République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mentouri de Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

N° d'ordre :

Série :

Ecole Doctorale en Informatique de L'Est

(Pôle Constantine)

Option : Génie Logiciel

Mémoire

Présenté en vue de l'obtention du diplôme de

Magistère en informatique

UN MODELE DE TRANSFORMATION DE DESIGN PATTERN VERS DES PROGRAMMES ORIENTES ASPECTS

Présenté par : Mr. BERKANE MOHAMED LAMINE

Dirigé par : Pr. BOUFAIDA MAHMOUD

Président :
Mme. Faiza Belala Maitre de conférence à l'université Mentouri de Constantine

Rapporteur :
Mr Mahmoud BOUFAÏDA Professeur à l'université Mentouri de Constantine

Examinateurs :

Mr. Nacereddine Zarour Maitre de conférence à l'université Mentouri de Constantine
Mr. Ramdane Maameri Maitre de conférence à l'université Mentouri de Constantine

Sommaire

Sommaire

Chapitre 02 : Présentation des Design Patterns.

Chapitre 03 : Une Approche de transformation des design pattern vers des programmes orientés aspects.

Chapitre 04 : Implémentation et études de cas

Annexes

Table des illustrations

Table des illustrations

Chapitre 1 :

Chapitre 2 :

Chapitre 3 :

Chapitre 4 :

Annexes :

Les tableaux

Introduction générale

Introduction générale

Les évolutions récentes dans le monde de l'informatique tel que la taille et la nature des logiciels créent une demande croissante de la part de l'industrie de concepteurs informaticiens. Pour cela, le génie logiciel est le domaine cible pour rassembler les activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi.

En effet, une bonne compréhension, lisibilité des logiciels, la possibilité de réutiliser ceux-ci, de les faire évoluer facilement et de manière fiable, sont des objectifs très recherchés en génie logiciel.

Pour atteindre ces objectifs, de nombreux modèles (Objets, Relationnel), méthodes de développement (Merise), langages (UML), et outils (Rational Rose, StarUML) ont été adoptés ces dernières années. Le modèle objet, ainsi que le langage UML sont les plus largement utilisées. Cependant, si ces modèles, méthodes, langages et outils ont des avantages certains, ils ont aussi des inconvénients et souffrent d'un certain nombre de limites. Ainsi, devant la complexité croissante des logiciels rendant leur développement plus difficile et plus coûteux. L'ingénierie des logiciels se tourne, de plus en plus, vers de nouvelles études, à savoir, l'ingénierie de nouvelles techniques et méthodes de développement facilitant l'évolution et favorisant la réutilisation. Ces nouvelles techniques et méthodes se traduisent aujourd'hui par l'existence d'approches de développement basées sur la notion d'aspect, l'ingénierie dirigée par les modèles (Model Driven Architecture) et l'utilisation des design pattern…etc.

L'utilisation des design patterns constituent comme un moyen efficace pour la conception, la composition de plusieurs types de composants réutilisables, et pour le développement de grands systèmes complexes. Ils permettent d'améliorer la qualité et la compréhension des programmes, facilitent leur évolution et augmentent notamment leur réutilisation.

Par ailleurs la programmation orientée aspect a émergé suite à différents travaux de recherche, dont l'objectif principal était d'améliorer la modularité des applications afin de faciliter leur évolution et leur réutilisation. L'implémentation de certains design patterns à l'aide de la programmation orientée aspect à permis de dégager certains avantages à savoir : une meilleure localisation, ainsi une meilleure réutilisation, la possibilité de faire participer un même objet à plusieurs design patterns de manière transparente pour lui (composition), et enfin le couplage faible entre l'application et les design patterns qu'elle utilise (adaptabilité).

Dans des domaines relativement neufs comme l'aspect, les informations sur le développement de bons logiciels font souvent défaut. Les développeurs ont tendance à réinventer la roue, ou bien simplement à commettre fréquemment de coûteuses erreurs. Sans disposer d'un répertoire de meilleures pratiques, le travail de développeur devient très difficile. Ainsi, l'apprentissage d'une bonne démarche de conception tel que l'utilisation des design patterns avec l'approche aspect est particulièrement difficile pour les nouveaux venus, parce que beaucoup d'entre eux n'ont jamais participé auparavant au développement des logiciels de grandes tailles.

Afin d'améliorer les applications objets tout en bénéficiant du paradigme aspect basée sur la notion design pattern, nous proposons un modèle de transformation, pour facilité la transformation des patterns de l'objet vers l'aspect. Cette automatisation permet au développeur de réduire le temps, l'effort ainsi le coût dans le développement des logiciels orientés aspects. Ce modèle de transformation peut être vue comme une génération de code orienté aspect. Il est vu aussi comme une opération de rétro-ingénierie.

Dans cette thèse, nous adopterons une organisation comportant quatre différents chapitres. Les deux premiers présentent l'état de l'art sur le concept de la POA et celui des design patterns.

- Dans le premier chapitre, nous allons présenter le concept Aspect. Nous allons faire un aperçu sur les avantages, ainsi les limites de la programmation orientée objet. Ensuite, nous introduisons les concepts de la programmation orientée aspect. Afin de montrer les apports de la programmation orientée aspect, nous présentons aussi le langage AspectJ (le langage de programmation qui supporte l'aspect) ainsi ses concepts. Nous terminons ce chapitre par la présentation du paradigme aspect au niveau conceptuel.

- Le deuxième chapitre, sera consacré au deuxième concept qui est le design pattern. Dans ce chapitre, nous allons présenter les différents types des patterns d'ingénierie, avec quelques exemples pour chaque type, ainsi leurs avantages dans le développement des systèmes. Nous illustrons, dans ce chapitre, les principales collections pour regroupés les patterns, ainsi des formalismes pour décrire d'une manière uniforme ces dernier. Toutefois, le type des patterns d'ingénierie qui sera étudié et qui constitue l'objectif de notre projet est celui des design patterns. Ce type est très présent lorsque nous parlons des patterns. Nous allons présenter en détail ce type de pattern, nous discutons ainsi des différentes implémentations aspect des patterns. Nous terminons ce chapitre par la présentation de la rétro-ingénerie des design patterns.

- Le troisième chapitre sera consacré entièrement à la présentation de notre modèle de transformation pour répondre au problème posé de l'automatisation de passage des patterns de l'objet vers l'aspect. Dans ce chapitre nous présentons l'approche globale pour la transformation, permettant d'assurer le passage de l'OO vers l'OA. Cette approche constitue en un ensemble d'algorithmes destinés à traduire des patterns. Nous proposons des situations d'application de ce modèle. Ces situations représentent des différents cas de transformation. Cette transformation peut apparaître, comme une génération de code. Elle est vue aussi comme une opération de rétro-ingénierie.

- Le quatrième chapitre est destiné pour la présentation de deux outils que nous avons réalisés. Le premier intitulé ‘’générateur d’une seule instance à la fois’’ spécialisée dans la générateur de code aspect des patterns pour une seul instance comme son nom l’indique. Le second outil consiste à générer plusieurs instances à la fois. L’opération de la rétro-ingénierie sera validée uniquement par une étude de cas.

La réalisation de ces projets nous donne l’occasion d’illustrer et de clarifier notre modèle de transformation.

Enfin nous concluons sur nos travaux avant de dégager des perspectives d’évolutions inspirées d’autres technologies.

A la fin de ce mémoire on trouvera des annexes, détaillant des sujets non étoffés dans les chapitres principaux. Ainsi l’utilisateur intéressé peut les consulter pour de plus amples informations.

Chapitre 01

Chapitre

I

Caractéristiques de la programmation orientée aspect

1. Introduction :

La programmation orientée objet [Meyer 2000] et [Brad et al.,1986], a vu son départ dans les années 1990, et s'est imposée en tant que standard dans le domaine du développement logiciel, d'abord au niveau des langages de programmation (C++, Java). Puis peu à peu au niveau des techniques de conception et de modélisation (UML). Cette approche à permis de faciliter les techniques de développement sous plusieurs aspects, en établissant un rapport plus net entre les composants logiciels et les systèmes qu'ils doivent gérer, en permettant de s'affranchir de certaines contraintes liées au matériel utilisé, et en offrant des perspectives de réutilisabilité des programmes.

La programmation orientée objet permet d'obtenir des programmes plus modulaires au sens de regrouper les traitements concernant une même donnée, au sein d'une même entité logicielle « la classe ». Cette programmation permet aussi une meilleure réutilisation, car il est facile de réutiliser des classes dans des contextes applicatifs différents. La sûreté est un autre avantage qui permet d'encapsuler des données au sein d'un objet. Enfin, l'extensibilité qui est définie par le concept d'héritage permet de créer de nouvelles classes à partir des classes existantes.

Munie de ces avantages, la programmation orientée objet reste dans le coeur des applications des éléments de code qui sont impossibles à centraliser ou, à factoriser. De plus l'héritage ne permet pas d'éviter complètement la duplication de code.

Pour cela la programmation orientée aspect [Kiczales et al., 97], [Pawlak et al., 04] viennent pour régler les limites de la programmation orienté objet.

Dans ce chapitre nous allons faire un aperçu sur les avantages, ainsi les limites de la programmation orientée objet. Puis nous introduisons les concepts de la programmation orientée aspect. Afin de montrer les apports de la programmation orientée aspect, nous présentons aussi le langage AspectJ (le langage de programmation qui supporte l'aspect) ainsi que ses concepts. Nous terminons ce chapitre par la présentation du paradigme aspect au niveau conceptuel.

2. De la programmation orientée objet vers la programmation orientée aspect :

La programmation orientée objet facilite le développement d'applications complexes, par le découpage du programme en entités indépendantes, appelé « classes », dont les données et les traitements sont propres à chacune d'elles.

2.1 Avantages de la programmation orientée objet :

La programmation orientée objet permet d'obtenir des programmes plus modulaires, réutilisables, sûrs et extensibles [Pawlak et al., 04] :

- **Modularité :** Elle permet de regrouper les traitements concernant une même donnée, au sein d'une même entité logicielle ''la classe''.
- **Réutilisation :** Elle permet de réutiliser des classes dans des contextes applicatifs différents.
- **Sûreté :** elle supporte l'encapsulation des données au sein d'un objet.
- **Extensibilité :** Cette extensibilité est définie par le concept d'héritage, permettant de créer de nouvelles classes à partir des classes existantes.

2.2 Limites de la programmation orientée objet :

Munie de ces avantages, la programmation orientée objet ne fournit pas de solution pour aboutir à des programmes clairs et élégants. Ces cas concernent les fonctionnalités transversales et la dispersion du code [Pawlak et al., 04].

2.2.1 Fonctionnalités transversales :

La programmation orientée objet permet de découper une application en des classes cohérentes et indépendantes. Cependant ces classes, représentant des « ensembles d'objets » peuvent parfois être reliées.

[Pawlak et al., 04], montre cette relation, par un cas de contrainte d'intégrité référentielle. Par exemple, un objet client ne peut être supprimé s'il n'a pas honoré toutes ses commandes. La classe client ne connaît pas les contraintes imposées par les autres classes et la classe commande n'a aucune raison de permettre la suppression d'un client. Finalement, ni la classe commande, ni la classe client ne sont adaptées à l'implémentation de la contrainte d'intégrité référentielle. Cette contrainte n'est pas strictement liée à la classe client ni à la classe commande mais, elle est transversale à ces deux types d'entités.

Alors que le découpage des données en classes a pour but de rendre les classes indépendantes entre elles. Les fonctionnalités transversales telles que les contraintes d'intégrités référentielles viennent se superposer à ce découpage et brisent l'indépendance des classes. La programmation orientée objet ne fournit aucune solution, pour ce genre de problème.

2.2.2 Dispersion du code :

Dans la programmation orientée objet, il y a deux types de services, ceux qui sont offerts par une classe et ceux qui sont utilisés par cette dernière (comme par exemple les appels de méthodes).

Il est plus facile de rassembler des services offerts dans une seule classe, mais il est impossible de rassembler l'utilisation d'un service. De plus l'implémentation d'une méthode peut être localisée (dans une classe), alors que les appels vers cette méthode sont dispersés dans différentes classes (voir Figure 1.1).

La modification d'une méthode peut être effectuée très facilement, contrairement à sa signature. Une modification de la signature d'une méthode implique la modification de toutes les classes invoquant cette méthode, ce qui rend la maintenance et l'évolution du code difficiles.

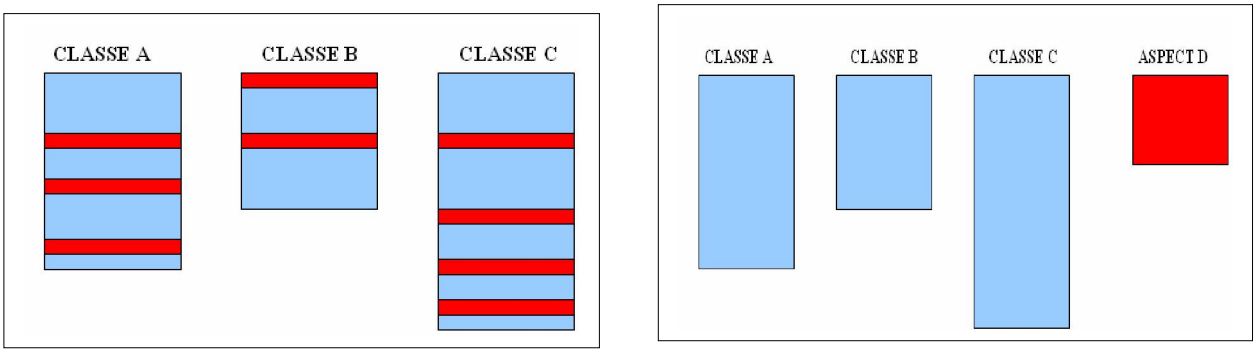


Figure 1.1. Dispersion du code d'une fonctionnalité (à gauche) et localisation du code d'une fonctionnalité transversale dans un aspect (à droite).

3. Concepts de la programmation orientée aspect :

La programmation orientée aspect (POA) est un nouveau paradigme de programmation qui trouve ses racines en 1996, suite aux travaux de Gregor Kiczales et de son équipe au Centre de Recherche Xerox à Palo Alto [Kiczales et al., 97]. La POA est donc une technologie relativement jeune, puisque les premiers outils destinés à son utilisation ne sont apparus qu'à la fin des années 90.

Dans cette section, nous allons introduire les concepts de ce nouveau paradigme, nous allons expliquer exactement les constituants de l'aspect, à savoir les points de jonction, les coupes, le 'codes advice', et d'autres.

3.1 Tissage d'aspects :

Les aspects définissent des morceaux de code et les endroits de l'application où ils vont s'appliquer. Un traitement automatique est donc nécessaire pour intégrer ces aspects dans l'application afin d'obtenir un programme fonctionnel fusionnant classes et aspects.
Cette opération, représentée, se nomme le tissage (weaving) et est réalisée par le tisseur d'aspects. (voir figure 1.2).

Le tissage peut se réaliser soit à la compilation, soit à l'exécution. Lorsqu'il est effectué à la compilation, il prend en entrée les classes et les aspects pour qu'il produit en sortie une application tissée. La sortie peut être sous la forme d'exécutable, de bytecode ou encore de code source. Le langage AspectJ utilise le tissage à la compilation. Lorsque le tissage est effectué à l'exécution, le tisseur d'aspect se présente sous la forme d'un programme qui permet d'exécuter à la fois l'application et les aspects. Les aspects ne sont donc pas intégrés dans l'application à l'avance. Cette technique permet d'ajouter, de modifier ou d'enlever des aspects au cours d'exécution. Le framework JAC utilise le tissage à la compilation [Pawlak et al., 04].

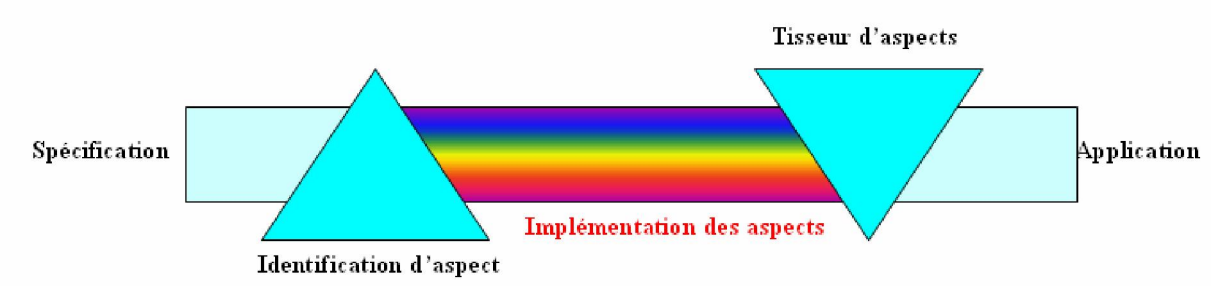


Figure 1.2 Tissage des aspects

3.2 Point de jonction :

Les aspects définissent les endroits du code où ils vont intégrer les fonctionnalités transversales. Cette spécification se réalise à l'aide de coupes qui sont elles-mêmes spécifiées à l'aide de points de jonction.

Un point de jonction est un point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés. [Pawlak et al., 04].

Une coupe définissant tous les points de l'application auxquels elle souhaite ajouter l'aspect. Ces points de l'application peuvent être l'appel d'une méthode, la lecture d'un attribut,...etc

Un point de jonction représente un événement dans l'exécution du programme qui peut être intercepté pour exécuter un code advice correspondant.

Pour cela, plusieurs types d'événements peuvent faire figures de points de jonction :

- **Méthodes :** Les méthodes sont les éléments principaux qui structurent l'exécution des programmes orientés objets. Les événements liés aux méthodes qui constituent des points de jonction sont l'appel d'une méthode et l'exécution de celle-ci.

- **Constructeur** : Les constructeurs peuvent être considérés comme des méthodes particulières. Il s'agit de méthodes invoquées lorsqu'un objet est instancié. Comme pour les méthodes, la programmation orientée aspect permet d'intercepter cet événement.

- **Attributs :** La lecture et la modification d'attributs constituent également des points de jonction.

- **Exception** : Les événements de levée et de récupération d'exceptions peuvent aussi constituer des points de jonction. Le code à exécuter lors de la levée de cette exception sera défini dans un aspect.

Les points de jonction associés aux méthodes, aux constructeurs, aux attributs et aux exceptions sont les plus courants et les plus utilisés en programmation orientée aspect. Il y en a d'autres, comme par exemple blocs de code, où ce dernier par exemple prend toutefois en compte les blocs de code `static` associés aux classes. Cela permet de développer des aspects qui effectuent des traitements après le chargement d'une classe et avant toute instanciation de cette classe (ce type de point de jonction peut être apparaître dans le langage AspectJ) [Pawlak et al., 04].

La POA peut connaître des limites quant à la granularité correspondante aux points de jonction. En effet, les instructions (if, while, for, switch, ...) sont jugées trop fines par certains outils de POA et ne sont donc pas saisies.

3.3 Coupes :

Une coupe désigne un ensemble de points de jonction. [Pawlak et al., 2004].

Les points de jonction et les coupes sont liés par leur définition, mais leur nature est très différente. Un point de jonction représente un point dans l'exécution d'un programme, une coupe est un morceau de code défini dans un aspect. Une coupe est définie à l'intérieur d'un aspect. Dans le cas simples, une seule coupe suffit pour définir la structure transversale d'un aspect. Dans les cas plus complexes, un aspect est associé à plusieurs coupes.

En général, la définition d'une coupe passe par l'utilisation d'une syntaxe et de mots-clés. Chaque outil de la POA fournit sa propre syntaxe et ses propres mots-clés. Ces mots-clés identifiant des ensembles de points de jonction. Ces ensembles sont unis dans la coupe par des opérations ensemblistes comme par exemple : intersection, union et complémentarité.

Les mots-clés utilisés par la coupe désignent chacun un ensemble de points de jonction en spécifiant son type (appel de méthode, lecture d'attributs, ...), et une expression qui précise le type (la méthode A, l'attribut B, ...). Ces expressions peuvent faire usage de quantificateurs, par exemple toutes les méthodes dont le nom est *add*.

Nous reviendrons en détail sur ces mots clés lors de la présentation du langage AspectJ.

3.4 Codes advice :

Un 'code advice' est un bloc de code définissant le comportement d'un aspect. [Pawlak et al., 04].

Un code advice définit donc un bloc de code qui va venir s' insérer sur les points de jonction définis par la coupe à laquelle il est lié. Un aspect nécessite souvent plusieurs codes advices pour caractériser la fonctionnalité transversale.

Il y a différentes manières d'ajouter un code advice sur un point de jonction.
Les trois principaux types d'ajout de codes advices sont :

- Avant les points de jonction ;
- Ou, après les points de jonction ;
- Ou bien autour des points de jonction.

Les codes advices peuvent donc être classés en trois types : code advice *before*, code advice *after* et code advice *around*.

Pour une insertion d'un code advice autour des points de jonction de la coupe, il est nécessaire de spécifier la partie du code qui s'effectue avant le point de jonction et la partie qui s'effectue après. Les outils de la POA fournissent à cet effet un mot-clé qui permet d'exécuter le point de jonction : *proceed*.

Un code advice around implique l'exécution suivante du programme :

1. Exécution normale du programme ;
2. Juste avant un point de jonction appartenant à la coupe, exécution de la partie avant du code advice.
3. Appel à *proceed*.
4. Exécution du code correspondant au point de jonction.
5. Exécution de la partie après du code advice.
6. Reprise de l'exécution du programme juste après le point de jonction.

L'appel à *proceed* est facultatif. . Dans le cas où il n'y a pas d'appel à *proceed*, le code correspondant au point de jonction n'est pas exécuté, et le bloc de code advice remplace le point de jonction.

Les types de codes advices, ne sont pas limités qu'à *before*, *after* et *around*. Le langage AspectJ propose, par exemple, les types *after returning* (pour designer le retour normal du point de jonction) et *after throwing* (pour designer le retour anormal avec levée d'exception du point de jonction).

3.5 Mécanisme d'introduction :

Le mécanisme d'introduction de la POA permet d'étendre des classes par l'ajout des éléments. Ces éléments peuvent être des attributs ou des méthodes, mais ce ne sont pas les seuls exemples [Pawlak et al., 04]. Nous allons voir par la suite en détail ces éléments dans le langage AspectJ.

A la différence de l'héritage en POA, l'introduction permet d'étendre les classes tout en rajoutant de nouveaux éléments. Contrairement à l'héritage, l'introduction ne permet pas de redéfinir une méthode. L'introduction est donc un mécanisme purement additif, qui ajoute de nouveaux éléments à une classe. Le mécanisme d'introduction est toujours sans condition, l'extension est réalisée dans tous les cas.

4. Apports de la programmation orientée aspect :

La POA complémente la POO en apportant des solutions aux deux limites que sont l'implémentation de fonctionnalités transversales et le phénomène de dispersion du code.

Une application orientée aspect est composée de deux parties, les classes et les aspects [Pawlak et al., 04] :

- Les classes constituent le support de l'application. Il s'agit des données et des traitements qui sont au cœur de la problématique de l'application et qui répondent aux besoins premiers de celle-ci.

- Les aspects intègrent aux classes des éléments (classes, méthodes, données) supplémentaires, qui correspondent à des fonctionnalités transversales ou à des fonctionnalités dont l'utilisation est dispersée.

Concevoir une application orientée aspect consiste à [Pawlak et al., 04] :

- Dans un premier temps, à opérer un processus de décomposition. Celui-ci permet d'identifier les classes et les aspects en fonction de l'analyse des besoins réalisée.

- Dans un second temps, la conception d'une application orientée aspect s'accompagne d'un processus de recomposition. Il s'agit de mettre en commun les classes et les aspects pour obtenir une application opérationnelle. Ce processus est réalisé à l'aide des notions de la programmation orientée aspect de coupe et de point de jonction. Ces notions permettent de designer les emplacements du programme autour desquels vont être insérés les aspects, et donc les fonctionnalités qu'ils implémentent, cette opération s'appelle le tissage.

5. Présentation du langage AspectJ :

Nous allons exposer dans ce point, le langage AspectJ [AspectJ]. Nous reprendrons les concepts présentés précédemment, mais plus spécifiquement pour le langage AspectJ.

Ce langage, est conçu et développé par Gregor Kiczales, l'inventeur du concept de programmation orientée aspect, et son équipe de centre de recherche PARC (Palo Alto Research Center) de Xerox [Kiczales et al., 01], AspectJ est l'outil référence pour la programmation orientée aspect [Pawlak et al., 04]. AspectJ offre un tisseur d'aspects à la compilation qui permet de composer le code des classes de base (écrit en java) avec celui relatif aux aspects, afin de générer le code de l'application finale.

Le langage AspectJ est une extension de langage Java, en plus des concepts de l'Objet (Classe, Attribut, Méthode, …), il introduit de nouveaux concepts et mécanismes Aspect (Aspect, Introduction, Advice, Pointcut…[AspectJ]).

La programmation orientée aspect peut être intégrée à moindre coût par les entreprises par l'extension de leurs outils. Par exemple, la programmation orientée aspect pour Java est intégrée dans Eclipse via un plugin qui permet l'utilisation du langage AspectJ (voir annexe A).

Nous allons détailler dans ce qui suit sur les concepts relatifs à ce langage.

5.1 Aspects :

La définition d'un aspect avec AspectJ est très proche à la définition d'une classe ou d'une interface en Java. Comme les classes et les interfaces, les aspects portent un nom et peuvent être définis au sein de packages.

AspectJ introduit le mot-clé *aspect* qui permet de spécifier que l'entité logicielle définie est un aspect. La figure 1.3 montre un exemple de définition d'un aspect :

```
package aop.aspectj;

public aspect MonAspect {
// code de l'aspect
}
```

Figure 1.3. Définition d'un aspect

La partie '// code de l'aspect' comprend aussi la définition des coupes, des codes advices, la déclaration de variables et/ou méthodes propres à l'aspect…etc. Ces concepts seront présentés au cours de cette section.

Un aspect est défini dans un fichier qui, porte le même nom que le nom de l'aspect. Ce fichier porter l'extension **.java**, mais il peut porter aussi l'extension **.aj** ou **.ajava** . En général, **.aj** ou **.ajava** sont utilisés pour designer les fichiers aspects et **.java** pour designer les fichiers classes. Pour bien séparer les fonctionnalités métiers et transversales.

5.2 Points de jonction :

Un ensemble de points de jonction est identifié dans une coupe à l'aide d'un mot-clé. Une expression est également fournie pour filtrer l'ensemble obtenu. Dans cette sous-section, nous verrons les mots-clés disponibles dans AspectJ pour chacun des types de point de jonction.

5.2.1 Types de point de jonction :

Les types de points de jonction fournis par AspectJ peuvent concerner des méthodes, des attributs, des exceptions, des constructeurs ou des blocs de code **static.** Un dernier type concerne les éxecutions de code advice. [Kiczales et al., 01].
Les points de jonction, les plus utilisés, dans la POA sont : méthodes, attributs, constructeurs et enfin exceptions.

a) Méthodes :

AspectJ définit deux types de points de jonction sur les méthodes : les appels de méthodes (‘*call’*) et les exécutions de méthode (‘*execution’*).

- Appel de méthode : *call(methexpr)* : Ce mot-clé identifie tous les appels vers des méthodes dont le profil correspond à la description donnée par ‘methexpr’.

- Exécution de méthode : *execution(methexpr)* : Ce mot-clé identifie toutes les exécutions de méthodes dont le profil correspond à l'expression ‘methexpr’.

La différence principale entre les types *call* et *execution* concerne le contexte dans lequel le se trouve l'application lors de l'exécution du code advice associé. Un point de jonction donnée par *call(methexpr)* correspond à l'appel de la méthode à methexpr. Il se trouve donc dans le code de la méthode appelante. Alors qu'un point de jonction donné par *execution(methexpr)* se trouve dans la méthode appelée. L'ordre d'exécution des différents codes advices serait le suivant :

1. Dans la méthode appelante : Exécution de la première partie du code advice associé au point de jonction *call*.

2. Dans la méthode appelée : Exécution de la première partie du code advice associé au point de jonction *execution*.

3. Dans la méthode appelée : Exécution de la méthode appelée.

4. Dans la méthode appelée : Exécution de la deuxième partie du code advice associé au point de jonction *execution*.

5. Dans la méthode appelante : Exécution de la deuxième partie du code advice associé au point de jonction *call*.

b) Attributs :

Les points de jonctions de type *get* et *set* permettent d'intercepter respectivement les lectures et les écritures des attributs.

- Lecture d'attribut : *get(attrexpr)* : Identifie tous les points de jonction représentant la lecture d'un attribut dont le profil vérifie attrexpr.

- Modification d'attribut : *set(attrexpr)* : Ce mot-clé identifie toutes les modifications d'un attribut dont le profil est compatible à attrexpr.

Ces mots-clés sont intéressants pour des aspects qui souhaitent intercepter la modification de l'état d'un objet.

c) Constructeurs :

AspectJ fournit deux types de points de jonctions, pour les constructeurs : *initialization* et *preinitialization.*

1. Exécution de constructeur : *initialization(constrexpr)* : Identifie toutes les exécutions d'un constructeur dont le profil vérifie constrexpr.

2. Exécution de constructeur hérité : *preinitialization(constrexpr)* : Ce mot-clé identifie toutes les exécutions d'un constructeur hérité dont le profil correspond à l'expression constrexpr.

L'exécution d'un constructeur hérité se fait en Java avec l'aide du mot-clé **super(..)** où **..** représente les paramètres fournis. Java impose que ce genre d'appel soit la première instruction d'un constructeur. Alors que *initialization* identifie l'exécution de constructeurs, *preinitialization* identifie l'exécution de constructeurs appelés via cette instruction. Il est intéressant de remarquer qu'il y a une possibilité d'intercepter les appels de constructeur avec le mot-clé *call.*
Le profil methexpr identifie les méthodes de nom *new.* AspectJ ne permet pas la définition d'un code advice de type *around* sur ces points de jonction.

d) Exceptions :

Récupération d'exception : *handler(exceptexpr)* : Ce mot-clé identifie toutes les récupérations d'exception dont le profil vérifie exceptexpr. Il s'agit donc, en Java, de l'exécution des blocs *catch*. AspectJ ne permet pour le moment que l'utilisation de codes advices de type before sur les points de jonction de récupération d'exception.

AspectJ introduit d'autres de point de jonction, comme par exemple :

- Initialisation de classe : *staticinitialization(classexpr)* : Ce mot-clé identifie l'exécution des blocs statiques d'initialisation d'une classe correspondant à l'expression classexpr.

- Exécution de code advice : *adviceexecution()* : Identifie toutes les exécutions d'un code advice. Il ne requiert pas d'expression en paramètre. Il identifie simplement toutes les exécutions de code advice.

Le tableau 1.1 récapitule les différents points de jonction, concernant les méthodes, attributs et constructeurs.

Type de point de jonction	Description
call (methexpr)	Appel d'une méthode dont le nom vérifie methexpr.
execution(methexper)	Exécution d'une méthode dont le nom vérifie methexper.
get(attrexpr)	Lecture d'un attribut dont le nom vérifie attrexpr.
set(attrexpr)	Ecriture d'un attribut dont le nom vérifie

	attrexpr.
initialization(constrexpr)	Exécution d'un constructeur dont le nom vérifie constrexpr.
preinitialization(constrexpr)	Exécution d'un constructeur hérité dans un constructeur dont le nom vérifie constrexpr.

Tableau 1.1 Différents types de points de jonction définis dans AspectJ [Kiczales et al., 01].

5.2.2 Définition des profils :

Tous les mots-clés, présentés dans la sous-section des types de point de jonction nécessitent un paramètre, qui est une expression permettant, de spécifier un profil, et de filtrer l'ensemble de points de jonction donné par le mot-clé. L'expression précisant le profil des éléments peut faire usage de quantificateurs (appelés wildcards). Ces wildcards sont ***, .. et +**.[Kiczales et al., 01].

a) Wildcard *

Le symbole * peut être utilisé pour remplacer des noms de classe, de méthode et d'attribut. Il peut soit remplacer le nom complet, soit une partie de celui-ci.

Exemples :

public void aop.aspectj.MaClass.*(String)

Représente toutes les méthodes publiques de la classe MaClass (dans le package aop.aspectj) prenant un String en paramètre et ne retournant rien.

public *aop.aspectj.MaClass.*add*(String)

Représente toutes les méthodes publiques de la classe MaClass (dans le package aop.aspectj) prenant un String en paramètre et dont le nom contient la chaîne add.

private aop.*.MaClass.*

Représente tous les attributs privés des classes MaClass qui appartiennent à un sous-package de premier niveau du package aop.

*** * aop.aspectj.MaClass.*(String) ou * aop.aspectj.MaClass.*(String)**

Représente toutes les méthodes de la classe MaClass (dans le packageaop.aspect) prenant un String en paramètre.

b) Wildcard ..

Le symbole .. permet de prendre en compte le polymorphisme des méthodes, ainsi la généricité dans la hiérarchie des packages.

Exemples :

public void aop.aspectj.MaClass.*(..)

Représente toutes les méthodes publiques de la classe MaClass (dans le package aop.aspectj) qui ne retournent rien.

public aop.aspectj.MaClass.new(..)

Représente tous les constructeurs de la classe MaClass (dans le package aop.aspectj).

aop..*.*(..)

Représente toutes les méthodes de toutes les classes de n'importe quel package de la hiérarchie aop.

c) Wildcard +

Le symbole + est utilisé en tant qu'opérateur de sous-typage. Il permet d'identifier l'ensemble contenant cette classe et toutes ses sous-classes.

Exemples :

public void aop.aspectj.MaClass+.*(..)

Représente toutes les méthodes publiques de la classe MaClass (dans le package aop.aspectj) et de toutes ses sous-classes, ne retournant rien.

*** java.awt.Shape+.*(..)**

Représente toutes les méthodes des classes implémentant l'interface java.awt.Shape.

5.3 Notion de coupes :

Une coupe est un ensemble de points de jonction résultant d'un calcul ensembliste sur ces ensembles. Les opérations ensemblistes de base disponibles sont l'union, l'intersection et la complémentarité. Elles sont respectivement représentées dans AspectJ par les symboles ||, && et !. [Kiczales et al., 01].

Exemples :

call(* aop.aspectj.MaClass.*(..)) && !call(* aop.aspectj.MaClass.*get*(..))

Représente l'ensemble des appels de méthode, dont le nom ne contient pas get, appartenant à la classe MaClass (dans le package aop.aspectj).

aop..MaClass.*(..)

Représente toutes les méthodes de toutes les classes MaClass dans n'importe quel package de la hiérarchie aop.

Une coupe est définie par le mot-clé *pointcut*, comme par exemple :
pointcut MaCoupe(): call(* aop.aspectj.MaClass.*(..)) ;

Les parenthèses sont obligatoires, car il est possible de paramétrer une coupe.

5.3.1 Filtrage :

Les points de jonction permettent de construire de nombreuses expressions de coupe.

call(*aop.aspectj.MaClass1.*(..)) || call(* aop.aspectj.MaClass2.*(..))

Représente l'ensemble des appels de méthodes appartenant à l'ensemble des méthodes des classes MaClass1 et MaClass2 (dans le package aop.aspectj).

