UFR 919 Sciences de l'ingénieur
Master Informatique – Spécialité STL
ALADYN

Objets, exceptions et transactions

Jacques Malenfant
professeur des universités

version 1.0 du 30/08/2013

Résumé

L'objectif de ce projet est de comprendre les possibilités offertes par l'outil Javassist, grâce à ses capacités de génération dynamique de code, et ce par la réalisation d'un système de transactions sur les objets pour la restauration de leur état en cas d'erreurs provoquant le lancement d'exceptions.

1 Pourquoi des transactions ?

Considérez le programme suivant :

```
public class Test {
    protected int i ;
    public Test() { this.i = 0 ; }

    public void m2(int i) throws Exception {
        // supposons une erreur d'exécution et lançons une exception
        throw new Exception() ;
    }

    public void m1() {
        try {
            this.i = 10 ;
            System.out.println("dans m1: i = " + this.i) ;
            this.m2(this.i) ;
        } catch(Exception e) {
            // code palliant l'erreur ayant provoqué l'exception
        }
        System.out.println("catch de m1: i = " + this.i) ;
    }

    public void m1_transactionnee() {
        // copie de la valeur modifiée spéculativement
        int ancien_i = this.i ;
        try {
            this.i = 10 ;
            System.out.println("dans m1_transactionnee: i = " + this.i) ;
            this.m2(this.i) ;
        } catch(Exception e) {
            // remise en palce de la valeur initiale après erreur
            this.i = ancien_i ;
            // code palliant l'erreur ayant provoqué l'exception
        }
        System.out.println("catch m1_transactionnee: i = " + this.i) ;
    }
```

```
    public static void main(String[] args) {
        System.out.println("Java standard :");
        (new Test()).m1() ;
        System.out.println("version dans l'esprit transaction :");
        (new Test()).m1_transactionnee() ;
    }
}
```

L'exécution de ce programme produit le résultat suivant :

```
Java standard :
dans m1: i = 10
catch de m1: i = 10
version dans l'esprit transaction :
dans m1_transactionnee: i = 10
catch m1_transactionnee: i = 0
```

Dans le premier cas, lorsque l'erreur provoquant le lancement de l'exception se produit, la modification faite par la méthode `m1` à la variable d'instance `i` reste effective. Dans plusieurs cas, ce genre de modifications sont faites dans le but de faire un calcul et de modifier l'état des objets participant à ce calcul *dans leur totalité*. Si une erreur se produit pendant ce calcul et qu'une exception est lancée, le code palliant à cette erreur va devoir remettre les objets dans un état cohérent pour continuer l'exécution correctement.

Dans plusieurs cas, la solution la plus simple est de remettre tous les objets dans l'état où ils étaient au début du calcul qui devrait donc se faire *entièrement* ou *pas du tout*. C'est l'esprit de ce qu'on appelle une *transaction*. Les transactions sont importantes en informatique et en particulier dans le domaine des bases de données, L'idée fondamentale est que les structures de données, qu'elles soient en mémoire ou sur disque, doivent souvent obéir à des invariants de représentation globaux (à chaque structure de données), et lorsqu'on commence à toucher à certaines valeurs dans ces structures, il est important de modifier toutes les valeurs de manière à rétablir l'invariant avant de continuer l'exécution.

Considérez par exemple le cas d'un tableau utilisé comme collection de valeurs. Un tableau sert à contenir les références aux objets et une variable n contient le nombre de valeurs dans la collection, et donc dans le tableau. Considérez maintenant la séquence suivante cherchant à retirer la dernière valeur de la collection :

```
tableau[n-1] = null ;
... ; // code provoquant une erreur et le lancement d'une exception
n = n - 1 ;
```

Le code remplace la référence au dernier objet en mettant null à la position n - 1 (indiçage à partir de 0 jusqu'à n - 1 pour n valeurs), puis il faut réduire la valeur de n. Si du code entre les deux provoque une erreur, n ne sera pas mis à jour et l'invariant de la collection qui veut que n donne le nombre d'objets dans la collection rangés aux indices 0 à n - 1 dans le tableau ne sera plus vrai. Les opérations suivantes sur la collection provoqueront inévitablement une erreur fatale.

Beaucoup d'erreurs fatales dans les systèmes sont dues à des interruptions en cours de modification de structures de données, qui sont alors laissées dans un état incohérent vis-à-vis de leur invariant, ce qui empêche alors de poursuivre l'exécution correctement et force l'abandon et le redémarrage du programme. Remettre en état cohérent les structures de données est souvent la clé de la robustesse des programmes et des systèmes.

