

Philippe PRADOS
site@philippe.prados.name

AspectJ, la programmation par aspect



*Préservez l'environnement,
n'imprimez pas ce document*

TABLE DES MATIÈRES

Table des matières.....	2
Table des matières.....	2
1 Les principes.....	3
2 Un « bon » exemple.....	4
3 Torture test.....	6
4 Quand tisser ?.....	7
5 Les autres usages.....	8
6 Quand utiliser la programmation par aspect ?.....	8
7 Les points faibles.....	9
8 Pour conclure.....	9

Avant-propos

La programmation par Aspect peut se résumer en quelques mots : Injecter de nouvelles instructions ou de nouvelles structures à partir d'une requête sur un code Java. Ou bien, pour les geeks, lecteur du magazine : INSERT <code> WHERE <critères>. Parmi les implémentations de ces principes, AspectJ est la plus avancé.

Nous avons vu dans un article précédant l'utilisation du JSR269 pour générer des ressources et des classes, lors de la compilation de classes java annotée. Nous allons étudier une autre technologie permet de dépasser les limitations.

Il n'est pas possible de faire une introduction avancé d'AspectJ, tellement il est riche et subtile. Nous nous proposons de survoler les principes pour bien comprendre à quoi cela sert et quand l'utiliser.

1 LES PRINCIPES

Parmi les différents objectifs de la programmation par Aspect, nous pouvons citer les plus importants :

- Permettre une programmation « déclarative » via des annotations, en complément d'une programmation « impérative » via du code java. J'entends pas là que l'utilisateur d'un framework Aspect se contente d'annoter son code pour en bénéficier.
- Mutualiser les usages dans l'exploitation de frameworks. Les frameworks sont généralement limités, car les utilisateurs doivent constamment appliquer les mêmes patterns de développement. La programmation par Aspect permettra de mutualiser cela.
- Séparer les POJOs des frameworks intrusif. Quelques frameworks impose aux classes des héritages, d'avoir quelques méthodes ou d'implémenter quelques interfaces. Cela pollue les objets métiers. Cela pourra être évité grâce à ces nouveaux outils.

Pour commencer, résumons ce qu'est un Aspect ou l'AOP (aspect-oriented programming). C'est la combinaison entre des « points d'insertions » et des « greffons ». Ce vocabulaire n'est pas très claire pour le moment. Les points d'insertions sont identifiés par des requêtes sur les sources du langage lui même. Une syntaxe spécifique permet d'identifier des formes de programmations par l'aspect du code. Tiens, tiens. Voilà d'où tous cela viens. Aux résultats de ces requêtes, nous pouvons appliquer des greffons. En fait, insérer du code. Nous pouvons résumer ces quelques phrases en une équation : Aspect = Point d'insertion + Greffon, ou en anglais, Aspect = Point cut + Advice.

Les programmes sont donc enrichi d'un certains nombres d'Aspects. Suivant les aspects présents, les transformations du code seront différentes. Certains aspect ne seront utilisés qu'en phase de déverminage, d'autres en production. Ces choix peuvent être tardif dans le développement. L'ajout d'un Aspect ne remet pas en cause le code lui-même. C'est un très gros avantage par rapport aux frameworks classiques.

Prenons quelques exemples de points d'insertions :

- Sélection de classes, de méthodes
- Sélection d'usage (invocation de méthodes, gestion d'exceptions, de contexte d'exécution)
- Sélection d'attributs (set et get)

Et quelques exemples de greffons :

- Avant ou autour de l'invocation d'une méthode,
- Dans le corps d'une méthode
- Après le retour d'une méthode
- Lors d'une exception dans la méthode

Lorsque la combinaison est rencontré, il est possible d'injecter du code Java. Ce dernier a la possibilité d'intervenir sur les paramètres, le contexte d'appel, l'objet retourné, l'exception produite, etc.

