Ingénieurs 2000

Exposé POA / AspectJ

Programmation Orientée Aspect avec AspectJ

Sophie LEVY (sophie.levy@terravirtual.net)

Février 2006

Ingénieurs 2000

Informatique et Réseau 3ème année – Groupe 1

Introduction

Cet exposé est le troisième volet des exposés concernant la programmation orientée aspect. Ces exposés ont été réalisés en 2005-2006 au cours de la troisième année Ingénieurs 2000 de l'université de Marne La Vallée, filière informatique et réseau.

Ce document n'a pas pour objectif de présenter les tenants et les aboutissants de la programmation orientée aspect, ces points ayant été traités dans les exposés précédents. Cependant, pour une meilleurs compréhension du lecteur, la première partie rappellera les principes de base de la programmation orientée aspect. Le glossaire permettra de rappeler les différents termes utilisés tout au long de ce discours.

Contrairement aux autres exposés, nous nous intéresserons ici à la programmation orientée aspect avec l'outil AspectJ. Après une rapide vision de l'outil, ses avantages et inconvénients, nous verrons comment programmer en aspect avec celui-ci. Ainsi, le chapitre II s'attardera sur la manière de faire des coupes génériques, comment paramétrer des coupes et faire de l'introspection à l'intérieur de celles-ci. Nous terminerons ce chapitre avec la description et des exemples du mécanisme d'introduction. Avec ces éléments nous serons capables de programmer des Design Patterns à l'aide d'aspects. La troisième partie présentera la programmation de 2 Design Patterns : singleton et observateur.

Table des matières

Introduction..... 2

I. Programmation orientée aspect et AspectJ..... 4

1.Qu'est ce que la programmation orientée aspect ?..... 4
a) Pourquoi l'aspect ?..... 4
b) Comment et avec quoi faire de l'aspect ?..... 4

2.Qu'est ce que AspectJ ?..... 5
a) Origine..... 5
b) Différences avec les autres outils..... 5
c) Comment installer AspectJ avec Eclipse..... 6

II. Programmation avec AspectJ..... 7

1.Premier aspect : coupes et code advice simple..... 7
a) Définir un aspect..... 7
b) Coupes simples..... 8
c) Coupes génériques : wildcards..... 10
d) Opérateurs logiques et mot-clef de filtrage..... 11
e) Code Advice..... 13

2.Introspection et paramétrage des coupes..... 16
a) Introspection..... 16
b) Paramétrage des coupes..... 18

3.Mécanisme d'introduction..... 20

III. Implémentation de patrons de conception..... 23

1.Pourquoi implémenter des design patterns en aspect..... 23

2.Design pattern singleton..... 23
a) Design pattern singleton et son implémentation standard..... 23
b) Implémentation simple d'un singleton en aspect..... 24
c) Avantages et inconvénients de l'implémentation aspect..... 26

3. Design Pattern Observateur..... 27
a) Design pattern observateur et son implémentation standard..... 27
b) Implémentation en aspect..... 31
c) Réflexion sur l'implémentation aspect..... 34

I. Conclusion..... 35

Références..... 36

Tables des codes source et illustrations..... 37

Glossaire..... 38

I. Programmation orientée aspect et AspectJ

1. Qu'est ce que la programmation orientée aspect ?

a) Pourquoi l'aspect ?

Les techniques de conception logicielles actuelles tentent d'architecturer les applications en modules a priori indépendants les uns des autres car gérant des aspects différents du système conçu. C'est le principe même de la programmation orientée objet ou le logiciel est découpé en unité de sens : les objets.

Dans la pratique cependant, on s'aperçoit que ces couches logicielles sont en fait intimement liées : c'est l'entrecroisement des aspects techniques. Ainsi, une couche logicielle initialement dédiée à gérer la logique métier applicative (par exemple un système bancaire), va se retrouver dépendante de modules gérant les aspects transactionnels, journalisation, etc. Se limiter aux méthodes de programmation 'classiques' conduit ainsi à une complexification du code, de son développement et de sa maintenance. Il a donc été introduit un nouveau concept permettant de gérer les aspects transversaux d'une application : la programmation orientée aspect.

L'inversion de contrôle mise en œuvre par la programmation par aspect permet d'extraire les dépendances entre objets/modules concernant des aspects techniques entrecroisés. Ces dépendances sont gérées depuis l'extérieur de ces modules. Elles sont spécifiées dans des composants du logiciel nommés aspects.

b) Comment et avec quoi faire de l'aspect ?

La programmation orientée aspect ne tend pas à remplacer la programmation orientée objet mais bien à la compléter sur les éléments transversaux des logiciels. Il est cependant à noter que la programmation orientée aspect est un paradigme de programmation qui peut se greffer sur autre chose que de la programmation orientée objet. Il est possible de faire de la programmation avec tout type de langage, il faut cependant posséder un tisseur d'aspects pour ce langage. Un tisseur d'aspect permet d'injecter les codes liés aux aspects (codes advice) dans le code de base du programme

au niveau des points de jonction.

Voici une liste de quelques tisseurs d'aspects existant pour différents langages

C : Aspect-C
C++ : Aspect-C ++.
PHP : phpAspect
En C#, VB.NET : AspectDNG
Caml : Aspectual Caml
Java : AspectJ, JAC, JBoss AOP

2. Qu'est ce que AspectJ ?

a) Origine

Comme mentionné ci-dessus, AspectJ est un tisseur d'aspect pour le langage Java. Il est en réalité la solution historique de la POA Cette solution qui fut la première, a l'avantage de présenter d'excellentes performances et d'intégrer toutes les possibilités de la POA. L'inconvénient d'AspectJ est qu'il est purement statique (voir tisseur statique et dynamique) et de s'écarter du standard Java en nécessitant un compilateur spécifique.

La première version du compilateur date de 1998 mais la première version de AspectJ date de 2001. AspectJ s'est joint à la communauté open source Eclipse en 2002. Il est ainsi devenu un plugin standard pour la plate-forme de développement Eclipse. En 2005, AspectWerks (un ancien concurrent) a rejoint le projet enrichissant encore l'outil qui en est ainsi à sa 5ème version.

b) Différences avec les autres outils

Contrairement à ses concurrents Jboss AOP et JAC, AspectJ n'est pas un framework de travail mais est basé sur des extensions au langage java. AspectJ ajoute ainsi des mots-clefs au langage : aspect, pointcut, call, execution, after, around...

c) Comment installer AspectJ avec Eclipse

Pour programmer en aspect avec AspectJ sous eclipse, il faut commencer par installer éclipse. Si vous ne l'avez pas encore, la dernière version d'éclipse peut se télécharger sur http://www.eclipse.org/downloads/.

Une fois installé, il est possible d'ajouter AspectJ directement depuis éclipse. Pour cela, allez dans le menu Help>Software Updates>Find And Install... Une nouvelle fenêtre s'ouvre. Pour cherchez le plugin choisissez « Search for new features to install » et cliquez sur "Next". Dans la fenêtre de dialogue d'installation, appuyez sur le bouton « New Remote Site... » Vous pouvez alors saisir le site ayant les propriétés suivantes :

Nom : « AJDT Update Site »

URL : « http://download.eclipse.org/technology/ajdt/31/dev/update » (pour eclipse 3.1 et supérieur).

Appuyez maintenant sur « Finish » Un arbre avec les sites et les installations possibles apparaît. Déployez le noeud « AJDT Update Site » et cochez « AspectJ ». Appuyez sur « Next ». Une fenêtre de fonctionnalité apparaît depuis le site de mise à jour. Choisissez « Eclipse AspectJ Development Tools » et pressez « Next ». Vous entrez dans l'installation réelle de AspectJ. Suivez les différents écran de licences et complétez l'installation.

Remarque : Si vous êtes derrière un proxy, vous ne pourrez accéder aux mises à jour que si vous avez configuré les informations de proxy. Pour cela utilisez le menu d'eclipse « Windows » > « Preferences ». Dans l'arborescence choisissez « Install/Update » et eidtez l'encadré « Proxy settings ».

Une fois AspectJ installé, de nouveaux menus sont accessibles depuis Eclipse (new AspectJ Project, new Aspect, run as AspectJ application...). Vous êtes maintenant prêt à faire vos premières applications en programmation orientée aspect.

II. Programmation avec AspectJ

Une fois AspectJ installé il est possible de créer un nouveau projet aspect grâce à l'icône  ou au menu File > new >AspectJ project. Il est aussi possible de convertir un projet java classique en projet AspectJ avec un clic droit sur le projet, menu new >AspectJ Project.

