un langage de modélisation graphique adapté, ainsi que celui d'un catalogue de bonnes pratiques de Conception Orientée Aspects. Pour répondre au premier besoin, nous allons vous proposer une **extension minimale d'UML** adaptée aux Aspects. Et concernant le **catalogue des Aspect Design Patterns**, son élaboration complète dépasse l'ambition de ce livre mais nous essaierons toutefois d'explorer quelques pistes dans le chapitre suivant.

7.2. Concevoir par Aspects

La subtilité, dans le domaine de l'AOD, réside dans le fait que la conception est multiple:

- Le code cible doit être conçu dans les règles de l'art et peut éventuellement s'appuyer sur des (Object) Design Patterns.
- Les Aspects sont eux-mêmes des classes (sauf pour AspectJ) et doivent eux aussi être bien conçus à la mode Orientée Objet. Rien n'interdit qu'un Aspect soit une "Fabrique de Commandes" par exemple, et plus généralement rien n'empêche d'appliquer les (Object) Design Patterns aux Aspects.
- Enfin, il existe plusieurs manières de procéder à la greffe des Aspects sur les classes cibles. C'est dans ce dernier domaine que nous avons le moins de recul et seule l'expérience nous permettra d'ériger certains tissages en Aspect Design Patterns.

Puisque trois types de conception doivent être menés de front, il est plus que probable que l'organisation des projets utilisant l'AOD adopteront une organisation différente des responsabilités à travers les équipes de développement. Aujourd'hui, les phases d'analyse, de conception (fonctionnelle puis technique lors de la conception détaillée) et d'implémentation s'enchaînent au sein d'une itération. Cela impose d'enrichir le modèle d'analyse lors de la conception, mais dès lors il devient difficile de continuer à travailler en tant qu'analyste sur le même modèle dans les itérations suivantes; on peut également maintenir un modèle d'analyse et un modèle de conception en parallèle et assurer la traçabilité entre les deux, mais cela occasionne un travail supplémentaire dont la valeur ajoutée fonctionnelle est faible.

Avec l'AOD, nous avons enfin l'occasion de procéder à une analyse et à une conception purement fonctionnelles, et de concevoir puis de greffer tous les éléments techniques séparément. Cela va probablement améliorer l'isolation des responsabilités sur un projet et augmenter l'agilité des processus. En particulier, il sera plus simple de changer d'orientation technique que si nous reposions uniquement sur des frameworks (sauf peut-être avec les frameworks d'inversion de contrôle).

Certaines organisations de projet sont moins structurées, en particulier quand l'équipe est réduite. Dans ce cas, il n'est pas rare que la programmation prenne une place prépondérante. Dans ce cas, la Programmation Orientée Aspect donnera certainement lieu à un refactoring. Bien sûr, pour l'heure, un tel refactoring devra être fait manuellement. Mais à terme, nous pouvons nous attendre à voir les environnements de développement (en particulier Eclipse et l'AJDT, l'environnement de développement d'AspectJ) proposer des refactoring automatiques tels que *"extraire ces instructions dans une méthode d'Aspect"* ou *"extraire cet attribut dans une classe d'Aspect"*... Soit toute une panoplie de **Refactoring Orientés Aspect**.

7.3. Modéliser graphiquement en Aspects

Il existe aujourd'hui plusieurs langages de modélisation graphique adaptés au monde relationnel, au recueil de besoin, à la traçabilité, etc... Dans le domaine de la modélisation Orientée Objet, UML fait quasiment l'unanimité aujourd'hui même si de nouveaux outils introduiront certainement leur propre notation d'ici quelques mois (Microsoft WhiteHorse, certains outils MDA dédiés à un domaine particulier). C'est donc ce langage que nous avons choisi pour exprimer nos modèles de Conception Orientée Aspects.

Plusieurs initiatives dans le domaine de la recherche proposent déjà leur propre notation pour parler d'Aspects. Avant d'arriver à un consensus et de le faire voter par l'OMG pour qu'il devienne un élément à part entière d'UML, nous verrons cohabiter ces langages un certain temps; mais rassurez-vous, malgré leurs différences syntaxiques, ils expriment bien tous les mêmes idées.

Dans notre expérience de modélisation d'Aspects, nous nous sommes rendu compte du besoin de disposer de deux notations distinctes mais sémantiquement équivalentes pour la Conception Orientée aspect:

- Une notation complète, qui permet de bien distinguer les classes cibles, les Aspects et l'opération de tissage
- Une notation abrégée, qui rend les diagrammes plus concis et représente plutôt les classes une fois le tissage effectué.

La première notation se base sur l'élément de modélisation UML appelé **Collaboration**, à ne pas confondre avec le diagramme du même nom (d'ailleurs, à cause de cette ambiguïté, la Collaboration a été renommée *Communication* en UML 2.0). Une collaboration correspond, dans la norme UML, à **un objectif du système**; dans le cas de l'AOD, **la Collaboration représentera le tissage d'un Aspect sur un ensemble de classes** (sous-entendu: pour implémenter un objectif du système, d'où l'usage d'une Collaboration). L'exemple suivant représente un tissage dans lequel:

- un attribut est introduit dans plusieurs classes cibles,
- le corps d'une méthode est inséré au début de toutes les méthodes des classes cibles.

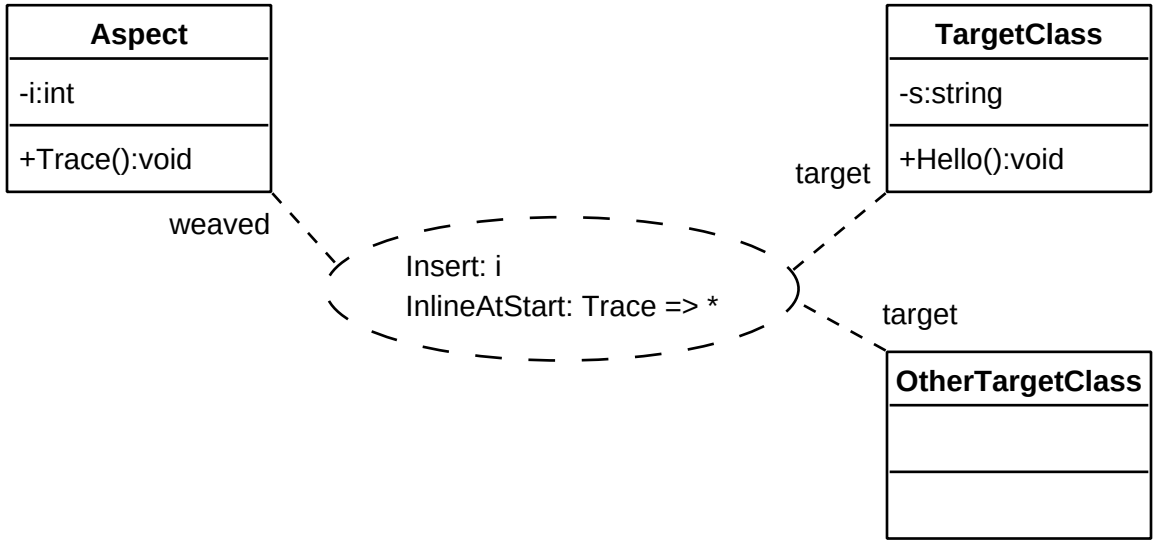


Figure 25. Notation complète

La seconde notation exprime les mêmes idées mais représente une classe après tissage. Elle utilise des stéréotypes pour:

- distinguer les attributs ou les méthodes originels de la classe de ceux qui ont été introduits par tissage,
- qualifier les méthodes dont le corps a été modifié par tissage (au début, avant le retour, avant l'invocation d'autres méthodes, etc...).

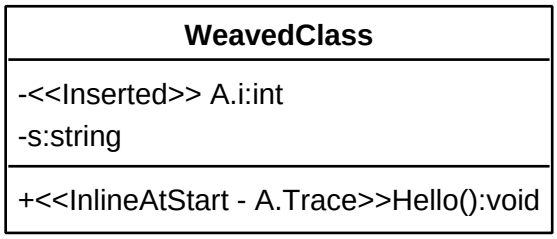


Figure 26. Notation abrégée

La première notation se prête très bien à la séparation des responsabilités et à la maintenance en parallèle d'un modèle purement fonctionnel et d'un modèle d'Aspects réutilisables (techniques ou eux-mêmes fonctionnels). Elle permet de bien visualiser le tissage et

peut donc être utilisée pour comprendre les relations entre Aspects et classes cibles (ce qui est crucial pour établir une traçabilité ou éventuellement une analyse d'impact lorsqu'un Aspect évolue).

La seconde notation sera plutôt utilisée en conception détaillée sur un modèle très localisé, pour bien comprendre sur quels éléments peut se reposer une classe ou un petit ensemble de classes.

Bien sûr, si ces notations venaient à être standardisées, il faudrait que les outils de modélisation d'Aspects proposent de passer automatiquement d'une notation à l'autre.

Dans les chapitres suivants, nous nous efforcerons d'expliquer l'usage de l'AOD en passant par des modèles UML enrichis plutôt que par des exemples systématiques de code. Plus parlants, puisque graphiques, ces modèles offriront également l'avantage d'être indépendants des tisseurs d'Aspects. Ils offrent un niveau d'abstraction supplémentaire, indispensable à l'élaboration d'une bibliothèque de Patterns Orientés Aspect.

Chapitre 8.

Tout est Aspect

Un certain nombre d'usages typiques commencent à être connus dans le domaine de la Conception et de la Programmation Orientées Aspect. Par exemple, certains Design Patterns du GOF peuvent très bien être implémentés sous forme d'Aspects standard. C'est le cas de l'Observer et du Visiteur par exemple. Nous allons donc commencer ce chapitre par la transformation de ces Patterns en Aspects.

D'autres besoins, souvent techniques, se prêtent bien à une implémentation sous forme d'Aspects. Le reste du chapitre en explorera certains mais ce sera à vous, ou plutôt à nous tous dans les années qui viennent, de faire en sorte que ces exemples (ainsi que de tous les autres qui seront tirés de nos expériences respectives) deviennent une véritable bibliothèque de bonnes pratiques de Conception Orientée Aspect: une bibliothèque d'**Aspect Design Patterns**.

8.1. Pattern Observer

Comme nous l'avons dit de manière un peu provoquante dans un chapitre précédent, les Design Patterns sont contre nature, c'est-à-dire qu'ils ne devraient pas polluer le modèle des objets métier. L'AOD est l'outil qui nous manquait pour parvenir à dissocier complètement les Patterns du modèle métier, il est donc logique que les passionnés des Aspects se soient déjà penchés sur la question. En particulier, il faut citer l'excellente initiative de Jan Hannemann qui a publié une implémentation des Design Patterns du GOF basée sur AspectJ (disponible à l'adresse: *http://www.cs.ubc.ca/~jan/AODPs*).

Pour nous familiariser avec la re-conception des Patterns sous forme d'Aspects, commençons par l'Observateur/Observable. Comme vous le savez ce Pattern permet de mettre en place un mécanisme d'abonnement-publication d'événements entre un sujet observable et d'autres objets intéressés par les changements d'état du sujet: les observateurs.

Ce Pattern a un caractère systématique: il s'agit toujours de gérer une liste d'observateurs dans la classe de l'observable (ainsi que l'ajout et la suppression d'observateurs dans cette liste), puis de notifier tous les observateurs. Voici donc ce à quoi il pourrait ressembler sous forme d'Aspect:

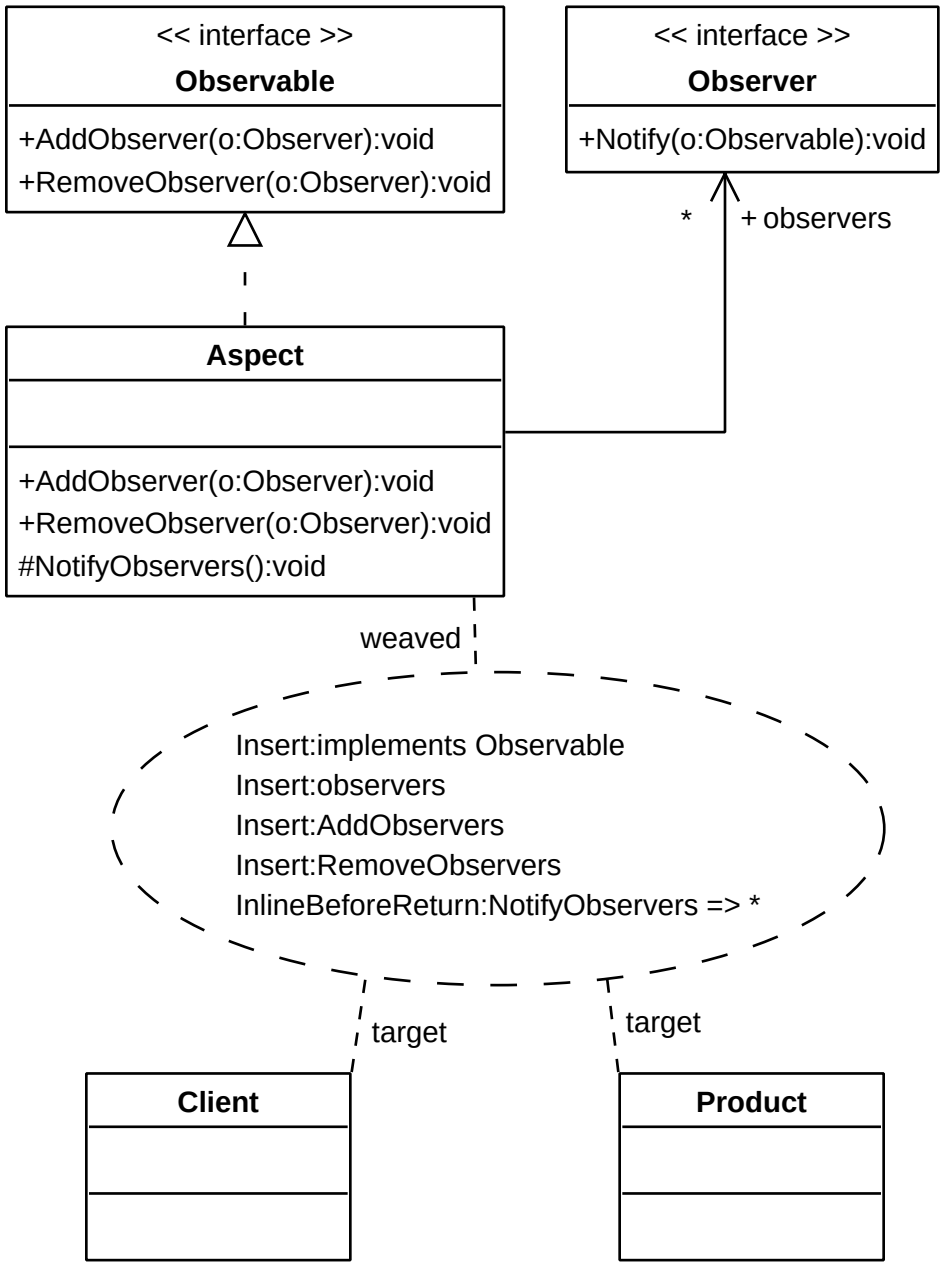


Figure 27. Observateur - Observables

Puisque c'est notre premier exemple de ce type, prenons quelques instants pour le commenter.

La complexité technique de la gestion des observateurs a été externalisée dans une classe d'Aspect. La liste des observateurs ainsi que les méthodes d'ajout et de suppression peuvent ensuite être introduites dans toutes les classes que nous souhaitons rendre observables.

D'autre part, une dernière méthode permet de notifier tous les

observateurs du fait que l'objet courant a été modifié. Son corps sera introduit à la fin de chaque méthode des observables.

Enfin, comme les observateurs implémentent leur interface, utilisée comme type apparent dans les méthodes d'ajout et de suppression précédentes, il en va de même pour les observables, puisqu'ils sont passés en paramètre de la méthode de notification. Il faut donc aussi que notre tissage introduise sur chaque classe cible la relation d'implémentation de l'interface Observable.

Ainsi, il est possible de rendre n'importe quelle classe observable, après coup, sans qu'elle en ait conscience ni qu'elle ait à prévoir quoi que ce soit!

Cela semble trop beau pour être vrai. Quelle est la contre-partie? En réalité, tout réside dans la finesse de la notification: le tissage précédent est un peu grossier car il greffe la notification à la fin de chaque méthode, alors que certaines d'entre elles n'ont aucun impact sur l'état de l'objet observable. Il faudrait donc ne greffer la notification qu'à la fin des méthodes qui modifient effectivement un ou plusieurs attributs. Malheureusement, très peu de tisseurs sont suffisamment fins pour permettre ce type de greffe chirurgicale. A notre connaissance, seul AspectDNG permet de le faire avec la syntaxe XPath suivante: **//Method[Instruction/@code ='stfld']**.

8.2. Pattern Visiteur

En début de livre, nous avions évoqué ce Design Pattern dans le cadre de la construction d'un document XML au fil du parcours d'un graphe d'objets métier. Reprenons cet exemple et adaptons-le à nos classes métier Client et Product.

Tout d'abord, remettons-nous ce Pattern en tête: afin de découpler un graphe structuré d'objets métier de l'objet, plus technique, qui doit le parcourir, il suffit de faire en sorte que chaque objet du graphe (appelons-les "les visités") accepte un visiteur et invoque sur celui-ci une méthode dédiée (**Visit()**) dont le paramètre correspond au type de l'objet visité.

La portion récurrente de ce Pattern concerne les objets visités: chacun d'entre eux doit implémenter une interface **Visitable** qui les oblige à implémenter la méthode **Accept(v:Visitor)**. De plus, le corps de cette méthode Accept(Visitor v) est toujours le même: **v.Visit(this);**

Essayons de re-concevoir ce Pattern en Aspect:

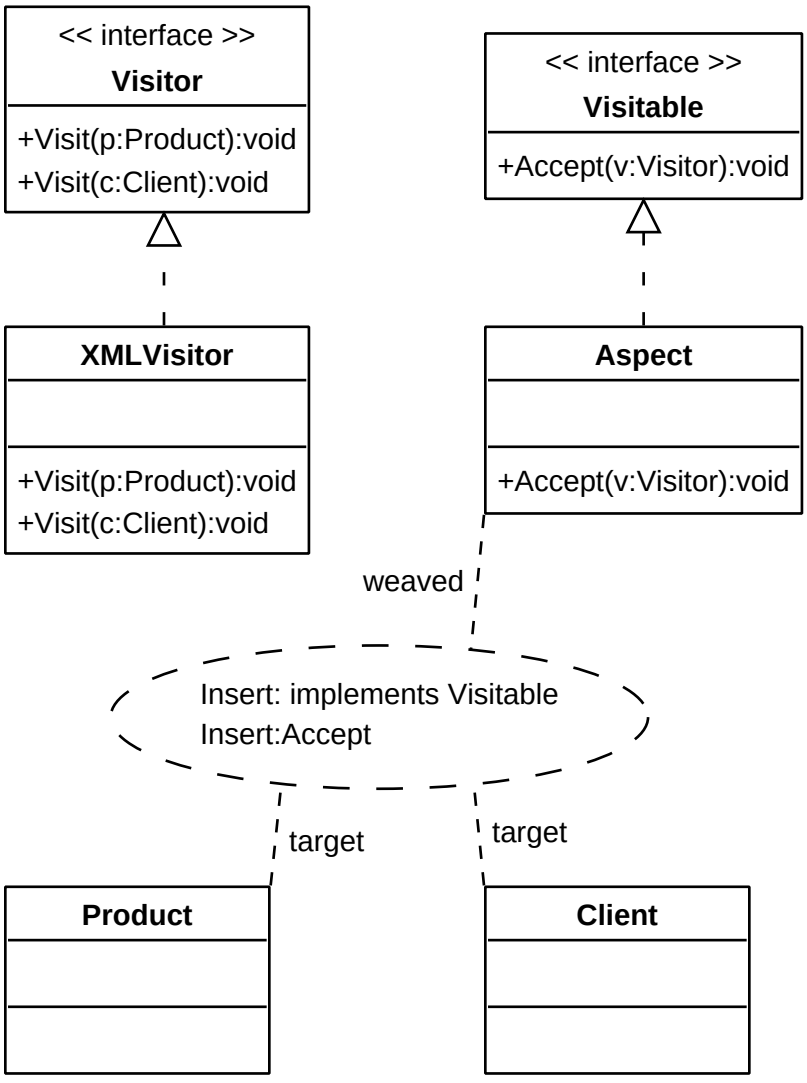


Figure 28. Visiteur

Visiblement, ce deuxième Pattern est lui aussi bien adapté pour le tissage. Aucun objet visité n'a conscience d'être traversé par un visiteur!

