Université Bordeaux – Collège Sciences et Technologies — Jeudi 8 décembre 2016

Master Informatique — Examen UE: 4TIN706U – Approche Objet

Durée: 3 heures — Documents autorisés — Ce sujet comporte 5 pages

1 Questions de cours (2 points)

1. Dans quel(s) cas doit-on construire une classe abstraite ?
   — Dans le cas où cette classe est dérivée plus d'une fois vers des classes qui exploitent la même implémentation d'une méthode. En d'autres termes c'est pour partager du code.
2. Donner les conditions suivant lesquelles une classe est liée à une autre dans un diagramme de classes UML.
   — Une classe hérite d'une autre
   — Une classe contient une référence vers un objet instance d'une autre
   — Une classe contient une méthode dont le corps ou les arguments font référence vers un objet instance d'une autre

2 Questions conception objet (6 points)

Le jeu *Lemmings* est jeu vidéo des années 90 où le joueur doit guider une cinquantaine de personnages vers la sortie d'un labyrinthe parsemé de dangers mortels (cf figure 1). Au cours de la partie, chaque *lemming* est désigné pour une tâche particulière par le joueur (creuser, repousser les autres, grimper, construire un pont, sauter en parachute, etc.). Le but du jeu est de sauver le maximum de *lemmings* dans le temps imparti. Sans l'intervention du joueur, tous les *lemmings* tomberaient dans les pièges mortels ou seraient bloqués au lieu de rejoindre la sortie.

FIGURE 1 – Lemmings

Une équipe d'étudiants a écrit le code suivant pour réaliser en partie le jeu :

```
package lemmings;

public class LemmingsBoss {

        int[][] map;
        int[] lemmings;
        Point[] lemmingsCoordinates;
        int[] states;
        Point[] statesCoordinates;
        int actualState;
        boolean alive;

        public LemmingsBoss(){
                ...
        }

        public void run(){
                initMap();
                initLemmings();
                initStates();
                while (alive){
                        alive = false;
                        for (int i=0; i<states.length ; i++)
                                if (mouseClick(statesCoordinates[i])
                                        && states[i] != 0){
                                        actualState = states[i];
                                        break;
                                }
                        for (int i=0; i<lemmings.length ; i++){
                                if (mouseClick(lemmingsCoordinates[i])
                                        && actualState != 0){
                                        lemmings[i] = actualState;
                                }
                                moveLemming(i);
```

```
                                if (lemmings[i] != 0)
                                        alive = true;
                        }
                }
        }

        ...

}
```

Dans ce listing, pour gagner en place nous n'avons imprimé qu'une partie du code de la classe et laissé des pointillés à la place du code manquant.

Le plan est matérialisé par un double tableau d'entiers `map` où la valeur donne le type d'obstacles (0 pour vide, 1 pour de la pierre, 2 pour du métal, etc.).

Les *lemmings* sont matérialisés par leurs coordonnées (`lemmingsCoordinates`) et leur état (`lemmings`). L'état est un entier (0 signifie que le lemming est mort, 1 qu'il marche vers la droite, 2 vers la gauche, 3 qu'il grimpe, 4 qu'il creuse, etc.).

Le bouton permettant au joueur de sélectionner le prochain état est matérialisé par ses coordonnées (`statesCoordinates`). `actualState` est l'état courant que prendra le prochain *lemming* sélectionné (3 pour sélectionner un grimpeur, 4 pour un mineur, etc.).

La méthode `run()` initialise tous les éléments, puis lance une itération tant qu'il existe au moins un *lemming* en vie. L'itération passe en revue tous les boutons pour savoir si le joueur sélectionne un état. Puis elle passe en revue tous les lemmings pour

— changer leur état en fonction de l'état courant s'ils sont sélectionnés
— les faire évoluer en fonction de leur état
— vérifier qu'ils ne sont pas morts

