SPRING AOP

Historique

Les concepts de la **programmation orientée aspect** ont été formulés par Gregor Kiczales et son équipe, qui travaillait alors pour le Xerox PARC.

Limites technologiques

Les techniques actuelles de conception logicielles et de programmation amènent à découper un logiciel en modules techniques *a priori* indépendants les uns des autres car gérant des aspects différents du système conçu.

Par exemple, on aura différents modules ou couches logicielles pour gérer les aspects techniques suivants :

1. gestion des utilisateurs (authentification),
2. archivage des données (persistance),
3. programmation concurrentielle (multi-threading),
4. information pendant l'exécution du logiciel (trace),
5. logique métier (par exemple informatique de gestion, système d'information géographique, commerce électronique, ...)
6. etc.

Dans la pratique, on s'aperçoit que ces modules sont en fait intimement liés : c'est l'**intrication ou 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., conduisant à une complexification du code, de son développement et de sa maintenance.

L'**inversion de contrôle** mise en œuvre par la **programmation par aspect** va permettre d'extraire les dépendances entre modules concernant des aspects techniques entrecroisés et de les gérer depuis l'extérieur de ces modules en les spécifiant dans des composants du système à développer nommés **aspects**.

Principe

Ainsi, au lieu d'avoir un appel direct à un module technique depuis un module métier, ou entre deux modules techniques différents, en **programmation par aspect**, le code du module en cours de développement est concentré sur le but poursuivi (la logique bancaire, pour reprendre notre exemple) , tandis qu'un **aspect** est spécifié de façon autonome, prenant en charge de faire appel au module technique requis (l'authentification de l'utilisateur) à un certain point d'exécution du système.

Bien sûr, si chaque **aspect** créé devait lui-même effectuer un appel au module technique à un point d'exécution explicite, c'est à dire par exemple avec une dépendance directe vers le module métier où devra s'intercaler le code technique, on n'aurait alors fait que décaler le problème.

Aussi, l'astuce particulière de la **programmation par aspect** consiste à utiliser un système d'expressions régulières pour préciser à quels points d'exécution (en anglais, **joinpoint**) du système l'aspect spécifié devra être activé.

Définition

Un **aspect** permet donc de spécifier :

1. les points d'action(**poincut**), qui définissent les points de jonction satisfaisants aux conditions d'activation de l'**aspect**, donc le ou les moments où l'interaction va avoir lieu,
2. les **greffons** c'est à dire les programmes (**advice**) qui seront activés avant, autour de ou après les **points d'action** définis.

La **programmation orientée aspect**, parce qu'elle propose un paradigme de programmation et de nouveaux concepts, a développé un jargon bien spécifique qui ne facilite pas la compréhension de ses concepts qui sont, en définitive, simples mais puissants.

1. **aspect** : un module définissant des *greffons* et leurs *points d'activation*,
2. **greffon** (en anglais, **advice**) : un programme qui sera activé à un certain *point d'exécution* du système, précisé par un *point de jonction*,
3. **tissage** ou **tramage** (en anglais, **weaving**) : insertion statique ou dynamique dans le système logiciel de l'appel aux *greffons*,
4. **point d'action**, de **coupure**, de **greffe** (en anglais, **pointcut**) : endroit du logiciel ou est inséré un *greffon* par le *tisseur d'aspect*,
5. **point de jonction**, d'**exécution** (en anglais, **join point**) : endroit spécifique dans le flot d'exécution du système, où il est valide d'insérer un *greffon*. Pour clarifier le propos, il n'est pas possible, par exemple, d'insérer un *greffon* 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.
6. **considérations entrecroisées** (en anglais, cross-cutting concerns) : mélange, au sein d'un même programme, de sous-programmes distincts couvrant des aspects techniques séparés.

Bilan / Avantages

Le couplage entre les modules gérant des aspect techniques peut être réduit de façon très importante, en utilisant ce principe, ce qui présente de nombreux avantages :

1. Maintenabilité accrue : chaque module peut être maintenu indépendamment des autres,
2. Meilleure réutilisabilité : tout module peut être réutilisé sans se préoccuper de son environnement. Chaque module implémentant une fonctionnalité technique précise, on n'a pas besoin de se préoccuper des évolutions futures : de nouvelles fonctionnalités pourront être implémentées dans de nouveaux modules qui intéragiront avec le système au travers des **aspects**.
3. Gain de productivité : le programmeur ne se préoccupe que de l'**aspect** de l'application qui le concerne, ce qui simplifie son travail, et permet d'augmenter la parallélisation du développement.
4. Amélioration de la qualité du code : La simplification du code qu'entraine la **programmation par aspect** permet de le rendre plus lisible et donc de meilleure qualité.

5. Installation des outils.

Le noyau AOP de **Spring** est basé sur les librairies *spring.jar,* l'API d'AspectJ à savoir *aspectJ.jar, aspectweaver.jar,* et *cglib.jar* pour l'introduction des proxy (modification de bytecode).