IN201 : 3 – Gestion d'étiquettes

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

Solution

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 Contenu

Ce corrigé succinct contient les réponses au TP numéro 3. Vous y trouverez en particulier les diagrammes UML de la partie conception. Le corrigé de la partie implantation en Java est disponible sur http://www.tofgarion.net/lectures/IN201.

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.

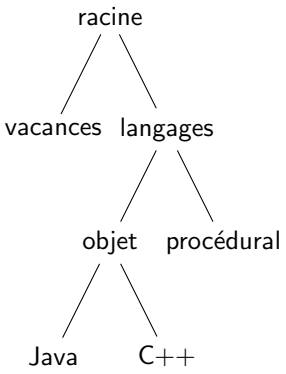

FIGURE 1 – Un exemple d'ontologie sur les étiquettes

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 ;

   Solution :

   Une solution est proposée sur la figure 2. Rien de bien particulier ici, les multiplicités se déduisent naturellement de l'énoncé. Je n'ai pas mis d'agrégation ni de composition car je n'en voyais pas l'intérêt.

   Les multiplicités de l'association entre `Etiquette` et `MarquePage` se déduisaient de l'énoncé. On peut très bien avoir un marque-page sans étiquette et une étiquette qui n'a pas de marque-page particulier.

   La relation d'inclusion entre étiquettes est représentée par une association réflexive sur la classe `Etiquette`. Je n'ai pas choisi d'utiliser une classe intermédiaire qui représenterait l'arbre (de toute façon on ne vous demandait pas de développer une classe de ce type), il suffit ensuite de connaître l'étiquette racine pour pouvoir manipuler l'arbre. Les multiplicités se déduisaient encore une fois de l'énoncé : une étiquette peut avoir plusieurs filles dans l'arbre ou ne pas en avoir, et une étiquette a au plus une étiquette mère (l'étiquette racine n'en a pas).

   Je n'ai pas mis de multiplicité du côté de la classe `RechercheEtiquette`, car on n'en *a priori* pas besoin. On remarquera que le rôle de l'étiquette en association avec la classe `RechercheEtiquette` est `racine`, car on a juste besoin de connaître l'étiquette racine pour parcourir l'arbre.

   Enfin, la classe `Afficheur` est en association avec la classe `RechercheEtiquette` : on suppose qu'un afficheur donné utilise une instance de `RechercheEtiquette`.

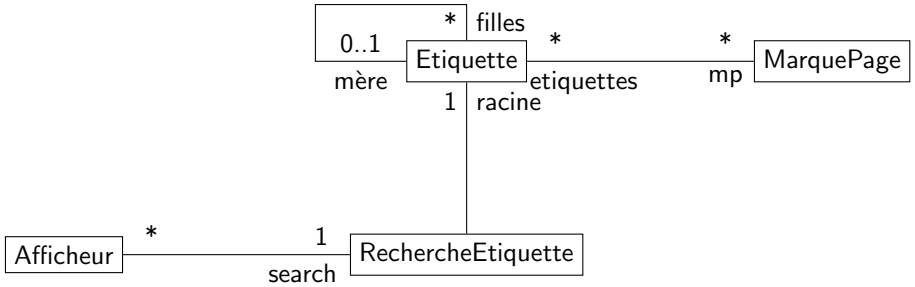

FIGURE 2 – Un diagramme de classes d'analyse du problème

2. affiner ce diagramme en utilisant des navigabilités et visibilités et en proposant une vue plus « implantation » de votre solution ;

   Solution :

   Rien de bien difficile ici : classiquement, on impose une navigabilité vers certaines classes et on restreint la visibilité pour respecter le principe d'encapsulation. Une solution est proposée sur 3.