Puis, intervient le « tissage » de code. C'est l'opération consistant à injecter réellement le code aux endroits stratégiques. Le « tissage » consiste à appliquer les « greffons » aux « points d'insertions ». Ce dernier peut être effectué à la compilation ou au chargement des classes. AspectJ intervient au niveau du byte-code java, pour injecter de nouveaux comportements au points identifiés par les points-cuts. L'analyse du byte-code permet d'identifier l'initialisation d'instances (`new`) ou de classes, l'appel de méthodes, la manipulation d'attributs (`get` et `set`), le traitement d'exceptions. Il permet également de modifier la structure de la classe (Inter-type déclarations), d'ajouter des interfaces, des méthodes, des attributs ou des annotations et même de modifier l'héritage.

Par contre, il ne permet pas de supprimer un attribut, une interface ou une annotation, de modifier le type d'un attribut, de modifier la signature d'une méthode, de modifier profondément le comportement d'une méthode. Seul les ajouts sont possibles et la modification de l'héritage.

AspectJ peut intervenir à plusieurs niveaux dans le cycle de développement. Premièrement, lors de la phase de compilation des sources Java. S'il doit intervenir sur une classe déjà compilé, il en extrait une nouvelle version, injecté de code. Cette approche à l'avantage de faire payer le sur-coût de l'intégration des aspects au développeur et non à l'application. La compilation est plus lente, pas l'exécution.

AspectJ peut également intervenir lors du chargement des classes par un ClassLoader spécial. Toutes les classes instanciées par ce derniers peuvent alors être enrichies. Cette approche est très sympathique car elle permet de modifier le comportement de classes, sans devoir tous recompiler. En ajoutant une archive avec un Aspect « inside », le comportement de toutes les classes peut être impacté. Cette approche a un impact à l'exécution, car chaque classe doit être ausculté lors du chargement, pour vérifier s'il n'est pas nécessaire d'y injecter des modifications. Des paramètres permettent de limiter cette dégradation.

Pour éviter la gestion parfois difficile du chargeur de classes, il est également possible d'ajouter un agent java à la JVM. Le paramètre `-javaagent:aspectjweaver.jar` enrichit ainsi toutes les classes java.

D'autres frameworks de programmation par aspect travaillent à l'exécution. Ils utilisent généralement un AutoProxy (implémentation dynamique d'une interface) et l'inspection pour identifier les candidats à un enrichissement du code. L'avantage est de pouvoir modifier à chaud les comportements, mais cela est au prix d'une dégradation des performances. JbossAOP et SpringAOP utilisent cette démarche.

AspectJ propose deux approches pour décrire les aspects. La première est historique. Elle a été proposée avant l'introduction des aspects à Java. Une nouvelle syntaxe spécialisée est proposée, dans des fichiers d'extensions `*.aj`. Cette vision permet une vérification par le compilateur, une expressivité très forte et finalement une simplification dans l'utilisation. Par contre, cela impose d'utiliser un compilateur spécifique pour toutes les classes. Le compilateur `ajc` est compatible `javac`. Il permet l'enrichissement des classes lors de la compilation. Notez que les classes ainsi produites sont des pures classes Java. Elles peuvent être distribués sans aucun problèmes.

Une autre approche, fruit du mariage d'AspectJ avec le projet AspectWerkz, consiste à n'utiliser que des annotations Java 5. Cette vision présente l'avantage de ne pas exiger de compilateur particulier. Le code est directement compatible avec tous les outils d'analyses de code, de vérification, de documentation, etc. Votre chef de projet ne pourra pas râler ;-) Mais, les limitations des annotations rendent parfois le code plus complexe. Par exemple, tous les noms des classes doivent être complet. Contrairement aux fichiers `*.aj`, il n'est pas possible d'utiliser les informations d'import de java. De plus, par respect de la compilation Java, les ajouts d'attributs ou de méthodes sont plus complexes à coder.

