Université Bordeaux – Collège Sciences et Technologies — Jeudi 29 juin 2017

Master Informatique — Examen UE: 4TIN706U – Approche Objet

Durée: 3 heures — Documents autorisés — Ce sujet comporte 8 pages

# 1 Questions de cours (2 points)

1. Dans quel(s) cas doit-on construire un héritage entre deux classes ?
   - Extension
   - Adaptation
   - Factorisation
2. Dans quel(s) cas doit-on surcharger une méthode ?
   - Dans le cas où la sémantique de la méthode dépend du type de ses arguments

# 2 Questions conception objet (6 points)

Le jeu *Pac-Man* est jeu vidéo sorti en 1980 où le joueur doit conduire un petit personnage dans toutes les galeries d'un labyrinthe en mangeant des fruits et en évitant d'être touché par des fantômes.

FIGURE 1 – Pac-Man

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

```
public enum CardinalPoint {NORTH, EAST,SOUTH,WEST;}

import java.util.ArrayList;

public class DagNode {
        Point coordinate;
        ArrayList<DagNode> successors;
}

public class Point {

        private int x;
        private int y;

        public boolean collision(Point p) {
                return p.x==x && p.y==y;
        }

        public void step(CardinalPoint pacmanDirection, DagNode labyrinth) {
                ...
        }
}


public class Pacman {

        enum State {KIND,NASTY;}

        DagNode labyrinth;
        State ghostState;
        Point [] ghostCoordinates;
        CardinalPoint pacmanDirection;
        Point pacmanCoordinate;
        int lives;

        PacmanMain(){
                initLabyrinth();
                initGhosts();
                initPacman();
        }

        public void run(){
                while (lives > 0){
                        pacmanDirection = decodeToDirection(keyboard());
                        pacmanCoordinate.step(pacmanDirection, labyrinth);
                        ghostState = randomChangeState();
                        for (int i=0; i<=4; i++){
                                ghostCoordinates[i] = labyrinthStep(ghostCoordinates[i]);
                                if (ghostCoordinates[i].collision(pacmanCoordinate)){
                                        lives--;
                                        tryAgain();
                                }
                        }
                }
        }
        ...
}
```

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

Le labyrinthe est implémenté sous la forme d'un graphe acyclique où chaque cellule contient un pointeur vers les cellules adjacentes.

Les coordonnées cartésiennes sont implémentées par la classe `Point`.

Les directions sont représentées par les points cardinaux et implémentées par l'énuméré `CardinalPoint`.

Les fantômes, au nombre de 4, sont représentés par leurs coordonnées et l'état qu'ils prennent ensemble (énuméré State).

Le pacman (personnage que dirige le joueur) est matérialisé par ses coordonnées et la direction qu'il prend.

Pour le reste, le code se laisse déchiffrer de lui-même en devinant l'usage des différentes méthodes.

1. Pourquoi la classe `Pacman` a-t-elle une très faible cohérence ?

   — Parce qu'elle rassemble un ensemble d'attributs qui n'ont aucun lien entre eux.

2. Pour augmenter la cohérence de cette classe, proposer une classe `Map` qui représente le labyrinthe et qui contient les méthodes permettant de le parcourir sans traverser les murs. Il est inutile d'implémenter le code, seule la conception orientée objet nous intéresse.

3. Pour améliorer encore la conception de ce programme, créer une classe `Ghost` qui est responsable d'un fantôme, de son état et de son déplacement.

4. On cherche maintenant à utiliser la programmation événementielle pour que l'état des 4 fantômes soient tous synchronisés et dépendent de l'action du Pacman :

   Quand le Pacman passe sur l'un des fruits disposés dans le labyrinthe, les 4 fantômes passent ensemble de l'état `KIND` à l'état `NASTY`.

   Écrire les classes `Ghost` et `Pacman` pour permettre cette synchronisation.