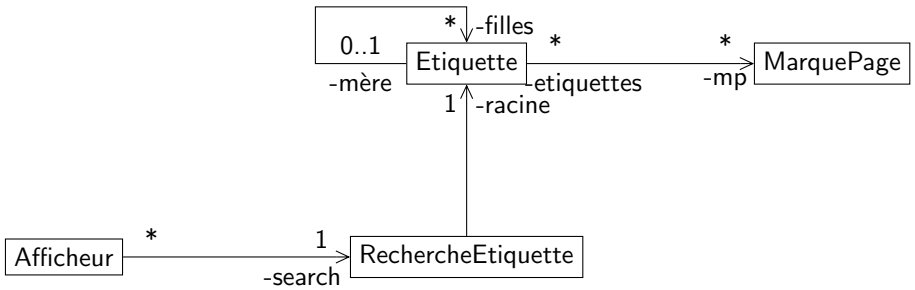

FIGURE 3 – Un diagramme de classes du problème avec navigabilité et visibilité

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).

   Solution :

   Pour chaque classe, j'ai fait apparaître la méthode `toString` qui permet d'obtenir une représentation sous forme de chaîne de caractères d'un objet. Il ne fallait pas oublier d'écrire les constructeurs de chaque classe.

   Le diagramme de la classe `MarquePage` est présenté sur la figure 4. Rien de bien particulier pour cette classe, elle sert juste à encapsuler des données. D'après le sujet, il n'y a besoin d'un modifieur que pour l'attribut `titre`.

   Le diagramme de la classe `Etiquette` est présenté sur la figure 5. Le constructeur de la classe prend en paramètre un nom et l'étiquette mère de l'étiquette à construire. Ce dernier paramètre sera **`null`** pour l'étiquette racine. Je n'ai pas fait de modifieurs pour les attributs de la classe, car je considère qu'ils ne changent pas. On aurait par exemple pu introduire une méthode `setMere` pour « déplacer » l'étiquette dans l'arbre. La classe possède également une méthode `ajouter` qui permet d'ajouter un marque-page à l'étiquette. On aurait pu placer une méthode `ajouter(etiquette: Etiquette)` dans la classe `MarquePage`, ce qui semblerait plus logique. Cependant, ceci nous impose de disposer de la classe `Etiquette` pour pouvoir tester `MarquePage`. Il me semble plus commode de conserver `MarquePage` « simple » pour pouvoir la tester rapidement.

   Les diagrammes des classes `RechercheEtiquette` et `Afficheur` sont présentés sur les figures 6 et 7. Rien de bien difficile pour ces classes, les quelques méthodes qui les composent se déduisaient assez facilement de l'énoncé. On n'a pas besoin d'accesseurs ou de modifieurs pour les attributs de ces classes. J'ai mis une méthode `toString` dans `RechercheEtiquette` et dans `Afficheur` pour pouvoir afficher l'arbre sur lequel l'instance de la classe travaille.