Ces types peuvent être combinés à l'aide d'opérateurs logique et que des filtrages permettent de sélectionner seulement certains points de jonction parmi tous ceux possibles.
Le tableau suivant détail sur d'autres opérateurs de filtrage (en dehors des opérateurs booléens ||, && et !) :

withincode(methexpr)	Vrai lorsque le point de jonction est défini dans une méthode dont le profil vérifie methexpr.
within(typeexpr)	Vrai lorsque le point de jonction est défini dans un type (classe ou interface) dont le nom vérifie typeexpr.

this(typeexpr)	Vrai lorsque le type de l'objet source du point de jonction vérifie typeexpr.
target(typeexpr)	Vrai lorsque le type de l'objet destination du point de jonction vérifie typeexpr.
cflow(coupe)	Vrai pour tout point de jonction situé entre l'entrée dans la coupe et sa sortie (y compris l'entrée et la sortie)
cflowbelow(coupe)	Vrai pour tout point de jonction situé entre l'entrée dans la coupe et sa sortie (sauf pour l'entrée et la sortie)

Tableau 1.2. Opérateurs de filtrage

Les opérateurs *withincode, within, this* et *target* sont des opérateurs statiques, ils ne dépendent pas de la façon dont s'exécute le programme. Par contre *cflow* et *cflowbelow* prennent en compte la dynamique d'exécution d'un programme. L'opérateur *cflow* prend une coupe en paramètre. Il identifie tous les points de jonction situés entre le moment où l'application passe par un des points de jonction de la coupe et le moment où l'application sort de ce point de jonction. L'ensemble de points de jonction fourni par *cflow* contient les points de jonction de la coupe. Par exemple, soit la coupe *MaCoupe*, fournie en paramètre à *cflow*, cette coupe identifie tous les appels à une méthode *add*. *cflow(MaCoupe)* identifie les appels à *add*, tous les points de jonction rencontrés pendant l'exécution de *add*. La méthode *add* peut faire appel à d'autres méthodes, leur contenu sera aussi considéré. La seule différence entre *cflow* et *cflowbelow* est le fait que *cflowbelow* ne considère pas les points de jonction fournis dans la coupe.

Une autre technique utilisée par le filtrage qui est basée sur des expressions booléennes. Cette technique permet d'utiliser l'opérateur *if* dans la définition d'une coupe. Par exemple, if(thisJoinPoint.getArgs().length==1) identifie tous les points de jonction dont le nombre d'arguments égale à un.

5.3.2 Paramétrage des coupes :

En général, il est possible de paramétrer une coupe, Les paramètres sont, dans ce cas, des informations qui sont transmises de la coupe vers les codes advices qui l'utilisent.

Il y a trois types d'informations qui peuvent être transmettre: l'objet source, l'objet cible et les arguments des points de jonction de la coupe. L'objet source et l'objet cible sont transmis à l'aide de mots-clés : *this* et *target*. Les arguments des points de jonction désignés par la coupe sont, quant à eux, transmis à l'aide du mot-clé *args*.

Par exemple :

pointcut example(Rectangle src, Point cib, int x, int y): call(* *..*(..)) && this(src) && target(cib) && args(x,y);
Représente tous les appels de méthode où l'objet appelant est un Rectangle, où l'objet cible est un Point et où les paramètres de la méthode sont deux entiers. Les variables src, cib, x et y peuvent être utilisées dans les codes advices liés à cette coupe.

5.4 Codes Advices :

AspectJ introduit deux nouveaux types : *after returning* et *after throwing*. [Kiczales et al., 01]. Dans cette sous-section, nous allons détailler les différents types de code advice proposés par AspectJ.

5.4.1 Le type before :

La figure 1.4 donne un exemple de code advice de type before.

```
//définition de la coupe
pointcut MaCoupe(): call(* *..*(..));
//définition du code advice de type before
before(): MaCoupe() {
System.out.println("Avant un appel de méthode");
}
```

Figure 1.4. Exemple de code advice de type *before*

Ce code advice affiche à l'écran le message suivant "Avant un appel de méthode" avant chaque appel de méthode de l'application.

5.4.2 Le type after :

Le code advice de type *before* s'exécute avant les points de jonction de sa coupe, un code advice de type *after* s'exécute après les points de jonction. C'est la seule différence entre ces deux types.

5.4.3 Le type around :

Les codes advices de type *around* exécutent du code avant et après les points de jonction. Les deux parties sont séparées dans le code advice par le mot-clé *proceed*. Ce mot-clé a pour but d'exécuter le point de jonction courant. Le type *around* possède un type de retour qui correspond au type de retour des points de jonction. La figure 1.5 illustre un exemple.

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

Figure 1.5. Exemple de code advice de type *around*

Dans le cas ou il n y a pas de retour le type de retour du code advice est ‘void’.
La figure 1.6, montre un exemple de ce cas :

```
void around():call(* *..*(..))  {
System.out.println("Avant un appel de méthode");
proceed();
System.out.println("Après un appel de méthode");
}
```

Figure 1.6. Exemple de code advice de type *around* (type de retour du code advice est void)

Ce code advice écrit le string ”Avant un appel de méthode” avant chaque appel d’une méthode de l’application et écrit le string ”Après un appel de méthode” après chaque appel. Il se peut que *proceed* ne soit pas appelé par le code advice, ceci impliquant que le point de jonction ne sera pas exécuté.

5.4.4 Le type after returning :

Les codes advices de type *after returning* s’exécutent à chaque terminaison normale d’un des points de jonction de la coupe. La figure 1.7 montre un exemple :

```
after() returning (double d): call(double
MaClass.add (..)) {
System.out.println("add a retourné : " + d);
}
```

Figure 1.7. Exemple de code advice de type *after returning*

5.4.5 Le type after throwing :

Les codes advices de type *after throwing* s’exécutent à chaque terminaison anormale d’un des points de jonction de la coupe. La figure 1.8 montre un exemple :

```
after() throwing (Exception e): call(double
MaClass.add (..)) {
System.out.println("add a levé l’exception : " + e);
}
```

Figure 1.8. Exemple de code advice de type *after returning*

5.5 Introspection de point de jonction :

Dans ce point nous allons présenter les moyens fournis par AspectJ pour obtenir des informations sur le point de jonction actuel qui a provoqué l'exécution du code advice. Pour cela AspectJ introduit un nouveau mot-clé : *thisJoinPoint* [Kiczales et al., 01]. Le type de *thisJoinPoint* est la classe *org.aspectj.lang.JoinPoint*. Le tableau suivant reprend les méthodes offertes par cette classe :

Méthode	Description
Object getThis()	Retourne l'objet source du point de jonction. Si le point de jonction concerne une méthode, l'objet source est l'objet appelant.
Object getTarget()	Retourne l'objet cible du point de jonction. Si le point de jonction concerne une méthode, l'objet cible est l'objet appelé.
Object[] getArgs()	Retourne les arguments du point de jonction. Si le point de jonction concerne une méthode, les arguments sont les paramètres fournis `a celle-ci.
String getKind()	Retourne le type du point de jonction sous forme de String.
Signature getSignature()	Retourne la signature du point de jonction. Par exemple, si le point de jonction concerne une méthode, l'interface Signature permet de récupérer le nom, la visibilité, la classe et le type de retour de cette méthode.
SourceLocation getSourceLocation()	Retourne la localisation du point de jonction dans le code de l'application. L'interface SourceLocation permet notamment de récupérer le nom de fichier et le numéro de ligne du point de jonction.

Tableau 1.3. Méthodes principales de la classe org.aspectj.lang.JointPoint

La figure 1.9 , présente un exemple d'utilisation de la classe *thisJoinPoint,* avec la méthode *getSignature()*

```
Package aop.aspectj;
public aspect MonAspect{
before(): call(* *..*.*(..)) {
System.out.println(thisJoinPoint.getSignature().getName());
}
}
```

Figure 1.9. Exemple d'utilisation de la classe *thisJoinPoint*

Cet exemple affiche dans l'écran le nom de chaque méthode appelée.

5.6 Mécanisme d'introduction :

Dans la programmation orientée aspect, il est possible d'étendre les classes d'une application en leur ajoutant divers éléments. AspectJ permet l'ajout de cinq types d'éléments : attribut, méthode, constructeur, classe héritée et interface implémentée. [Kiczales et al., 01].

Le mécanisme d'introduction permet à l'aspect de déclarer des éléments (attributs, méthodes et constructeurs) pour le compte des classes.

Nous allons décrire sur ce qui suit, expliquer les différents types d'éléments utilisés dans le mécanisme d'introduction.

5.6.1 Attribut :

Cette déclaration se fait comme dans une classe si ce n'est qu'il faut préfixer le nom de l'attribut par le nom de la classe visée. Sans ce préfixe, l'attribut serait un champ de l'aspect.

5.6.2 Méthode :

Le mécanisme d'introduction dans les méthodes est semblable à l'introduction d'attribut. Elle se réalise comme la définition d'une méthode au sein d'une classe, sauf avec l'ajout comme préfixe le nom de la classe cible.

5.6.3 Constructeur :

Pour le mécanisme d'introduction dans les constructeurs, AspectJ considère qu'un constructeur est juste une méthode dont le nom est *new*.

5.6.4 Classe héritée et Interface implémentée :

AspectJ permet également de modifier la hiérarchie d'héritage des classes. A l'aide du mot-clé *declare parents*. AspectJ offre la possibilité de rendre une classe héritière d'une autre. Mais cette introduction n'est pas sans condition, il se peut en effet que la classe visée hérite déjà d'une surclasse. Dans ce cas, l'introduction ne sera possible que si la nouvelle surclasse est une sous-classe de l'ancienne surclasse.

Le mécanisme d'introduction d'interface implémentée est similaire à l'introduction de classe héritée. Elle se réalise à l'aide du même mot-clé mais l'introduction d'interface implémentée est sans condition. La figure 1.10 montre un exemple de cinq types d'éléments du mécanisme d'introduction.

```
public aspect MonAspect {

declare parents: MaClass1 extends MaClass2;
declare parents: MaClass3 implements MonInterface;

private int MaClass.id;
public int MaClass.add () {id=id+1 ; return id; }
public MaClass.new(int id) {this.id=id;}
}
```

Figure 1.10. Exemple pour cinq types d'éléments du mécanisme d'introduction

5.7 Héritage dans la POA:

Un aspect peut étendre un aspect abstrait. Il est défini à l'aide du mot-clé *abstract* placé devant le mot-clé *aspect*. [Kiczales et al., 01].
Seules les coupes peuvent être étendues dans un sous-aspect. Ce dernier spécifie l'ensemble des points de jonction auxquels vont s'appliquer les codes advices liés à cette coupe dans l'aspect abstrait. La figure 1.11 montre un exemple d'un aspect abstrait et un sous-aspect.

```
public abstract aspect MonSuperAspect {
abstract pointcut Macoupe();
before(): Macoupe() { ... }
}

public aspect SousAspect extends MonSuperAspect {
pointcut Macoupe(): call(* MaClass.*(..));
}
```

Figure 1.11. Exemple d'un aspect abstrait et un sous-aspect.

6. Le paradigme aspect dans la phase de conception :

Le terme programmation orientée aspect (aspect) est en général lié à la phase d'implémentation, et puisque l'aspect s'intéresse à la séparation des préoccupations transversales, il sera intéressant d'identifier les aspects au niveau des phases préalables à la phase de programmation.

Plusieurs travaux ont été proposés pour définir l'aspect (comme entité) dans les phases d'analyse et de conception. Ils se sont intéressés à la séparation des préoccupations transversales (aspects) tout au long du cycle de développement de logiciel, et en particulier au niveau de la phase de conception.

6.1 Différentes représentations :

Nous allons effectuer une petite synthèse des propositions basée sur les propriétés suivantes [Hachani 2006] :

- L'extension : La plupart des approches existantes proposent, pour la définition de leurs concepts et relations Aspect, d'étendre le métamodele d'UML. Elles se distinguent cependant par leurs façons de faire : extension par stéréotypage, par metamodélisation ou proposition d'un profile UML.

- Les vues : Pour représenter un système à base d'aspects il convient le plus souvent de s'intéresser non seulement à la modélisation de sa structure statique, mais aussi a la modélisation de sa partie dynamique et donc à l'expression du résultat du tissage des aspects. Une proposition peut s'intéresser exclusivement à la vue structurelle ou comportementale d'un système, comme elle peut considérer les deux vues.

Le tableau 1.4 montre les propositions, selon les propriétés, cités ci-dessus :

Approches	Extension	Vues
[Suzuki et al., 99]	metamodélisation et stéréotypage	Structurelle
[Stein 02]	Stéréotypage	Structurelle Comportementale
[Zakaria et al., 02]	stéréotypage	Structurelle
[Basch et al., 03]	profile UML	Structurelle Comportementale
[Han et al., 04]	metamodélisation	Structurelle

Tableau 1.4. Synthèse des travaux paradigme aspect dans les phases préalables à la phase de programmation. [Hachani 2006].

6.2 Metamodélisation et Stéréotypage des aspects:

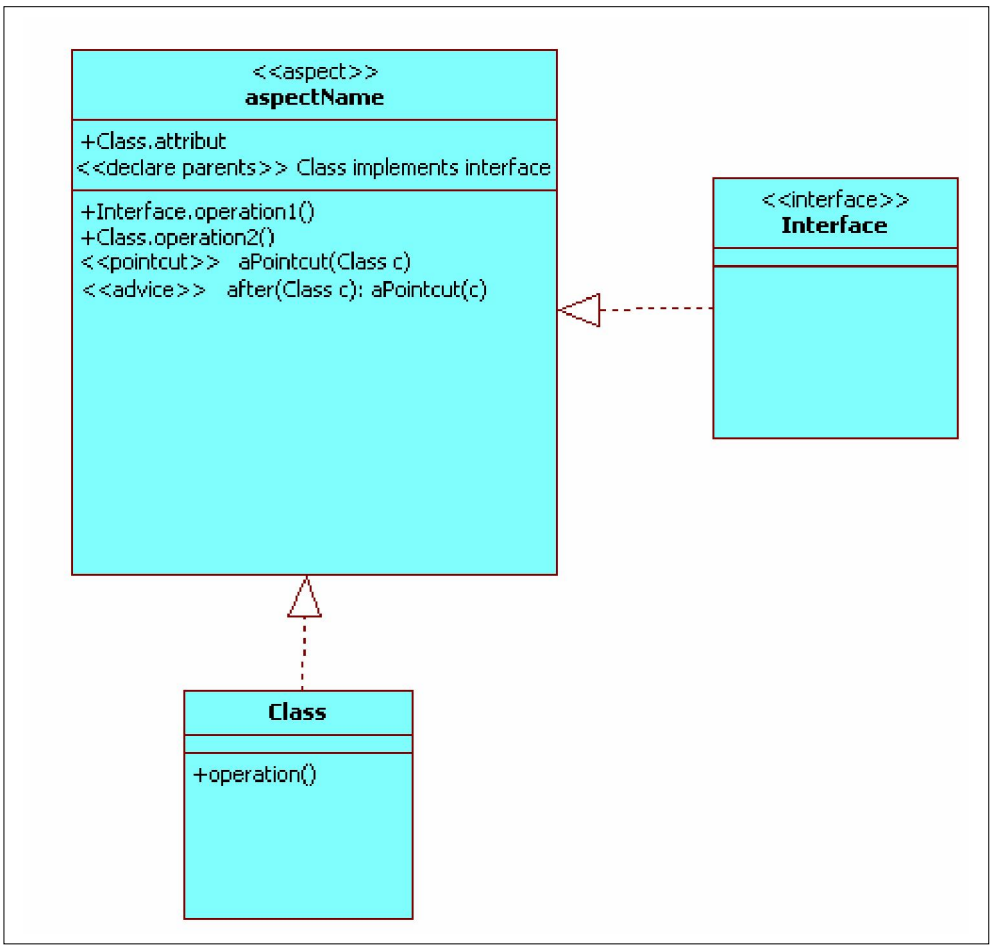
UML définit un ensemble d'éléments de modélisation, comme par exemple des concepts et des relations dont chacun est représenté par une métaclasse au niveau de son métamodele. Il offre aussi des mécanismes d'extension de ces éléments : les stéréotypes, les valeurs marquées et les contraintes, permettent l'extension et la spécialisation des classes de concepts et relations standards d'UML. Les stéréotypes permettent d'ajouter de nouvelles classes d'éléments de modélisation au métamodele d'UML, par dérivation des métaclasses existantes. Les valeurs marquées étendent les attributs des classes d'éléments du métamodele et les contraintes sont des relations sémantiques entre éléments de modélisation qui définissent des conditions que doit vérifier le système. Ces mécanismes existent dans plusieurs travaux, qui visent a étendre UML pour intégrer les aspects, comme [Suzuki et al., 99], [Stein 02].

Une autre possibilité d'extension d'UML consiste à étendre son métamodéle par l'ajout de nouveaux éléments originaux de modélisation et donc de nouveaux concepts et relations. Elle peut être complémentaire de l'utilisation des mécanismes d'extension. L'usage exclusif des

mécanismes d'extension est insuffisant parce qu'il repose trop fortement sur la définition de stéréotypes : introduire un concept ou une relation propre a l'approche aspect en se limitant a la définition d'un stéréotype consiste plus à se rapprocher d'un élément de modélisation déjà existant dans UML que de définir un nouvel élément et de rechercher sa place dans le noyau d'UML déjà constitué.
La considération d'un aspect comme un stéréotype de la métaclasse Class du métamodele d'UML comme le proposent [Suzuki et al., 99], [Stein 02], etc., oppose la définition originale d'un aspect dans [Kiczales et al., 97]. De plus, l'utilisation de stéréotypes peut très rapidement conduire à une divergence conséquente des extensions proposées.
La métamodélisation offre la possibilités de structuration dans la définition de nouveaux éléments de modélisation tout en permettant de les mettre en relation entre eux, mais également avec les concepts standards déjà existant dans le noyau d'UML. L'approche par métamodélisation présente un inconvénient principal par rapport à l'approche par stéréotypage, ou profile UML. Il est difficile, d'utiliser un outil de modélisation standard UML déjà existant, pour exprimer des modèles instances du nouveau métamodéle résultat de l'extension.

6.3 Représentation de Suzuki :

Pour le langage AspectJ, la solution de [Suzuki et al., 99], est la plus utilisée pour les raisons suivantes. Il vise la metamodélisation et le stéréotypage.
Il s'agit d'une extension d'UML pour les aspects. Dans cette représentation, une relation de réalisation connectant une classe (ou une interface) à un aspect signifie que cet aspect entrecroise (crosscut) cette classe (ou cette interface). Un aspect est représenté par un stéréotype particulier de classe : « aspect ». (voir figure 1.12).



7. Conclusion :

Nous avons présenté dans ce chapitre les principes concepts de la programmation orientée aspect, à savoir, les points de jonction, les coupes, les codes advice, et enfin le mécanisme d'introduction. Nous avons également présenté ces mêmes concepts pour le langage AspectJ.

La programmation orientée aspect est un domaine d'actualité dans les paradigmes de programmation. De plus il touche plusieurs domaines et plusieurs sujets. Parmi ces sujets, nous pouvons noter les design patterns. Ce sujet fera l'objet d'une étude plus approfondie dans le prochain chapitre.

Dans le cadre de notre travail, nous nous intéressons plus particulièrement à l'utilisation conjointe de l'approche aspect et les design patterns dans notre approche de transformation. Un de nos principaux objectifs, dans ce travail, est de garantir une meilleure réutilisation de l'implémentation aspect des patterns, de manière à pouvoir faciliter la conception à base d'aspect des systèmes et améliorer par conséquent leur évolution et leur réutilisation.

Chapitre 02

Chapitre

II

Présentation des Design Patterns

1. Introduction :

Les design patterns sont les plus connus et les plus utilisés parmi les patterns d'ingénierie. Ils identifient, nomment et rendent plus abstraits les connaissances convenues de plusieurs expériences. Ils répondent à des problèmes de conception communs, qu'il s'agisse de conception détaille, globale ou liée à des architectures particulières.

Les patterns constituent un moyen efficace pour la conception, la composition de plusieurs types de composants réutilisables, et pour le développement de grands systèmes complexes. Ils permettent d'améliorer la qualité et la compréhension des programmes, facilitent leur évolution et augmentent notamment leur réutilisation.

Dans ce chapitre, nous allons présenter les différents types des patterns d'ingénierie, avec quelques exemples pour chaque type, ainsi leurs avantages dans le développement des systèmes.

Nous illustrons, dans ce chapitre, les principales collections pour regrouper les patterns, ainsi des formalismes pour décrire d'une manière uniforme ces derniers.

Toutefois, le type de pattern d'ingénierie qui sera étudié et qui constitue l'objectif de notre projet est celui des design patterns. Ce type est très présent lorsque nous parlons des patterns.

Nous allons présenter en détail ce type de pattern, nous discutons ainsi des différentes implémentations aspect des patterns. Nous terminons ce chapitre par la présentation de la rétro-ingénerie des design patterns.

En général, les patterns peuvent être regroupés dans un groupe appelé collection, de plus ils sont décrits d'une manière uniforme par des formalismes.

2. Patterns d'ingénierie :

Cette section présente une introduction au concept de pattern d'ingénierie, par un petit historique, des différentes définitions de ce concept, ainsi les intérêts, et enfin les différents types des patterns d'ingénierie.

2.1 Historique :

La notion de pattern a été proposé par C. Alexander, dans le domaine de l'architecture des constructions (bâtiments) [Alexander et al., 77], [Alexander 79].

C. Alexander notait que les grands constructeurs, n'avaient pas suivi un modèle préconçu avec des règles précis, mais que les architectures avaient été adaptées les unes après les autres et à leurs environnements. Il a remarqué que certaines solutions de conception, considérées comme efficaces s'appliquaient dans différentes situations de construction. Cette idée est réalisée au travers de 253 patrons propres aux problèmes récurrents en architecture décrits d'une manière uniforme [Alexander et al., 77].

Des approches semblables à celle d'Alexander sont apparues dans d'autres domaines. Par exemple, les patterns décrites dans le domaine de l'hydraulique [Asplund et al., 73].

L'un des patterns de C. Alexander le plus fréquemment cité est « a place to wait » (un endroit pour attendre), qui propose des solutions conceptuelles au problème de l'attente. (Voir figure 2.1)

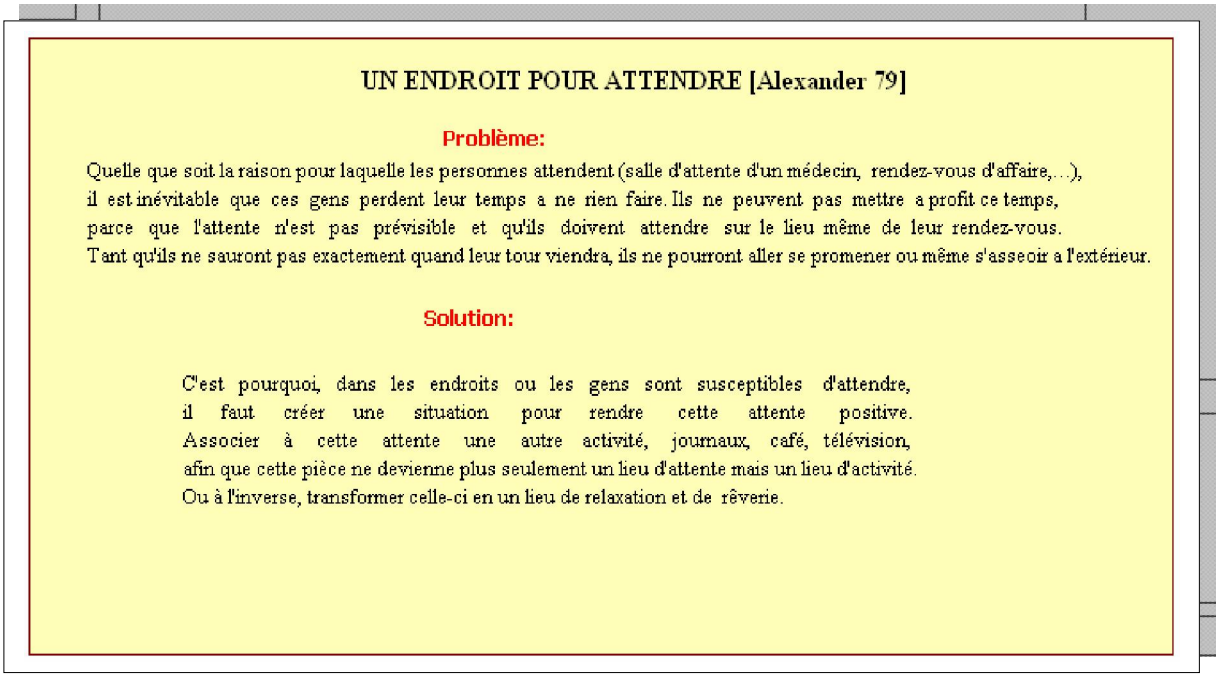


Figure 2.1. Présentation d'un pattern de C.Alexander

Dans le domaine de l'informatique, les premiers patterns ont été proposés par K. Beck et W. Cunninghan [Beck et al., 87]. Il s'agissait d'un première pratique du langage de patterns d'Alexander à la conception et à la programmation Objet. Un ensemble de patterns spécifique à la conception par objets a été créé, un peu plus tard, par E. Gamma [Gamma 91]. Ces travaux ont été suivis en 1995 d'un premier ouvrage collectif [Gamma et al., 95].

Dans le cadre de génie logiciel P. Coad [Coad 92] proposait de faciliter l'analyse d'un système par l'identification des besoins selon 7 patterns préétablis. Il définit un pattern objet comme une abstraction de structures de classes similaires. Une abstraction qui peut être réutilisée, pour le développement d'applications. Un pattern de P.Coad sera étudier en détail par la suite.

Plusieurs, articles et ouvrages ont été publies sur les patterns [Gamma et al., 95], [Coad 95], [Coplien 96], [Buschmann 96], [Coad et al., 97], [Booch 97], [Fowler 97], [Ambler 98], [Larman 02], etc.

2.2 Définition :

Il n y a pas, une définition exacte des patterns. Mais la définition la plus utilisée dans la plupart des ouvrages et articles [Gamma et al., 95], [Coad 95], [Coplien 96], [Buschmann 96], [Coad et al., 97], [Booch 97], [Fowler 97], [Ambler 98], [Larman 02], etc, est la définition de C.Alexander qui est :

« Un pattern rassemble un savoir-faire qui permet de résoudre un problème récurrent d'un domaine (comme l'architecture ou l'informatique).
Chaque pattern décrit à la fois un problème qui se produit très fréquemment dans votre environnement et l'architecture de la solution à ce problème, de telle façon que vous puissiez utiliser cette solution des millions de fois sans jamais l'adapter deux fois de la même manière »

De manière générale, un pattern constitue une bonne solution pour résoudre un problème récurrent dans un domaine particulier.

La spécification de ces connaissances réutilisables, consiste en un certain nombre de fonction :

- Permettre d'identifier le problème à résoudre par capitalisation et organisation de connaissances d'expériences ;
- Proposer une solution possible, correcte et générale pour y répondre ;
- Offrir les moyens d'adapter cette solution au contexte spécifique.

Dans le domaine de génie logiciel les patterns sont des composants réutilisables associés aux problèmes, contextes et solutions.

2.3 Avantages des patterns d'ingénierie :

Nous allons exposer dans ce point certains des avantages offerts par les patterns d'ingénierie.

Les patterns d'ingénierie apportent des solutions efficaces à divers problèmes d'analyse et de conception, permettant de créer des conceptions réutilisables. Ils proposent des bonnes solutions pour facilité l'écriture des programmes.

De plus, ils offrent un vocabulaire commun à l'équipe de développement [Fowler 1997], [Gamma et al., 1995], [Larman 2002]. Ils améliorent ainsi la compréhension des applications en cours de développement, pour faciliter l'évaluation et la maintenance de ces dernières.

En termes d'intérêt, les patterns d'ingénierie offrent un gain de temps et d'efforts, par la réduction du temps des essaies. Puisqu'ils offrent directement des bonnes solutions, facilitent ainsi la compréhension des systèmes, et permettent, en effet, d'aborder des systèmes complexes de manière plus simple.

2.4 Différents types des patterns d'ingénierie :

Il existe plusieurs classifications des patterns d'ingénierie, cette classification dépend des critères de classification choisis. La classification la plus utilisée, est celle de [Conte et al., 01b]. Les différents critères de classification des composants [Conte et al., 01a] sont utilisés pour classifier les patterns d'ingénierie, à savoir types de connaissances, couvertures et portées .

2.4.1 Types de connaissances :

Les composants peuvent capitaliser des connaissances de type *produit* ou *processus*. Les composants de type produit correspondent au but à atteindre. Ils sont destinés à être réutilisés et intègrés directement dans le système en cours de développement. Les patterns de [Gamma et al., 95] et les architectures logicielles sont des exemples de composants produit.

Les composants processus correspondent plutôt au chemin à parcourir pour atteindre le résultat. Ils servent à définir des démarches facilitant le développement. Les patterns de [Ambler 98] sont des exemples de composants processus.

2.4.2 Couvertures :

La couverture caractérise le degré de généralité des composants. Certains peuvent être généraux, d'autres plus spécifique à un domaine ou à une entreprise.

C'est le problème traité par le composant qui détermine son type. Si le problème est très fréquent dans de nombreux domaines d'application, ce type est dit *général*. Si le problème est spécifique à un domaine particulier, ou à une entreprise particulière celui-ci est dit respectivement de *domaine*, ou d'*entreprise*.

Les frameworks spécifiques à la comptabilité sont des exemples de composants de domaine.

Les patterns proposés par L. Gzara [Gzara 00] sont spécifiques à un domaine d'application (les systèmes d'information Produit).
Les patterns de GoF [Gamma et al., 95] et de GRASP [Larman 02] par exemple, sont des patterns généraux.

2.4.3 Portées :

La portée d'un composant détermine la phase du cycle de développement concernée par celui-ci. Nous distinguons par exemple les composants d'analyse, de conception ou d'implémentation.

Les patterns les plus connus, comme ceux du GoF [Gamma et al., 95] et les GRASP [Larman 02] par exemple, sont des patterns dédies à la phase de conception. Les patrons de P. Coad [Coad 95] sont cependant dédies à la phase d'analyse. Les patterns de [Adams et al., 95] concernent la phase de recensement des besoins. Les patterns de S.W. Ambler [Ambler 98] sont des patterns qui couvrent l'ensemble du cycle de développement des systèmes.
Les patterns proposes par L. Gzara [Gzara 00] couvrent les étapes d'analyse et de conception.

2.5 Exemples des patterns d'ingénierie :

Selon le type de connaissances capitalisées par les patterns. Il y a deux types de patterns, à savoir les patterns produits et les patterns processus. Dans ce qui suit nous allons présenter un exemple de chaque type.

2.5.1 Patterns processus :

Un pattern processus décrit en général un processus de développement dans le but de répondre à des besoins spécifiques. Le pattern Revue Technique (Technical Review), extrait du système de patterns proposé par Scott W.Ambler [Ambler 98], est un exemple d'un pattern processus. (Voir figure 2.2).

Pattern Revue Technique

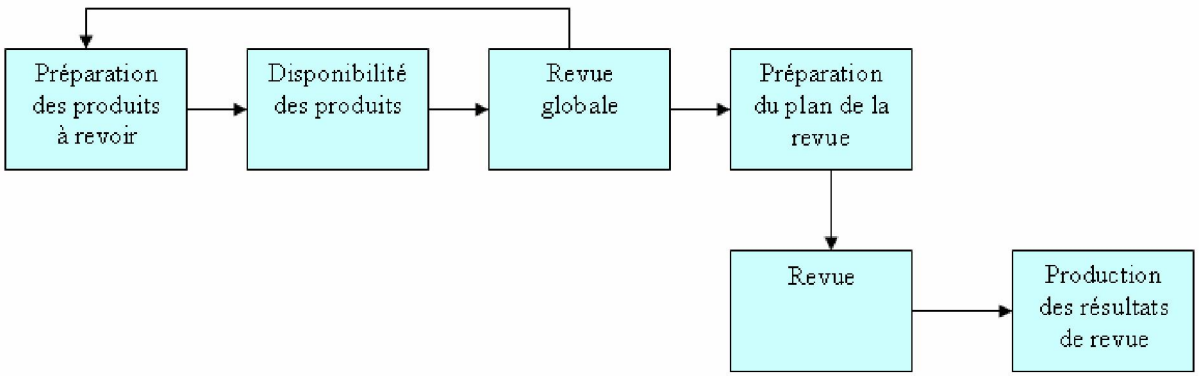
But : Décrit comment organiser, conduire et suivre la revue des délivrables d'un logiciel.

Forces (Contexte d'application):
Il y a 4 raisons pour lesquelles le pattern "Revue Technique" doit être appliqué :

- Les délivrables ont besoin d'être validés pour que le développement progresse.
- Les détections plus tôt des défauts d'un delivrable font baisser le coût de développement.
- Un travail est mieux valide par les individus.
- L'avancement du travail doit être notifié aux autres.

Contexte initial : Il y a des délivrables à inspecter et ils sont prêts à être inspecté.

Solution :



Contexte résultant :

Les qualités des delivrables (par rapport aux exigences des utilisateurs) sont assurées par l'équipe d'inspection. L'équipe de développement comprend mieux les delivrables produits et comment ils sont intègres au projet en cours.

Figure 2.2. Pattern Revue Technique (Pattern processus).

2.5.2 Patterns produit :

Ces patterns sont classifiés en fonction de la phase de développement du logiciel, tel que, les patterns d'analyse, de conception,…etc.
Les patterns d'architecture sont d'autres types de patterns qui s'adressent généralement à la phase de conception.

a) Patterns destinés à la phase de recensement des besoins :

Permettent une description des besoins des utilisateurs, sous la forme de comportements génériques attendus du système à développer. En général, il s'agit de comportements fonctionnels. Dans certains cas, les comportements non fonctionnels, tels que la performance ou la fiabilité, peuvent cependant être considérés. Les patterns destinés aux systèmes de télécommunication tolérants aux pannes [Adams et al., 95] traitent, par exemple, quelques besoins non fonctionnels tels que la fiabilité [Rising 00].

Généralement, les patterns de recensement des besoins font partie des patterns d'analyse.

b) Patterns d'analyse :

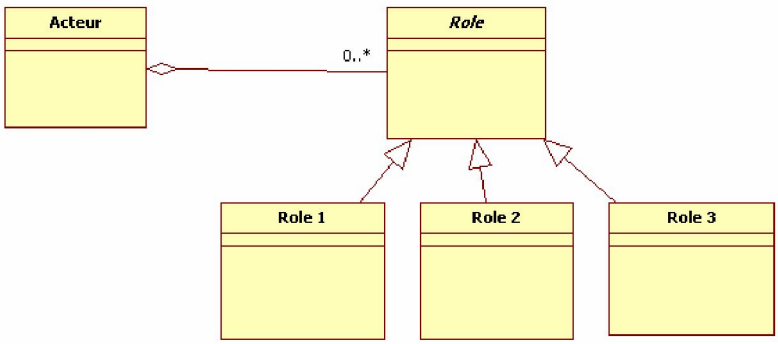
Ces patterns traitent des problèmes qui apparaissent lors de l'analyse des besoins des systèmes. Un pattern de gestion de ressources permet, par exemple, de répondre à un problème de ressources humaines, qui comprend l'affectation de personnes à des activités, ou de répondre à un problème de gestion de ressources matérielles. Pour réaliser l'affectation de machines dans un contexte de production. Le pattern Role propose par P.Coad [Coad 92] est un exemple de pattern d'analyse. (Voir figure 2.3).

Pattern Role

Intention :
Le pattern Role donne la possibilité à un objet d'attacher et/ou de perdre plusieurs rôles.

Motivation :
Tout objet peut avoir des propriétés variables selon le role joué. Un livre de bibliothèque par exemple, a des propriétés tels que le numéro d'exemplaire, le N° ISBN, le nombre de pages, etc. Il pourra jouer plusieurs rôles fortement dépendants des processus métiers de la bibliothèque : l'emprunt, la réservation et la maintenance.