Mais à y regarder de plus près, notre Aspect a plusieurs limitations. La plus grave est qu'il est incomplet: il ne traite que les cas très simples dans lesquels les objets sont des éléments terminaux du graphe, c'est-à-dire quand ils ne contiennent pas d'objets dépendants (ne parlons pas des références cycliques entre objets).

Le deuxième problème vient du Pattern Visiteur lui-même. Celui-ci exige un couplage fort entre l'ensemble des types d'objets visités et la liste des surcharges méthodes du visiteur que l'on retrouve non seulement dans la classe **XMLVisitor** mais également dans l'interface **Visitor**. Vous me direz que c'est normal puisque c'est l'objectif même de

ce Pattern que d'avoir une méthode spécifique pour traiter (ici pour représenter en XML) chacun des objets visités. Certes, mais cela induit un problème de maintenabilité lorsque nous ajoutons (ou supprimons) un objet métier du modèle.

Pour plus de souplesse, vous souhaiterez parfois simplifier, voire éliminer l'interface Visitor du Pattern précédent. Examinons l'Aspect Pattern suivant:

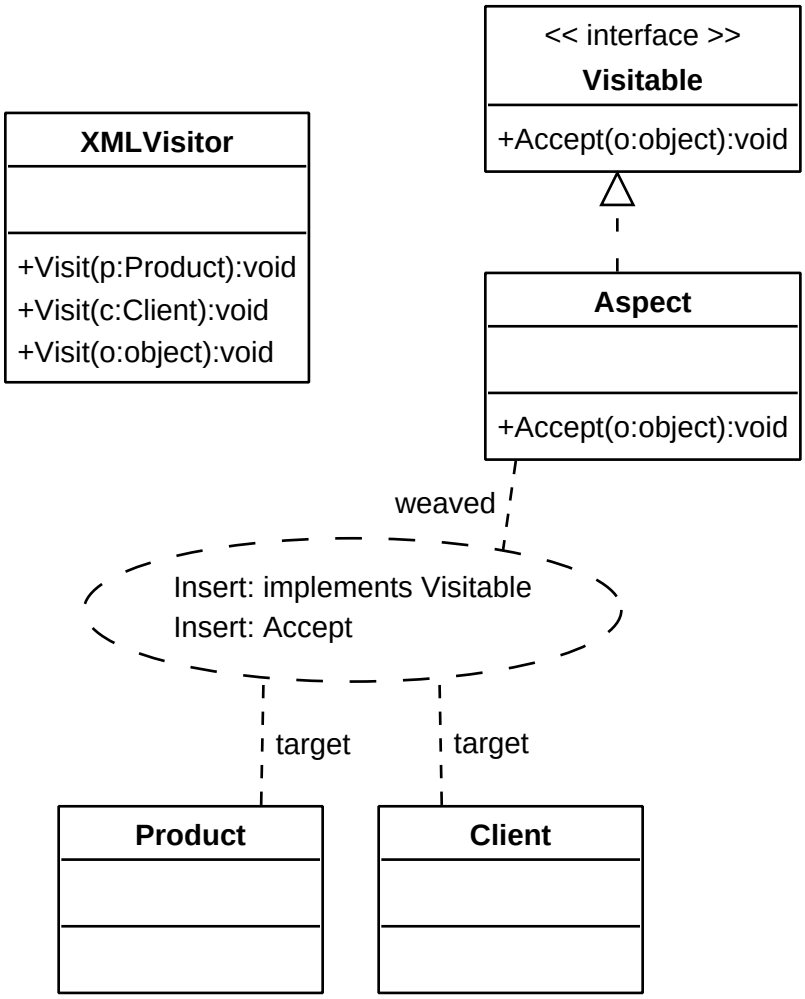


Figure 29. Visiteur générique

Bien entendu, la finesse réside dans l'implémentation de la méthode **Accept(o:object)** que nous tissons sur chaque objet visité. Celle-ci repose sur l'introspection (ou réflexion, disponible en Java comme en .NET) pour invoquer sur le visiteur la bonne surcharge de méthode, celle qui prend en paramètre le type de l'objet courant.

Du côté du visiteur, l'interface a disparu, mais on voit apparaître une

nouvelle méthode **Visit(o:object)** qui joue le rôle de garde-fou: si par hasard le visiteur traversait un objet pour lequel il n'a pas de surcharge, cette méthode générique serait appelée et le visiteur pourrait ainsi décider soit d'interrompre son parcours, soit de continuer sans tenir compte de cet objet inconnu, ou encore de lui appliquer un traitement générique.

De même, dans le corps de la méthode Accept(o:object), rien n'empêche l'Aspect de faire de l'introspection sur son objet cible pour découvrir quels sont les attributs de type référence (vers un objet, une collection...) et de programmer ainsi un parcours récursif et générique.

Cette seconde implémentation du Visiteur est bien plus souple que la précédente, mais en contre-partie, puisqu'elle repose sur l'introspection, elle sera moins efficace et moins robuste: il faudra attendre l'exécution de notre application pour s'apercevoir que nous avons oublié une surcharge de méthode dans notre visiteur (à condition, cela s'entend, d'imprimer une trace dans la méthode garde-fou).

8.3. Patterns Décorateur et Proxy

Ne nous méprenons par sur l'objectif de cette section: la Programmation Orientée Aspects n'est pas un outil de génération de code et n'est donc pas vouée à produire automatiquement un Décorateur ou un Proxy qui s'interposerait entre une classe et ses utilisateurs.

Par contre, à supposer que nous développions manuellement une classe qui joue le rôle de Proxy, l'AOP est l'outil rêvé pour substituer les références aux instances d'une classe par des références aux proxies associés.

Prenons l'exemple de la classe **EntryPoint** qui utilise **Product** et **Client**. Remettons-nous son corps en mémoire:

```
01. namespace DotNetGuru.StockManagement{
02.     using System;
03.     using System.Collections;
04.     using DotNetGuru.StockManagement.BLL;
05.
06.     public class EntryPoint{
07.         public static void Main(){
08.             // 100 Clients buy Products among
09.              // 50 available in the system
10.
11.             ArrayList clients = new ArrayList();
12.             for(int i=0; i<100; i++){
13.                 clients.Add
14.                     (new Client
15.                         ("client-" + i, 1000 + 10 * i));
16.             }
17.
18.             ArrayList products = new ArrayList();
19.             for(int i=0; i<50; i++){
20.                 products.Add
21.                     (new Product
22.                         ("product-" + i, 100, 200 + 10 * i));
23.             }
24.
```

```
25.             for(int i=0; i<clients.Count; i++){
26.                 Client c = clients[i] as Client;
27.                 Product p = products[i % products.Count]
28.                     as Product;
29.                 p.BoughtBy(c);
30.             }
31.         }
32.     }
33. }
```

EntryPoint.cs

Imaginons d'autre part que nous ayons développé deux Proxies (**ProductProxy** et **ClientProxy**) qui pourraient déléguer les invocations de méthodes à distance, ou cacher le résultat de certaines méthodes pendant quelques secondes, ou encore jouer le rôle de **Poids Mouche** sur un objet fréquemment référencé. Sans l'aide de l'AOP, nous serions obligés d'instancier manuellement ces proxies dans le code de la classe EntryPoint qui pourrait alors ressembler à ceci:

```
01. namespace DotNetGuru.StockManagement{
02.     using System;
03.     using System.Collections;
04.     using DotNetGuru.StockManagement.BLL;
05.
06.     public class EntryPoint{
07.         public static void Main(){
08.             // 100 Clients buy Products among
09.              // 50 available in the system
10.
11.             ArrayList clients = new ArrayList();
12.             for(int i=0; i<100; i++){
13.                 clients.Add
14.                     (new ClientProxy
15.                         (new Client
16.                             ("client-" + i, 1000 + 10 * i)));
17.             }
18.
19.             ArrayList products = new ArrayList();
20.             for(int i=0; i<50; i++){
```

```
21.             products.Add
22.                 (new ProductProxy
23.                     (new Product
24.                         ("product-" + i, 100, 200 + 10 * i)));
25.         }
26.
27.         for(int i=0; i<clients.Count; i++){
28.             Client c = clients[i] as Client;
29.             Product p = products[i % products.Count]
30.                 as Product;
31.             p.BoughtBy(c);
32.         }
33.     }
34. }
35. }
```

EntryPointProxyWithoutAOP.cs

Généralement, pour éviter d'arriver à une telle situation, nous utilisons une **Fabrique** éventuellement **Abstraite** chargée d'instancier les Product et les Clients et dans laquelle il est trivial d'instancier aussi les ProductProxy et ClientProxy et de les renvoyer aux classes utilisatrices qui n'y verraient que du feu. Mais passer par une Fabrique alourdit la syntaxe des classes utilisatrices et représente un point d'extensibilité du logiciel qu'il faut avoir prévu dès le début. Ce qui n'est pas toujours simple.

Envisageons la même situation mais avec l'angle de l'AOP: l'opérateur **new** invoque les constructeurs de nos classes, il n'y a donc aucune raison que l'on ne puisse pas tisser un Aspect autour de cette invocation afin de renvoyer aux classes utilisatrices une instance de ClientProxy à la place de Client, une instance de ProductProxy à la place de Product, etc... Voici le diagramme de conception correspondant à ce tissage:

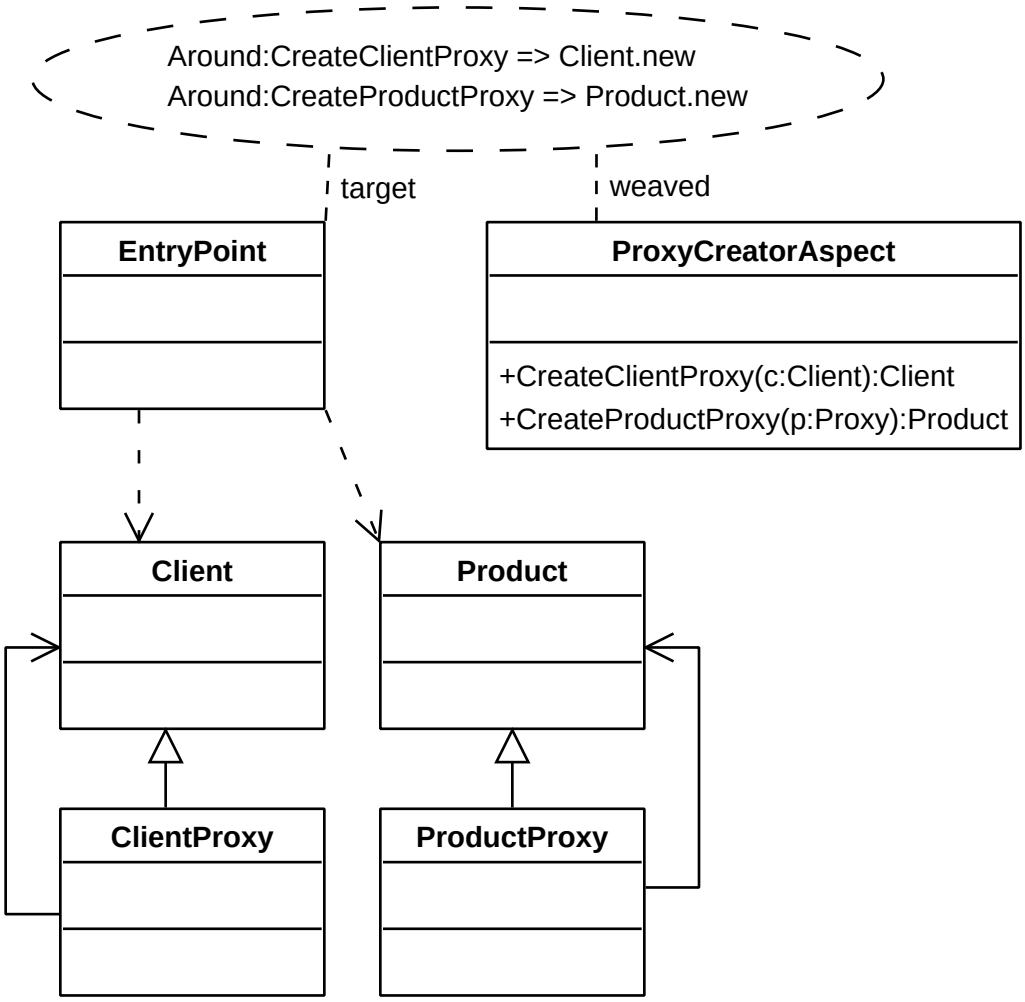


Figure 30. Substitution d'objet par son Proxy

Grâce à un tel Aspect, les classes utilisatrices peuvent continuer à instancier leurs objets de manière naturelle, sans passer par une Fabrique, et utiliser sans s'en rendre compte des Proxies intermédiaires vers leurs objet cibles.

Le même principe s'applique bien sûr aux Design Patterns proches: le **Décorateur**, le **Poids Mouche** et la **Chaîne de Responsabilité**.

8.4. Sécurité

Soyons humbles, nous n'allons bien évidemment pas couvrir tous les champs du domaine de la sécurité dans cette section! Nous ne traiterons que de la protection de l'invocation d'une méthode dans une application Orientée Objet, ce qui est effectivement très restrictif, mais illustre bien le rôle que peuvent jouer les Aspects dans ce domaine.

8.4.1. Sécurité dynamique

La technique la plus évidente pour empêcher tout utilisateur non accrédité d'invoquer une méthode est de tester son identité ou son rôle avant d'effectuer quelque traitement que ce soit. Ces tests peuvent être réalisés:

- dans le code des classes métier ou applicatives elles-mêmes (ce qui bien entendu est à éviter),
- dans un framework technique qui intègre ces tests et lit son paramétrage dans un fichier ou dans un serveur de configuration (c'est le fonctionnement nominal des applications Web ou EJB dans la plate-forme J2EE, les informations de configuration étant externalisées dans les descripteurs de déploiement web.xml ou ejb-jar.xml et dans un annuaire JNDI),
- dans un (ou plusieurs) Aspect.

L'approche par Aspects est surtout intéressante dans le cadre d'une application qui ne reposerait pas déjà sur un Framework technique. Sans quoi, elle ferait double emploi.

Voici un Aspect très simple de vérification d'identité:

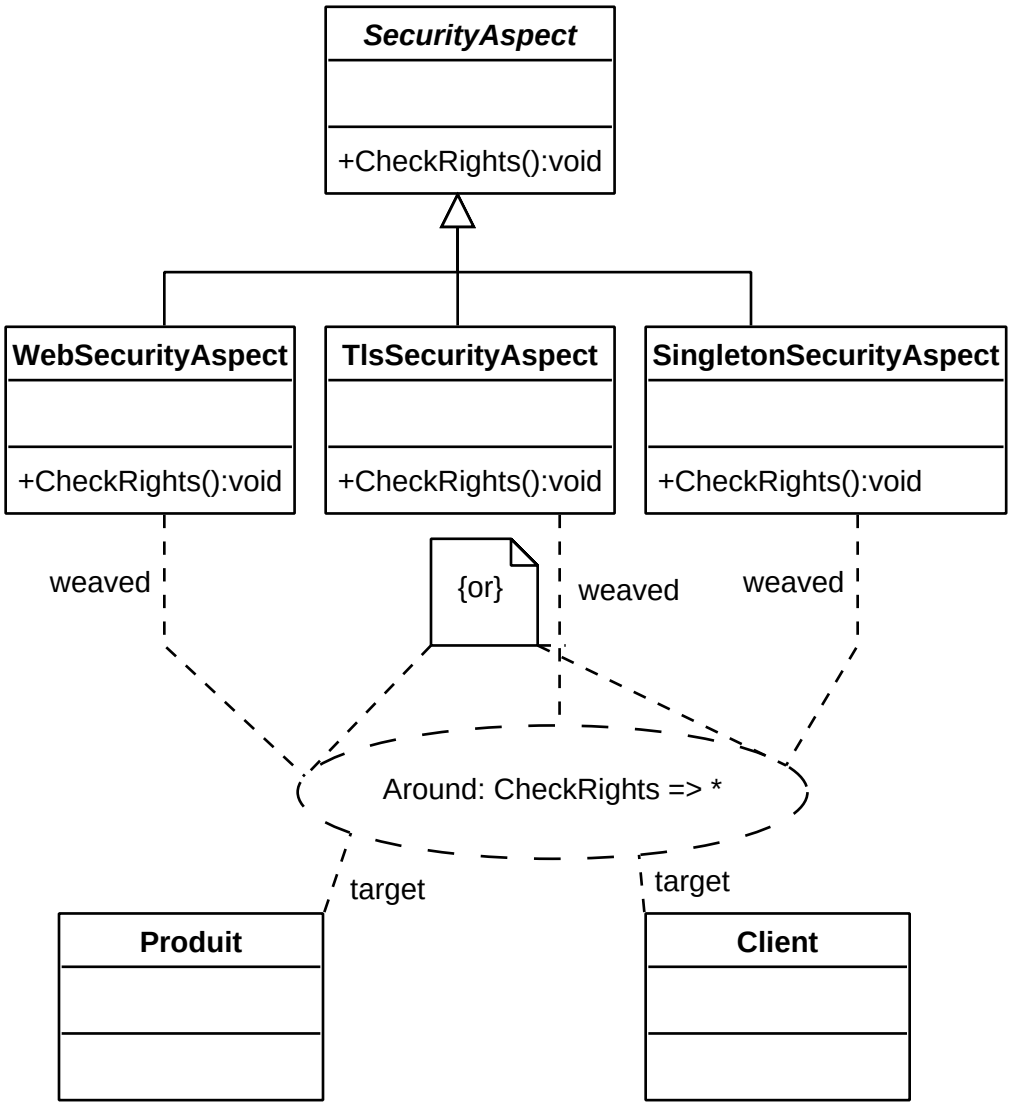


Figure 31. Aspect garant des droits

Quelques explications s'imposent: il n'existe pas de mécanisme universel permettant de récupérer l'identité de l'utilisateur. Il n'est pas rare que les applications ré-implémentent leurs propres classes représentant l'identité, le rôle et les droits associés à chaque utilisateur. C'est pour cela que nous avons représenté un Aspect abstrait, chargé de vérifier les droits d'un utilisateur et voué à être tissé sur toutes les classes à protéger; mais l'implémentation concrète de la récupération de l'identité et du test d'accréditation est déléguée aux Aspects fils.

Par exemple, dans une application Web, l'identité de l'utilisateur est souvent stockée dans la **Session HTTP**. Le WebSecurityAspect pourrait donc récupérer cette identité en passant par les classes techniques liées au Web (à adapter selon le framework) et bloquer ou non l'invocation de

la méthode visée.

Dans une application de type client riche, cette même identité pourrait simplement être portée par un **Singleton**.

Et dans le cas d'une application non Web mais tout de même multi-utilisateurs (une application .NET Remoting par exemple), on pourrait imaginer que les objets distribués (ou les proxies côté serveur qui exposent les objets distribués) placent l'identité de l'utilisateur dans le **TLS** (Thread Local Storage), ce qui peut faire l'objet d'un premier Aspect. Le TlsSecurityAspect prendrait ainsi sa décision en fonction de l'identité trouvée dans ce même TLS, qui est unique pour le Thread courant.

Quel que soit le moyen retenu pour stocker l'identité de l'utilisateur courant, notre Aspect peut aisément décider de laisser l'invocation de méthode se poursuivre ou lever une exception de sécurité dans le cas où l'utilisateur n'est pas accrédité pour cette méthode.