2 UN « BON » EXEMPLE

L'intérêt de la programmation par aspect est très souvent occulté par l'utilisation de mauvais exemples. Au même titre qu'il existe un fameux mauvais exemple pour expliquer la récursivité, il existe de mauvais exemples pour l'AOP. (Vous connaissez tous l'exemple du factoriel. Pourquoi ne pas utiliser une simple boucle à la place ? La récursivité est pertinente s'il y a au moins deux appels à soi même. Sinon, une boucle suffit. De toute façon, vous ne pouvez pas comprendre la récursivité sans avoir d'abord compris la récursivité. ;-)

Revenons aux mauvais exemples, rencontrés partout pour expliquer la programmation par Aspect. « *Pour ajouter des traces* ». N'est-ce pas plus simple avec les APIs que nous utilisons déjà ? « *Ajouter du bench (cpu ou nombre d'invocation)* ». C'est quelques lignes à ajouter le temps du bench, ou bien, il est préférable d'utiliser des outils spécialisés. « *Rédiger des pré-conditions*. » Que c'est compliqué pour faire cela ! Un simple `assert` suffit.

Si l'AOP ne sert qu'à cela, l'investissement en formation n'est pas rentable. C'est un outil puissant, mais pas toujours facile à appréhender.

Prenons un exemple plus crédible, dont la programmation par Aspect va apporter une réelle valeur, en quelques minutes. Imaginons une architecture client/serveur classique. Le client consomme des traitements sur le serveur (via RMI, EJB ou autres). Le serveur exécute les services et retourne des résultats. Comme le serveur est à distance, sur un autre serveur physique, il peut arriver des problèmes dans la communication. Chaque méthode du serveur indique alors qu'elle peut générer une exception `RemoteException`.

```
public class Serveur
{
    private String name_="The remote services";
    private List<String> datas_=new ArrayList<String>();
    public String getName() throws RemoteException
    {
        return name_;
    }
    public void setName(String name) throws RemoteException
    {
        name_=name;
    }
    public void addData(String data) throws RemoteException
    {
        datas_.add(data);
    }
}
```

Le client consomme les services du serveur.

```
public class Client
{
    private Serveur remote_;
    // Injection de dépendance
    public void setServeur(Serveur remote)
    {
        remote_=remote;
    }
    public void sample() throws RemoteException
    {

```

```

        System.out.println("Service :"+remote_.getName());
        remote_.addData("Hello");
    }
}

```

Comme le service doit être publié en dehors de l'entreprise, pour son image de marque, il ne doit pas s'interrompre. Il faut alors mettre en place une architecture de haute disponibilité. Le client a impérativement besoin que le serveur fonctionne. Si ce dernier tombe, par effet domino, le client tombe également. C'est inacceptable pour le directeur informatique.

Naïvement, l'architecte propose alors d'ajouter un deuxième serveur. Ce dernier peut être en mode actif/passif ou actif/actif. Les deux serveurs utilisent une adresse IP virtuelle, une VIP. Le paramétrage des serveurs ou l'ajout de composants complémentaires permettent que lors d'un problème sur un serveur, l'autre prend le relais. Ainsi, toutes les requêtes vers le serveur seront satisfaites. Par l'un ou l'autre serveur, peu importe, du moment que le service est rendu. Le problème est résolu.

Et que se passe-t'il au moment précis du crash d'un serveur, lorsque sa carte réseau s'enflamme ? La connexion en cours vers le serveur est cassée ! Le client reçoit une `RemoteException`. Le traitement est dans un état instable. L'erreur s'est-elle produite juste avant de faire le traitement ? Juste après ? Au milieu ? Puis-je le rejouer ? Ce n'est pas grave pour un site de diffusion de vidéo, mais cela l'est plus s'il s'agit de livrer des médicaments en moins d'une heure, sans en oublier aucun ! Ajouter un répartiteur de charge, une VIP et un deuxième serveur ne traite pas complètement le risque. Aux limites, le service n'est pas rendu.

N'étant pas capable de connaître l'état du traitement lors du crash, nous ne savons pas si ce dernier peut être rejouer. Si c'était le cas, il serait sympathique de le relancer. Ni vu, ni connu, l'autre serveur prend la main et le service est rendu sans perturbation. Le client ne voit pas l'exception. Pour cela, il faut pouvoir rejouer des traitements en cas d'échec. Ils doivent être « idempotent », c'est à dire qu'ils peuvent être rejouer sans effet de bord. Dans les faits, les traitements de lectures sont idempotents. Les autres le sont rarement.