Solution :
La classe Acteur est associée à une classe Role abstraite. Chaque acteur est lie à des instances des classes Rôles spécifiques concrètes (Role1, Role2, etc.). Chacune d'elles représentant un de ses rôles. Cette approche a l'avantage d'être plus flexible que l'utilisation de l'héritage multiple.



Le pattern Rôle est destiné à un contexte général. Il existe aussi des patterns d'analyse qui sont destinés à des situations particulières, comme par exemple le pattern Nouvelle-Operation-Bancaire [Front et al., 99].

Figure 2.3. Pattern Role (Pattern d'analyse).

c) Patterns de conception (design patterns) :

Ces patterns sont les plus connus et les plus utilisés parmi les patterns d'ingénierie. Ils identifient, nomment et abstraient les connaissances convenues de plusieurs expériences, répondant à des problèmes de conception. Tel que les patterns de GoF qui seront détailles par la suite, dans la section 4.2.

d) Patterns d'architecture :

Ce sont des patterns de conception qui déterminent la structure de base d'une application. Ils offrent un ensemble de sous-systèmes prédéfinis, spécifient leurs responsabilités, et les collaborations et communication entre ceux-ci. Le pattern MVC2 est un exemple de ce type de pattern [Turner et al., 2003]. (Voir figure 2.4).

Pattern Modèle-Vue-Controleur 2

Intention :
Assure l'indépendance du modèle par rapport à l'interface. Les objets sont séparés en trois composants : le modèle, la vue et le contrôleur. Le modèle contient la logique et l'état de l'application. La vue représente l'interface utilisateur. Le contrôleur réagit aux requêtes de l'utilisateur en effectuant les actions nécessaires sur le modèle.

Solution :

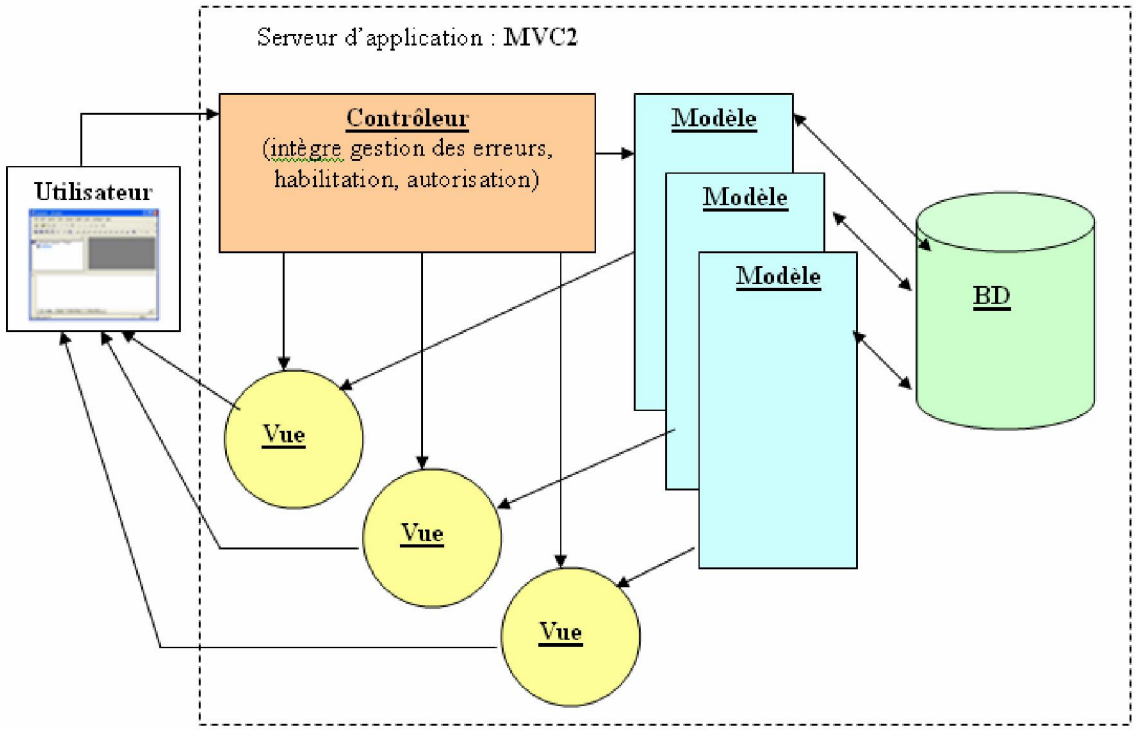


Figure 2.4. Pattern M.V.C 2 (Pattern d'architecture).

e) Pattern d'implémentation :

Les patterns d'implémentation (ou idiomes) sont des patterns de bas niveau qui traitent des problèmes d'implémentation. Ils peuvent décrire, par exemple, comment implémenter certains mécanismes absents dans des langages (simulation de l'héritage multiple en Java, par exemple). Ils sont généralement spécifiques à un langage de programmation donnée.

En général, il existe d'autres types de pattern, comme par exemple les antipatterns qui représentent des solutions fausses couramment rencontrées dans des résolutions de problèmes [Brown et al., 98].

3 Collections et formalismes de description de patterns :

En général, les patterns peuvent être regroupés dans un groupe appelé collection, de plus ils sont décrits d'une manière uniforme par des formalismes.

3.1 Formalismes de description de patterns :

En général, la spécification d'un pattern peut, être décrite sous une forme narrative (comme le cas pour le pattern « a place to wait »), ou sous une forme structurée qui améliore la lisibilité des formalismes (par exemple les patterns de [Gamma et al., 95]).

Un formalisme est la structure d'un ensemble de rubriques qui peuvent être utilisé, pour décrire d'une manière uniforme les spécifications de plusieurs patterns.

Plusieurs formalismes de description sont proposés [Coad 92], [Gamma et al., 95], [Ambler 97], [Fowler 97]. La plupart de ceux-ci sont équivalents. Ils différent par le nombre et le degré de détail de leurs rubriques, ainsi par le type des patterns qu'ils décrivent. Chaque formalisme doit décrire obligatoirement pour chaque pattern, le contexte, le problème, et la solution proposée pour celui-ci.

Nous retiendrons celles qui se rapportent au contexte de notre travail. C'est le formalisme de [Gamma et al., 95]. Où la représentation est décrite comme suit :

- Nom : Identifie le pattern.
- Intention : Précise le problème de conception traité par le pattern.
- Alias : Donne d'autres noms connus pour le pattern.
- Motivation : Présente un scénario d'application du pattern posant son problème dans un contexte particulier. Elle permet de mieux comprendre l'intérêt et la solution du pattern.
- Indications d'utilisation : Présente des situations dans lesquelles le pattern peut être utilisé.
- Structure : Représente les classes participant dans la solution du pattern et montre leurs interactions dans des diagrammes OMT.
- Constituants (Participants) : Définit les classes et/ou objets participants ainsi que leurs responsabilités.
- Collaboration : Précise la manière dont les constituants collaborent pour assumer leurs responsabilités.
- Conséquences : Décrit comment le pattern réalise ses objectifs et présente les résultats et les impacts de l'application de la solution du pattern.
- Implantation : Explique la technique d'implémentation du pattern. Elle présente les pièges existants et propose des solutions types en fonction du langage utilise, si elles existent.
- Exemple de code : Donne des parties de code (dans le langage C++) illustrant la solution du pattern.
- Utilisations remarquables : Présente des imitations du pattern considère dans des applications connues.
- Patterns apparentés : Référence d'autres patterns utilisant ou utilisés par celui-ci.

3.2 Collections de description de patterns :

Les patterns sont organisés et regroupés dans des collections de patterns. Ces groupes peuvent constitués des catalogues, des systèmes ou des langages. Ces trois types de collections se différencient par leurs organisations et leurs objectifs.

3.2.1 Catalogue de patterns :

Un catalogue de patterns est une collection des patterns regroupés dans un petit nombre de catégories variées [Buschmann et al., 1996]. Chaque catalogue de patterns propose sa propre classification, ainsi des relations entre les patterns qu'il regroupe, dont le but d'aider les développeurs à sélectionner et à choisir les patterns approprient à leurs besoins spécifiques.

Dans [Rising 00], propose un catalogue de patterns, qui regroupe les patterns par domaine d'application. Un autre exemple de catalogue, [Gamma et al., 95], classifie les patterns selon deux critères de sélection à savoir le rôle et le domaine. Et les relations entre patterns se résument à la relation patterns apparentés. Le rôle qui traduit ce que fait le pattern (création, structuration, comportement), et le domaine qui précise si le pattern s'applique en général à une ou plusieurs classes (ou objets).

3.2.2 Système de patterns :

Par rapport aux catalogues, le système de patterns est plus uniforme de plus il étend ce dernier par l'ajout des interactions plus précises [Appleton 97]. Il permet de combiner les patterns. Ainsi il montre la nécessite de ces derniers par rapport à d'autres patterns pour exprimer les combinaisons possibles. Ainsi, ils définissent des règles pour permettre ces combinaisons [Hachani 2006].

Ces systèmes offrent des architectures réutilisables. Ils présentent des collections de solutions qui coopèrent pour résoudre un problème complexe, de façon ordonnée. Pour atteindre un but prédéfini. En général, cette collection doit être organisée pour répondre à une problématique commune [Gzara 00].

3.2.3 Langage des patterns :

Un langage des patterns est une collection structurée de patterns, permettant de transformer des besoins et des contraintes en une architecture réutilisable [Coplien 98]. Il définit des règles précisant comment et quand appliquer un ensemble de patterns pour résoudre un problème spécifique. Ces règles montrent l'ordre d'application des patterns du langage.

La différence entre un système de patterns et un langage, est que ce dernier offre toutes les combinaisons possibles de patterns et leurs variations pour produire des architectures complètes et confirmées.

Le langage de pattern introduit une démarche d'utilisation de patterns partant d'un problème complexe et proposant, au travers des patterns, un enchaînement de solutions de plus en plus fines. Dans un langage de patterns, le guidage dans l'enchaînement d'utilisation des patterns est plus directif que dans un système de patterns [Hachani 2006].

4. Patterns de conception (Design Patterns) :

Nous avons introduit le concept du pattern d'une manière général. Dans ce qui suit, nous allons nous focaliser sur les design patterns (modèle de conception), et exactement sur les design patterns de GoF.

4.1 Pattern, instance du pattern et imitation :

Un pattern décrit une solution abstraite à un problème de conception. Il s'énonce nécessaire en termes génériques, mettant en évidence les noms des classes qui composent un pattern, et les relations qu'elles entretiennent. Il faudra ensuite particulariser ce discours à plusieurs niveaux, avec des noms de classes spécifiques au contexte de réalisation et en tenant compte de plusieurs compromis d'implémentation, pour obtenir une réalisation concrète, instance de ce pattern [Sunyé 1999].

L'imitation d'un pattern consiste à dupliquer puis adapter sa solution à un contexte spécifique.
Adapter un duplicata d'un pattern revient, à renommer, à redéfinir, à ajouter ou encore à supprimer une ou plusieurs propriétés des classes de la solution de ce pattern [Hachani 2006].

Nous allons considérer comme exemple le pattern 'Adapter', pour illustrer l'opération d'imitation dans le contexte de construction d'un éditeur graphique. Il s'agit tout simplement de réaliser une duplication de la solution de ce pattern ('Adapter'), suivie d'une adaptation concrète des 4 classes génériques (figure 2.5). Cette adaptation consiste à renommer les classes Client, Target, Adapter et Adaptee qui deviennent respectivement : Main, Editeurgraph, Cercle et CercleAdaptee.

Cette adaptation peut se poursuivre par la redéfinition, l'ajout ou la suppression d'autres propriétés. C'est le cas aussi pour la méthode Request() qui devient Dessiner(), ou encore la méthode SpecificRequest() pour Draw().

Il est possible d'ajouter d'autre méthodes ou d'attributs à ces différentes classes. Comme par exemple une méthode de déplacement Deplacer() pour la classe Editeurgraph.

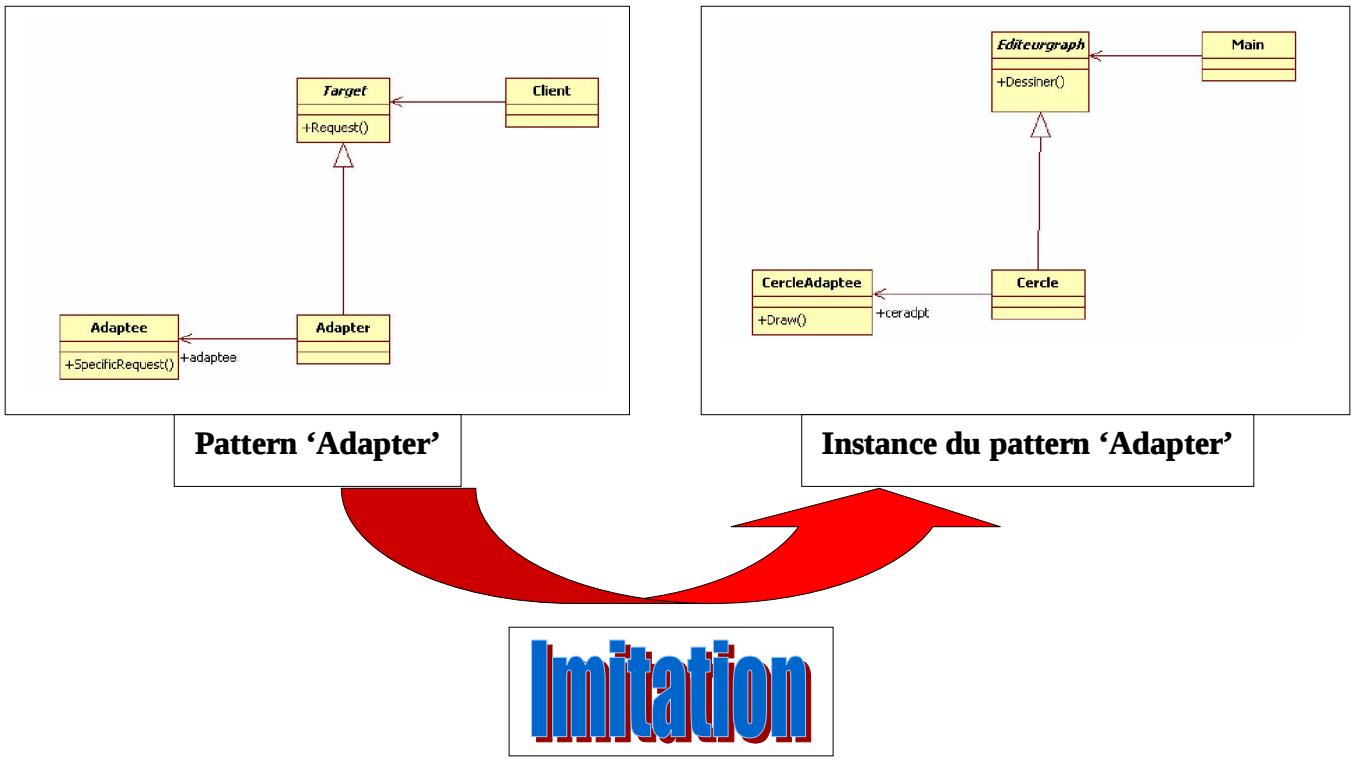


Figure 2.5. Pattern, Instance du pattern et Imitation.

4.2 Patterns de GoF :

Les patterns de GoF(Gang of Four) sont les plus connus et les plus utilisés parmi les design patterns [Hachani 2006]. De plus il rapporte au contexte de notre travail.

4.2.1 Catégories de Design Patterns de GoF :

Cette section montre la classification des design patterns de GoF. La classification aide à apprendre plus rapidement les patterns du catalogue, et elle permet également, d'orienter le travail de recherche.

Les design patterns sont classés selon deux critères (voir table 1). Le premier critère appelé **Role,** traduit ce que fait le pattern. Les patterns peuvent avoir un rôle soit **créateur**, soit **structurel,** soit **comportemental.** Les patterns créateurs concernent le processus de création d'objets. Les patterns structuraux s'occupent de la composition de classes ou d'objets. Les patterns comportements spécifient les façons d'interagir de classes et d'objets et de se répartir les responsabilités. Le second critère, appelé **Domaine,** précise si le pattern s'applique, dans le principe, à des **classes** ou à des **objets**. Les patterns de classes traitent des relations entre les classes et leurs sous-classes. Ces relations sont établies par héritage, elles sont donc statiques, établies préalablement à l'exécution, lors de la compilation. Les patterns d'objets traitent des relations entre objets, qui peuvent être modifiées à l'exécution, et sont donc plutôt dynamiques. [Gamma et al., 95]

a) Patterns de création :
La liste des patterns de cette catégorie est décrite comme suit :

Abstract Factory : Fournit une interface, pour créer des familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes.

Builder : Dans un objet complexe, dissocie sa construction de sa représentation, de sorte que, le même procédé de construction puisse engendrer des représentations différentes.

Factory method : Définit une interface pour la création d'un objet, tout en laissant à des sous-classes le choix de la classe à instancier. Une fabrication permet de déférer à des sous-classes les instanciations d'une classe.

Prototype : Spécifie les espèces d'objets à créer, en utilisant une instance de type prototype, et crée de nouveaux objets par copies de ce prototype.

Singleton : Garantit qu'une classe n'a qu'une seule instance, et fournit à celle-ci, un point d'accès de type global.

b) Patterns de structure :
La liste des patterns de cette catégorie est décrite comme suit :

Adapter : Convertit l'interface d'une classe en une interface distincte, conforme à l'attente de l'utilisateur. L'adaptateur permet à des classes de travailler ensemble, qui n'auraient pu le faire autrement pour causes d'interfaces incompatibles.

Bridge : Découple une abstraction de son implémentation associée, afin que les deux puissent être modifiés indépendamment.

Composite : Organise les objets en structure arborescente représentant la hiérarchie de bas en haut. Le composite permet aux utilisateurs de traiter des objets individuels, et des ensembles organisés de ces objets, de la même façon.

Decorator : Attache des responsabilités supplémentaires à un objet de façon dynamique. Il permet une solution alternative pratique pour l'extension des fonctionnalités, à celle de dérivation de classes.

Facade : Fournit une interface unifiée pour un ensemble d'interface d'un sous-systèm. Façade définit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.

Flyweight : Assure en mode partagé le support efficace d'un grand nombre d'objets à fine granularité.

Proxy : Fournit un subrogé ou un remplaçant d'un autre objet, pour en contrôler l'accès.

c) Patterns comportement :
La liste des patterns de cette catégorie est décrite comme suit :

Chain of responsability : Permet d'éviter de coupler l'expéditeur d'une requête à son destinataire, en donnant la possibilité à plusieurs objets de prendre en charge la requête. Pour ce faire, il chaîne les objets récepteurs, et fait passer la requête tout au long de cette chaîne jusqu'à ce que des objets la prennent en charge.

Command : Encapsule une requête comme un objet, ce qui permet de faire un paramétrage des clients avec différentes requêtes, files d'attente, ou historiques de requêtes, et d'assurer le traitement des opérations réversibles.

Interpreter : Dans un langage donné, il définit une représentation de sa grammaire, ainsi qu'un interprète qui utilise cette représentation pour l'analyser la syntaxe du langage.

Iterator : Fournit un moyen pour accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation sous-jacente.

Mediator : Définit un objet qui encapsule les modalités d'interaction de divers objets. Le médiateur favorise les couplages faibles, en dispensant les objets d'avoir à faire référence explicite les uns aux autres ; de plus, il permet de modifier une relation indépendamment des autres.

Memento : Sans violer l'encapsulation, acquiert et délivre à l'extérieur une information sur l'état interne d'un objet, afin que celui-ci puisse être rétabli ultérieurement dans cet état.

Observer : Définit une corrélation entre du type un à plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent, en soient notifiés et mis à jour automatiquement.

State : Permet à un objet de modifier son comportement lorsque son état interne change. L'objet paraîtra changer de classe.

Strategy : Définit une famille d'algorithmes, encapsule chacun d'entre eux, et les rend interchangeables. Une stratégie permet de modifier un algorithme indépendamment de ses clients.

Template method : Définit le squelette de l'algorithme d'une opération, en déléguant le traitement de certaines étapes à des sous-classes. Le patron de méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de l'algorithme.

Visitor : Représente une opération à effectuer sur les éléments d'une structure d'objet. Le visiteur permet de définir une nouvelle opération sans modifier les classes des éléments sur lesquelles il opère.

Le tableau 2.1, résume la classification des design patterns de GoF :

		Rôle		
		Créateur	**Structurel**	**Comportemental**
Domaine	**Classes**	Factory Method	Adapter	Interpreter Template Method
	Objets	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tableau 2.1. La Classification des patterns de GoF

4.3 Exemple du pattern composite :

La figure 2.6 présente le pattern Composite, il appartient au pattern de structure.

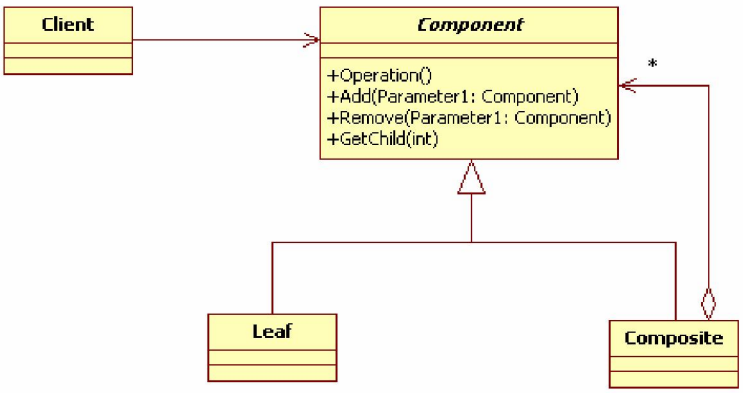
Patron Composite

Intention : Organise les objets en structure arborescente représentant la hiérarchie de bas en haut. Le composite permet aux utilisateurs de traiter des objets individuels, et des ensembles organisés de ces objets, de la même façon.

Indications d'utilisation : le pattern Composite est utilisé dans les situations suivantes.

- Pour représenter des hiérarchies de l'individu a l'ensemble.
- Dans le cas où, les clients veulent traiter de façon uniforme tous les objets d'une structure. A savoir les objets composés et les objets individuels

Structure :



Participants :

- **Component** : Représente l'interface des objets entrant dans la composition.
- **Leaf** : Représente des objets feuille dans la composition. Elle définit le comportement d'objets primitifs dans la composition.

Figure 2.6. Pattern Composite

4.4 Intérêt des design patterns :

Les design patterns sont très utilisé dans le domaine du génie logiciel, surtout dans les frameworks et les composants logiciels. Par exemple, dans [Johnson 92] documente le framework d'éditeurs graphiques HotDraw par l'utilisation d'un ensemble des patterns de GoF [Gamma et al., 95] tel que 'Composite', 'Decorator', 'Observer' et 'State'. Les composants comme EJB et .Net utilisent aussi les patterns dans leurs conceptions internes.

Les design patterns peuvent utiliser pour gérer certains services non fonctionnels (tels que la persistance, la sécurité…etc), dans les systèmes. Afin d'améliorer la réutilisation des systèmes, par la séparation de la partie fonctionnelle de la partie non fonctionnelle.

Malgré que les design patterns exposent une large utilisation, ils présentent, toutefois, quelques limites et problèmes qui seront présentés dans la section suivante.

4.5 Problèmes et limites d'utilisation des design patterns :

En général les design patterns se basent sur l'héritage, la composition, ou encore la délégation explicite pour définir leurs structures, ainsi leurs comportements. Ils ne sont pas liés uniquement au niveau conceptuel, mais ils peuvent toucher l'implémentation des applications. Ils dictent une structure aux conceptions et aux programmes qui pose plusieurs problèmes et limites lies directement à la dispersion et à l'enchevêtrement des éléments conceptuels de ces patterns, ainsi de leurs codes.

Selon [Hachani 2006], il y a quatre problèmes et deux limites. Les problèmes liés à l'utilisation les patterns sont : l'héritage, la violation d'encapsulation, la surcharge de l'implémentation et la tracabilité. Les deux limites portent sur le code non réutilisable, la difficulté de la maintenance et l'évolution.

4.5.1 Problèmes d'utilisation des design patterns :

Dans ce qui suit, nous allons détailler les principaux points :

a) Héritage :

Plusieurs patterns utilisent l'héritage. Ce mécanisme garantit la liaison du code des classes de l'application et de celui des patterns, et réduit ainsi la réutilisation séparée. De plus, si le langage de programmation ne supporte pas l'héritage multiple tel que Java, l'utilisation des patterns sera difficile. Surtout pour les classes possédants une super-classe.

b) Violation d'encapsulation :

Ce problème se pose, lors de l'utilisation de la délégation explicite et de la composition pour placer des comportements relatifs à certaines classes dans d'autres classes. Prenons exemple d'utilisation du pattern Visitor. Les classes qui jouent le rôle de visiteur accèdent à des attributs (ou des méthodes) définis dans les classes de l'application pour pouvoir effectuer un traitement. Pour cela ces dernières doivent posséder des attributs et des méthodes publiques, pour faciliter l'accès, alors que nous aurons pu souhaiter les déclarer privés ou protégés.

c) Surcharge de l'implémentation :

La délégation explicite est très utilisée dans les patterns. Elle permet de repartir les comportements entre les objets en interaction. Un problème majeur de cette délégation est l'augmentation du nombre de messages entre les objets. Ce qui implique que le code devient de plus en plus illisible. Par exemple dans le pattern 'Command', chaque instance de **Invoker** est associée à une instance de **ConcreteCommand**. Ainsi que chaque instance de **ConcreteCommand** est associée à une instance de **Receiver**. De plus chaque appel à l'instance **ConcreteCommand** résulte en l'exécution de l'opération de l'objet **Receiver**.

d) Tracabilité :

L'implémentation des instances des patterns est généralement dispersée dans plusieurs classes. Il est difficile de repérer les parties de programme propres à l'instance d'un pattern du reste du code de l'application.

4.5.2 Limites d'utilisation des design patterns :

Les quatre problèmes présentés sont liés à l'utilisation des patterns dans l'approche objet. Par contre les limitées sont liés principalement à la dispersion et à l'enchevêtrement du code, des instances des patterns [Hachani 2006].Ces limites sont :

a) Code non réutilisable :

En général, le code des instances des patterns est dispersé dans l'ensemble des classes qu'ils touchent. Donc il sera difficile de le réutiliser indépendamment de l'application. En effet, les design patterns offrent une réutilisation conceptuelle, mais non une réutilisation au niveau implémentation. Par ailleurs, l'enchevêtrement du code lié aux instances des patterns, avec les classes sur lesquelles il s'applique, rend ces derniers difficiles à réutiliser dans d'autres contextes, indépendamment de ces instances.

b) Maintenance et évolution difficiles :

En général, la dispersion et l'enchevêtrement du code des instances des patterns rendent la maintenance et l'évolution difficiles. Il lié à une instance d'un pattern est difficilement localisé, à cause qu'il est dispersé sur plusieurs classes de l'application. De plus, l'enchevêtrement des instances des patterns, avec le reste de l'application, rend le code non lisible. Pour cela, il sera par conséquent difficile à maintenir et à faire évoluer.

5 Implémentation des design pattern à l'aide de l'approche par aspect :

L'implémentation des design patterns à l'aide de l'approche objet ne fournit pas une bonne solution pour aboutir à des programmes clairs et élégants. La programmation orientée aspect [Kiczales et al., 97], [Pawlak et al., 04] viennent pour régler les limites de la programmation orienté objet. Pour cela, il sera intéressent de voir comment les patterns peuvent être implémentés en aspect.

Il existe plusieurs travaux qui implémentent les design patterns à l'aide de l'approche par aspects. Nous présentons par la suite rapidement chacun de ces travaux. Puis nous nous interresons à l'implémentation de Hannemann et Kiczales [Hannemann et Kiczales, 2002].

5.1 Différentes implémentations :

Nous présentons brièvement chacun de ces travaux [Hachani 2006]:

- [Lorenz 98] : Montre que le pattern 'Visitor', lorsqu'il est implémenté d'une certaine manière dans une version modifiée de Java est une forme de programmation par aspects.

- [Noda et al., 01] : Se sont intéressés a l'implémentation des patterns 'Observer', 'Composite' et 'Adapteur' de [Gamma et al., 95] en AspectJ et Hyper/J. Ils définissent un aspect correspondant à chaque classe introduite dans l'implémentation par objets d'un pattern. Puis à maintenir la connexion entre les classes d'application et ces aspects par l'introduction des aspects supplémentaires.

- [Hannemann et al., 02] : Fournissent aussi des implémentions des patterns de GoF [Gamma et al., 95] en AspectJ, et, ils reproduisent totalement ou partiellement, les structures par objets des solutions proposées par Gamma.

- [Nordberg 01] : Propose de nouvelles implémentations en AspectJ pour 12 design pattern par objets. Il montre comment ces implémentations peuvent éliminer les dépendances dans la réalisation de ces patterns.

- [Hirschfeld et al., 03] : Ils ont implémenté en AspectS quelques design patterns de GoF.

- [Clarke et al., 01] : Leur travail consiste donc à trouver des design patterns par aspects.

- [Hachani 02] : Son travail est un peu similaire à [Hannemann et al., 02], sauf pour quelques différences (comme par exemple le pattern 'Visitor' où [Hannemann et al., 02] réalisent le pattern 'Visitor' par aspects par la reproduction d'une partie de la structure proposée dans [Gamma et al.,95], alors que [Hachani 02] se focalise sur l'intention de ce pattern, où elle a abouti à une solution différente, ayant un nombre de constituants plus réduit).

Nous remarquons que :

- Chaque auteur a sa propre méthode de voir l'aspect dans les patterns (pour un même pattern, il est possible d'avoir plusieurs implémentations aspect). De plus chaque implémentation a ses avantages et ses inconvénients par rapport à l'autre.

- Le passage des design patterns en objet vers la solution aspect est faite à travers le langage naturel.

En général, il y a deux types d'approches pour implémenter les design patterns en aspect :

- Approche directe : c'est une implémentation directe des design pattern à l'aide de l'aspect, c'est-à-dire, passer du problème de conception des patterns vers une solution native à l'aide des aspects, tel que la solution de [Hirschfeld et al., 03].

- Approche indirecte : Dans ce cas, il y a un passage intermédiaire entre le problème de conception des patterns et l'implémentation à l'aide de l'aspect. Ce passage est la solution objet des design patterns, telle que la solution proposée par [Hannemann et Kiczales, 2002], que nous détaillons dans la section suivante.

5.2 Implémentation de Hannemann et Kiczales :

Nous allons discuter dans ce point l'implémentation aspect de Hannemann et Kiczales. Cette solution présente beaucoup d'avantage par rapport à d'autre solution pour implémenter les design pattern à l'aide de l'approche par aspect. Ces avantages sont :

- La considération de tous les patterns (contrairement à [Noda et al., 01] qui a considéré 3 patterns et [Nordberg 01] 12 patterns).

- Il se base directement sur les concepts Aspect pour définir les nouvelles solutions par aspects des patterns (contrairement à [Lorenz 98] qui se focalisent sur le pattern 'Visitor') .

- Le langage utilisé c'est AspectJ (contrairement à [Hirschfeld et al., 03] qu'utilise le langage AspectS).

5.2.1 Présentation de la solution de Hannemann et Kiczales :

Dans ce point nous allons représenter la solution de Hannemann et Kiczales qui constitue à implémenter les patterns à l'aide d'aspect (paradigme aspect). Dans ce travail, l'intention du pattern ne change pas mais qui change, c'est sa structure. Dans certain cas cette dernière change avec une réduction des participants. Dans d'autre cas la structure ne change pas, mais uniquement l'implémentation au niveau physique qui change. La solution de Hannemann et Kiczales ce base sur la notion de rôle qui sera définie dans le point suivant.

a) Définition du rôle :

Les rôles sont les fonctionnalités des participants dans le contexte du pattern. Par exemple, pour le pattern Observer assigne deux rôles à ses participants, qui sont Subject (Sujet) et Observer (Observateur).
En général, il y a deux types de rôles :

- Rôle surimposé : Il vient de s'ajouter à la préoccupation principale de la classe. Il se mélange avec celle-ci pour faire un aspect.

- Rôle de définition : Il constitue la préoccupation principale de la classe. La classe définit un comportement spécifique pour ce rôle, difficile d'extraire.

b) Aspects et Patterns :

Un certain nombre de patterns incluent une structure transversale dans la relation entre les rôles et les classes des participants pour chaque instance de pattern. Par exemple dans le pattern Observer, l'opération qui change les 'Sujets' doit déclencher les notifications à leurs Observateur, (c'est-à-dire l'acte de notification entrecoupe une ou plusieurs opérations dans chaque 'Suject' pour chaque instance de ce pattern). Dans le pattern 'Chaine of Responsability' tous les 'Handlers' (gestionnaires) doit avoir la capacité d'accepter les requêtes, ou les événements pour les manipuler, ou bien pour acheminer les requêtes au successeur. Le mécanisme de gestion (acceptation et acheminement) entrecoupe les 'Handlers' gestionnaires.
L'implémentation aspect des design patterns, change d'un pattern à un autre. Nous allons voir par la suite un exemple pour le pattern 'Observer'.

c) Exemple du pattern 'Observer' :

Dans la structure du pattern 'Observer', il y a des parties communes pour toutes les instances du pattern, et d'autre spécifique pour chaque instance.

La partie commune à toutes les instances du pattern définit :

- Des classes jouant les rôles de 'Sujet' et d' 'Observateur'.

- La relation entre les Sujets à leurs Observateurs. Suivant la cardinalité : un Sujet peut avoir un ou plusieurs Observateurs ;

- La définition de la mise à jour. Le Sujet changé déclenche les mises à jour à ses Observateurs.

Cette partie est définie dans l'aspect abstrait appelé aussi Protocole. Chaque point représenté ci-dessus est présenté par les numéros 1,2 et 3 dans la figure 2.7.

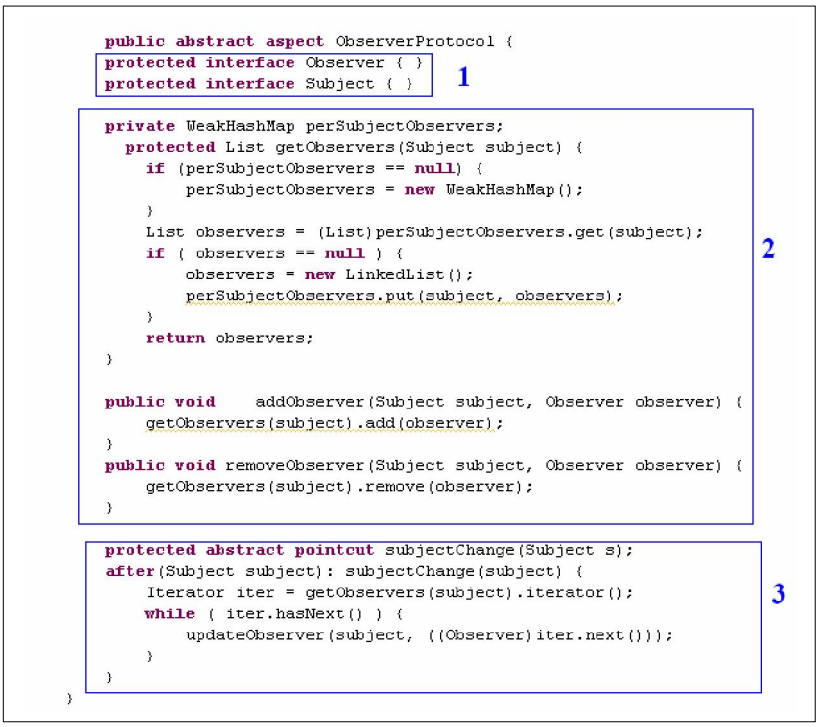


Figure 2.7. Aspect abstrait du 'Observer' (Protocole du 'Observer').