Remarque: notez qu'une alternative à l'Aspect précédent serait d'intercaler un **Décorateur** entre une classe et ses utilisatrices (par la technique vue dans la section précédente), qui aurait justement la responsabilité de propager les invocations de méthodes ou de lever une exception de sécurité. L'inconvénient serait une certaine lourdeur car le Décorateur doit impérativement définir toutes les méthodes de l'objet décoré, puis déléguer l'invocation le cas échéant. L'autre contre-partie serait la multiplication du nombre d'objets et le surcoût d'une invocation de méthode supplémentaire, que l'on ne subit pas ici.

8.4.2. Sécurité statique

Dans la section précédente, le même code exécutable pouvait être sollicité par plusieurs utilisateurs, d'où la nécessité de tisser un Aspect qui prend ses décisions à l'exécution.

Imaginons maintenant une application, par exemple un client riche (Java Swing, SWT ou Windows Forms), qui serait utilisée soit par un utilisateur standard, soit par un administrateur. Dans cette situation il suffirait:

- de développer normalement l'application complète, destinée aux administrateurs,
- puis de tisser un Aspect qui lève une exception dès qu'une fonctionnalité avancée est déclenchée. Le résultat de ce tissage est l'application destinée aux utilisateurs standard.

En effet, aucune décision n'est à prendre à l'exécution: dans l'application "utilisateur", il faut systématiquement bloquer les invocations de fonctionnalités avancées et lever une exception de sécurité. Alors que dans l'application "administrateur", aucune contrainte de sécurité n'est à appliquer.

Enfin, dans la couche de présentation de l'application "utilisateur", un autre Aspect pourrait récupérer l'exception de sécurité et afficher une boîte de dialogue disant: "Cette fonctionnalité n'est pas disponible. Seul un administrateur peut y avoir accès.". Aucun problème, donc.

Mais allons plus loin: transposons cet exemple dans le cadre d'une application commerciale dont une version limitée en fonctionnalités pourrait être distribuée gratuitement par son éditeur. Le problème est le même que précédemment, à ceci près que le code correspondant aux fonctionnalités avancées ne doit surtout pas se trouver dans l'application exécutable gratuite. En effet, en Java comme en .NET, il est beaucoup trop simple de décompiler une application, d'en modifier le code et de le recompiler de manière à avoir finalement accès à la totalité du logiciel.

Dans ce cas, déclencher une exception au début des méthodes avancées ne suffit plus. Il faut que le tissage **remplace** le corps de la méthode par le lancement de cette exception. Ainsi, même un développeur expérimenté ne pourrait pas "déverrouiller" cette application puisque le code avancé ne se trouve plus dans l'application exécutable finale.

Le remplacement ou la suppression de code n'est pas (encore) permis par beaucoup de tisseurs. Mais prenons l'exemple d'AspectDNG, avec lequel il suffit d'écrire:

<Delete targetCondition= "self::Method[@name='MethodeASupprimer']"/>

Ce mécanisme s'applique non seulement aux méthodes, mais aussi aux

classes que vous souhaiteriez voir disparaître de l'application finale.

8.5. Programmation par contrats

Bien comprendre le contrat d'utilisation d'une classe est primordial pour assurer la qualité d'un logiciel. Mais où se trouve exactement la description formelle de ce contrat?

Dans le code source, certes, on trouve de bonnes pistes pour comprendre les conditions d'utilisation des méthodes d'une classe. Mais nous n'y avons pas toujours accès. De plus, le développeur de ces classes a certainement fait des hypothèses qui ne se retrouvent pas dans le code, sauf peut-être sous forme de commentaires dans le meilleur des cas.

Très souvent, c'est dans la documentation ou dans les tests unitaires accompagnant les classes que nous en trouvons les conditions d'usage. Ces dernières ne peuvent donc être exploitées que par les développeurs des applications utilisatrices, de manière informelle (lecture, copier-coller).

Pourtant, certains langages tels que Eiffel offrent directement la possibilité d'intégrer les contrats aux classes qui les implémentent. Pour les langages moins puissants tels que Java ou C#, des frameworks ou des extensions au langage permettent de faire de même (*jContractor* et *JMSAssert* pour le langage Java par exemple).

L'approche par Aspects permet elle aussi d'outiller les préceptes de la Conception par Contrat, mais sans introduire de dépendance entre les classes métier et un quelconque framework, ni nécessiter de compilateur spécifique à une extension du langage Java ou C#. Cela va-t-il donner un nouveau souffle à cet outil inestimable qu'est le Contrat pour la qualité logicielle, initialement introduit par Bertrand Meyer? Seul l'avenir nous le dira... D'ici là, voyons ensemble **comment tisser nos contrats**.

Le contrat d'une classe est constitué par:

- **Un ensemble d'invariants** qui doivent être vérifiés par tous les états cohérents des objets (c'est-à-dire quand aucune méthode n'est en cours d'exécution sur un objet donné, puisque dans ce cas, son état est variable et peut temporairement violer un ou plusieurs invariants).

- **Un ensemble de pré-conditions** associées aux méthodes de la classe qui peuvent porter sur les attributs de l'objet visé ou sur les paramètres d'invocation des méthodes. Avant que l'on ait le droit d'invoquer une méthode, il faut que toutes ses pré-conditions soient réunies. Sans quoi, l'invocation n'a pas de sens et doit être interdite.
- **Un ensemble de post-conditions**, également associées aux méthodes. Cette fois, il s'agit d'une garantie offerte par la méthode qui s'engage à respecter ses post-conditions et à laisser l'objet courant dans un certain état.

Vous aurez traduit en termes Orientés Aspect à la lecture des lignes précédentes que:

- un premier Aspect doit vérifier les **invariants** avant et après l'exécution des méthodes de la classe,
- d'autres Aspects doivent vérifier les **pré-conditions** de chaque méthode, avant son exécution,
- les derniers Aspects doivent vérifier les **post-conditions** de chaque méthode, après son exécution.

Considérons la classe Product et sa méthode **BoughtBy(c:Client)**. Voici comment lui greffer son contrat:

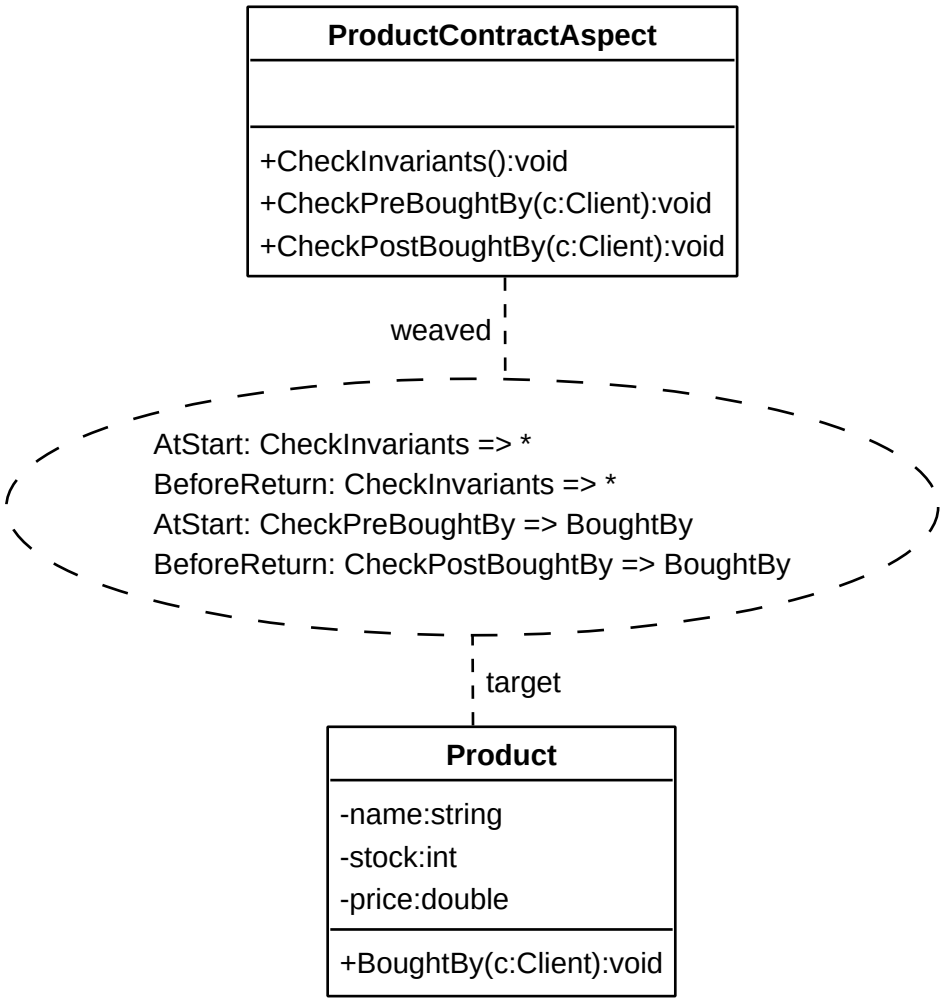


Figure 32. Contrat Orienté Aspect

Et sans descendre jusqu'au niveau du code, disons simplement que:

- La méthode **CheckInvariants()** vérifie que le nom d'un produit n'est jamais *null*, que son prix est toujours supérieur ou égal à 0 et qu'il en va de même pour son stock.
- **CheckPreBoughtBy(c:Client)** vérifie que le client a suffisamment d'argent pour acheter ce produit et que le stock est supérieur ou égal à 1.
- **CheckPostBoughtBy(c:Client)** vérifie que le client n'a pas un compte en banque négatif et que le stock a bien été décrémenté d'une unité.

A nouveau, l'avantage de l'AOD par rapport à d'autres techniques est qu'elle permet de concevoir, de développer et de maintenir les Contrats

en parallèle des classes métier elles-mêmes. En termes d'organisation de projet, cela peut amener à mieux affecter les responsabilités des concepteurs-développeurs et à paralléliser davantage le travail des équipes.

8.6. Paramétrage uniforme

Les langages de programmation positionnent souvent les attributs des objets à une valeur par défaut en fonction de leur type: **0** pour les numériques, **false** pour les booléens, **null** pour les références vers un objet...

Mais il peut se trouver que dans certaines situations, ces valeurs par défaut ne conviennent pas. Si par exemple vous souhaitez affecter une chaîne vide ("") et non pas une référence nulle à vos attributs de type **String**, il n'y a qu'un recours: initialiser manuellement tous ces attributs soit dès leur déclaration, soit dans les constructeurs de la classe associée.

Si votre projet compte de nombreuses classes, et que beaucoup d'entre elles portent des attributs qui sont dans ce cas, d'une part ce sera fastidieux à développer et à maintenir, et d'autre part vous n'êtes pas à l'abri d'un oubli qui se paiera en temps de recherche d'anomalies. Au lieu de cela, utilisez donc la puissance des Aspects!

Il suffit en effet de **tisser l'initialisation des attributs** soit au moment de leur premier accès (en lecture ou en écriture), soit dès le constructeur. Dans le premier cas, rien de plus simple: une méthode d'Aspect sera tissée avant chaque accès aux attributs de type String et affectera la chaîne vide si l'attribut est encore *null*. L'inconvénient est un certain surcoût à l'exécution puisque le test de nullité des attributs aura lieu à chaque accès, ce qui peut vite devenir préjudiciable aux performances globales de l'application.

Pour faire de même dans le constructeur, il suffirait de tisser une méthode d'Aspect qui utiliserait l'introspection pour découvrir quels sont les attributs de type **String** dans la classe cible et qui les affecterait automatiquement à la chaîne vide. Là aussi, nous payons un petit surcoût par rapport à une initialisation "en dur" dans le code de la classe, mais l'intéressant avec cette approche, c'est qu'elle se généralise à tous les types d'attributs, qu'ils soient primitifs ou objets. Et au final, les valeurs "par défaut" des attributs pourront bel et bien être uniformes, à un niveau que l'on peut choisir: la classe, l'espace de nommage, un ensemble d'espaces de nommage ou encore le projet entier.

8.7. Services et exceptions techniques

L'un des principaux objectifs de l'AOP est de découpler le code applicatif des problématiques techniques. Jusqu'à présent, nous avons surtout travaillé sur les objets métier. Mais l'AOP rend également de nombreux services aux classes plus techniques, et pourquoi pas, aux classes d'un framework technique.

Prenons l'exemple récurrent d'une classe qui aurait besoin d'une connexion à une base de données relationnelle. Elle va utiliser les API **ADO.NET** ou **JDBC** pour récupérer (dans le cas d'un pool) ou instancier une connexion, puis elle interagira avec la base et enfin, elle devra fermer proprement cette connexion. Entre-temps, si une exception technique survient, il faudra l'intercepter, généralement annuler la transaction en cours et propager l'exception à l'appelant pour qu'il soit informé du fait que la méthode n'a pas pu aboutir.

Voici un exemple typique d'une classe C# utilisant ADO.NET (sans Aspects):

```
01. namespace DatabaseAccess {
02.     using System;
03.     using System.Data;
04.     using System.Data.SqlClient;
05.
06.     class DatabaseInteractionWithoutAOP {
07.         public void SampleInteraction(){
08.             IDbConnection cnx = null;
09.             IDbTransaction tx = null;
10.             try{
11.                 cnx = new SqlConnection
12.                     (@"Initial Catalog=Northwind;
13.                         Data Source=DNG;User Id=sa;");
14.                 cnx.Open();
15.                 tx = cnx.BeginTransaction
16.                         (IsolationLevel.ReadCommitted);
17.                 IDbCommand cmd = cnx.CreateCommand();
```

```
18.             cmd.Transaction = tx;
19.             cmd.CommandText = "select * from products";
20.             IDataReader reader = cmd.ExecuteReader();
21.             while (reader.Read()){
22.                 string name = reader["productName"] as string;
23.                 Console.WriteLine(name);
24.             }
25.             tx.Commit();
26.         }
27.         catch(Exception e){
28.             tx.Rollback();
29.             // exception management
30.         }
31.         finally{
32.             try{
33.                 cnx.Close();
34.                     }
35.             catch(Exception e){
36.                 // exception management
37.             }
38.                 }
39.     }
40.
41.     static void Main(string[] args) {
42.         DatabaseInteractionWithoutAOP dbi =
43.             new DatabaseInteractionWithoutAOP();
44.         dbi.SampleInteraction();
45.     }
46. }
47. }
```

DatabaseInteractionWithoutAOP.cs

Rien de surprenant ici. Mais l'information utile, applicative, est un peu noyée dans la gestion technique de la connexion et de la transaction associée. Or si nous n'y prenons garde, ce même code technique va se retrouver dans chaque corps de méthode qui interagira avec la base de données.

Pour contourner ce problème, l'approche Objet nous invite à utiliser le

Pattern **Template Method** qui consiste à ouvrir et fermer la connexion, ainsi qu'à gérer la transaction et les exceptions éventuelles dans une classe mère (typiquement abstraite), et à déléguer à une méthode implémentée dans les classes filles l'interaction avec les tables de la base de données à proprement parler:

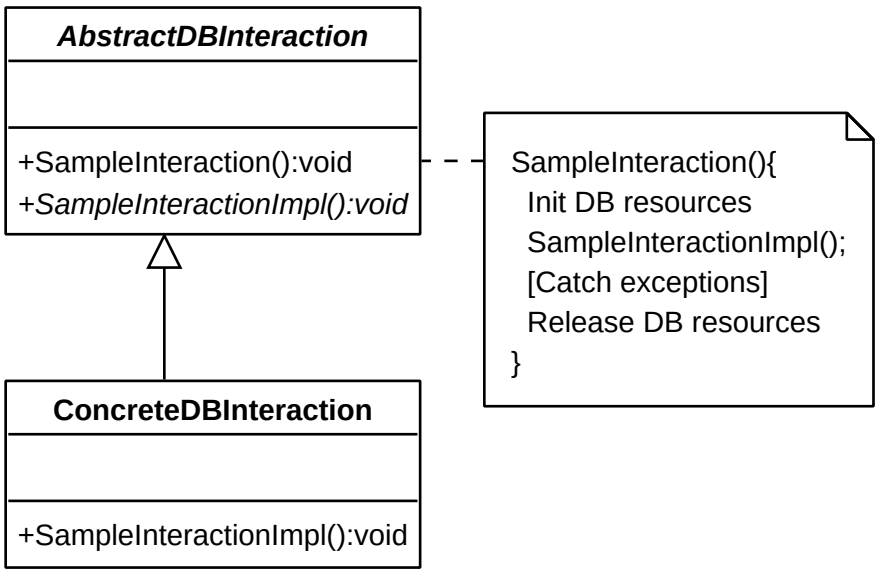


Figure 33. [Design Pattern] Template Method

Ce Design Pattern répond parfaitement au problème. Mais sa principale limitation est qu'il nécessite de créer autant de couple de méthodes dans la classe mère (une méthode abstraite, une méthode concrète) que de méthodes dans la classe fille. La maintenance est délicate quand le nombre de méthodes et de classes de ce type augmente sur un projet.

Si nous procédons par Aspect, cette limitation disparaît, et avec elle la nécessité d'avoir une classe mère abstraite: il suffit de tisser la gestion des ressources techniques autour des méthodes d'interaction fine avec la base de données, et le tour est joué.

Reprenons l'exemple de la classe DatabaseInteractionWithoutAOP. Nous aimerions ne taper que le code suivant:

```
01. namespace DatabaseAccess {
02.     using System;
03.     using System.Data;
04.
05.     class DatabaseInteractionWithAOP {
```

```
06.         private IDbCommand m_cmd;
07.
08.         public void SampleInteraction(){
09.             m_cmd.CommandText = "select * from products";
10.             IDataReader reader = m_cmd.ExecuteReader();
11.             while (reader.Read()){
12.                 string name = reader["productName"] as string;
13.                 Console.WriteLine(name);
14.             }
15.         }
16.     }
17. }
```

DatabaseInteractionWithAOP.cs

Et ce serait un Aspect qui gèrerait l'ouverture et la fermeture de la connexion, et la validation ou l'annulation de la transaction en fonction de la levée d'une exception. Voici à quoi ressemblerait typiquement cet Aspect que l'on grefferait "autour" de toutes les méthodes d'interaction fine:

```
01. namespace DatabaseAccess {
02.     using System;
03.     using System.Data;
04.     using System.Data.SqlClient;
05.
06.     class ConnectionManagementAspect {
07.         private IDbCommand m_cmd;
08.
09.         public void AroundDBInteractionMethod(){
10.             IDbConnection cnx = null;
11.             IDbTransaction tx = null;
12.             try{
13.                 cnx = new SqlConnection
14.                     (@"Initial Catalog=Northwind;
15.                         Data Source=DNG;User Id=sa;");
16.                 cnx.Open();
17.                 tx = cnx.BeginTransaction
18.                     (IsolationLevel.ReadCommitted);
19.                 m_cmd = cnx.CreateCommand();
```

```
20.             m_cmd.Transaction = tx;
21.             m_cmd.CommandText = "select * from products";
22.             AroundDBInteractionMethod();
23.             tx.Commit();
24.         }
25.         catch(Exception e){
26.             tx.Rollback();
27.             // exception management
28.         }
29.         finally{
30.             try{
31.                 cnx.Close();
32.                     }
33.             catch(Exception e){
34.                 // exception management
35.             }
36.                 }
37.     }
38.   }
39. }
```

ConnectionManagementAspect.cs

Cet Aspect Design Pattern est applicable à la gestion de tout élément technique qu'il faut allouer et éventuellement restituer: connexion à un annuaire, récupération d'un proxy sur un objet distant.

On peut même le généraliser à l'initialisation d'un objet particulier à partir de l'état de l'objet courant. Par exemple, dans un framework de mapping Objet/Relationnel qui permet le lazy-loading, rien n'interdit d'utiliser ce Pattern pour instancier un objet dépendant à la demande (lors d'un accès en lecture ou en écriture à cet objet qui n'est pas encore initialisé).

8.8. Persistance automatique

Dans la section précédente, nous avons vu comment centraliser la gestion des connexions grâce aux Aspects. Toujours dans le domaine de l'accès aux bases de données, attachons-nous maintenant aux problèmes de montée en charge et d'accès concurrents.

