Interfaces des Programmes d'Application - Corrigé

INF302 - Master Informatique
Université Bordeaux 1
27 juin 2006

Partie 1 - Graphes orientés

Question 1 - 1 point
Ecrire une classe `ArcImpl` qui implémente `Arc`.

```
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;
    }
}
```

Question 2 - 8 points Compléter la classe `GrapheOrienteImpl`. On terminera par la méthode `arcs()` plus complexe.

2 points pour n(), m(), contient(Object) et sommets(). Rendre l'ensemble non modifiable n'a pas été pris en compte.
1 point pour degreSortant() et degreEntrant(),
3 points pour les itérateurs sur les arcs et les sommets (autres que sommets(),
1 point pour l'utilisation correcte des exceptions,
1 point pour la non-duplication.

```
public class GrapheOrienteImpl implements GrapheOriente {
    ...
```

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

public int m() {
    return m;
}

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

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

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

private int degreAux(Object s, Map<Object, Map<Object, Set<Arc>>> map) {
    int d = 0;
    verifierPresence(s);
    for (Iterator<Set<Arc>> its = map.get(s).values().iterator(); its
             .hasNext();) {
        d += its.next().size();
    }
    return d;
}

public int degreEntrant(Object s) {
    return degreAux(s, precedences);
}

public int degreSortant(Object s) {
    return degreAux(s, successions);
}

private Iterator<Object> successeursOuPredecesseurs(
    Map<Object, Map<Object, Set<Arc>>> table, Object s) {
    verifierPresence(s);
    return Collections.unmodifiableSet(table.get(s).keySet()).iterator();
```

```
}

public Iterator<Object> successeurs(Object s) {
    return successeursOuPredecesseurs(successions, s);
}

public Iterator<Object> predecesseurs(Object s) {
    return successeursOuPredecesseurs(precedences, s);
}

private final Set<Arc> EMPTY_SET = Collections.emptySet();

public Iterator<Arc> arcs(Object s1, Object s2) {
    verifierPresence(s1);
    verifierPresence(s2);
    return (successions.get(s1).get(s2) == null ? Collections
            .unmodifiableSet(successions.get(s1).get(s2)) : EMPTY_SET)
        .iterator();
}

private Iterator<Arc> auxiliaireArcs(final Object s,
    final Map<Object, Map<Object, Set<Arc>>> map) {
    verifierPresence(s);
    return new Iterator<Arc>() {
        Iterator<Set<Arc>> its = map.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> arcsSortants(Object s) {
    return auxiliaireArcs(s, successions);
}

public Iterator<Arc> arcsEntrants(Object s) {
    return auxiliaireArcs(s, precedences);
}

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();
            }
        };
    }
}
```

Question 3 - 2 points Modifier les interfaces `GrapheOriente` et `Arc` pour paramétrer le type des sommets en utilisant la généricité (classes génériques).

```
public interface Arc<T> {
    /** Sommet origine. */
    public T origine();
    /** Sommet destination */
    public T destination();
}

public interface GrapheOriente<T> {
    /** Nombre de sommets. */
    public int n();
    /** Nombre d'arcs. */
    public int m();
    /** Presence de o comme sommet du graphe. */
    public boolean contient(T o);
    /** Iteration sur les sommets. */
    public Iterator<T> sommets();
    /** Iteration sur les arcs. */
    public Iterator<Arc<T>> arcs();
    /** Degre sortant. Leve une IllegalArgumentException si s n'est pas
     *  un sommet du graphe. */
    public int degreSortant(T s);
    /** Degre entrant. Leve une IllegalArgumentException si s n'est pas
     *  un sommet du graphe. */
    public int degreEntrant(T s);
    /** Iteration sur les arcs sortants. Leve une IllegalArgumentException
     *  si s n'est pas un sommet du graphe. */
    public Iterator<Arc<T>> arcsSortants(T s);
    /** Iteration sur les arcs entrants. Leve une IllegalArgumentException
     *  si s n'est pas un sommet du graphe. */
    public Iterator<Arc<T>> arcsEntrants(T s);
```

```
    /** Iteration sur les successeurs. Leve une IllegalArgumentException
     *  si s n'est pas un sommet du graphe. */
    public Iterator<T> successeurs(T s);
    /** Iteration sur les predecesseurs. Leve une IllegalArgumentException
     *  si s n'est pas un sommet du graphe. */
    public Iterator<T> predecesseurs(T 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<T>> arcs(T s1, T s2);
}
```

Expliquez brièvement (deux ou trois lignes) les modifications à apporter à `GrapheOrienteImpl`.

- *L'entête devient*
  `public class GrapheOrienteImpl<T> implements GrapheOriente<T> {`
- *Il faut remplacer dans le code les occurences d'* `Object` *par* `T` *et celles d'*`Arc` *par* `Arc<T>`.

.

Partie 2 - Graphes non orientés

Question 1 - 2 points Proposer deux interfaces `GrapheNonOriente` et `Arete` pour représenter un graphe non orienté.

