IUT de Paris Descartes – Base de la Programmation Objet

Travaux Dirigés – Sujet n°10

Objectifs

— Restructuration de code existant en vue de généralisation.
— Héritage et sous-typage.

Un (des) jeu(x) de société

Le point de départ de notre travail est le code source fourni en annexe. La classe `JeuDeMarienbad` permet de joueur à une version du jeu de Nim. Le principe du jeu est simple. Deux joueurs s'affrontent en jouant à tour de rôle. Jouer consiste à choisir une ligne et à y retirer au moins une allumette. Initialement, il y a quatre lignes contenant respectivement 1, 3, 5 et 7 allumettes. Le joueur retirant la dernière allumette a perdu.

L'objectif de la séance de travaux dirigés est de retravailler ce programme de façon à pouvoir réemployer la structure commune à de nombreux jeux de société

1. Lire le programme donné en annexe et déterminer ce qui est propre à cette version de jeu de Nim et ce qui est commun à de nombreux jeu de société.

   Solution: Le déroulement d'une partie nécessite cinq actions distinctes qui soient spécifiques au jeu :
   — Initialiser le plateau en début de partie.
   — Faire jouer un joueur.
   — Déterminer si la partie est finie.
   — Déterminer le gagnant.
   — Déterminer le nombre de joueurs.

2. Proposer une solution permettant de spécialiser *a posteriori* ce qui est particulier à un jeu (tel que le jeu de Marienbad par exemple).

   Solution: Une solution basée sur l'héritage consiste à prévoir une méthode abstraite par action distincte. Ces méthodes abstraites seront spécialisées dans les sous-classes concrétisant un jeu particulier.

3. Restructurer le programme en conséquence. Vous écrirez le code complet de la classe abstraite puis donnerez les grandes lignes de la classe concrétisant le jeu de Marienbad.

   Solution: Les méthodes `partieFinie` et `annoncerGagnant` reçoivent en paramètre le numéro du joueur courant. Dans le le jeu de Marienbad, seule `annoncerGagnant` emploiera ce numéro. Il est passé à `partieFinie` uniquement pour que ce soit homogène. Vous noterez que `jouerUnePartie` est déclarée `final`.

```
public abstract class JeuDeSociété {

  final public void jouerUnePartie() {
    int j = 0;
```

```
    débutPartie();
    while (true) {
      faireJouer(j);
      if (partieFinie(j))
        break;
      else
        j = (j + 1) % getNbJoueurs();
    }
    annoncerGagnant(j);
  }

  abstract public int getNbJoueurs();
  abstract public void débutPartie();
  abstract public boolean partieFinie(int j);
  abstract public void annoncerGagnant(int j);
  abstract public void faireJouer(int j);
}
```

La plus grande part du code qui suit vient de la classe originale. Vous noterez :

- L'introduction (la spécialisation) des méthodes `getNbJoueurs`, `débutPartie` et `annoncerGagnant`.
- La réécriture du constructeur.

```
public class JeuDeMarienbad extends JeuDeSociété {
  private static final int NB_JOUEURS = 2;
  private static final int[] ALLUMETTES = { 1, 3, 5, 7 };
  private int[] allumettes;

  public JeuDeMarienbad() {
    débutPartie();
  }

  @Override
  public int getNbJoueurs() {
    return NB_JOUEURS;
  }

  @Override
  public void débutPartie() {
    allumettes = ALLUMETTES.clone();
  }

  @Override
  public boolean partieFinie(int j) {
    for (int n : allumettes)
      if (n != 0)
        return false;
    return true;
  }

  @Override
  public void faireJouer(int j) {
    assert (!partieFinie(j));
    Scanner sc = new Scanner(System.in);
    int ligne, nombre;
    do {
      System.out.println(this);
```

```
      System.out.println("Joueur␣n°" + j);
      System.out
          .print("n°␣de␣la␣ligne␣où␣prendre␣des␣allumettes␣:␣");
      ligne = sc.nextInt();
      System.out
          .print("nbre␣d'allumettes␣à␣retirer␣de␣cette␣ligne␣:␣");
      nombre = sc.nextInt();
    } while (ligne < 0 || ligne >= allumettes.length || nombre <= 0
        || nombre > allumettes[ligne]);
    allumettes[ligne] -= nombre;
  }

  @Override
  public void annoncerGagnant(int j) {
    System.out.println("le␣joueur␣n°" + ((j + 1) % getNbJoueurs())
        + "␣a␣gagné␣la␣partie");
  }

  @Override
  public String toString() {
    String s = "";
    for (int i = 0; i < allumettes.length; ++i)
      s += "ligne␣n°" + i + "␣:␣" + allumettes[i] + "␣allumette(s)\n";
    return s;
  }

  public static void main(String[] args) {
    JeuDeSociété j = new JeuDeMarienbad();
    j.jouerUnePartie();
  }
}
```