La partie spécifique à chaque instance du pattern définit:

- Les classes 'Sujets' et les classes 'Observateurs'.
- L'ensemble des modifications des sujets qui déclenchent des mises à jour sur l'Observateurs.
- Spécification de la mise à jour pour chaque type de Observateurs.

Cette partie est définie dans les aspects concrets.

d) Classification des patterns selon Hannemann et Kiczales :

Les patterns sont regroupés par leurs caractéristiques communes (à savoir la structure) ou bien selon leurs ressemblances en implémentation aspect. Les groupes des patterns sont :

Groupe 1 : « Les rôles sont uniquement utilisé dans l'aspect pattern » :

Les patterns appartenant à cette catégorie sont : Observer, Composite, Command, Mediator, et Chain of Responsibility.
Ces patterns sont similaires au pattern 'Observer'. Ils introduisent les rôles avec des interfaces protégés, qui sont utilisés uniquement dans le pattern. (Par exemple le rôle Observateur et Sujet dans le pattern Observer). Dans AspectJ, ces rôles sont réalisés par des interfaces vides et protégées. Par exemple : **Protected interface Subject {}**
Ils sont utilisés dans le protocole du pattern (aspect abstrait). Ce dernier introduit aussi une coupe abstraite. Puis par la suite concrétisé cette coupe pour chaque instance du pattern (aspect concret). Dans lequel capture des points de jonction qui doit déclencher les événements importants. (tel que l'exécution du Command dans le pattern Command).

Groupe 2 : « Les aspects comme créateurs d'objet » :

Les patterns appartenant à cette catégorie sont : Singleton, Prototype, Memento, Iterator, et Flyweight.
Ces patterns gèrent l'accès à des instances d'objet spécifique (tel que l'objet Singleton). Tous ces patterns offrent des méthodes de création au client. Ils permettent la création des objets à la demande. Ces patterns permettant la réutilisation par l'offert des aspects abstraits. Ces derniers contiennent le code pour la création. Dans l'implémentation d'AspectJ de ces patterns, les méthodes créateurs sont soit des méthodes paramétrées dans l'aspect abstrait, soit des méthodes attachées aux participants.

Groupe 3 : « Constructions du langage » :

Les patterns appartenant à cette catégorie sont : Adapter, Decorator, Strategy, Visitor, et Proxy.
Ces patterns peuvent être implémenté directement à l'aide des mécanismes et des concepts aspect (tel que le mécanisme aspect d'extension d'interfaces). Le pattern 'Adapter' et 'Visitor' peuvent être réalisé par le mécanisme aspect d'extension d'interfaces. Par contre, les patterns 'Decorator', 'Strategy' et 'Proxy' ont des implémentations alternatives basées sur l'attachement du 'code advice'.

Groupe 4 : « Héritage multiple » :

Les patterns appartenant à cette catégorie sont : Abstract Factory, Factory Method, Template Method, Builder, et Bridge.
Ces patterns ont des structures similaires. Ils utilisent l'héritage dans leurs implémentations (ainsi l'héritage multiple). Le langage Java ne permet pas d'utiliser la forme d'héritage multiple. Mais avec les mécanismes et les concepts d'AspectJ (tels que la déclaration d'héritage dans AspectJ). Il sera possible de modifier de façon transparente la hiérarchie d'héritage des classes existantes, pour permettre la forme d'héritage multiple.

Groupe 5 « Modularité du code disperser » :

Les patterns appartenant à cette catégorie sont : State et Interpreter.
Ces patterns introduisent un couplage fort entre leurs participants. L'implémentation AspectJ, permet de modulariser les parties des codes dispersées. Par exemple, dans le pattern 'State', l'entrecoupement du code entre les transitions d'états peuvent être modularisés dans l'aspect par l'utilisation du 'code advice' **after**.

Groupe 6 « Aucune bénéfices » :

Il y a uniquement le pattern Facade.
L'implémentation AspectJ est similaire à l'implémentation Java. Il fournit une interface unifier à des ensembles d'interfaces pour des sous-système, pour rendre ces derniers faciles à utiliser.

5.2.2 Avantages d'implémenter les design patterns à l'aide de l'approche par aspect :

Ces avantages sont :

- Une meilleure modularisation du code en concentrant celui-ci au sein de la notion aspect (localisation) ;
- Une meilleure réutilisation (en factorisant du code au sein d'aspect pouvant s'accompagner d'un meilleur degré d'abstraction) ;
- La possibilité de faire participer un même objet à plusieurs design patterns de manière transparente pour lui (composition) ;
- Couplage faible entre l'application et les design patterns qu'elle utilise (adaptabilité).

6 Rétro-ingénerie des design patterns :

La rétro-ingénierie des design patterns, est une technique consistant à analyser un programme, afin d'extraire des éventuelles instances. Ce point a un rapport au contexte de notre travail. Afin d'extraire la logique métier, à partir d'un code source toute en basant sur l'implémentation aspect des patterns.
Plusieurs travaux basés sur la détection des design patterns à partir d'un code source ont été proposé. Certains s'intéressent seulement à la spécification statique d'un pattern [Karmer et Prechlet 1996]. D'autres s'intéressent aussi à la description dynamique des design patterns [Heuzeroth et al., 2003]. Dans notre cadre de travail, nous nous intéressons, à les approches qui s'intéressent à la description statique et dynamique, tel que [Shi et al., 2006] et [Lee et al., 2007]. Dans [Lee et al., 2007] , le processus de détection des instances patterns, ce effectue par une analyse statique, puis une analyse dynamique et enfin par une analyse spécifique :

- L'analyse statique : Cette analyse, détecte uniquement les relations entre les classes, telles que le lien d'association ou le lien d'héritage…etc. Le pattern 'Adapter' est un cas d'un pattern qui peut être détecté par cette analyse.

- L'analyse dynamique : Il y a des patterns qui ne peuvent pas être détecter, par l'analyse statique seulement. Dans ce cas la détection de ce genre de pattern ce fait par la gestion des traces des appels des méthodes, plus le résultat de l'analyse statique. Parce qu'il y a des patterns qui ont une similarité dans leurs structures. Mais se distingue uniquement par leur comportement.

- L'analyse spécifique : Il y a des patterns qui peuvent être détectés uniquement par des mots-clés tels que le pattern 'Prototype' avec le mot-clé *clone,* ou bien, par une implémentation spécifique comme le cas pour singleton.

7 Conclusion :

Nous avons présenté dans ce chapitre les différents types des patterns, ainsi les différents types des design patterns. Nous avons montré à travers des travaux que la plupart des design patterns peuvent être bénéficiés des avantages de la programmation orientée aspect.

Cette implémentation aspect des patterns, constitue un passage de l'orienté objet vers l'orienté aspect dans l'implémentation des différentes instances des patterns. Pour réutiliser cette implémentation dans différents contextes de réalisation, nous présenterons dans le chapitre suivant, notre contribution à savoir la proposition d'un modèle de transformation conduisant ce passage.

Chapitre 03

Chapitre

III

Une Approche de transformation des Design Pattern vers des Programmes Orientés Aspects

1. Introduction :

L'implémentation des design pattern à l'aide de l'approche aspect, permet de séparer le code des instances des patterns de celui de l'application. Afin de bénéficier des avantages suivantes : la localisation, la réutilisation, la composition et enfin l'adaptabilité. Cette implémentation ou bien ce passage de l'objet vers l'aspect est effectué par le langage naturel. Ce problème est considéré comme le problème majeur de notre travail.

Ce chapitre présente notre contribution à savoir la proposition d'un modèle de transformation des design patterns vers des programmes orientés aspects. Pour répondre au problème posé de l'automatisation de passage des patterns de l'objet vers l'aspect. Cette automatisation permet au développeur de réduire le temps, l'effort ainsi le coût dans le développement des systèmes orientés aspects.

Dans ce chapitre nous présentons l'approche globale pour la transformation, permettant d'assurer le passage de l'OO vers l'OA. Cette approche constitue en un ensemble d'algorithmes destiné à traduire des patterns.

Nous proposons des situations d'application de ce modèle. Ces situations représentent des différents cas de transformation. Cette transformation peut apparaître, comme une génération de code. Elle est vue aussi comme une opération de rétro-ingénierie.

2. Approche globale pour la transformation:

Dans cette section, nous allons présenter un aperçu sur le modèle de transformation d'une manière globale et abstraite. Puis nous présentons par la suite, des situations d'application de ce modèle, pour considérer le point concret de ce dernier.

Comme nous avons vu dans le chapitre 2, que la plupart des travaux existants visent à montrer l'implémentation aspect des design patterns et à exposer les avantages de cette implémentation [Hannemann et Kiczales, 2002], [Noda et al., 2001] et, [Nordberg 2001]. Cette implémentation aspect des design patterns est en quelque sorte, un passage d'une solution objet vers une solution aspect des différentes instances des patterns. Mais aucun de ces travaux n'a mis le point sur la réutilisation de cette transformation, afin de l'appliquer à différents contextes de réalisation. De plus aucun outil n'a été proposé pour supporter ce genre de transformation.

Pour cela, notre modèle de transformation vise à définir des règles de transformation qui assure le passage d'une instance présenté dans le paradigme objet vers une nouvelle instance présenté dans le paradigme aspect. Nous allons considérer que ces instances qui seront transformées, sont présentées comme un modèle conceptuel orienté objet, ou bien, comme des programmes orientés objets. Après l'application du processus de transformation, nous obtenons respectivement, des programmes orientés aspects, ou bien un modèle conceptuel orienté aspect.

Pour réaliser cette transformation, nous allons suivre certaines étapes, qui seront présentées par la suite. Les règles de transformation sont représentées par des algorithmes de transformation, où chaque pattern possède son propre algorithme de transformation. Pour chaque algorithme, nous avons implémenté des règles pour traduire les participants d'une instance objet (Classes, Classes Abstraites, Interfaces) vers des nouveaux participants en aspect (Aspects, Aspects Abstraits, Classes, Classes abstraites, Interfaces).

La transformation des instances des patterns de l'objet vers l'aspect, revient en général à :

- Ajouter des aspects et/ou des aspects abstraits ;
- Supprimer des classes et/ou des classes abstraites et/ou des interfaces ;
- Modifier des classes et/ou des classes abstraites et/ou des interfaces. (la modification revient à ajouter et/ou à supprimer des méthodes et/ou des attributs).

Dans certains cas, un aspect contient des classes et/ou des interfaces. Dans ce cas précis l'idée revient à déplacer ces classes et/ou ces interfaces vers l'aspect (c'est-à-dire supprimer des classes et/ou des interfaces et ajouter un aspect qui contient ces dernières). Nous reviendrons en détail par la suite au chapitre 4.

Nous avons présenté, au dessus, un aperçu du principe de transformation du modèle. Cet aperçu décrit d'une manière général et globale le principe de transformation sans spécifier ni les types des modèles utilisés lors de la transformation, ni les différentes représentations des instances des patterns. Dans ce qui suit nous décrivons les différentes scénarios de transformation des instances des patterns.

Le terme design pattern est un concept lié au niveau conceptuel. Mais il est possible de voir plusieurs scénarios d'application de ce terme :

- Une instance du pattern au niveau conceptuel qui présente le modèle entier ;
- Une instance du pattern au niveau conceptuel qui présente un modèle partiel d'un modèle d'une application.
- Une instance du pattern au niveau physique, qui est représentée par des programmes.

Nous n'oublions pas que ces instances sont représentées dans le paradigme orienté objet.

Notre modèle de transformation sera utilisé pour traduire, les instances des patterns de l'objet vers l'aspect, dans les trois situations comme suit :

- Générer le code aspect des instances des patterns, à partir d'un modèle conceptuel orienté objet (le modèle représente uniquement une instance d'un pattern à la fois) [Berkane et al., 2008].
- Générer le code aspect des instances des patterns, à partir d'un modèle conceptuel orienté objet (le modèle sera conçu par les outils CASE existants).
- Effectuer la rétro-ingénierie afin d'extraire des éventuelles instances des patterns, en se basant sur l'implémentation aspect des patterns. Les instances des patterns seront représentées par un modèle conceptuel orienté aspect [Berkane et al., 2008a].

Nous allons proposer, par la suite un seul modèle de transformation qui vise les différentes situations, mais la représentation et la façon d'utiliser ce modèle seront différentes d'une situation à une autre.

3. Modèle de transformation des design patterns :

Avant d'exposer notre approche, nous allons répondre à ces questions :

- Quelle est l'implémentation aspect des patterns choisis ?
- Pourquoi cette nouvelle approche ?

3.1 Implémentation des patterns à l'aide de l'approche aspect :

Le modèle de transformation proposé, est fondée sur l'approche de [Hannemann et Kiczales, 2002] parce qu'ils considèrent tous les patterns. De plus c'est une approche indirecte pour effectuer le passage de l'objet vers l'aspect, la structure objet du pattern ne change pas beaucoup.

3.2 Définition d'une nouvelle approche de transformation :

Il existe beaucoup d'approches qui traitent de la transformation des modèles comme MDA (Model Driven Architecturel) la plus connus [MDA], ou bien QVT(Querry/View/ Transformation) [QVT], le langage ATL (Atals Transformation Language) [ATL] ….etc.

La transformation des modèles, est une transformation d'un concept du modèle source vers un concept du modèle cible. Cette transformation ne peut être effectuée que si nous définissons un méta-modèle pour le modèle source et un autre pour le modèle cible.

De plus, le problème de l'aspect, c'est qu'il n'existe pas un standard reconnu pour ce paradigme. Ils existent plusieurs tentatives pour définir un méta-modèle dans UML pour l'aspect (la metamodélisation et le stéréotypage) [Hachani, 2006]. Ils existent aussi plusieurs outils aspect tels que AspectJ, JAC, et d'autres, où chaque outil présente l'aspect d'une manière différente par rapport à l'autre.

Cependant, notre modèle de transformation, traduit un modèle source (Ce modèle doit être en orienté objet. Il est représenté soit par UML pour le modèle conceptuel, soit par Java pour le modèle physique) vers un modèle cible (Ce modèle doit être en orienté aspect. Il est décrit soit par UML/AspectJ pour le modèle conceptuel, soit par AspectJ pour le modèle physique). Les règles de transformation sont les patterns (C'est-à-dire remplacer chaque instance objet qui peut apparaître dans le modèle source par son équivalente en aspect).

Pour cela, nous proposons un modèle de transformation qui reflète cette idée de transformation. Le modèle source est un modèle contenant des instances des patterns en objet. Par contre, le modèle cible est un modèle contenant ces mêmes instances mais en aspect.

3.2.1 Equivalence des instances :

Après avoir supposé que chaque instance en objet a son équivalent en aspect. Nous avons ré-implémenté la solution aspect pour certains patterns de [Hannemann et Kiczales, 2002], afin de garantir notre supposition.

Certains patterns de [Hannemann et Kiczales, 2002] ont une solution aspect différente par rapport à son similaire en objet. Cette différence peut être vue comme suit :

- **Une solution de composition :** Certain patterns ont une solution de composition avec d'autres patterns, tel que le pattern 'Composite' avec le pattern 'Visitor'. Dans ce cas nous avons supprimé cette dépendance. Dans le pattern 'Composite', nous supprimons les fonctionnalités du pattern 'Visitor'. Cette suppression revient à supprimer les opérations de 'Visitor' qui se trouvent dans le protocole de 'Composite'.

- **Une différence en cours d'exécution :** L'exécution de certains patterns implémentés en aspect ne donne pas la même solution qu'en objet, tel que **'**ChainOfResponsibility'. Dans ce cas nous avons modifié la solution objet, pour l'adapter à la solution aspect.

3.2.2 Nouvelle Classification des patterns :

Nous avons classifié les patterns en deux classes, une classe pour les patterns pouvant être transformés, et la deuxième classe pour les patterns exclus de la transformation. Les raisons d'écarter ces patterns sont présentées comme suit :

- Le pattern Façade, ne sera pas considéré par la transformation, parce qu'il a une implémentation orientée aspect strictement identique à l'implémentation orientée objet [Hannemann et Kiczales, 2002], [Pawlak et al., 2004], [Hachani 2006] .

- Le pattern 'Singleton', ne sera pas également considéré par la détection, (malgre que ce pattern montre des avantages pour sa solution aspect) parce que l'implémentation aspect de ce pattern, peut avoir des effets de bord nuisibles [Pawlak et al., 2004].

- Les patterns 'Template Methode', 'Bridge' 'Abstract Factory', 'Factory Method' et 'Builder', ont peu de bénéfice [Hannemann et Kiczales, 2002].

- Les patterns 'State' et 'Interpreter' parce qu'ils définissent des rôles de définition [Hannemann et Kiczales, 2002]. Ces rôles constituent la préoccupation principale de la classe. Pour cela notre approche ne peut pas détecter ce genre de préoccupation, parce qu'elle change d'une application à une autre.

Le tableau 3.1 résume la nouvelle classification :

Classes	Patterns
Patterns exclus de la transformation	Facade, Singleton, Template Methode, Bridge, Abstract Factory, Factory Methode, Builder, Stat, Interpreter.
Patterns pouvant être transformés	Adapter, Composite, Iterator, Prototype, Proxy, Visitor, Chain of Responsibility, Command, Decorator, Flyweight, Mediator, Memento, Observer, Strategy.

Tableau 3.1. Nouvelle classification des patterns (selon la transformation).

3.2.3 Représentation des modèles :

Il est nécessaire de spécifier les modèles utilisés avant d'entamer la transformation. Nous allons considérer les représentations des modèles comme suit:

- Modèle conceptuel orienté objet représenté par UML [Muller et al., 2002];
- Modèle conceptuel orienté aspect représenté par UML pour AspectJ [Suzuki et al., 1999] ;
- Modèle physique orienté objet représenté par JAVA ;
- Modèle physique orienté aspect représenté par AspectJ [AspectJ].

Nous avons choisi le langage UML, comme représentation du modèle conceptuel orienté objet (est exactement le diagramme de classes et de séquences), car il permet d'exprimer et d'élaborer des modèles objets, indépendamment de tout langage de programmation. De plus il permet un gain de précision, de stabilité, de plus il encourage l'utilisation d'outils.

Nous avons sélectionné UML pour AspectJ, pour présenter le modèle conceptuel orienté aspect, et nous avons choisi exactement la solution Suzuki [Suzuki et al., 1999], car cette solution représente l'aspect par les stéréotypes. Les stéréotypes permettent d'utiliser un outil de modélisation standard UML déjà existant, pour exprimer des modèles.

Le langage Java est choisi comme langage pour représenter le modèle physique orienté objet. Parce que, il est le plus utilisé pour le développement. De plus il supporte plus d'extension par les outils qui supportent l'aspect, comme AspectJ, JAC, …etc.

Le langage AspectJ est le langage utilisé pour représenter le modèle physique orienté aspect. Ce langage supporte, plus de types de points de jonction, plus de type de 'code advice', enfin la plupart des implémentations de patterns à l'aide de l'approche aspect utilisent ce langage.

3.3 Principe de la transformation :

Nous allons exposer dans ce point notre modèle de transformation générale.

La description des design pattern dans le livre de GoF est faite dans un format unique [Gamma et al., 1995]. Chaque pattern est représenté en section conformément au canevas. La structure représente un élément de ce canevas, qui est une représentation graphique des classes du pattern. Chaque design pattern comporte au moins un diagramme de classes, le diagramme de séquences est utilisé pour décrire le comportement de certains patterns [Gamma et al., 1995]. Pour cela, nous considérons deux types de diagrammes dans le modèle conceptuel, à savoir le diagramme de classes UML et le diagramme de séquences UML. Le premier représente une structure statique et le deuxième représente un comportement dynamique.

Dans le modèle physique, la structure statique des instances des patterns est représentée par les classes, et la relation entre les participants tel que l'héritage. Pour, le comportement dynamique est représenté par les différents appels des méthodes entre les classes.

Le modèle de transformation vise à transformer des instances orientées objet vers des instances orientées aspect. Pour cela, quatre cas possibles de transformations sont à considérer (figure 3.1) :

- Transformation d'une instance du niveau conceptuel O.O vers une instance d'un niveau conceptuel O.A.

- Transformation d'une instance du niveau conceptuel O.O vers une instance d'un niveau physique O.A.

- Transformation d'une instance du niveau physique O.O vers une instance d'un niveau physique O.A.

- Transformation d'une instance du niveau physique O.O vers une instance d'un niveau physique O.A.

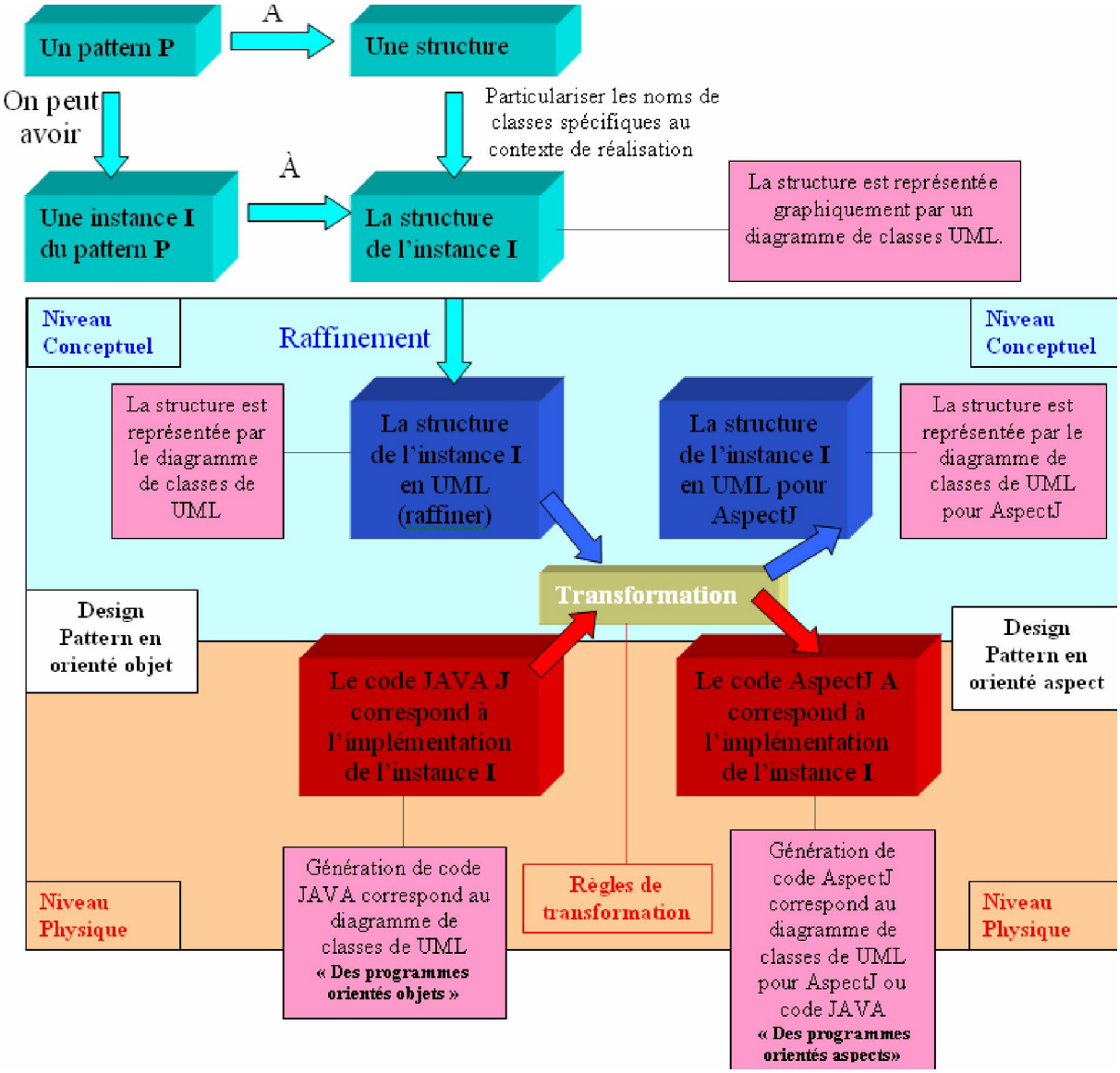


Figure 3.1. Modèle de transformation des design pattern vers des programmes orientés aspect.

La transformation d'une instance du niveau conceptuel vers une instance d'un niveau physique, ou inversement, peut donner des résultats efficaces :

- Génération du squelette de code, afin de faciliter l'implémentation. Dans le cas du passage du modèle conceptuel vers le modèle physique.

- Effectuer la rétro-ingénierie, afin d'extraire la logique métier d'une application, par la séparation des fonctionnalités des instances des patterns de ceux de l'application.

Par contre, la transformation effectuée au même niveau, ne donne pas beaucoup d'intérêt, pour plusieurs raisons :

- Une fausse transformation peut avoir un échec lors de l'exécution de l'application en aspect.
- Le passage définit dans le même modèle conceptuel, peut changer la sémantique de l'application.

Pour effectuer les différents passages, nous avons défini des règles de transformation, où leur syntaxe est définie dans le point suivant.

3.4 Syntaxe algorithmique pour la transformation:

La syntaxe algorithmique utilisée pour implémenter les algorithmes de transformation, sert à faciliter la transformation pour les deux niveaux conceptuel et physique. Pour cela nous définissons d'abord la syntaxe algorithmique commune pour le niveau conceptuel (UML et UML/AspectJ) et physique (JAVA et AspectJ), puis la syntaxe spécifique pour chaque niveau.

3.4.1 Syntaxe algorithmique commune pour le niveau conceptuel et physique :

La figure 3.2 représente la syntaxe algorithmique commune pour les deux niveaux conceptuel (UML et UML/AspectJ) et physique (JAVA et AspectJ) telles que : les classes, les interfaces et enfin les aspects.

Concept	Syntaxe algorithmique	Code JAVA et AspectJ	Notation UML et UML pour AspectJ

CLASSE	Créer class client {
	Nom_de_classe := « Client » ;
	Type_de_classe :=Concrete ;
	Heritage := « » ;
	
	Attributs :={**private** int i, **public** String s }
	Operations:={**public** void AddClient(Client c), public void RemoveClient(Client c)}
	}
		public class Client {
		private int i ;
		public String s;
		
		public void AddClient(Client c)
		{.....}
		
		public void RemoveClient(Client c)
		{.....}
		}
			

INTERFACE	Créer interface but {
	Nom_de_interface := «But » ;
	Heritage := « » ;
	
	Operations :={**public** void add(int i)}
	}
		public interface But {
		public void add(int i);
		}
			

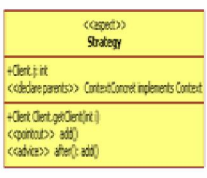
ASPECT	Créer aspect strategy {
	Nom_de_aspect:= « Strategy» ;
	Type_de_aspect := Concret;
	Heritage := « » ;
	
	Declare_introd_attributs :={**public** int Client.j}
	Declare_parents :={ContextConcret **implements** Context}
	Declare_introd_operations :={**public** Client Client.getClient(int i)}
	Pointcut:={ **public** add()
	:**call(void Client.AddClient(Client)** }
	Advice(add):={after():add()}
		public aspect Strategy {
		public int Client.j;
		declare parents: ContextConcret implements Context;
		
		public Client Client.getClient(int i)
		{.....}
		
		public pointcut add : call void Client(Client);
		
		after():add() {.....}
		}
			

Figure 3.2. Syntaxe algorithmique commune pour le niveau conceptuel et physique

Nous notons dans ce cadre aussi des mots-clés spécifiques à une utilisation restreinte (voir tableau 3.2) :

Mot-clés	Sémantique
Transformation_Prototype (Classe Client, **?** Interface Prototype ,….)	Le point d'interrogation signifie que l'interface Prototype est facultative.
Operations :=ConcretePrototype.Operations **/**{public * clone()} ;	Le slache signifie toutes les méthodes de la classe ConcretePrototype à l'exception la méthode **clone.**
Operations :=ConcretePrototype.Operations /{public ***** clone()} ;	L'étoile signifie que quelque soit le retour de la méthode **clone**. Il peut être utilisé dans les paramètres.
Adaptee.nom_de_classe**'.'**Adapter.opera….	Le point entre deux guillemets signifie un point réel. (Ce point est utilisé dans le mécanisme d'introduction).
Pointcut :={**Pour non_fin de** Command.Operations **faire** {private commandTrigger…..}}	Il faut dupliquer ce point de jonction pour chaque méthode de Command.

Tableau 3.2. Mots-clés spécifiques

3.4.2 Syntaxe algorithmique spécifique pour chaque niveau :

Dans ce point, nous définissons la syntaxe spécifique à chaque niveau qui ne doit pas apparaître en même temps pour les deux niveaux conceptuel et physique. Par exemple pour le niveau conceptuel, les associations sont représentées différemment du niveau physique.

a) Syntaxe spécifique pour le niveau conceptuel (UML et UML pour AspectJ) :

Nous montrons dans ce point la syntaxe algorithmique spécifique au niveau conceptuel, comme par exemple les associations (voir figure 3.3) :

La procédure **AfficherNote (N1, N2,…Nm, classe)** : a pour but d'afficher la note correspondante à une classe ou aspect donné. Les symboles N1,N2,…,Nm, peuvent être des classes et/ou des aspects et/ou des opérations et/ou des attributs et/ou des pointcuts…etc

La procédure **VerifierLien ()** : a pour but de mettre à jour les liens entre les classes, les aspects et classes-aspects comme le lien d'héritage, d'association, ou bien le lien de réalisation. Par exemple, pour établir la relation d'association entre les classes, la procédure vérifie l'occurrence des instances des classes dans les attributs.

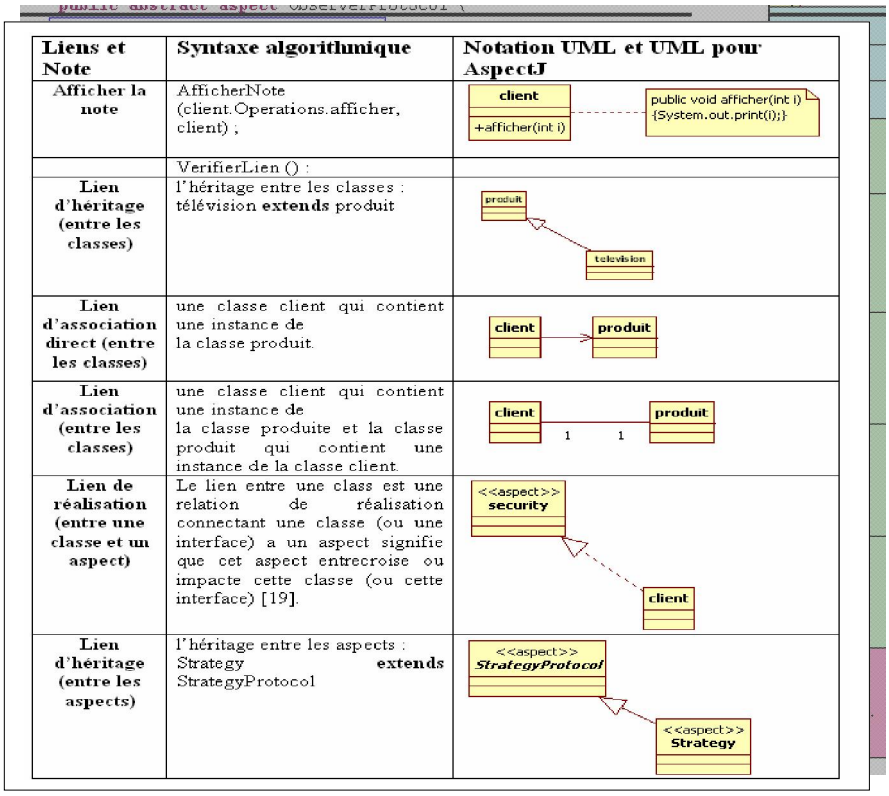
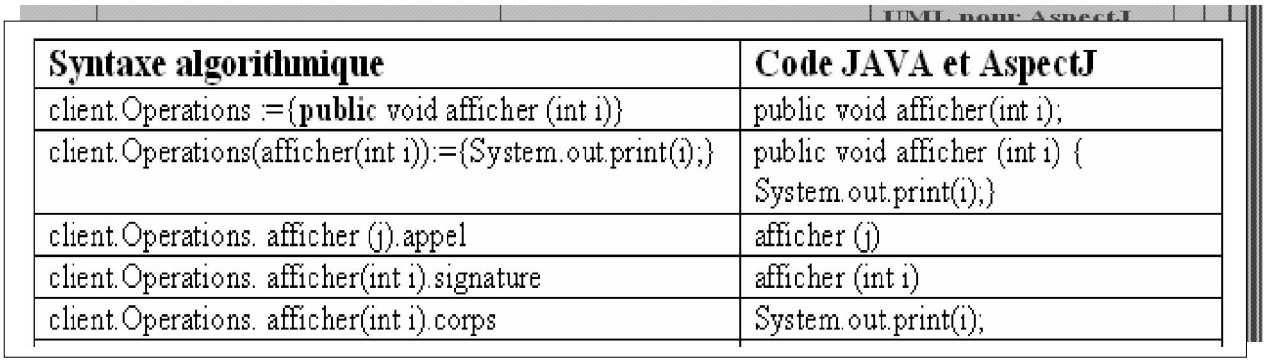


Figure 3.3. Syntaxe algorithmique spécifique pour le niveau conceptuel

b) Syntaxe spécifique pour le niveau physique (JAVA et AspectJ) :

Nous représentons dans ce point la syntaxe algorithmique spécifique au niveau physique, comme par exemple l'implémentation des méthodes (voir figure 3.4) :



Syntaxe algorithmique	Code JAVA et AspectJ
client.Operations :={**public** void afficher (int i)}	public void afficher(int i);
client.Operations(afficher(int i)):={System.out.print(i);}	public void afficher (int i) { System.out.print(i);}
client.Operations. afficher (j).appel	afficher (j)
client.Operations. afficher(int i).signature	afficher (int i)
client.Operations. afficher(int i).corps	System.out.print(i);

Figure 3.4. Syntaxe algorithmique spécifique pour le niveau physique

Exemple d'un extrait d'un algorithme de transformation du design pattern 'Adapter', écrit à l'aide de cette syntaxe. Cet algorithme vise les deux modèles en même temps, à savoir le conceptuel et le physique. Dans ce cas les classes A,B,C et D sont respectivement les classes Target, Adapter, Client et Adapte, du pattern 'Adapter'. (voir figure 3.5).

Algorithme Transformation_Adapter (A,B,C,D)

Debut
……
Créer classe target {
Nom_de_classe :=A.nom_de_interface ;
Type_de_classe :=Abstraite ;
Heritage := « » ;

Operations :=A.Operations ;
}
…….
Créer Aspect adapter_aspect {
Nom_de_aspect :=B.nom_de_classe ;
Type_de_aspect :=Concret ;
Heritage := « » ;
Declare_parents :={D.nom_de_classe implements A.nom_de_interface} ;
Declare_introd_operation :={D.nom_de_classe'.'B.Operations(Requete).signatu
re }
}
Fin

Figure 3.5. Extrait d'un algorithme de transformation du design pattern 'Adapter'.

