Ange Diable Java UML

Introduction

Le jeu de l'Ange et du Diable oppose un ange et un diable sur un damier de taille N constitué de NxN cases. Au départ du jeu, l'ange est situé sur la case centrale du damier et les autres cases sont vides. Les deux joueurs jouent à tour de rôle. A son tour, l'ange se déplace vers une case vide située à une distance inférieure à sa puissance. A son tour, le diable bouche une case vide du damier. Le but de l'ange est d'atteindre une case du bord du damier et celui du diable de bloquer l'ange sur une case du damier qui ne soit pas une case du bord.

Objectif et méthode

En annexe figure un programme Java permettant de jouer à ce jeu. Mais il n'a pas été écrit en utilisant la toute la puissance de l'orientation objet de Java et de la programmation orientée objet en général. On souhaite donc améliorer ce programme. Pour ce faire, nous allons analyser le programme et construire son modèle UML. Ensuite, nous améliorerons le modèle. Ainsi les méthodes compliquées du programme Java existant en annexe seront ré-écrites plus simplement.

Question 1 (0,5 pt) un bug…

Le programme principal est situé dans la classe `Partie`. La classe `Keyboard` est la classe habituelle utilisée en TP. Pourquoi java ne peut compiler ce `main()` ? Modifier l'unique instruction du `main()` qui est erronée.

L'instruction `Partie p` ; déclare une référence non initialisée. L'instruction p.initialiser() ; est rejetée par javac.

Question 2 (1 pt) l'affichage du damier

Les classes `Damier` et `Case` sont données en annexe. Si l'utilisateur choisit une taille de damier égale à 3, qu'affiche le programme au début d'une partie ?

```
     1     2     3
1    +     +     +     1
2    +    >A<    +     2
3    +     +     +     3
     1     2     3
```

Question 3 (3 pts) le modèle UML

a) Dessiner le diagramme de généralisation UML de ce programme.

Diagrammes UML à faire…

b) Dessiner le diagramme de classes UML de ce programme. La classe `Ange` est-elle associée à la classe `Case` ? La classe `Diable` est-elle associée à la classe `Case` ?

Diagrammes UML à faire…
OUI la classe Ange est associée à la classe Case
NON la classe Diable ne l'est pas.

c) Dessiner la description UML de chaque classe (attributs et méthodes). Quelles sont les méthodes redéfinies ?

Diagrammes UML à faire…
Les méthodes redéfinies sont les méthodes jouer() dans les classes AH, DH, AA, DA.

Question 4 (1 + 1 = 2 pts) spécifications externes

a) Ce programme permet-il de faire plusieurs parties à la suite ? Quelle est la méthode qui correspond au déroulement d'une partie complète ?

oui, on peut faire autant de parties que l'on souhaite.
La méthode faire() correspond au déroulement d'une partie complète.

b) Peut-on faire jouer deux joueurs humains l'un contre l'autre ?
c) Peut-on faire jouer deux joueurs aléatoires l'un contre l'autre ?
d) Peut-on faire jouer un joueur humain contre un joueur aléatoire ?
e) Peut-on faire jouer deux anges l'un contre l'autre ?
f) Peut-on faire jouer deux diables l'un contre l'autre ?
g) Un joueur humain peut-il jouer le diable ? l'ange ?
h) Un joueur aléatoire peut-il jouer le diable ? l'ange ?

Il permet de jouer au jeu de l'ange et du diable. L'utilisateur peut jouer au choix le rôle de l'ange, du diable ou des deux contre un joueur aléatoire qui joue les rôles vacants. La réponse a chaque question a)b)c)d)g)h) est OUI.
On ne peut faire jouer deux diables l'un contre l'autre, ni deux anges l'un contre l'autre. La réponse aux questions e) et f) est NON.

i) dessiner un diagramme de séquence UML correspondant à l'interaction de l'utilisateur avec ce programme dans le cas d).

a faire

Question 5 (1 pt) un diagramme d'instances

On suppose que l'état du programme correspond à l'affichage du damier suivant :

```
      1     2     3
1     +     +     +     1
2     +     >A<   @     2
3     +     @     @     3
      1     2     3
```

Dessiner le diagramme d'instances UML correspondant.

