Cours Spring + AOP

1 Spring et les « *Aspects* »

- Spring propose d'intégrer, dans les architectures applicatives, la méthodologie de « *Programmation Orienté Aspect* » (POA).

  Elle reprend notamment certains principes du Design Pattern *Decorator* (famille « *Structuration* »), en offrant une approche particulière/singulière, permettant d'enrichir dynamiquement le comportement de classe(s) ou d'objet(s)), sur certains services et besoins transverses
  (Log, Sécurité, Transactionnels, Monitoring de performances, …).

  Le constat initial étant que les implémentations de ces fonctionnalités sont dispersées au sein d'une application ; ce qui la pollue, frêne sa maintenance, son évolutivité/exploitabilité, sa réutilisation, et entraine une perte de productivité de par une charge incombant importante….

- La POA permet d'améliorer l'identification et la segmentation des responsabilités d'une application, en externalisant des classes concernées, ce genre d'appels ou de fonctionnalités, et en les logeant dans des classes d'aspects dédiées, ce qui permet par ailleurs leur uniformisation.

  Quelques exemples:

  - Traiter les logs d'une seule et même façon pour toutes les classes de services.
  - Gestion de l'ouverture/la fermeture de Transaction autour d'appel de méthode de DAO (bien qu'aujourd'hui une apprôche par @nnotations Spring de Transaction soit devenu la norme)
  - Opérer des traitements sur des objets de modèle renvoyés par des DAO pour compléter (si nécessaire) les relations en opérant des requêtes supplémentaires (ex pour des relations polymorphes n'ayant pas de classe racine/commune).

- Spring fournit sa propre version d'AOP, ainsi qu'un support partiel d'AspectJ, framework leader sur cette technologie, au travers des namespace XML, et des @nnotations.

  Il est recommandé d'utiliser le support d'AspectJ, plutôt que Spring AOP classique (AspectJ étant plus puissant en terme d'AOP que Spring).

  JBoss fournit lui aussi un Framework d'AOP.

CC
Droits réservés

http://creativecommons.org/licenses/by-nc-nd/2.0/fr/

- Définitions des entités principales de la POA :
  - Un *Aspect* prend en charge une fonctionnalité transversale d'une application : Le code relatif est entièrement situé dans la classe de l'aspect, les classes métiers sont alors épurées de toute intrusion relative à ce besoin.

    Un *Aspect* est associé à une ou plusieurs *Coupe*.
  - Une *Coupe* (*PointCut*) définie le(s) *point(s) de jonction* dans la(les) classe(s) ou doivent s'appliquer/intégrer la fonctionnalité transverse;

    Par exemple : « ProductBO.* » désigne une coupe pour toutes les méthodes de cette classe.
  - Un *Point de jonction* (*JoinPoint*) défini un point dans l'exécution d'une séquence de code autour duquel on peut ajouter/tisser des aspects (en général, il s'agit d'une invocation de méthode) ;

    Il peut éventuellement appartenir à plusieurs coupes.

    On ne peut jamais insérer un greffon au milieu d'une méthode.
  - Un *Advice* se charge de fournir l'implémentation concrète d'une fonctionnalité d'aspect (il est donc relié à un *Aspect*, ainsi que ses *Coupes & Points de Jonctions*).

    L'*Advice* est souvent appelé *Greffon* (pour des raisons évidentes…), Spring propose les types suivants :
    - *before* (invoqué avant les *JoinPoint* d'exécution de méthodes),
    - *after returning* (invoqué après les *JoinPoint*),
    - *after throwing* (invoqué après les *JoinPoint*, si une exception est lancée),
    - *after* (invoqué après les *JoinPoint,* même si une exception est lancée),
    - *around*(invoqué avant ET après les *JoinPoint*, utilisation de l'instruction *proceed* entre les 2…, cf exemple dev…),

- La POA mise en place par Spring est moins évoluée que par le Framework AspectJ (notamment un seul type de points de jonction pour Spring, lors de la définition des coupes).