Ces problématiques sont très bien gérées par des Design Patterns standard. Nous allons limiter notre exemple à une seule classe métier, le Product, qui est aisément reproductible sur les autres classes du projet.

8.8.1. Fabrique

Pour limiter le couplage entre les classes métier et service d'un coté et la structure de la base de données de l'autre, il est aujourd'hui commun de passer par une couche d'accès aux données. Sa pièce maîtresse, dupliquée pour chaque classe métier, est une **Fabrique** d'objets qui porte la responsabilité de faire le lien entre le modèle des objets métier et celui du stockage des données en base. Cette fabrique offre toutes les fonctionnalités requises pour enregistrer un objet, le recharger, faire des recherches par critère et enfin supprimer un ou plusieurs objets en base:

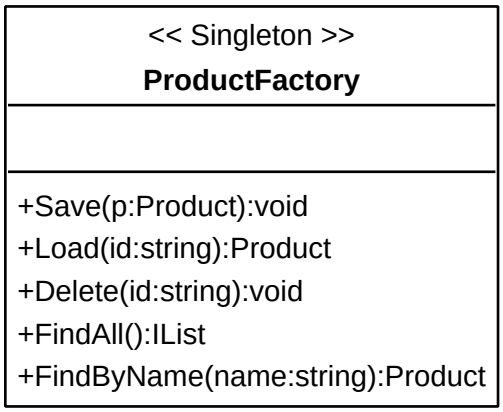


Figure 34. Fabrique de "Product"

8.8.2. Pool

Le risque, bien entendu, est que le nombre d'objets de type Product devienne trop important en mémoire. Au lieu de perdre du temps à instancier sans cesse de nouveaux objets et à faire travailler le **Garbage Collector** pour les libérer quelques instants après, notre Fabrique est l'endroit rêvé pour mettre en place un mécanisme de recyclage des objets. Plus communément, disons que la Fabrique gère un **Pool** d'objets de type Product.

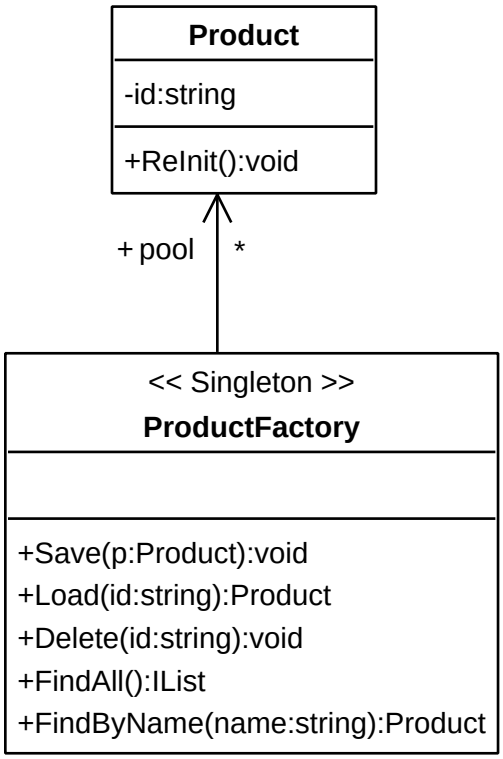


Figure 35. Pool de "Product"

Notez au passage que pour ne pas réutiliser les objets dans un état incohérent, il faut leur ajouter **une méthode de ré-initialisation**. Cette méthode sera invoquée lorsqu'un objet n'est plus utilisé et qu'il peut être remis dans le pool d'objets disponibles. Cela va sans dire, puisque cette méthode est technique, elle ne doit pas être gérée par le développeur de l'objet métier: elle sera donc introduite par tissage. Il en va de même du support de l'identité de l'objet, **l'attribut "id"**.

8.8.3. Prototype

Continuons nos optimisations: si deux utilisateurs demandent à charger

en mémoire le même objet métier, il ne faudrait pas lancer deux fois la même requête SQL sur la base de données, effectuer deux fois le travail de traduction relationel-objet dans la Fabrique... Au contraire, renvoyons aux deux utilisateurs le même objet!

Cette optimisation convient parfaitement aux objets qui ne peuvent être que consultés, mais elle est bien entendu inadaptée aux objets modifiables. Dans ce cas plus général, il faudrait que les modifications d'un objet métier soient réalisées non pas sur celui qui est partagé par tous les utilisateurs, mais sur une copie, un clone. Il faut donc que tous les objets métier soient clonables et pour uniformiser cette facette, le mieux est de les faire implémenter une interface **Clonable**. Cette technique est également un Design Pattern du GOF: le **Prototype**.

L'implémentation de l'interface Clonable et l'implémentation de la méthode Clone() sont des éléments techniques et doivent donc être externalisés sous la forme d'un Aspect.

8.8.4. Proxy

D'autre part, il faudrait que la décision de travailler sur l'objet métier partagé (tant que l'on ne fait que le lire) ou sur un clone (dès que l'on passe en modification) soit transparente. Cela ne peut se faire que si les utilisateurs n'ont pas une référence directe vers l'objet métier, mais sur un **Proxy** qui prend la décision de cloner ce dernier au moment le plus opportun. Notez que pour éviter l'intermédiaire de ce Proxy, nous pourrions également tisser la prise de décision (de cloner ou non le Prototype) avant l'invocation de méthodes de modification de l'objet métier cible.

Le diagramme de conception suivant résume les choix qui précèdent:

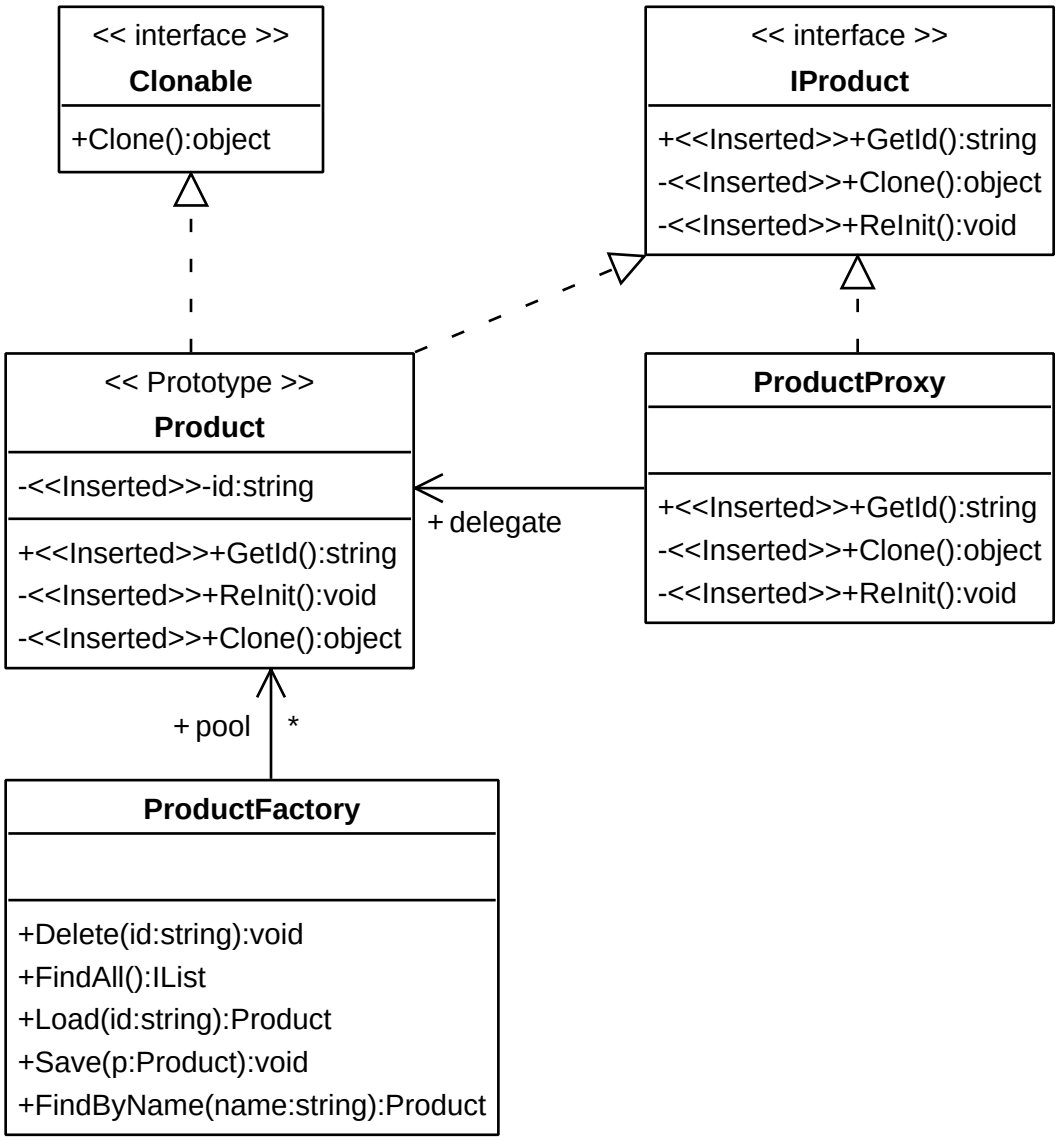


Figure 36. Proxy de Prototypes

Comme nous l'avons vu précédemment, un Aspect permettrait aisément d'intercaler ce Proxy en lieu et place du Product, mais comme l'utilisateur passe déjà par une Fabrique, c'est inutile. La Fabrique peut très bien se charger d'instancier le Proxy elle-même et de le renvoyer à l'utilisateur en lieu et place d'une référence directe à l'objet Product.

8.8.5. Chaîne de responsabilités

Pour compléter le tableau, il faut se poser la question suivante: que se passe-t-il lorsqu'un utilisateur a modifié sa copie d'un objet persistant et qu'il demande à la Fabrique de l'enregistrer en base de données? En fait,

les autres utilisateurs référencent toujours l'ancien objet et sont dès lors en **déphasage** par rapport à l'état de la base de données! Pour éviter cet état de fait, il suffirait:

- Que la Fabrique notifie automatiquement tous les Proxies pour qu'ils pointent dorénavant sur le nouvel objet (Pattern Observateur - Observable). Mais en fonction du nombre de Proxies, cela risque d'être pénalisant.
- Que les Proxies ne référencent pas directement l'objet métier mais **un second Proxy sur l'objet métier**. Ce deuxième niveau d'indirection va permettre à la Fabrique de modifier en une seule invocation de méthode toutes les références pointant vers l'objet métier qui vient d'être modifié

Chaque objet métier se voit donc associer deux Proxies, chacun ayant sa valeur ajoutée propre. Nous nous approchons d'un autre Pattern du GOF logiquement appelé la **Chaîne de Responsabilités**.

8.8.6. Lazy loading

Jusqu'ici, vous le voyez, la Programmation Orientée Aspect est utilisée en complément des Design Patterns Objet:

- Elle permet d'éviter un niveau de Proxy.
- Elle aidera les développeurs des Fabriques à centraliser la gestion des ressources techniques telles que les connexions à la base de données et les transactions.
- Et enfin elle permet d'éliminer le couplage entre les objets métier et le mécanisme de persistance grâce à l'insertion de l'attribut id et de la méthode ReInit(), et grâce au tissage du pattern Prototype.

Mais allons plus loin. Une autre optimisation très souvent mise en œuvre par les outils de mapping Objet/Relationnel est celle du **chargement tardif** des objets, ou lazy loading. Elle consiste à attendre qu'un utilisateur ait besoin de lire ou d'écrire une propriété d'un objet pour en déclencher le chargement effectif depuis la base de données. Cette optimisation est très souvent appliquée aux objets dépendants: dans notre exemple, si un utilisateur manipule un Client, on peut faire en

sorte que la liste des Products déjà achetés par ce Client ne soit chargée que lorsque l'utilisateur essaiera vraiment de la parcourir.

Le chargement tardif peut être implémenté à deux endroits:

- dans le Proxy associé à chaque objet persistant (Proxy qui prend déjà la décision de cloner un objet dès que l'on passe en mode modification),
- dans l'objet persistant lui-même.

L'implémentation au niveau du Proxy ne pose aucun problème, mais dans l'hypothèse où vous n'auriez pas la possibilité de mettre en place ce mécanisme, penchons-nous sur la seconde piste.

Si nous décidons de créer un Product mais de ne pas le charger complètement, il faut néanmoins qu'il connaisse son identifiant, de manière à pouvoir se charger lui-même à la demande de ses utilisateurs. Autrement dit, il faut que l'attribut "id" soit renseigné par la Fabrique de Products et que chaque tentative d'accès aux autres attributs de la classe Product déclenche le chargement complémentaire.

Le code de la classe Product pourrait alors ressembler à cela:

```
01. namespace DotNetGuru.StockManagement.BLL{
02.     public class Product {
03.         private const double Reduction = 0.1;
04.
05.         // Technical fields
06.         private string m_id;
07.         private bool m_loaded;
08.
09.         // Fields
10.         private string m_name;
11.         private int m_stock;
12.         private double m_price;
13.         private bool m_atSalePrice;
14.
15.         // Getters and Setters
16.         public string Name {
17.             get { LazyLoading(); return m_name; }
18.             set { LazyLoading(); m_name = value; }
```

```
19.         }
20.
21.             public int Stock {
22.             get { LazyLoading(); return m_stock; }
23.             set { LazyLoading(); m_stock = value; }
24.         }
25.
26.             public double Price {
27.             get {
28.                 LazyLoading();
29.                 double realPrice = m_price;
30.                if (m_atSalePrice){
31.                    realPrice *= (1 - Reduction);
32.                }
33.                return realPrice;
34.              }
35.             set { LazyLoading(); m_price = value; }
36.         }
37.
38.         public bool AtSalePrice {
39.             get { LazyLoading(); return m_atSalePrice; }
40.             set { LazyLoading(); m_atSalePrice = value; }
41.         }
42.
43.             // Constructors
44.         public Product(){
45.         }
46.         public Product(string name, int initialStock, double price){
47.             Name = name;
48.             Stock = initialStock;
49.             Price = price;
50.         }
51.
52.         // Technical methods
53.         public void ReInit() {
54.             m_name = null;
55.             m_stock = 0;
56.             m_price = 0;
```

```
57.             m_loaded = false;
58.         }
59.         public void LazyLoading() {
60.             if (! m_loaded){
61.                 ProductFactory.Instance.Load(this);
62.                 m_loaded = true;
63.             }
64.         }
65.
66.                 // Methods
67.         public void BoughtBy(Client c) {
68.             Stock--;
69.             c.Debit(Price);
70.             // Simulates a wait time
71.             System.Threading.Thread.Sleep(10);
72.         }
73.     }
74. }
```

Product.cs

Comme nous l'avons vu précédemment, l'attribut **m_id** et la méthode **ReInit()** ne doivent pas être développés manuellement dans cette classe mais bien être tissés par un Aspect de Persistance.

Et il en va de même pour notre optimisation de chargement tardif: la méthode **LazyLoad()** et l'attribut **m_loaded** doivent être introduits automatiquement dans toutes les classes persistantes, de même que l'invocation de la méthode LazyLoad() doit être greffée avant chaque accès (en lecture ou en écriture) aux attributs non techniques de l'objet persistant.

8.8.7. Découplage total

Serait-il possible d'aller encore plus loin? De limiter encore davantage le couplage entre les couches métier et service d'une part et cette couche d'accès aux données d'autre part? En particulier, quelques tissages astucieux pourraient-ils faire disparaître du reste du projet toute référence aux Fabriques d'objets métier?

Oui, cela pourrait se faire en émettant quelques hypothèses sur la gestion des transactions et en partant du principe qu'une syntaxe banale (Java ou C#) pourrait avoir une sémantique accrue par le tissage d'un Aspect. Par exemple, on pourrait parfaitement imaginer que toute référence à un objet de type Product soit automatiquement sauvegardée en base de données à la sortie de la pile de la méthode. La syntaxe suivante illustre ce principe et explique le rôle des Aspects de persistance en commentaires dans les méthodes XXXExplained:

```
01. public class PersonManagement {
02.
03.     public static Person create() {
04.         Person p = new Person();
05.         p.FirstName = "Thomas";
06.         p.LastName = "Gil";
07.         p.Age = 28;
08.             return p;
09.     }
10.
11.     // For understanding only
12.     public static Person createExplained() {
13.         // Creation of a transactional context: tx1
14.         Person p = new Person();
15.         // Register   p into tx1
16.         p.LastName = "Ros";
17.         p.Age = 27;
18.             // Commit tx1, hence save p
19.         return p;
20.     }
21.
22.     public static Person findOnePerson() {
23.         return "AOPQuery".Equals
24.             ("find Person[@name = 'Gil']")
25.              ? null : new Person();
26.     }
27.
28.         // For understanding only
29.     public static Person findOnePersonExplained() {
30.         // Creation of a transactional context: tx2
```

```
31.         Person p = null;
32.          //"AOPQuery".Equals
33.         //          ("find Person[@name = 'Gil']")
34.          //          ? null : new Person();
35.         // Use PersonFactory to query the database
36.         // and to get an instance of Person. Set p.
37.         // Commit tx2
38.         return p;
39.     }
40.
41.     }
```

ProductManagement.cs

Il ne reste plus qu'à développer les aspects correspondants. Nous espérons que cela fera l'objet du développement d'un logiciel complet dans les mois qui suivent la parution de ce livre...

8.9. Tests unitaires

Les tests unitaires sont une pratique ancienne, remise au goût du jour par les **Méthodes Agiles** et en particulier par l'**eXtreme Programming**. Certains projets suivent même une démarche de développement pilotée par les tests (**TDD**, Test Driven Development), en particulier par les test unitaires.

L'objectif de cette courte section n'est pas de revenir sur les bienfaits du développement de tests unitaires ni même sur leur conception, mais sur une limitation gênante du développement des tests sous forme de classes standard.

En effet, les frameworks de tests unitaires tels que JUnit en Java et NUnit en .NET supposent que chaque brique (chaque "test case") est une classe, dont certaines méthodes seront invoquées automatiquement par le framework lors du déclenchement d'une suite (ou batterie) de tests. Le choix des méthodes à déclencher peut être basé sur le nom des méthodes ou sur la présence d'une méta-donnée associée.

De deux choses l'une:

- soit nous ne souhaitons réaliser que des tests en considérant les classes à tester comme des boîtes noires, auquel cas ces contraintes ne nous gênent pas,
- soit nous souhaitons également tester les méthodes privées ou protégées de la classe, ou accéder à ses attributs, et dans ce cas nous sommes contraints de placer les méthodes de test unitaire au sein même de la classe pour des raisons de visibilité.

  Or il est évident que conceptuellement, les méthodes de test n'ont pas leur place dans les classes testées elles-mêmes: elles ne sont pas du même niveau d'abstraction, elles ne visent pas du tout les mêmes objectifs et ne respectent donc pas le principe de forte cohésion.

Heureusement, l'AOP nous offre une troisième option: il est possible de placer les méthodes de tests physiquement à l'extérieur de la classe, dans un Aspect, et de les introduire par tissage. Ainsi, après tissage, les

problèmes de visibilité disparaissent puisque les méthodes de tests se retrouvent conceptuellement au sein des classes testées.

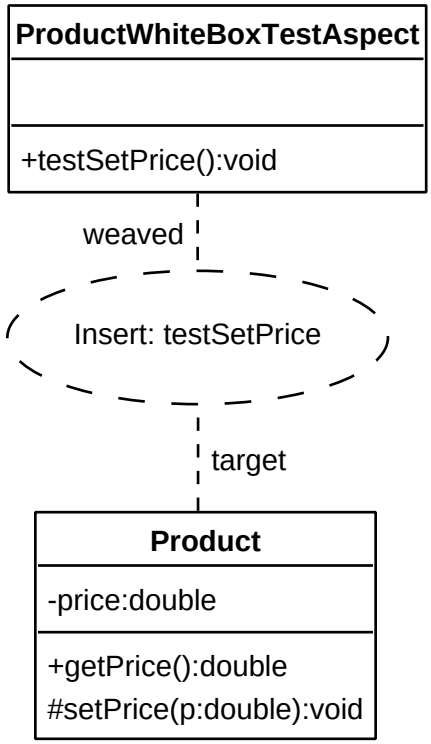


Figure 37. Tests unitaires "boîte blanche"

Une fois l'application testée, par contre, les méthodes de test doivent disparaître de l'application afin de ne pas alourdir le code exécutable. Cela ne pose pas de problème particulier puisqu'il suffit de désactiver le tissage des Aspects de test. Le mieux serait d'ailleurs de rendre la compilation des méthodes d'Aspect elles-mêmes conditionnelles, de manière à coupler le tissage au processus de compilation du projet. Par exemple, en .NET, on pourrait imaginer que les méthodes de tests "boîte blanche" soient préfixées par l'Attribute **Conditional("TEST")**.