1. Premier aspect : coupes et codes advice simples

a) Définir un aspect

Dans votre projet vous pouvez créer un nouvel aspect (new > Aspect). Définir un nouvel aspect est très semblable à créer une nouvelle classe. Le mot-clef class est juste remplacé par aspect. De base, l'aspect est vide et ne définit pas un nouveau comportement. Dans un aspect nous allons pouvoir définir des coupes et des codes advice qui induiront les fonctionnalités liées à cet aspect. Un aspect qui se charge d'afficher « Hello Word ! » lorsque la fonction main est appelée est le suivant :

```
public aspect HelloWorld {
	// Coupe :
	pointcut mainCall() :
		call(public static void main(String[] args));

	// Code Advice :
	before() : mainCall() {
		System.out.println("Hello world!");
	}
}
```

Figure 1a : HelloWorld.aj

La figure 1a définit une coupe sur le point de jonction d'appel de la méthode main. Nous verrons dans le paragraphe suivant comment définir une coupe. L'aspect HelloWord.aj définit également le code à exécuter lorsque le programme rencontre un point de jonction pris en compte par la coupe (ici la méthode main).

Ainsi, la classe Main contenant la méthode main se limite à ceci :

```
public class MainHelloWord {
      // Point de jonction (joinpoint) :
      public static void main(String[] args){
      }
}
```

Figure 1b : MainHelloWord.java

Au moment de la compilation, le tisseur d'aspects ajoutera le comportement défini dans le code advice HelloWorld.aj au moment de l'appel de la méthode main.

Il est à noter qu'un aspect tout comme une classe peut être défini comme abstrait (mot clé abstract) ou étendre un aspect existant (mot clé extends). Les principes sont les mêmes que pour une classe. De même une coupe peut être défini comme abstrait. Il s'agit alors d'aspects un peu plus avancés et ils ne seront donc pas définis à ce niveau. Des exemples d'aspect/coupe abstrait sont disponibles dans l'implémentation des design patterns.

b) Coupes simples

La figure 1 présentait un exemple simple de coupe mais souvent utilisé.

Une coupe est introduite grâce au mot clé pointcut. La syntaxe est :

```
pointcut nomDeLaCoupe(paramètres) :
           définitionDeLaCoupe ;
```

Figure 2 : Syntaxe pour définir une nouvelle coupe

Les paramètres seront décrits dans la partie 2 de ce chapitre. Nous nous attarderons ici sur la définition des coupes simples.

Une coupe permet de regrouper un ou plusieurs points de jonction. Dans ce paragraphe nous verrons comme mettre un seul point de jonction. Nous verrons dans le suivant comment généraliser des coupes et regrouper des points de jonction.

Pour savoir quelle syntaxe de définition de coupe utiliser, il faut savoir sur quelle sorte de point de jonction on souhaite faire une coupe pour exécuter du code. Un point de jonction peut être une méthode. Le mot clé alors utilisé pour définir la coupe peut être call ou execution. Cela dépend si l'on souhaite exécuter le code advice lors de l'appel ou dans le

code de la méthode exécutée. Pour résumer la différence entre call et exécution, voici l'ordre d'exécution des codes advice liés :

1 - Code Advice de type « before » associé au point de jonction call

2 - Code Advice de type « before » associé au point de jonction Exécution

3 - Code de la méthode

4 - Code Advice de type « after » associé au point de jonction call

5 - Code Advice de type « after » associé au point de jonction execution

(Les différents types de codes advice seront décrits dans la partie code advice de ce même chapitre).

Outre les méthodes, un point de jonction peut concerner une lecture ou une écriture d'un attribut de classe, la récupération d'une exception, l'exécution d'un constructeur hérité, l'exécution d'un bloc de codes statiques particulier ou encore l'exécution d'un code advice.

Selon le type de point de jonction, le mot clé à utiliser pour définir la coupe est différent. La syntaxe est résumée dans le tableau suivant (extrait du livre *Programmation orientée aspect pour Java / J2EE* de Renaud Pawlak, Jean-Philippe Retaillé et Lionel Seinturier, édition Eyrolles, 2004) :

Syntaxe	Description du point de jonction
call(methodeExpression)	Appel d'une méthode dont le nom vérifie methodeExpression.
execution(methodeExpression)	Exécution d'une méthode dont le nom vérifie methodeExpression.
get(attributExpression)	Lecture d'un attribut dont le nom vérifie attributExpression. Exemple : **get**(int Point.x)
set(attributExpression)	Ecriture d'un attribut dont le nom vérifie attributExpression.
handler(exceptionExpression)	Exécution d'un bloc de récupération d'une exception (catch) dont le nom vérifie exceptionExpression Exemple : **handler**(IOException+)
initialization(constanteExpression)	Exécution d'un constructeur de classe dont le nom vérifie constanteExpression Exemple : **initialization**(Customer.**new**(..))

Syntaxe	Description du point de jonction
preinitialization(constanteExpression)	Exécution d'un constructeur hérité dont le nom vérifie constanteExpression
staticinitialization(classeExpression)	Exécution d'un bloc de code static dans une classe dont le nom vérifie classeExpression Exemple : **staticinitialization**(Point)
adviceexecution()	Exécution d'un code advice

Figure 3 : Tableau récapitulatif des points de jonction possibles et la syntaxe à utiliser pour définir la coupe

Remarque : Il est à noter que les xxExpression des exemples de la figure 3 utilisent des caractères spéciaux appelés wildcards. Ceux-ci seront décrits dans le paragraphe suivant.

Nous pouvons voir que les mots-clefs utilisent des paramètres. Il faut maintenant comprendre la forme de ceux-ci. Par exemple, le mot-clef call prend le paramètre « methodeExpression » celui-ci permet de décrire une méthode située dans une classe. Pour définir à quelle méthode s'applique le call, il est possible d'écrire la signature complète de la méthode.
Exemple :

```
pointcut getXValue(): // Coupe
        call(public int Point.getX());
```

La visibilité de la méthode (public/private/protected/paackage) peut être omise. Ce type de syntaxe où toute la signature est décrite permet de ne pointer que sur un seul point de jonction.

Nous allons maintenant voir comment généraliser les coupes.

c) Coupes génériques : wildcards

Une première manière consiste à utiliser des wildcards. Des wildcards sont des caractères permettant de définir de manière plus vaste une méthode ou une classe, etc... Dans la figure 1a nous avions par exemple utilisé le caractère « * » afin de ne pas préciser la classe contenant la méthode main. Les wildcards permettent d'obtenir ce que

l'on pourrait appeler des expressions régulières simples.

Les wildcards existants sont récapitulés dans le tableau ci-joint.

Wildcard	Utilisation
*	Remplace un nom (de classe, de paquetage, de méthode, d'attribut, etc..) ou simplement une partie de nom. Il peut aussi remplacer un type de retour ou un paramètre . Il signifie « n'importe quel nom » ou « n'importe quel type ». Exemple pour une expression sur une méthode : `public * fr.umlv.*.test.*.start*(int, String, *)`
..	Utilisé pour omettre les paramètres des méthodes ou le chemin complet des paquetages Exemple pour une expression sur une méthode : `public void fr..Test.SetParams(..)` Cette exemple signifie « toutes les méthodes publiques SetParam quel que soient leurs paramètres, retournant void et situées dans des classes Test situées dans n'importe quel sous paquetage de fr ».
+	Permet de définir n'importe quel sous-type d'une classe ou d'une interface Exemple pour une expression sur une méthode : `void fr.umlv.test.IMouseListener+.set* (..) ;` Cet exemple signifie « toutes les méthodes commençant par set des classes implémentant l'interface IMouseListener. »

Figure 4 : Liste des wildcard et leur signification

Ainsi les wildcards permettent de généraliser des coupes. En effet, l'expression suivante permet par exemple de réunir tous les points de jonctions de type appel à une méthode commençant par set :

```
pointcut allSet() : call(* *..set* (..) );
```

d) Opérateurs logiques et mot-clef de filtrage

Il est également possible de généraliser des coupes via des opérateurs logiques. En effet, il est possible de combiner des points de jonction avec l'opérateur logique ou (||).

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

Figure 5 : Exemple de combinaison de point de jonction avec le ou logique.

