TP1 Patron composite et d'autres

Jean-Claude Royer

2 octobre 2015

1 Le sujet

Il s'agit de mettre en oeuvre au moins une fois le composite dans une version simple pour comprendre quelques subtilités qui échappent souvent à la première lecture.

1.1 Patron composite

Implémenter en Java une classe générique pour représenter des listes d'éléments du même type. Le diagramme UML est décrit dans la figure 1.

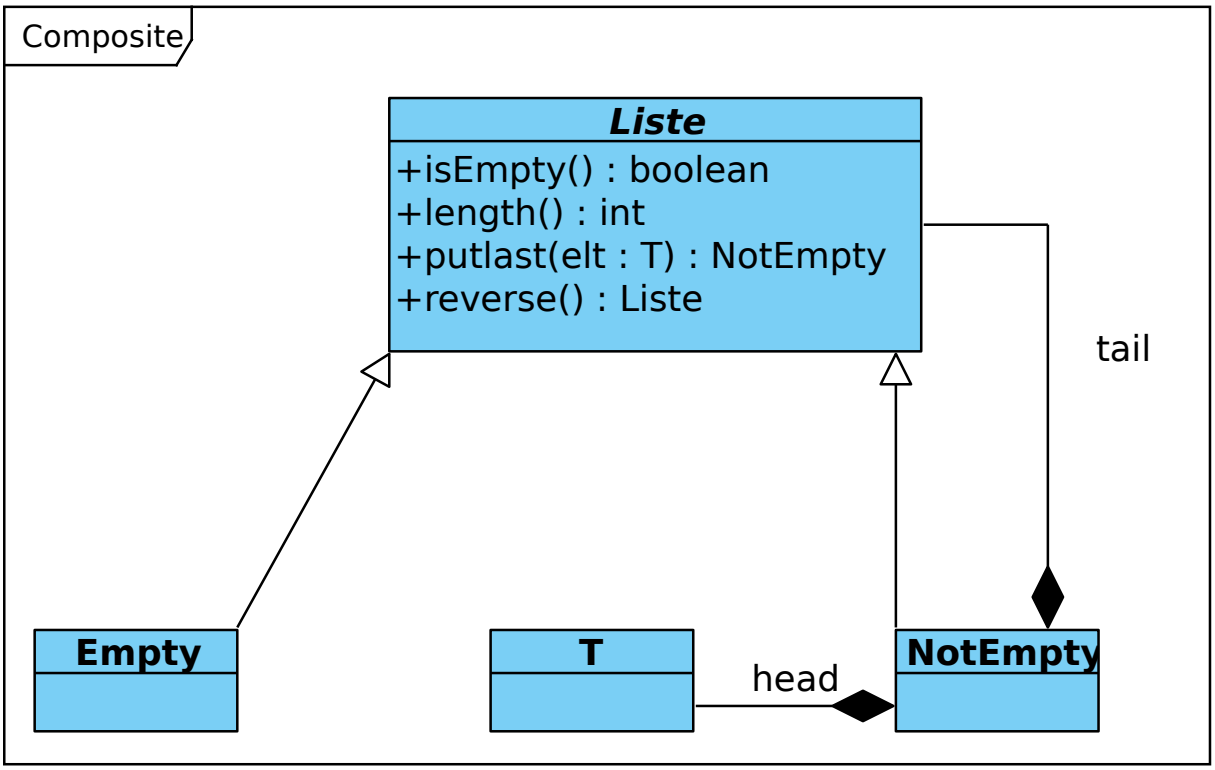

FIGURE 1 – Le composite

1. Vous devez compléter le diagramme avec les redéfinitions manquantes. Quelles règles pouvez vous ou devez vous appliquer pour ces redéfinitions ?