4. Génération du code aspect :

Comme nous l'avons vu précédemment, le modèle de transformation est décrit d'une manière abstraite. Nous allons appliquer le principe de transformation de ce modèle dans les trois situations suivantes :

- Une instance du pattern au niveau conceptuel qui présente le modèle entier ;
- Une instance du pattern au niveau conceptuel qui présente un modèle partiel d'un modèle d'une application.
- Une instance du pattern au niveau physique, qui est représentée par des programmes.

Dans ce qui suit nous discutons des deux premières situations qui vise la génération du code (une instance comme un modèle entier et une instance comme un modèle partiel). Dans la section 6, nous discutons de la dernière situation qui vise la rétro-ingénierie.

4.1 Principe de génération du code aspect :

Dans la génération du code aspect, nous allons :

- Raffiner le diagramme de classes UML.
- Générer le code Java correspond au diagramme de classe UML raffiné.

- Transformer le code Java généré (OO) en un code AspectJ (OA).

Nous pouvons classifier la transformation en deux catégories. Une catégorie qui est purement orientée objet, du niveau conceptuel vers le niveau physique. La deuxième est de l'orienté objet vers l'orienté aspect, mais dans le même niveau physique. Nous appliquons cette transformation pour toutes les instances de chaque pattern.

4.1.1 Transformation d'un modèle conceptuel orienté objet vers un modèle physique orienté objet :

Cette transformation est purement orientée objet, car elle traduit un diagramme de classe UML vers des programmes orientés objet (Java). La raison de cette transformation est que la programmation orientée aspect complémente la programmation orientée objet avec l'apport des solutions aux challenges que sont l'implémentation de fonctionnalité transversale et le phénomène de dispersion du code. De plus, en programmation orientée aspect, une application est composée de classes et d'aspects. Il devient donc nécessaire d'avoir en premier lieu des programmes orientés objet [Pawlak et al., 2004].

a) Raffinement du diagramme de classes de UML :

Pour le raffinement il s'agit de transformer un diagramme de classes UML vers un autre diagramme UML. Mais cette fois, il faut considérer les détails de l'implémentation (c'est-à-dire les attributs de visibilité « public, private, etc », et transformer les classes abstraites qui ne contiennent pas une méthode implémentée à des interfaces….etc). Nous avons créé cette étape, pour lier le diagramme de classes UML aux détails de l'implémentation.

b) Génération de code JAVA à partir du diagramme de classe UML :

Dans, la POA, une application est composée de classes et aspect. Il sera donc nécessaire de générer d'abord les classes (classes, interfaces). Pour cela nous nous basons sur la génération objet (Java) de [Muller et al., 2002].

4.1.2 Transformation d'un modèle physique orienté objet vers un modèle physique orienté aspect :

Cette transformation est un peu spécifique, c'est-à-dire que chaque pattern a sa logique de transformation [Hannemann et Kiczales, 2002]. Nous avons défini des règles de transformation à travers des algorithme, qui a, en entrée les participants (les classes en JAVA) et elle génère ou bien elle transforme ces classes vers des classes et des aspects en orienté aspect (AspectJ).

Nous avons déjà mentionné que dans le cas de génération du code aspect, nous pouvons avoir deux situations. Une instance du pattern au niveau conceptuel qui présente le modèle entier, et une instance du pattern au niveau conceptuel qui présente un modèle partiel d'un modèle d'une application.

4.2 Instance comme un modèle complet :

Dans cette situation les instances des patterns représentent le modèle entier.

Pour cela il s'agit d'appliquer directement le principe de génération du code. Pour cela nous avons développé un outil qui reflète cette idée, c'est-à-dire un générateur orienté aspect des design patterns. (voir chapitre 4).

Le principe de cette transformation, est qu'à partir d'une instance d'un pattern au niveau orienté objet, nous générons un code aspect correspondant à cette instance (au lieu de générer un code JAVA correspondant à cette instance, puis transformer ce dernier vers un code aspect en AspectJ. Il suffit de générer le code en AspectJ directement, puisque nous connaissons la manière de générer le code JAVA et la manière de transformer cette instance). (voir figure 3.6).

Pour cela lors de la transformation des instances des patterns nous notons deux aspects : aspect statique et aspect dynamique de la transformation [Berkane et al., 2008].

Aspect statique : Ce sont les participants (classes ou aspects), les liens ou bien les opérations qui ne changent pas d'une instance à une autre du même pattern lors de la transformation. Par exemple lors de la transformation du pattern 'Observer', toujours un protocole du pattern 'Observer' (aspect abstrait) sera toujours utilisé pour toute transformation du pattern 'Observer'.

Aspect dynamique : Ce sont les entités qui peuvent changer d'une instance à une autre du même pattern lors de la transformation comme le nom des participants, ou les opérations reliées au contexte de réalisation. Par exemple les noms des participants changent d'un contexte de réalisation à un autre.

Pour cela deux bases ont été définies :

- Une Base de Procédures de transformation : Chaque pattern (sauf pour les patterns qui sont exclus de la transformation) possède son algorithme de transformation qui sert à transformer ces instances. Nous considérons ici l'aspect statique de la transformation qui est enregistrée dans cette base de programmes.

- Une Base de Protocoles : Pour les patterns qui contiennent des aspects abstraits (pour la solution de [Hannemann et Kiczales, 2002]). La base de Protocoles contient les aspects abstraitd des patterns suivants : Chaine of Responsability, Command, Composite, Flyweight, Mediator, Memento, Observer, Prototype, Proxy, Strategy, Visitor.

Le chapitre 4 présente l'outil qui reflète cette idée à savoir le générateur orienté aspect des design patterns.

4.3 Instance comme un modèle partiel :

Dans cette situation les instances des patterns représentent un modèle partiel d'un modèle d'une application.
Dans ce cas, la génération de code aspect, est constituée de cinq étapes principales. La première étape consiste à concevoir un système par un outil CASE qui supporte la norme XMI. La deuxième étape consiste à générer le document XMI [Grose et al. 2002] à partir de l'outil CASE. La troisième étape consiste à détecter des instances des patterns à partir du document XMI. Ces instances appelées des instances candidats attendent une confirmation du concepteur. Cette confirmation constitue la quatrième étape. La dernière étape consiste à générer du code aspect des instances des patterns confirmé par le concepteur et de l'objet pour le reste de l'application (le code généré constitue le squelette du code).

Les différentes étapes de la génération sont numérotées par 1,2,3,4 et 5 dans la figure 3.7.

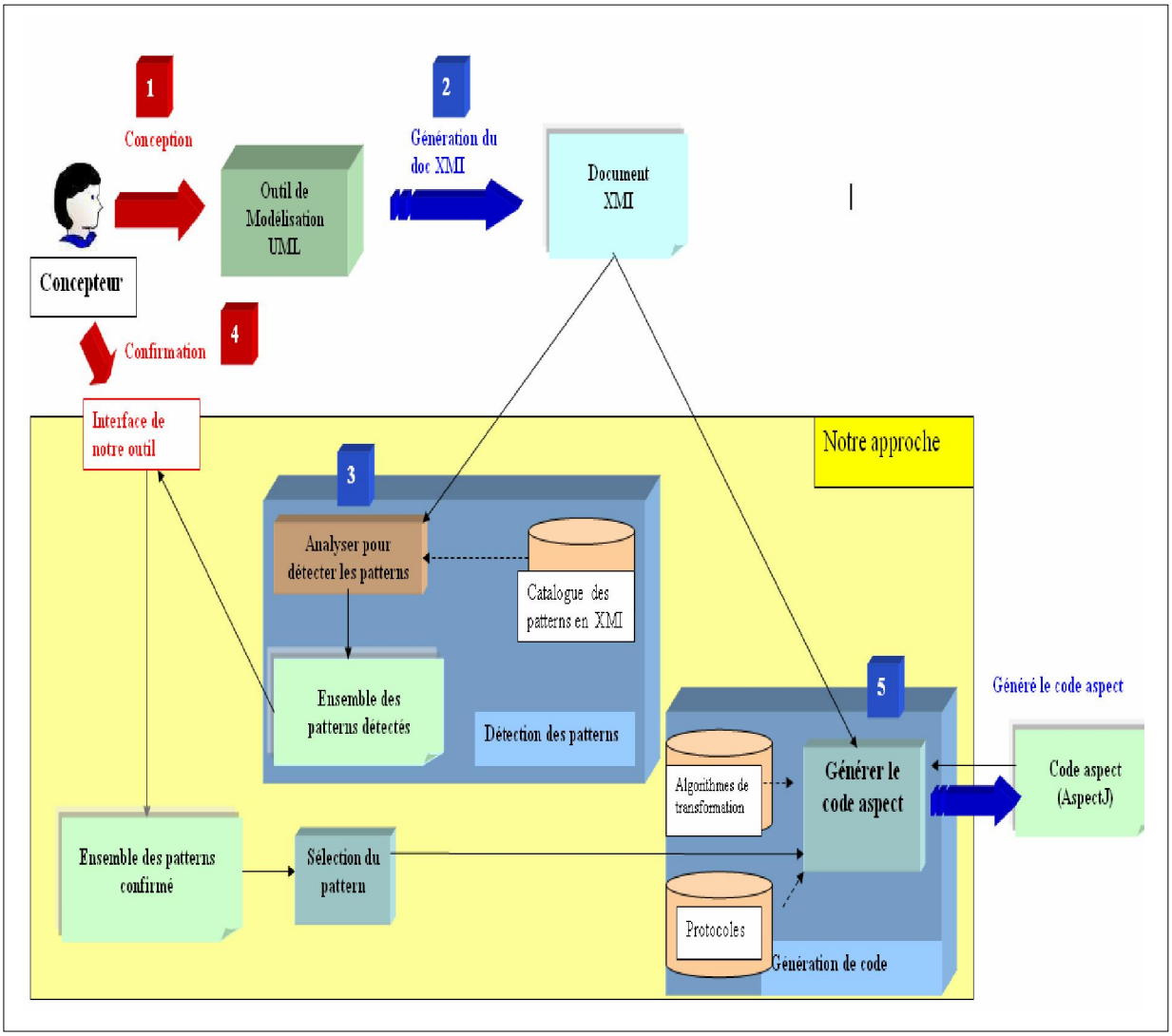


Figure 3.7. Principe de la génération de code aspect. (Cas d'une instance comme un modèle partiel).

4.3.1 Conception avec un outil CASE et la génération du document XMI :

Dans cette étape le concepteur du système, conscrit son système, avec son outil CASE qui supporte la génération du document XMI. En général la plupart des outils CASE les plus connus peuvent faciliter la tâche de la conception et donne l'utilisation des design patterns.

Pour cela, nous considérons deux types de diagrammes dans le modèle conceptuel, à savoir le diagramme de classes UML et le diagramme de séquences UML. Le premier représente une structure statique et le deuxième représente un comportement dynamique.

Quand le concepteur termine la conception, il peut générer le document XMI. La partie qui nous intéresse dans ce document XMI est la partie UML:Model. Cette dernière contient le modèle conceptuel (représenté par le diagramme de classes et séquences). (voir figure 3.8)

<UML:Class …..……>…
<UML:Association …> …
<UML:Generalization …> …
<UML:ClassifierRole……. >…
<UML:Message……>…

Figure 3.8. Extrait d'un document XMI.

4.3.2 Analyse du document XMI pour détecter d'éventuels patterns :

Pour cela nous allons détecter les patterns selon deux types d'analyse, un pour détecter une structure statique et l'autre pour détecter un comportement dynamique.

- Analyse structurelle : Cette phase détecte uniquement les relations entre les classes, en général ces relations sont la généralisation, la réalisation, l'association et l'aggrégation.

- Analyse comportementale : Cette phase détecte l'aspect dynamique des instances des patterns. Ce comportement ce résume par la relation entre les objets de l'instance des patterns. Cette analyse est faite en combinaison avec l'analyse structurelle, parce qu'il y a beaucoup des patterns qui ont des structures similaires, mais le comportement les distingue [Shi et al., 2006].

a) Nouvelle classification des patterns :

Avant de détecter les instances des patterns, nous avons classifié de nouveau les patterns selon l'analyse structurelle et analyse comportementale. De plus, nous avons créé une autre classe des patterns, qui considère les patterns exclus de la détection. Les patterns qui sont exclus de la détection sont ceux qui sont exclus de la transformation.

En général, il y a trois types de classe :

- Les patterns qui sont exclus de la détection.
- Les patterns qui peuvent être détectés selon l'analyse structurelle.
- Les patterns qui peuvent être détectés selon l'analyse structurelle plus l'analyse comportementale.

Les patterns qui sont exclus de la détection sont : Facade, Singleton, Template Methode, Bridge, Abstract Factory, Factory Methode, Builder, Stat, Interpreteur.

Les patterns qui peuvent être détectés par l'analyse structurelle uniquement sont:
Adapter, Composite, Iterator, Prototype, Proxy, Visitor.

Les patterns qui peuvent être détectés par l'analyse structurelle et l'analyse comportementale (dans ce cas l'analyse structurelle est insuffisante) :
Chain of Responsibility, Command, Decorator, Flyweight, Mediator, Memento, Observer, Strategy.

La structure statique et le comportement dynamique des instances des patterns ne changent pas, parce que ces propriétés sont liées à la nature des patterns, mais non à la manière de les présenter ou de les détecter.

Le tableau 3.3 résume la nouvelle classification.

Classes	Patterns
Patterns exclus de la détection	Facade, Singleton, Template Methode, Bridge, Abstract Factory, Factory Methode, Builder, Stat, Interpreteur.
Patterns peuvent être détectés par l'analyse structurelle	Adapter, Composite, Iterator, Prototype, Proxy, Visitor.
Patterns peuvent être détectés par l'analyse structurelle et comportementale	Chain of Responsibility, Command, Decorator, Flyweight, Mediator, Memento, Observer, Strategy.

Tableau 3.3 : Nouvelle classification des patterns (selon la détection).

b) Structure et Sémantique de la syntaxe de détection :

Un extrait de l'algorithme de détection d'une instance d'un pattern 'Command'. Dans notre cas les classes A,B,C,D et E sont respectivement les classes ConreteCommand, Command, Receiver, Invoker et Client.

Algorithme de détection d'une instance du pattern 'Command' :
ChercherClasse(A,B,C,D,E)

Set(A) ; //Pour dire que A n'est pas une seule classe, mais un ensemble de classes
…
Si isClass(A) et isAbstractClass(B) et isClass(C) et isClass(D) et isClass(E)
et isExtends(A,B) et….et contOperation(all(E))
et …et (exist(Object(A)) call exist(Object(C)))
alors Return (PatCommand(A,B,C,D,E))

Où :

- **isAbstractClass(A)** : si A est une classe abstraite retourne true sinon false.
- **isClass(B)** : si B est une classe concrète retourne true sinon false.
- **isExtends(B,A)** : si B étends A alors true sinon false.
- **isDireAssociation(C,A)** : s'il y a une association directe entre C etA.
- **contOperation (A)** : si A contient une méthode alors true sinon false.
- **isClassName(A,Visitor)** : si le nom de la classe A est Visitor retourne true sinon false.
- **contOperationName(A, AddChild)** : si A contient une méthode du nom AddChild retourne true sinon false.
- **Object(nom d'une classe) :** pour designer un objet ou un ensemble d'objets. En général, nous utilisons les opérateurs de quantification, pour exprimer les différents objets, par exemple **exist** ou **all** pour spécifier respectivement qu'il existe des objets ou tous les objets d'une classe donnée. Ces mots ensemblistes utilisés dans l'algorithme peuvent être utilisés pour exprimer même les classes, les méthodes,…etc.
- Objets_appelants **call** Objets_appelés : Pour les appels entre les objets, nous utilisons le mot **call**, qui signifié qu'un objet ou un ensemble d'objets appelle un autre objet ou un ensemble d'objets à travers une méthode.

La sémantique de certains mots-clés utilisés dans cet algorithme est résumée dans le tableau 3.4 :

Mots-clé	Analyse dans la partie
isAbstractClasse(A)	<UML:Class ……isAbstract="**true**" ……>
isExtends(B,A)	<UML:Generalization …….child="le ID de B " parent="le ID de A " />
isDireAssociation(C,A)	<UML:Association ……> <UML:Association.connection> <UML:AssociationEnd …… type="**le ID de A**"/> <UML:AssociationEnd ……isNavigable="**true**" ……type="**le ID de C**"/> </UML:Association.connection> </UML:Association>
Object(nom d'une classe)	<UML:ClassifierRole xmi.id="**l'ID de l'objet**" name="**le nom de l'objet**"…… base="**l'ID de la classe de cet objet**" message2="**le message 2**" message1="**le message 1**" ……>
Objets_appelants **call** Objets_appelés	<UML:Message xmi.id="**l'ID de ce message**" name="**nom de la méthode**"…… sender="**l'ID de l'objet appelant**" receiver="**l'ID de l'objet appelé** " ……>

Tableau 3.4 Sémantique des mots-clé

Retourne (PatCommand(A,B,C,D)) : Retourne les noms des classes ainsi les méthodes (et des attributs s'il existe).

Pour **ChercherClasse** : s'effectue dans le corps de UML:Class : cette technique de recherche peut être implémentée de plusieurs manières, par exemple pour ChercherClasse(A,B,C) : nous cherchons par ordre d'apparition dans le document XMI, puis la recherche se fait comme le parcours des éléments d'une matrice.

4.3.3 Confirmation des patterns détectés :

Après détection des instances patterns candidats, suit l'étape de la confirmation de ces instances. Cette étape est très importante car il est possible de détecter des fausses instances, et puisque le concepteur connaît très bien son application, il peut valider facilement ces instances. De plus cette étape peut aider le concepteur encore une fois, dans le cas où ce dernier utilise une structure semblable à un pattern sans faire attention. Ce genre de cas est une conséquence de cette étape.

4.3.4 Sélection du pattern et génération du code aspect :

Après détection et confirmation des patterns, le développeur passe à l'étape de la génération de code. Dans cette étape la génération de code sera basée sur la notion aspect pour les patterns détectés et confirmés, et sur celle d'objet pour le reste de l'application.

Nous allons maintenir la même idée de génération du code aspect présenté dans la section 5.2, sauf que dans ce cas l'instance est représentée en document XMI. Avec les mêmes aspect de transformation (statique et dynamique), ainsi les deux bases (Base de procédures et Base de Protcoles).

Quand la génération de code aspect sera finie, il reste à générer le code objet pour le reste de l'application (à partir du document XMI).

L'algorithme suivant résume les différentes étapes :

AlgoGénéral
Entrer : Doc_XMI ;
Sortie : CodGénér ;
Debut
EnsInstDetct=Detection_instances_patterns(Doc_XMI) ;
EnsInstConf=Confirmation_des_instances(EnsInstDetct) ;
CodGénér=Génération_code_aspect_instances_patterns(EnsInstConf, Doc_XMI,CodGénér) ;
CodGénér=Génération_code_objet_instances_patterns (EnsInstConf, Doc_XMI,CodeGénér) ;
Fin

Où :
Doc_XMI : Designe le document XMI.
CodGénér : Designer le code généré.
EnsInstDetct : Définit l'ensemble des instances détectées.
EnsInstConf : Définit l'ensemble des instances confirmées.

L'algorithme de détection des instances des patterns:

Detection_instances_patterns(Doc_XMI)
Entrer : Doc_XMI ;
Sortie : EnsInstDetct

Debut
EnsInstDetct = EnsInstDetct U Detection_instances_adapter (Doc_XMI);
……
//tout les patterns qui ne sont pas exclus de la détection
……
EnsInstDetct = EnsInstDetct U Detection_instances_command(Doc_XMI);
Fin

La procédure **Confirmation_des_instances**, est destinée au concepteur. La procédure **Génération_code_aspect_instances_patterns,** a pour but de générer le code aspect des instances des patterns valider. La procédure **Génération_code_objet_instances_patterns**, a pour but de générer le code objet pour le reste de l'application.

5. Rétro-ingénierie des instances basées aspects :

Dans cette section, nous présentons notre approche pour extraire la logique métier à partir d'un code Java. Toute en basant sur l'implémentation des design patterns à l'aide de l'approche par aspect. Cette rétro-ingénierie est l'inverse de l'étape représenté ci-dessus. Elle est constituée de quatre étapes principales. La première étape consiste à faire une analyse du code source d'une application afin d'obtenir un arbre syntaxique abstrait (AST pour Abstract Syntax Tree). Cet arbre contient en fait les différentes classes de l'application. La deuxième étape consiste à détecter les différentes instances des patterns à partir de AST. La troisième étape consiste à transformer les instances détectées. La dernière étape consiste à générer du document XMI (XML Metadata Interchange) La représentation des aspects dans ce cas est faite selon la représentation de Suzuki. (par stéréotypage des aspect) [Berkane et al., 2008a].

Les différentes étapes de la rétro-ingénierie sont numérotées par 1,2,3 et 4 dans la figure 3.9.

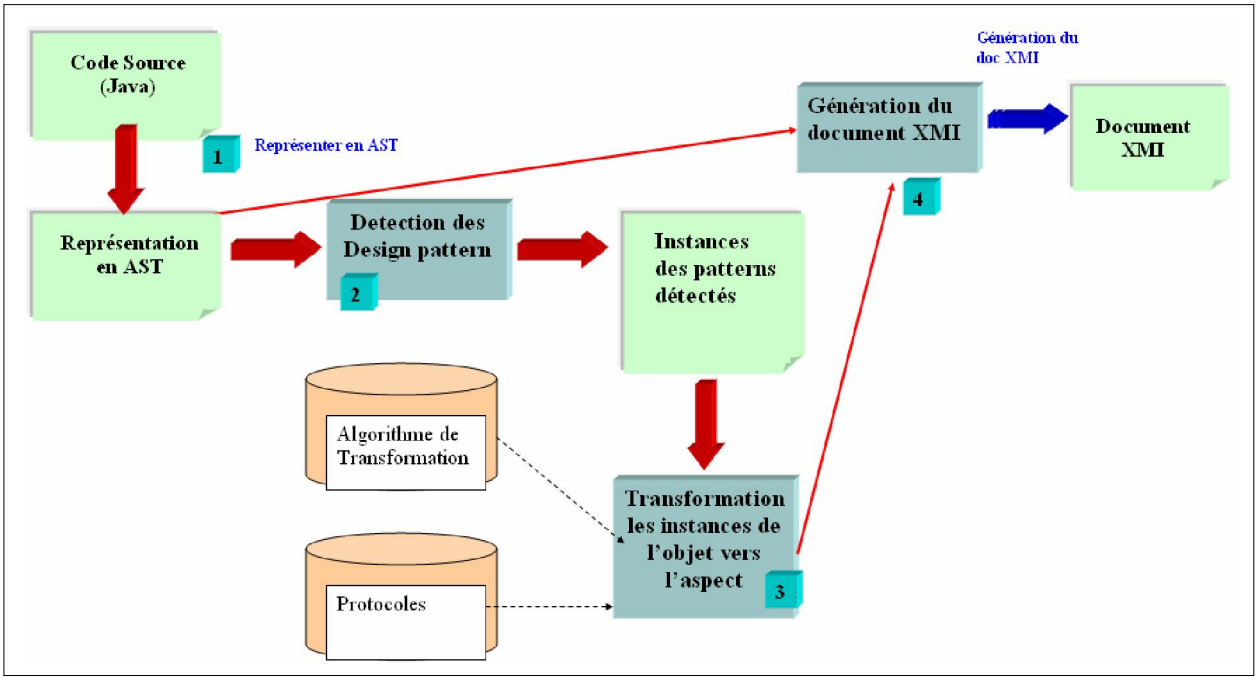


Figure 3.9. Principe de la rétro-ingénierie.

La première étape est basée sur des principes standard.

La deuxième étape consiste à détecter les instances des patterns. Comme nous avons vu dans le chapitre 2 qu'il existe plusieurs techniques de détection des instances des patterns. Dans notre cas nous tenons celle de [Lee et al., 2007], qui détecte plus de patterns par rapport à d'autres techniques.

La troisième étape consiste à transformer les instances détectées de l'orientées objets vers l'orientées aspect. Le principe de transformation est le même que les cas précédents.

La dernière étape consiste à générer le document XMI. La représentation des aspects dans ce cas est faite selon la représentation de Suzuki. Comme nous avons vu dans le chapitre 1 que dans cette représentation, une relation de réalisation connectant une classe (ou une interface) à un

aspect signifie que cet aspect entrecroisé (crosscut) cette classe (ou cette interface). Un aspect est représenté par un stéréotype particulier de classe : « aspect ».

La syntaxe qui représente, l'aspect par le stéréotype est :

```
<UML:Stereotype xmi.id="Le ID de stéréotype " name="aspect" extendedElement="L'ID de l'aspect" />
```

La syntaxe XMI, qui représente, la relation de réalisation est la suivante :

```
<UML:Abstraction xmi.id="L'ID de la relation de réalisation " .....client="L'ID de l'aspect" supplier=" L'ID de la classe ou l'interface " />
```

Les coupes et les 'codes advices' sont représentés par des méthodes. Mais à la différence qu'elles utilisent des mots-clé spécifique. **Pointcut** : pour designer les coupes. Et **after, before ou around** pour designer les 'codes advices'.

6. Conclusion :

Dans ce chapitre nous avons exposé notre modèle de transformation qui permet de traduire des instances des patterns de l'objet vers des instances en aspect. Pour cela, nous avons implémenté des règles qui facilitent ce passage, pour permettre la génération de code aspect, ainsi la rétro-ingénierie, pour extraire la logique métier.

Pour motiver notre travail d'un point de vue pratique, le chapitre suivant décrit une application pour la situation où une instance représente le modèle entier (générateur d'une seule instance à la fois), avec une étude de cas. Puis nous décrivons le principe de fonctionnement de l'outil de la deuxième situation où une instance représente un modèle partiel d'une application (générateur de plusieurs instances à la fois). Enfin, la troisième situation sera décrite pour illustrer le principe de la rétro-ingénierie des instances des patterns.

Chapitre 04

Chapitre

VI

Implémentation et étude de cas

1. Introduction :

La représentation théorique de notre modèle consiste à assurer le passage des instances des patterns de l'orientée objet vers l'orientée aspect. Le principe de cette transformation a été exposé dans le chapitre précédent. Nous avons défini un ensemble de règles pour effectuer ce passage. De plus, nous avons présenté les différentes situations de transformation.

Afin de valider le modèle de transformation proposé, nous essayons de valider chaque situation de transformation à part. La première situation est le générateur d'une seule instance à la fois, où le modèle conceptuel représente uniquement une instance d'un pattern à la fois. La deuxième situation est le générateur de plusieurs instances à la fois, où le modèle conceptuel peut contenir plusieurs patterns à la fois. La dernière situation concerne la rétro-ingénierie.

Dans le cas du générateur d'une seule instance à la fois, nous essayons de valider ce cas par un pattern de chaque groupe définit par Hannemann et Kiczales.

La validation du deuxième et troisième cas est faite par une étude de cas.

Nous présenterons également, dans ce chapitre les différents outils que nous avons utilisé pour valider notre modèle.

2 Cadre de validation de l'approche de transformation :

Nous avons présenté dans le chapitre précèdent, notre modèle de transformation. Nous avons aperçu trois situations, pour appliquer le principe de la transformation de notre modèle à savoir : une instance au niveau conceptuel qui représente le modèle entier, une instance représente un modèle partiel d'un modèle d'une application et enfin une instance comme des programmes au niveau physique.

Pour valider le modèle de transformation proposé, nous essayons de valider chaque situation de transformation à part :

- Dans le cas où une instance d'un pattern est aperçue comme un modèle conceptuel entier. Il s'agit d'appliquer directement le principe de transformation du modèle de traduction. Pour cela, nous avons développé un outil qui reflète cette idée, c'est-à-dire un générateur orienté aspect des design patterns pour une seule instance à la fois.

- Pour le cas où une instance d'un pattern représente un modèle partiel d'un modèle d'une application. Nous avons vue dans le chapitre 3, qu'il y a cinq étapes, à savoir la conception d'une application avec un outil CASE, la génération du document XMI, analyse du document XMI pour détecter des éventuels patterns, confirmation des patterns détectés, et enfin sélection du pattern et génération du code aspect. Pour les deux premières étapes (conception et génération du document XMI) seront supporter par un outil CASE supportant la génération du document XMI. Pour le reste des étapes (détection, confirmation et génération du code) seront supportées par notre outil. Pour cela nous allons présenter dans ce cas un générateur orienté aspect des design patterns pour plusieurs instances à la fois.

- Dans le dernier cas, des instances des patterns sont aperçues comme des programmes. Pour cela il s'agit de détecter des instances objet dans le code objet, puis la remplacer par son équivalent en aspect. Nous allons présenter dans ce cas uniquement le principe de fonctionnement de la Rétro-ingénierie.

3 Générateur d'une seule instance à la fois:

Dans cette section, nous présentons l'outil utilisé pour valider la première situation à savoir une instance d'un pattern est aperçue comme un modèle conceptuel entier. Nous essayons de valider ce cas par un pattern de chaque groupe définit par Hannemann et Kiczales.

3.1 Cadre de validation :

Selon [Hannemann et Kiczales, 2002], les patterns sont regroupés en six groupes (voir le chapitre 2, section 5.2.2.d). Pour une bonne validation, nous essayons de prendre un pattern de chaque groupe.
Les patterns appartenant aux groupes : héritage multiple, modularité du code disperser, et aucune bénéfice ne sont pas considérés par la transformation, parce qu'ils sont exclus de la transformation. Les raisons d'écarter ces patterns sont présentées dans le chapitre 3, section 3.2.b.
Pour cela nous allons étudier trois patterns, à savoir des instances des patterns 'Adapter', 'Prototype', et 'Mediator'.

Chaque un de ces patterns appartient à une catégorie de transformations de [Hannemann et Kiczales, 2002] :

- Le pattern ‘Adapter’ pour la catégorie ’’ Constructions du langage’’.
- Le pattern ‘Prototype’ pour la catégorie ’’Aspects comme fabriquants d'objet’’.
- Le pattern ‘Mediator’ pour la catégorie ’’Rôles seulement utilisés dans l'aspect du pattern’’.

Dans ce qui suit, nous allons présenter le modèle conceptuel orienté objet (représenté par le diagramme de classes), ainsi le modèle physique orienté objet (représenté par le langage Java) de chaque un de ces patterns (Mediator, Prototype, Adapter), ensuite, les algorithmes de transformation, puis les résultats obtenus de cette transformation à savoir le modèle conceptuel orienté aspect et le modèle physique orienté aspect, représenté respectivement par le diagramme de classe UML pour AspectJ et le langage AspectJ.
Pour concrétiser ces algorithmes de manière pratique, nous les avons implémenté dans notre outil, ensuite nous avons étudié des cas pour effectuer la transformation.

3.2 Aperçu de l’outil développé :

Nous avons développé un prototype, en utilisant le langage Delphi. Borland Delphi est un environnement de programmation visuelle orienté objet permettant de développer des applications 32 bits en vue de leur déploiement sous Windows et sous Linux. Nous pouvons créer de puissantes applications avec un minimum de programmation en utilisant Delphi 7. (voir figure 4.1).

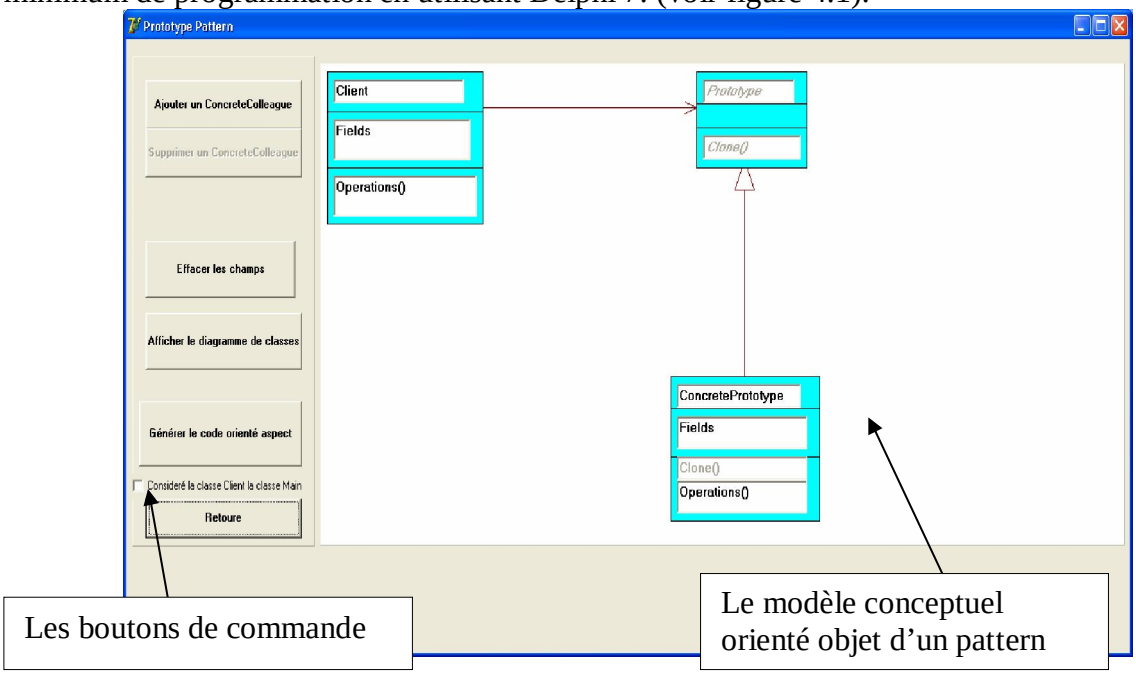


Figure 4.1. Présentation de l’outil du générateur d’une seule instance à la fois.

La figure 4.2 représente les fonctionnalités des boutons de commande.

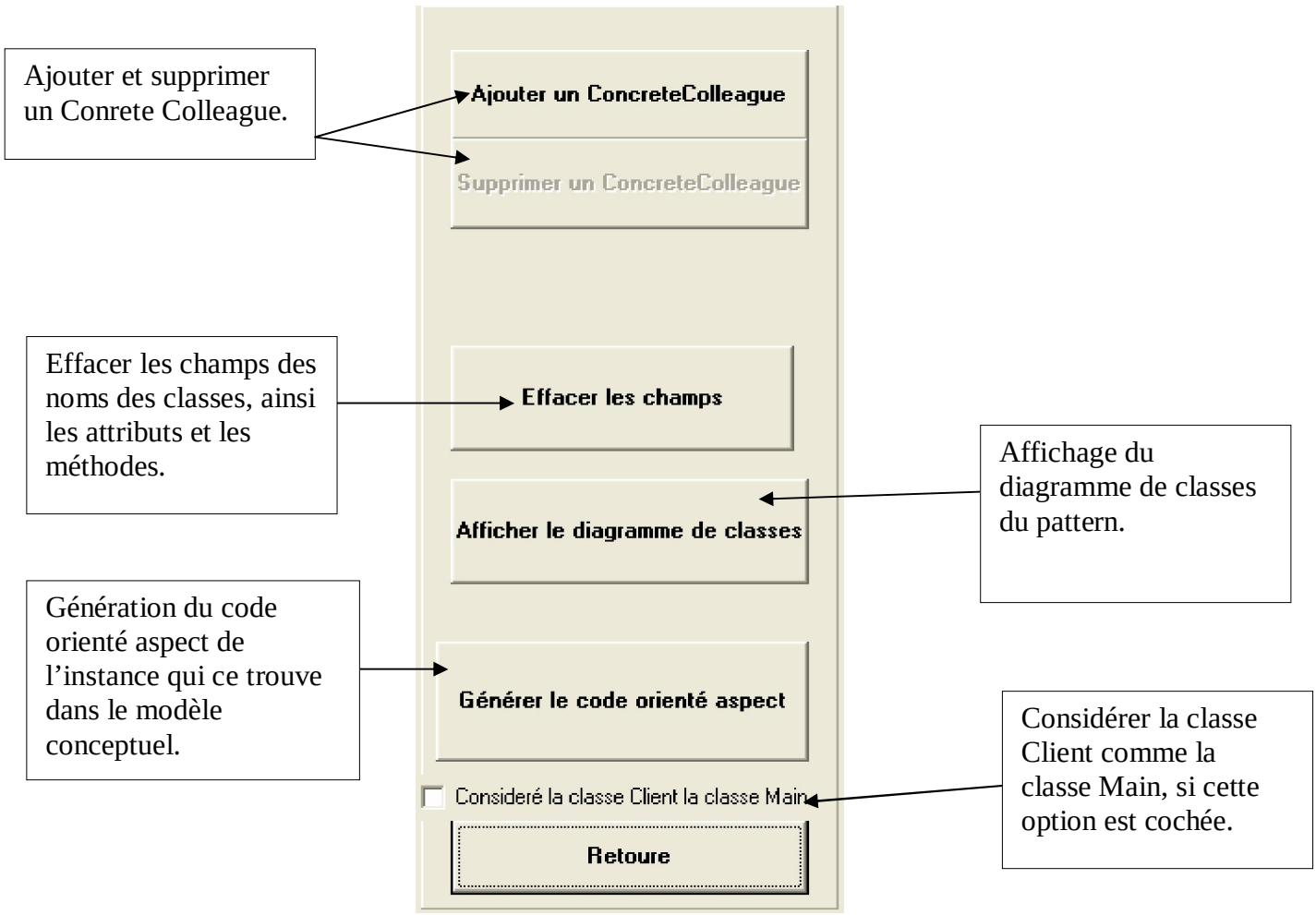


Figure 4.2. Présentation des fonctionnalités des boutons de commande.