Le point de coupure ci-dessus permet de joindre le point de jonction défini par les 2 call. Il est également possible de faire du filtrage grâce à des mots-clefs spécifiques. Ces mots-clefs permettent de préciser les points de jonction selon l'appelant, l'appelé, le flot de contrôle ou encore les arguments de fonction. Associé à des opérateurs logiques, ces mots clefs permettent de définir de manière très précise les points de jonction.

En résumé les opérateurs logiques et les mots-clefs existants sont les suivants :

Mot-clef	Signification
&&	Et logique
||	Ou logique
!	Négation logique
if(ExpressionBooléenne)	Evaluation de l'expression booléenne ExpressionBooléenne. Exemple : **pointcut** test(): **if** (**thisJoinPoint**.getArgs().length == 1) **&&execution**(* *.*(..));
withincode(methodeExpression)	Vrai lorsque le point de jonction est défini dans une méthode dont la signature vérifie methodeExpression
within(typeExpression)	Vrai si l'événement se passe pendant l'exécution d'un code embarqué par une classe définie par typeExpression (cela inclut les méthodes statiques). Permet essentiellement de limiter la portée d'un autre pointcut à une certaine classe.
this(typeExpression)	Vrai lorsque l'événement se passe à l'intérieur d'une instance de classe de type typeExpression
target(typeExpression)	Vrai lorsque le type de l'objet destination du point de jonction vérifie typeExpression
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)

Figure 6 : Opérateurs logiques, mots-clefs de filtrage et leur signification

Par exemple, l'expression suivante désigne tous les appels à la méthode setX de Point depuis la classe nommée MainExemple.

call(* fr.univ.mlv.Point.setX(..)) **&& this**(fr.univ.mlv.MainExemple)

Cela permet d'exclure de la coupe les appels à la méthode effectués depuis les autres

classes.

De même on peut désigner les appels depuis un code de méthode particulier

get(fr.univ.mlv.Point.x) **&&** !**withincode**(* fr.univ.mlv.Point.getX())

Dans cet exemple nous désignons tous les endroits où le paramètre x est le lu en dehors de la méthode getX. Il est également possible de définir toutes les lectures de x à l'extérieur de la classe Point grâce au mot-clef within.

get(fr.univ.mlv.Point.x) **&&** !**within**(fr.univ.mlv.Point)

On peut alors se demander quelle est la différence entre this et within. En réalité, "this"permet de capturer une classe précise et toutes les classes qui en dérivent : il s'agit d'une capture sémantique. Le mot-clef "within" a, quant à lui, une signification purement lexicale. La portée de cette primitive de coupe est l'ensemble des méthodes (incluant les méthodes statiques) des classes qui respectent l'expression régulière passée en paramètre de la primitive. Cependant, cela n'inclut pas les classes qui en dérivent.

Les coupes utilisant les mots-clefs cités ci-dessus peuvent également être paramétrées grâce à ces mots-clefs. Cette utilisation des mots-clefs sera décrite dans la partie nommée « Introspection et paramétrage des coupes ».

e) Code Advice

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

Le mot-clef **before** permet d'exécuter du code avant d'entrer dans le code lié à l'événement (exemple appel de méthode, lecture d'attribut etc...). Le mot-clef **after** permet quant à lui d'exécuter du code après l'événement. Le mot-clef **around** permet soit de remplacer l'événement en lui-même, soit d'exécuter du code avant et après le point de jonction. Pour exécuter le code de l'événement il faut utiliser le mot-clef **proceed** à l'intérieur du code advice. Si la méthode a un type de retour, alors il faut faire précéder le mot-clef around de Object. L'appel à proceed renvoie alors un type Object qu'il est possible de récupérer pour le renvoyer. Voici un exemple très simple permettant de tracer l'exécution de la méthode getX.

```
pointcut getXValue(): execution(int *..Point.getX());
Object around() : getXValue(){
      System.out.println("=> Entrée dans getX");
      Object ret =  proceed();
      System.out.println("<= Sortie de la méthode getX");
      return ret;
}
```

Figure 7 : exemple de code around

Dans de telles conditions, nous pouvons voir qu'il est très simple de remplacer la valeur de retour de la fonction ou même de ne pas appeler la fonction. Si le proceed n'est pas appelé alors le code destination n'est jamais exécuté. Il y a alors beaucoup d'applications possibles. Nous pouvons par exemple facilement imaginer un aspect permettant de contrôler les appels à une méthode. Il est en outre possible de vérifier que l'utilisateur désirant accéder à une méthode en a le droit, et si celui-ci ne l'a pas alors la méthode n'est jamais exécutée par proceed et une exception est levée. L'inconvénient cependant est le déboguage de l'application. Imaginons que le programmeur oublie l'appel à proceed. Dans le code basique de l'application l'erreur est indécelable. C'est pour cette raison que le plugin eclipse AspectJ permet de visualiser le tissage effectué. Lorsque l'on définit un aspect, l'interface graphique d'éclipse permet de savoir quels sont les points de jonction liés à un code Advice et vice-versa. Par exemple, dans le code Advice suivant, on peut visualiser une petite flèche dans la marge. Le hint associé indique « 2 AspectJ markers at this line »

```
pointcut getXValue(): call(int *..Point.getX());

before() : getXValue() {
    System.out.println("getting X attribute");
}
```

Figure 8a : Interface graphique, indication de l'existence des points de jonctions

Nous pouvons visualiser ces points de jonction dans la fenêtre Cross References :

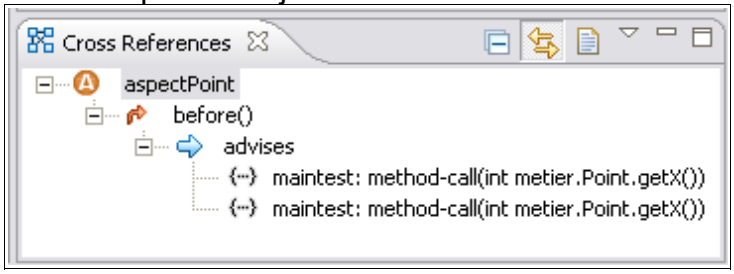


Figure 8b : Interface graphique, visualisation des points de jonctions

Un double clic permet d'aller directement à ces points de jonction :

```
public class maintest {
    public static void main(String[] args) {
        Point p = new Point(4,5);
        metier.arbo.Point p2 = new metier.arbo.Point(4,5);

        p.getX();
        p2.getX();
    }
}
```

Figure 8c : Interface graphique, indication au niveau des points de jonctions

Une fois positionné sur ce code, il est possible de voir dans la fenêtre Cross Réferences la liste des codes advice liés :

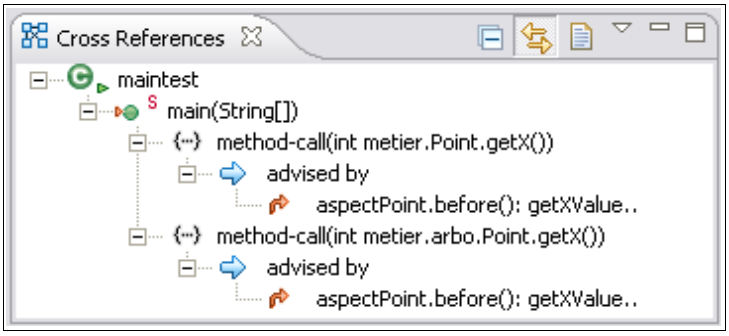


Figure 8d : Interface graphique, visualisation des codes advice liés

Ainsi, depuis le code objet de l'application il est possible de voir le lien avec les aspects. Si un bogue se produit, le développeur sait que l'erreur peut provenir du code de base ou des aspects qui sont indiqués dans la fenêtre Cross References.

Lors du développement des aspects le programmeur doit toujours faire attention aux endroits où sera exécuté le code advice. Reprenons l'exemple précédent du around. Imaginons qu'à la place du proceed, le code soit le suivant :

```
pointcut getXValue(): execution(int *..Point.getX());
Object around() : getXValue(){
      System.out.println("=> Entrée dans getX");
      // Object ret =  proceed();
      Object ret = ((Point)(thisJoinPoint.getTarget())).getX();
      System.out.println("<= Sortie de la méthode getX");
      return ret;
}
```

Figure 9 : Exemple de boucle infini sur un aspect

Le mot-clef **thisJoinPoint** sera étudié plus tard. Sachez que thisJoinPoint.getTarget() permet de récupérer l'instance de Point sur laquelle est appelée la méthode getX. Un tel

code entraîne une boucle infinie car ((Point)(**thisJoinPoint**.getTarget())).getX() est pris en compte par la coupe getXValue. Ce type d'appel récursif est d'autant plus difficile à détecter que les points de jointure à l'intérieur des aspects ne sont pas repérables par l'interface graphique Eclipse.