4. Une autre solution consiste à paramétrer la méthode `jouerUnePartie` par un objet représentant le jeu particulier. Déterminer la nouvelle structure du programme et le modifier en conséquence.

Solution: On introduit une interface (nommée `Jeu`) qui portera toute les méthodes abstraites précédentes. La méthode `jouerUnePartie` est paramétrée par un objet de ce type. Vous noterez qu'à présent cette méthode peut devenir une méthode de classe (statique).

```
public interface Jeu {
  int getNbJoueurs();
  void débutPartie();
  boolean partieFinie(int j);
  void annoncerGagnant(int j);
  void faireJouer(int j);
}
```

```
public class JeuDeSociété {
  public static void jouerUnePartie(Jeu jeu) {
    int j = 0;
    jeu.débutPartie();
    while (true) {
      jeu.faireJouer(j);
      if (jeu.partieFinie(j))
        break;
      else
```

```
        j = (j + 1) % jeu.getNbJoueurs();
    }
    jeu.annoncerGagnant(j);
  }
}
```

```
public class JeuDeMarienbad implements Jeu {

  // Rien ne change sauf la façon de s'en servir :

  public static void main(String[] args) {
    JeuDeSociété.jouerUnePartie(new JeuDeMarienbad());
  }
}
```

Les deux solutions sont équivalentes. La seconde à l'avantage de séparer l'algorithme (la méthode statique) du contrat (l'interface). Ces deux formes de solution (des patrons de conception) sont employées dans la bibliothèque standard.

Annexe

```
public class JeuDeMarienbad {
  private static final int NB_JOUEURS = 2;
  private static final int[] ALLUMETTES = { 1, 3, 5, 7 };
  private int[] allumettes;

  public JeuDeMarienbad() {
    allumettes = ALLUMETTES.clone();
  }

  public void jouerUnePartie() {
    int j = 0;
    allumettes = ALLUMETTES.clone();

    while (true) {
      faireJouer(j);
      if (partieFinie())
        break;
      else
        j = (j + 1) % NB_JOUEURS;
    }
    System.out.println("le␣joueur␣n°" + ((j + 1) % NB_JOUEURS)
        + "␣a␣gagné␣la␣partie");
  }

  public void faireJouer(int j) {
    assert (!partieFinie());
    Scanner sc = new Scanner(System.in);
    int ligne, nombre;
    do {
      System.out.println(this);
      System.out.println("Joueur␣n°" + j);
      System.out
          .print("n°␣de␣la␣ligne␣où␣prendre␣des␣allumettes␣:␣");
```

```
      ligne = sc.nextInt();
      System.out
          .print("nbre␣d'allumettes␣à␣retirer␣de␣cette␣ligne␣:␣");
      nombre = sc.nextInt();
    } while (ligne < 0 || ligne >= allumettes.length || nombre <= 0
        || nombre > allumettes[ligne]);
    allumettes[ligne] -= nombre;
  }

  public boolean partieFinie() {
    for (int n : allumettes)
      if (n != 0)
        return false;
    return true;
  }

  @Override
  public String toString() {
    String s = "";
    for (int i = 0; i < allumettes.length; ++i)
      s += "ligne␣n°" + i + "␣:␣" + allumettes[i] + "␣allumette(s)\n";
    return s;
  }

  public static void main(String[] args) {
    JeuDeMarienbad j = new JeuDeMarienbad();
    j.jouerUnePartie();
  }
}
```