1. Pourquoi cette classe a-t-elle une très faible cohérence ?
   — Parce qu'elle rassemble un ensemble d'attributs qui n'ont aucun lien entre eux. par exemple l'état d'un lemming n'a aucun lien avec le plan du jeu. Ces deux éléments doivent donc être séparés dans deux classes aux responsabilités différentes.
2. Pour augmenter la cohérence de cette classe, proposer une autre classe `Map` qui représente le plan du jeu.
   Il faut pour cela :
   — Construire une classe `Map`
   — Construire le constructeur `Map()` qui remplace `initMap()`
   — Coder une méthode qui permet de trouver un obstacle aux coordonnées x, y
   On propose ceci :

```
public class Map {
        Rectangle map;
        Map(int width, int height){
                map = new Rectangle(width, height);
        }
        boolean isObstacle(int x, int y){
                return map.isObstacle();
        }
}
```

3. Pour améliorer encore la conception de ce programme, créer une classe `Lemming` qui est responsable de l'état d'un *lemming* et de son déplacement.
   — La classe Lemming contient les coordonnées et l'état courant du *lemming*.
   — Elle doit avoir une référence vers `Map` pour connaître l'état du terrain où évolue le *lemming* (on doit agréger cet attribut).
   On propose ceci :

```
public class Lemming {
        Point coordinates;
        State state;
        void move(Map map, Vector speed){
```

```
                Point newCoo = coordinates.translation(speed);
                if (!map.isObstacle(newCoo))
                        coordinates.translation(newCoo);
        }
}
```

4. On cherche maintenant à utiliser la programmation événementielle pour que l'état courant soit assigné par l'événement généré par le clic sur le bouton correspondant. En utilisant la bibliothèque `javax.swing.JButton`, nous écrivons cela :

```
JButton climberButton = new JButton("Grimpeur");
climberButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
                actualState = 3;
        }
});
```

Adapter ce code pour l'insérer dans l'application en respectant au mieux le principe `MVC`.

— Il faut déplacer la partie qui concerne le nouvel état du lemming dans le modèle. Le contrôleur se charge de faire communiquer vue et modèle

— On propose que le contrôleur implémente ActionListener et demande à la vue les ajouts d'ActionListener lors de la construction (c'est ce que nous avions fait lors du développement de la calculette) :

Contrôleur :

```
public class Controleur {
        public Controleur(){
                view.addActionListener(this);
                ...
        }
        public void actionPerformed(ActionEvent e) {
                model.changeState(e.getActionCommand());
        }
        ...
}
```

Vue :

```
public class Vue {
        public void addActionListener(){
                climberButton.addActionListener(controleur)
                ...
        }
        ...
}
```

Modèle :

```
public class Modele{
        public void changeState(String actionCommand) {
                setState(actionCommand);
        }
        ...
}
```

3 Questions Design Pattern (5 points)

Les réponses à ces questions ne dépendent pas des réponses de la section précédente.

On souhaite maintenant ajouter une série de modifications au jeu *Lemmings* en introduisant des éléments qui affectent l'ensemble des personnages.

Ces éléments sont les suivants :

— Ajout d'un prédateur qui suit les *lemmings* et tente de les dévorer. La réaction des *lemmings* est de fuir en courant vers la direction opposée (en risquant de tomber dans l'un des pièges).

— Ajout de ressources qui permettent aux *lemmings* de boire et manger. Les *lemmings* se dirigent naturellement vers les ressources.

— Ajout de conditions météorologiques sur le plateau du jeu. Le déplacement des *lemmings* est paramétré par ces conditions météorologiques (déplacement ralenti, déplacement influencé par le vent, etc.)

Soit le code suivant :

```
public interface Observable {
        public void ajouterObservateur(Observateur o);
        public void supprimerObservateur(Observateur o);
        public void notifierObservateurs();
}
```

```
public interface Observateur {
        public void actualiser (Observable o);
}
```

1. Écrire la classe `Lemming` qui implémente l'interface `Observateur`. On se contentera d'y implémenter la méthode `void actualiser (Observable o)`.

   — Lemming doit être responsable des déplacements d'un lemming, de sa fuite, de la recherche de nourriture, etc. Il doit recevoir de façon asynchrone les informations venant de l'observable.

```
public class Lemming implements Observateur {

        @Override
        public void actualiser(Observable o){
                ObservableConcret op = (ObservableConcret) o;
                this.chases(op.getRessources());
                this.flees(op.getPredators());
                this.move(op.getBreeze());
        }

}
```

2. Écrire la classe `ObservableConcret` qui implémente l'interface `Observable` et influence chacun des *lemmings*.