Ici l'appel à getX à la place de proceed correspond effectivement à une erreur. Nous pouvons cependant imaginer des cas où cela n'est pas le cas. Il est alors possible de ne pas boucler grâce à un opérateur logique de filtrage :

```
pointcut println(): execution(void *..println(String))
      && !this(aspectPoint); // Filtre pour éviter la boucle
void around() : println(){
      //log
      logger.append(thisJoinPoint.getArgs()[0]);
      // Appel toujours le System.out
      System.out.println(thisJoinPoint.getArgs()[0]);
}
```

Figure 10 : Exemple de filtrage afin d'éviter une boucle infini

La coupe ne s'appliquera pas dans le cas d'appel à l'intérieur de l'aspect. Ici encore nous utilisons le mot-clef thisJoinPoint afin de récupérer les informations sur les arguments. Ce mécanisme s'appelle l'introspection.

2. Introspection et paramétrage des coupes

a) Introspection

L'introspection permet de connaître dans un code advice quelles sont les informations sur le point de jonction ayant provoqué l'événement. Le mot-clef **thisJoinPoint** utilisé dans un code advice renvoie un objet de la classe org.aspectj.lang.JoinPoint

Mot-clef	Signification
Object[] **getArgs**()	Dans le cas d'un point de jonction concernant une méthode, getArgs retourne les arguments de l'appel.
String **getKind**()	Retourne une chaîne de caractères représentant le type du point de jonction.
Signature **getSignature**()	Retourne la signature d'un point de jonction. Signature est une interface. Dans le cas d'une

Mot-clef	Signification
	signature de type méthode, Signature fournit des méthodes pour récupérer son nom, sa visibilité, sa classe, son type de retour etc...
SourceLocation **getSourceLocation**()	Retourne la localisation du point de jonction dans le code source. SourceLocation est une interface permettant de connaître le nom du fichier, le numéro de ligne et la classe définissant le point de jonction. Remarque : même objet de retour que getStaticPart().getSourceLocation()
JoinPoint.StaticPart **getStaticPart**()	Retourne un objet encapsulant les informations statiques concernant ce point de jonction.
Object **getTarget**()	Dans le cas d'un point de jonction concernant une méthode renvoie l'instance de l'objet appelé.
Object **getThis**()	Retourne l'objet appelant, c'est-à-dire l'objet en cours d'exécution lorsque le code advice a été appelé.
String **toLongString**()	Retourne une chaîne de caractères représentant une description complète du point de jonction.
String **toShortString**()	Retourne une chaîne décrivant succinctement le point de jonction.

Figure 11 : Méthodes principales de la class JoinPoint

Le mot-clef thisJoinPointStaticPart est quant à lui équivalent à thisJoinPoint.getStaticPart(). Ces méthodes permettent de récupérer les éléments nécessaires au traitement. Imaginons par exemple que l'on souhaite récupérer les éléments appelant et appelé lors d'un point de jonction de type call. Nous pouvons utiliser le code suivant :

```
import mygraphic.Point;
public aspect aspectPoint {
      pointcut getXValue(): call(int *..Point.getX())
                        && !this(aspectPoint);
      after() : getXValue(){
            Object      src         =  thisJoinPoint.getThis();
            Object      dst         =  thisJoinPoint.getTarget();
            Point       castedDst   = (Point)dst;
            MainTest    castedSrc   = (MainTest)src;
            System.out.println(castedSrc+" has call Point : "+
                        castedDst.getX()+"."+castedDst.getY());
      }
}
```

Figure 12a : Récupération des éléments appelant et appelé avec thisJoinPoint

Ainsi, afin de pouvoir atteindre les méthodes spécifiques aux classes appelante et appelée, nous sommes obligés de faire un cast. Ici si la méthode Point est appelé depuis un autre lieu que le main alors le code lèvera une exception au moment de faire le cast. De même s'il existe un autre paquetage que mygraphic qui contient la classe Point, alors l'appel de getX sur cette autre classe provoquera une exception car l'aspect ne pourra pas faire un cast de cette classe vers mygraphic.Point.

Pour pallier ce problème, il est possible de préciser les paramètres de la coupe. Imaginons que dans le code précédent nous ayons défini la coupe comme suit :

```
      pointcut getXValue(): call(int mygraphic.Point.getX()) &&
                                 this(MainTest) &&
                                 target(mygraphic.Point);
```

Figure 12b : Précision de la coupe afin d'assurer le cast des éléments appelant et appelé récupérés avec thisJoinPoint

Avec cette méthode nous assurons que le cast suivant pourra se faire. Cependant nous pouvons remarquer que le processus de vérification du type this et target est vérifié une fois au niveau de la coupe et une fois au niveau du code advice. Toujours dans l'optique de précision de la coupe, il existe une autre manière de faire ce type d'introspection : le paramétrage de coupe.

b) Paramétrage des coupes

Il est possible de passer des paramètres à l'aide des primitives de filtrage défini dans le

tableau de la figure 6 (args, this, target, within,...). Ainsi, en modifiant la coupe de la figure 12, nous pouvons paramétrer la coupe getXValue comme suit :

```
pointcut getXValue(MainTest src, mygraphic.Point dst):
            call(int mygraphic.Point.getX()) &&
            this(src) &&
            target(dst);
```

Figure 13a : Paramétrage de coupe, récupération des éléments appelant et appelé

Ceci a 2 effets, le premier est que la coupe vérifiera que l'appelant est bien de type MainTest et l'appelé de la classe mygraphic.Point. C'est une opération de filtrage. Le deuxième effet, si l'événement respecte les conditions de filtrage, est que la coupe mettra dans src l'appelant et dans dst l'élément appelé. Ces éléments seront alors disponibles à l'intérieur du ou des codes advice éventuellement liés. La syntaxe du code advice doit donc également être modifiée. Le code de celui-ci est très nettement simplifié car il n'y a plus besoin de faire appel à thisJoinPoint pour obtenir les informations :

```
Object around(MainTest src, mygraphic.Point dst) : getXValue(src,dst){
      System.out.println("=> "+src+" has call Point : "+
                         dst.getX()+"."+dst.getY());
      Object ret = proceed(src, dst);
      System.out.println("<= "+src+" has call and return Point "+ret);
      return ret ;
}
```

Figure 13b : Récupération des paramètres de la coupe dans le code advice

Le code utilise volontairement un code advice de type around afin de montrer que l'appel à proceed nécessite les paramètres de la coupe. Dans ce code nous pouvons voir que les cast ne sont plus nécessaires, il n'y a donc plus aucun risque d'exception.

Nous pouvons noter ici encore la différence fondamentale entre **execution** et **call**. Dans le cas d'un call l'appelant est bien MainTest, c'est a dire la classe appelant la méthode. Dans le cas de la primitive execution, le point de jointure se fait dans le code d'exécution de la méthode getX de Point. Le mot-clef **this** testera donc une instance de type Point et non MainTest. Il est également important de faire attention au fait que nous parlons ici d'instance appelante et appelée. Si getX est appelée depuis la méthode main de MainTest, alors la condition n'est pas vérifiée. En effet, la méthode main est **static**. Elle n'appartient à aucune instance et **thisJointPoint.getThis()** retourne donc **null**.

Si les paramètres d'une coupe sont souvent l'appelant et/ou l'appelé, il est également fréquent de récupérer les arguments de la méthode appelée. La coupe utilise alors le mot-clef args afin de récupérer l'ensemble des arguments.

```
pointcut setXYValue(  MainTest src, mygraphic.Point dst,
                      int valX, int valY ):
      call(int mygraphic.Point.setXY(..)) &&
      this(src) &&
      target(dst) &&
      args(valX, valY);
```

Figure 14 : Paramétrage de coupe, récupération des arguments

Dans un tel cas la coupe vérifiera en plus que les deux premiers paramètres sont bien de type int. Il est à noter que dû à l'autoBoxing apparue avec Java1.5, si la méthode prend en paramètre un type Integer, la coupe s'appliquera. Un comportement plus étrange constaté est que si la méthode setXY prend en paramètre de type short, la coupe s'applique aussi.