Question 6 (0,5 pt) généralités sur la méthode void jouer()

Les méthodes `jouer()` des classes `AngeHumain`, `AngeAléatoire`, `DiableHumain`, `DiableAléatoire` ont des ressemblances et des différences.

a) Peut-on dire que les 4 méthodes `jouer()` font la même chose : d'abord choisir le coup, puis effectuer le coup ?

Oui

b) On veut que la définition de la méthode `jouer()` soit identique pour les 4 classes concrètes de joueur. Dans quelle classe la placer ?

Dans Joueur.

Question 7 (1 pt) généralités sur la méthode Case choisirUneCase()

On veut que la définition de la méthode `jouer()` de la classe Joueur soit la suivante :

```
public void jouer() {
  Case c = choisirUneCase();
  effectuerCoupSurCase(c);
}
```

a) Peut-on dire que les 4 méthodes `Case choisirUneCase()` font la même chose ?

non

b) Peut-dire que les 2 méthodes `Case choisirUneCase()` des classes `AngeHumain` et `DiableHumain` font approximativement la même chose ?

oui

c) Que les 2 méthodes `Case choisirUneCase()` des classes `AngeAleatoire` et `DiableAleatoire` font approximativement la même chose ?

oui

d) Que les méthodes `Case choisirUneCase()` des joueurs « humains » sont très différentes de celles de joueurs « aléatoires » ?

oui

e) Donner une définition de la méthode `Case choisirUneCase()` de la classe `Joueur`.

```
class Joueur {
    …
  public Case choisirUneCase() {
    return null;
  }
}
```

Question 8 (2 pts) généralités sur la méthode void effectuerCoupSurCase(Case)

Dans cette question on traite la méthode `void effectuerCoupSurCase(Case)`.

a) Peut-on dire que les 4 méthodes `void effectuerCoupSurCase(Case)` font la même chose ?

non

b) Peut-dire que les 2 méthodes `void effectuerCoupSurCase(Case)` des classes `AngeHumain` et `AngeAleatoire` font approximativement la même chose ?

oui

c) Que les 2 méthodes `void effectuerCoupSurCase(Case)` des classes `DiableHumain` et `DiableAleatoire` font approximativement la même chose ?

oui

d) Que les méthodes `void effectuerCoupSurCase(Case)` des anges sont différentes de celles des diables ?

oui

e) En déduire la définition dans la classe `Joueur` et les redéfinitions de la méthode `void effectuerCoupSurCase(Case)` dans les classes adéquates.

Il faut placer cette méthode dans la classe Joueur et la redéfinir dans la classe Ange et la classe Diable.

```
class Joueur {
      …
    public void effectuerCoupSurCase(Case c) {
      ;
    }
}
class Ange extends Joueur {
      …
    public void effectuerCoupSurCase(Case c) {
      maCase.monAnge = null;
      maCase = maPartie.monDamier.mesCases[c.x][c.y];
      maPartie.monDamier.mesCases[c.x][c.y].monAnge = this;
    }
}
class Diable extends Joueur {
      …
    public void effectuerCoupSurCase(Case c) {
      maPartie.monDamier.mesCases[c.x][c.y].bouchee = true;
    }
}
```

Question 9 (2,5 pts) les méthodes boolean caseInaccessible(Case) et boolean verifier(int, int)

Pour simplifier les méthodes `Case choisirUneCase()` des joueurs « humains », on veut définir et éventuellement redéfinir dans les classes adéquates la méthode `boolean verifier(int, int)` correspondant à la vérification d'un coup désigné par les valeurs de `x` et `y` tapées par un joueur « humain ».

a) En supposant que l'on définisse une méthode `boolean verifier(int, int)` dans la classe `AngeHumain` et une autre dans la classe `DiableHumain`, quelle serait la différence entre ces deux méthodes ?