Essayons de limiter la casse. Par convention, les traitements `get*()` n'ont pas d'effet de bord. Ces traitements ne font que lire des données. Il est donc possible en cas d'exception, de redemander la même information. La VIP se charge alors de déléguer le traitement au bon serveur. Nous souhaitons que lors de l'invocation de toutes les méthodes `get*()`, l'appelant essaye plusieurs fois avant de signaler un échec. Un truc du genre :

```

int cnt=1;
for(;;)
{
    try
    {
        return ... // Execute le traitement
    }
    catch (RemoteException e)
    {
        if (++cnt>MAX_RETRY) throw (RuntimeException)e.fillInStackTrace();
    }
}

```

C'est maintenant qu'AspectJ va nous aider. Nous déclarons un pointcut (un critère de recherche) pour trouver toutes les invocations à des méthodes `get*()` qui propagent une exception `RemoteException` et ses dérivées.

```

public aspect ManageHighAvailability
{
    private pointcut idempotentRemoteMethods()
        : call(* *.get*(..) throws RemoteException+);
    ...
}

```

Puis nous déclarons un « conseil », autour des méthodes d'invocations, pour gérer le re-jeu.

```

public aspect ManageHighAvailability
{
    ...
    Object around() : idempotentRemoteMethods()
    {
        int cnt=1;
        for(;;)
        {
            try
            {
                return proceed();
            }
            catch (RemoteException e)
            {

```

```

        System.err.println("Echec, retry "+cnt);
        if (++cnt>MAX_RETRY) throw (RuntimeException)e.fillInStackTrace();
    }
}
}
}
}

```

En appliquant cet aspect à notre code, nous avons réduit le risque aux méthodes différentes de `get*()`. Il existe pourtant d'autres méthodes idempotentes. Il serait sympathique d'utiliser la même approche pour celles-ci. Oui, mais pour quelles méthodes ? Impossible de le savoir par la simple lecture de la syntaxe du code. La programmation par aspect ne nous aide plus. Sauf à utiliser AspectJ5, la dernière version qui intègre la prise en compte des annotations.

Commençons par modifier autant que possible, toutes les méthodes du serveur pour quelles soient idempotentes. Pour cela, il faut mémoriser les données sur un support de masse haute-dispo. Généralement, un cache temporel de quelques minutes avec les dernières requêtes et réponses d'un utilisateur permettent de régler le problème. Lors d'une nouvelle requête, si elle est identique à une requête précédente, il faut renvoyer le même résultat et ne pas effectuer le traitement.

Notez que ce problème est également rencontré avec HTTP. Que renvoyer à l'utilisateur qui valide deux fois son panier ? Ce dernier ne verra que le résultat de la deuxième soumission. Doit-on lui renvoyer une erreur ou lui informer que son panier a bien été pris en compte ? Certains frameworks MVC savent gérer cela mais c'est une autre histoire.

Avec JDK5, nous avons les annotations pour identifier les méthodes idempotentes. Déclarons une annotation.

```

@Target(ElementType.METHOD)
public @interface Idempotent
{
}

```

La méta-annotation `@Target` de l'annotation `@Idempotent`, indique que cette dernière ne peut être utilisée que sur des méthodes. Voilà enfin à quoi servent les annotations ! Nous pouvons alors annoter notre serveur.

```

public class Serveur
{
    @Idempotent
    public void setName(String name)
        throws RemoteException
    {
        System.out.println("use remote setName()");
        name_=name;
    }
    ...
}

```

La méthode `setName()` indique qu'elle ne possède pas d'effet de bord. Elle peut être rejouée sans problème.

Nous ajoutons à notre aspect toutes les méthodes `@Idempotent` et qui propagent une exception `RemoteException` et dérivée.

```

public aspect ManageHighAvailability
{
    private pointcut idempotentRemoteMethods() :
        call(* *.get*(..) throws RemoteException+)
        || call(@Idempotent * *.*(..) throws RemoteException+);
    ...
}

```

En ajoutant cet aspect, toutes les méthodes d'invocations vers des méthodes idempotentes et propageant une `RemoteException` sont capable de rejouer l'invocation. Le problème de la perte de connexion en cours de traitement, lors de la perte d'un serveur est réglé élégamment. A condition que tous les traitements du serveurs soient idempotent. Un risque résiduel existe avec les autres méthodes. L'équipe de développement du serveur doit revoir sa copie pour permettre de déclarer toutes les méthodes comme `@Idempotent`.