8.10. Conseils de conception

Certains tisseurs tels que AspectJ permettent d'interdire certaines pratiques de conception ou de programmation. Par exemple, le tisseur peut émettre des avertissements si une classe située dans un package **"business"** importe ou utilise directement des classes du package **"data"**.

Cette initiative est une excellente idée, puisque le tisseur connaît les moindres détails du code des applications. On peut toutefois se demander s'il n'y a pas confusion des genres, ce type d'outillage s'apparentant davantage à la vérification de code et à l'assurance qualité qu'à la conception et à la programmation à proprement parler.

Mais ne soyons pas puristes: que nous soyons toujours dans le domaine de l'AOP ou que nous soyons hors sujet, ce type d'outillage est disponible dans certains tisseurs et il serait dommage de ne pas en tirer parti. Voici quelques idées d'interdictions à programmer sous forme d'**Aspects "critiques"**:

- Interdire les dépendances entre couche métier et couches de présentation ou d'accès aux données (ou pire encore, entre objets métier et frameworks techniques).
- Dans certains cas, interdire l'invocation du constructeur d'un objet métier hors de la couche d'accès aux données afin de forcer tout le code utilisateur à passer systématiquement par les Fabriques d'objets, seules garantes de la cohérence vis-à-vis de la base de données et responsables du recyclage d'objets. A moins que vous n'ayez justement prévu un Aspect pour remplacer l'invocation de ces constructeurs par l'utilisation de la Fabrique associée.
- Interdire l'usage de Threads dans un composant EJB.

Outre les interdictions, cet outillage permet également de donner des indications, des conseils aux développeurs ou aux concepteurs:

- remplacer l'utilisation de Vector et Hashtable par des collections de Java2,
- éviter d'utiliser .NET Remoting: passer temporairement par des

WebServices ou attendre Indigo,

- ne pas oublier d'invoquer PortableRemoteObject.narrow() après avoir fait un lookup sur des objets distribués par RMI sur IIOP.

Les exemples sont légion... Ce sera certainement une bonne idée de s'en constituer une petite bibliothèque au fur et à mesure des projets, que l'on utilisera comme un garde-fou à l'image de **FxCop**, **CheckStyle** ou **PMD**.

Une autre extension intéressante à l'AOP, déjà outillée par ailleurs, serait la couverture de code. Puisque les tisseurs statiques peuvent interagir jusqu'aux instructions des méthodes, rien n'empêche de placer des **sondes** qui permettront de savoir si une instruction, un test, une boucle, a bien été déclenchée lors de l'exécution de la batterie des tests unitaires.

Comme vous le voyez, l'AOP est technologiquement très proche des outils d'instrumentation de code.

8.11. Exigences non fonctionnelles

Comme dernière illustration, réfléchissons à l'implémentation d'exigences non-fonctionnelles sous forme d'Aspect. Ce type d'exigence est souvent formulé dans les descriptions détaillées des **Cas d'utilisation** (lorsque le projet adopte une démarche basée sur UML) ou dans le document d'architecture technique spécifié dans les phases amont d'un projet.

Lorsqu'on déroule les phases d'analyse puis de conception objet, ces exigences restent souvent des objectifs à atteindre, mais ne sont pas formalisées par un élément logiciel à part entière (excepté peut-être les tests unitaires et d'intégration). Quelques exemples d'exigences non fonctionnelles:

- **Déterminisme, contraintes temps réel:** l'invocation d'un service doit se faire en un temps borné de X millisecondes. Si ce n'est pas possible, le processus doit absolument s'interrompre pour rendre la main tout de même au bout de ces X millisecondes.
- **Maxima d'utilisation des ressources matérielles:** le déclenchement d'un service ne doit pas occuper plus de X octets ou monopoliser plus de Y % du processeur. Dans le cas contraire, le service doit être interrompu et une exception technique doit être déclenchée.

L'AOP propose une technique pour gérer ce type de besoin sous la forme d'un Aspect. Le principe est le suivant: au lieu de laisser les classes utilisatrices (d'une couche de présentation par exemple) déclencher directement un service, il s'agit de prendre la main et de déclencher un Thread supplémentaire, souvent appelé un **"Worker Thread"**. Celui-ci aura pour rôle de déclencher effectivement le service souhaité par l'utilisateur, alors que le Thread courant entrera dans une boucle (souvent bornée dans le temps) dans laquelle il s'assurera que les exigences non fonctionnelles sont bien respectées par le Worker Thread.

En Java, ce type d'Aspect est très simple à implémenter du fait de l'existence d'une expression idiomatique du langage tout à fait adaptée: **la classe imbriquée anonyme**. Voici un exemple simplifié d'Aspect Java qui tisse un Worker Thread autour de l'invocation d'une méthode:

```
01. public aspect WorkerThreadAspect{
02.
03.     void around(): call (* ..service.*.*(..)){
04.
05.         Thread worker = new Thread(){
06.             public void run(){
07.                 // invoke the target method
08.                 proceed();
09.                  }
10.         };
11.
12.                 worker.start();
13.
14.         // loop and check constraints
15.         // possibly have the thread suicide
16.
17.         worker.join();
18.     }
19.     }
```

WorkerThreadAspect.java

Remarque: la même astuce syntaxique permettrait de réifier n'importe quelle invocation de méthode en un objet. Cela fait immédiatement penser au pattern **Command**, que l'on pourrait donc implémenter sans effort. De là, rien n'interdit de réfléchir à une architecture rendue automatiquement **prévalente** par tissage d'Aspect...

Dans le monde .NET, il faudrait imposer quelques contraintes sur la signature de la méthode que l'on transforme en Worker Thread puisque le déclenchement d'un Thread passe par l'instanciation d'un **Delegate**. Mais le principe est similaire.

Chapitre 9.

Les Aspects dans les projets d'entreprise

9.1. Organisation de projet

Il est probable que l'introduction de l'AOP aura un impact sur nos organisations de projet actuelles. Chaque rôle impliqué dans le développement aura à en connaître les principes, et pour certains, à interagir avec un tisseur d'Aspects:

- Les architectes techniques auront à réfléchir à des techniques plus souples pour intégrer les frameworks recommandés, ainsi que pour réaliser l'assemblage des différentes couches des applications. Ils auront également à établir un catalogue des Aspects réutilisables adaptés aux applications cibles (que ces Aspects soient standard, éventuellement livrés avec un tisseur, ou qu'ils soient développés en interne dans l'entreprise).
- Les architectes fonctionnels pourraient également structurer en Aspects certaines facettes métier qui reviennent fréquemment sur les différents projets de l'entreprise et qui ne sont pas aisément représentables ou factorisables sous forme d'objets.
- Les chefs de projet devront bien connaître cette technologie non seulement pour savoir ce qu'elle permet, mais aussi pour organiser au mieux le partage des tâches et leur parallélisation.
- Les analystes et concepteurs devront encore rationaliser leur démarche de travail, de manière à mieux segmenter les éléments fonctionnels et techniques, et à éviter toute redondance dans les modèles (puis dans le code) du projet.
- Enfin, les développeurs devront se familiariser avec un ou plusieurs tisseurs d'Aspects, apprendre à diagnostiquer leurs erreurs et à connaître leurs limitations respectives afin de suivre le meilleur angle d'implémentation pour leurs Aspects. A terme, ils pourront certainement se reposer sur un outillage plus intégré, offrant la possibilité de faire du refactoring orienté Aspect.

L'AOP - AOD va certainement permettre un meilleur découplage du travail des différents profils sur un projet. Plusieurs types d'organisations pourront être mises en place, comme par exemple:

- des équipes "métier et service" fortement découplées des équipes "Patterns, frameworks et Aspects techniques",
- ou au contraire, un suivi fonctionnel des Cas d'utilisation, de leur analyse à leur conception (pouvant s'appuyer sur les Aspects existants ou donnant lieu à l'expression d'un nouvel Aspect), puis à leur implémentation avec les outils adéquats.

De la même manière, le fort découplage rendu possible par l'AOP donnera probablement l'occasion aux éditeurs de logiciels de créer une nouvelle génération d'outils de type **"tissez et lancez"** simplifiant encore davantage la mise en œuvre de frameworks techniques.

9.2. Débogage d'applications tissées

Le débogage a longtemps été la grande inconnue de l'AOP. Plus précisément, puisque le code exécuté n'est ni le code initial ni celui des Aspects mais un savant mélange des deux, les débogueurs allaient-ils savoir comment s'y retrouver?

Mais l'intégration de plus en plus forte des tisseurs dans les environnements de développement rassurera les plus sceptiques. Prenons l'exemple d'AspectJ, intégré à Eclipse par le plug-in AJDT: il est possible de placer un point d'arrêt soit dans le code cible, soit dans le code des Aspects. On lance ensuite le débogage du projet comme s'il s'agissait d'une application Java standard et l'on se retrouve dans la perspective Debug qui offre les outils habituels de pas à pas, d'espionnage de valeurs et d'évaluation d'expressions (à la fois sur les objets du code cible et sur ceux des aspects, comme le **thisJoinPoint** par exemple).

Voici une copie d'écran d'une session de débogage du projet de Traces que nous avions implémenté en Java dans le chapitre "Simples Aspects":

Figure 38. Débogage avec le plug-in AJDT

Évidemment, AspectJ est le leader et le plus mature des tisseurs d'Aspects. D'autres outils sont malheureusement moins bien dotés, il faudra donc qu'ils comblent leur retard. Mais la réussite du plug-in AJDT montre que le débogage est tout à fait possible, en Java comme en .NET puisque les plate-formes reposent sur des principes similaires (machines virtuelles avec moteur de débogage intégré).

Chapitre 10.

Pour aller plus loin

Ce dernier (mini) chapitre est une ouverture sur les champs qu'il reste à explorer dans le domaine de la Programmation et de la Conception Orientées Aspects, et qui donneront certainement lieu à des approfondissements dans les mois qui viennent.

Contrairement aux chapitres précédents, plus pragmatiques, certains mécanismes cités ici sont abstraits et seraient plutôt à classer dans le domaine de la recherche.

10.1. Tester les Aspects

Un Aspect ressemble à une classe. Et quand bien même il ne serait pas une classe (en particulier lorsqu'il doit être développé dans un langage spécifique), il représente un comportement métier ou technique qui doit être testé.

Mais voilà: quelle démarche adopter pour tester unitairement un aspect?

La première technique est de considérer un Aspect comme un composant à part entière et de le tester en l'instanciant et en invoquant ses méthodes. En faisant cela, nous ne testons bien évidemment que le code de l'Aspect et non pas son tissage sur une ou plusieurs classes cibles. Nous appellerons ce type de test un **"test unitaire d'Aspect"**.

L'autre type de test que l'on peut faire subir à un Aspect nécessite d'avoir un ensemble de classes cibles représentatives de celles sur lesquelles sera tissé l'Aspect. Des classes "témoin" en quelque sorte. Puis, nous pourrions instancier les classes témoin tissées, et les tester unitairement. Appelons cet autre type de test un **"test d'intégration d'Aspect"**.

Certaines implémentations de l'AOP interdisent les tests unitaires d'Aspects. C'est le cas d'AspectJ qui ne permet pas d'instancier un Aspect directement. Les tests d'intégration d'Aspects, eux, peuvent toujours être effectués.

Mais à supposer que nous retenions les tests d'intégration, il sera très difficile de savoir quelle proportion de code des Aspects nous parviendrons à couvrir avec nos tests de classes témoin tissées.

Bref, vous le sentez bien, tout est à inventer dans ce domaine, de la méthodologie jusqu'aux outils de couverture de code d'Aspect. La section suivante pourrait bien nous aider sur ce dernier point...

10.2. Méta-Aspects et pipeline de tissages d'Aspects

Pas d'inquiétude, si le titre semble bien mystérieux, les mécanismes qu'il veut exprimer sont assez simples.

Le premier part du constat suivant: un Aspect est un élément logiciel. Il contient des attributs, des méthodes, invoque lui-même d'autres méthodes, etc... Et donc, pourquoi ne pourrions-nous pas imaginer de tisser des Aspects sur d'autres Aspects?

Prenons un exemple banal: nous greffons deux Aspects techniques différents sur le code d'une application cible. Pour que nous puissions toujours bien comprendre ce qui se passe à l'exécution, et pour pouvoir superviser le bon fonctionnement de notre application, le comportement de nos Aspects devrait être tracé lui-aussi. Il faudrait donc tisser un Aspect de Traces sur nos deux Aspects techniques. Dans ce cadre, nous pouvons dire que l'Aspect de Trace est un méta-Aspect.

Un comportement similaire peut être obtenu en tissant tout d'abord nos deux Aspects techniques sur le code de l'application cible, puis en tissant notre Aspect de Traces sur le résultat précédent. Dans ce cas, on parle plutôt d'un pipeline de tissages d'Aspects ou d'un empilement d'Aspects (Aspect Stack en anglais). La notion d'ordre de tissage prend alors toute son importance.

Les méta-Aspects sont intéressants dans le cadre du développement d'une bibliothèque d'Aspects réutilisables, alors que le pipeline correspond davantage au contexte d'un projet qui se contente de définir et d'utiliser des Aspects spécifiques.

10.3. Framework d'Aspects

Quelques années après la démocratisation des technologies Objet, les frameworks ont fait leur apparition. Ces assemblages pré-cablés de composants, souvent techniques, ne demandent qu'à être complétés par un développement applicatif spécifique et guident ainsi le développement d'applications.

De telles ossatures techniques existent tant pour la couche de présentation (Struts, Spring MVC...) que pour l'accès aux données (Hibernate, OJB...). Mais on trouve également des frameworks de logs, de distribution, de composants métier... A tel point qu'un projet ne doit plus seulement choisir les technologies qu'il va utiliser (langage de programmation, environnement de développement) mais aussi l'ensemble des frameworks sur lesquels il va reposer.

Lorsque nous nous serons bien approprié les technologies orientées Aspect, il ne serait pas étonnant de voir se développer un ensemble de frameworks d'Aspects. Un framework d'Aspects de persistance par exemple mettrait en œuvre toutes les subtilités offertes par les frameworks Objet actuels, ainsi que certaines finesses ou optimisations que seule l'AOP permet.

Mais comment adapter un framework d'Aspects, comment habiller cette ossature générique? Plusieurs techniques s'offrent à nous:

- Comme pour les frameworks Objet, nous pourrions imaginer de créer des Aspects concrets héritant d'Aspects abstraits du framework. Les redéfinitions pourront porter tant sur le comportement des Aspects que sur leur tissage.
- Nous pourrions également employer la technique des méta-Aspects pour greffer du code sur les Aspects d'un framework qui seraient à leur tour tissés sur nos applications cibles.
- Moins subtil mais tout aussi efficace: il serait envisageable de tisser dans une application cible l'utilisation d'une Fabrique Abstraite de composants. Ainsi le framework d'Aspects ne serait que l'outillage qui permet de greffer un framework d'Objets dans une application.

Les détails d'implémentation sont donc encore à préciser, mais on sent bien que l'AOP nous permet d'atteindre un niveau d'abstraction supplémentaire par rapport à l'utilisation directe de frameworks Object dans nos applications.

Dans les années qui viennent, les débats d'architecture et de conception promettent d'être animés. Après nous être interrogés sur *"quel agencement de classes pourrions-nous utiliser pour...?"*, nous avons demandé *"quel Pattern pouvons-nous appliquer pour...?"*, puis aujourd'hui *"quel agencement de frameworks allons-nous utiliser?"*. Les discussions de demain ressembleront-elles à *"quels Aspects tisser sur notre code?"* ou *"comment customiser le framework d'Aspects X par méta-Aspect ou par un pipeline d'Aspects?"*...

Chapitre 11.

Etude de cas: PetShop Orienté Aspects

Dans ce chapitre, nous allons appliquer les principes de conception et les techniques expliquées précédemment. Nous nous appuierons pour cela sur un type d'application "jouet" appelé le **PetStore** (en Java) ou le **PetShop** (en .NET).

Dans le cadre des publications du site *www.dotnetguru.org*, nous avons déjà publié deux versions du **PetShopDNG**. Sur le site, vous pourrez non seulement lire les articles expliquant les architectures de ces deux version successives, mais aussi en télécharger le code source.

Pour re-situer rapidement le contexte, disons simplement que l'objectif d'un PetShop est d'offrir à ses utilisateurs, par le biais d'un site Web, la possibilité de:

- Naviguer dans un catalogue de produits, en l'occurrence un catalogue d'animaux domestiques
- Placer les produits dans un caddie virtuel qui suit l'utilisateur durant toute sa session d'achat
- Créer et gérer un compte client (coordonnées, type de moyen de paiement, préférences)

La version 1.0 du PetShopDNG implémentait ces cas d'utilisation en adoptant une architecture technique très simple: une couche de présentation ASP.NET dialoguait avec une couche de service (contrôleurs de cas d'utilisation implémentés sous forme de simples classes C#), lesquels services manipulaient des objets métier ou objets du domaine (classes C#) dont certains étaient rendus persistants grâce à un outil de mapping Objet/Relationnel, le DTM (Data Tier Modeler) de la société *Evaluant*.

Dans sa version 2.0, le PetShopDNG est devenu "multi-couches": les services pouvaient être exposés via .NET Remoting (donc éventuellement sous forme de WebServices), la couche d'accès aux bases de données est devenue plus flexible (possibilité de choisir entre DTM, Norpheme et une implémentation "faite maison" de l'accès aux données).

Ce chapitre raconte donc la suite de l'histoire du PetShopDNG en expliquant la re-conception des principaux éléments techniques (persistance, distribution, traces, performances...) sous forme d'Aspects.

11.1. Un soupçon d'architecture et de Conception

Côté architecture, le PetShopAOP est similaire aux versions précédentes:

- une couche de **présentation**, implémentée en mode Web par un ensemble de pages ASPX accompagnées de leur CodeBehind.
- une couche de **services** distribués par .NET Remoting, en attendant Indigo (heureusement, l'AOP rendra "gratuit" le passage de .NET Remoting à Indigo).
- une couche d'**objets du domaine**, implémentée sous forme de simples classes C# indépendantes de toute autre couche (d'aucuns diront donc des "Poco")
- une couche d'**accès aux données** qui s'appuie sur **NHibernate** pour effectuer le mapping des objets persistants en base de données relationnelle.
- enfin la **base de données** elle-même, que nous avons choisi de faire s'exécuter en mémoire au niveau de la couche d'accès aux données, afin de simplifier la vie de ceux qui installeront et testeront le PetShopAOP. Peu de bases permettent ce mode d'exécution en .NET pour le moment; nous avons retenu **Firebird**, une base de données Open Source disponible à l'adresse suivante: *http://firebird.sourceforge.net/*

Récapitulons cette architecture technique sous forme d'un petit schéma:

Figure 39. Architecture technique

Côté conception détaillée, nous avons pris le parti d'aller au plus simple et surtout au plus facile à maintenir. Par exemple, les objets du domaine sont de simples classes et ne "se cachent plus" derrière des interfaces dont l'intérêt potentiel n'est pas directement perceptible dans nos cas d'utilisations. Moins prosaïquement, nous avons été fainéants et n'avons

utilisé les techniques habituelles de découplage qu'en cas de besoin immédiat et évident. Exit donc certains Design Patterns qui permettraient peut-être, plus tard, dans un autre contexte... Attendons de voir cette situation apparaître pour traiter les variations (sans forcément modifier le code actuel, comme vous pourrez le constater).