La différence est mineure : dans AH, on teste en plus que la case est inaccessible.

b) Définir une méthode `boolean caseInaccessible(Case)` dans la classe `Ange`, retournant la valeur de la condition supplémentaire existant dans la méthode `boolean verifier(int, int)` de la classe `AngeHumain`.

```
class Ange extends Joueur {
      …
    public boolean caseInaccessible(Case c) {
      return (c.distance(maCase) > puissance);
    }
}
```

c) Transformer la condition supplémentaire existant dans la méthode `boolean verifier(int, int)` de la classe `AngeHumain` en utilisant un appel à la méthode `boolean caseInaccessible(Case)` de la classe `Ange`.

```
          if (caseInaccessible(c)) {
```

d) On souhaite que les méthodes `boolean verifier(int, int)` des classes `AngeHumain` et `DiableHumain` soient strictement identiques. On ajoute donc la condition précédente dans la méthode `boolean verifier(int, int)` de la classe `DiableHumain`. Définir une autre méthode `boolean caseInaccessible(Case)` retournant toujours `false` et la placer dans la classe adéquate.

```
class Joueur {
```

```
      …
    public boolean caseInaccessible(Case c) {
      return false;
    }
}
```

e) Donner la définition, désormais identique, de `boolean verifier(int, int)` des classes `AngeHumain` et `DiableHumain`.

```
    public boolean verifier(int x, int y) {
      boolean ok;
      if ((x>0) && (y>0) &&
          (x<=maPartie.monDamier.taille) &&
          (y<=maPartie.monDamier.taille) ) {
          ok = true;
          Case c = maPartie.monDamier.mesCases[x-1][y-1];
          if (c.bouchee) {
            System.out.println("Erreur: case bouchee.");
            ok = false;
          }
          if (c.monAnge!=null) {
            System.out.println("Erreur: case occupee par l'ange.");
            ok = false;
          }
          if (caseInaccessible(c)) {
            System.out.println("Erreur: case inaccessible.");
            ok = false;
          }
      }
      else ok = false;
      return ok;
    }
```

f) Les 2 méthodes `boolean verifier(int, int)` des classes `AngeHumain` et `DiableHumain` étant strictement identiques peut-on les remplacer par une méthode identique plus générale ?

Si on voulait avoir une seule définition, alors il faudrait mettre cette méthode dans Joueur. Mais cette méthode serait alors utilisable par les joueurs « aléatoires ». Donc la réponse est NON.

Question 10 (0,5 pt) la méthode void afficherPrompt()

Pour rendre identiques les 2 méthodes `Case choisirUneCase()` des joueurs « humains », on veut définir une méthode void `afficherPrompt()`. Donner les définitions nécessaires de cette méthode.

```
class AngeHumain extends Ange {
      …
    public void afficherPrompt() {
      System.out.print("Ange > ");
    }
}
class DiableHumain extends Diable {
      …
    public void afficherPrompt() {
      System.out.print("Diable > ");
    }
```

Question 11 (3 pts) la nouvelle définition de Case choisirUneCase()

a) En utilisant les appels de `void afficherPrompt()` et de `boolean verifier(int, int)` donner la redéfinition de `Case choisirUneCase()` des classes « humaines ».

```
public Case choisirUneCase () {
      int x, y, z;
      boolean ok;
```

```
      do {
          System.out.print("x ? ");
          afficherPrompt();
          x = Keyboard.getInt();
          System.out.print("y ? ");
          afficherPrompt();
          y = Keyboard.getInt();
          System.out.println("x = " + x + " y = " + y);
          ok = verifier(x, y);
      } while (!ok);
      return maPartie.monDamier.mesCases[x-1][y-1];
    }
```

b) En utilisant l'appel de `boolean caseInaccessible()`, donner la définition de `Case choisirUneCase()` des classes « aléatoires ».

```
    public Case choisirUneCase () {
      int x = 0;
      int y = 0;
      int t = maPartie.monDamier.taille;
      int i, j, n, r;
      Case c = null;
      n = 0;  // on compte le nombre de coups possibles.
      for (i=0; i<t; i++) {
          for (j=0; j<t; j++) {
            c = maPartie.monDamier.mesCases[i][j];
            if (!caseInaccessible(c))
            if (!(c.bouchee) && (c.monAnge==null)) n++;
          }
      }
      r = Alea.engendrer(n);
      n = 0; // on selectionne un coup aleatoire dans les coups possibles.
      for (i=0; i<t; i++) {
          for (j=0; j<t; j++) {
            c = maPartie.monDamier.mesCases[i][j];
            if (!caseInaccessible(c))
            if (!(c.bouchee) && (c.monAnge==null))
                if (++n == r) return c;
          }
      }
      return null;
    }
```