   —

```
import java.util.ArrayList;

public class ObservableConcret implements Observable {

        private Predators predators;
        private Ressources ressources;
        private Freeze freeze;
        private ArrayList<Observateur> observateurs;

        public ObservableConcret() {
                observateurs = new ArrayList<Observateur>();
        }

        public Predators getPredators() {
                return predators;
        }

        public Ressources getRessources() {
```

```
                return ressources;
        }

        public Freeze getFreeze() {
                return freeze;
        }

        @Override
        public void ajouterObservateur(Observateur o) {
                observateurs.add(o);
        }

        @Override
        public void supprimerObservateur(Observateur o) {
                observateurs.remove(o);
        }

        @Override
        public void notifierObservateurs() {
                for (Observateur o : observateurs)
                        o.actualiser(this);
        }

}
```

4 Questions interface graphique (3 points)

Soit l'interface suivante qui permet à un utilisateur de se connecter à une base de données :

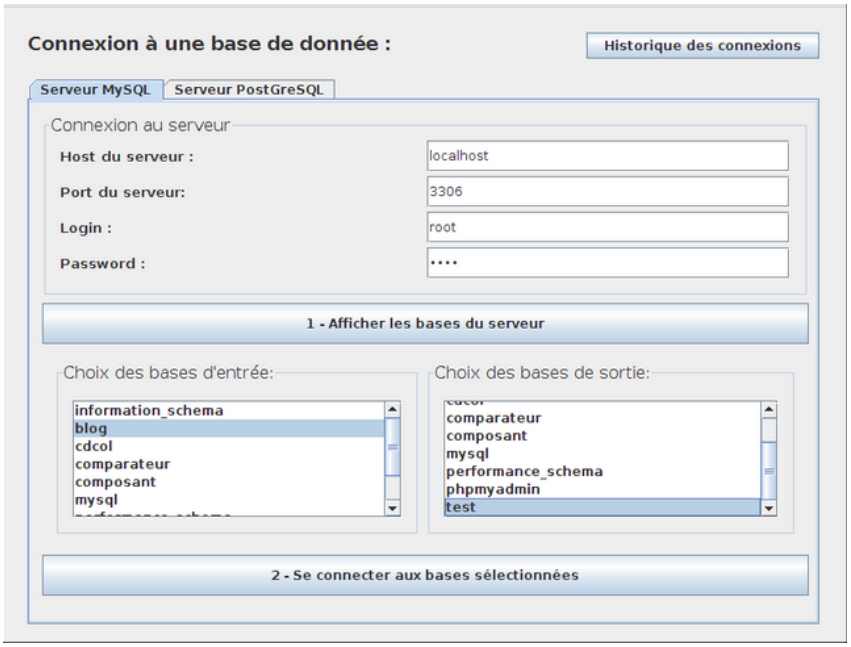

FIGURE 2 – Interface graphique (source : Jonathanhaehnel.fr)

— Décrivez sous forme schématique (à l'aide d'un arbre) l'ensemble des éléments de la bibliothèque `javax.swing` permettant d'obtenir cette interface.