CC
Droits réservés
http://creativecommons.org/licenses/by-nc-nd/2.0/fr/

- Un *Tisseur d'Aspect* (*Aspect Weaver*) se charge de l'intégration entre un ensemble de classes et un ensemble d'aspect. Ce mécanisme peut se faire soit :
  - A la **compilation** (CTW – « *Compile Time Weaving* » via AspectJ) par la génération d'une nouvelle classe fille (qui utilisera l'aspect et ses codes Advice + l'invocation de la méthode de la classe mère),

    Ou bien par manipulation du pseudo-code (byteCode) de la classe (cf Framework *BCEL*, ou *Javassist*), qui va directement relier l'appel vers la classe du greffon.
  - Au **chargement** (LTW – « *Load Time Weaving* » toujours via AspectJ), un *ClassLoader* spécifique va charger la définition des classes dans la machine virtuel, et opérer des modifications sur la logique de leurs méthodes, afin d'y greffer les appels vers les aspects concernés.
  - A l'**exécution** (comme c'est le cas dans Spring AOP) par l'utilisation de *Proxy* (Design Pattern *Proxy*, cf aussi *java.lang.reflect.Proxy*) dynamiques, qui interceptent les appels de méthodes, exécutent les *Greffons* des *Aspects* (en fonction de leur type : *before, around, …*) et redispatchent les appels de méthodes finaux.

    Remarques :

    Les proxy dynamiques de JavaSE se basent sur des interfaces (d'ou la nécessité de pratiquer le développement des services par le biais de cette définition…), dans la cas contraire, Spring AOP utilisera la library CGLIB (Code Generation Library) pour générer le *Proxy*[1].

    Les proxy imposent des limitations, notamment lorsque l'on souhaite intercepter une méthode depuis l'objet courant, le proxy ne peut jouer son rôle dans ce cas précis, les solutions de contournement sont alors de passer en LTW ou CTW.

- Spring fournit des aspects « out-of-the-box », prêt à l'emploi (pour le debug, les traces, les performances), dans *org.springframework.aop.interceptor*, ainsi que dans *org.springframework.transaction.interceptor* pour la gestion des aspects de transactions.

- Spring AOP n'a pas la possibilité d'intercepter une classe qui ne serait pas définit sous forme de bean (dans un context Spring).

[1] Le *Proxy* d'un objet, c'est comme du Canada Dry, ça ressemble à l'objet, ça a le goût de l'objet, l'odeur de l'objet, mais ce n'est pas votre objet !!! :Dim

CC
Droits réservés
http://creativecommons.org/licenses/by-nc-nd/2.0/fr/

2 Contenu du TP

- Le code source repart de la version la plus avancée du TP sur Spring (automatisation de la création et de l'injection des beans par @*nnotations* de classes et de propriétés).

- Ajout dans le fichier *de context.xml* de :

  <aop:aspectj-autoproxy /> : ajout d'un *autoProxy* de la Library AspectJ, afin d'automatiser la définition et le tissage des aspects, dans le container Spring.

  <bean id="logAspect" class=" fr.shortcircuit.tp9.aspect.notify.NotifierAdvice" />

  Déclaration d'un bean d'*Aspect* chargé de gérer les logs au niveau des services, de façon transverse, et unifiée.

- Création de la classe *NotifierAdvice* :

  Comparaison de l'implémentation d'un *Advice* avec l'implémentation d'une *@nnotations @Aspect* de la bibliothèque *AspectJ* (préférée à l'Interface *AfterReturningAdvice*, de *SpringAOP,* plus contraignante et moins puissante).

- Présentation de plusieurs PointCut (déclarés dans une @nnotations @AfterReturning) :

  - Pour une méthode de l'interface *IProductDAO*.
  - Pour toutes les méthodes de l'interface *IProductDAO* (via *regexp*)
  - Pour toutes les classes business ainsi que leurs méthodes (via *regexp*)

  Ainsi que certaines options supplémentaires sur les *PointCut* & *Aspects*…

CC
Droits réservés
http://creativecommons.org/licenses/by-nc-nd/2.0/fr/