Dans le pattern 'Mediator' il est possible d'avoir plusieurs Concrete Colleague. Pour cela, il est possible d'ajouter ou supprimer des Conrete Colleague avec les boutons spécifiques. Dans le cas du pattern 'Prototype', nous avons d'ajouter et supprimer un Conrete Prototype. Le pattern 'Adapte' ne contient pas ce genre de bouton.

L'option « Considéré la classe Client la classe Main » est spécifique au pattern qui contient la classe Client. Si nous cochons cette option, nous allons avoir dans la classe Client la méthode **main**.
Les boutons « Effacer les champs », « Afficher le diagramme de classes », et « Générer le code orienté aspect » sont apparus dans tout les patterns de notre outils.

3.3 Instance du pattern 'Adapter' :

Le pattern 'Adapter' convertit l'interface d'une classe en une interface distincte, conforme à l'attente de l'utilisateur. Adapter permet à des classes de travailler ensemble, qui n'auraient pu le faire autrement pour causes d'interfaces incompatibles.[Gamma et al., 1995].

3.3.1 Présentation du pattern ‘Adapter’ en O.O:

Ce pattern constitue de quatre participants qui sont : Adapter, Adaptee, Target, et, Client.
Où :
Target : Définit l'interface métier utilisée par le Client.
Client : Travaille avec des objets implémentant l'interface Target.
Adaptee : Définit une classe existante devant être adaptée.
Adapter : Fait correspondre la classe Adaptee à l'interface Target.

Nous allons considérer une instance du pattern ‘Adapter’ dans un contexte de réalisation où les participants de cette instance portent les mêmes noms de ceux du pattern lui-même (contexte général). La figure 4.3 représente le modèle conceptuel (représenté par le diagramme de classes) de ce pattern.

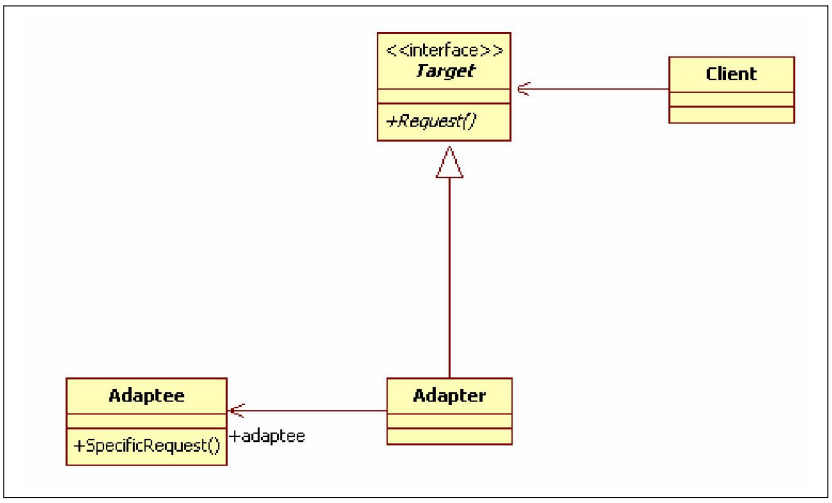


Figure 4.3. Instance du pattern ‘Adapter’ au niveau conceptuel représenté par le diagramme de classes.

Le modèle physique (représenté par le langage Java) d’une instance du pattern ‘Adapter’ dans le contexte général est illustré dans la figure 4.4.

```
// La classe Client
public class Client {
      }

// La classe Adapter implementant l'interface Target
public class Adapter implements Target {
      private Adpatee adaptee;
      public void Request () {adaptee.SpecificRequest();}
}

// L'interface Target
public interface Target {
      public void Request();}

// La classe Adaptee
public class Adaptee {
      public void SpecificRequest(){}
}
```

Figure 4.4. Instance du pattern ‘Adapter’ au niveau physique représenté par le langage Java.

3.3.2 Transformation de ‘Adapter’ :

Le pattern Adapter est directement implémenté grâce aux mécanismes Aspect d’extension d’interfaces [Hannemann et Kiczales, 2002]. La transformation en AspectJ est d’utiliser le concept d’introduction offert par ce langage permet de doter directement la classe impactée Adaptee par une nouvelle interface conforme à celle du Target (interface requise par le classe Client), tout en lui ajoutant les opérations nécessaires (Requeste()). Chacune des opérations nouvellement introduites fait, tout simplement, appel à son opération correspondante spécifique à la classe impactée (RequeteSpecifique())[Hachani 2006].

a) Avantages de la transformation :

L’implémentation du pattern ‘Adapter’ à l’aide de l’approche aspect permet de bénéficier des avantages suivants : la localisation, la composition et l’adaptabilité. [Hannemann et Kiczales, 2002].

Du point de vue de la localisation, il n’y a pas la classe Adapter pour contenir le code gérant l’adaptation. Par ailleurs, l’essentiel de cette gestion est implémenté dans un aspect concret. Ce dernier contient la spécificité de la classe Adapter.

Du point de vue de la composition, la classe Adapter (si elle contient d’autres méthodes que la méthode d’adaptation) peut participer sans difficulté à d’autre instances.

Du point de vue de l’adaptabilité, le lien entre les classes Adapter et les classes Adaptee est beaucoup plus faible que précédemment. Il est possible d’avoir une gestion complexe des Adapter et Adaptee. Par exemple, la gestion de plusieurs Adapter et Adaptee dans un seul aspect.

b) Principe de transformation :

Avant d’entamer l’algorithme de transformation, nous allons décrire les opérations effectuées par ce dernier d’une manière concrète. Il génère un aspect concret qui porte le même nom de la classe Adapter. Cet aspect concret contient la déclaration inter-type entre l’interface Target et la classe Adaptee (modification hiérarchie d’héritage des classes). La raison de la déclaration inter-type pour que le mécanisme d’introduction soit possible (introduire la méthode Request dans la classe Adaptee). Il génère aussi les classes suivante : Client, Target, et Adaptee. Cette génération est purement orientée objet.

Si nous comparons la solution objet par rapport la solution aspect, nous remarquons que la classe Adapter est remplacée par un aspect. (suppression de la classe, puis l’ajout d’un aspect).

c) Algorithme de transformation :

L’algorithme suivant transforme une instance du pattern ‘Adapter’ de l’orienté objet vers l’orienté aspect dans les deux niveaux conceptuel et physique.

Transformation_Adapter (Classe Client, Interface Target, Classe Adapter, Classe Adaptee)

Debut

Créer Classe client {
Nom_de_classe := Client.nom_de_classe ;
Type_de_classe :=Concrete ;
Heritage := « » ;
Attributs :=Client.Attributs ;
Operations :=Client.Operations ;}

Créer Interface target {
Nom_de_interface := Target.nom_de_interface ;
Heritage := « » ;
Operations := « » ;}

Créer Aspect adapter {
Nom_de_aspect :=Adapter.nom_de_classe ;
Type_de_aspect :=Concret ;
Heritage := « » ;
Declare_parents :={Adaptee.nom_de_classe **implements** Target.nom_de_interface}
Declare_introd_operation :=
{Adaptee.nom_de_classe'.'Adapter.operations(Requete).signature}
Declare_introd_operation
(Adaptee.nom_de_classe'.'Adapter.operations(Requete).signature) :=
{Operation(RequeteSpecifique).appel ;}}

Créer Classe adaptee
{
Nom_de_classe := «Adaptee» ;
Type_de_classe :=Concrete ;
Heritage := « » ;
Attributs :={**private** Adaptee.nom_de_classe adaptee}
Operations :=Adaptee.Operations() ;}

Si Modèle=Conceptuel alors
Debut
AfficherNote(Adapter,Adapter) ;
VérifierLien() ;
Fin
Fin

d) Résultat de la transformation :

Apres l'application de l'algorithme de transformation dans le contexte de réalisation générale, nous obtenons le modèle conceptuel (représenté par le diagramme de classes pour AspectJ) illustré dans la figure 4.5.

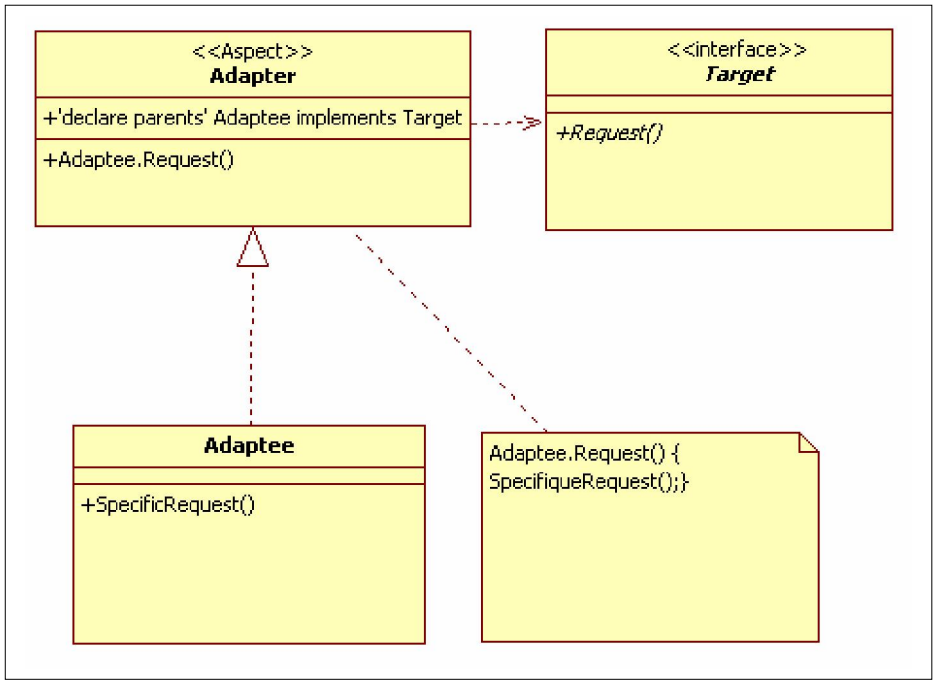


Figure 4.5. Instance du pattern 'Adapter' au niveau conceptuel O.A représenté par le diagramme de classe UML pour AspectJ.

Le modèle physique (représenté par le langage AspectJ) obtenu après la transformation est illustré dans la figure 4.6.

```
//La classe Client
public class Client {

}

// L'aspect Adapter
public aspect Adapter {
      declare parents: Adaptee implements Target ;
public void Adaptee.Request()
{ SpecificRequest();}
}

//L'interface Target
public interface Target {
      public void Request();}

// La classe Adaptee
public class Adaptee {
      public void SpecificRequest(){}
}
```

Figure 4.6. Instance du pattern 'Adapter' au niveau physique O.A représenté par le langage AspectJ.

3.3.3 Etude de cas :

Cette étude de cas est prise de l'exemple du pattern 'Adapter' de [Hannemann et Kiczales, 2002]. Dans ce cas, il y a une adaptation de la classe **Ecran** (est exactement la méthode afficher()), par l'adaptateur la classe **ImprEcran** (à la méthode imprimer()).(voir figure 4.7).

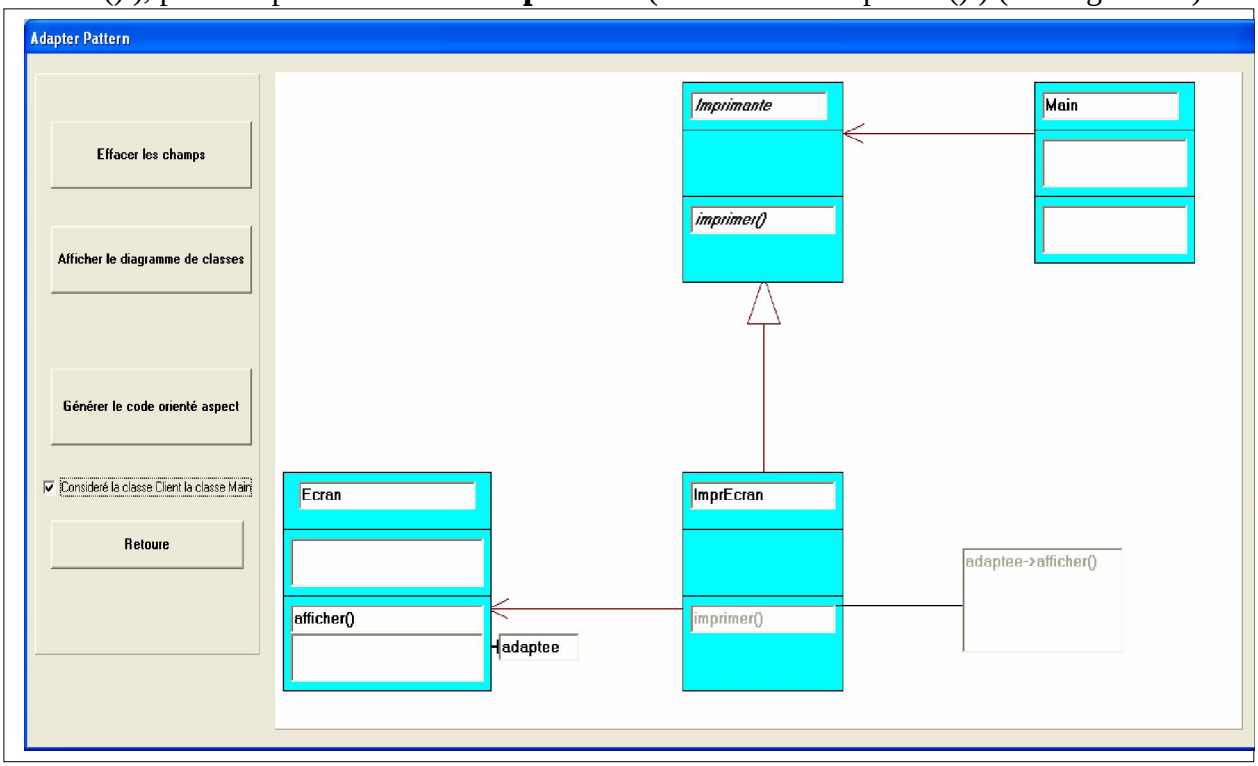


Figure 4.7. Instance du pattern 'Adapter' dans notre outil

La figure 4.8 montre un extrait du code aspect généré pour notre exemple Imprimante/Ecran.

```
//L'aspect ImprEcran
public aspect ImprEcran {
      declare parents: Ecran implements Imprimante ;
public void Ecran.imprimer() { afficher() ;}}


//L'interface Imprimante
public interface Imprimante {  public void imprimer() ; }
```

Figure 4.8. Extrait du code généré de l'exemple étudié de 'Adapter'.

Le code obtenu par le générateur n'est pas prêt à être exécuté (c'est un squelette de code). Il faut le compléter, c'est-à-dire il faut implémenter les différentes méthodes. L'annexe B présente le reste du code généré par l'outil, ainsi le code final (après une modification).

Puis le programme qui est obtenu par des modifications sera prêt à être exécuté. Il est déjà testé avec succès dans l'environnement Eclipse version 3.3.0 avec Eclipse AspectJ Development Tools version 1.5.0. Nous avons supposé que la méthode **imprimer** affiche un message à l'écran. (voir figure 4.9).

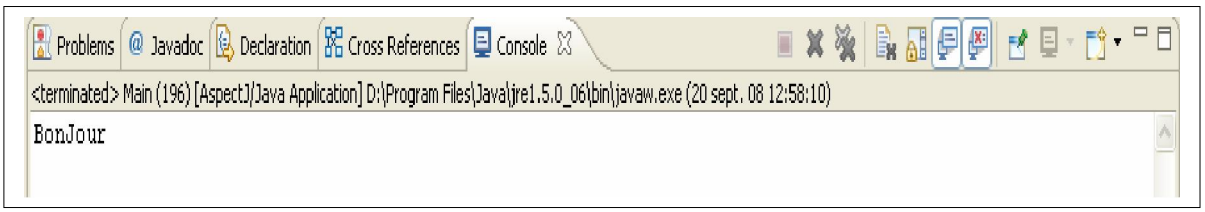


Figure 4.9. Résultat de l'exécution de l'exemple étudié de 'Adapter'

3.4 Instance du pattern 'Prototype' :

Le pattern 'Prototype' spécifie les espèces d'objets à créer, en utilisant une instance de type prototype, et crée de nouveaux objets par copies de ce prototype. Ce pattern constitue de trois participants : Prototype, Concrete Prototype, et Client. [Gamma et al., 1995].
L'interface **Prototype** sert de modèle principal pour la création de nouvelles copies. La classe **Concrete Prototype** vient spécialiser la classe Prototype en venant par exemple modifier certains attributs. La méthode **clone()** doit retourner une copie de l'objet concerné. La dernière classe (la classe **Client**) va se charger d'appeler les méthodes de clonage via sa méthode operation().Le modèle conceptuel et le modèle physique d'une instance de ce pattern dans le contexte général seront présentés dans l'annexe C.

3.4.1 Transformation de 'Prototype' :

Le pattern Prototype peut être implémenté en aspect par la création de deux aspects : un aspect abstrait et un aspect concret. La partie abstraite définit l'opération de clonage (gestion des copies). La partie concrète spécifiée les classes qui peuvent jouer le rôle de Concrete Prototype. [Hannemann et Kiczales, 2002].

a) Avantage de la transformation :

Si nous l'analysons l'implémentation aspect du pattern 'Prototype' selon les quatre critères définis par Hannemann et Kiczales [Hannemann et Kiczales, 2002], nous pouvons dresser les constats suivants :

Du point de vue de la localisation, la classe qui joue le rôle de Concrete Prototype ne contient plus de code gérant le clonage. L'essentiel de cette dernière est centralisé dans l'aspect abstrait **PrototypeProtocol**. Les spécificités des Concrete Prototype sont prises en compte dans les aspects concrets dérivés de **PrototypeProtocol.**

Du point de vue de la réutilisation, la situation est nettement améliorée par rapport à la solution orientée objet. L'essentiel de la gestion des copies de prototype est défini de manière générique dans l'aspect abstrait et est réutilisé systématique pour chaque Concrete Prototype. Il suffit de définir un aspect concret capturant les éléments spécifiques du contexte d'application du pattern Prototype (identifié les classes qui jouent le rôle de Prototype).

Du point de vue de la composition, les classes jouant le rôle de Concrete Prototype peuvent participer sans difficulté à d'autres design patterns car les aspects développés ici ne sont pas intrusifs, le comportement du Concrete Prototype n'étant pas modifié.

Du point de vue de l'adaptabilité, la situation est inchangée par rapport à l'implémentation orientée objet puisque il y a un seul rôle (Prototype).

b) Principe de transformation :

Les opérations importantes effectuées par l'algorithme de transformation sont : la création d'un aspect abstrait (PrototypeProtocol), la création d'un aspect concret, et la suppressions des méthodes de gestion de copie dans les classes jouant le rôle de Concrete Prototype.

L'aspect abstrait (**PrototypeProtocol**) déclare une interface interne (Prototype) et une introduction dans cette interface de l'opération **clone()** . Cet aspect contient deux autres méthodes à savoir: **cloneObject** et **createCloneFor** afin de gérer l'exception.

L'aspect concret spécifie les classes qui jouent le rôle de Concrete Prototype. Cette spécification est faite à l'aide de la déclaration inter-type (modification hiérarchie d'héritage des classes) entre l'interface Prototype (du protocole) et la classe jouant le rôle de Concrete Prototype.

c) Algorithme de transformation :

L'algorithme suivant transforme une instance du pattern 'Prototype' de l'orienté objet vers l'orienté aspect dans les deux niveaux conceptuel et physique.

Transformation_Prototype (Classe Client, ? Interface Prototype , Classe ConcretePrototype1, ConcretePrototype2,….,ConcretePrototypeN)

Debut

Créer Classe client {
Nom_de_classe := Client.nom_de_classe ;
Type_de_classe :=Concrete ;
Heritage := « » ;
Attributs :=Client.Attributs ;
Operations :=Client.Operations ;
}

Créer_Protocols(Prototypes) ;

Créer Aspect Prototype
{
Nom_de_aspect :=Prototype+'_'+ConcretePrototype.nom_de_classe;
Type_de_aspect :=Concret ;
Heritage := « **extends** PrototypeProtocol» ;
Declare_parents :={ConcretePrototype1.nom_de_classe **implements '**Prototype',
ConcretePrototype2.nom_de_classe **implements '**Prototype',
…….,
ConcretePrototypeN.nom_de_classe **implements '**Prototype'}
Operations := {**protected** Object creatCloneFor(Prototype object)} ;
Operations(creatCloneFor(Prototype object)**)** :={return null} ;
}

Créer Classe concreteprototype1 {
Nom_de_classe := ConcretePrototype1.nom_de_classe ;
Type_de_classe :=Concrete ;
Heritage := « **implements** Cloneable» ;
Attributs := ConcretePrototype.Attributs ;
Operations :=ConcretePrototype1.Operations/{public * clone()} ;
}

Créer Classe concreteprototype2 {
Nom_de_classe := ConcretePrototype2.nom_de_classe ;
Type_de_classe :=Concrete ;
Heritage := « **implements** Cloneable» ;
Operations :=ConcretePrototype2.Operations/{public * clone()} ;
}
…….
….…….
Créer Classe concreteprototypeN {
Nom_de_classe := ConcretePrototypeN.nom_de_classe ;
Type_de_classe :=Concrete ;
Heritage := « **implements** Cloneable» ;
Operations :=ConcretePrototypeN.Operations/{public * clone()} ;
}

Si Modèle=Conceptuel alors
Debut
AfficherNote(PrototypeProtocol, PrototypeProtocol) ;
VérifierLien() ;
Fin
Fin

Les résultats de l'application de l'algorithme de transformation dans le contexte de réalisation seront présentés dans l'annexe C.

3.4.2 Etude de cas :

Cette étude de cas est prise de l'exemple du pattern 'Prototype' de [Hannemann et Kiczales, 2002]. Dans cet exemple, nous supposons qu'il y a une classe qui a pour nom **Chaine**, à pour but la gestion des chaînes de caractère. Nous voulons dans la classe **Main** gérer des objets de la classe **Chaine.** Pour cela il est possible de déclarer la classe **Chaine** comme une classe Prototype, puis de gérer ces différents objets dans la classe **Main** comme des clones (au lieu de créer à nouveau un objet), surtout si nous voulons garder les mêmes caractéristiques. (voir figure 4.10).

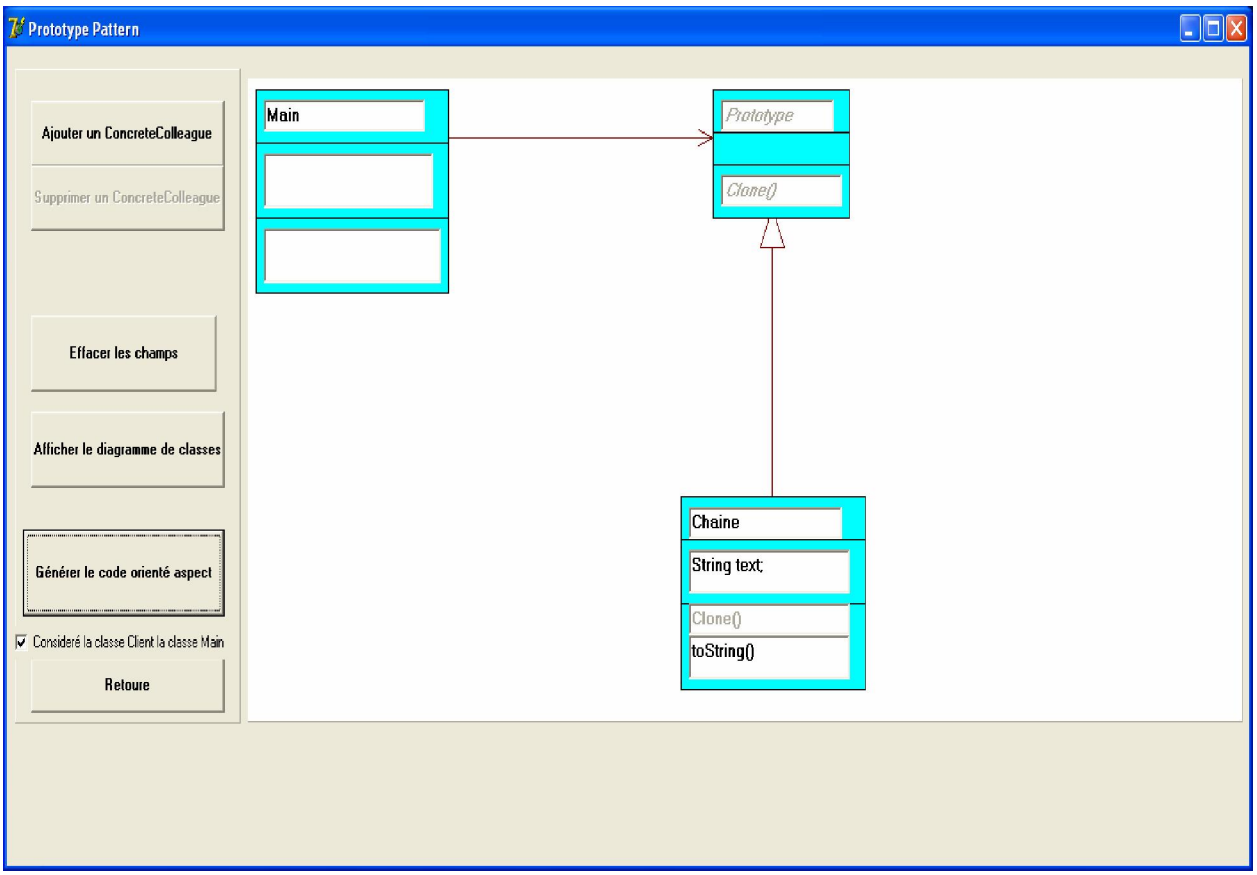


Figure 4.10. Instance du pattern 'Prototype' dans notre outil

L'annexe C présente le code généré (squelette de code) à partir de notre outil, ainsi le code final après l'implémentation des méthodes. Nous allons supposer que nous allons gérer un objet **orig** qui affiche à l'écran le message (« C'est le prototype 1 »). Puis un clone **copy** de cet objet qui affiche le même message. (voir figure 4.11).

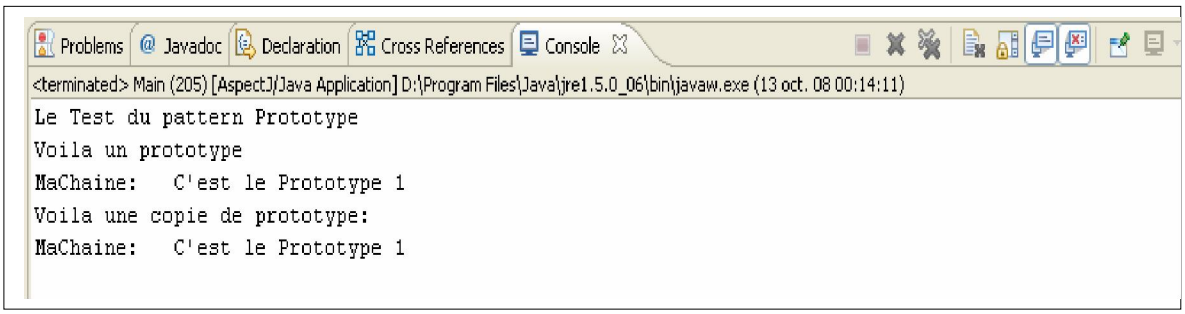


Figure 4.11. Résultat de l'exécution de l'exemple de 'Prototype'

3.5 Instance du pattern 'Mediator' :

Le pattern 'Mediator' définit un objet qui encapsule les modalités d'interaction de divers objets. Le médiateur favorise les couplages faibles, en dispensant les objets d'avoir à faire référence explicite les uns aux autres ; de plus, il permet de modifier une relation indépendamment des autres. Ce pattern constitue de quatre participants : Mediator, Colleague, Concrete Mediator, et Concrete Colleague. [Gamma et al., 1995].
L'interface **Mediator** est définie pour la communication entre les objets Colleague. Ainsi, pour l'interface **Colleague** est définie pour la communication avec les objets Mediator. Les classes **Concrete Mediator** implémente l'interface Mediator. Ils coordonnent la communication entre

les objets Colleague. Les classes **Concrete Colleague** communique avec les autres Colleague à travers le Mediator. (gamam).
Le modèle conceptuel et le modèle physique d'une instance de ce pattern dans le contexte général seront présentés dans l'annexe D.

3.5.1 Transformation de 'Mediator' :

Le pattern Mediator définit pour son implémentation aspect, un aspect abstrait pour gérer la relation entre le rôle Mediator et le rôle Colleague, et un aspect concret pour spécifier les classes jouant les rôles de Mediator et de Colleague. Ce dernier définit aussi l'opération de notification. . [Hannemann et Kiczales, 2002].

a) Avantage de la transformation :

L'implémentation du pattern 'Prototype' à l'aide de l'approche aspect permet de bénéficier des avantages suivants : la localisation, la composition, la réutilisation et l'adaptabilité. . [Hannemann et Kiczales, 2002].

Du point de vue de la localisation, les classes jouent les rôles de Mediator et Colleague ne contiennent plus de code gérant la relation entre ces derniers. L'essentiel de cette gestion est centralisé dans l'aspect abstrait **MediatorProtocol.** Les spécificités des Mediators et Colleagues, ainsi la méthode déclenchant la notification au Mediator est prise en compte dans les aspects concrets dérivés de **MediatorProtocol.**

Du point de vue de la réutilisation, la situation est améliorée par rapport à la solution orientée objet. La gestion de la relation entre les rôles Mediator et Colleague est définie de manière générique dans l'aspect abstrait (**MediatorProtocol**) et réutilisé systémiquement pour chaque Mediator et Colleague. Il suffit de définir un aspect concret capturant les éléments spécifiques du contexte d'application du pattern Mediator (identifié les classes qui jouent le rôle de Mediator et Colleague, ainsi la méthode déclenchant la notification au Mediator).

Du point de vue de la composition, les classes jouant les rôle de Mediator et Colleague peuvent participer sans difficulté à d'autres design patterns car les aspects développés ici ne sont pas intrusifs, le comportement du Mediator et Colleague n'étant pas modifié.

Du point de vue de l'adaptabilité, le lien entre les classes Mediator et les classes Colleague est beaucoup plus faible que précédemment. Le moment où la notification est déclenchée est paramétrable dans l'aspect concret grâce à la coupe **change(Colleague c),** ce qui ne permet pas l'implémentation orientée objet.

b) Principe de transformation :

Les opérations importantes effectuées par l'algorithme de transformation sont : la création d'un aspect abstrait (MediatorProtocol), la création d'un aspect concret, la suppressions du code de la gestion du changement de Colleague dans les classes de Concrete Mediator, et la suppression des méthodes de la gestion de l'addition de Mediator dans les classes de Concrete Colleague.

L'aspect abstrait (**MediatorProtocol**) déclare deux interfaces internes (Mediator et Colleague). Il définit une coupe abstraite **change(Colleague c),** et le code Advice. (Liée à cette coupe) pour traiter la notification de Mediator après le changement d'un ou de plusieurs Colleague. Ce protocole définit aussi deux méthodes : **getMediator,** et **setMediator** afin de bien géré les Mediator et Colleague.

L'aspect concret spécifie les classes jouant le rôle de Mediator, ainsi de Colleague. De plus, il concrétise la coupe par la définition exacte des méthodes qui peuvent déclencher la notification au Mediator. La méthode qui traite la notification est définie dans cet aspect concret.

c) Algorithme de transformation :

L'algorithme suivant transforme une instance du pattern 'Mediator' de l'orienté objet vers l'orienté aspect dans les deux niveaux conceptuel et physique. Nous allons supposer que nous pouvons avoir un seul Mediator et plusieurs Colleague

Transformation_Mediator (Interface Mediator, Interface Colleague , Classe ConcreteMediator, ConcreteColleague1,….,ConcreteColleagueN)

Debut

Créer_Protocols(Mediators) ;

Créer Aspect Mediator
{
Nom_de_aspect :=Mediator+'_'+ ConcreteMediator.nom_de_classe
+ConcreteColleague.nom_de_classe ;
Type_de_aspect :=Concret ;
Heritage := « **extends** MediatorProtocol » ;
Decalre_parents :={ConreteMediator.nom_de_classe **implements** 'Mediator',
ConreteColleague1.nom_de_classe **implements** 'Colleague',
……,
ConreteColleagueN.nom_de_classe **implements** 'Colleague'}
Pointcut :={ **protected** change(Colleague c) : (call ((void
ConcreteColleague1.Operations.colleagueChanged().signature) || ….||(void
ConcreteColleagueN.Operations.colleagueChanged().signature))&& target(c))}
Operations := {**protected** void notifyMediator(Colleague c, Mediator m)}
}

Créer Classe concretemediator {
Nom_de_classe := ConcreteMediator.nom_de_classe ;
Type_de_classe :=Concrete ;
Heritage := « » ;
Attributs := ConcreteMediator.Attributs ;
Operations := ConcreteMediator.Operations ;
}

Créer Classe concretecolleague1 {
Nom_de_classe := ConcreteColleague1.nom_de_classe ;
Type_de_classe :=Concrete ;
Heritage := « » ;
Attributs := ConcreteColleague1.Attributs ;

Operations := ConcreteColleague1.Operations/{public void setMediator(Mediator mediator)} ;
}
…….. ;
…… ;
Créer Classe concretecolleagueN {
Nom_de_classe := ConcreteColleagueN.nom_de_classe ;
Type_de_classe :=Concrete ;
Heritage := « » ;
Attributs := ConcreteColleagueN.Attributs ;
Operations := ConcreteColleagueN.Operations/{public void setMediator(Mediator mediator)} ;
}
Si Modèle=Conceptuel alors
Debut
AfficherNote(MediatorProtocol,MediatorProtocol) ;
AfficherNote(Mediator,Mediator) ;
VérifierLien() ;
Fin
Fin

Les résultats de l'application de l'algorithme de transformation dans le contexte de réalisation seront présentés dans l'annexe D.