2. Le principe [1] sera de modéliser ce diagramme avec un outil UML et d'utiliser un générateur de code pour produire un squelette de l'implémentation. Vous pouvez utiliser ArgoUML ou Modelio, voire peut-être d'autres outils. La génération de code automatique n'est pas essentielle ici mais en général c'est un outil très utile.

Questions : quelle est la cardinalité de la composition `tail` ? Si elle est de 2 qu'est ce que ce diagramme représente ? comment l'implémenter ? Et si elle est `*` ?

3. Ensuite donner une implémentation en Java de ce diagramme. Dans un premier temps implémenter toutes les opérations dans un style fonctionnel pure (on n'utilise pas d'effet de bord, on construit si nécessaire de nouveaux objets).
4. Ajouter un patron singleton dans votre conception.

Réflexion : Dans le cas de l'étoile peut on implémenter ce cas sans utiliser un type de structure prédéfinie de Java (vector, liste chainée etc) ?

1.2 Pages et visiteur

Faire un composite spécifique représentant une liste de pages, une page sera représentée par un entier. Vous ferez une copie de votre code générique et remplacerez correctement les paramètres.

1. Ajouter une implémentation simple et récursive permettant le tri d'une liste de pages. Les principes sont les suivants :
   (a) Définir une opération d'insertion d'une page dans une liste déjà triée, par exemple `insert(3, [2, 4, 5] = [2, 3, 4, 5]`.
   (b) En utilisant cette opération et un parcours récursif définir une opération de tri d'une liste de pages quelconque.
2. Relire le cours et comprendre les consignes pour le visiteur
3. Ajouter un visiteur `visiteur2` qui compte le nombre de fois que 2 apparaît dans une liste.
4. Ajouter un visiteur `visiteurP` qui compte le nombre de fois que la page numéro p apparaît dans une liste.

2 Commentaires

2.1 Redéfinition des méthodes

Quand on parle de redéfinition de méthode en Java cela signifie observer des règles de typage entre les signatures de méthodes. Si dans la classe `S` vous avez `scope T m(A arg) {...}` alors une redéfinition dans une sous-classe `C` aura la déclaration `large R m(A arg) {...}`, avec `large` une visibilité supérieure ou égale à `scope` et `R` égale à `T` ou une sous-classe. Si vous ne respectez pas ces règles vous aurez une erreur de type ou un cas de liaison statique avec surcharge.

Un exemple classique qui peut poser problème à cet égard est le cas de `equal` qu'il faut redéfinir correctement en ne spécialisant pas son paramètre (qui reste donc `Object`).

1. Un principe, donc à défaut la génération peut-être faite entièrement manuellement, mais c'est moins drôle.

2.2 Généricité

Pour la gestion de `T` dans le diagramme : soit vous utilisée la généricité (bien) avec `<T>` ou vous remplacez par `Object` ou une classe. Pour instancier un paramètre générique il faut une classe (ex : `Integer`) et bien sûr utiliser l'opérateur de comparaison qui va avec. Le composite générique supporterait facilement un tri générique mais certaines restrictions existent (voir plus bas). Il est possible de définir le tri sur le composite générique en utilisant l'interface `Comparable`. Pour le passage du cas générique au cas particulier des pages plusieurs solutions sont possibles :

1. Faire une copie du code et remplacer le type correctement où il faut.
2. Utiliser l'héritage, attention cela peut être délicat.

Pour se convaincre et comprendre le schéma du composite je vous suggère de d'abord travailler sur papier. Est-ce que je sais construire une liste vide ? ou une liste avec 1, 2 et 3. Ensuite s'interroger sur le fonctionnement de `is_empty`, `length`, comment sur papier cela peut marcher. Une fois compris le codage en Java est simple.