3 TORTURE TEST

C'est bien joli cela, mais comment vérifier le code ? Il faudrait simuler la génération d'une `RemoteException` pour toutes les méthodes. C'est impossible en pratique. Donc le test se limite, lors des tests unitaires, à tuer un serveur au hasard. Certaines méthodes seront vérifiées, en espérant que le comportement est le même pour toutes.

Mais, la programmation pas Aspect est là ! Nous voulons vérifier que le client est capable d'encaisser la production d'exception à n'importe quel moment. Nous déclarons un point-cut pour trouver toutes les méthodes qui propagent une exception `RemoteException` (idempotentes ou non).

```

public aspect InjectRandomRemoteException
{

```

```

private pointcut remoteExceptionMethods()
: call(* *.*(..) throws RemoteException+);
...
}

```

Puis nous déclarons un « conseil » avant l'invocation des méthodes, pour générer aléatoirement des exceptions.

```

public aspect InjectRandomRemoteException
{
...
before() throws RemoteException
: remoteExceptionMethods()
{
if (Math.random()<0.3) // <30%
throw new RemoteException("Simulate fail call");
}
}

```

Dans trente pour-cent des cas, ces méthodes généreront une exception. Si ça ce n'est pas un torture test... Sauf à faire preuve de malchance, l'application doit résister à ces traitements, pour les méthodes `get` ou `@idempotent`. Avec de la malchance, les tirages aléatoires génèrent une suite d'exception dont la taille est supérieur au nombre de re-jeu. Sans la programmation par Aspect, il est impossible de qualifier toutes les méthodes ! En quelques minutes, nous avons réduit le risque résiduel de l'architecture haute-disponibilité et nous l'avons vérifié dans des conditions pire que dans la vie réelle.

AspectJ permet également d'utiliser les annotations pour déclarer les aspects, les points d'insertion et les conseils. La version annoté de l'aspect de haute disponibilité est la suivante. Notez les ressembles avec l'approche syntaxique ci-dessus.

```

@Aspect
public class ManageHighAvailability
{
private static final int MAX_RETRY=5;

@Pointcut(
"call(* *.get*(..) throws java.rmi.RemoteException)" +
"call(@eu.prados.ha.Idempotent * *.*(..) "+
"throws java.rmi.RemoteException)")
void idempotentRemoteMethods() {}

@Around("idempotentRemoteMethods()")
public Object retryCall(
ProceedingJoinPoint thisJoinPoint)
throws Throwable
{ ...
}
}

```

Il est donc possible de livrer l'application avec une haute disponibilité de 99,9%, en n'incluant pas le JAR des aspects ou une version à 99,99% en ajoutant le JAR des aspects. Le première version est Open-Source et gratuite, la deuxième est payante. Enfin,... gratuite aussi, faut pas exagérer.

Avec une utilisation d'AspectJ via un chargeur de classes, les modifications du codes s'effectuent lors du chargement. Il n'y a pas besoin de recompiler. Donc, le simple ajout d'une archive permet d'améliorer la qualité du code exécuté. Si le client et le serveur sont physiquement placé sur le même nœud, on enlève l'archive de l'aspect et c'est réglé. En effet, il n'y a plus de risque réseau. Vous connaissez plus simple comme optimisation ?

4 QUAND TISSER ?

Quand s'effectue le tissage des classes ? Le compilateur d'AspectJ génère un fichier `META-INF/aop-ajp.xml` pour signaler toutes les classes candidates à une modification. Il est également possible de rédiger soit même un fichier `META-INF/aop.xml` pour indiquer les packages ou la forme des classes à ausculter. Il est possible d'inclure ou d'exclure des classes, voir de déclarer de nouveaux aspect directement dans ce fichier. Ces fichiers permettent d'optimiser le chargement des classes ne devant pas être tissées. En effet, ces dernières n'ont pas besoin d'être analysées. Le chargeur de classe ne fait alors que les installer en mémoire. Il n'est pas nécessaire d'ajouter du code, des interfaces, des annotations, des attributs, des méthodes ou de modifier l'héritage. Finalement, cela permet d'optimiser le non-tissage des classes.