Si l'introspection et le paramétrage des coupes permet de récupérer les mêmes informations, le paramétrage de coupe à l'avantage d'être mieux optimisé et d'avoir un code plus élégant et plus robuste.

3. Mécanisme d'introduction

Un aspect peut ajouter des comportements comme de la journalisation à des classes mais il peut également ajouter des attributs, des méthodes, des constructeurs ou encore des interfaces ou une classe parente. Ce mécanisme s'appelle l'introduction et utilise encore une fois avec AspectJ une syntaxe particulière. Imaginons que nous ayons un certain nombre de classes pour lesquelles nous souhaitons pouvoir positionner une date et la récupérer. Nous souhaitons de plus initialiser cette date dans le constructeur. Nous allons alors définir une interface DateInside précisant les méthodes possibles sur ce type de classe (typiquement getDate et setDate). Nous allons ensuite définir un aspect permettant d'ajouter ces fonctions à une classe Class1.
Le code de l'interface est le suivant :

```
package date;
import java.util.Date;

public interface DateInside {
	public Date getDate();
	public void setDate(Date date);
}
```

Le code de l'aspect est :

```
package date;
import java.util.Date;

public aspect IntroDate {
	// Ajout d'une interface à Class1
	declare parents: Class1 implements DateInside;

	// Ajout d'un attrinut date à Class1
	private Date Class1.date;

	// Ajout d'une méthode getDate à Class1
	public Date Class1.getDate(){
		return date;
	}

	// Ajout d'une méthode setDate à Class1
	public void Class1.setDate(Date d){
		date=d;
	}

	// Initialization de la date dans le constructeur
	after(): initialization(Class1.new(..)){
		Class1 o = (Class1)thisJoinPoint.getTarget();
		o.date = new Date();
	}
}
```

Figure 15 : Introduction d'interface, d'attributs et de méthodes...

Il est à noter un problème du plugin Eclipse avec le mécanisme d'introduction. En effet, les méthodes ajoutées ne sont pas visibles depuis l'éditeur de texte éclipse. L'éditeur marque donc une erreur. L'arborescence quant à elle ne marque bien aucune erreur et l'exécution se fait correctement.

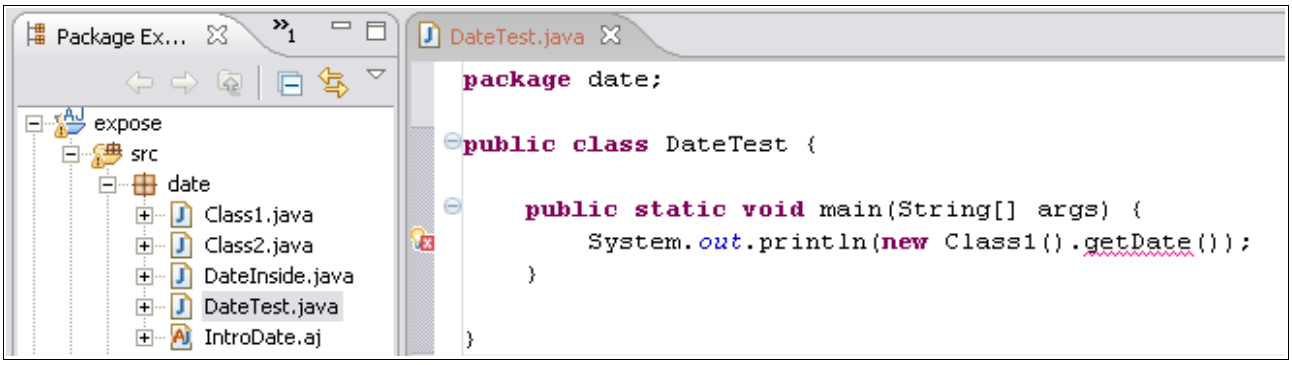
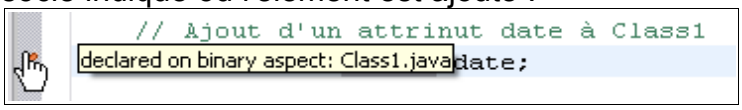


Figure 16 : une mauvaise détection d'Eclipse des méthodes introduite

Notons que lorsqu'on introduit un élément à une classe dans un aspect, une nouvelle flèche apparaît dans la marge au niveau de la ligne de l'aspect où l'on introduit un élément. Le hint associé indique où l'élément est ajouté :



La fenêtre Cross Références permet de suivre l'ensemble des éléments ajoutés dans l'aspect.

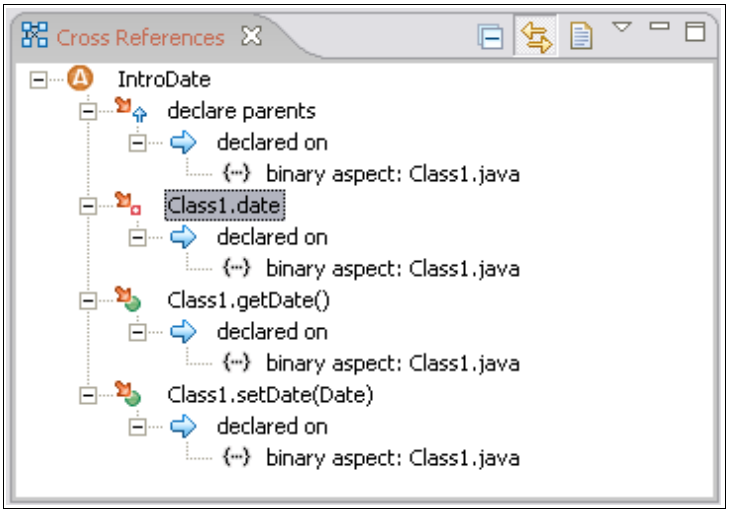


Figure 17 : Suivis des éléments introduits grâce à Eclipse.

Si l'on souhaite ajouter le getDate, setDate à plusieurs classes sans répéter le code, la solution est d'utiliser l'héritage. Une classe abstraite DateInsideImpl définissant le getDate, setDate est écrite. L'aspect se contente d'ajouter DateInsideImpl en classe parente de Class1 (et autre). L'inconvénient est qu'une classe ne peut hériter que d'une seule classe. Ce mécanisme ne fonctionne donc que si la classe n'a pas déjà de classe parente. Une autre solution consiste à définir des aspects abstraits. Un exemple de ce type d'implémentation est la version aspect du design pattern observateur décrit dans la suite de ce document.

III. Implémentation de patrons de conception

1. Pourquoi implémenter des design patterns en aspect

Les design patterns permettent de standardiser des solutions à des problèmes classiques. Ils consistent généralement à ajouter une ou plusieurs responsabilités dans une ou des classes existantes ou non. A partir de cette constatation, pour chaque implémentation d'un design pattern nous pouvons distinguer les classes qui ont des responsabilités en plus de celles rajoutées par le design pattern. Avec l'aspect, il est possible de modulariser ces comportements et responsabilités uniquement liés aux design patterns implémentés.

L'aspect s'adapte particulièrement aux design patterns car il s'agit de méthodes de programmation standards et donc applicables à différentes classes. Une partie du code est propre aux design patterns, et non à la classe qui l'implémente. Les design patterns peuvent ainsi être vus comme des fonctionnalités transversales où le code commun aux différentes classes peut être centralisé dans un aspect.

2. Design pattern singleton

a) Design pattern singleton et son implémentation standard

Le singleton permet de s'assurer qu'une seule instance d'un objet donné sera instanciée pendant toute la durée d'une application. Un singleton consiste donc à n'avoir toujours qu'une seule instance en mémoire, chaque appel de méthode sur un objet de cette classe référence en réalité un appel sur l'unique instance de la classe. La méthode la plus classique de l'implémenter est la suivante:

```java
public class Singleton {
    /**
     * Constructeur redéfini comme étant privé pour interdire
     * son appel et forcer à passer par la méthode getInstance
     */
    private Singleton() {}

    /** L'instance statique */
    private static Singleton instance;

    /**
     * Récupère l'instance unique de la class Singleton.<p>
     * Remarque : le constructeur est rendu inaccessible
     */
    public static Singleton getInstance() {
        if (instance == null) { // Premier appel => Création de l'instance
            instance = new Singleton(); // unique
        }
        return instance;
    }
}
```

Figure 18 : Implémentation classique du design pattern singleton

L'utilisation d'une variable statique garantie l'unicité. Mettre le constructeur en visibilité private empêche l'utilisation du constructeur depuis l'extérieur. Le seul moyen pour récupérer une instance de la classe est donc de passer par la méthode getInstance. Il est à noter alors que les développeurs des classes utilisatrices du singleton savent qu'il s'agit d'un singleton du fait de l'obligation de passer par getInstance pour récupérer un objet.