   Ici encore, le détail de l'implémentation des algorithmes ne nous intéresse pas, mais seulement la conception et l'organisation des classes et des envois de messages.

```java
import java.util.Observable;

public class Pacman extends Observable{

        enum State {KIND, NASTY};
        State state;

        Pacman(){
                state = State.KIND;
        }

        void becomeKind(){
                state = State.KIND;
                setChanged();
                notifyObservers();
        }

        void becomeNasty(){
                state = State.NASTY;
                setChanged();
                notifyObservers();
        }

}
```

```java
import java.util.Observable;
import java.util.Observer;

public class Ghost implements Observer {

        Ghost(Pacman pacman){
                pacman.addObserver(this);
        }
```

```
        @Override
        public void update(Observable o, Object arg) {
                System.out.println("I am the Observer " + this.toString());
                System.out.println("The Observable is " + o.toString()
                + " he is " + ((Pacman)o).state);
        }


}
```

```
public class Main {

        public static void main(String[] args) {
                Pacman pm = new Pacman();
                Ghost gh1 = new Ghost(pm);
                Ghost gh2 = new Ghost(pm);
                pm.becomeKind();
                pm.becomeNasty();
        }

}
```

# 3 Questions Design Pattern (5 points)

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

Dans le jeu `Pacman`, deux états seulement permettent de gérer le comportement des fantômes : `NASTY` où les fantômes pourchassent Pacman et le menacent, `KIND` où les fantômes fuient et sont menacés à leur tour.

Nous allons ajouter des états supplémentaires pour gérer l'ensemble du jeu : création de nouveaux fantômes, leur mort, la mort de Pacman, l'accélération du jeu, etc.

L'automate des états est représenté dans la figure suivante :

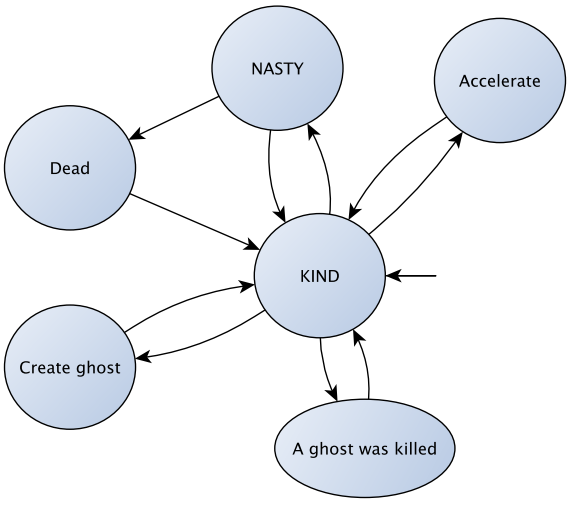


FIGURE 2 – States

Soit le code suivant :

```
public class Pacman {
        ...
        public void chases() {
          ...
        }
        public void flees() {
          ...
        }
}
```

```
public class Context {

        private Pacman pacman;
        State state;

        public Context(){
                state = new ConcreteStateKind();
        }

        public Pacman getPacman() {
                return pacman;
        }

        ...
}
```

Où `Context` implémente l'état courant du jeu (`Kind`, `Nasty`, etc.).

1. Écrire l'interface `State` qui décrit un état quelconque du jeu. On y ajoutera la méthode `transition()` qui provoque une modification de l'état courant en fonction du contexte et `run()` qui dicte le comportement de Pacman en fonction de l'état courant.

```
public class Context {

        private Pacman pacman;
        State state;

        public static interface State {
                static void setState (Context context, State state){
                        context.state = state;
                }
                public void transition(Context context);
                public void run(Context context);
        }

        public Context(){
                state = new ConcreteStateKind();
        }

        public Pacman getPacman() {
                return pacman;
        }

}
```

2. Écrire la classe `ConcreteStateKind` qui implémente l'interface `State`. Cette classe doit implémenter les méthodes `transition()`, et `run()`.

   Les détails de l'encodage des algorithmiques ne sont pas nécessaires, seule la conception sera évaluée.

```
package state;
```

```java
import state.Context.State;

public class ConcreteStateKind implements State {

        @Override
        public void transition(Context context) {
          // selon comportement de context
          if (...)
             context.state = new ConcreteStateKind();
        }

        @Override
        public void run(Context context) {
                context.getPacman().chases();
        }


}
```

# 4 Questions interface graphique (3 points)

Soit le code suivant :

```
import javax.swing.JFrame;

public class Main
    {
        public static void main(String [] ergs)
        {
            JFrame f = new Mystery();
            f.setVisible(true);
        }
    }
```

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

    class Mystery extends JFrame implements ActionListener
    {
        private JButton b1, b2, b3;
        private JPanel panneau; private JTextField champ;

        public Mystery()
        {
            setTitle("Tester les boutons");
          setSize(250, 150);
            Container c = getContentPane();

            panneau = new JPanel();
            c.add(panneau, "West");
            champ = new JTextField();
            c.add(champ);