```
public interface Arete<T> {
    public T sommet1();
    public T sommet2();
}

public interface GrapheNonOriente<T> {
        /** Nombre de sommets. */
        public int n();
        /** Nombre d'arcs. */
        public int m();
        /** Presence de o comme sommet du graphe. */
        public boolean contient(T o);
        /** Iteration sur les sommets. */
        public Iterator<T> sommets();
        /** Iteration sur les aretes. */
        public Iterator<Arete<T>> aretes();
        /** Degre. Leve une IllegalArgumentException si s n'est pas un sommet
         * du graphe. */
        public int degre(T s);
```

```
        /** Iteration sur les aretes incidentes. Leve une
         *  IllegalArgumentException si s n'est pas un sommet du graphe. */
        public Iterator<Arete<T>> aretes(T s);
        /** Iteration sur les voisins. Leve une IllegalArgumentException si s
         * n'est pas un sommet du graphe. */
        public Iterator<T> voisins(T s);
        /** Iteration sur les aretes entre deux sommets. Leve une
         * IllegalArgumentException si l'un des objets n'est pas un sommet du
         * graphe. */
        public Iterator<Arete<T>> aretes(T s1, T s2);
}
```

Question 2 - 3 points Donnez une implémentation de ces deux interfaces.

```
public class AreteImpl<T> implements Arete<T> {
    T s1;
    T s2;

    public AreteImpl(T s1, T s2) {
        this.s1 = s1;
        this.s2 = s2;
    }

    public T sommet1() {
        return s1;
    }

    public T sommet2() {
        return s2;
    }
}

public class GrapheNonOrienteImpl<T> implements GrapheNonOriente<T> {

    private GrapheOriente<T> delegue;

    public GrapheNonOrienteImpl(T[] sommets, Arete<T>[] aretes) {
        Arc<T>[] arcs = new ArcArete[aretes.length];
        int i = 0;
        for (Arete<T> a : aretes) {
            arcs[i++] = new ArcArete<T>(a);
        }
        delegue = new GrapheOrienteImpl<T>(sommets, arcs);
    }
```

```
public int n() {
    return delegue.n();
}

public int m() {
    return delegue.m();
}

public boolean contient(T o) {
    return delegue.contient(o);
}

public Iterator<T> sommets() {
    return delegue.sommets();
}

public Iterator<Arete<T>> aretes() {
    return new Iterator<Arete<T>>() {
        Iterator<Arc<T>> arcs = delegue.arcs();

        public boolean hasNext() {
            return arcs.hasNext();
        }

        public Arete<T> next() {
            return ((ArcArete<T>) arcs.next()).arete();
        }

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

public int degre(T s) {
    return delegue.degreEntrant(s) + delegue.degreSortant(s);
}

public Iterator<Arete<T>> aretes(T s) {
    final T t = s;
    return new Iterator<Arete<T>>() {

        Iterator<Arc<T>> sortants = delegue.arcsSortants(t);
        Iterator<Arc<T>> entrants = delegue.arcsEntrants(t);
```

```
        Arc<T> suivant = null;

        public boolean hasNext() {
            if (sortants.hasNext()) {
                return true;
            }
            while (suivant == null && entrants.hasNext()) {
                suivant = entrants.next();
                if (suivant.origine() == t) {
                    // boucle, deja retourne comme sortante
                    suivant = null;
                }
            }
            return suivant != null;
        }

        public Arete<T> next() {
            ArcArete<T> arc;
            if (sortants.hasNext()) {
                arc = (ArcArete<T>) sortants.next();
            } else if (!hasNext()) {
                throw new NoSuchElementException();
            } else {
                arc = (ArcArete<T>) suivant;
                suivant = null;
            }
            return arc.arete();
        }

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

public Iterator<T> voisins(T s) {
    final T t = s;
    return new Iterator<T>() {

        Iterator<T> successeurs = delegue.successeurs(t);
        Iterator<T> predecesseurs = delegue.predecesseurs(t);
        T suivant = null;

        public boolean hasNext() {
            if (successeurs.hasNext()) {
```

```
                return true;
            }
            while (suivant == null && predecesseurs.hasNext()) {
                suivant = predecesseurs.next();
                if (delegue.arcs(t, suivant) != null) {
                    // suivant a deja ete retourne comme successeur
                    suivant = null;
                }
            }
            return suivant != null;
        }

        public T next() {
            if (successeurs.hasNext()) {
                return successeurs.next();
            } else if (!hasNext()) {
                throw new NoSuchElementException();
            } else {
                T t = suivant;
                suivant = null;
                return t;
            }
        }