2 Spécification du problème

Le projet consiste à étendre Java pour supporter automatiquement la définition de classes et de méthodes « *transactionnables* » (*sic*). Quand une méthode sera déclarée « *transactionnable* », cela

voudra dire que si une exception est lancée pendant son exécution, y compris pendant l'exécution de tous les appels faits directement ou indirectement à d'autres méthodes, alors tous les objets « *transactionnables* » modifiés depuis le début de cette méthode devront être remis dans l'état où ils étaient au début de l'exécution de la méthode. Les objets « *transactionnables* » sont les instances des classes déclarées « *transactionnables* », et uniquement celles-là.

Pour marquer les méthodes et les classes « *transactionnables* », l'annotation suivante sera utilisée :

```
//  Transactionable.java --- annotation for "transactionable" classes and methods.
package fr.upmc.aladyn.transactionables.annotations;

import java.lang.annotation.Retention ;
import java.lang.annotation.Target ;
import java.lang.annotation.RetentionPolicy ;
import java.lang.annotation.ElementType; ;

/**
 * The annotation <code>Transactionable</code> marks classes of objects that
 * must be made restorables in a previously checkpointed state as well as the
 * methods that will be considered as transactions when called.
 *
 * <p><strong>Description</strong></p>
 *
 * The annotation is retained at run-time in order to be used by a meta-object
 * based or a Javassist class transformation that will provide the transaction
 * semantics.
 *
 * <p>Created on : 2013-08-29</p>
 *
 * @author  <a href="mailto:Jacques.Malenfant@lip6.fr">Jacques Malenfant</a>
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Transactionable { }
```

Plus précisément, la sémantique de transaction est la suivante. L'état d'un objet sera défini comme les valeurs de ses variables d'instances déclarées et héritées par leur classe, peu importe leur visibilité (`public`, `protected`, ...). Si une variable contient une référence à un autre objet, c'est la référence et non le contenu de cet autre objet qui est considéré comme la valeur. Cet autre objet ne sera restauré à son état initial que s'il est lui-même instance d'une classe déclarée « *transactionnable* ».

La restauration de l'état des objets étant définie, considérons le comportement lors du lancement d'une exception. Que la méthode « *transactionnable* » possède un traitant d'exception (« *exception handler* », ou clause `catch`) ou non, si une exception est lancée pendant son exécution, les objets « *transactionnables* » modifiés depuis le début de son exécution doivent être remis dans leur état initial au début de l'exécution de la méthode « *transactionnable* ».

Notez bien qu'il s'agit de réaliser des transactions *imbriquées*, c'est-à-dire qu'une méthode « *transactionnable* » peut en appeler d'autres directement ou indirectement. Si une méthode « *transactionnable* » `m1` appelle une autre méthode « *transactionnable* » `m2`, et qu'une exception est lancée pendant l'exécution de `m2`, les objets « *transactionnables* » modifiés par `m2` sont d'abord remis dans leur état initial au début de l'exécution de `m2`. Si `m2` possède un traitant d'exception qui arrive à remettre le calcul sur les rails et que ce dernier se poursuit jusqu'à la fin normale de la méthode `m1`, alors la transaction de `m1` réussit et ses changements sont conservés. Mais si `m2` n'arrive pas à remettre le calcul sur les rails ou qu'une nouvelle exception est levée, alors on remontra à `m1` et les objets modifiés entre le début de `m1` et le début de `m2` seront à leur tour remis dans leur état initial au début de l'exécution de `m1`.

3 Première partie : une solution programmée à la main

Pour réaliser le projet, vous allez avoir besoin d'outils de réflexion en Java qui ne seront introduits que vers les troisième et quatrième semaines du cours. Si vous attendez jusque-là pour commencer, vous devrez mobiliser pas mal de votre temps pour faire le projet dans les dernières semaines du cours, ce qui va très certainement vous mettre en difficulté non seulement pour ALADYN mais pour tous vos autres cours pour la fin de la vague.

Pour prendre de l'avance dans un premier temps, écrivez un programme Java représentatif du problème et annotez-le avec `@Transactionable` pour marquer les classes d'objets et les méthodes « *transactionnables* ». Ensuite, programmez à la main une solution pour que l'exécution de votre programme se passe selon les spécifications précédentes. Cela veut dire, un peu comme dans l'exemple que nous avons donné au début de cet énoncé, que vous introduirez à la main tout le code nécessaire au traitement des transactions dans votre programme. En prenant soin d'avoir des transactions imbriquées, vous pourrez alors simuler des erreurs en lançant des exceptions dans votre programme à différents endroits et différents niveaux pour ainsi vous assurer du bon fonctionnement de votre solution manuelle.

Pour bénéficier au mieux de ce travail dans la suite du projet, prenez soin d'écrire un code de traitement des transactions assez générique. Pour traiter une transaction, vous allez devoir ajouter du code au début et à la fin des méthodes « *transactionnables* », de même que dans son traitant d'exception. Un code suffisamment générique devrait pouvoir être copié et collé avec le minimum de modifications d'une méthode « *transactionnable* » à une autre.