            Box boxVertical = Box.createVerticalBox();
            panneau.add(boxVertical);

            b1 = new JButton("bouton 1");
            boxVertical.add(b1);
            b1.addActionListener(this);

            b2 = new JButton("bouton 2");
            boxVertical.add(b2);
            b2.addActionListener(this);

            b3 = new JButton("bouton 3");
            boxVertical.add(b3);
            b3.addActionListener(this);

        }

        public void actionPerformed(ActionEvent a)
        {
            if(a.getSource() == b1)
            {
                champ.setText("clic sur bouton 1");
            }
            else if(a.getSource() == b2)
            {
                champ.setText("clic sur bouton 2");
            }
```

```
                else
                {
                    champ.setText("clic sur bouton 3");
                }
            }
        }
```

— Décrivez sous la forme d'un schéma l'interface graphique proposée par ce listing. Expliquer l'usage des boutons.

# 5 Questions thread (4 points)

Soit le code suivant :

```
package thread;

public class Compte {

        private int solde;

        Compte(int solde){
                this.solde = solde;
        }

        public  void retrait(int somme, String personne){
                if (this.solde-somme < 0)
                        System.out.println("Retrait impossible de " +
                        somme + " par " + personne);
                else {
                        solde -= somme;
                        System.out.println("Retrait de " + somme + "
                                        effectué par " + personne + ", nouveau solde: " + solde);
                }
        }

        public  void depot(int somme, String personne){
                solde += somme;
                System.out.println("Dépot de " + somme + " effectué
                                par " + personne + ", nouveau solde: " + solde);
        }


}
```

```
package thread;

import java.util.Random;

public class RunImp implements Runnable {

        private Compte compte;
        private String name;

        RunImp(Compte compte, String name){
                this.compte = compte;
                this.name = name;
        }

        @Override
        public void run() {
                for (int i = 1; i<=4; i++){
                        Random rand = new Random();
                        compte.depot(rand.nextInt(10), name);
                        compte.retrait(rand.nextInt(10), name);
                }
        }
}
```

```
package thread;

public class Test {
```

```
        public static void main(String[] args) {
                Compte compte = new Compte(0);
                Thread utilisateur1 = new Thread(new RunImp(compte, "Jean"));
                Thread utilisateur2 = new Thread(new RunImp(compte, "Marie"));
                utilisateur1.start();
                utilisateur2.start();
        }

}
```

En effectuant un essai, le résultat affiché ne semble pas correct ; les calculs du solde sont erronés.

```
Dépot de 5 effectué par Marie, nouveau solde: 11
Retrait de 1 effectué par Marie, nouveau solde: 10
Dépot de 7 effectué par Marie, nouveau solde: 17
Retrait de 5 effectué par Marie, nouveau solde: 12
Dépot de 2 effectué par Marie, nouveau solde: 14
Retrait de 7 effectué par Marie, nouveau solde: 7
Dépot de 6 effectué par Marie, nouveau solde: 13
Retrait de 7 effectué par Marie, nouveau solde: 6
Dépot de 6 effectué par Jean, nouveau solde: 11
Retrait de 1 effectué par Jean, nouveau solde: 5
Dépot de 0 effectué par Jean, nouveau solde: 5
Retrait impossible de 7 par Jean
Dépot de 9 effectué par Jean, nouveau solde: 14
Retrait de 6 effectué par Jean, nouveau solde: 8
Dépot de 0 effectué par Jean, nouveau solde: 8
Retrait de 6 effectué par Jean, nouveau solde: 2
```

— Expliquer l'origine des erreurs de calcul.
— Apporter une correction au code pour éviter ces erreurs.

```
package thread;

public class Compte {

        private int solde;

        Compte(int solde){
                this.solde = solde;
        }

        public synchronized void retrait(int somme, String personne){
                if (this.solde-somme < 0)
                        System.out.println("Retrait impossible de " + somme + " par " + p
                else {
                        solde -= somme;
                        System.out.println("Retrait de " + somme + " effectué par " + pers
                }
        }

        public synchronized void depot(int somme, String personne){
                solde += somme;
                System.out.println("Dépot de " + somme + " effectué par " + personne + ",
        }


}
```