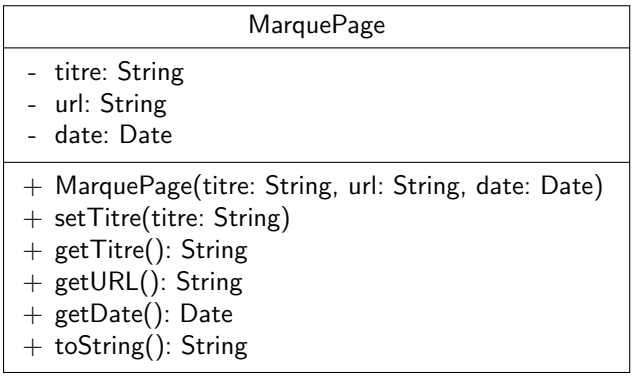

FIGURE 4 – Diagramme détaillé de la classe `MarquePage`

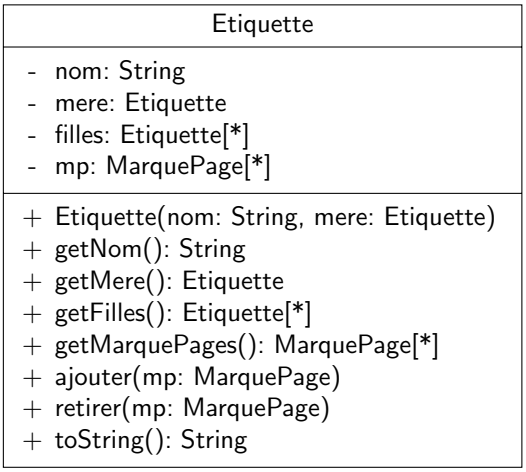

FIGURE 5 – Diagramme détaillé de la classe `Etiquette`

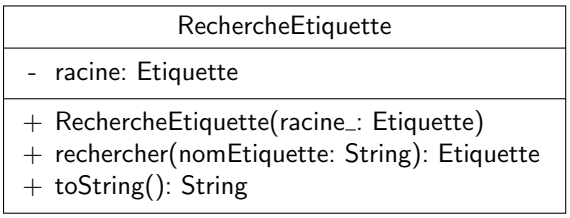

FIGURE 6 – Diagramme détaillé de la classe `RechercheEtiquette`

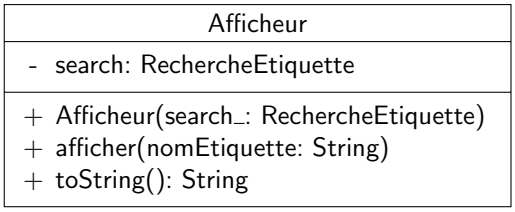

FIGURE 7 – Diagramme détaillé de la classe `Afficheur`

4 Implantation de la solution

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.

Solution :
Comme d'habitude, les sources sont disponibles sur le site. Vous y trouverez également les classes de test JUnit correspondant aux classes `MarquePage`, `Etiquette` et `RechercheEtiquette`. Il était difficile de tester avec JUnit la classe `Afficheur` car celle-ci utilise la console pour afficher ses résultats. Il aurait été judicieux de créer des méthodes renvoyant dans un premier temps une chaîne de caractères que l'on peut ensuite afficher pour pouvoir la tester.
La classe `MarquePage` se développait sans aucune difficulté. L'utilisation de la classe `StringBuffer` dans la méthode `toString` sera expliquée en section 5. La classe `Etiquette` ne posait pas de problèmes non plus, il fallait juste faire attention à ajouter une étiquette que l'on construit dans la liste des étiquettes filles de son étiquette mère. Comme cela se faisait dans le constructeur de la classe `Etiquette`, on pouvait accéder directement à l'attribut `filles` de l'étiquette mère. J'ai imposé l'utilisation d'une méthode `getMere` qui ne justifiait pas au vu du diagramme d'analyse présenté sur la figure 3, mais j'en ai besoin dans les tests que j'ai écrits pour la classe.
La classe `RechercheEtiquette` se développait plutôt facilement si l'on faisait attention aux indications de l'énoncé. L'algorithme de recherche d'une étiquette présenté dans la fonction recherche prend non seulement le nom de l'étiquette à chercher en paramètre, mais également l'étiquette à partir de laquelle on commence la recherche car c'est un algorithme récursif). J'ai donc créé une méthode *privée* qui est la copie conforme de la fonction recherche. Cette méthode est privée, car elle ne fait pas partie de l'*interface* de `RechercheEtiquette` et n'a pas a priori à être utilisée par un utilisateur extérieur.
Vous remarquerez que j'ai appliqué le même principe pour l'affichage de l'arbre dans la méthode `toString`. Une fois la classe `RechercheEtiquette` implantée, la classe `Afficheur` ne posait pas de difficulté particulière : elle utilise le même algorithme.
Je vous propose dans la section 5 de parler d'aspects un avancés de programmation, en particulier du test des méthodes privés de `RechercheEtiquette` et de `Afficheur`.

5 Pour aller plus loin...

5.1 Utilisation de la classe `StringBuilder`

Vous remarquerez que j'ai utilisé la classe `StringBuilder` au lieu de la classe `String` dans les méthodes `toString` de `MarquePage` et de `RechercheEtiquette`. Les objets de type `String` sont des objets qui ne peuvent pas être modifiés (immuables), lorsque l'on exécute le code suivant

```
public class ExempleString {
    public void essai() {
        String s = "coucou";
        s = s + " Christophe";
    }
}
```

on crée en fait trois objets de type `String` : `"coucou"`, `" Christophe"` et `"coucou Christophe"`. Lorsque l'on concatène un nombre important de chaînes de caractères représentées par des objets de type `String` en une seule chaîne de caractères, on va donc obtenir un nombre important d'objets intermédiaires qui ne servent à rien.
Pour pallier ce problème, on peut utiliser la classe `StringBuilder` (ou la classe `StringBuffer` qui a le même comportement mais qui est synchronisée) qui représente des chaînes de caractères modifiables. On ne crée donc ici qu'un seul objet de type `StringBuffer`, même si on concatène beaucoup de chaînes de caractères et on obtient un code plus rapide.

