Interface des Programmes d'Application

Master Informatique
Université Bordeaux 1
27 juin 2006
Aucun document autorisé, durée 3 heures

Justifier clairement et succinctement vos choix, et décrivez brièvement vos algorithmes lorsque vous le jugez nécessaire.

Toute réponse difficile à comprendre sera considérée comme fausse.

Vous pouvez rajouter du code (méthode, donnée ou classe) non demandé à chaque fois que vous le jugerez nécessaire. Ecrivez entièrement ce code, en particulier dans le cas d'exceptions, de classes abstraites ou d'interfaces.

Vous pouvez toujours considérer une classe ou une méthode demandée dans une question précédente comme disponible, même si vous n'avez pas traité cette question. Il vous est également conseillé de lire le sujet entièrement avant de commencer votre travail.

Si vous souhaitez utiliser une classe (ou une méthode particulière) de l'API Java dont vous ne vous souvenez pas du nom, donnez-lui un nom explicite et précisez-le clairement sur votre copie.

Vous pouvez à chaque fois que vous le jugez compréhensible utiliser des abréviations à la place des noms complets de classe et méthode ou encore '...' pour remplacer du code non modifié.

Les instructions `import` et `package` ne seront pas précisées.

Un *graphe* est un couple $G = (V, E)$ où E est un ensemble de paires d'éléments de V. Les éléments de V sont appelés les *sommets* de G et ceux de E ses *arêtes*. Deux sommets u et v sont dits *voisins* s'il existe une arête $\{u, v\}$. Une arête qui possède deux fois le même sommet est appelée une *boucle*. Le degré du sommet u est le nombre d'arêtes qui contiennent u, une boucle étant comptée deux fois.

Lorsque le graphe est *orienté*, les deux sommets d'une arêtes sont distingués en un sommet *origine* et un sommet *destination*. On parle alors d'*arcs* et non plus d'arêtes.

Par exemple le graphe orienté de la figure a pour sommets les entiers de 1 à 5 et pour arcs $(3, 5), (2, 3), (2, 3), (2, 2), (2, 1), (1, 2)$. Le sommet 2 est de degré sortant 4 et de degré entrant 2. Ses successeurs sont 1, 2 et 3 et ses prédecesseurs 1 et 2.

On se donne les classes suivantes pour représenter un graphe : deux interfaces `GrapheOriente` et `Arc` et deux classes d'implémentation `GrapheOrienteImpl` et `ArcImpl`.

Un graphe est codé à l'aide de deux tables : `successions` et `precedences`. Dans la table `successions` (resp. `precedences`), les entrées sont tous les sommets du graphe. A chaque sommet `s` est associé une table `successions.get(s)` (resp. `precedences.get(s)`), éventuellement vide si `s` n'a pas de successeur (resp.