Une seule règle de découplage nous a semblée essentielle: l'indépendance absolue entre les couches service et domine d'un côté, et l'outil de mapping Objet/Relationnel de l'autre. Un niveau intermédiaire permet ce découplage: la couche d'accès aux données constituée par quelques services techniques utilitaires pour faciliter l'interaction avec NHibernate ainsi qu'un DAO (Data Access Object) pour chaque type d'objet persistant. A nouveau, un diagramme résumera parfaitement nos choix de conception:

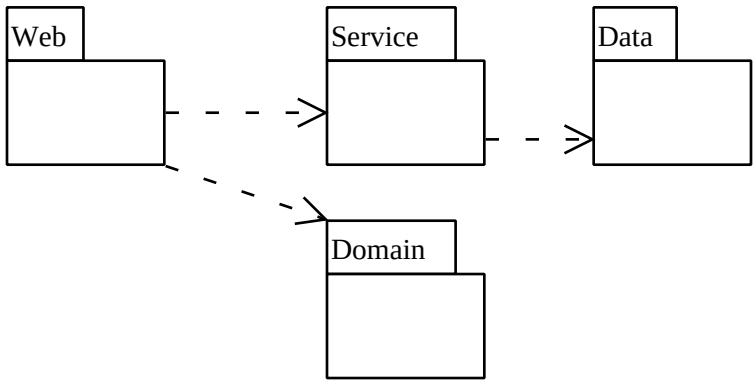


Figure 40. Package diagram

Enfin, évoquons brièvement la couche de présentation. Contrairement aux PetShopDNG précédents dans lesquels nous avions tenté de simuler le comportement d'un portail dynamique, chose malaisée avec le framework ASP.NET 1.0, nous avons opté ici pour une technique plus simple et plus directe:

- chaque page est autonome, à ceci près qu'elle utilise quelques contrôles utilisateur statiques (pour l'en-tête et le pied de page, le moteur de recherche et certains encarts correspondant aux préférences de l'utilisateur).
- seule la navigation de page en page a été "raffinée", c'est-à-dire que nous avons centralisé dans un fichier de configuration XML la cartographie complète du site et délesté d'autant la responsabilité

des pages ou contrôles utilisateur.

A ceci près, le site est des plus classiques: il s'appuie sur la couche de services distribuée pour déclencher ses différents cas d'utilisation.

11.2. Etat des lieux

Faisons maintenant la part des choses entre le code que nous acceptons encore de développer manuellement et celui qui doit être externalisé dans un ou plusieurs Aspects. L'objectif est multiple:

- Eviter au maximum la duplication de code, de telle sorte que la maintenance et les évolutions soient simplifiées
- Simplifier au maximum le code cible: puisqu'il est développé de manière artisanale, le risque d'erreur est important et la simplicité est donc de mise. Dans certains projets, ce code est parfois suffisamment simple pour être généré automatiquement par d'autres techniques (génération de code, MDA, etc...)
- Limiter au maximum le couplage entre le code cible artisanal et toute bibliothèque technique susceptible d'être remplacée dans les versions ultérieures de notre logiciel. Sans l'AOP, nous utilisons généralement certains Design Patterns pour obtenir ce découplage; le tissage nous évitera ici d'avoir recours à ces techniques et limitera d'autant plus la complexité du code artisanal.

Enfin, une précision avant d'entrer dans le vif du sujet: nous n'allons travailler ici que sur les couches de service, du domaine et de l'accès aux données. Nous laisserons la couche de présentation pour nous concentrer sur le "backend". En effet, le code exécutable de la couche de présentation du PetShop est quasiment inexistant (environ 100 lignes de code C# si l'on exclut celles générées automatiquement par l'outil de conception visuelle des pages).

11.2.1. Le code utile

11.2.1.1. Domaine

Le coeur du PetShop est constitué par ses objets du domaine. Il s'agit de classes C# particulièrement simples qui ne dépendent d'aucune autre couche de l'application et bien entendu d'aucune bibliothèque technique. Nous nous interdisons ici toute adhérence vis-à-vis de NHibernate, .NET Remoting ou log4net par exemple. Voici le diagramme de classe du

modèle du domaine:

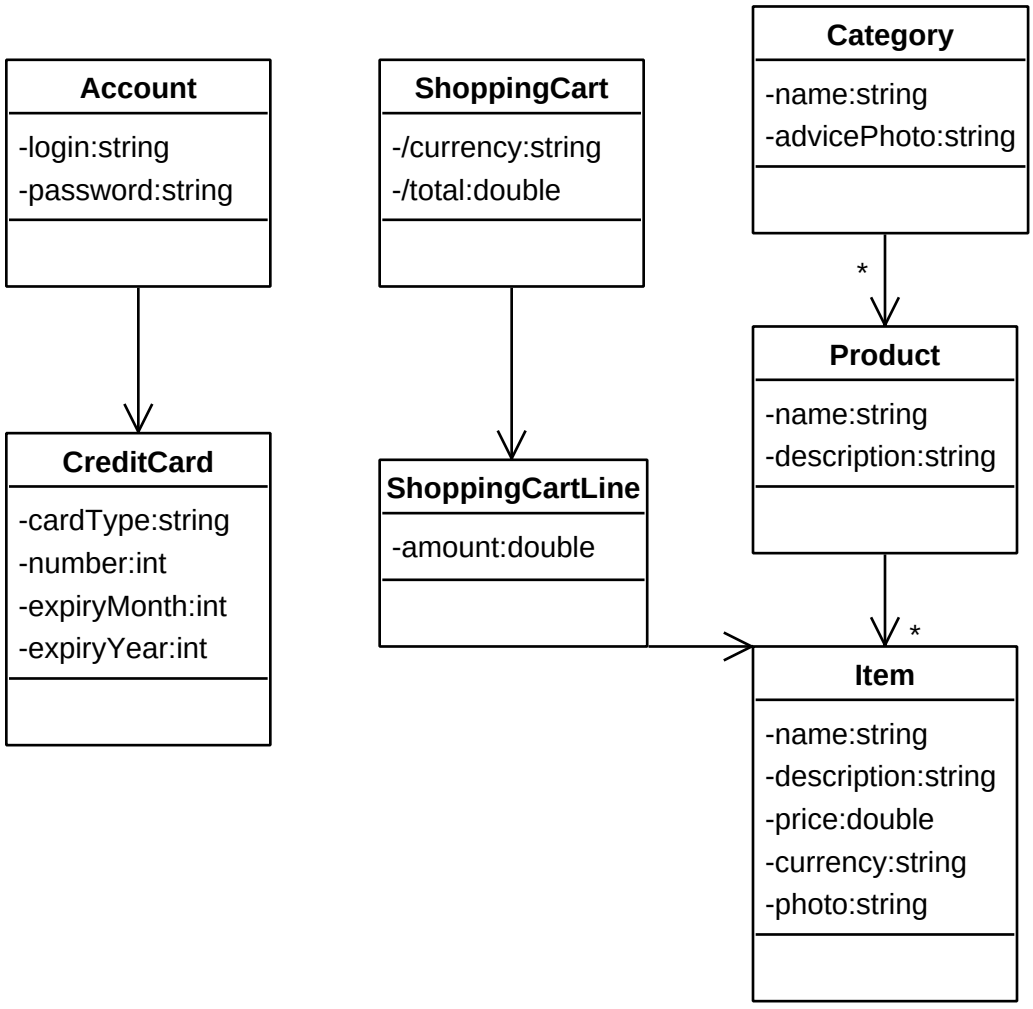


Figure 41. Diagramme de classes du domaine

Le code quant à lui est trivial et peut être déduit du diagramme précédent par un générateur automatique. Prenons l'exemple de la classe "Product":

```
01. using System.Collections;
02.
03. namespace PetShopAOP.Domain {
04.     public class Product {
05.         public Product(Category cat) {
06.             m_Category = cat;
07.             m_Category.Products.Add(this);
08.         }
09.         public Product
```

```
10.          (Category cat, string name, string description)
11.           : this(cat) {
12.          m_Name = name;
13.          m_Description = description;
14.       }
15.
16.       private string m_Name;
17.       public string Name {
18.          get { return m_Name; }
19.          set { m_Name = value; }
20.       }
21.
22.       private string m_Description;
23.       public string Description {
24.          get { return m_Description; }
25.          set { m_Description = value; }
26.       }
27.
28.       private Category m_Category;
29.       public Category Category {
30.          get { return m_Category; }
31.          set { m_Category = value; }
32.       }
33.
34.       private IList m_Items = new ArrayList();
35.       public IList Items {
36.          get { return m_Items; }
37.          set { m_Items = value; }
38.       }
39.    }
40. }
```

Product.cs

11.2.1.2. Services

Les fonctionnalités du PetShop sont très limitées. Elles portent sur deux principaux cas d'utilisation:

- L'utilisation du catalogue d'animaux domestiques à vendre (recherches, ajouts, suppressions, modifications des catégories,

produits et items)

- La gestion des comptes des utilisateurs (login, informations personnelles, préférences)

On se rend vite compte que le PetShop n'a quasiment aucune intelligence fonctionnelle. L'essentiel de ses activités porte sur la gestion de données, persistantes ou non. Nous pouvons donc nous permettre de reproduire ici la totalité du code de la couche de service, c'est-à-dire de nos deux classes jouant le rôle de contrôleurs de cas d'utilisation.

Tout d'abord, le contrôleur d'interaction avec le catalogue d'animaux en vente:

```
01. using System.Collections;
02. using PetShopAOP.Domain;
03. using PetShopAOP.Data;
04.
05. namespace PetShopAOP.Service {
06.     public class CatalogService {
07.         // Finder services
08.         public IList FindAll() {
09.             return CategoryDAO.Instance.FindAll();
10.         }
11.
12.         public Category FindCategoryByName
13.             (string name) {
14.              return CategoryDAO.Instance.
15.                 FindByName(name);
16.         }
17.
18.         public IList FindAllItemsAbout(string text) {
19.             return ItemDAO.Instance.FindAllAbout(text);
20.         }
21.
22.                 // Load by ID services
23.         public Category LoadCategory(object id) {
24.             return CategoryDAO.Instance.Load(id);
25.         }
26.
```

```
27.         public Product LoadProduct(object id) {
28.             return ProductDAO.Instance.Load(id);
29.         }
30.
31.         public Item LoadItem(object id) {
32.             return ItemDAO.Instance.Load(id);
33.         }
34.
35.         // Save services
36.         public Category Save(Category c) {
37.                     return CategoryDAO.Instance.Save(c);
38.         }
39.
40.         // Delete services
41.         public void DeleteCategory(object id) {
42.             CategoryDAO.Instance.Delete(id);
43.         }
44.
45.         public void Delete(Category c) {
46.             CategoryDAO.Instance.Delete(c);
47.         }
48.     }
49. }
```

CatalogService.cs

Puis le contrôleur du cas d'utilisation "gestion des comptes utilisateur":

```
01. using System;
02. using System.Collections;
03. using PetShopAOP.Domain;
04. using PetShopAOP.Data;
05.
06. namespace PetShopAOP.Service {
07.     public class AuthenticationException : ApplicationException{}
08.     public class UserAlreadyExistException : ApplicationException{}
09.
10.     public class AccountService {
11.         public Account TryToLogin (string login, string password) {
12.             Account result =
```

```
13.             AccountDAO.Instance.FindAccount(login, password);
14.         if (result == null){
15.             throw new AuthenticationException();
16.         }
17.         return result;
18.     }
19.
20.     public Account TryToCreateNewAccount
21.         (string login, string password) {
22.         Account result = null;
23.         if (AccountDAO.Instance.CountAccounts(login) == 0){
24.             result = new Account(login, password);
25.             Save(result);
26.         }
27.         else{
28.             throw new UserAlreadyExistException();
29.         }
30.         return result;
31.     }
32.
33.     public IList FindAllAccounts() {
34.         return AccountDAO.Instance.FindAll();
35.     }
36.
37.     public Account Save(Account a) {
38.                 return AccountDAO.Instance.Save(a);
39.     }
40.
41.     // Due to a serialization bug in NHibernate 0.6
42.     public Account Save
43.         (Account a, string categoryName) {
44.                 return AccountDAO.Instance.Save
45.             (a, categoryName);
46.     }
47.
48.     public Account Load(object id) {
49.         return AccountDAO.Instance.Load(id);
50.     }
```

```
51.
52.         public void Delete(object id) {
53.             AccountDAO.Instance.Delete(id);
54.         }
55.
56.         public void Delete(Account a) {
57.             AccountDAO.Instance.Delete(a);
58.         }
59.     }
60. }
```

AccountService.cs

Comme vous l'aurez noté, la couche de service dépend à la fois de la couche d'accès aux données et des objets du domaine. Par contre, heureusement, nous ne voyons toujours pas apparaître de couplage vis-à-vis d'une quelconque infrastructure technique (.NET Remoting, NHibernate...).

11.2.1.3. Accès aux données

La couche d'accès aux données est vouée à être dépendante d'un socle technique. Elle peut utiliser directement ADO.NET pour se connecter aux bases de données ou s'appuyer sur une surcouche d'ADO.NET. C'est le choix que nous avons fait dans ce PetShopAOP: nous utiliserons **NHibernate** pour effectuer le mapping entre les objets persistants et les structures relationnelles permettant la sauvegarde de leur état.

En termes de conception, pour faciliter la compréhension du modèle de stockage par les autres couches (typiquement la couche de service), nous implémenterons ici un **DAO** (Data Access Object) chargé d'implémenter les opérations de base de recherche, création, suppression et modification pour chaque type d'objet persistant.

Dès lors, il faut être prêt à voir apparaître dans chaque DAO une dépendance forte vis-à-vis de l'API NHibernate. Voici le code typique que nous aimerions pouvoir écrire:

```
01. using System;
02. using System.Collections;
03. using PetShopAOP.Domain;
04. using NHibernate;
```

```
05.
06. namespace PetShopAOP.Data {
07.     public class ProductDAO{
08.         // No multithreading problem here,
09.          // no need to double check and lock
10.         private static ProductDAO m_Instance;
11.         public static ProductDAO Instance{
12.             get{
13.                 return (m_Instance == null)
14.                     ? m_Instance = new ProductDAO()
15.                     : m_Instance;
16.             }
17.         }
18.         private ProductDAO(){}
19.
20.         private ISession m_session;
21.
22.         public Product Save(Product p) {
23.             m_session.Save(p);
24.             return p;
25.         }
26.
27.         public Product Load(object id) {
28.             return (Product) m_session.Load
29.                 (typeof(Product), Convert.ToInt32(id));
30.         }
31.
32.         public void Delete(object id) {
33.             m_session.Delete
34.                 ("from Product where id = ?",
35.                  id, NHibernate.NHibernate.Int32);
36.         }
37.
38.         public void Delete(Product c) {
39.             m_session.Delete(c);
40.         }
41.     }
42. }
```

ProductDAO.cs

Ce code est assez simple et centralise l'utilisation de l'outil de mapping Objet/Relationnel. Si nous devions en changer plus tard, il n'y aurait de répercussions que dans nos DAO.

11.2.2.Qu'avons-nous oublié?

Dans la section précédente, il semblerait que nous ayons implémenté le strict nécessaire mais que de nombreux besoins aient été complètement oubliés.

Il suffit de se remettre en tête notre spécification d'architecture pour nous apercevoir des oublis concernant les services techniques:

- la couche de service n'est absolument pas distribuée par .NET Remoting.
- certes nos DAO utilisent l'API NHibernate, mais quid de l'initialisation des sessions, de la gestion des transactions et des exceptions techniques?
- si certains objets du domaine doivent être persistants, où est géré leur identifiant technique?
- d'autre part, si les objets du domaine doivent être passés par valeur à travers un framework de distribution, pourquoi ne sont-ils pas sérialisables?

Et bien entendu, certains services techniques "de support" sont également importants et souvent omis ou sous-spécifiés par la spécification technique (c'est exactement notre cas!):

- la gestion des traces
- l'optimisation des services coûteux
- les statistiques d'exécution de notre application

Enfin du point de vue de la conception Objet un certain nombre de précautions n'ont pas été prises. En particulier, on ne passe pas par une **AbstractFactory** pour récupérer une référence vers l'interface d'un

DAO. Comment migrer d'une implémentation à une autre des DAO sans ce découplage?

Aucun de ces "oublis" n'est anodin. Toutes ces choses sont indispensables à notre application mais les implémenter directement dans le code artisanal constituerait une pollution importante. Cela serait nuisible:

- à la simplicité et donc à la facilité qu'auraient de nouveaux membres d'un projet à appréhender le code
- à l'évolutivité car plus de code signifie souvent plus de modifications à apporter en cas d'évolution
- à la cohérence du code car des redondances apparaîtraient forcément si l'on devait gérer tout cela manuellement et les évolutions successives du logiciels risqueraient de briser la cohérence de ces redondances
- à la productivité puisqu'il y aurait plus de code à écrire

Non, vous l'avez compris, tout ceci est motivé: nous allons gérer chacun des points mentionnés en tissant plusieurs Aspects sur le PetShop de base.

11.3. Le métier à tisser

Passons maintenant en revue tous les Aspects mis en oeuvre dans le PetShopAOP. Pour les premiers, nous décrirons à la fois le comportement du tisseur et l'objectif des Aspects; puis cela sera de plus en plus naturel donc nous finirons par nous focaliser sur les Aspects et ferons passer AspectDNG au second plan.

Nous irons du tissage le plus simple au plus subtil. Ce n'est pas forcément l'approche la plus logique d'un point de vue architectural mais cela nous permettra de nous approprier progressivement les outils, conceptuellement et syntaxiquement.

11.3.1. Objets non identifiés

Certains objets du domaine vont être rendus persistants par NHibernate. A cette fin, il est indispensable que chacun d'eux dispose d'une propriété (ou du moins d'un attribut) stockant un identifiant unique utilisé pour assurer la cohérence entre le modèle objet en mémoire et le modèle de stockage relationnel en base de données. Dans certaines situations, les couches utilisatrices (typiquement la couche de présentation) auront certainement besoin d'avoir accès en lecture à cette information donc disposer d'une propriété readonly ne fera pas de mal. Il pourrait également s'avérer intéressant de redéfinir le comportement de la méthode GetHashCode() pour qu'elle renvoie également cette valeur unique (il existe quelques contre-indications à cette pratique mais ne nous attardons ici pas sur ces détails).

Si nous regroupons dans une classe tout ce que nous aimerions "ajouter" aux objets du domaine, cela pourrait donner quelque chose de ce genre:

```
01. namespace PetShopAOP.Aspects {
02.     public class Domain {
03.         private int m_Id;
04.
05.                 public int Id {
06.             get { return m_Id; }
```

```
07.         }
08.
09.         public override int GetHashCode() {
10.             return m_Id;
11.         }
12.     }
13. }
```

Domain.cs

Il ne reste plus qu'à indiquer à AspectDNG que nous souhaitons **greffer ces membres** (m_Id, Id et GetHashCode()) **sur les objets du domaine**. Cela pourrait se faire dans le code précédent en ajoutant quelques Attributes, mais ce serait redondant. En réalité, nous souhaitons greffer **tous** les membres de la classe Domain sur **tous** les objets du domaine. Ce genre de besoin doublement ensembliste ne peut être exprimé que de manière externe, dans le fichier de configuration AspectDNG.xml dont voici un extrait:

```
01. <!DOCTYPE AspectDngConfig[
02.     <!ENTITY ServiceTypes
03.      "//Type
04.         [@namespace='PetShopAOP.Service']
05.         [contains(@name, 'Service')]">
06.     <!ENTITY DAOTypes
07.      "//Type
08.         [@namespace='PetShopAOP.Data']
09.         [contains(@name, 'DAO')]">
10.     <!ENTITY DomainTypes
11.      "//Type
12.         [@namespace='PetShopAOP.Domain']" >
13.     <!ENTITY AspectTypes
14.      "//Type
15.         [@namespace='PetShopAOP.Aspects']" >
16. ]>
17.
18. <AspectDngConfig
19.      xmlns="http://www.dotnetguru.org/AspectDNG"
20.     debug="false"
21.      logIlml="true"
```

```
22.      logIlmlPath="../Service/bin/ilml-log.xml"
23.     logWeaving="true"
24.      logWeavingPath="../Service/bin/weaving-log.xml">
25.     <BaseAssembly>
26.        ../Service/bin/PetShopAOP.exe
27.     </BaseAssembly>
28.     <AspectsAssembly>
29.        ../Service/bin/PetShopAOPAspects.dll
30.     </AspectsAssembly>
31.
32.        <Advice>
33.        <Insert
34.            aspectXPath="AspectTypes;[@name='Domain']/*"
35.            targetXPath="DomainTypes;"/>
36.              </Advice>
37. </AspectDngConfig>
```

AspectDNG.XML (version 1, partiel)

Dès lors, chaque objet du domaine peut être rendu persistant par NHibernate. Simple, non? Et cela nous aura évité de copier/coller ces membres manuellement dans chaque classe persistante.