Ce code d'exemple prouve également que l'implémentation est sensiblement la même d'une classe à une autre. La programmation orientée aspect va permettre de centraliser ce code en un unique aspect.

b) Implémentation simple d'un singleton en aspect

Le code propre au design pattern singleton sera écrit dans un aspect abstrait particulier. Les classes devant être mises en tant que singleton doivent alors avoir un aspect

attaché qui précise que la classe est un singleton. Nous allons profiter de la particularité des codes advice de type around afin de ne pas appeler le constructeur lorsque celui-ci existe.

L'aspect générique est donc le suivant :

```
public abstract aspect AbstractSingletonAspect {
	//Point de coupure abstrait, les classes devant implémenter un singleton doivent avoir un aspect qui défini le point de coupure sur leur constructeur
	abstract pointcut singletonPointcut();

	// Instance unique de la classe devant implémenté le singleton
	private Object singletonUniqueInstance;

	// Ne faire l'appel réel au constructeur que si l'instance n'existe pas déjà
	Object around(): singletonPointcut(){
		if( singletonUniqueInstance == null){
			singletonUniqueInstance = proceed();
		}
		return singletonUniqueInstance;
	}
}
```

Figure 19a : Implémentation aspect simple du design pattern singleton, aspect abstrait

Maintenant, pour faire qu'une classe TestSingleton soit un singleton, il faut ajouter l'aspect concret suivant :

```
public aspect SingletonAspect extends AbstractSingletonAspect {
	pointcut singletonPointcut() : call(TestSingleton.new(..));
}
```

Figure 19b : Implémentation aspect simple du design pattern singleton, aspect concret

Le singleton est testable depuis la fonction main suivante :

```
public static void main(String[] args) {
	TestSingleton s1 = new TestSingleton();
	TestSingleton s2 = new TestSingleton();
	if (s1 == s2){
		System.out.println("Singleton OK");
	}else{
		System.err.println("Singleton Not OK");
	}
}
```

Figure 19c : Test du singleton

A remarquer, il y a une raison pour laquelle nous n'avons pas fait un aspect unique avec une coupe regroupant l'ensemble des singletons. En effet, un aspect est par nature statique. Cela signifie que les classes singletons partageraient l'objet nommé singletonUniqueInstance de l'aspect. Ceci est à prohiber car chaque type de classes doit posséder un unique objet singleton. Les mots-clefs **perthis**(*pointcut*) et **pertarget** (*pointcut)* ne sont d'aucune utilité ici. Perthis permet de définir un aspect par instance de classe appelante et pertarget par classe appelée.

Une autre solution que d'utiliser un type abstrait, est de définir une table de hachage. Mais l'aspect singleton regrouperait en un unique endroit tous les singletons ce qui n'est pas toujours judicieux dans une application modulaire.

c) Avantages et inconvénients de l'implémentation aspect

L'aspect singleton agit de manière intrusive dans la classe transformée. Dans la classe utilisatrice d'une classe singleton (figure 19c), nous pouvons remarquer que la classe utilisatrice ignore qu'elle travaille avec un singleton. Le coté positif est au niveau de la transparence et du polymorphisme. La classe cliente ignore l'implémentation de ce avec quoi elle travaille (transparence) et de plus elle utilise un singleton de la même manière que n'importe quelle autre classe (polymorphisme). Le risque est cependant de faire des erreurs de manipulation du fait de cette ignorance. Être un singleton est une caractéristique forte pour une classe, il est donc discutable de ne pas le signaler dans la classe elle-même mais dans un élément ajouté tel qu'un aspect...

Il est à noter aussi que l'implémentation aspect du singleton fourni ici n'est pas parfaite. Il peut par exemple être discutable le fait de renvoyer toujours la même instance quel que soit le constructeur appelé. C'est normalement le cas de la majorité des singletons. Cependant dans le cas où il faut distinguer l'instance retournée selon le constructeur appelé, il sera nécessaire d'utiliser une table de hachage et donc une autre implémentation aspect du singleton.

3. Design Pattern Observateur

a) Design pattern observateur et son implémentation standard

Le design pattern observateur (observer en anglais) s'utilise dans différents cas comme:

• Une abstraction à plusieurs aspects dépendant l'un de l'autre et que l'on souhaite encapsuler ces aspects indépendamment et les réutiliser séparément.

• Quand le changement d'un objet se répercute vers d'autres.

• Quand un objet doit prévenir d'autres objets sans pour autant les connaître.

Le design pattern est notamment utilisé en interface graphique pour le design pattern Modèle Vue Contrôleur. Lorsque le modèle de donnée change, celui-ci doit prévenir les vues qui se sont enregistrées.

Par exemple, nous souhaitons qu'un bouton prévienne un ensemble d'objets lorsque celui-ci est cliqué. Une implémentation standard du design pattern est alors la suivante :

```
public interface MyButtonListener {
        public void onButtonClick();
}
```

Figure 20a : Implémentation standard du patron observateur, exemple d'interface pour des observateurs

Le sujet qui doit prévenir ses observateur de type MyButtonListener est le suivant :

```
public class MyButtonSubject extends JButton {
	private List <MyButtonListener> objectToWarn =
			new ArrayList<MyButtonListener>();
	public void addObserver(MyButtonListener listener){
		objectToWarn.add(listener);
	}
	public void removedObserver(MyButtonListener listener){
		objectToWarn.remove(listener);
	}

	private void fireOnClick(){
		for (MyButtonListener listener : objectToWarn) {
			listener.onButtonClick();
		}
	}
	public MyButtonSubject(String arg0) {
		super(arg0);
		addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				MyButtonSubject.this.click();
			}
		});
	}
	public void click() {
		fireOnClick();
	}
}
```

Figure 20b : Implémentation standard du patron observateur, exemple de sujet observé

Une classe souhaitant observer le sujet doit alors répondre à l'interface :

```
public class LabelObserver       extends JLabel
                                 implements MyButtonListener {

      /**
       * Stock l'objet observé afin de lire des éventuelles données à
       * l'interieur
       */
      private MyButtonSubject subject;
      /**
       * positionne l'objet observé
       */
      void setSubject(MyButtonSubject subjectToObserve){
            if (this.subject != null)
                  this.subject.removedObserver(this);
            this.subject = subjectToObserve;
            // Enregistrement auprès du modèle
            this.subject.addObserver(this);
      }
      /**
       * Récupère l'objet observé
       */
      MyButtonSubject getSubject(){
            return this.subject;

      }

      /**
       * Implémentation de l'interface
       */
      public void onButtonClick() {
            //possibilité de lire des informations au sein
            //de l'objet subject
            colorCycle();
      }

      final static Color[] colors = {Color.red, Color.blue, Color.green,
                                     Color.magenta};
      private int colorIndex = 0;
      private int cycleCount = 0;
      void colorCycle() {
            cycleCount++;
            colorIndex = (colorIndex + 1) % colors.length;
            this.setBackground(colors[colorIndex]);
            setText("" + cycleCount);
      }
      public LabelObserver() {
            super("0");
            this.setOpaque(true);
            this.setBackground(Color.red);
      }
}
```

Figure 20c : Implémentation standard du patron observateur, exemple d'observateur

Il est alors possible de tester avec par exemple la classe suivante :

```
public class TestObserver {

      public static void main(String[] args) {
            JFrame frame = new JFrame("Subject/Observer Demo");
            frame.addWindowListener(new WindowAdapter() {
                  public void windowClosing(WindowEvent e){
                              System.exit(0);
                  }
            });

            MyButtonSubject b1    = new MyButtonSubject("Bouton 1");
            MyButtonSubject b2    = new MyButtonSubject("Bouton 2");
            LabelObserver c1      = new LabelObserver();
            LabelObserver c2      = new LabelObserver();
            LabelObserver c3      = new LabelObserver();

            c1.setSubject(b1); // appel de ce fait b1.addObserver(c1)
            c2.setSubject(b2); // appel de ce fait b2.addObserver(c2)
            c3.setSubject(b2); // appel de ce fait b2.addObserver(c3)

            frame.getContentPane().setLayout(new FlowLayout());
            frame.getContentPane().add(b1);
            frame.getContentPane().add(b2);
            frame.getContentPane().add(c1);
            frame.getContentPane().add(c2);
            frame.getContentPane().add(c3);
            frame.pack();
            frame.setVisible(true);
      }
}
```

Figure 20d : Implémentation standard du patron observateur, classe de test

Nous obtenons la fenêtre suivante:

Remarquons qu'il n'est pas nécessaire obligatoirement que l'observateur connaisse son sujet. Ici par exemple le code effectué en cas de changement de sujet ne nécessite pas de lire des propriétés du sujet bouton. Il est donc possible de retirer cette référence. La fonction main au lieu de faire setSubject peut alors se contenter de faire un `b1.addObserver(c1)`.