Par exemple, pour restaurer l'état des objets, ne faites pas ce qui est montré dans l'exemple du début de l'énoncé, à savoir d'écrire du code spécifique pour copier les valeurs initiales des variables d'instance d'un objet particulier et pour les remettre en état à la fin. Recherchez une solution générique immédiatement, en utilisant la réflexion de structure de Java que nous verrons dès la première semaine du cours.

Et n'oubliez pas que plusieurs transactions peuvent se faire en même temps : les transactions imbriquées dans un même fil d'exécution (« *thread* ») ou des transactions que nous supposerons indépendantes dans d'autres fils d'exécution.

Note importante : pour simplifier le traitement des objets « *transactionnables* », vous pourrez supposer que tous les accès à leurs variables d'instance vont utiliser des accesseurs en lecture et écriture. Vous pourrez par la suite essayer de vous passer de cette hypothèse...

4 Deuxième partie : étude et comparaison de deux solutions réflexives

Nous allons voir au cours d'ALADYN deux approches en réflexion de comportement pour résoudre ce genre de problème : une approche plus interprétée et une approche plus compilée. Dans la première, des méta-objets sont associés aux objets pour intercepter les appels de méthodes qu'ils reçoivent, et alors il est possible d'intervenir avant et après leur exécution. Dans la seconde, on se sert de la réflexion au chargement pour modifier et ajouter du code dans les classes et les méthodes une fois pour toute au lancemrent du programme. Dans les deux cas, nous allons utiliser l'outil Javassist pour illustrer ces deux approches.

Nous vous demandons donc, dans un second temps, de produire deux solutions au problème de gestion des transactions : une à base d'injection de code au chargement et une fondée sur l'utillisation de méta-objets. Nous vous demandons de comparer ces deux approches ; préparez-vous à répondre à des questions à ce sujet lors des soutenances pour l'évaluation de votre travail.

5 Réalisations attendues et modalités

Nous vous demandons de réaliser le projet sous Eclipse en prévoyant tout le code nécessaire pour lancer l'application et dans votre rendu vous devrez également joindre des tests unitaires JUnit dans des sous-packages tests.

Modalités de remise des projets :

- Le projet se fait **obligatoirement** en **binôme**. Tous les fichiers du projet doivent comporter les noms (`authors`) de tous les auteurs en Javadoc.
- Le projet est à rendre le **vendredi 15 novembre 2013 à minuit** au plus tard sous la forme d'une archive `tgz` si vous travaillez sous Unix ou `zip` si vous travaillez sous Windows que vous m'enverrez à `Jacques.Malenfant@lip6.fr` comme attachement fait proprement avec votre programme de gestion de courrier préféré. Donner pour nom à votre archive vos numéros de dossier d'étudiants sous la forme n-n (ex. : 2700001-2800002.tgz).
- Le projet doit être réalisé avec Java SE 7 et l'outil Javassist en version 3.18.0-GA. Méfiez-vous cependant que le code compilable par Javassist ne correspond qu'à une version assez ancienne de Java (1.4) où il n'existe ni générique ni conversion automatique entre types de base et objets (`int` vers `Integer` et vice et versa, par exemple). Attention, peu importe le système d'exploitation sur lequel vous travaillez, il faudra que votre projet s'exécute correctement sous Eclipse en Mac Os X (que j'utilise et sur lequel vous devrez me faire votre soutenance).
- Votre projet utilisera JUnit pour faire des tests de vos classes. Il comportera aussi des classes concrètes (application) permettant de réaliser les tests. Tous vos tests seront regroupés dans un répertoire spécifique `tests` au même niveau que le répertoire `src` dans votre projet Eclipse. Utilisez exactement la même structure de paquetage dans le répertoire `tests` que dans le répertoire `src`.
- Le code doit être documenté en Javadoc (générée et incluse dans votre livraison dans un répertoire `doc` au même niveau que votre répertoire `src`) et commenté. Le code doit également être correctement formatté (indentation, ...).
- L'évaluation comportera une soutenance d'une quinzaine de minutes où vous devrez démontrer votre projet (prévoyez dans le code envoyé tout ce qu'il faut pour faire une telle démonstration) et répondre à des questions sur ce dernier. **Les soutenances auront lieu dans la semaine du 18 novembre 2012** selon un horaire de passage qui sera publié le moment venu sur le site de l'UE. Tout retard à ces soutenances entraînera son annulation et une pénalité sera appliquée sur l'évaluation du projet.
- **Tout manquement à ces règles élémentaires entraînera une pénalité dans la note du projet !**