5 LES AUTRES USAGES

Parmi les différentes fonctionnalités proposées par AspectJ, il est possible de lui faire découvrir des problèmes dans le code, directement lors de la compilation. Voici quelques exemples simples.

```
/** warn if setting non-public field outside a setter */
declare warning :
    set(!public * *) && !withincode(* set*(..))
    : "writing field outside setter" ;

/** Error when factory not used */
declare error :
    !withincode(Point+ SubPoint+.create(..))
    && call(Point+.new(..))
    : "use SubPoint.create() to create Point";

/** signal error if Thread constructor called outside
    our Thread factory */
declare error : call(Thread+.new(..))
    && !withincode(Thread acme.makeThread(..))
    : "constructing threads prohibited - use Factory";
```

6 QUAND UTILISER LA PROGRAMMATION PAR ASPECT ?

Comme pour les frameworks, il y a deux populations de développeurs : Les consommateurs d'Aspect et les rédacteur d'Aspect. Vous utilisez déjà la programmation par aspect lorsque vous utilisez des EJBs (injection des transactions et de la sécurité) ou consommez du JBoss ou du Spring.

Seul les rédacteurs de frameworks doivent maîtriser parfaitement AspectJ ou ces équivalents. Lorsqu'un framework impose des règles d'écriture de code pour l'utiliser, impose d'avoir des classes qui évoluent en parallèles, l'AOP permet de mutualiser les usages. Par exemples, pour un framework de persistance, il faut valoriser l'attribut `isDirty` lors de chaque modification d'une instance. Cela permet d'identifier les objets à sauver sur disque. Si l'attribut reste à `false`, l'objet n'a pas évolué. Il ne sert à rien de le coucher sur une mémoire de masse. AspectJ peut s'occuper d'ajouter cette valorisation automatiquement, sans risque d'oubli. L'AOP permet d'ausculter les modifications des attributs, d'ajouter automatiquement l'attribut `isdirty` à toutes les classes `@Persistante` et de le modifier pour les `set` d'attributs.

Pour gérer la sécurité, il faut invoquer certains traitements vérifiant les privilèges associés au thread. L'AOP peut les injecter via des annotations. La gestion des transactions peut être injecté et vérifié via l'AOP (`@Transactional`, ...). Une méthode transactionnelle peut être interdite d'invoquer une méthode qui refuse les transactions. Certains algorithmes de caches bénéficient de l'AOP. Si une classe est déclarée `@Immutable`, et/ou `@Serializable`, le cache peut maintenir plus ou moins d'information en mémoire.

Beaucoup de frameworks préconisent la programmation par `interface`. C'est à dire qu'il ne faut utiliser dans le code que des interfaces. Les classes d'implémentations ne sont utilisées que lors de la construction des objets ou via des factories. Il est possible de modifier tous les `new` pour utiliser un wrapper, implémentant l'interface, mais proposant une autre implémentation. Cela permet, par exemple, de déporter l'implémentation sur un autre serveur physique. Notez que cela est également possible sans AOP, avec un factory paramétrable. Spring utilise beaucoup ce pattern.

L'AOP permet une programmation déclarative. Une bonne approche consiste à exploiter les annotations. Par exemple : `Contract4j` permet la programmation par contrat (pre, post-conditions, invariant) en ajoutant des annotations. Ce framework utilise AspectJ en arrière plan. Il n'est pas nécessaire de maîtriser AspectJ pour utiliser `Contract4j`. Voici le serveur enrichie de la programmation par contrat.

@Contract

```
public class Serveur
{
    @Post("$return!=null")
    public String getName() throws RemoteException
    {
        return name_;
    }
    @Pre("name!=null")
    @Idempotent
    public void setName(String name) throws RemoteException
    {
        name_=name;
    }
    @Pre("data!=null")
    public void addData(String data) throws RemoteException
}
```

```
{
  datas_.add(data);
}
```

Dans l'absolue, le code java peut être enrichie d'annotations sémantiques, permettant éventuellement, l'injection de codes. Voici quelques exemples de méta-informations, décrété par le développeur et permettant ou non la prise en compte par les frameworks.