FIG. 1 – Graphe orienté G1

prédécesseur). Dans `successions.get(s)` (resp. `precedences.get(s)`), les entrées sont les successeurs (resp. prédécesseurs) de `s`. Si `t` et `t` (resp. `t` et `s`).

```
import java.util.Iterator;

public interface GrapheOriente {
    /** Nombre de sommets. */
    public int n();

    /** Nombre d'arcs. */
    public int m();

    /** Presence de o comme sommet du graphe. */
    public boolean contient(Object o);

    /** Iteration sur les sommets. */
    public Iterator<Object> sommets();

    /** Iteration sur les arcs. */
    public Iterator<Arc> arcs();

    /**
     * Degre sortant. Leve une IllegalArgumentException si s n'est pas un sommet
     * du graphe.
     */
    public int degreSortant(Object s);

    /**
     * Degre entrant. Leve une IllegalArgumentException si s n'est pas un sommet
```

```
     * du graphe.
     */
    public int degreEntrant(Object s);

    /**
     * Iteration sur les arcs sortants. Leve une IllegalArgumentException si s
     * n'est pas un sommet du graphe.
     */
    public Iterator<Arc> arcsSortants(Object s);

    /**
     * Iteration sur les arcs entrants. Leve une IllegalArgumentException si s
     * n'est pas un sommet du graphe.
     */
    public Iterator<Arc> arcsEntrants(Object s);

    /**
     * Iteration sur les successeurs. Leve une IllegalArgumentException si s
     * n'est pas un sommet du graphe.
     */
    public Iterator<Object> successeurs(Object s);

    /**
     * Iteration sur les predecesseurs. Leve une IllegalArgumentException si s
     * n'est pas un sommet du graphe.
     */
    public Iterator<Object> predecesseurs(Object s);

    /**
     * Iteration sur les arcs entre deux sommets. Leve une
     * IllegalArgumentException si l'un des objets n'est pas un sommet du
     * graphe.
     *
     */
    public Iterator<Arc> arcs(Object s1, Object s2);
}

public interface Arc {

        /**
         * Sommet origine.
         */
        public Object origine();

        /**
```

```
         * Sommet destination
         */
        public Object destination();

        /**
         * Sommet oppose de s. Si s n'est pas un sommet de l'arc, lve une
         * IllegalArgumentException.
         */
        public Object oppose(Object s);
}

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
import java.util.NoSuchElementException;
import java.util.Set;

public class GrapheOrienteImpl implements GrapheOriente {

    private int m = 0;

    private Map<Object, Map<Object, Set<Arc>>> successions =
        new HashMap<Object, Map<Object, Set<Arc>>>();

    private Map<Object, Map<Object, Set<Arc>>> precedences =
        new HashMap<Object, Map<Object, Set<Arc>>>();

    private boolean ajouterSommet(Object s) {
        if (successions.containsKey(s)) { // s est deja un sommet du graphe
            return false;
        }
        successions.put(s, new HashMap<Object, Set<Arc>>());
        precedences.put(s, new HashMap<Object, Set<Arc>>());
        return true;
    }

    private boolean ajouterArc(Arc a) {
        Object o = a.origine();
        Object d = a.destination();

        if (!contient(o)) {
            ajouterSommet(o);
        }
        if (!contient(d)) {
```

```
            ajouterSommet(d);
        }
        Set<Arc> succ = successions.get(o).get(d);
        Set<Arc> prec = precedences.get(d).get(o);
        if (succ == null) { // Pas d'arc existant de o vers d
            succ = new HashSet<Arc>();
            successions.get(o).put(d, succ);
            prec = new HashSet<Arc>();
            precedences.get(d).put(o, prec);
        }
        prec.add(a);
        m++;
        return succ.add(a); // true ssi a n'est pas deja presente.
    }

    /**
     * Cree un graphe oriente dont les sommets sont les elements de sommets
     * ainsi que les extremites des arcs non inclues dans sommets, et les arcs
     * les elements de arcs.
     */
    public GrapheOrienteImpl(Object[] sommets, Arc[] arcs) {
        for (Object s : sommets) {
            ajouterSommet(s);
        }
        for (Arc a : arcs) {
            ajouterArc(a);
        }
    }

    public int n() {
        return successions.size();
    }

    public int m() {
        return m;
    }

    public Iterator<Object> sommets() {
        return successions.keySet().iterator();
    }

    public Iterator<Arc> arcs() {
        return new Iterator<Arc>() {
            // iteration des arcs entre deux memes sommets.
            Iterator<Arc> incidence = null;
```

```
        // iteration sur les ensembles d'arcs sortants d'un meme sommet.
        Iterator<Set<Arc>> itSets = null;

        // iteration sur les tables de succession
        Iterator<Map<Object, Set<Arc>>> itSuccessions = successions
            .values().iterator();

        public boolean hasNext() {
