Interface des Programmes d'Application,
UE INF302
Corrigé

Master Informatique 1
Université Bordeaux 1

Partie 1 - Automate déterministe

Question 1 Rajouter les méthodes `public Etat etatInitial()` et `public Transition transition(Etat source, Object etiquette)`.

```
public Etat etatInitial() {
    return etatInitial;
}

public Transition transition(Etat e, Object etiquette) {
    return transitions.get(e).get(etiquette);
}
```

Question 2 Rajouter la méthode `public boolean reconnait(Object[] mot)`.

```
public boolean reconnait(Object[] mot) {
    Etat etatCourant = etatInitial();
    for (Object o : mot) {
        Transition transition = transition(etatCourant, o);
        if (transition == null) {
            return false;
        } else {
            etatCourant = transition.destination();
        }
    }
    return etatCourant.terminal();
}
```

Question 3 Modifier la classe `AutomateDeterministe` pour que le constructeur `AutomateDeterministe(Transition[] transitions)` lève une exception

- `TransitionNonDeterministeException` si deux éléments du tableau `transition` ont le même état source et la même étiquette ;

- `EtatInitialNonDeterministeException` s'il existe plusieurs états initiaux parmi les états sources et destination des transitions du tableau `transition` ;
- `EtatInitialInconnuException` s'il n'existe aucun état initial parmi les états sources et destinations des transitions du tableau `transition`.

```
public AutomateDeterministe(Transition[] transitions)
    throws TransitionNonDeterministeException, EtatInitialNonDeterministeException,
    EtatInitialInconnuException {
    this.transitions = new HashMap<Etat, Map<Object, Transition>>();
    for (Transition t : transitions) {
        insererEtat(t.source());
        insererEtat(t.destination());
        Map<Object, Transition> map = this.transitions.get(t.source());
        if (map.containsKey(t.etiquette())) {
            throw new TransitionNonDeterministeException(t, map.get(t.etiquette()));
        }
        map.put(t.etiquette(), t);
    }
    if (etatInitial == null) {
        throw new EtatInitialInconnuException();
    }
}

protected final void insererEtat(Etat e) throws EtatInitialNonDeterministeException {
    if (!transitions.containsKey(e)) {
        transitions.put(e, new HashMap<Object, Transition>());
        if (e.initial()) {
            if (etatInitial == null) {
                etatInitial = e;
            } else {
                throw new EtatInitialNonDeterministeException(e, etatInitial);
            }
         }
    }
}
```

Donner également le code de l'exception `TransitionNonDeterministeException`.

```
public class TransitionNonDeterministeException extends Exception {
    private final Transition t1;
    private final Transition t2;

    public TransitionNonDeterministeException(Transition t1, Transition t2) {
        this.t1 = t1;
        this.t2 = t2;
```

```
    }

    public Transition[] transitions() {
        return new Transition[] {t1, t2};
    }

    public String getMessage() {
        return "Transition dupliquée " + t1 + ", " + t2;
    }
}
```

Question 4 Donner le code des classes `TransitionImpl` et `EtatImpl`.

```
public class EtatImpl implements Etat {
    protected final boolean initial;
    protected final boolean terminal;

    public EtatImpl(boolean initial, boolean terminal) {
        this.initial = initial;
        this.terminal = terminal;
    }

    public boolean initial() {
        return initial;
    }

    public boolean terminal() {
        return terminal;
    }
}

public class TransitionImpl implements Transition {
    protected final Etat source;
    protected final Etat destination;
    protected final Object etiquette;

    public TransitionImpl(Etat source, Etat destination, Object etiquette) {
        this.source = source;
        this.destination = destination;
        this.etiquette = etiquette;
    }

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

```
    }

    public Object etiquette() {
        return etiquette;
    }

    public Etat source() {
        return source;
    }
}
```

Question 5 On souhaite rendre générique le type des étiquettes des transitions. Donner les classes à modifier et décrire en quelques lignes les modifications à apporter (sans réécrire tout le code).

```
public interface Transition<T> {
    ...
    public T etiquette();
}

public class TransitionImpl<T> implements Transition<T> {
    ....
    protected final T etiquette;

    public TransitionImpl(Etat source, Etat destination, T etiquette) {
        ....
        this.etiquette = etiquette;
    }

    public T etiquette() {
        return etiquette;
    }
    ....
}

public class AutomateDeterministe<T> {
   ...
}

public class TransitionNonDeterministeException extends
        Exception {
    private final Transition<?> t1;
    private final Transition<?> t2;
```

```
    public TransitionNonDeterministeException(Transition<?> t1, Transition<?> t2) {...}

    public Transition<?>[] transitions() {
        return new Transition<?>[] {t1, t2};
    }
    ...
}
```

Il faut également dans `AutomateDeterministe` remplacer `Transition` par `Transition<T>` et `Object` par `T`.

Partie 2 - Automate observable

Question 1 On souhaite qu'un `AutomateObservable` soit un `AutomateDeterministe`. Quel problème cela pose-t-il ?

En java, il est imposible d'hériter de deux classes. Donc si un `AutomateObservable` est un `AutomateDeterministe`, il ne peut pas être un `Observable`. Et donc il y a un problème pour appeler la méthode `setChanged()` qui est `protected`.

Question 2 Compléter le code d'`AutomateObservable`.