3.5.2 Etude de cas :

Cette étude cas est prise de l'exemple du pattern 'Mediator' de [Hannemann et Kiczales, 2002]. Dans cet exemple, nous supposons que nous voulons développer une application qui contient deux boutons, et une zone de texte. Si nous cliquons sur un des boutons, la zone du texte affiche le nom du bouton cliqué. Pour cela, nous allons avoir une classe **Bouton**, et une classe **ZoneTxt** (pour Zone de texte). Afin que la zone de texte puisse affiche le nom du bouton cliqué, nous utilisons le pattern 'Mediator', où la classe ZoneTxt joue le rôle de Mediator (pour l'affichage), et la classe Bouton joue le rôle de Colleague (pour la déclaration de la notification). (voir figure 4.12).

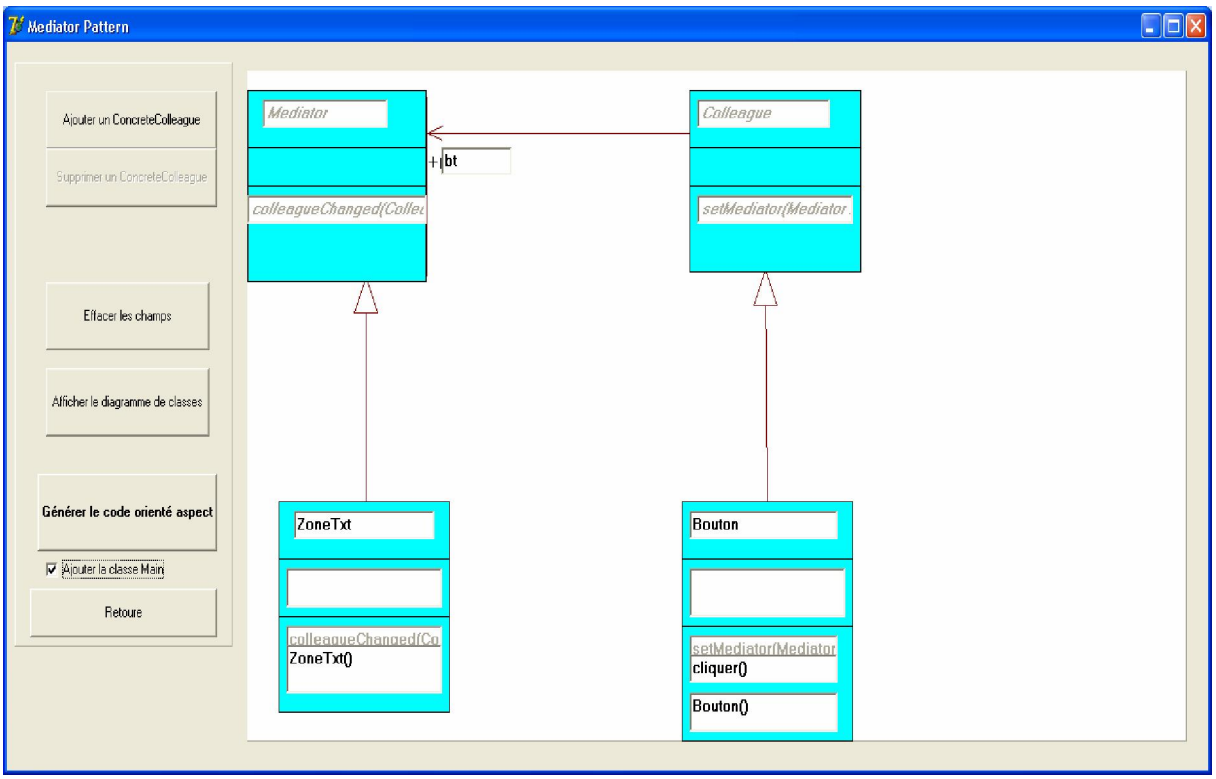


Figure 4.12 Instance du pattern 'Mediator' dans notre outil

L'annexe D présente le code généré (squelette de code) à partir de notre outil, ainsi le code final après l'implémentation des méthodes. La figure 4.13 présente l'exécution du code finale.

Figure 4.13. Résultat de l'exécution de l'exemple de 'Mediator'

4 Générateur de plusieurs instances à la fois:

Dans cette section, nous présentons une étude de cas afin de valider le générateur de plusieurs instances à la fois. Ensuite nous présentons les différentes possibilités rencontrées lors de la conception et la génération de code des design patterns dans certains cas.

4.1 Aperçu de notre outil et l'outil CASE (StarUML)

Nous avons développé un prototype, en utilisant le langage Java « JBuilder », avec l'API JDOM (Java Document Object Model) [JDOM] pour parcourir notre document XMI, afin de détecter des éventuelles instances des patterns, selon leurs algorithmes de détection.

L'interface principale contient uniquement deux boutons, un pour parcourir le document XMI, et l'autre pour générer le code orienté aspect.

Lors de la détection d'un pattern, l'interface de la confirmation sera affichée. (voir figure 4.14). Pour l'outil CASE, nous avons choisi l'outil CASE StarUML [StarUML], supportant la génération de documents XMI.

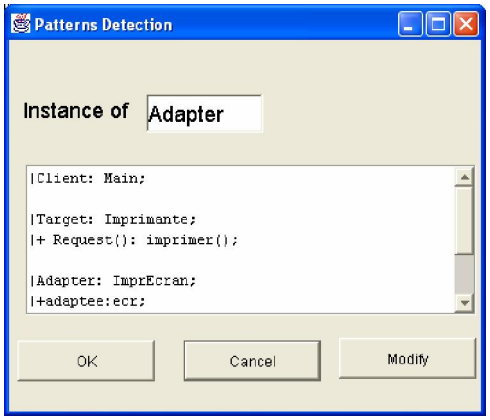


Figure 4.14. Interface de la confirmation de l'instance du pattern 'Adapter'

4.2 Cadre de validation

Dans le cadre de validation, une étude de cas pour valider notre approche sera présentée. Puis par la suite nous présentons les différentes possibilités rencontrées lors de la conception et la génération de code des design patterns qui se résument comme suit :

- Le cas où une classe appartient à deux ou plusieurs instances de patterns ;
- Le cas où il y a deux ou plusieurs types d'instances du même pattern ;
- Le cas général ;

4.2.1 Etude de cas

Nous allons étudier un exemple d'une application permettant de gérer des expressions mathématiques dont la grammaire est donnée comme suit [Hachani, 2006] :

Expression := VariableExpression | NumberExpression | PlusOperator | MinusOperator
PlusOperator := Expression '+' Expression
MinusOperator := Expression '-' Expression
VariableExpression := ('A'| 'B' | 'C' | ... | 'Z') +
NumberExpression := ('0' | '1' | '2' | ... | '9') +

L'application permet la représentation d'expressions de la forme « C+3 ». Le pattern Composite est le mieux adapté pour ce cas, où, la classe abstraite Expression joue le rôle de Component. Les classes VariableExpression et NumberExpression jouent le rôle de feuille. Enfin, les classes PlusOperator et MinusOperator jouent le rôle de Composite. Elle permet en plus l'évaluation, l'affichage et la vérification syntaxique et sémantique d'expressions. Ces opérations sont effectuées à l'intérieur des classes de cette application. Pour cela le pattern Visitor est le mieux adapté pour ce genre de cas. Trois classes sont définies pour jouer le rôle de ConcretVisitor à savoir Evaluate (pour l'opération d'évaluation), Display (pour l'opération d'affichage) et Check (pour l'opération de vérification syntaxique et sémantique). Elle offre une fonction de « logging » d'operations (un journal d'exécution). Le pattern Observer peut être efficace pour ce genre de cas, où, la classe Logger joue le rôle d'Observer et la classe Expression pour le rôle Subject (le sujet observé).
Pour ce cas, nous allons expliquer les différentes étapes de notre approche.

a) Conception avec StarUML et génération du document XMI

Dans ce cas, nous considérons 15 classes. Ainsi 3 instances : Visitor, Composite et Observer.

Les classes de l'instance du pattern 'Composite' sont : **Expression, BinaryOperation, Literal, PlusOperator, MinusOperator, VariableExpression, et NumberExpression.**
Les classes de l'instance du pattern 'Visitor' sont : **Expression, BinaryOperation, Literal, PlusOperator, MinusOperator, VariableExpression, NumberExpression, Client, Visitor, Display, Evaluate, et Check.**
Les classes de l'instance du pattern 'Observer' sont : **Expression, Logger, Observer, et Subject.**

La figure 4.15 représente le diagramme de classes de cette application. Le diagramme de séquences pour cette application contient uniquement l'appel d'un objet Subject à la méthode Update d'un objet Observer.

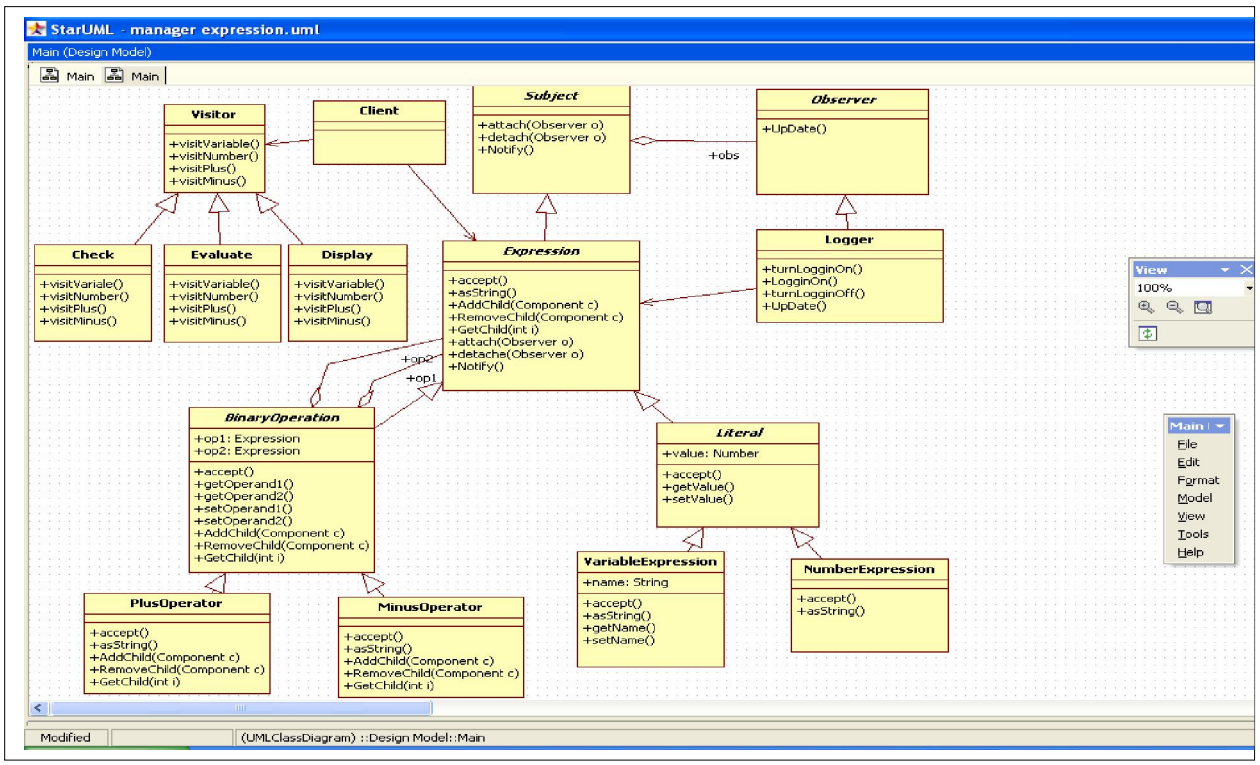


Figure 4.15 Diagramme de classes de l'application (Gestionnaire des expressions).

À partir de l'outil StarUML, il est facile de générer le document XMI.

b) Analyse du document XMI pour détecter d'éventuels patterns

L'instance du pattern 'Composite' peut être détecté par l'analyse structurelle, et exactement par le lien d'héritage et le lien d'aggrégation entre la classe Component (Expression) et les classes Composites (**PlusOperator** et **MinusOperator**), et un autre lien d'héritage entre la classe Component et les classes feuilles (**VariableExpression** et **NumberExpression**). De plus, ce pattern peut être détecté par leurs opérations (c'est-à-dire les mots-clés utilisés par ce pattern),

à savoir **AddChild** et **RemoveChild** dans les classes de Composites et Component. Cependant l'instance du pattern 'Visitor' peut être détectée aussi par l'analyse structurelle (lien d'héritage, ainsi par la classe **Visitor**). La dernière instance peut être détectée par l'analyse structurelle et comportementale (le lien d'héritage, d'aggrégation, d'association et les classes Observer et Subject pour l'analyse structurelle, et l'appel d'un objet Subject à la méthode Update d'un objet Observer pour l'analyse comportementale).

c) Confirmation et génération des instances détectés

Ces instances peuvent être validées parce qu'elles respectent complètement la structure du catalogue de GoF.

Dans cette étape, nous allons indiquer les importantes opérations effectuées par les algorithmes de transformation (génération) pour chaque instance :

Pour l'instance du pattern Composite, l'algorithme de transformations crée un aspect abstrait (**CompositeProtocole**) à partir de la base de protocoles. Ce dernier contient les opérations de ce pattern tel que : **AddChild** et **RemoveChild**. Puis il crée un aspect concret pour spécifier les classes qui jouent les rôles de Feuille et Composite.

L'intention du pattern 'Visitor' permet d'ajouter des comportements à une hiérarchie de classes d'éléments existantes. L'utilisation d'introductions d'AspectJ permet une réalisation simple et directe d'une telle préoccupation [Hachani, 2006]. Pour cela, l'algorithme du pattern 'Visitor' crée pour chaque ConcreteVisitor dans le modèle conceptuel, un aspect correspondant pour jouer le rôle de Visitor. Dans notre cas l'algorithme de 'Visitor' effectue une importante opération donnant la création de trois aspects **Display**, **Evaluate**, et **Check** correspondants à trois ConcreteVisitor.

Pour l'instance du Observer, l'algorithme de transformation crée un (**ObserverProtocol**) à partir de la base de protocoles. Ce protocole comporte les opérations de ce pattern à savoir **Attach**, **Detach** et le code de **Notify** (définis dans le 'Code Advice'). Ensuite, il crée pour chaque méthode dans Subject un aspect qui concrétise **ObserverProtocol**. Cet aspect concret spécifier les classes qui jouent les rôles de Subject, Observer et il concrétise le 'pointcut' de **ObserverProtocol**. Dans notre cas, un seul aspect concret crée, parce que nous avons une seule méthode dans la classe **Expression** (cette classe joue le rôle de Subject) à savoir **asString** (les autres méthodes dans cette classe sont liées aux opérations des patterns).

Le nom de l'aspect concret crée est **ObserverExpressionLogger_asString**. Les mots-clés utilisé dans le nom cet aspect sont **Observer**, **Expression**, **Logger**, et **asString** pour designer respectivement un aspect de Observer, le sujet observé, l'observateur et la méthode qui peut déclencher la mise à jour dans la classe Observer.

Dans certains cas, nous pouvons avoir un aspect créer pour une méthode de Subject et qui n'est pas considéré pour déclencher la mise à jour dans les classes Observer. Dans ce cas le développeur peut intervenir pour effacer cet aspect.

La figure 4.16 représente un extrait du code aspect généré.

```
// L'aspect concrets Composite Expression
public aspect CompositeExpression extends CompositeProtocol {
        declare parents MinusOperator implements Composite
        declare parents PlusOperator implements Composite
        declare parents VariableExpression implements Leaf
        declare parents NumberExpression implements Leaf }

// La classe Expression
public class Expression {
```

```
        public void asString()          {} }

// L'aspect concret ObserverExpressionLogger_asString
public aspect ObserverExpressionLogger_asString extends ObserverProtocol {
        declare parents Expression implements Subject
        declare parents Logger implements Observer

        protected pointcut subjectChange(Subject s):call(void asString()) && target(s);
        protected void updateObserver(Subject s, Observer o) {}}


// L'aspect Display
public aspect Display {
        public void BinaryOperator.display()
        {}
        public void NumberExpression.display()
        {}
        public void VariableExpression.display()
        {}
        public void BinaryOperator.display()
        {}
        public void PlusOperator.display()
        {}
        public void MinusOperator.dispaly()
        {} }
```

Figure 4.16. Extrait du code aspect généré.

Nous remarquons que la classe **Expression** dans le modèle conceptuel (ainsi que ses sous-classes) contient beaucoup d'opérations qui ne sont pas liées au code métier de cette classe (il y a les opérations des patterns Observer, Visitor et Composite). Ce qui rend l'application difficile à être compréhensible, surtout si nous générons le code objet. Pour cela notre approche sépare entre les opérations des patterns (aspects) et les opérations de l'application (classes) pour que l'application devienne plus compréhensible.
Si nous générons le code objet de cette application, nous devons implémenter les opérations liées aux patterns (par exemple **AddChild**). Par contre en aspect elle est déjà implémentée dans le protocole.
Le code aspect des patterns est plus lisible et compréhensible que le code objet, ce qui rend la maintenance et l'évolution de l'application plus facile.
Cette approche maintient la logique de conception orientée objet grâce à l'utilisation des outils CASE 'objet'. (Les outils CASE 'aspect' sont peu connus. De plus, il n y a pas un standard connu pour présenter l'aspect dans le niveau conceptuel). Ces problèmes ne se posent pas dans l'objet).
Ces avantages peuvent être vu comme la différence entre la génération du code objet et la génération du code aspect des patterns.

4.2.2 Cas où une classe appartient à deux ou plusieurs instances de patterns

Dans ce cas la génération est faite pour la première instance du pattern, puis il est possible de modifier cette classe pour les autres instances.
Dans la plupart des cas, cette classe appartient à la logique de l'application, mais non au pattern, pour cela, elle ne sera jamais supprimée. Ce genre de classe appartient au rôle surimposé [Hannemann et Kiczales, 2002].

La modification qui peut être effectuée par les autres instances, c'est la suppression d'une ou plusieurs méthodes ou d'un ou plusieurs attributs. Un exemple de ce cas, une classe qui appartient, à deux instances de pattern, 'Observer' et 'Chain of Responsibility'.
Nous supposons qu'une classe appartient à deux instances de patterns différents, à savoir 'Observer' et 'Chain of Responsibility'. Cette classe joue le rôle de 'ConcreteHandler', pour le pattern 'Chain of Responsibility' et 'ConcreteObserver' pour le pattern 'Observer'. Pour la génération de code, la première instance détectée sera générée, par exemple 'Observer', puis 'Chain of Responsibility'. Il sera possible de supprimer la méthode HandelRequest de cette classe, pour la placer dans l'aspect.

4.2.3 Cas où il y a deux ou plusieurs types d'instances du même pattern

Dans ce cas, il est possible d'avoir deux types de cas selon d'avoir ou non un aspect abstrait.
Dans le cas où il n y a aucun aspect abstrait pouvant être généré pour un pattern, la logique de la génération de code ne change pas. Mais pour le deuxième cas, il y a une seule génération de l'aspect abstrait lors de la première occurrence d'une instance de ce pattern.

4.2.4 Cas général

Apres la génération de code aspect des instances des patterns, il reste à générer le code objet du reste de l'application, c'est-à-dire parcourir le document XMI encore une fois et détecter les différentes classes qui ne sont pas générées (les classes qui sont supprimées par une instance d'un pattern ne seront pas générés).

5. Application de la rétro-ingénierie:

Dans cette section, nous allons présenter, la technique de détection des instances des patterns, ainsi l'application de notre approche de rétro-ingénierie proposé dans le chapitre 3, section 5 dans une étude de cas.

5.1 Détection des patterns :

Les instances des patterns seront détectées à partir des programmes, pour cela nous adoptons la solution de [Lee et al., 2007]. Parce qu'ils combinent l'analyse statique et dynamique pour détecter les instances des patterns, et ils définissent une autre technique de détection qui est l'analyse spécifique. De plus, ils détectent plus de patterns par rapport à d'autres techniques.

Le principe de détection est décrit comme suit :

Premier étape :
Dans cette étape, le code source sera présenté selon une représentation de haut niveau. La représentation de haut niveau peut être exprimée par divers façons de représentation tels que le AST (Abstract Syntax Tree) [Heuzeroth et al., 2003].

Deuxième étape :

Ensuite, avec les algorithmes définis pour l'analyse statique vérifient si les parties de la représentations de haut niveau sont correctes. Si la détection est satisfaisante, elle peut être stockée à la suite de l'analyse statique et peut être détecté comme un design pattern.
La détection des patterns dans l'analyse statique sera faite par l'utilisation du XMI (XML Metadata Interchange) [Grose et al. 2002] pour effectuer le 'matching check'.

Troisième étape :

Puis le programme sera exécuté, afin de détecter les appels entre les méthodes. Si les appels des méthodes satisferont les règles proposées pour détecter les patterns, l'analyse dynamique peut être déclaré comme une détection satisfaisante.

La détection des patterns dans l'analyse dynamique sera faite par l'utilisation du JDI (Java Debug Interface), afin de détecter les appels entre les méthodes.

Quatrième étape :

Cette analyse détecte uniquement les mots-clés utilisés par les patterns, tels que le pattern 'Prototype' avec le mot-clé clone, ou bien, par une implémentation spécifique comme le cas du pattern 'Singleton'.

Apres la détection du pattern, nous allons appliquer notre approche de rétro-ingénierie qui se base sur la représentation aspect des patterns. La section suivante décrit notre approche de rétro-ingénierie.

5.2 Rétro-ingénierie des instances basées aspects :

Dans cette section, nous allons appliquer la rétro-ingénierie sur le même cas proposé dans la section 4.2.1, sauf dans ce cas nous supposons que nous avons le code orienté objet de cette application. Nous allons expliquer les différentes étapes de notre approche de Rétro-ingénierie :

La première étape est basée sur des principes standard, qui est une analyse du code source d'une application afin d'obtenir un arbre syntaxique abstrait (AST pour Abstract Syntax Tree)

La deuxième étape consiste à détecter les instances des patterns. Les instances dans notre cas sont faciles à détecter parce qu'elles respectent complètement la structure de GoF.

La troisième étape consiste à transformer les instances détectées de l'orientées objets vers l'orientées aspect. Le principe de transformation est le même que le cas de la génération de code (voir section 4.2.1.b).

La dernière étape consiste à générer le document XMI. La figure 4.17 représente le diagramme de classe pour AspectJ de cette étude de cas.

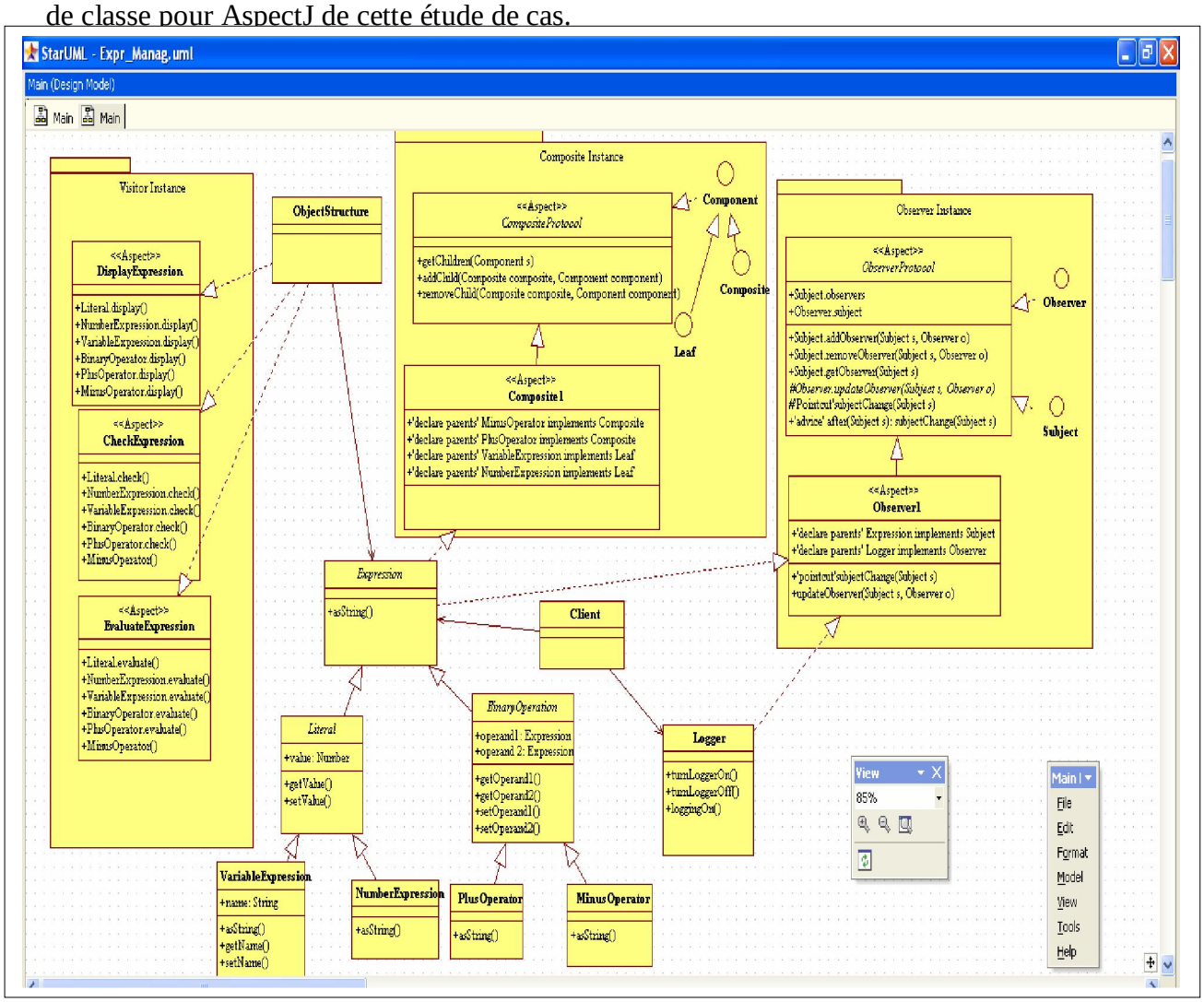


Figure 4.17. Diagramme de classes pour l'aspect de l'application (Gestionnaire des expressions).

6. Conclusion :

Dans ce présent chapitre, nous avons proposé un prototype pour le générateur orienté aspect des design pattern pour une seule instance à la fois. Ce dernier présente un pattern pour chaque classe de transformation.

Ensuite, nous avons exposé une application pour générer le code orienté aspect pour plusieurs instances à la fois. Cette application fonctionne avec les outils CASE (les outils qui supportent la conception UML, ainsi la génération du document XMI). Dans ce cas la validation est faite par l'outil CASE StarUML pour concevoir les applications objets et générer le code XMI, ensuite, notre outil qui accepte ce dernier comme entrée, afin de détecter les patterns et, il génère le code aspect pour les patterns détectés et le code objet pour le reste de l'application.

Pour terminer ce chapitre, nous avons décrit le principe de la Rétro-ingénierie.

Nous avons cité les differents outils necessaries à la realisation des applications. Enfin, nous avons également exposé les différentes interfaces réalisées.

Le chapitre suivant conclut notre travail mais permet surtout d'établir des perspectives de recherche dans le domaine de la POA et les design patterns.

Conclusion générale

Conclusion générale

Les design patterns constituent un moyen efficace pour la conception, la composition de plusieurs types de composants réutilisables, et pour le développement de grands systèmes complexes. Ils permettent d'améliorer la qualité et la compréhension des programmes, facilitent leur évolution et augmentent notamment leur réutilisation. Par ailleurs, la programmation orientée aspect a émergé suite à différents travaux de recherche, dont l'objectif principal était d'améliorer la modularité des applications afin de faciliter leur évolution et leur réutilisation. Nous avons présenté dans cette thèse un état de l'art de la programmation par aspects (POA) et du langage AspectJ. Les mécanismes et concepts Aspect (fournit par le langage AspectJ) offre une meilleure structuration des programmes par la séparation des préoccupations transversales au découpage des applications en termes de classes et aspects. Nous avons particulièrement présenté une étude approfondie des design patterns, et nous avons discuté ainsi de l'implémentations aspect des patterns. L'implémentation des design pattern à l'aide de l'approche aspect, permet de séparer le code des instances des patterns de celui de l'application, afin de bénéficier des avantages suivantes : la localisation, la réutilisation, la composition et enfin l'adaptabilité. Cette implémentation ou bien ce passage de l'objet vers l'aspect est effectué par le langage naturel. Ce problème a été considéré comme le problème majeur de notre travail.

L'objectif du travail présenté a consisté à la proposition d'un modèle de transformation des design patterns vers des programmes orientés aspects, pour répondre au problème posé de l'automatisation de passage des patterns de l'objet vers l'aspect. Cette automatisation permet au développeur de réduire le temps, l'effort ainsi le coût dans le développement des systèmes orientés aspects.

Nous avons présenté une approche globale pour la transformation, permettant d'assurer le passage de l'OO vers l'OA. Cette approche constitue en un ensemble d'algorithmes destiné à traduire des patterns. Ensuite, nous avons proposé des situations d'application de ce modèle. Ces situations représentent des différents cas de transformation. Cette transformation peut apparaître, comme une génération de code. Elle est vue aussi comme une opération de rétro-ingénierie.

Afin de valider notre modèle de transformation, nous avons essayé de valider chaque situation de transformation proposée à part. Dans la situation de la génération de code, nous avons développé deux outils. Le premier est un générateur d'une seule instance à la fois, où le modèle conceptuel représente uniquement une instance d'un pattern à la fois. Le deuxième est un générateur de plusieurs instances à la fois, où le modèle conceptuel peut contenir plusieurs patterns à la fois. La dernière situation consiste à une présentation du principe de la rétro-ingénierie des instances basée sur l'approche aspect.

Ce travail nous a permis d'avoir de nouvelles perspectives à savoir :

- Nous avons déjà mentionné qu'il n'y a pas une unique façon d'implémenter les patterns en aspect, même pour l'objet. Notre travail considère seulement l'implémentation de Hannemann et Kiczales pour l'aspect. Pour le niveau conceptuel nous nous sommes limités par la structure proposée par le catalogue de GoF. Nous pourrions également explorer d'autres implémentations aspect, par exemple mettre des critères entre les différentes implémentations orientées aspect pour un même pattern, à savoir la réutilisation, le mieux adapté pour éviter le problème de FPP (Fragile Poincut Problem).

- Le générateur de plusieurs instances à la fois détecte uniquement les patterns qui respectent la structure de GoF. Il est possible de l'enrichir par d'autre implémentation du même pattern. (par exemple le pattern Observer possède au moins douze implémentations différentes).

- La composition des patterns dans l'approche aspect est autre perspective pour aboutir à des programmes plus réutilisables.

- Une autre perspective très intéressante, est la création des design patterns destinée à résoudre des problèmes entre les aspects.

Références bibliographies

Références bibliographiques

[Adams et al., 95] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve and K. Nicodemus *''Fault-Tolerent Telecommunication System Patterns''*. AT&T Bell Laboratoires, (1995).

[Alexander et al., 77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-king and S. Aangel *''A Pattern Language''*. Oxford University Press, New York, NY, (1977).

[Alexander 79] C. Alexander *''The Timeless Way of Building''*, Oxford University Press, New York, NY, (1979).

[Ambler 98] S.W. Ambler *''Process Patterns: Building Large-Scale Systems Using Object Technology''*. Cambridge University Press, (1998).

[Appleton 97] B. Appleton *''Pattern and Software: Essential Concepts and Technology''*. (1997).

[Asplund et al., 73] K. Asplund, and G. Swift ''Downstream Water Volume'', (1973).

[Basch et al., 03] M. Basch and A. Sanchez *''Incorporating aspects into the UML''*. 3rd International Workshop on Aspect-Oriented Modeling (in AOSD 2003), Boston, USA, (2003).

[Beck 97] K. Beck *''Best Practice Patterns''*. Vol. 1: Coding, Prentice Hall, NJ, (1997).

[Berkane et al., 2008] M.L. Berkane, and M. Boufaida, *"Un processus de transformation pour la mise en oeuvre des patterns basée sur l'approche par aspects"*, séminaire national en informatique snib'08, p253-259, Biskra 06-08 mai 2008.

[Berkane et al., 2008a] M.L. Berkane, and M. Boufaida, *'' A process to reverse engineering based on aspect-oriented implementation of design patterns''*, ACIT'08, article soumis et accépté en Tunisie, 2008.

[Booch 97] G. Booch *''Ingénierie du logiciel avec ADA: de la conception à la réalisation''* InterEditions, (1997).

[Brad 1986., et al] J.Brad, and J. Novobilski. *''Object-Oriented Programming: An Evolutionary Approach''*, ISBN 0201548348. (1986).

[Brown et al., 98] W.J. Brown, R. C. Malveau, W.H. Brown and T.J. Mowbray *''Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis''*. John Wiley and Sons, 1st edition, (1998).

[Buschmann 96] F. Buschmann *''What is a pattern?''*. Object Expert, vol. 1, n°3, p.17-18, (1996).

[Buschmann et al., 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. *Stal '' Pattern-Oriented Software Architecture, A System of Patterns''*. Wiley and Sons Ltd., New York, (1996).

[Clarke et al., 01] S. Clarke and R.J. Walker *''Composition Patterns: An Approach to Designing Reusable Aspects''*. In Proceedings of ICSE 2001, IEEE Computer Society, pp. 5-14, (2001).

[Coad 92] P. Coad *''Object-Oriented Patterns''*. Communications of the ACM 35(9):152-159, (1992).

[Coad et al., 97] P. Coad, D. North and M. Mayfield *''Object Models: Strategies, Patterns, and Applications''*. Prentice Hall, (1997).

[Conte et al., 01a] A. Conte, M. Fredj, J.P. Giraudin and D. Rieu *''P-Sigma : un formalisme pour une représentation unifiée de patrons''*. Inforsid'01, Genève, Mai 2001, pp. 67-86.

[Conte et al., 01b] A. Conte, J.P. Giraudin, I. Hassine and D. Rieu *''Un environnement et un formalisme pour la définition, la gestion et l'application de patrons. Revue ISI, numéro spécial Formalismes et Modèles pour les Systèmes d'Informations''*, Vol. 6, n° 2, Hermés, (2001).

[Coplien 96] J.O. Coplien *''Patterns, the Patterns White Paper''*. (1996).

[Coplien 98] J.O. Coplien *''The patterns Handbook: Techniques, Strategies and Applications''*. Cambridge University Press, New York, (1998).

[Fowler 97] M. Fowler *''Analysis Patterns – Reusable Object Models''*. Addison-Wesley, (1997).

[Front et al., 99] A. Front, J-P. Giraudin, D. Rieu and C. Saint-Marcel *''Réutilisation et patrons d'ingénierie''*. Chapitre 4 du livre Génie Objet - Analyse et Conception de l'Evolution, Editions Hermès, , pp.91-136. (1999).

[Gamma 91] E. Gamma Thèse de PhD *''Object-Oriented Software Developments based on ET++: Design Patterns, Class Library, Tools''*, University of Zurich, (1991).

[Gamma et al., 1995] E. Gamma, R. Helm, R. Johnson and J. Vlissides *"Design Patterns, Elements of reusable Object-Oriented Software"*. Addison-Wesley Publishing Company, (1995).

[Grose et al. 2002] T.J. Gross, Doney, G.C. and Brodsky, S.A. *''Mastering XMI :Java Programming with XMI, XML, and UML''*. (2002)

[Gzara 00] L. Gzara ''Les *patrons pour l'ingénierie des Systèmes d'Information Produit''*. Thèse de doctorat de l'INPG, spécialité Génie Industriel, (2000).