Dans cette implémentation du design pattern, des codes source sont propres aux responsabilités ajoutées par le design pattern. Par exemple, le sujet observé contient la liste des observateurs et la gestion de ceux ci. Cette implémentation se retrouvera dans

tous les design patterns observateur. La fonction déclenchant l'appel aux observateurs est quant à elle propre à cet objet particulier.

b) Implémentation en aspect

Dans la version aspect de ce design pattern nous allons séparer les éléments en 3 différents types de classe/aspect. Les objets de bases Label et Button seront mis dans des classe classiques. Aucun lien ne sera fait entre eux. Parallèlement nous aurons des interfaces propres au design pattern : une interface subject et une interface observateur. Pour implémenter les parties standards du design pattern, nous utiliserons un aspect abstrait. C'est celui-ci qui contiendra les codes sources liés à tous les sujets et observateurs. Enfin nous créerons des aspects concrets afin de définir chaque lien observateur/observé. Il y aura un aspect concret par couple de classe que l'on souhaite définir comme observateur et observé.

L'implémentation est donc composée tout d'abords des interfaces suivantes :

```
public interface Subject {
    public void addObserver(Observer obs);
    public void removeObserver(Observer obs);
    Collection<Observer> getObservers();
}
```

```
public interface Observer {
    void setSubject(Subject s);
    Subject getSubject();
    void update();
}
```

Figure 21a : Implémentation aspect du patron observateur, interfaces

L'aspect abstrait générique est le suivant :

```
abstract aspect SubjectObserverProtocol {
	// Le point de coupure sera à définir dans l'aspect concret
	abstract pointcut stateChanges(Subject s);
	// Ajout d'une liste des observateurs de l'observé
	private	ArrayList<Observer>	Subject.observers	=	new
ArrayList<Observer>()
	// Lorsque l'observé change, prévenir les observateurs
	after(Subject s): stateChanges(s) {
		for (Observer observer : s.getObservers()) {
			observer.update();
		}
	}
	// Permet d'ajouter un observateur à un objet observé
	public void Subject.addObserver(Observer obs) {
		observers.add(obs);
		obs.setSubject(this);
	}
	// Permet de retiré un observateur à un objet observé
	public void Subject.removeObserver(Observer obs) {
		observers.remove(obs);
		obs.setSubject(null);
	}
	// Permet de récupéré la liste des observateurs
	public ArrayList<Observer> Subject.getObservers() {
		return observers;
	}

	// L'observateur stock l'objet observé
	private Subject Observer.subject = null;
	// Permet de positionné le sujet dans l'observateur
	public void Observer.setSubject(Subject s) {
		subject = s;
	}
	// Permet de récupérer le sujet dans l'observateur
	public Subject Observer.getSubject() {
		return subject;
	}
}
```

Figure 21b : Implémentation aspect du patron observateur, aspect abstrait

Les classes qui deviendront un observateur et un observé restent dépouillées de code lié au design pattern. Leur implémentation est la suivante :

```
public class MyButton extends JButton {
	private static final long serialVersionUID = 1L;

	public MyButton(String arg0) {
		super(arg0);
		addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				MyButton.this.click();
			}
		});
	}
	public void click() {
	}
}
```

```
public class MyLabel extends JLabel{
	private static final long serialVersionUID = 1L;

	final static Color[] colors = {Color.red, Color.blue, Color.green,
Color.magenta};
	private int colorIndex = 0;
	private int cycleCount = 0;
	void colorCycle() {
		cycleCount++;
		colorIndex = (colorIndex + 1) % colors.length;
		this.setBackground(colors[colorIndex]);
		setText("" + cycleCount);
	}
	public MyLabel() {
		super("0");
		this.setOpaque(true);
		this.setBackground(Color.red);
	}
}
```

Figure 21c : Implémentation aspect du patron observateur, classes standards souhaitant à terme être des observé/observateur

Nous devons alors définir MyLabel comme observateur de MyButton dans un aspect concret.

```
aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {
    declare parents: MyButton implements Subject;
    declare parents: MyLabel implements Observer;

    public void MyLabel.update() {
        colorCycle();
    }
    pointcut stateChanges(Subject s):
        target(s) &&
        call(void MyButton.click());
}
```

Figure 21d : Implémentation aspect du patron observateur, aspect permettant de définir le couple observateur/observé pour un couple de classes standards

Ainsi lorsqu'on souhaite ajouter le design pattern à de nouveaux objets, seul l'aspect concret est à développer.

c) Réflexion sur l'implémentation aspect

La solution aspect est bien plus satisfaisante que la solution orientée objet classique. En effet, d'un point de vue de la localisation du code, la classe observé ne contient plus la liste de ces observateurs desquels elle est complètement indépendante. L'essentiel de la gestion du design pattern est géré au niveau de l'aspect abstrait. La spécificité des objets liés à l'observation est concentrée dans les aspects concrets.

Du point de vue de la réutilisation, la situation est encore une fois améliorée par rapport à la solution orientée objet. En effet, l'essentiel de la gestion des observateurs est défini de manière générique dans l'aspect abstrait et peut être réutilisé systématiquement.

Enfin, du point de vue de la composition, le sujet d'observation peut participer sans difficulté à d'autres design patterns car les aspects ne sont pas intrusifs (contrairement au design pattern singleton), le comportement du sujet d'observation n'étant pas modifié.

I. Conclusion

AspectJ permet d'exploiter tous les aspects de la POA. Il permet de faire des coupes très fines grâce à l'ensemble des mots-clefs du langage qu'il introduit. Paramétrage de coupes, introspection et introduction ajoute toute une nouvelle dimension à la programmation : la transversalité. La POA permet un gain de productivité. La relecture et la maintenance du code est également simplifiée car du code source qui était avant dupliqué d'un objet à l'autre peut désormais être écrit dans un aspect. Des aspects génériques peuvent être définis et ainsi réutilisés sur d'autres projets. Le revers de ces possibilités, c'est qu'AspectJ est un langage à part entière à apprendre avec le java. Un programmeur qui s'aventure sur un projet AspectJ peut se retrouver perdu au moment du déboguage s'il ne fait pas attention aux points de greffes des différents aspects. Un autre soucis est lié aux grandes possibilités de AspectJ. A chaque fois qu'un élément ou design pattern peut être un éventuel aspect, il est très important d'analyser les gains et pertes obtenus par une telle implémentation POA. Dan ce document par exemple nous avons vu une implémentation discutable du singletons en POA. S'il est possible d'ajouter des aspects à bien des endroits d'un programme il n'est pas toujours judicieux de le faire.