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

public Iterator<Arete<T>> aretes(T s1, T s2) {
    final T t1 = s1;
    final T t2 = s2;
    return new Iterator<Arete<T>>() {

        Iterator<Arc<T>> arcs12 = delegue.arcs(t1, t2);
        Iterator<Arc<T>> arcs21 = delegue.arcs(t2, t1);

        public boolean hasNext() {
            return arcs12.hasNext() || (t1 != t2 && arcs21.hasNext());
        }

        public Arete<T> next() {
            ArcArete<T> arc;
            if (arcs12.hasNext()) {
```

```
                    arc = (ArcArete<T>) arcs12.next();
                } else if (t1 == t2 || !arcs21.hasNext()) {
                    throw new NoSuchElementException();
                } else {
                    arc = (ArcArete<T>) arcs21.next();
                }
                return arc.arete();
            }

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

Partie 3 - Dessin de graphes

Question 1 - 2 points Proposer une solution utilisant l'héritage. Les sommets seront des instances de `Dessin` et les arêtes des instances à la fois de `Ligne` et d'`Arete`.

Remarque : il aurait été plus simple de ne pas mettre les arêtes instances de Ligne, mais simplement de faire un adaptateur d'objet (et non de classe comme dans la solution demandée).

```
public class AreteDessinable extends AreteImpl<Dessin> implements Ligne {

    private Ligne ligne;

    public AreteDessinable(Dessin s1, Dessin s2, Ligne l) {
        super(s1, s2);
        this.ligne = l;
    }

    public void dessiner(Graphics g, Point p1, Point p2) {
        ligne.dessiner(g, p1, p2);
    }
}

public interface GrapheDessinable extends GrapheNonOriente<Dessin> {
    public void dessiner(Graphics g);
}
```

```
public class GrapheDessinableImpl extends GrapheNonOrienteImpl<Dessin>
    implements GrapheDessinable {

    public GrapheDessinableImpl(Dessin[] sommets, AreteDessinable[] aretes) {
        super(sommets, aretes);
    }

    public void dessiner(Graphics g) {
        for (Iterator<Arete<Dessin>> ita =  aretes(); ita.hasNext();) {
            AreteDessinable a = (AreteDessinable) ita.next();
            a.dessiner(g, a.sommet1().coordonnees(), a.sommet2().coordonnees());
        }
        for (Iterator<Dessin> its =  sommets(); its.hasNext();) {
            its.next().dessiner(g);
        }
    }
}
```

Question 2 - 2 points Proposer une solution utilisant la décoration. Les instances de `Dessin` et de `Ligne` seront fournies à l'instanciation du décorateur.

```
public abstract class GrapheDecore<T> implements GrapheNonOriente<T> {

    private GrapheNonOriente<T> graphe;

    public GrapheDecore(GrapheNonOriente<T> graphe) {
        this.graphe = graphe;
    }

    public int n() {
        return graphe.n();
    }

    public int m() {
        return graphe.m();
    }

    public boolean contient(T o) {
        return graphe.contient(o);
    }

    public Iterator<T> sommets() {
        return graphe.sommets();
    }
```

```
    public Iterator<Arete<T>> aretes() {
        return graphe.aretes();
    }

    public int degre(T s) {
        return graphe.degre(s);
    }

    public Iterator<Arete<T>> aretes(T s) {
        return graphe.aretes(s);
    }

    public Iterator<T> voisins(T s) {
        return graphe.voisins(s);
    }

    public Iterator<Arete<T>> aretes(T s1, T s2) {
        return graphe.aretes(s1, s2);
    }
}

public class GrapheDessinableImpl<T> extends GrapheDecore<T>
    implements GrapheDessinable<T> {

    private Map<T, Dessin> dessinsSommets;
    private Map<Arete<T>, Ligne> dessinsAretes;

    public GrapheDessinableImpl(GrapheNonOriente<T> graphe,
        Map<T, Dessin> dessinsSommets,
        Map<Arete<T>, Ligne> dessinsAretes) {
        super(graphe);
        this.dessinsSommets = dessinsSommets;
        this.dessinsAretes = dessinsAretes;
    }

    public void dessiner(Graphics g) {
        for (Iterator<Arete<T>> ita = aretes(); ita.hasNext();) {
            Arete<T> a = ita.next();
            Ligne l = dessinsAretes.get(a);
            l.dessiner(g, dessinsSommets.get(a.sommet1()).coordonnees(),
                       dessinsSommets.get(a.sommet1()).coordonnees());
        }
        for (Iterator<T> its = sommets(); its.hasNext();) {
            dessinsSommets.get(its.next()).dessiner(g);
```

```
        }
    }
}
```

Quelle est l'avantage de cette solution par rapport à la précédente ?
Un même graphe peut avoir plusieurs dessins.