Méta-programmation & Programmation Orientée Aspect

TP

Laurent Broudoux (@lbroudoux)
Janvier 2014

Exercice 1

Objectifs : Appréhender la notion de Proxy Dynamique en Java, avec interception et avec remplacement de code.

Conditions : Projet sous Eclipse sans dépendance externe. Les librairies junit-4.11.jar et harmcrest-core-1.3.jar pourront éventuellement être utilisées pour compléter le projet par des tests unitaires.

Partie 1 : Proxy Dynamique avec interception

- Commencer par créer une interface et une implémentation basiques

- Créer ensuite un InvocationHandler qui permettra :
  - De tracer un message avant l'invocation de la méthode visée
  - De tracer un message après l'invocation de la méthode visée
  - De tracer le temps passé à exécuter cette méthode

Aide : commencer par créer une implémentation de java.lang.reflect.InvocationHandler et laissez vous guider par Eclipse. Regarder ensuite la javadoc de la classe java.lang.reflect.Method.

- Créer maintenant en utilisant le design pattern Factory (voir http://www.oodesign.com/factory-pattern.html), un Proxy Dynamique qui interceptera l'appel à votre interface et qui sera retourné en lieu et place de l'implémentation de base.

Aide : regarder ensuite la javadoc de la classe java.lang.reflect.Proxy. Pour le ClassLoader, utiliser la classe Thread.

- Créer une classe de Test Junit ou une classe main utilisant votre classe basique du début.
  - Ecrire une version utilisant directement l'implémentation écrite
  - Ecrire une version utilisant l'implémentation retournée par votre Factory
  - Voir les différences

Partie 2 : Proxy Dynamique avec remplacement

- Commencer par créer une simple classe (ex : User) possédant

quelques attributs String différents (ex : username, firstname, lastname)

- Créer une interface en utilisant le design pattern Repository (voir http://martinfowler.com/eaaCatalog/repository.html). Elle portera des méthodes permettant de récupérer des listes de l'objet que vous avez créé précédemment (ex : List<User>)
  - Définir plusieurs méthodes de type findBy*()
  - Définir une méthode findAll()
  - Définir une méthode count()
  - Définir une méthode add()

- Créer maintenant un nouvel InvocationHandler. Ce handler sera en charge d'exécuter les traitements définis par l'interface Repository.

Aide : afin de conserver les choses simples, considérer que votre handler est initialisé avec une liste d'objets instance de votre simple classe. Pour sélectionner les objets, regarder la javadoc de la classe java.lang.Class.

- Créer maintenant en utilisant le design pattern Factory (voir http://www.oodesign.com/factory-pattern.html), un Proxy Dynamique qui interceptera l'appel à votre interface Repository et déléguera l'exécution à votre handler

- Créer finalement une classe de Test Junit ou une classe main permettant de tester. Cette classe aura les responsabilités suivantes :
  - Initialiser un ensemble d'objets de votre classe simple (ex : User) qui serviront de test
  - Appeler la factory et utiliser l'implémentation retournée
  - Tester les méthodes de votre repository

- Bonus : essayer de combiner l'InvocationHandler repository avec l'InvocationHandler écrit dans la partie 1.

Exercice 2

Objectifs : Définir et manipuler une annotation personnalisée en Java. Ecrire un Proxy Dynamique capable d'exploiter les métadonnées de cette annotation.

Conditions : Projet sous Eclipse sans dépendance externe. Les librairies junit-4.11.jar et harmcrest-core-1.3.jar pourront éventuellement être utilisées pour compléter le projet par des tests unitaires.

Histoire : on veut pouvoir cacher (au sens « mettre dans » et extraire d'un cache) les résultats de l'invocation de nos méthodes findBy*() définit dans l'exercice précédent.

- Créer une nouvelle annotation nommée Cacheable. Cette annotation doit être applicable sur les méthodes et accessible au runtime (on veut l'exploiter ensuite). Avec cette annotation, je dois pouvoir spécifier :
  - Que je souhaite alimenter le cache après obtention du résultat (comportement par défaut),
  - Que je souhaite vider le cache après ajout d'un nouvel objet (appel à la méthode add() notamment)
  - Bonus : Que je souhaite lire le cache mais pas forcément y enregistrer mon résultat

Aide : regarder les classes et interfaces du package java.lang.annotation. Penser aux énumérations pour définir les commandes.

- Créer un nouvel InvocationHandler. Ce handler sera en charge d'intercepter les appels aux méthodes marquées @Cacheable et d'activer le cache à ce moment.

Aide : afin de conserver les choses simples, considérer simplement le cache comme une table de hashage mettant en correspondance clé et valeur. Penser à un algorithme permettant de créer des clés facilement.

- Reprendre et modifier la factory créée dans l'exercice 1. Celle-ci devra maintenant intégrer le comportement Cacheable dans le Prody Dynamique qu'elle renvoie.

- Créer ou réutiliser la classe de Test Junit ou la classe main permettant de tester. Cette classe aura les responsabilités suivantes :
  - Initialiser un ensemble d'objet de votre classe simple (ex : User) qui serviront de test
  - Appeler la factory et utiliser l'implémentation retournée
  - Tester les méthodes de votre repository celles avec Cache, celles sans Cache. Pensez qu'il faut appeler au moins 2 fois la même méthode pour ce servir du cache …
  - Positionner des traces ou des mesures de temps de réponse pour voir l'effet du Cache.

Exercice 3

Objectifs : Appréhender la programmation orientée aspect en Java avec AspectJ. Définir un advice basique, utiliser la notation d'ITD.

Conditions : Projet sous Eclipse sans dépendance externe. Le projet doit impérativement être converti en « AspectJ Project » et non en simple « Java Project ». Si les plugins AspectJ ne sont pas installés dans votre environnement Eclipse, récupérer la bonne URL d'Install Site à partir de http://www.eclipse.org/ajdt/downloads/ et procéder à l'installation des plugins depuis *Help > Install New Software...* . Les librairies junit-4.11.jar et harmcrest-core-1.3.jar pourront éventuellement être utilisées pour compléter le projet par des tests unitaires.

Partie 1 : Simple aspect avec around advice

- Transformer l'InvocationHandler de l'exercice 1 en Aspect. Les aspects fonctionnant directement sur des classes, vous n'avez plus besoin de l'interface basique que vous aviez créer
  - Créer un nouvel aspect dans Eclipse
  - Définir votre pointcut
  - Définir votre advice utilisant ce pointcut

Aide : des exemples de pointcuts sont disponibles dans la présentation. Ne pas oublier qu'un aspect est un singleton ! Il ne doit pas conserver d'état ...

- Créer une classe de Test Junit ou une classe main utilisant votre classe basique maintenant instrumentée en utilisant l'aspect.
  - Exécuter cette classe depuis Eclipse pour bénéficier du compilateur AspectJ intégré
  - Inspecter le bytecode du .class produit. Que constatez-vous ?

Partie 2 : Aspect avec ITD

- Reprendre la classe Java simple créé dans l'exercice 1 (ex : User) sur laquelle avait été produit un Repository

- Ecrire un aspect permettant :
  - De rendre cette classe Serializable au sens Java (implémentant java.io.Serializable)
  - De rendre cette classe Serializable au sens Xml (capacité à s'exporter sous une forme chaîne de caractères représentant

l'objet)

- Créer une classe de Test Junit ou une classe main utilisant votre classe basique (ex : User) maintenant instrumentée en utilisant l'aspect.
  - Exécuter cette classe depuis Eclipse pour bénéficier du compilateur AspectJ intégré
  - Inspecter le bytecode du .class produit. Que constatez-vous ?