EXAMEN
Design Pattern 2014-2015

ING-2, Génie Informatique

Durée de l'examen : 2h
Conditions d'examen :

- Accès à une feuille recto-verso seulement (aucune machine n'est tolérée).

A- Concepts de base (5 points)

Question 1)

Qu'est-ce que l'héritage et la composition ? Qu'est-ce qui distingue ces deux approches ?

Question 2)

Indiquez des différences entre les *design patterns* Méthode de fabrique et Fabrique abstraite.

Question 3)

Dans un simulateur de jeu de course, des obstacles apparaissent aléatoirement. De même, le véhicule peut être endommagé ou tomber en panne. Selon la situation, la performance du pilote est affectée et les options du jeu évoluent. Justifiez votre choix de *design pattern*.

____
____
____
____
____
____
____

Question 4)

Y a-t-il des liens entre le patron MVC et les *design patterns* vus au cours ? Expliquez.

____
____
____
____
____
____
____
____

Question 5)

Décrivez un cas d'application pour lequel le *design pattern* Visiteur serait le plus approprié.

____
____
____
____
____
____
____

B- Problème :*Crackout* (15 points)

Crackout est un jeu de casse-briques édité par *Konami* et sorti sur la console *Nes* en 1991. Le but du jeu est de détruire toutes les briques d'un niveau. Pour cela, une balle traverse l'écran, rebondissant sur la gauche, la droite et le haut de l'écran. Lorsque la balle touche une brique, elle rebondit et la brique est détruite. Cette destruction rapporte un certain nombre de points au joueur et peut également déclencher certains effets (*e.g.,* réduire la vitesse de la balle), en fonction du type de brique détruite. Le joueur perd une vie lorsque la balle atteint le bas de l'écran. Pour éviter cela, il contrôle une raquette de gauche à droite afin de faire rebondir la balle vers le haut. La figure 1 montre une capture d'écran du jeu.

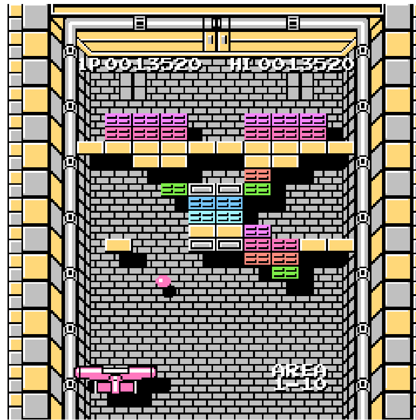

Figure 1 - Capture d'écran du jeu *Crackout*

Question 1)

La figure 2 illustre un extrait de l'architecture logicielle du jeu *Crackout*. Identifiez-y trois *design patterns*, ainsi que les classes, les méthodes et les attributs liés à chacun d'eux. Un extrait d'une implémentation de cette architecture est fourni en annexe.

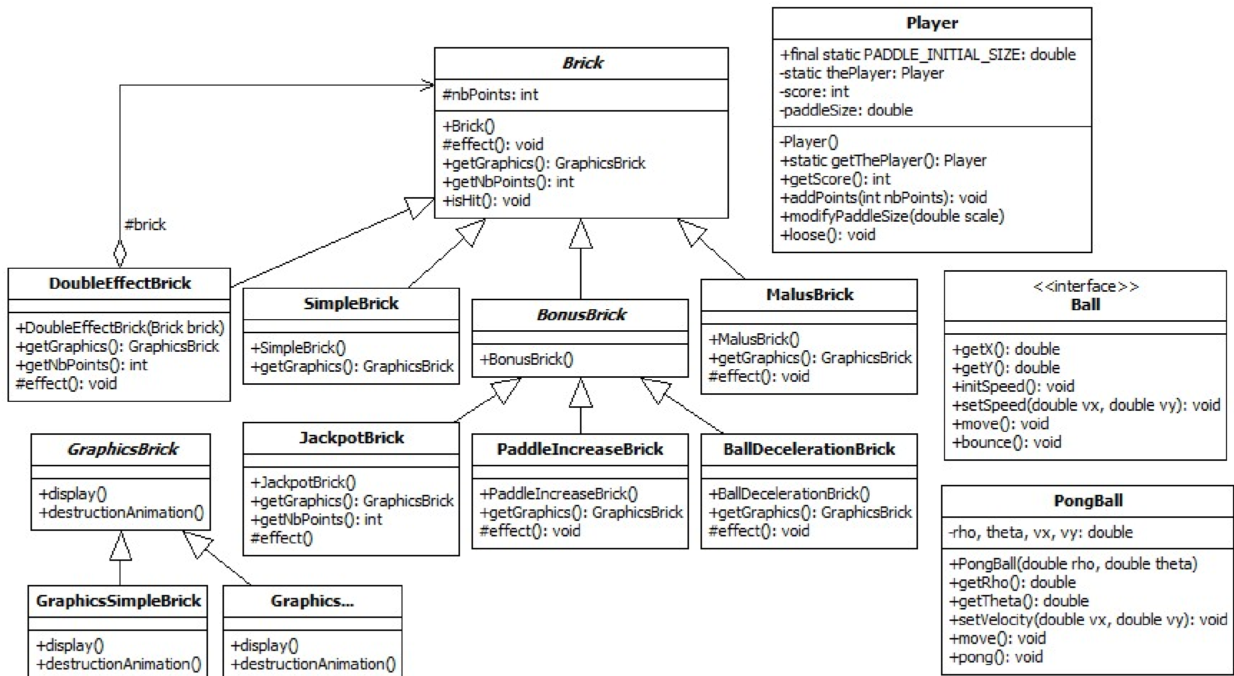

Figure 2 - Extrait de l'architecture logicielle du jeu *Crackout*

Question 2)