            while (incidence == null || !incidence.hasNext()) {
                while (itSets == null || !itSets.hasNext()) {
                    if (itSuccessions.hasNext()) {
                        itSets = itSuccessions.next().values().iterator();
                    } else {
                        return false;
                    }
                }
                incidence = itSets.next().iterator();
            }
            return true;
        }

        public Arc next() throws NoSuchElementException {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return incidence.next();
        }

        public void remove() throws UnsupportedOperationException {
            throw new UnsupportedOperationException();
        }

    };
}

private void verifierPresence(Object s) throws IllegalArgumentException {
    if (!contient(s)) {
        throw new IllegalArgumentException();
    }
}

public Iterator<Object> successeurs(Object s) {
    verifierPresence(s);
    return successions.get(s).keySet().iterator();
```

```
}

public Iterator<Object> predecesseurs(Object s) {
    verifierPresence(s);
    return precedences.get(s).keySet().iterator();
}

public Iterator<Arc> arcs(Object s1, Object s2) {
    verifierPresence(s1);
    verifierPresence(s2);
    return successions.get(s1).get(s2).iterator();
}

public boolean contient(Object o) {
    return successions.containsKey(o);
}

public int degreEntrant(Object s) {
    int d = 0;
    verifierPresence(s);
    for (Iterator<Set<Arc>> its = precedences.get(s).values().iterator(); its
            .hasNext();) {
        d += its.next().size();
    }
    return d;
}

public int degreSortant(Object s) {
    int d = 0;
    verifierPresence(s);
    for (Iterator<Set<Arc>> its = successions.get(s).values().iterator(); its
            .hasNext();) {
        d += its.next().size();
    }
    return d;
}

public Iterator<Arc> arcsSortants(final Object s) {
    verifierPresence(s);
    return new Iterator<Arc>() {
        Iterator<Set<Arc>> its = successions.get(s).values().iterator();

        // iteration sur les arcs sortant vers une meme destination.
        Iterator<Arc> ita = null;
```

```
            public boolean hasNext() {
                while (ita == null || !ita.hasNext()) {
                    if (!its.hasNext()) {
                        return false;
                    } else {
                        ita = its.next().iterator();
                    }
                }
                return true;
            }

            public Arc next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                return ita.next();
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }

        };
    }

    public Iterator<Arc> arcsEntrants(final Object s) {
        verifierPresence(s);
        return new Iterator<Arc>() {
            Iterator<Set<Arc>> its = precedences.get(s).values().iterator();

            // iteration sur les arcs sortant vers une meme destination.
            Iterator<Arc> ita = null;

            public boolean hasNext() {
                while (ita == null || !ita.hasNext()) {
                    if (!its.hasNext()) {
                        return false;
                    } else {
                        ita = its.next().iterator();
                    }
                }
                return true;
            }

            public Arc next() {
```

```
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                return ita.next();
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }

        };
    }

    public String toString() {
        StringBuffer sb = new StringBuffer();
        for (Iterator<Object> itSommets = sommets(); itSommets.hasNext();) {
            Object s = itSommets.next();
            sb.append(s + " : ");
            for (Iterator<Arc> itSortants = arcsSortants(s); itSortants
                    .hasNext();) {
                Arc a = itSortants.next();
                sb.append(a.destination() + ", ");
            }
            sb.append('\n');
        }
        return sb.toString();
    }
}

public class ArcImpl implements Arc {
    private Object origine;

    private Object destination;

    public ArcImpl(Object origine, Object destination) {
        this.origine = origine;
        this.destination = destination;
    }

    public Object origine() {
        return origine;
    }

    public Object destination() {
        return destination;
```

```
    }

    public Object oppose(Object s) {
        if (s == origine) {
            return destination;
        } else if (s == destination) {
            return origine;
        } else {
            throw new IllegalArgumentException();
        }
    }

    public String toString() {
        return "(" + origine + "," + destination + ")";
    }
}
```