On remarquera toutefois que le compilateur du JDK à partir de la version 1.5 utilise automatiquement `StringBuilder` lorsque l'on utilise la concaténation de chaînes de caractères représentées par des objets de type `String`.

5.2 La classe `DataGenerator`

Pour éviter de dupliquer du code dans les tests unitaires que j'ai écrits, j'ai créé une classe `DataGenerator` qui me permettait :

- d'obtenir un ensemble de marque-pages ;
- d'obtenir un arbre d'étiquettes avec des marque-pages appartenant à l'ensemble précédemment créé.

Ces méthodes sont *statiques* : il n'y a pas besoin de créer une instance de `DataGenerator` pour les utiliser. Par exemple, on appelle `generateMarquePages` de la façon suivante : `DataGenerator.generateMarquePages()`.
La méthode `generateEtiquettes` appelle `generateMarquePages` pour obtenir un ensemble de marque-pages à affecter aux étiquettes. Ceci est plutôt maladroit : les marque-pages sont définis par un nom, une URL et une date de création (représentée par la classe `Date`). À chaque appel à `generateMarquePages`, on obtient un ensemble de marque-pages *différent* de celui obtenu à un appel précédent, car même si les marque-pages ont toujours le même nom et la même URL, la date de création change. Si l'on veut utiliser dans un test l'ensemble de marque-pages généré par `generateMarquePages` pour vérifier que l'on obtient bien les bons marque-pages dans l'arbre d'étiquettes généré par `generateEtiquettes`, cela ne fonctionnera pas : les comparaisons échoueront puisque les marque-pages ne sont pas identiques, les dates de création n'étant pas les mêmes.
La solution est de ne créer qu'une seule fois l'ensemble de marque-pages. Pour cela, j'utilise un principe souvent utilisé en informatique, celui du cache. Un attribut de `DataGenerator`, `mp`, contient l'ensemble des marque-pages. Au premier appel à `generateMarquePages`, on crée l'ensemble de marque-pages et on l'affecte à `mp`. Lorsque l'on appelle de nouveau `generateMarquePages`, on renvoie la valeur de `mp` et on travaille donc toujours avec les mêmes marque-pages.
Évidemment, le code de `DataGenerator` est très maladroit : il aurait mieux valu écrire une classe sans méthodes statiques et initialiser des attributs représentant l'ensemble de marque-pages et d'étiquettes via un constructeur.

5.3 Une méthode `ajouter` avec un nombre de paramètres variable

Si vous consultez la javadoc ou le code de classe `Etiquette`, vous remarquerez qu'elle possède une méthode `ajouter` un peu particulière. supplémentaire par rapport au modèle de conception proposé. Le code de cette méthode est le suivant :

```
public void ajouter(MarquePage... mp) {
    for (MarquePage m : mp) {
        this.mp.add(m);
    }
}
```

Dans un premier temps, on peut remarquer que le type du paramètre `mp` est `MarquePage...`. Les « `...` » ajoutés à un type de paramètre d'une méthode permettent d'indiquer que la méthode a un nombre d'arguments *variable* que l'on ne connaît pas forcément. On ne peut utiliser dans une méthode qu'un seul type utilisant `...` et c'est obligatoirement le dernier paramètre de la méthode. On pourra donc appeler `ajouter` avec un marque-page, deux marque-pages, trois marque-pages etc. et c'est cette méthode ajouter qui sera utilisée.
On récupére l'ensemble de marque-pages sous la forme d'un tableau d'éléments de type `MarquePage` que l'on peut ensuite parcourir avec une boucle **`for`** (de la même façon que pour les collections).