`@Idempotent`, `@Immutable`, `@Transactional`, `@Remote`, `@Serializable`, `@Override`, `@PostConstruct`, `@PreDestroy`, `@Test`, etc.

Sans AspectJ, l'alternative est d'utiliser le JSR 269 et le JDK6 (ou l'Annotation Processing Tools du JDK5). Cet outils permet de générer différents fichiers ou d'autres classes. Il a vocation à remplacer l'approche XdocLet qui exploitait des annotations dans les Javadocs. Cet outil est bien plus complexe à appréhender qu'AspectJ. Comme les exemples ci-dessus le démontre, quelques lignes suffisent à enrichir le code. Avec JSR 269, il faut maîtriser une API complexe, toute la syntaxe de java, etc. De mon point de vue, les annotations sont les meilleurs amis de la programmation par Aspect.

Dans une certaine mesure, l'AOP évite de mélanger le code métier POJO avec du code de framework. En pratique, il est plus simple d'écrire une log directement dans le code, même si cela crée une adhérence avec un framework de log. L'AOP c'est bien, mais il ne faut pas en abuser.

7 LES POINTS FAIBLES

AspectJ présente également des points faibles. Le déverminage n'est pas toujours aisé car java ne peut associer plusieurs fichiers pour la même classe (contrairement à C/C++). Un paramètre du compilateur permet de générer un code non « `inline` », facilitant le pas à pas. Pour aider le développeur, un plug-in Eclipse aide à identifier les pointcuts et les aspects associés. Des icônes présentes devant les lignes permettent de savoir si une ligne sera prise en charge par un aspect.

Il est impossible d'avoir des pointcuts dynamiques, calculés par le code. Les pointcuts sont statiques car défini lors de la phase de compilation. Il existe une porte de sortie comme l'instruction `if()` dans un pointcut. Cette dernière permet d'injecter un test effectué à l'exécution pour savoir s'il faut ou non invoquer le code complémentaire.

En tissage retardé lors du chargement des classes, le démarrage de l'application est plus lent. Mais cela est compensé par une souplesse de déploiement. Les impacts sur les performances sont négligeable, comparé à un code injecté à la main. C'est un des objectifs affichés par l'équipe d'AspectJ. J'ai vérifié cela en décompilant les classes générées.

8 POUR CONCLURE

Il existe de nombreux framework de programmation par aspect pour java ou pour d'autres langages. Il existe même un `aspectjs` pour le javascript !

L'AOP Alliance est un regroupement de plusieurs fournisseurs de frameworks, avec pour objectif, l'utilisation d'une API de base commune. Ainsi, il sera possible de mélanger plusieurs approches. AspectJ n'est pas encore compatible avec ces recommandations.

J'espère vous avoir donnée envie d'étudier AspectJ ou d'autres framework de programmation par aspect. Il faut au minimum 5 jours pour appréhender tous les concepts et la syntaxe. Le premier jour est consacré au tutoriel de la première version d'AspectJ (sans annotation et JDK5). Le deuxième jour au tutoriel des compléments d'AspectJ5. Les jours suivants pour tester, re-tester, comprendre, etc.

C'est un investissement non négligeable, mais les bénéfices peuvent être très important. Cela ne doit être entrepris que si vous désirez améliorer un framework existant ou proposer un nouveau framework à base d'annotations. En tant que simple utilisateur, la seule chose à comprendre est résumé dans cet article.

Pour traiter les annotations, plusieurs solutions s'offrent au développeur. Il peut utiliser les annotations à l'exécution en exploitant l'introspection de java. S'il souhaite uniquement ajouter des classes ou des ressources à partir annotations, le JSR269 est la solution. Pour modifier les classes sans en modifier les sources, AspectJ est la solution.

Une combinaison de JSR269 et d'AspectJ permet de produire des fichiers de paramètres et d'intervenir sur les classes, en s'appuyant sur les annotations. Localement, l'introspection peut être utilisé mais au prix d'un impact sur les performances.

Philippe Prados
articles@philippe.prados.name
Architecte chez Atos Origin