IN201 : 3 – Gestion d'étiquettes

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Résumé

Ce TP sert de récapitulatif aux quatre premières séances du cours IN201. Il consiste à concevoir et implanter un système permettant de gérer des étiquettes sous forme d'une ontologie.

1 Objectifs

Les objectifs de ce TP sont les suivants :
- vérifier que vous êtes capable de proposer une solution de conception simple en utilisant le langage UML ;
- vérifier que vous êtes capable d'implanter une classe en respectant les principes énoncés en cours ;
- vérifier que vous êtes capable de tester une classe via JUnit.

2 Présentation du problème

Les systèmes d'indexation de documents proposent de plus en plus l'utilisation d'étiquettes (*tags*) pour caractériser sémantiquement des documents. Les étiquettes sont maintenant utilisées pour classifier par exemple des photos, des documents PDF, des adresses Web, des mails etc.
Nous nous intéressons ici à l'indexation de marque-pages provenant d'un navigateur Web. On souhaite organiser les étiquettes caractérisant les marque-pages suivant une *ontologie* utilisant une relation d'inclusion. Par exemple, les étiquettes « C++ » et « Java » seront considérées comme des sous-types d'une étiquette « Langages ». Cette relation d'inclusion peut être représentée par un arbre. Un exemple vous est donné sur la figure 1.

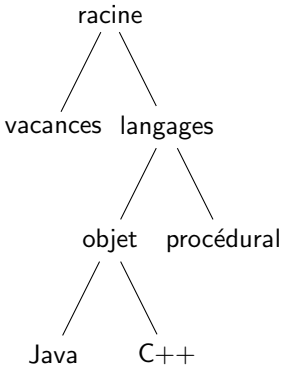

FIGURE 1 – Un exemple d'ontologie sur les étiquettes

On souhaite disposer d'une classe permettant de trouver une étiquette dans l'arbre à partir de son nom (on suppose que les noms d'étiquettes sont uniques) et d'une autre permettant d'afficher tous les marque-pages liés à une étiquette **et à ses sous-étiquettes**. La classe permettant de trouver une étiquette sera liée à un arbre d'étiquettes particulier.
Les marques pages sont constitués d'un titre, d'une URL (*Uniform Resource Locator*) et d'une date d'enregistrement du marque-page. Les étiquettes sont caractérisées par un nom et un ensemble de marque-pages. Lorsque l'on affiche un marque-page, on affiche ces trois informations. Un marque-page peut être indexé par plusieurs étiquettes, ou ne pas être indexé du tout. On peut modifier le titre d'un marque-page, mais pas ses autres caractéristiques. Enfin, il est possible d'ajouter ou d'enlever un marque-page de la liste des marque-pages caractérisés par une étiquette. On supposera que les étiquettes sont placées dans l'arbre à leur création et que l'on ne peut pas les changer de place par la suite. On remarquera que l'on a une étiquette « racine » en haut de l'arbre. On supposera que l'étiquette définissant la racine de l'arbre porte toujours ce nom.

3 Conception d'une solution

On va chercher à modéliser le problème en utilisant une conception orientée objet. On ne s'intéresse absolument pas à l'aspect algorithmique du problème pour l'instant. On propose d'utiliser les classes suivantes pour résoudre le problème :

- une classe `MarquePage` qui représente les marque-pages ;
- une classe `Etiquette` qui représente les étiquettes et permet également de modéliser l'arbre représentant l'ontologie ;
- une classe `RechercheEtiquette` qui permet de rechercher une étiquette dans un arbre en le parcourant avec un certain algorithme ;
- une classe `Afficheur` qui permet d'afficher les marque-pages d'une étiquette et de ses sous-étiquettes.

On supposera que l'on dispose d'une classe `Date` pour représenter les dates.

1. dans un premier temps, écrire un diagramme UML représentant les différentes relations qui existent entre ces classes. N'oubliez pas les noms de rôles et les multiplicités ;
2. affiner ce diagramme en utilisant des navigabilités et visibilités et en proposant une vue plus « implantation » de votre solution ;
3. proposer enfin un diagramme de classe UML détaillé faisant apparaître attributs et opérations pour les classes `MarquePage`, `Etiquette`, `RechercheEtiquette` et `Afficheur`.

   Dans ces diagrammes détaillés, on ne fera pas apparaître les éventuels détails d'implantation en ce qui concerne les multiplicités de type $*$ (choix d'un tableau, d'un objet de type `ArrayList` etc.). On utilisera la syntaxe UML suivante :

   ```
   nomAttribut : TypeAttribut [n]
   ```

   qui précise que `nomAttribut` est un attribut « contenant » *n* objets de type `TypeAttribut` (cette syntaxe n'impose pas que cet attribut soit ensuite implanté sous la forme d'un tableau).

4 Dépôt Subversion

À partir de cette semaine, comme les binômes sont définitifs, l'URL de votre dépôt est `https://eduforge.isae.fr/repos/IN201/GPE/bNUM` où :

- `GPE` est le nom de votre groupe de PC en majuscule
- `NUM` est votre numéro de binôme (votre PC(wo)man les possède)

Par exemple, l'URL du dépôt du binôme 9 du groupe S est `https://eduforge.isae.fr/repos/IN201/0921/S/b9`. Pour connaître votre numéro de binôme, connectez-vous sur `http://www.tofgarion.net/lectures/IN201`.