2.3 Problème de typage

Le schéma du composite que vous choisissez impose des règles de typage et dans le cas simple cela entraîne des difficultés dans la définition de certaines opérations comme `getSecond`. Regardez la solution dans `WithPages` et n'oubliez pas d'utiliser les tests de types et les cast avec parcimonie et en connaissance de cause. Le principe est le suivant, avec la définition d'un accesseur pour le 2ème élément d'une liste dans le cas du composite ? Ce besoin apparaît avec la fonction de tri car il faut savoir comparer deux éléments successifs. Si vous n'avez pas vu ce problème là il est possible que vous ayez définit `head` sur les listes vides. La solution passe par l'utilisation d'un cast, mais ce concept ne doit être utilisé que si on est sûr que cela marche bien.

Je veux définir `getSecond` dans la classe `WithPage` et le problème est que le contrôleur de type dit que ce n'est pas correct.

```
return this.getRest().getFirst();
        ^      ^         ^
        |      |         |
   WithPage   Page    méthode inconnue
```

Si on regarde les annotations de type cela est clair car `getFirst()` est inconnue de `Page`. En fait il faut distinguer le cas où nous avons une seule page ou strictement plus d'une et cela peut se faire en testant si `this.getRest()` est vide ou pas. Ensuite il suffit de placer un "cast" au bon endroit, je vais écrire cela, mais ceci est justifié par le fait que le cast est appliqué sur le cas d'une liste non vide.

```
public int getSecond() throws PageException {
      if (this.getRest().isEmpty()) {
            throw new PageException();
      } else {
            // le cast est correct ici
            WithPage wp = (WithPage) this.getRest();
            return wp.getFirst();
      }
}
```

2.4 Truc pour le singleton

Un point également à noter avec la question sur le singleton. Avec la définition générique vous aurez

```
//private static final Empty<T> single = new Empty<T>();
// cannot make a static reference on a non static type T
```

Mais après instantiation il est tout à fait possible de créer un tel singleton.

```
/**
 * @author jcroyer
 *
 */
public class Singleton extends Empty<Integer> {

      private static final Singleton single = new Singleton();

      private Singleton() {}

      /**
       * @return
       */
      public Singleton getSingle() {
            return single;
      }
}
```

Voir http ://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createStatic

```
 If static fields of type parameters were allowed, then the following code would be confused:

MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();

Because the static field os is shared by phone, pager, and pc, what is the actual type of os?
It cannot be Smartphone, Pager, and TabletPC at the same time. You cannot, therefore,
create static fields of type parameters.
```

2.5 Etoile sans filet

La question est : est'il possible d'implémenter le cas de l'étoile sans utiliser des structures de données prédéfinies ? L'idée est d'essayer de représenter l'étoile par un composite en "bricolant" un peu. Mais le bricolage a ses limites dès que la complexité est en jeu, vous pouvez essayer ... Ici nous pouvons essayer d'autoriser (en plus des listes précédentes) de produire des listes qui s'imbriquent sans restriction.

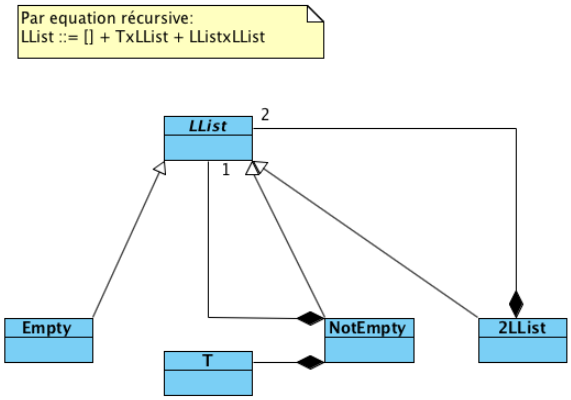

FIGURE 2 – Cas de l'étoile

Par exemple [1, [2, 3], [[4]]] peut-être représentée par :

```
new NotEmpty(1,
             new 2LList(
                 new NotEmpty(new NotEmpty(2, new NotEmpty(3, new Empty()))),
                 new 2LList(new 2LList(new NotEmpty(4, new Empty()))
                            new Empty())))
```