11.3.2.Infrastructure de distribution

Dans notre diagramme d'architecture technique, nous avons prévu de distribuer la couche de service. Une question se pose dès lors: quels types de données échange la couche de service avec la couche de présentation? Voici quelques temps, le réflexe à la mode aurait été de placer le **Design Pattern DTO** (Data Transfer Object). Mais celui-ci a récemment été très **critiqué**: appliqué de manière systématique, il n'avait pas forcément de valeur ajoutée importante pour chaque type de donnée échangé par rapport à un graphe d'objets sérialisés; or son élaboration et surtout sa maintenance peuvent s'avérer coûteuses. Nous avons donc choisi ici de nous passer de ce Pattern et de rendre simplement les objets du domaine sérialisables.

Il nous faut donc répondre à deux besoins particuliers:

- rendre les objets du domaine **sérialisables**
- rendre les services **distribués** (par .NET Remoting ici mais le problème serait similaire avec d'autres infrastructures de distribution)

Il est très simple de régler le premier: ce n'est qu'une petite extension du tissage précédent. Jetons un oeil sans détour:

```
<!DOCTYPE AspectDngConfig[
    <!ENTITY ServiceTypes
     "//Type
       [@namespace='PetShopAOP.Service']
       [contains(@name, 'Service')]">
    <!ENTITY DAOTypes
     "//Type
       [@namespace='PetShopAOP.Data']
       [contains(@name, 'DAO')]">
    <!ENTITY DomainTypes
     "//Type
       [@namespace='PetShopAOP.Domain']" >
    <!ENTITY AspectTypes
     "//Type
       [@namespace='PetShopAOP.Aspects']" >
]>

<AspectDngConfig
     xmlns="http://www.dotnetguru.org/AspectDNG"
    debug="false"
     logIlml="true"
     logIlmlPath="../Service/bin/ilml-log.xml"
    logWeaving="true"
     logWeavingPath="../Service/bin/weaving-log.xml">
    <BaseAssembly>
       ../Service/bin/PetShopAOP.exe
    </BaseAssembly>
    <AspectsAssembly>
       ../Service/bin/PetShopAOPAspects.dll
    </AspectsAssembly>

```

```
32.         <Advice>
33.         <!-- Make Domain classes serializable
34.             and insert the ID management on each of them -->
35.         <MakeSerializable
36.             targetXPath="DomainTypes;"/>
37.         <Insert
38.             aspectXPath="AspectTypes;[@name='Domain']/*"
39.             targetXPath="DomainTypes;"/>
40.               </Advice>
41. </AspectDngConfig>
```

AspectDNG.XML (version 2, partiel)

Pour ce qui est de rendre les services distribués, décomposons le besoin en étapes plus simples. Il nous faut:

- faire en sorte que chaque service hérite de **MarshalByRefObject**
- si les services sont susceptibles de lever des exceptions, celles-ci doivent être rendues sérialisables tout comme les objets du domaine mais elles doivent également disposer d'un constructeur particulier de manière à ce que l'infrastructure côté client puisse les désérialiser sans encombre (un constructeur prenant en paramètres SerializationInfo et StreamingContext).
- enfin, il faudra bien sûr démarrer le serveur .NET Remoting, choisir un port d'écoute, un protocole de transport, un encodage... et se mettre en attente des requêtes des clients.

Cette fois, notre besoin de tissage n'est pas doublement ensembliste ni très réutilisable. Nous allons donc spécifier les étapes du tissages dans des **Attributes** situés dans la classe d'Aspect elle-même. Jugez plutôt:

```
01. using System;
02. using System.Configuration;
03. using System.Reflection;
04. using System.Runtime.Serialization;
05. using System.Runtime.Remoting;
06. using System.Runtime.Remoting.Channels;
07. using System.Runtime.Remoting.Channels.Tcp;
08. using System.Runtime.Remoting.Channels.Http;
09. using PetShopAOP.Service;
```

```
10. using AspectDNG;
11.
12. namespace System{
13.     [SetBaseType
14.         (PetShopAOP.Aspects.Constants.ServiceClassesXPath)]
15.     public class MarshalByRefObject {
16.     }
17. }
18.
19. namespace PetShopAOP.Aspects {
20.     [MakeSerializable(Constants.ExceptionsXPath)]
21.     public class DistributedException{
22.         [Insert(Constants.ExceptionsXPath)]
23.         protected DistributedException
24.             (SerializationInfo si, StreamingContext sc){}
25.           }
26.
27.     public class Server{
28.         [Insert(Constants.EntryPointClassXPath)]
29.         public static void InitRemotingServices(){
30.             int tcpPort = int.Parse
31.                 (ConfigurationSettings.AppSettings["server.tcp.port"]);
32.             int httpPort = int.Parse
33.                 (ConfigurationSettings.AppSettings["server.http.port"]);
34.
35.             ChannelServices.RegisterChannel
36.                 (new TcpChannel(tcpPort));
37.             ChannelServices.RegisterChannel
38.                 (new HttpChannel(httpPort));
39.
40.             Type sampleServiceType  = typeof(CatalogService);
41.             foreach(Type t in
42.                  sampleServiceType.Assembly.GetTypes()){
43.                 if (t.BaseType  == typeof(MarshalByRefObject)){
44.                     RemotingConfiguration.RegisterWellKnownServiceType
45.                         (t, t.Name, WellKnownObjectMode.Singleton);
46.                 }
47.             }
```

```
48.
49.             Console.WriteLine("Server started");
50.             Console.WriteLine("Hit <<Enter>> to stop");
51.             Console.ReadLine();
52.         }
53.
54.         [InlineBeforeReturn
55.             (Constants.EntryPointClassXPath
56.              + "/Method[@name='Main']")]
57.         public static void StartServer(){
58.             InitRemotingServices();
59.         }
60.     }
61. }
```

DistributedService.cs

Profitons-en pour consulter la classe dans laquelle nous avons consigné toutes les expressions XPath utilisées par les Attributs de tissage:

```
01. using System;
02.
03. namespace PetShopAOP.Aspects {
04.     public class Constants {
05.         public const string DomainAndServiceClassesXPath =
06.             "xpath: //Type[@namespace='PetShopAOP.Domain' or "
07.             + " (@namespace='PetShopAOP.Service' "
08.             +"and contains(@name, 'Service'))]";
09.         public const string ServiceClassesXPath =
10.             "xpath: //Type[@namespace='PetShopAOP.Service']"
11.             + "[contains(@name, 'Service')]";
12.         public const string DAOClassesXPath =
13.             "xpath: //Type[@namespace='PetShopAOP.Data']"
14.             +"[contains(@name, 'DAO')]";
15.         public const string EntryPointClassXPath =
16.             "xpath:
//Type[@fullName='PetShopAOP.Service.MainClass']" ;
17.         public const string ExceptionsXPath =
18.             "xpath: //Type[starts-with(@namespace,'PetShopAOP') "
19.             +"and contains(@name, 'Exception')]";
```

```
20.     }
21. }
```
Constants.cs

Maintenant que nous disposons de tous les éléments, détaillons le comportement de l'**Aspect DistributedService**:

- **[SetBaseType]** permet de modifier la relation d'héritage des classes cibles. Ainsi, ces classes hériteront de MarshalByRefObject et pourront être distribuées par .NET Remoting
- Nous avions déjà rencontré **[MakeSerializable]** (dans le document AspectDNG.xml, mais l'effet est le même bien entendu), cette fois son objectif est de rendre sérialisables les exceptions du niveau Service
- **[Insert(Constants.ExceptionsXPath)]** permet de greffer le constructeur nécessaire à la désérialisation sur toutes les exceptions du niveau Service
- **[Insert(Constants.EntryPointClassXPath)]** nous permet de greffer, sans aucune modification, la méthode InitRemotingServices() sur la classe MainClass. Cette méthode procède au paramétrage de l'infrastructure .NET Remoting et à la mise en attente bloquante du Thread qui l'exécutera.
- **[InlineBeforeReturn(Constants.EntryPointClassXPath + "/Method[@name='Main']")]** fait en sorte qu'après le comportement nominal de la méthode Main(), on déclenche l'invocation de InitRemotingServices() provoquant ainsi le lancement du serveur .NET Remoting.

A l'issue de ce tissage un peu plus élaboré que le précédent, nos services sont devenus distribués et le lancement du serveur est aussi simple que... l'exécution de l'assembly .EXE elle-même puisque sa méthode Main() a été modifiée et s'occupe de tout.

Bien sûr, certains points sont contestables dans ce tissage simpliste. Par exemple, nous nous sommes permis de modifier la relation d'héritage des services en partant du principe qu'ils héritaient tous de la classe System.Object. Rediriger l'héritage n'a donc pas de répercussion sur le comportement nominal des services. Mais dans l'absolu, il faudrait

parcourir leur graphe d'héritage et ne rediriger vers MarshalByRefObject que les services dont le père est System.Object, ce qui leur permettrait de tirer parti de l'héritage traditionnel malgré le tissage de notre Aspect.

11.3.3.Optimisation

Dans notre application, certains services risquent d'être sollicités très souvent: en particulier les services de requétage sur le catalogue d'animaux. Nous pourrions nous reposer sur les mécanismes de cache de la base de données pour optimiser cela mais nous subirions systématiquement le mapping Objet/Relationnel! Dans ce cas, faisons confiance à NHibernate qui gère un cache objet en mémoire... mais nous traverserions tout de même la couche de service, d'accès aux données et nous solliciterions NHibernate.

Nous pouvons faire mieux et à peu de frais: il suffirait en première approximation de **stocker les résultats des invocations de méthodes dans une structure de cache temporaire pendant une certaine période** (10 secondes, 20 minutes: à définir). Implémenter ce cache peut être aussi simple que de partager une Hashtable. Mais comment faire en sorte que chaque méthode de recherche:

- stocke automatiquement son résultat dans le cache
- et ne s'exécute que si les paramètres qu'on lui passe ne correspondent pas déjà à une entrée du cache car dans ce cas il suffit de renvoyer la valeur mémorisée.

Eh bien il suffit d'**intercepter l'invocation de toutes les méthodes de recherche dans les classes de service**, et d'implémenter cette logique de cache dans un Aspect avant de propager l'invocation de la méthode visée ou au contraire de court-circuiter son invocation et de renvoyer le résultat de l'invocation précédente. Voici comment:

```
01. using System;
02. using System.Text;
03. using System.Collections;
04. using System.Reflection;
05. using System.Configuration;
06. using PetShopAOP.Data;
```

```
07. using AspectDNG;
08. using NHibernate;
09.
10. namespace PetShopAOP.Aspects {
11.     [Insert("")]
12.     public class CacheManager {
13.         public readonly static IDictionary Cache = new Hashtable();
14.         public readonly static int Period = 10;
15.
16.         static CacheManager(){
17.             Period = int.Parse
18.                 (ConfigurationSettings.AppSettings
19.                 ["cache.period.in.seconds"]);
20.         }
21.
22.     }
23.
24.     public class Optim {
25.         [GenericAroundBody
26.             (Constants.ServiceClassesXPath +
27.              "/Method[starts-with(@name, 'Find') or "
28.             +"starts-with(@name, 'Load')][not(contains(@name,
'_'))]")]
29.         public object CacheLastResult
30.             (object[] data, MethodBase targetOperation){
31.             object result = null;
32.
33.             // Create a full name of the operation,
34.              // including parameters values
35.             // since the operation may have different
36.              // results for different parameters
37.             StringBuilder buf = new StringBuilder
38.                 (targetOperation.DeclaringType +
targetOperation.Name);
39.             foreach(object param in data){
40.                 buf.Append(param);
41.             }
42.             string fullName = buf.ToString();
```

```
43.             string fullNameLastExec = fullName + "-lastExec";
44.
45.             // Decide to re-execute the target operation or not
46.             bool shouldExecute = !
47.                 CacheManager.Cache.Contains(fullNameLastExec);
48.             if (! shouldExecute){
49.                 DateTime lastExec = (DateTime)
50.                     CacheManager.Cache[fullNameLastExec];
51.                 shouldExecute =
52.                     ((DateTime.Now - lastExec).Seconds
53.                     > CacheManager.Period);
54.             }
55.
56.             if (shouldExecute){
57.                 result = targetOperation.Invoke(this, data);
58.             }
59.
60.             // Only lock the Cache to get or set data.
61.              // Hence, in fact some (few) threads may run
62.              // the operation in parallel with the same parameters,
63.              // but this is better for parallelism
64.             lock(CacheManager.Cache){
65.                 if (shouldExecute){
66.                     CacheManager.Cache[fullNameLastExec] =
DateTime.Now;
67.                     CacheManager.Cache[fullName] = result;
68.                 }
69.                 else{
70.                     result = CacheManager.Cache[fullName];
71.                 }
72.                     }
73.
74.             return result;
75.         }
76.
77.         [InlineAtStart
78.             (Constants.ServiceClassesXPath
79.              + "/Method[not(starts-with(@name, 'Find') "
```

```
80.             + "or starts-with(@name, 'Load'))]"
81.             + "[not(contains(@name, '_'))]")]
82.         public void InvalidateCacheFromService(){
83.             CacheManager.Cache.Clear();
84.         }
85.     }
86. }
```

Optim.cs

Vous aurez compris que:

- La classe **CacheManager** est copiée sans modification particulière et sera donc disponible dans l'assembly cible à l'exécution
- La méthode **CacheLastResult(object[] data, MethodBase targetOperation)** encapsule toutes les invocations de méthodes (de recherche) de la couche service. Le premier paramètre **data** nous donne accès aux valeurs de tous les paramètres passés à la méthode cible et le second **targetOperation** est une référence permettant de connaître et d'invoquer la méthode cible au moment voulu dans notre "méthode-intercepteur".
- **InvalidateCacheFromService()** invalide notre cache dès qu'une méthode autre que de consultation est invoquée. C'est un peu pessimiste, il serait tout à fait possible de ne tisser cette invalidation que dans certains cas, à condition de bien connaître le comportement fonctionnel de notre application.

Un argument revient souvent dans les débats traitant de ce genre de technique d'optimisation: finalement, ne perdons-nous pas plus de temps ou d'espace mémoire à mettre en oeuvre ces optimisations plutôt que de ne pas le faire, tout simplement? Eh bien n'hésitons plus: grâce à l'Aspect précédent, activer ou désactiver le mécanisme de cache ne coûte qu'un tissage, aucune modification de code!

11.3.4.Sessions, transactions et exceptions

En apercevant tout à l'heure le code d'un DAO, vous vous êtes certainement demandé: "mais quand est-ce que les Sessions NHibernate

sont initialisées? Je vois bien que le code utilise une Session, mais à aucun moment on ne gère son cycle de vie".

Or en y réfléchissant, la notion de Session NHibernate est indissociable de celle de Transaction et de la gestion des exceptions. En effet, si un service donne lieu à des modifications persistantes par le bais des DAO, il est important que cela se fasse dans une transaction unique. Et donc qu'une Session ait déjà été ouverte. D'autre part, si au cours de l'exécution d'un service un DAO déclenche une exception, il est important que toutes les modifications effectuées au préalables soient elles-aussi annulées, de manière transactionnelle. Bref, ces trois facettes techniques sont liées.

Voici les principes que nous avons retenus:

- les Sessions NHibernates sont initialisées lors du déclenchement d'un service par la couche de présentation. Par contre, un service invoqué par un autre service réutilisera la même Session.
- les Transactions sont initialisées par le premier DAO effectuant une opération de modification persistante. Toutes les autres sollicitations de DAO dans le cadre d'exécution du même service seront incluses dans cette transaction.
- à la fin de l'exécution d'un service, si une Transaction est ouverte elle doit être validée (commit). Puis, dans tous les cas, la Session est à son tour validée puis fermée.
- si une exception est levée par un DAO, un objet du domaine ou un service, le service de plus haut niveau (celui qui a été déclenché par la couche de présentation, et donc qui a initialisé la Session) doit annuler la Transaction en cours si elle existe et mettre fin à la Session.

Enfin, puisque les Sessions ne sont plus gérées au niveau des DAO, il faut trouver une technique pour donner à chaque DAO le moyen de manipuler la Session courante, initialisée par un service englobant. De façon plus générale, tout acteur déclenché directement ou indirectement par un Service doit pouvoir manipuler la Session courante. La technique généralement employée pour partager une ressource technique entre tous les acteurs traversés par un graphe d'appel est de placer cette ressource sur le Thread Local Storage (TLS). Ainsi, associée au Thread,

la ressource l'accompagne et reste disponible quel que soit la méthode en cours d'exécution.

11.3.4.1. Cycle de vie

Mettons tout cela en pratique: nous avons deux Aspects à tisser, le premier gérant le cycle de vie des Sessions et des Transactions NHibernate, le second donnant accès à la Session courante à chaque DAO par le biais de son attribut m_session. Pour simplifier le travail du premier, nous nous appuierons sur une classe utilitaire, NHibernateHelper, qui se charge le paramétrage de NHibernate, crée automatiquement le schéma de la base de données et enfin initialise une Fabrique de Sessions NHibernate:

```
01. using System;
02. using NHibernate;
03. using NHibernate.Cfg;
04. using NHibernate.Tool.hbm2ddl;
05.
06. namespace PetShopAOP.Data {
07.     public class NHibernateHelper {
08.         // No multithreading problem here,
09.          // no need to double check and lock
10.         private static NHibernateHelper m_Instance;
11.         public static NHibernateHelper Instance{
12.             get{
13.                 return (m_Instance == null)
14.                 ? m_Instance = new NHibernateHelper
15.                   ("GlobalModel.hbm.xml")
16.                 : m_Instance;
17.             }
18.         }
19.         private NHibernateHelper(){}
20.
21.         public readonly ISessionFactory m_factory;
22.
23.         private NHibernateHelper(string configPath){
24.             Configuration m_config = new Configuration();
25.             m_config.AddXmlFile(configPath);
26.                 m_factory = m_config.BuildSessionFactory();
```

```
27.
28.             // Create the database schema
29.             SchemaExport export = new SchemaExport(m_config);
30.             export.SetOutputFile("generated-database-schema.sql");
31.             export.Create(false, true);
32.         }
33.     }
34. }
```

NHibernateHelper.cs

Notre premier Aspect s'appuie sur cette classe pour gérer le cycle de vie des Sessions:

```
01. using System;
02. using System.Collections;
03. using System.Threading;
04. using System.Reflection;
05. using PetShopAOP.Data;
06. using AspectDNG;
07. using NHibernate;
08.
09. namespace PetShopAOP.Aspects {
10.     [Insert("")]
11.     public class NHibernateSessionManager{
12.         [ThreadStatic]
13.         public static ISession Session;
14.     }
15.
16.     public class NHibernateUser {
17.         // This m_session will be defined on target objects
18.         [ThreadStatic]
19.         private ISession m_session;
20.
21.         [InlineAtStart(Constants.DAOClassesXPath
22.             + "/Method[not(contains(@name, '_'))]")]
23.         public void NHibernateAutomaticSessionGetter(){
24.             m_session = NHibernateSessionManager.Session;
25.         }
26.
```

```
27.             [GenericAroundBody(Constants.ServiceClassesXPath
28.          + "/Method[not(contains(@name, '_'))]")]
29.       public object NHibernateAutomaticSession
30.          (object[] data, MethodBase targetOperation){
31.          object result = null;
32.
33.          if (NHibernateSessionManager.Session == null){
34.             ISession session =
35.                 NHibernateHelper.Instance.m_factory.OpenSession();
36.             m_session =
37.                 NHibernateSessionManager.Session =
38.                 session;
39.
40.             // Start session and transaction.
41.              // They will be closed at the same level.
42.             ITransaction transaction = session.BeginTransaction();
43.             try{
44.                // Method invocation propagation
45.                result = targetOperation.Invoke(this, data);
46.                transaction.Commit();
47.             }
48.             catch(Exception e){
49.                transaction.Rollback();
50.                throw e;
51.             }
52.
53.                         session.Flush();
54.             session.Close();
55.             m_session =
56.                 NHibernateSessionManager.Session =
57.                 null;
58.          }
59.          else{
60.             // Enlist this operation in the current session/transaction
61.             m_session = NHibernateSessionManager.Session;
62.
63.             // Simple propagation
64.             result = targetOperation.Invoke(this, data);
```

```
65.             }
66.             return result;
67.         }
68.     }
69. }
```

NHibernateUser.cs

Concentrons-nous tout d'abord sur la méthode **NHibernateAutomaticSession**. Il s'agit d'un intercepteur qui sera déclenché à l'appel de toute méthode de service. Si l'invocation provient de la couche de présentation, alors aucune Session NHibernate n'existe; l'intercepteur en initialise une, et en profite pour initialiser immédiatement une Transaction. Ce modèle est plus coûteux que le précédent car il implique une lecture transactionnelle; si un tel luxe de cohérence de données n'est pas indispensable, il serait trivial d'attendre la première modification (cette fois, au niveau d'un DAO) pour débuter cette Transaction.