Pour réaliser le TP, vous allez devoir créer un projet Java sous Eclipse en utilisant votre dépôt Subversion. Si vous avez configuré votre dépôt pour qu'il soit disponible dans la vue *SVN Repository Exploring* d'Eclipse, vous créerez votre projet en faisant un *checkout* **à partir du répertoire TP3** de votre dépôt en cliquant droit dessus. N'oubliez pas de configurer votre *build path* pour pouvoir utiliser les éventuelles archives JAR placées dans le répertoire `lib`.

5 Implantation de la solution

Pour réaliser le TP, vous allez devoir créer un projet Java sous Eclipse en utilisant votre dépôt Subversion. Si vous avez configuré votre dépôt pour qu'il soit disponible dans la vue *SVN Repository Exploring* d'Eclipse, vous créerez votre projet en faisant un *checkout* **à partir du répertoire TP3** de votre dépôt en cliquant droit dessus. N'oubliez pas de configurer votre *build path* pour pouvoir utiliser les éventuelles archives JAR placées dans le répertoire `lib`.

Vous allez devoir écrire les quatre classes `MarquePage`, `Etiquette`, `RechercheEtiquette` et `Afficheur` et les tester avec JUnit. Toutes les classes devront appartenir au paquetage `fr.isae.tags`. La fonction recherche présente un algorithme

récursif permettant d'effectuer une recherche en profondeur d'abord dans un arbre. Attention, lorsque vous l'implanterez, vous aurez besoin d'ajouter une méthode dans la classe `RechercheEtiquette` par exemple (la méthode `rechercher` de la classe `RechercheEtiquette` ne prenant en paramètre que le nom de l'étiquette à chercher).

Fonction recherche(nd, nomRecherche) : un algorithme récursif de recherche de nœud dans un arbre en profondeur en utilisant le nom du nœud

entrées : un nœud de départ de recherche *nd*, le nom du nœud à chercher *nomRecherche*
sortie : le nœud dont le nom correspond dans l'arbre ou **`null`** si aucun nœud de l'arbre ne correspond

```
1  si nom de nd = nomRecherche alors
2  |  retourner noeud ;
3  fin
4  pour chaque nf nœud fils de nd faire
5  |  aux ← recherche(nf, nomRecherche) ;
6  |  si aux ≠ null alors
7  |  |  retourner aux ;
8  |  fin
9  fin
10 retourner null ;
```

Vous disposez de plusieurs classes qui vous vous aider à réaliser le TP et qui vous sont fournies sous forme *compilée* dans une archive JAR disponible sous le répertoire `lib` (leur documentation Javadoc est également disponible) :

- une classe `DataGenerator` permettant de générer l'arbre d'étiquettes présenté sur la figure 1 et d'associer des marque-pages à ces étiquettes. Attention, les méthodes de la classe sont statiques, elles s'appelent donc en utilisant le nom de la classe et non pas une instance de la classe ;
- une classe `TestAfficheur` utilisant la classe précédente pour construire un arbre d'étiquettes et affichant les marque-pages associés à diverses étiquettes ;
- une classe `RechercheEtiquetteTest` qui est une classe de tests unitaires pour la classe `RechercheEtiquette` utilisant l'arbre généré par `DataGenerator`. Les différentes méthodes de test de la classe vous permettront de mieux cerner les éventuels problèmes de votre algorithme (voir également la javadoc de la classe).

Les classes `RechercheEtiquette` et `Afficheur` vous sont également fournies sous forme *compilée* dans une archive JAR `lab3-search.jar` pour vous permettre d'avancer dans votre TP. Pour qu'Eclipse les utilise prioritairement, configurer le *build path* de votre projet dans l'onglet « Order and Export » en faisant « remonter » l'archive JAR avant le répertoire `src` via le bouton `Up`. N'oubliez pas d'enlever de votre CLASSPATH ou de votre projet Eclipse l'archive JAR les contenant lorsque vous allez les développer sinon ce sont les classes fournies qui seront utilisées.
Pour que tout le monde parte de la même solution, nous vous fournissons le diagramme détaillé de chaque classe. Vous devrez implanter vos classes (`MarquePage`, `Etiquette`, `RechercheEtiquette`, `Afficheur`) en respectant ces diagrammes si vous voulez que le programme de test et la classe de test JUnit fournis fonctionnent. Vous trouverez dans votre dépôt les squelettes des classes que vous devez développer sous le répertoire `src`.
Quelques précisions techniques en ce qui concerne l'implantation :

- pour stocker des séquences ou des ensembles, on choisira d'utiliser des instances de `java.util.ArrayList` ;
- lorsque vous voulez comparer deux chaînes de caractères (qui sont des instances de la classe `String`), il faut utiliser la méthode `equals` de `String` ;
- la classe `java.util.Date` est fournie par l'API Java et représente une date.
  La documentation javadoc de la classe sur le site d'Oracle [1] vous fournira plus de détails.

6 Travail à rendre

Vous avez plusieurs documents à rendre :

- lorsque vous avez fini votre conception sur papier, vous la remettez à votre PC(wo)man avant d'aller en TP ;
- à la fin de la séance de TP, vous commitez vos classes sur votre dépôt ;
- vous commitez le TP complet jeudi soir.

Références

[1] Oracle. *Java API specifications*. 2013. URL : http://docs.oracle.com/javase/7/docs/api/index.html.

License CC BY-NC-SA 3.0