Partie 1 - Labyrinthe

On considère les interfaces `Labyrinthe` et `Salle` et les classes `Grille` et `SalleGrille`, qui implémentent ces interfaces.

On appellera par la suite *coordonnées* les valeurs des variables `i` et `j` d'une instance de `SalleGrille`.

Question 1 Compléter les méthodes `hauteur`, `largeur`, `entree` et `passage` de la classe `Grille`. La salle d'entrée sera la salle de coordonnées $i = 0$ et $j = 0$.

Question 2 Modifier la classe `SalleGrille` pour que l'instruction `System.out.print(s);` où `s` désigne une instance de `SalleGrille` de coordonnées i_0 et j_0 produise l'affichage sur la sortie standard de "(i_0,j_0)".

Quel sera alors le résultat de l'exécution des instructions suivantes :

```
Labyrinthe l = new Grille(2,3);
for (Iterator<Salle> it = l.salles(); it.hasNext();) {
        System.out.print(it.next() + " ");
}
System.out.println();
```

Question 3 Ecrire une classe `GrilleAvecMur` qui hérite de la classe `Grille` et dont le but est de permettre l'ajout de murs entre les salles d'une grille. L'ajout d'un mur horizontal (respectivement vertical) de coordonnées (i, j) empêchera le passage de la salle de coordonnées (i, j) à la salle de coordonnées $(i + 1, j)$ (respectivement $(i, j + 1)$).

La classe `GrilleAvecMur` possède deux nouvelles méthodes :
`public void ajouterMurHorizontal(int i, int j)` et
`public void ajouterMurVertical(int i, int j)`
et redéfinit la méthode
`public boolean passage(Salle s1, Salle s2)` en renvoyant vrai si et seulement si les salles `s1` et `s2` sont voisines et il n'existe pas de mur entre ces deux salles. Les murs seront mémorisés à l'aide de tableaux de type `boolean [][]`.

Partie 2 - Personnage

On considère l'interface `Personnage` qui représente un personnage dans un labyrinthe.

Question 1 Ecrire une classe `PersonnageDefaut` qui donne une implémentation de `Personnage`.

Question 2 Ecrire la classe `DeplacementInterditException`.

On considère l'interface `Deplacement` qui calcule un déplacement à chaque appel de la méthode `suivant`.

Question 3 Ecrire une classe `PersonnageAvecDeplacement` qui décore un personnage en lui ajoutant une méthode `public void deplacer()`. La méthode `deplacer()` déplace le personnage à l'aide de la méthode `suivant` d'une instance de `Deplacement`, passée en paramètre à la construction.

Question 4 Ecrire une classe `PersonnageAvecDessin` qui décore un personnage en lui ajoutant une méthode `public void dessine(Graphics g, int x, int y, int l, int h)` dessinant un "bonhomme" de hauteur h, largeur l aux coordonnées (x,y) d'un graphics g. Le code de la méthode `dessine` n'est pas demandé (on mettra ...).

Partie 3 - Plus Court Chemin

On dispose d'une classe `AlgorithmesGraphes` implémentant une méthode `public static Iterator<Sommet> plusCourtChemin(Sommet depart, Sommet destination)` qui renvoie un itérateur dont les éléments successifs constituent un plus court chemin entre les sommets depart et destination (depart et destination compris).

`Sommet` est une interface décrite ci-dessous.

Question 1 Ecrire une classe `SalleGrilleAdaptee` qui étend la classe `SalleGrille` et qui implémente `Sommet`.

Question 2 Ecrire une classe `GrilleAvecMurAdaptee` qui étend la classe `GrilleAvecMurs` et dont les salles sont des instances de `SalleGrilleAdaptee`.

Question 3 Ajouter à la classe `GrilleAvecMurAdaptee` une méthode `public Deplacement creerDeplacementPCC()` qui crée un déplacement dont chaque appel à la méthode `public Salle suivant(Salle s)` renvoie le sommet suivant d'un plus court chemin entre l'entrée de la grille et la sortie, ou `s` si aucun chemin n'existe.

Partie 4 - Application

On dispose d'une classe `DessinGrilleAvecMursEtPersonnage` qui étend la classe `java.swing.JComponent` et qui dessine une grille avec murs et un personnage si la position de celui-ci est non nulle.

FIG. 2 – Exemple de dessin de grille avec murs et personnage

Question 1 Expliquer en quelques phrases (10 maximum) le comportement de l'application `LabyrintheApplicationPCC` décrite ci-dessous.