[Hachani 02] O. Hachani, Rapport de DEA,*''Apports de la programmation par aspects dans l'implémentation des patrons de conception par objets''*. Rapport de DEA de l'UJF Grenoble, (2002).

[Hachani 2006] O. Hachani, Thèse de Doctorat, *"Patrons de conception a base d'aspects pour l'ingénierie des systèmes d'information par réutilisation*", Université de Joseph Fourier Grenoble 1, (2006).

[Han et al., 04b] Y. Han, G. Kniesel and A. B. Cremers *''A meta model and modelling notation for AspectJ''*. In the 5th Aspect-Oriented Modeling Workshop in conjunction with UML 2004, Lisbon, Portugal, (2004).

[Hannemann et Kiczales, 2002] J. Hannemann and G. Kiczales *"Design Pattern Implementation in Java and AspectJ"*. In Proceedings of OOPSLA 2002, ACM SIGPLAN Notices, Vol. 37, n° 11, p. 161-173, (2002).

[Heuzeroth et al., 2003] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe, *"Automatic Design Pattern Detection"*, Proc. of the 11th IEEE International Workshop on Program Comprehension. (2003).

[Hirschfeld et al., 2003] R. Hirschfeld, R. Lammel and M. Wagner *"Design Patterns and Aspects Modular Designs with Seamless Run-Time Integration"*. In the 3rd German Workshop on AOSD, p. 25-32, (2003).

[Johnson 92] R.E. Johnson *''Documenting Frameworks using Patterns''*. In Proccedings of OOPSLA'92, Vancouver BC, (1992).

[Karmer et Prechlet 1996] C. Kramer and Prechelt. Design Recovery by Automated Search for Structurel Design Patterns in Object-Oriented Software. In Proceeddings of Third Working Conference on Reverse Engineering. Monterey, CA, USA, IEEE Computer Society Press :pp 208-215.(1996).

[Kiczales 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier and J. Irwin *"Aspect-Oriented Programming"*. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). LNCS vol.1241, Springer-Verlag, (1997).

[Kiczales et al., 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm et W.G. Griswold *''An Overview of AspectJ. In Proceedings of ECOOP 2001, European Conference on Object-Oriented Programming''*, Budapest, Hungary, LNCS, vol. 2072, Springer, pp. 327-353, (2001).

[Koppen et Stoerzer 2004] C.Koppen and M.Stoerzer. "*Attacking the fragile pointcut problem*". In First European Interactive Workshop on Aspects in Software (EIWAS), (2004).

[Larman 02] C. Larman *''UML et les designs patterns''*. Campus press, (2002).

[Lee et al., 2007] H. Lee H.Youn, and E.Lee, *"Automatic Detection of Design Pattern for ReverseEngineering"*. In Proceedings of Fifth International Conference on Software Engineering Research, Management and Applications. IEEE Computer Society. (2007).

[Lorenz 1998] D.H. Lorenz *"Visitor Beans: An Aspect-Oriented Pattern"* In ECOOP'98 Workshop on Aspect-Oriented Programming, (1998).

[Meyer 2000] B.Meyer. *''Conception et programmation orientées objet''*, ISBN 2-212-09111-7. (2000).

[Muller et al., 2002] P.A.Muller, N.Gaertner *''Modélisation objet avec UML''*, (2002).

[Noda et al., 2001] N. Noda and T. Kishi *"Implementing Design Patterns Using Advanced Separation of Concerns"*. In OOPSLA 2001 Workshop on ASoC in OOS, (2001).

[Nordberg 2001] M.E. Nordberg *"Aspect-Oriented Dependency Inversion"*In OOPSLA 2001 Workshop on ASoC in OOS, (2001).

[Pawlak et al., 2004] R.Pawlak , J.P.Retaillé, and L.Seinturier *"Programmation orienté aspect pour Java/J2EE"* , (2004).

[Rising 00] L. Rizing *''The Pattern Almanac''*. Software Pattern Series, Addison-Wesley, (2000).

[Shi et al., 2006] N.Shi and R.A.Olsson "*Reverse Engineering of Design Patterns from Java Source Code* ", Proc 21st IEEE International Conference on Automated Software Engineering, (2006).

[Stein 02] D. Stein, Thèse de Master *''An Aspect-Oriented Design Model Based on AspectJ and UML''*, University of Essen, Germany, (2002).

[Sunyé 99] G. Sunyé Génération de code à l'aide de patrons de conception. In Langages et Modéles à Objets - LMO'99, Villefranche sur mer, (1999).

[Suzuki et al., 1999] J. Suzuki and Y. Yamamoto " *Extending UML with Aspects: Aspect Support in the Design Phase"*. ECOOP'99 Workshop on Aspect-Oriented Programming, (1999).

[Turner et al., 03] J. Turner and K. Bedell *''Stratuts''*. Edition CampusPress, (2003).

[Zakaria et al., 02] A. A. Zakaria, H. Hosny and A. Zeid *'' A UML Extension for Modeling Aspect-Oriented Systems''*. 2nd International Workshop on Aspect-Oriented Modeling with UML. (2002).

[AspectJ] http://www.eclipse.org/aspectj

[ATL] www.eclipse.org/m2m/**atl**/

[MDA] Model Driven Architecture (MDA) guide v 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[QVT] Query/View/Transformation (QVT) Specification, http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[JDOM] http://www.jdom.org

[StarUML] http://www.staruml.com/

Annexe A

Annexe A : les modèles et les codes du pattern ‘Mediator’

1. Modèles orientés objets :

Nous allons présenter dans ce point le modèle conceptuel et le modèle physique orientés objets. Nous allons considérer une instance du pattern ‘Mediator’ dans un contexte de réalisation où les participants de cette instance portent les mêmes noms de ceux du pattern lui-même (contexte général). Ce pattern constitue de quatre participants : Mediator, Colleague, Concrete Mediator, et Concrete Colleague.

1.1 Modèle conceptuel orienté objet :

Le modèle conceptuel d’une instance du pattern ‘Mediator’ dans le contexte général est représenté par deux diagrammes. Le diagramme de classes pour illustrer la structure statique. (voir figure D.1). Le diagramme de séquences pour illustrer le comportement dynamique. (voir figure D.2).

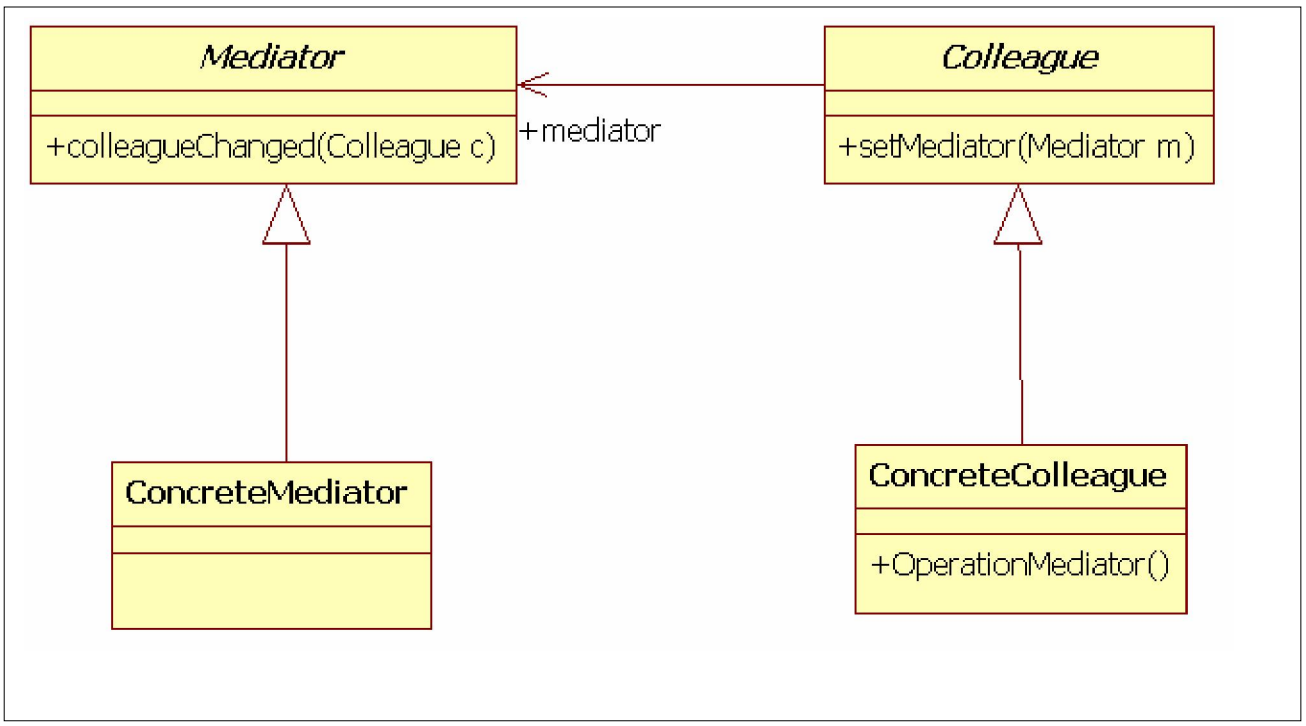


Figure D.1. Instance du pattern ‘Mediator’ au niveau conceptuel représenté par le diagramme de classes.

Le diagramme de séquences de ce pattern est représenté comme suit :

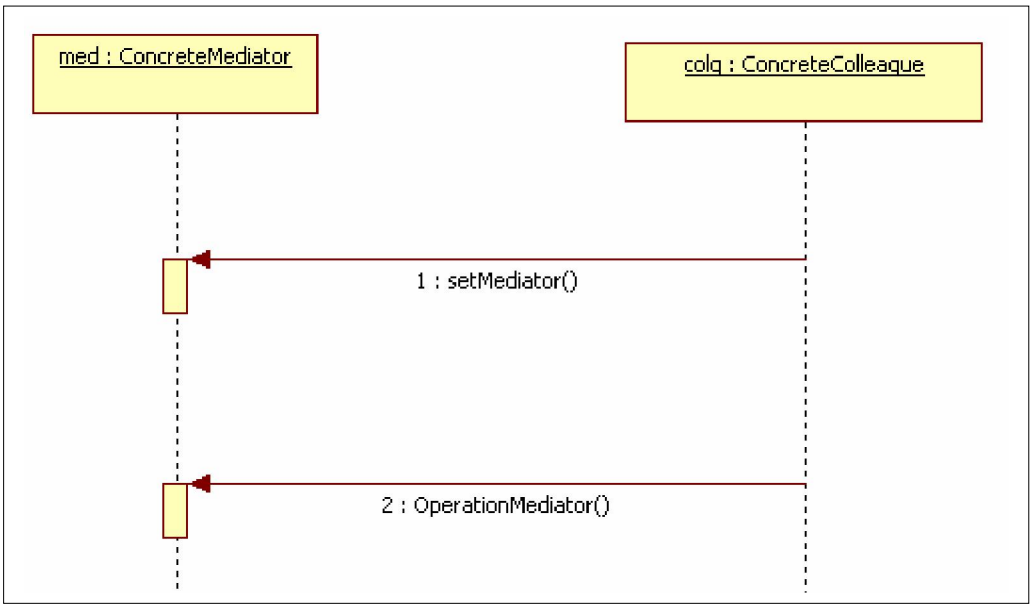


Figure D.2. Instance du pattern 'Mediator' au niveau conceptuel représenté par le diagramme de séquences.

1.2 Modèle physique orienté objet :

Le modèle physique (représenté par le langage Java) d'une instance du pattern 'Mediator' dans le contexte général est illustré dans la figure D.3.

```
// L'interface Mediator
public interface Mediator {

    public void colleagueChanged(Colleague colleague);
}

// L'interface Colleague
public interface Colleague {

    public void setMediator(Mediator mediator);
}

// La classe ConcreteColleague
public class ConcreteColleague implements Colleague {

    private Mediator mediator;
    public void OperationMediator() {}
        public void setMediator(Mediator mediator) {
        this.mediator = mediator;}
        }

// La classe ConcreteMediator
public class ConcreteMediator implements Mediator {

    public void colleagueChanged(Colleague colleague) { }
}
```

Figure D.3. Instance du pattern 'Mediator' au niveau physique représenté par le langage Java.

2. Modèles orientés aspects :

Apres l'application de l'algorithme de transformation dans le contexte de réalisation générale, nous obtenons le modèle conceptuel (représenté par le diagramme de classes pour AspectJ) illustré dans la figure D.4.
Le comportement de ce pattern est défini dans les aspects (abstrait et concret).

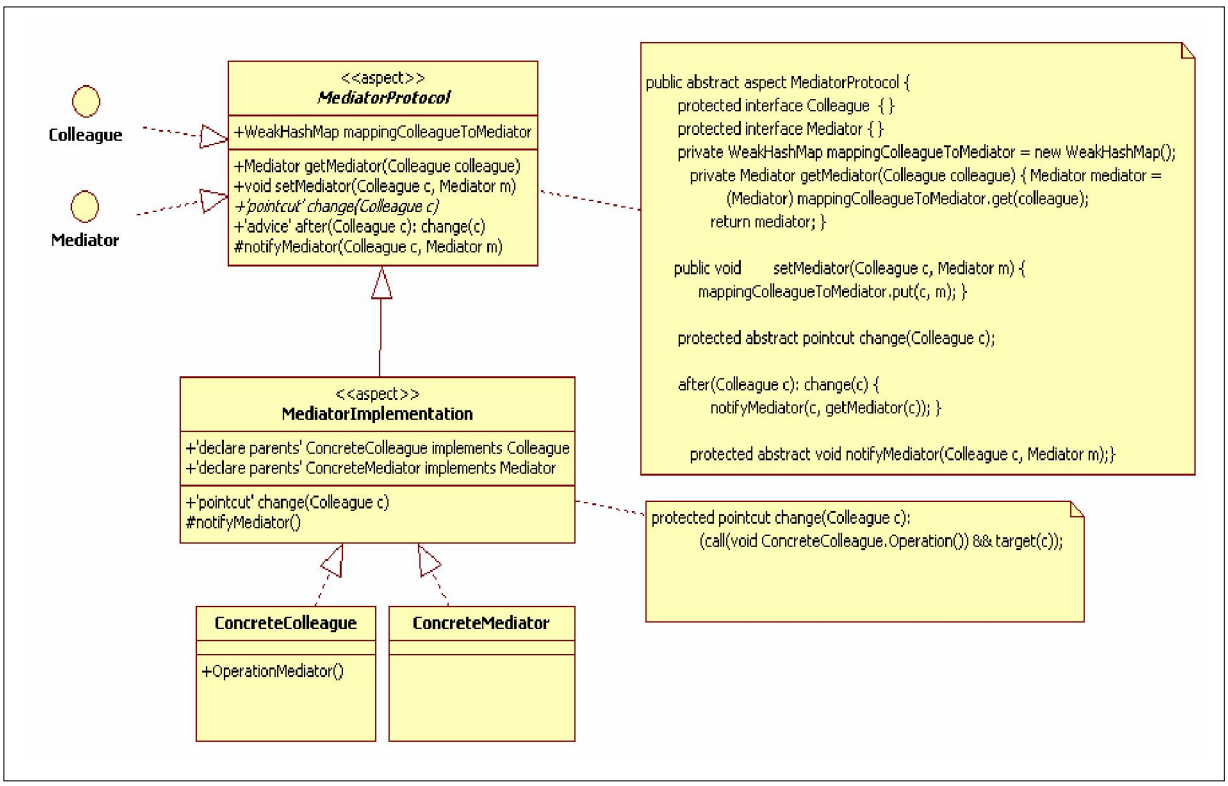


Figure D.4. Instance du pattern 'Mediator' au niveau conceptuel O.A représenté par le diagramme de classes UML pour AspectJ.

Le modèle physique (représenté par le langage AspectJ) obtenu après la transformation est illustré dans la figure D.5.

```
public aspect MediatorImplementation extends MediatorProtocol {

    declare parents: ConcreteColleague implements Colleague;

    declare parents: ConcreteMediator implements Mediator;

    protected pointcut change(Colleague c):
        (call(void ConcreteColleague.OperationMediator())) && target(c);

    protected void notifyMediator(Colleague c, Mediator m) { }

}

public class ConcreteColleague
{
    OperationMediator(){}
}

public class ConreteMediator {}
import java.util.WeakHashMap;
import java.util.Collection;
import java.util.LinkedList;
import java.util.Iterator;

public abstract aspect MediatorProtocol {

    protected interface Colleague  { }
    protected interface Mediator { }

    private WeakHashMap mappingColleagueToMediator = new WeakHashMap();
    private Mediator getMediator(Colleague colleague) {
        Mediator mediator =
            (Mediator) mappingColleagueToMediator.get(colleague);
        return mediator;}

    public void setMediator(Colleague c, Mediator m) {
        mappingColleagueToMediator.put(c, m);}

    protected abstract pointcut change(Colleague c);

    after(Colleague c): change(c) {
        notifyMediator(c, getMediator(c));}
```

Figure D.5. Instance du pattern ‘Mediator’ au niveau physique O.A représenté par le langage AspectJ.

3. Etude de cas :

Nous avons supposé dans le chapitre 4 dans l'étude de cas du pattern 'Mediator' que nous voulons développer une application qui contient deux boutons, et une zone de texte. Si nous cliquons sur un des boutons, la zone du texte affiche le nom du bouton cliqué.
Le code généré par notre outil de cette application est présenté dans la figure D.6.

```
// Le protocole MediatorProtocol

import java.util.WeakHashMap;
import java.util.Collection;
import java.util.LinkedList;
import java.util.Iterator;

public abstract aspect MediatorProtocol {

	protected interface Colleague  { }

	protected interface Mediator { }

	private WeakHashMap mappingColleagueToMediator = new WeakHashMap();

	private Mediator getMediator(Colleague colleague) {
		Mediator mediator =
		    (Mediator) mappingColleagueToMediator.get(colleague);
		return mediator;
	}

	public void setMediator(Colleague c, Mediator m) {
	    mappingColleagueToMediator.put(c, m);
	}

	protected abstract pointcut change(Colleague c);

	after(Colleague c): change(c) {
		notifyMediator(c, getMediator(c));
	}

	protected abstract void notifyMediator(Colleague c, Mediator m);
}

// La classe Bouton

public class Bouton {

	public void cliquer() {}

	public void Bouton() {}
}

// La classe ZoneTxt

public class ZoneTxt  {
	public void ZoneTxt() {}

}



// L'aspect Mediator_Bouton_ZoneTxt
public aspect Mediator_Bouton_ZoneTxt extends MediatorProtocol {
```

```
    declare parents: Bouton implements Colleague;

    declare parents: ZoneTxt  implements Mediator;

protected pointcut change(Colleague c): (call(void Boutton.cliquer()) &&
target(c));

protected void notifyMediator(Colleague c, Mediator m) {}

}
```

Figure D.6. Code généré de l'exemple étudié de 'Mediator'.

Pour que ce programme devient prêt à être exécuté, nous allons implémenter les méthodes (constructeurs) des classes **Bouton**, **ZoneTxt**, et la déclaration de la classe **Main**. La seule chose qui peut être faite dans l'aspect, c'est l'implémentation de la méthode **`notifyMediator.`** La figure D.7 représente ces modifications.

```
// La classe Bouton

import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

class Bouton extends JButton {


      public Bouton(String name) {
            super(name);
            this.setActionCommand(name);
            this.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        clicked();
                  }
            });
      }

      public void cliquer() {}
      }

// La classe ZoneTxt

import javax.swing.*;

public class ZoneTxt extends JLabel {

      public ZoneTxt(String s) {
            super(s);
      }
}




// La méthode notifyMediator

protected void notifyMediator(Colleague c, Mediator m) {
            Bouton bt = (Bouton) c;
            ZoneTxt  zt  = (ZoneTxt)  m;
            if (bt == Main.bouton1) {
```

```
                zt.setText("Bouton1 cliqué");
            } else if (button == Main.button2) {
                zt.setText("Bouton2 cliqué");
            }
            bt.setText("(Bouton cliqué)");
        }


// extrait de la méthode Main de la classe Main
      public static void main(String[] args) {


            ……

          MediatorImplementation.aspectOf().setMediator(bouton1, zontxt);
          MediatorImplementation.aspectOf().setMediator(bouton2, zontxt);

// Le mot-clé aspectOf() est utilisé pour fair un appel à une méthode dans
l'aspect.
      }
}
```

Figure D.7. Extrait du code final de l'exemple étudié de 'Mediator'.

Annexe D

Annexe D : les modèles et les codes du pattern ‘Mediator’

4. Modèles orientés objets :

Nous allons présenter dans ce point le modèle conceptuel et le modèle physique orientés objets. Nous allons considérer une instance du pattern ‘Mediator’ dans un contexte de réalisation où les participants de cette instance portent les mêmes noms de ceux du pattern lui-même (contexte général). Ce pattern constitue de quatre participants : Mediator, Colleague, Concrete Mediator, et Concrete Colleague.

4.1 Modèle conceptuel orienté objet :

Le modèle conceptuel d’une instance du pattern ‘Mediator’ dans le contexte général est représenté par deux diagrammes. Le diagramme de classes pour illustrer la structure statique. (voir figure D.1). Le diagramme de séquences pour illustrer le comportement dynamique. (voir figure D.2).

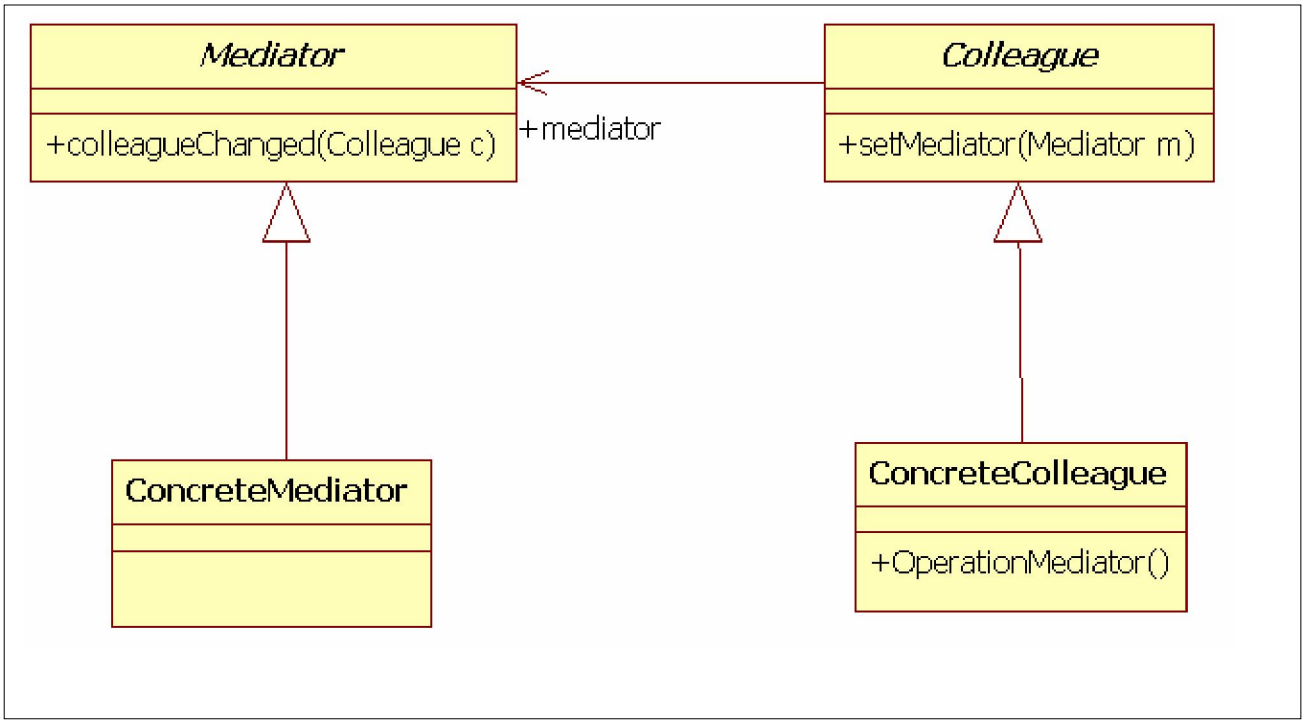


Figure D.1. Instance du pattern ‘Mediator’ au niveau conceptuel représenté par le diagramme de classes.

Le diagramme de séquences de ce pattern est représenté comme suit :

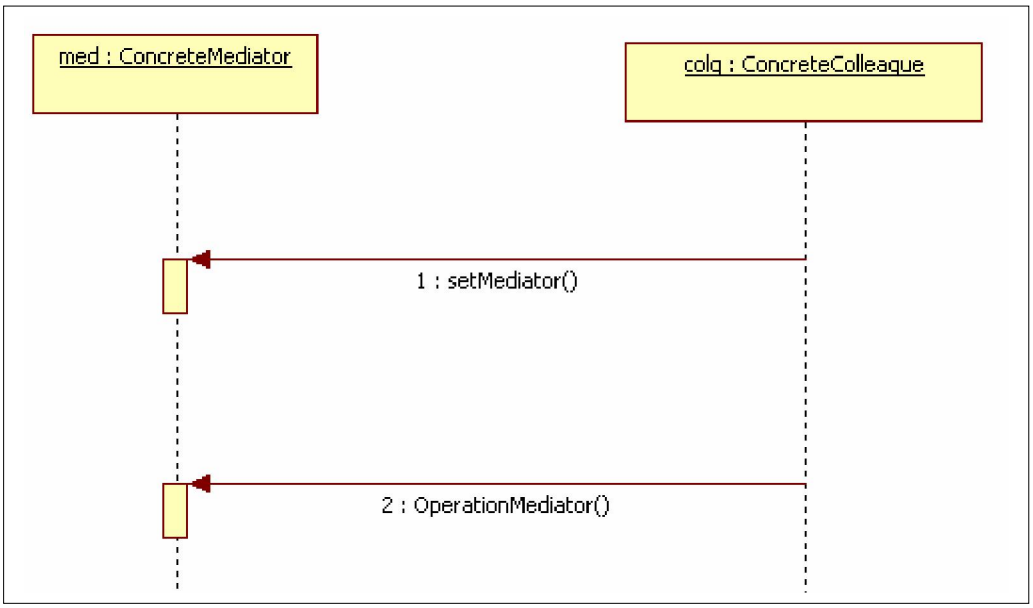


Figure D.2. Instance du pattern ‘Mediator’ au niveau conceptuel représenté par le diagramme de séquences.

4.2 Modèle physique orienté objet :

Le modèle physique (représenté par le langage Java) d’une instance du pattern ‘Mediator’ dans le contexte général est illustré dans la figure D.3.

```
// L'interface Mediator
public interface Mediator {

    public void colleagueChanged(Colleague colleague);
}

// L'interface Colleague
public interface Colleague {

    public void setMediator(Mediator mediator);
}

// La classe ConcreteColleague
public class ConcreteColleague implements Colleague {

    private Mediator mediator;
    public void OperationMediator() {}
        public void setMediator(Mediator mediator) {
        this.mediator = mediator;}
        }

// La classe ConcreteMediator
public class ConcreteMediator implements Mediator {

    public void colleagueChanged(Colleague colleague) { }
}
```

Figure D.3. Instance du pattern ‘Mediator’ au niveau physique représenté par le langage Java.

5. Modèles orientés aspects :

Apres l'application de l'algorithme de transformation dans le contexte de réalisation générale, nous obtenons le modèle conceptuel (représenté par le diagramme de classes pour AspectJ) illustré dans la figure D.4.
Le comportement de ce pattern est défini dans les aspects (abstrait et concret).

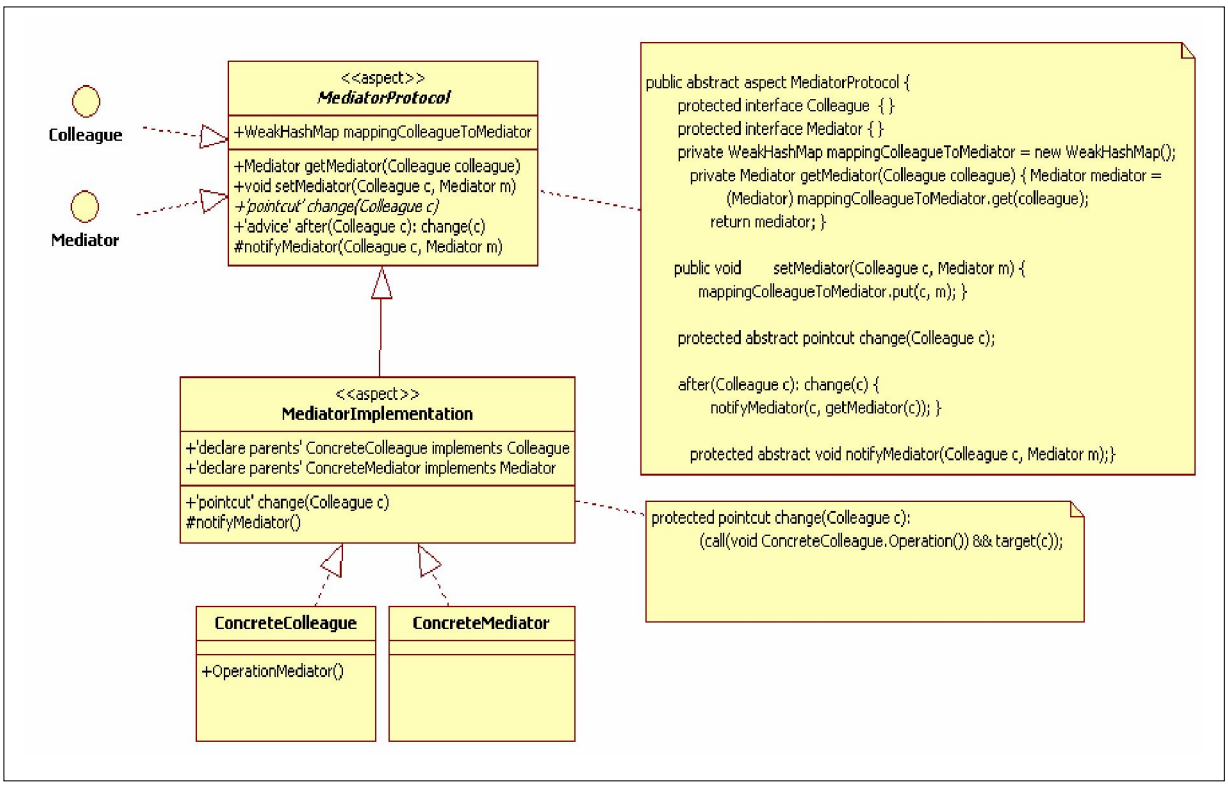


Figure D.4. Instance du pattern 'Mediator' au niveau conceptuel O.A représenté par le diagramme de classes UML pour AspectJ.

Le modèle physique (représenté par le langage AspectJ) obtenu après la transformation est illustré dans la figure D.5.

```
public aspect MediatorImplementation extends MediatorProtocol {

    declare parents: ConcreteColleague implements Colleague;

    declare parents: ConcreteMediator implements Mediator;

    protected pointcut change(Colleague c):
        (call(void ConcreteColleague.OperationMediator())) && target(c);

    protected void notifyMediator(Colleague c, Mediator m) { }

}

public class ConcreteColleague
{
    OperationMediator(){}
}

public class ConreteMediator {}
import java.util.WeakHashMap;
import java.util.Collection;
import java.util.LinkedList;
import java.util.Iterator;

public abstract aspect MediatorProtocol {

    protected interface Colleague  { }
    protected interface Mediator { }

    private WeakHashMap mappingColleagueToMediator = new WeakHashMap();
    private Mediator getMediator(Colleague colleague) {
        Mediator mediator =
            (Mediator) mappingColleagueToMediator.get(colleague);
        return mediator;}

    public void setMediator(Colleague c, Mediator m) {
        mappingColleagueToMediator.put(c, m);}

    protected abstract pointcut change(Colleague c);

    after(Colleague c): change(c) {
        notifyMediator(c, getMediator(c));}
```

Figure D.5. Instance du pattern ‘Mediator’ au niveau physique O.A représenté par le langage AspectJ.

6. Etude de cas :

Nous avons supposé dans le chapitre 4 dans l'étude de cas du pattern 'Mediator' que nous voulons développer une application qui contient deux boutons, et une zone de texte. Si nous cliquons sur un des boutons, la zone du texte affiche le nom du bouton cliqué.
Le code généré par notre outil de cette application est présenté dans la figure D.6.

```
// Le protocole MediatorProtocol

import java.util.WeakHashMap;
import java.util.Collection;
import java.util.LinkedList;
import java.util.Iterator;

public abstract aspect MediatorProtocol {

	protected interface Colleague  { }

	protected interface Mediator { }

	private WeakHashMap mappingColleagueToMediator = new WeakHashMap();

	private Mediator getMediator(Colleague colleague) {
		Mediator mediator =
		    (Mediator) mappingColleagueToMediator.get(colleague);
		return mediator;
	}

	public void setMediator(Colleague c, Mediator m) {
	    mappingColleagueToMediator.put(c, m);
	}

	protected abstract pointcut change(Colleague c);

	after(Colleague c): change(c) {
		notifyMediator(c, getMediator(c));
	}

	protected abstract void notifyMediator(Colleague c, Mediator m);
}

// La classe Bouton

public class Bouton {

	public void cliquer() {}

	public void Bouton() {}
}

// La classe ZoneTxt

public class ZoneTxt  {
	public void ZoneTxt() {}

}



// L'aspect Mediator_Bouton_ZoneTxt
public aspect Mediator_Bouton_ZoneTxt extends MediatorProtocol {
```

```
    declare parents: Bouton implements Colleague;

    declare parents: ZoneTxt  implements Mediator;

protected pointcut change(Colleague c): (call(void Boutton.cliquer()) &&
target(c));

protected void notifyMediator(Colleague c, Mediator m) {}

}
```

Figure D.6. Code généré de l'exemple étudié de 'Mediator'.

Pour que ce programme devient prêt à être exécuté, nous allons implémenter les méthodes (constructeurs) des classes **Bouton**, **ZoneTxt**, et la déclaration de la classe **Main**. La seule chose qui peut être faite dans l'aspect, c'est l'implémentation de la méthode **`notifyMediator.`** La figure D.7 représente ces modifications.

```
// La classe Bouton

import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

class Bouton extends JButton {


      public Bouton(String name) {
            super(name);
            this.setActionCommand(name);
            this.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        clicked();
                  }
            });
      }

      public void cliquer() {}
      }

// La classe ZoneTxt

import javax.swing.*;

public class ZoneTxt extends JLabel {

      public ZoneTxt(String s) {
            super(s);
      }
}




// La méthode notifyMediator

protected void notifyMediator(Colleague c, Mediator m) {
            Bouton bt = (Bouton) c;
            ZoneTxt  zt  = (ZoneTxt)  m;
            if (bt == Main.bouton1) {
```

```
                zt.setText("Bouton1 cliqué");
            } else if (button == Main.button2) {
                zt.setText("Bouton2 cliqué");
            }
            bt.setText("(Bouton cliqué)");
        }


// extrait de la méthode Main de la classe Main
      public static void main(String[] args) {


            ……

          MediatorImplementation.aspectOf().setMediator(bouton1, zontxt);
          MediatorImplementation.aspectOf().setMediator(bouton2, zontxt);

// Le mot-clé aspectOf() est utilisé pour fair un appel à une méthode dans
l'aspect.
      }
}
```

Figure D.7. Extrait du code final de l'exemple étudié de 'Mediator'.