5.4 Tester des méthodes privées

Il y a un point délicat dans la classe `RechercheEtiquette` : la visibilité privée de certaines méthodes. Ces méthodes sont en effet des méthodes « auxiliaires » : un utilisateur extérieur n'a pas normalement à les utiliser. On dit qu'elles n'appartiennent pas à l'*interface* de `RechercheEtiquette`. Seule la méthode `rechercher(String nomEtiquette)` doit être accessible à un utilisateur extérieur (et bien sûr le constructeur de la classe et des méthodes comme `tosString`).
Se pose alors la question du test de ces méthodes privées. On ne peut plus en effet les appeler sur un acteur de type `RechercheEtiquette` dans un test JUnit (cf. classe `RechercheEtiquetteAdvancedTest`). Il existe toutefois un mécanisme d'*introspection* dans Java qui permet d'appeler ces méthodes. L'introspection consiste à pouvoir représenter par des *objets* Java les *classes* Java. On peut donc avoir un *objet* qui représente la *classe* `RechercheEtiquette`. Cet objet sera de type `Class`. La plupart de ces classes appartiennent au paquetage `java.lang.reflect`.
`Class` est une classe qui représente les classes Java. Elle possède des méthodes qui permettent par exemple de trouver les méthodes contenues dans la classe, mais également de créer des objets de la classe (sans l'opérateur **`new`** !) ou de créer des *objets* représentant les méthodes de la classe. Ces objets « méthodes » permettent d'appeler des méthodes sur des objets de la classe de façon *programmatique* et de passer outre le mécanisme de visibilité.

Par exemple, si on possède un objet `search` de type `RechercheEtiquette` initialisé avec le problème présenté sur la figure 1, on peut créer un objet de type `Class` représentant la classe `RechercheEtiquette` :

```
Class searchClass = this.search.getClass();
```

On peut ensuite récupérer un objet qui représente la méthode privée `rechercher` qui prend deux paramètres :

```
Method methodRechercher = searchClass.getDeclaredMethod("rechercher",
                                                         java.lang.String.class,
                                                         fr.isae.tags.Etiquette.class);
```

La méthode `getDeclaredMethod` prend en paramètres le nom de la méthode concernée sous forme d'une chaîne de caractères, puis des objets de type `Class` représentant les types des paramètres de la méthode. Vous remarquerez que l'on peut obtenir directement un objet de type `Class` à partir du nom *qualifié* d'une classe en utilisant le suffixe **`.class`**.
Reste que l'on ne pourra toujours pas appeler la méthode via ces objets car elle est privée. On va donc modifier la visibilité des méthodes via l'objet la représentant :

```
methodRechercher.setAccessible(true);
```

On a donc modifié de façon *programmatique* la visibilité de la « méthode » `rechercher`. On peut ensuite invoquer la méthode via la méthode `invoke` qui s'applique sur un objet de type `Method` :

```
Etiquette e;

e = (Etiquette) methodRechercher.invoke(this.search,
                                        "objet", this.tags.get(3));
assertEquals(this.tags.get(3), e);
```

La méthode `invoke` prend en paramètre l'objet sur lequel on applique la méthode et les arguments que l'on donnerait à la méthode. Deux remarques :

- il est possible avec Java d'avoir des méthodes dont le nombre d'arguments n'est pas défini. Par exemple, on ne sait pas à l'avance le nombre d'arguments de la méthode `invoke`, car cela dépend de la méthode représentée. Les créateurs du langage auraient également pu choisir d'utiliser un vecteur ou un ensemble pour représenter les arguments de la méthode (cf. section 5.3).
- le résultat de la méthode `invoke` est de type `Object`. Or on a besoin d'un objet de type `Etiquette` donc on utilise un mécanisme, le *transtypage*, que l'on étudiera dans les prochains cours.

Enfin, les méthodes `getDeclaredMethod` et `invoke` peuvent lever des *exceptions*, notion que l'on verra dans un prochain cours, ce qui explique la signature de la méthode de test de la classe `RechercheEtiquetteAdvancedTest`.

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