```
public class AutomateObservable<T> extends AutomateDeterministe<T> {

    public AutomateObservable(Transition<T>[] transitions)
            throws TransitionNonDeterministeException,
            EtatInitialNonDeterministeException, EtatInitialInconnuException {
        super(transitions);
    }

    private class ObservableInterne extends Observable {
        public void setChanged() {
            super.setChanged();
        }
    }

    private final ObservableInterne observable = new ObservableInterne();

    public boolean parcours(T[] mot) {
        Etat etatCourant = etatInitial();
        for (T o : mot) {
            Transition<T> transition = transition(etatCourant, o);
            observable.setChanged();
            observable.notifyObservers(transition);
            if (transition == null) {
```

```
                return false;
            } else {
                etatCourant = transition.destination();
            }
        }
        return etatCourant.terminal();
    }


    public void ajouterObservateur(Observer o) {
        observable.addObserver(o);
    }
}
```

Question 3 Ecrire une classe qui implémente `Observer` et dont le but est d'afficher sur la sortie standard la chaîne de caractère retournée par la méthode `String toString()` appliquée à l'étiquette de la transition traversée.

```
public class Afficheur implements Observer {
    public void update(Observable arg0, Object arg1) {
        if (arg1 == null) {
            System.out.println(" echec !");
        } else {
            System.out.print(((Transition)arg1).etiquette());
        }
    }
}
```

Partie 3 - Machine d'états finie

Question 1 Ecrire une classe `TransitionAvecAction<T>` qui implémente `Transition<T>` et qui associe une action a à une transition t.

```
public class TransitionAvecAction<T> implements Transition<T> {

    protected final Transition<T> delegue;
    protected final Action<T> action;

    public TransitionAvecAction(Transition<T> t, Action<T> a) {
        this.delegue = t;
        this.action = a;
    }
```

```
    public Etat destination() {
        return delegue.destination();
    }

    public T etiquette() {
        return delegue.etiquette();
    }

    public Etat source() {
        return delegue.source();
    }

    public Etat traverse() {
        action.execute(delegue.etiquette());
        return delegue.destination();
    }
}
```

Question 2 Ecrire une classe `MachineEtatsFinie<T>`.

Question 3 Rajouter la méthode `boolean parcours(Iterator<T> mot)` à la classe `MachineEtatsFinie<T>`.

```
public class MachineEtatsFinie<T> {

    private final AutomateDeterministe<T> delegue;

    public MachineEtatsFinie(TransitionAvecAction<T>[] transitions)
            throws TransitionNonDeterministeException,
            EtatInitialNonDeterministeException, EtatInitialInconnuException {
        delegue = new AutomateDeterministe<T>(transitions);
    }

    public boolean parcours(T[] mot) {
        return parcours(Arrays.asList(mot).iterator());
    }

    public boolean parcours(Iterator<T> mot) {
        Etat etatCourant = delegue.etatInitial();
        int i = 0;
        while (mot.hasNext()) {
            T t = mot.next();
            i++;
            TransitionAvecAction transition = (TransitionAvecAction) delegue
```

```
                    .transition(etatCourant, t);
            if (transition == null) {
                return false;
            } else {
                etatCourant = transition.traverse();
            }
        }
        return true;
    }
}
```

Question 4 Donner le code nécessaire à l'instanciation d'un `Iterator<String>` correspondant au mot infini $(ab)*$ (suite infinie de "a" et de "b" alternés).

```
Iterator<String> it = new Iterator<String>() {
    public boolean impair = false;

    public boolean hasNext() {
        return true;
    }

    public String next() {
        impair = !impair;
        return impair ? "a" : "b";
    }

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

Partie 4 - Conclusion

Question Comparer les deux solutions proposées (utilisation du modèle Observable/Observé et de transitions avec action) en donnant leurs avantages et leurs inconvénients.

L'avantage essentiel de l'utilisation d'observateurs est qu'il est possible d'avoir plusieurs observateurs (simultanés ou successifs) et donc plusieurs actions possibles lors du passage d'une transition.

L'avantage essentiel des transitions avec action est que chaque transition peut être associée avec une action différente. Si on utilise un observateur, il sera nécessaire pour un traitement équivalent de mettre un `switch case` qui peut être très gros et difficile à maintenir.