Examen
Programmation Objet en Java

A. Pigeau

11 mai 2015

L'examen est à réaliser en 2h00 en monôme. Seuls vos projets Eclipse java (TP par exemple) et l'API de Java sont autorisés. Le TP doit être réalisé sur un poste fixe de la salle de TP.

Aucune question n'est acceptée. Si le cahier des charges vous semble flou, indiquez vos propres hypothèses en commentaire dans le code.

Les codes sources doivent appartenir à un *package* portant le nom du monôme, contenant lui-même un package pour chaque exercice (-1pt si ce n'est pas le cas) . Une archive *nomDuMonome.zip*, comprenant vos fichiers sources et votre *javadoc* est à déposer sur Madoc en fin de séance.

1 Graphe [20pt]

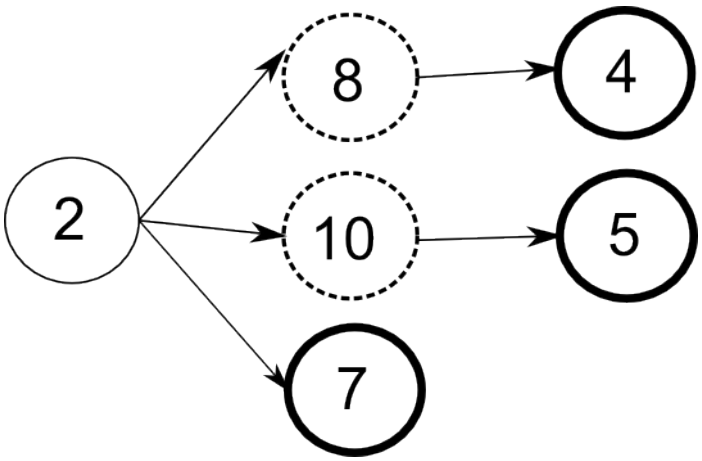

FIGURE 1 – Exemple de graphe. Le noeud 2 est l'état initial, les noeuds 8 et 10 sont des noeuds intermédiaires et les noeuds 4, 5 et 7 sont des états finaux.

L'objectif de l'exercice est de modéliser un graphe orienté. Le cahier des charges définit un noeud, un graphe et deux algorithmes de parcours de ce graphe.

Un noeud est défini par :
- une valeur de type entier
- un ensemble de noeuds suivants

- un type, défini parmi les valeurs suivantes : `INIT` (état initial), `INTER` (état intermédiaire) et `FINAL` (état final)
- un noeud peut être instancié en lui donnant sa valeur et son type
- un noeud peut être instancié en lui donnant seulement sa valeur. Dans ce cas, son type par défaut sera `INTER`
- l'opération d'ajout d'un noeud dans son ensemble de suivants doit être disponible
- l'opération de suppression d'un noeud dans son ensemble de suivants doit être disponible. La suppression se fera en passant la référence du noeud à supprimer.

Un graphe est défini par :

- un noeud de départ, ayant logiquement le type `INIT` mais non obligatoire (tout type de noeud est donc accepté)
- un type de parcours, permettant de définir comment sera parcouru le graphe, défini par les valeurs `LARGEUR` (parcours en largeur) et `PROFONDEUR` (parcours en profondeur)
- la classe `Graphe` doit vérifier l'interface `Iterable<Noeud>` afin de pouvoir être parcourue comme les collections. Dans cette interface, seule la méthode `iterator()` sera réellement implémentée, les autres méthodes lanceront une exception de type `UnsupportedOperationException`
  (l'interface `Iterable<T>` est différente suivant la version de votre compilateur - 1 seule méthode pour la version 7 mais 4 méthodes pour la version 8)
- la méthode `iterator()` retourne un objet permettant de parcourir le graphe :
  - si le type du graphe est `LARGEUR` alors un parcours en largeur doit être réalisé
  - si le type du graphe est `PROFONDEUR` alors un parcours en profondeur doit être réalisé
- une méthode `verifGraphe` doit vérifier que le graphe a bien un noeud de départ ayant le type `INIT` et que chaque noeud sans suivant est bien `FINAL`. Cette méthode lance une exception `ExceptionBadGraph` (à implémenter) si les deux conditions précédentes ne sont pas vérifiées. Rien n'est retourné si le graphe est correct.
- une méthode `getSum` doit retourner la somme des noeuds du graphe

Deux algorithmes de parcours sont possibles quand un graphe est itéré, un parcours en largeur ou en profondeur. Un algorithme de parcours de graphe est défini par :

- le graphe qu'il parcourt
- la classe d'un algorithme de parcours doit vérifier l'interface `Iterator<Noeud>`. Dans cette interface, seules les méthodes `next()` et `hasNext()` seront réellement implémentées, les autres méthodes lanceront une exception de type `UnsupportedOperationException`.
- la méthode `next()` retourne le noeud suivant du noeud courant
- la méthode `hasNext()` teste si le noeud courant a un noeud suivant

Suivant Wikipedia, l'algorithme de parcours en largeur utilise une file dans laquelle il prend le premier sommet et place en dernier ses voisins non encore explorés. L'algorithme est le suivant :

1. mettre le noeud de départ dans la file
2. retirer le noeud du début de la file pour l'examiner
3. mettre tous les voisins de ce noeud non examinés dans la file (à la fin)
4. si la file n'est pas vide reprendre à l'étape 2

Dans le cas où le graphe contient un cycle, il faut mémoriser les noeuds visités et ne pas ré-explorer un noeud déjà visité.

Pour le parcours en profondeur, il suffit d'utiliser une pile au lieu d'une file.

Questions : (barème indicatif)

1. lire l'ensemble des questions avant de commencer
2. (4 points) implémenter votre solution de modélisation d'un noeud
3. (1 point) réaliser un test en construisant le graphe de la figure 1. Chaque méthode implémentée sera testée et un affichage de chaque noeud sera réalisé.
4. (3 points) implémenter votre solution de modélisation d'un graphe
   (sans les méthodes `verifGraphe` et `getSum`)
5. (5 points) implémenter votre solution de modélisation des deux parcours de graphe. Attention au choix de vos conteneurs pour implémenter les algorithmes de parcours (la file et la pile sont des structures de données standards)
6. (1 point) réaliser un test de vos classes de graphe et parcours sur le graphe de la figure 1
7. (3 points) implémenter vos méthodes `verifGraphe` et `getSum` de la classe `Graphe`
8. (½ point) réaliser un test des méthodes `verifGraphe` et `getSum`
9. (1 point) si cela n'est pas déjà fait, améliorer votre code pour gérer les cycles dans le parcours d'un graphe
10. (½ point) réaliser un test de parcours de graphe sur un graphe contenant un cycle
11. (1 point) commenter le code de votre classe noeud et générer la Javadoc
12. zipper votre code source et votre javadoc dans un fichier *nomDuMonome.zip* et déposer le sur Madoc
13. vérifier que votre fichier *nomDuMonome.zip* déposé contient bien les sources de votre projet (et non pas les fichiers *.class*)