Références

- Wikipedia (http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_aspect)

- *Programmation orientée aspect pour Java / J2EE* de Renaud Pawlak, Jean-Philippe Retaillé et Lionel Seinturier, édition Eyrolles, 2004

- *Un modèle d'assemblage de composants par contrat et programmation orientée aspect* de Fabrice Legond-Aubry (Thèse de doctorat du "Conservatoire National des Arts et Métiérs - juillet 2005)

Tables des codes source et illustrations

Figure 1a : HelloWorld.aj....7
Figure 1b : MainHelloWord.java 8
Figure 2 : Syntaxe pour définir une nouvelle coupe....8
Figure 3 : Tableau récapitulatif des points de jonction possibles et la syntaxe à utiliser pour définir la coupe.... 10
Figure 4 : Liste des wildcard et leur signification.... 11
Figure 5 : Exemple de combinaison de point de jonction avec le ou logique.... 12
Figure 6 : Opérateurs logiques, mots-clefs de filtrage et leur signification....12
Figure 7 : exemple de code around.... 14
Figure 8a : Interface graphique, indication de l'existence des points de jonctions....14
Figure 8b : Interface graphique, visualisation des points de jonctions.... 14
Figure 8c : Interface graphique, indication au niveau des points de jonctions.... 15
Figure 8d : Interface graphique, visualisation des codes advice liés....15
Figure 9 : Exemple de boucle infini sur un aspect.... 15
Figure 10 : Exemple de filtrage afin d'éviter une boucle infini 16
Figure 11 : Méthodes principales de la class JoinPoint.... 17
Figure 12a : Récupération des éléments appelant et appelé avec thisJoinPoint.... 18
Figure 12b : Précision de la coupe afin d'assurer le cast des éléments appelant et appelé récupérés avec thisJoinPoint....18
Figure 13a : Paramétrage de coupe, récupération des éléments appelant et appelé 19
Figure 13b : Récupération des paramètres de la coupe dans le code advice....19
Figure 14 : Paramétrage de coupe, récupération des arguments....20
Figure 15 : Introduction d'interface, d'attributs et de méthodes....21
Figure 16 : une mauvaise détection d'Eclipse des méthodes introduite....22
Figure 17 : Suivis des éléments introduits grâce à Eclipse....22
Figure 18 : Implémentation classique du design pattern singleton....24
Figure 19a : Implémentation aspect simple du design pattern singleton, aspect abstrait....25
Figure 19b : Implémentation aspect simple du design pattern singleton, aspect concret....25
Figure 19c : Test du singleton 26
Figure 20a : Implémentation standard du patron observateur, exemple d'interface pour des observateurs....27
Figure 20b : Implémentation standard du patron observateur, exemple de sujet observé.... 28
Figure 20c : Implémentation standard du patron observateur, exemple d'observateur.... 29
Figure 20d : Implémentation standard du patron observateur, classe de test.... 30
Figure 21a : Implémentation aspect du patron observateur, interfaces.... 31
Figure 21b : Implémentation aspect du patron observateur, aspect abstrait.... 32
Figure 21c : Implémentation aspect du patron observateur, classes standards souhaitant à terme être des observé/observateur 33
Figure 21d : Implémentation aspect du patron observateur, aspect permettant de définir le couple observateur/observé pour un couple de classes standards....34

Glossaire

- **Aspect** :

un aspect est un module définissant des coupes et/ou codes advice et leurs points d'activation. Avec AspectJ définir un aspect est très semblable à définir une classe. (voir aussi programmation orientée aspect)

- *Exemple :*

```
package metier;

public aspect aspectPoint { // Aspect
      pointcut getXValue(): // Coupe
         call(int Point.getX());

      before() : getXValue() { // CodeAdvice de la coupe « getXValue »
            System.out.println("getting X attribute");
      }
}
```

- **Code advice** (nommé parfois aussi **greffon) :**

Bout de programme qui sera activé lors de l'exécution d'un point de jonction défini par une coupe. Ce bloc de code décrit quel est le comportement d'un aspect.

- *Exemple :*

```
      before() : getXValue() { //CodeAdvice de la coupe « getXValue »
            System.out.println("getting X attribute");
      }
```

- **Coupe** (nommée aussi **point d'action**, **de coupure**, ou encore **de greffe**), en anglais, **pointcut** :

Désigne un ensemble de points de jonction sur lequel le tisseur d'aspect insérera les codes advice.

- *Exemple :*

```
      pointcut getXValue():
          call(int Point.getX());// Coupe s'appliquant lors de l'appel à
la méthode getX sur une instance de la classe Point
```

- **Design Pattern** (**patron de conception** ou motif de conception en français) **:**

En génie informatique, un design pattern est un concept issu de la programmation orientée objet, destiné à résoudre les problèmes récurrents. Les patrons de conception décrivent des solutions standards pour répondre à des problèmes d'architecture et de conception des logiciels. À la différence d'un algorithme qui s'attache à décrire d'une manière formelle comment résoudre un problème particulier, les patrons de conception décrivent des procédés de conceptions généraux. Il ne s'agit pas de fragments de code, puisque les patrons de conception sont le plus souvent indépendants du langage de programmation, mais d'une méthode de conception, c'est-à-dire d'une manière standardisée de résoudre un problème qui s'est déjà posé par le passé. Le concept de patron de conception a donc une grande influence sur l'architecture logicielle d'un système. Utiliser des patrons de conceptions est fortement conseillé afin d'obtenir un code robuste. En effet, ils servent généralement à obtenir des couplages faibles entre les classes métier. De plus, la standardisation de la manière d'architecturer une solution à un problème récurrent permet une meilleure lisibilité du code.

- **Eclipse** :

Eclipse est une communauté open source. L'outil Eclipse est une plate-forme de développement Java. AspectJ est un plugin de Eclipse permettant de faire de l'aspect.

Site : http://www.eclipse.org/

- **Greffon** : voir *code advice*.

- **Introduction (Mécanisme d') :**

Le mécanisme d'introduction permet d'ajouter des éléments à une classe X. Avec AspectJ il est ainsi possible d'ajouter des attributs, des méthodes, des constructeurs, ou même de définir la classe parente de X ou encore une nouvelle interface que doit implémenter la classe X.

- **Introspection :**

Le terme d'**introspection** vient du latin « introspectus » action de regarder à l'intérieur. En aspect, cela signifie que à l'intérieur d'un code advice nous regardons l'élément ayant déclenché le code advice, c'est à dire le point de jonction. Avec AspectJ l'introspection peut être fait grâce au mot-clef thisJoinPoint ou à l'aide du paramétrage des coupes.

- **Paramétrage de coupe :**

Avec AspectJ cela correspond à une manière de faire de l'introspection. Lors de la définition de la coupe on définit les paramètres de la coupe. Les paramètres possibles correspondent aux informations disponibles sur le point de jonction tel que l'objet appelant, l'objet appelé, ou les paramètres de la méthode appelée.

- ***Point de greffe, de coupure ou point d'action :*** *voir coupe*

- **Point de jonction**, d'**exécution** (en anglais, **joinpoint**) :

Endroit spécifique dans le flot d'exécution du système, où il est valide d'insérer un code advice (greffon). Pour clarifier le propos, il n'est pas possible, par exemple, d'insérer un code advice au milieu du code d'une fonction. Par contre on pourra le faire avant, autour de, à la place ou après l'appel de la fonction.

  - *Exemples :* méthode, constructeur, R/W d'attribut, exceptions…

```
public static void main(String[] args) {
      Point p = new Point(4,5);
      p.getX();  // Point de jonction déclenchant les codes advice
lié aux coupes s'appliquant sur la méthode getX de Point
}
```

- **Programmation orientée aspect (POA,** en anglais **aspect-oriented programming** - **AOP):**

La POA est un paradigme de programmation qui permet de réduire fortement les couplages entre les différents aspects techniques d'un logiciel. La programmation orientée aspect est basée sur le principe de l'inversion de contrôle (en anglais, IOC, Inversion Of Control). La POA est une technologie dite transverse. Elle permet de centraliser le code non pas lié à un objet mais à un concept regroupant plusieurs objets (Exemple : affichage de message de débogue dans un traceur). Bien que l'ensemble des exposés présentait la POA avec Java, un langage objet, la POA en tant que concept n'est pas liée à un langage particulier. Elle peut être mise en œuvre aussi bien avec un langage orienté objet qu'avec un langage impératif comme le C. Le seul prérequis pour faire de la POA est l'existence d'un tisseur d'aspect pour le langage cible (cf. Tisseurs d'aspects).

Les concepts de la programmation orientée aspect ont été formulés par Chris Maeda et Gregor Kiczales, qui travaillaient alors pour le Xerox PARC.

- **Programmation orientée objet :**

La programmation orientée objet (POO), est une façon d'architecturer une application informatique en regroupant les données et les traitements sur ces dernières au sein d'une même entité, les objets.

- **Tissage** ou **tramage** (en anglais, **weaving**) :

Insertion statique ou dynamique dans le programme principale des codes advice définis dans les aspects.

- **Tisseurs d'aspects :**

Un tisseur d'aspects permet d'injecter les codes advice à l'intérieur du code formant la base de l'application. AspectJ est un tisseur d'aspects.

- **wildcards :**

En AspectJ cela correspond au caractère spéciaux utilisés lors de la définition des coupes afin de généraliser une coupe. Les wildcards permettent de créer des expressions régulières simples.

  - *Exemples de wildcards :* *, ., ..