Une fois les initialisations terminées, notre intercepteur propage l'invocation de méthode visée par la couche de présentation: **targetOperation.Invoke(this, data)** (à noter que le "this" sera l'objet sur lequel le tissage aura été opéré, c'est-à-dire le service). Puis à l'issue de cette propagation ferme la Transaction et la Session.

Dans le cas où la Session a déjà été initialisée, c'est nettement plus simple puisqu'il n'y a qu'à propager l'invocation de méthode. Vous remarquerez que dans les deux cas, l'attribut **m_session** est affecté et permet au service de manipuler la Session NHibernate. A priori, aucune manipulation explicite de cet attribut n'est nécessaire dans un service mais cela reste possible.

11.3.4.2. Manipulation des Sessions

Jusqu'à présent, nous nous sommes focalisés sur la couche de Service qui gère le cycle de vie des Sessions. Mais une question reste sans réponse: comment feront les DAO pour interagir avec la Session courante? Devront-ils manipuler le TLS? Cela n'est-il pas un peu trop technique?

Non, vous allez voir qu'au contraire le code des DAO sera excessivement simple. Le seul besoin "technique" dans un DAO est de

déclarer un attribut **Session m_session**. Souvenez-vous du code de ProductDAO que nous rappelons ici pour nous raffraîchir la mémoire:

```
01. using System;
02. using System.Collections;
03. using PetShopAOP.Domain;
04. using NHibernate;
05.
06. namespace PetShopAOP.Data {
07.     public class ProductDAO{
08.         // No multithreading problem here,
09.          // no need to double check and lock
10.         private static ProductDAO m_Instance;
11.         public static ProductDAO Instance{
12.             get{
13.                 return (m_Instance == null)
14.                     ? m_Instance = new ProductDAO()
15.                     : m_Instance;
16.             }
17.         }
18.         private ProductDAO(){}
19.
20.         private ISession m_session;
21.
22.         public Product Save(Product p) {
23.             m_session.Save(p);
24.             return p;
25.         }
26.
27.         public Product Load(object id) {
28.             return (Product) m_session.Load
29.                 (typeof(Product), Convert.ToInt32(id));
30.         }
31.
32.         public void Delete(object id) {
33.             m_session.Delete
34.                 ("from Product where id = ?",
35.                  id, NHibernate.NHibernate.Int32);
36.         }
```

```
37.
38.         public void Delete(Product c) {
39.             m_session.Delete(c);
40.         }
41.     }
42. }
```

ProductDAO.cs

Rien de plus simple en effet. Mais vous l'aurez compris, cela signifie forcément que:

- cet attribut doit être multi-thread safe, ou plutôt attaché au Thread Local Storage (TLS) pour garantir la cohérence multi-tâche de notre application. Or nous n'avons pas déclaré m_session de manière à ce qu'il soit attaché au TLS! Que va-t-il se passer?
- avant chaque invocation d'une méthode sur un DAO, il faut absolument s'assurer que l'attribut m_session pointe bel et bien sur la Session courante.

Tout ceci peut être pris en charge par un Aspect. Son premier rôle sera de déclarer un attribut Session m_session attaché au TLS sur tous les DAO. Si un attribut Session m_session existe déjà dans un DAO, il suffit de le supprimer et de le redéclarer convenablement. Voyons cela en étendant encore la configuration du tissage:

```
01. <!DOCTYPE AspectDngConfig[
02.     <!ENTITY ServiceTypes
03.      "//Type
04.         [@namespace='PetShopAOP.Service']
05.         [contains(@name, 'Service')]">
06.     <!ENTITY DAOTypes
07.      "//Type
08.         [@namespace='PetShopAOP.Data']
09.         [contains(@name, 'DAO')]">
10.     <!ENTITY DomainTypes
11.      "//Type
12.         [@namespace='PetShopAOP.Domain']" >
13.     <!ENTITY AspectTypes
14.      "//Type
```

```
15.         [@namespace='PetShopAOP.Aspects']">
16. ]>
17.
18. <AspectDngConfig
19.      xmlns="http://www.dotnetguru.org/AspectDNG"
20.     debug="false"
21.      logIlml="true"
22.      logIlmlPath="../Service/bin/ilml-log.xml"
23.     logWeaving="true"
24.      logWeavingPath="../Service/bin/weaving-log.xml">
25.     <BaseAssembly>
26.         ../Service/bin/PetShopAOP.exe
27.     </BaseAssembly>
28.     <AspectsAssembly>
29.         ../Service/bin/PetShopAOPAspects.dll
30.     </AspectsAssembly>
31.
32.         <Advice>
33.         <!-- Make Domain classes serializable
34.             and insert the ID management on each of them -->
35.         <MakeSerializable
36.             targetXPath="DomainTypes;"/>
37.         <Insert
38.             aspectXPath="AspectTypes;[@name='Domain']/*"
39.             targetXPath="DomainTypes;"/>
40.
41.                         <!-- Delete m_session fields from target objects
42.             (since they are not ThreadStatic) -->
43.         <Delete
44.
targetXPath="ServiceTypes;/Field[@name='m_session']"/>
45.
46.               <!-- Re-insert thread-static field -->
47.         <Insert
48.             aspectXPath="AspectTypes;[@name='NHibernateUser']
49.               /Field[@name='m_session']"
50.             targetXPath="ServiceTypes;"/>
51.         <Insert
```

```
52.             aspectXPath="AspectTypes;[@name='NHibernateUser']
53.                 /Field[@name='m_session']"
54.             targetXPath="DAOTypes;"/>
55.     </Advice>
56. </AspectDngConfig>
```
AspectDNG.XML (version 3, partiel)

D'autre part, il faut s'assurer que m_session pointe bien sur la session courante. Mais nous l'avons déjà implémenté dans l'Aspect NHibernateUser:

```
01. using System;
02. using System.Collections;
03. using System.Threading;
04. using System.Reflection;
05. using PetShopAOP.Data;
06. using AspectDNG;
07. using NHibernate;
08.
09. namespace PetShopAOP.Aspects {
10.     public class NHibernateUser {
11.         // This m_session will be defined on target objects
12.         [ThreadStatic]
13.         private ISession m_session;
14.
15.             [InlineAtStart(Constants.DAOClassesXPath
16.              + "/Method[not(contains(@name, '_'))]")]
17.         public void NHibernateAutomaticSessionGetter(){
18.             m_session = NHibernateSessionManager.Session;
19.         }
20.             }
21. }
```
NHibernateUser.cs (partiel)

Une petite précision a toute son importance ici, elle concerne les performances: quel que soit le DAO, quelle que soit la méthode invoquée et ses paramètres, notre aspect se borne toujours à faire la même chose. Dans une telle situation, il est donc superflu de réifier la pile ou les paramètres d'invocation de la méthode en un tableau d'objets

puisqu'on ne l'utiliserait pas. Nous attirons donc votre attention sur la technique de tissage : **[InlineAtStart]**. Son comportement est audacieux: il consiste à copier directement le corps de la méthode NHibernateAutomaticSessionGetter au début du corps de toutes les méthodes cibles (dans les DAO)! Ainsi, que nous écrivions du code une seule fois dans l'Aspect ou que nous le copions-collons manuellement dans toutes les méthodes de tous les DAO, les performances seront **exactement** les mêmes. D'ailleurs, si cela pique votre curiosité, n'hésitez pas à décompiler le code tissé et à vérifier cela par vous-mêmes. Nous n'avons besoin d'aspects **génériques** (à réification de pile) que si leur comportement est **contextuel**, qu'il dépend des valeurs de paramètres des méthodes interceptées; **dans les autres cas, tissez en mode "inline"**, les performances seront bien meilleures!

Faisons le point: Sessions, Transactions et Exceptions sont maintenant gérées automatiquement au niveau de la couche de Service, par un Aspect. Et du point de vue de l'utilisation dans nos DAO, il suffit de déclarer un attribut Session m_session et de l'utiliser dans nos méthodes; il sera initialisé automatiquement et pointera toujours vers la Session courante, elle-même stockée dans le TLS. Difficile de rendre le code artisanal plus simple!

11.3.5.Services de support

Vous savez tout des possibilités offertes par l'AOP et en particulier par AspectDNG. Forts de ces connaissances, rien ne vous empêcherait d'aller plus loin et d'implémenter bien d'autres services par simple tissage d'Aspects. En guise d'exemple, nous vous proposons de tracer automatiquement l'exécution du PetShopAOP (en nous appuyant sur le framework log4net, de manière à ce que le niveau de détail de ces traces reste finement paramétrable):

```
01. using System.Reflection;
02. using System.Collections;
03. using System.Text;
04. using AspectDNG;
05. using log4net;
06.
```

```
07. namespace PetShopAOP.Aspects {
08.     public class Traces {
09.         [GenericAroundBody
10.             (Constants.ServiceClassesXPath
11.              + "/Method[not(contains(@name, '_'))]")]
12.         public object TraceAtStart
13.             (object[] data, MethodBase targetOperation){
14.             // Get method name
15.              // (Delegate methods may look
16.              // like "FindAll_987987-esf9I987-...")
17.             string baseName = targetOperation.Name.Split('_')[0];
18.
19.                 // Get method parameter values
20.             StringBuilder buf =
21.                  new StringBuilder(baseName).Append("(");
22.             for(int i=0; i<data.Length; i++){
23.                 buf.Append(data[i]);
24.                 if (i < data.Length - 1){
25.                     buf.Append(", ");
26.                 }
27.             }
28.             buf.Append(")");
29.
30.                 // Log using lo4net
31.             LogManager.GetLogger(GetType()).Info(buf);
32.
33.                  // Delegate the method invocation
34.             return targetOperation.Invoke(this, data);
35.         }
36.     }
37. }
```

Traces.cs

Un peu plus subtil, amusons-nous à ajouter un mécanisme de statistiques en temps réel sur le PetShopAOP. En deux mots, cela consiste à compter le nombre d'invocations de méthodes effectuées au fur et à mesure du déroulement de l'application ainsi que le temps passé dans chacune d'entre elles. Les informations collectées sont stockées en

mémoire, mais sont également rendues disponibles à distance par un nouveau service .NET Remoting lui aussi injecté par tissage. Du point de vue graphique, vous pourrez voir en pied de page du PetShopAOP un lien hypertexte qui mène au tableau récapitulant ces statistiques (ce n'est pas une fonctionnalité métier, nous sommes d'accord, le but n'était que d'offrir au technicien le moyen de voir quelles portions de l'application sont les plus sollicitées ou les plus gourmandes).

Voici l'Aspect gérant les statistiques:

```
01. using System;
02. using System.Collections;
03. using System.Reflection;
04. using AspectDNG;
05.
06. namespace PetShopAOP.Aspects {
07.     [Insert("")]
08.     [Serializable]
09.     public class MethodStat{
10.         private string m_Name;
11.         public string Name{ get{ return m_Name; } }
12.
13.             private Type m_DeclaringType;
14.         public Type DeclaringType{ get{ return m_DeclaringType; } }
15.
16.             public string FullName{
17.              get{
18.                 return string.Format
19.                   ("{0}.{1}::{2}",
20.                    m_DeclaringType.Namespace,
21.                    m_DeclaringType.Name,
22.                    m_Name);
23.                 }
24.              }
25.
26.             private int m_NbInvocations;
27.         public int NbInvocations{ get{ return m_NbInvocations; } }
28.
29.                 private long m_TotalExecutionTimeMillis;
```

```
30.         public long TotalExecutionTimeMillis
31.             { get{ return m_TotalExecutionTimeMillis; } }
32.
33.             public MethodStat(Type declaringType, string name){
34.             m_Name = name;
35.             m_DeclaringType = declaringType;
36.         }
37.
38.             public void Increment(long milliseconds){
39.             m_NbInvocations++;
40.             m_TotalExecutionTimeMillis += milliseconds;
41.         }
42.     }
43.
44.         public class BasicStats {
45.         [GenericAroundBody
46.             (Constants.DomainAndServiceClassesXPath
47.              + "/Method[not(contains(@name, '_'))]")]
48.         public object TimeElapsed
49.             (object[] data, MethodBase targetOperation){
50.             // Full names may look like "FindAll_987987-esf9I987-..."
51.             string baseName = targetOperation.Name;
52.             int index = baseName.IndexOf("_");
53.             if (index > -1){
54.                 baseName = baseName.Substring(0, index);
55.             }
56.
57.                 long start = DateTime.Now.Ticks;
58.             object result = targetOperation.Invoke(this, data);
59.             long end = DateTime.Now.Ticks;
60.
61.                 MethodStat newStat = new MethodStat
62.                 (targetOperation.DeclaringType, baseName);
63.             lock(BasicStatsServer.Stats){
64.                 MethodStat stat = BasicStatsServer.Stats
65.                     [newStat.FullName] as MethodStat;
66.                 if (stat == null){
67.                     stat = newStat;
```

```
68.                 BasicStatsServer.Stats[stat.FullName] = stat;
69.             }
70.             stat.Increment((end - start) / 10000);
71.         }
72.
73.             return result;
74.       }
75.    }
76.
77.       [Insert("")]
78.    public class BasicStatsServer : MarshalByRefObject {
79.       public static readonly IDictionary Stats = new Hashtable();
80.
81.          public ICollection GetStats(){
82.          return Stats.Values;
83.       }
84.    }
85. }
```

BasicStatsServer.cs

Le résultat graphique peut donner quelque chose de ce genre:

Figure 42. Backend Statistics

11.4. Installation et tests

Le PetShopAOP est l'application de référence d'AspectDNG. Elle est donc intégrée à la distribution d'AspectDNG lui-même. Ainsi donc, pour l'installer, rien de plus simple:

- Téléchargez **le code source** de la dernière version stable d'AspectDNG, disponible sur www.dotnetguru.org ou sur sourceforge.net.
- Extrayez le ZIP dans le répertoire de votre choix. Disons **c:\AspectDNG** par exemple.
- En ligne de commande, rendez-vous dans ce répertoire et lancez: **build**. Cela déclenche un script NAnt (NAnt est inclus dans la distribution d'AspectDNG)
- Si tout se passe bien, félicitations! Vous avez re-construit AspectDNG, passé ses tests unitaires, reconstruit le PetShopAOP et également passé ses propres tests.

Il ne vous reste plus qu'à:

- lancer le Backend (couche de service, domaine, accès aux données et base de données embarquée) en déclenchant: **C:\AspectDNG\AspectDNG\PetShopAOP \Service\bin\PetShopAOP.exe**. Au bout de quelques secondes, un message devrait vous dire que le serveur est démarré, et qu'il vous suffira de taper <<Entrée>> pour l'arrêter.
- déclarer un répertoire virtuel dans IIS:
  - Dans l'interface d'administration de IIS, choisissez <<Nouveau | Répertoire virtuel >> dans le menu dynamique
  - Alias (c'est important): **PetShopAOPWeb**
  - Répertoire: **C:\AspectDNG\AspectDNG\PetShopAOP\Web**
- lancer un navigateur Web et vous rendre à l'adresse suivante: *http://localhost/PetShopAOPWeb/Welcome.aspx*

Faites bon usage du PetShopAOP, son code est complètement ouvert et

n'attend que vos critiques et vos optimisations! Bons tissages!

Chapitre 12.

Conclusion

Nous espérons que ce livre vous aura permis de vous familiariser avec les principes et les opportunités offertes par la Programmation et la Conception Orientées Aspect. Le principal objectif était de prendre du recul par rapport à la conception Objet et de tirer parti de l'AOP pour aller encore plus loin dans l'abstraction et la souplesse de nos applications.

Certains mécanismes issus de l'AOP sont directement applicables aux projets actuels et certains tisseurs sont d'une qualité suffisante pour envisager sans crainte de les utiliser sur du code de qualité "production". D'autres principes relèvent encore du domaine de la recherche; ils attendent les premiers retours d'expérience et la confrontation avec les situations réelles pour évaluer certaines idées et pour ajuster les choix que nous pourrons industrialiser dans quelques années.

Pour conclure ce livre, nous aimerions vous faire les dernières recommandations d'usage.

Tout d'abord, parce que l'AOP est un domaine jeune et très dynamique, il vous faudra être attentif aux évolutions des outils et des pratiques dans les mois qui viennent. Tenez-vous au courant, de temps en temps, des nouveautés du monde de l'AOP; il est toujours moins difficile d'entretenir des connaissances une fois par mois, par exemple, que de les ré-acquérir dans un an ou deux.

Pour les plus passionnés, ceux qui ont déjà mis en œuvre l'AOP ou qui vont le faire après avoir refermé ce livre, n'hésitez pas à inventer de nouvelles techniques ou de nouveaux usages du tissage d'Aspects. Si les tisseurs ne sont pas suffisamment puissants pour vous, contactez leurs auteurs et aidez-les à enrichir les fonctionnalités de leurs outils: il est probable qu'eux-mêmes n'auront pas songé à tous les usages que l'on pourra envisager pour l'AOP.

Et pour tous, préparez-vous à rencontrer parfois certaines réticences voire résistances contre l'introduction de l'AOP sur vos projets. Comme d'habitude, la nouveauté effraie et le manque de connaissances et les a priori créent des blocages psychologiques; ce sera à vous d'expliquer la technologie à ceux qui la découvrent et qui ont quelque appréhension. Vous serez les porte-parole, les évangélistes comme disent certains, de cette nouvelle technologie prometteuse que sont les Aspects. Bon courage.

Thomas GIL

Chapitre 13.

Bibliographie

13.1. Conception Objet

Design Patterns, Elements of Reusable Object-Oriented Software [*Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides* - Addison-Wesley - 1994]

Object Oriented Software Construction, Second Edition [*Bertrand Meyer* - Prentice Hall - 2000]

Design By Contract [*Bertrand Meyer* - Prentice Hall - 2002]

13.2. UML

UML Distilled, Third Edition [*Martin Fowler* - Addison-Wesley - 2003]

UML 2 en action [*Pascal Roques, Franck Vallée* - Eyrolles - 2004]

Les cahiers du Programmeur UML, Modéliser un site e-commerce [*Pascal Roques* - Eyrolles - 2002]

13.3. AOP

Programmation orientée aspect pour Java / J2EE [*Renaud Pawlak, Jean-Philippe Retaillé, Lionel Seinturier* - Eyrolles - 2004]

AspectJ In Action [*Ramnivas Laddad* - Manning Publications - 2003]

13.4. Développement

Pratique de .NET 2.0 et C# 2.0 [*Patrick Smacchia* - Éditions O'Reilly - 2005]