La méthode `addPoints` de la classe `Player` ajoute simplement un nombre de points (correspondant à la destruction d'une brique) passé en paramètre au score du joueur. Nous voulons pouvoir modifier dynamiquement ce comportement, afin de permettre différentes manières d'ajouter ce nombre de points. Par exemple, lorsque le joueur détruit une brique de type `JackpotBrick`, le nombre de points de la prochaine brique détruite sera multiplié par deux avant d'être ajouté au score, puis par trois pour la suivante, et ainsi de suite jusqu'à ce que le joueur perde (retour à un simple ajout des points) ou qu'il détruise une brique bonus/malus modifiant cet effet. Un autre exemple serait au contraire d'annuler l'ajout des points (*i.e.,* les réduire à zéro) pour les 10 prochaines briques détruites, lorsque le joueur détruit une brique de type `Malus`.

a. Modélisez à l'aide d'un diagramme de classes ce problème en utilisant un *design pattern* (à nommer).
b. Écrivez une implémentation des classes introduites par le *design pattern* utilisé, et modifiez, si nécessaire, le code[1] fourni en annexe, pour pouvoir traiter les différents exemples du problème.

Question 3)

L'interface `Ball` n'a pas encore d'implémentation. Néanmoins, elle est déjà utilisée par d'autres classes du jeu. Fournir une implémentation de cette interface est difficile, impliquant de nombreux calculs mathématiques complexes. Par conséquent, nous proposons de réutiliser la classe `PongBall`, issue du jeu *Pong*, qui réalise le comportement d'une balle rebondissante, tout en sachant que nous ne disposons pas de son code source. Cette classe est illustrée à la figure 2. Dans cette classe, la position de la balle est représentée en coordonnées polaires et les méthodes `setVelocity`, `move`, `pong`, permettent respectivement de modifier la vitesse de la balle, de la faire déplacer et de la faire rebondir.

a. Modélisez à l'aide d'un diagramme de classes ce problème en utilisant un *design pattern* (à nommer).
b. Écrivez une implémentation des classes introduites par le *design pattern* utilisé, pour fournir une implémentation de l'interface `Ball`.

[1] Vous pouvez utiliser ... pour indiquer la présence de code non modifié.

Annexe : extrait de l'implémentation du jeu *Crackout*

```
public abstract class Brick {
  protected int nbPoints;
  public Brick() {
    this.nbPoints = 100;
  }
  protected void effect() { }
  public abstract GraphicsBrick getGraphics();
  public int getNbPoints() {
    return this.nbPoints;
  }
  public void isHit() {
    this.getGraphicsBrick().destructionAnimation();
    this.effect();
    Player.getThePlayer().addPoints(this.getNbPoints());
  }
}

public class SimpleBrick extends Brick {
  public SimpleBrick() {
    super();
  }
  public GraphicsBrick getGraphics() {
    return new GraphicsSimpleBrick();
  }
}

public class PaddleIncreaseBrick extends BonusBrick {
  public PaddleIncreaseBrick() {
    super();
    this.nbPoints = 200;
  }
  public GraphicsBrick getGraphics() {
    return new GraphicsPaddleIncreaseBrick();
  }
  protected void effect() {
    super.effect();
    Player.getThePlayer().modifyPaddleSize(2);
  }
}

public class DoubleEffectBrick extends Brick {
  protected Brick brick;
  public DoubleEffectBrick(Brick brick) {
    super();
    this.brick = brick;
  }
  public GraphicsBrick getGraphics() {
    return new GraphicsDoubleEffectBrick(this.brick.getGraphics());
  }
  public int getNbPoints() {
    return 2 * this.brick.getNbPoints();
  }
  protected void effect() {
```

```
    this.brick.effect();
    this.brick.effect();
  }
}

public class Player {
  public final static double PADDLE_INITIAL_SIZE = 15;
  private static Player thePlayer = new Player();
  private int score;
  private double paddleSize;
  private Player() {
    this.score = 0;
    this.paddleSize = PADDLE_INITIAL_SIZE;
  }
  public static getThePlayer() {
    return thePlayer;
  }
  public int getScore() {
    return this.score;
  }
  public void addPoints(int nbPoints) {
    this.score += nbPoints;
  }
  public void modifyPaddleSize(double scale) {
    paddleSize *= scale;
  }
  public void loose {
    this.paddleSize = PADDLE_INITIAL_SIZE;
    ...
  }
}

public interface Ball {
  public double getX();
  public double getY();
  public void initSpeed();
  public void setSpeed(double vx, double vy);
  public void move();
  public void bounce();
}
```

FIN DE L'EXAMEN