

## TD n°11 - Programmation par aspects

- ▷ AspectJ (<http://www.eclipse.org/aspectj>) est une extension du langage de programmation Java permettant de faire de la programmation par aspects. Son utilisation est facilitée en l'utilisant à l'intérieur de l'environnement Eclipse, à travers les AJDT (AspectJ Development Tools).

### Exercice 1: Stockage de myrtilles

Considérons le problème suivant : plusieurs threads doivent partager une ressource commune, dans un système producteur/consommateur. Dans notre exemple, la ressource commune est une classe nommée **Storage** possédant deux méthodes d'ajout (**addResource**) et de retrait (**removeResource**).

 Sauf mention explicite, il est demandé de ne pas modifier le code existant.

1. Créer un nouveau projet **AspectJ** sous **Eclipse**, dans lequel vous incluez les fichiers source suivants :

Consumer.java, EmptyStorageException.java,  
LockedStorageException.java, Main.java,  
Producer.java, Storage.java

Vérifier que l'exécution de la classe **Main** s'exécute sans problèmes.

Tel quel, ce code n'est pas particulièrement pratique à manipuler : il n'effectue aucun affichage permettant de savoir si le code s'exécute. Plutôt que de modifier le code existant (a fortiori uniquement pour des fins de débogage), nous allons écrire un *aspect* permettant d'afficher des messages à l'écran.

- ▷ Un aspect se présente sous la forme d'une classe **Java** utilisant le mot-clé **aspect**, et contenant en plus des méthodes et attributs usuels :
  - des *pointcuts*, définissant des points du code à partir desquels il est possible d'insérer les aspects ;
  - des *advices*, définissant le code à insérer avant, pendant, après voire autour des pointcuts.

```

aspect PinceCoupante {
    pointcut tournevis(): execution(* Tournevis .*(..));
    before() : tournevis () {
        System.out.println ("Advice_ from_:" +
            thisJoinPointStaticPart .getSignature ().getName()); }}

```

Ainsi, l'aspect précédent permet de rajouter du code s'exécutant avant l'exécution de toutes méthodes de la classe **Tournevis**. Une courte liste des éléments de langage appartenant à **AspectJ** se trouve à l'adresse <http://eclipse.org/aspectj/doc/released/progguide/quick.html>.

2. Ajouter un aspect **Log.aj** dans le code affichant un message lors de l'entrée et de la sortie des méthodes **addResource**, **removeResource** et toute autre méthode considérée comme intéressante.

*Remarque :* Il n'existe pas de mécanisme simple en **AspectJ** permettant d'activer ou de désactiver des aspects. Pour faire simple, il est recommandé de commencer chacun de vos aspects avec une variable **ENABLED** :

```

final static boolean ENABLED = true;
pointcut marteau() : if (PinceCoupante.ENABLED) && // ...

```

Les aspects peuvent accéder à certaines informations du contexte appelant, en utilisant une variable nommée **thisJoinPoint**. Par exemple l'objet appelant peut être obtenu par **thisJoinPoint.getTarget()**.

3. Étendre l'aspect précédent en ajoutant un appel à la méthode **Storage.display()** lors de toute modification d'un objet **Storage**. Faire de manière à ce que les deux aspects soient activables indépendamment l'un de l'autre.

## Exercice 2: Race conditions

Un problème important lorsque l'on manipule des threads provient du partage des ressources, partage qui peut mener à des situations de concurrence (*race condition*). Ici, l'accès à la ressource **Storage** est fait de manière concurrente par plusieurs threads. Nous proposons de rajouter un tel mécanisme à travers un aspect, en ajoutant un mutex **Java** pour chaque objet de la classe **Storage** (sous la forme d'un **Lock** de la classe **ReentrantLock**)<sup>1</sup>

---

1. Dans un exemple aussi simple, il est possible de résoudre le problème simplement en utilisant des méthodes **synchronized**. Ici, on propose d'implémenter le mécanisme à la main afin de pouvoir détecter les accès concurrents.

1. Combien d'aspects sont nécessaires dans ce cas particulier ?

Proposer un pointcut ainsi qu'un ou plusieurs advices permettant d'implémenter un aspect créant un objet de type `ReentrantLock` pour chaque objet de type `Storage`, et utilisant les méthodes `lock()` et `unlock()` pour encadrer les appels aux méthodes de la classe `Storage`.

Dans l'exemple précédent, il est possible que les demandes d'entrée en section critique restent non détectées. En effet, l'appel à la méthode `lock()` est bloquant pour le thread courant, sans lever d'exception.

2. Proposer une manière de faire pour lever une exception `LockedStorageException` lorsqu'un appel à `lock()` est fait alors que le Mutex est déjà dans les mains d'un autre thread.

Les implémentations des classes comme `Storage` contiennent déjà les exceptions que l'on vient de rajouter. Dans l'esprit de la programmation par aspect, il serait bon de pouvoir les ajouter directement à travers l'aspect que nous venons d'écrire.

3. Est-il possible, en utilisant `AspectJ`, d'écrire un aspect faisant qu'une fonction qui initialement ne levait pas d'exception en lève une après application ?
4. Est-il possible, en utilisant `AspectJ`, de modifier le prototype d'une fonction existante ?

*Remarque :* L'utilisation de la FAQ d'`AspectJ` (à l'adresse <http://www.eclipse.org/aspectj/doc/released/faq.php>) peut aider pour répondre à ces questions.

### Exercice 3: Aspects génériques réutilisables

Il est dommage de ne pouvoir appliquer un tel mécanisme qu'à la classe `Storage`, alors qu'en fait il est naturel de vouloir spécifier des familles de méthodes qui doivent s'exécuter sans concurrence.

1. Inclure dans votre code<sup>2</sup> les fichiers source suivants :

```
Condition.java, CoordinationAction.java, Exclusion.java,  
Method.java, MethodState.java, Mutex.java, Selfex.java,  
TimeoutException.java, ainsi que le fichier Coordinator.aj
```

Le fichier `Coordinator.aj` définit un pointcut abstrait, et plusieurs advices dépendant de ce pointcut. Il est possible d'interagir avec cet aspect simplement avec les méthodes `addMutex` et `addSelfex`, qui prennent en paramètre respectivement un nom de méthode et un tableau de noms de méthodes.

2. Écrire un aspect étendant `Coordinator`, définissant un pointcut correct, afin de faire que les méthodes `addResource` et `removeResource` ne puissent pas être exécutées en même temps par deux threads différents.
3. Quels problèmes de maintenabilité la programmation avec `AspectJ` pose t'elle ?

---

2. Exemple tiré de la documentation d'`AspectJ` 1.6

#### Exercice 4: Aspect et modulaire

Considérons le module OCaml suivant définissant un type `storage`, ainsi que les opérations `addResource` et `removeResource` sur ce type :

```
module type STORAGE = sig
  type resource = string
  type t
  exception NotEnoughResource of resource
  val new_storage : unit -> t
  val add_resource : t -> resource -> int -> unit
  val remove_resource : t -> resource -> int -> unit
end
```

Les sources contiennent une implémentation de ce type de données basée sur des tables de hachage.

1. Comment implémenter un comportement équivalent à celui obtenu dans les exercices précédents sans modifier le code existant, simplement à l'aide d'un foncteur OCaml?
2. Pouvez-vous imaginer un exemple d'aspect qui ne soit pas simplement simulable à l'aide d'un foncteur OCaml? Et réciproquement?