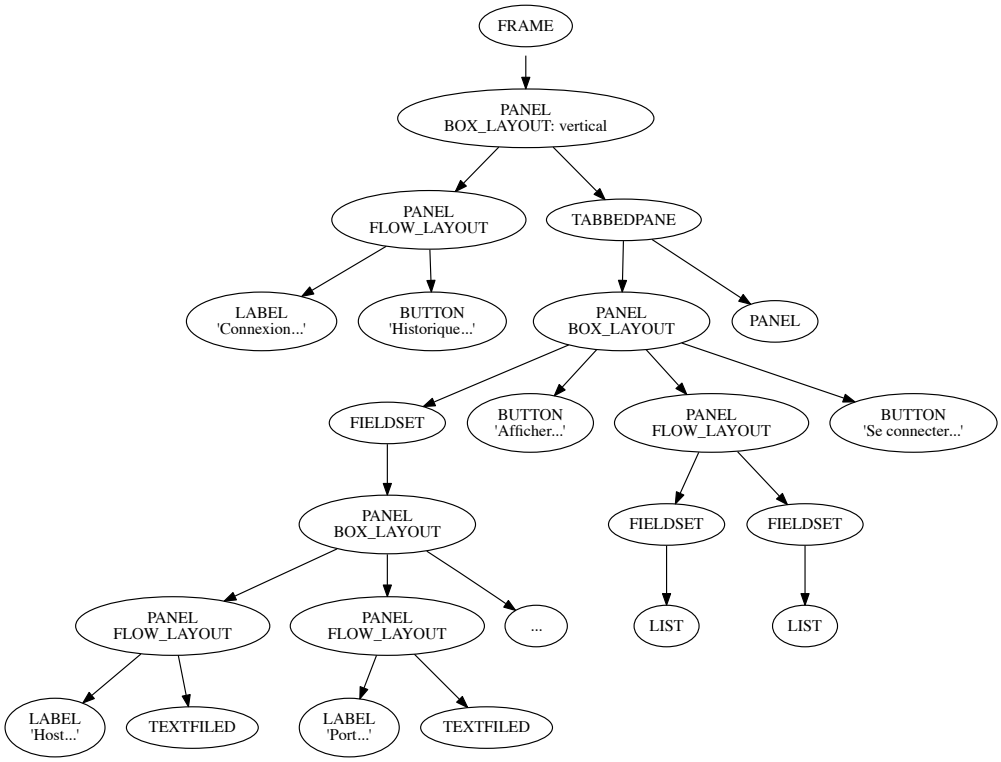

FIGURE 3 –

5 Questions thread (4 points)

L'algorithme de tri par fusion revient à trier deux sous parties du même tableau, puis de les fusionner ensemble. La complexité de la fusion étant linéaire et celle de la sous-division en deux parties égales logarithmique, nous obtenons un algorithme de tri en $O(n \times log(n))$

La classe suivante [1] décrit une implémentation en Java :

```
public class Trieur {

  private int[] t; // tableau à trier
  private int debut, fin; // tranche de ce tableau qu'il faut trier

  public Trieur(int[] t) {
    this(t, 0, t.length - 1);
  }

  private Trieur(int[] t, int debut, int fin) {
    this.t = t;
    this.debut = debut;
    this.fin = fin;
  }

  public void run() {
    if (fin - debut > 1) {
      int milieu = debut + (fin - debut) / 2;
      Trieur trieur1 = new Trieur(t, debut, milieu);
      Trieur trieur2 = new Trieur(t, milieu + 1, fin);
      trieur1.run();
```

1. http ://deptinfo.unice.fr/ grin

```
      trieur2.run();
      fusionner(debut, fin);
    }

  }

  /**
   * Fusionne 2 tranches déjà triées du tableau t.
   *   - 1ère tranche : de debut à milieu = (debut + fin) / 2
   *   - 2ème tranche : de milieu + 1 à fin
   * @param debut premier indice de la 1ère tranche
   * @param fin dernier indice de la 2ème tranche
   */
  private void fusionner(int debut, int fin) {
    // ...
  }

}
```

Nous remarquons que le tri des sous-parties étant indépendants, nous pouvons les confier à deux *threads* différents.

— Modifier le code pour réaliser le tri des sous-parties dans deux threads différents. Les deux sous-parties devant être totalement triées avant la fusion, utiliser le mot clef `join` pour synchroniser les deux événements.

```
// On étend la classe à la classe Thread
public class Trieur extends Thread {

  private int[] t; // tableau à trier
  private int debut, fin; // tranche de ce tableau qu'il faut trier

  public Trieur(int[] t) {
    this(t, 0, t.length - 1);
  }

  private Trieur(int[] t, int debut, int fin) {
    this.t = t;
    this.debut = debut;
    this.fin = fin;
  }

  public void run() {
    if (fin - debut > 1) {
      int milieu = debut + (fin - debut) / 2;
      Trieur trieur1 = new Trieur(t, debut, milieu);
      // On lance le thread trieur1
      trieur1.start();
      Trieur trieur2 = new Trieur(t, milieu + 1, fin);
      // On lance le thread trieur2
      trieur2.start();
      // On attend que les deux threads soient terminés
      try {
        trieur1.join();
        trieur2.join();
      }
      catch(InterruptedException e) {}
      // Puis on fait la fusion
      triFusion(debut, fin);
    }
  }
```

```
  private void triFusion(int debut, int fin) {
        ...
 }

}
```