c) Donner la nouvelle description des méthodes des classes Java descendantes de la classe Joueur.

```
class Joueur {
    void jouer() {…}
    Case choisirUneCase() { return null; }
    boolean caseInaccessible(Case){ return false; }
    void effectuerCoupSurCase(Case) { ; }
}
class Ange extends Joueur {
    boolean caseInaccessible(Case) { … }
    void effectuerCoupSurCase(Case) {…}
}
class AngeHumain extends Ange {
  void afficherPrompt(){…}
  Case choisirUneCase() {…}
  boolean verifier(int, int) {…}
}
class AngeAleatoire extends Ange {
  Case choisirUneCase(){…}
  boolean caseInaccessible(Case) {…}
}
class Diable extends Joueur {
  void effectuerCoupSurCase(Case) {…}
}
```

```
class DiableHumain extends Diable {
  void afficherPrompt() {…}
  Case choisirUneCase () {…}
  boolean verifier(int, int) {…}
}
class DiableAleatoire extends Diable {
  Case choisirUneCase () {…}
}
```

Question 12 (0,5 + 1 + 0,5 = 2 pts) pas d'héritage multiple en Java…

Actuellement un joueur « humain » ne peut appeler la méthode `Case choisirUneCase()` d'un joueur « aléatoire » et réciproquement, ce qui est un avantage. Mais l'inconvénient est que la méthode `Case choisirUneCase()` des joueurs « humains » est écrite deux fois. Idem pour `Case choisirUneCase()` des joueurs « aléatoires » et pour `boolean verifier()` des joueurs « humains ».

a) Afin de supprimer cet inconvénient, un première tentative est de changer l'ordre des discriminations du diagramme de généralisation : d'abord la discrimination `Humain-Aléatoire` puis la discrimination `Ange-Diable`. Ce changement résoud-il le problème ? Pourquoi ?

Ce changement permettra aux méthodes Humain.choisirUneCase, Aleatoire.choisirUneCase() et Humain.verifier de n'être écrites qu'une seule fois. Mais les méthodes des classes Ange et Diable, void effectuerCoupSurCase() et boolean caseInaccessible(), devront être écrites deux fois. On gagnera d'un côté pour perdre de l'autre. Le changement d'ordre des discrimination ne résoud donc pas le problème.

b) De quel outil de modélisation UML a-t-on besoin pour résoudre ce problème ? Cet outil existe-t-il en Java ?

L'héritage multiple. Non.

c) Proposer malgré tout une solution minimisant au mieux les répétitions de traitements. On pourra définir et placer des méthodes `Case choisirUneCaseParLInterface()` et une méthode `Case choisirUneCaseAleatoirement()`. On pourra replacer correctement la méthode `vérifier(int, int)`.

```
class Joueur {
      …
    public void jouer() {
      Case c = choisirUneCase();
      effectuerCoupSurCase(c);
    }
    public Case choisirUneCase() {
      return null;
    }
    public boolean caseInaccessible(Case c) {
      return false;
    }
    public Case choisirUneCaseAleatoirement() {
      int x = 0;
      int y = 0;
      int t = maPartie.monDamier.taille;
      int i, j, n, r;
      Case c = null;
      n = 0;  // on compte le nombre de coups possibles.
      for (i=0; i<t; i++) {
          for (j=0; j<t; j++) {
            c = maPartie.monDamier.mesCases[i][j];
            if (!caseInaccessible(c))
            if (!(c.bouchee) && (c.monAnge==null)) n++;
          }
      }
      r = Alea.engendrer(n);
      n = 0; // on selectionne un coup aleatoire dans les coups possibles.
      for (i=0; i<t; i++) {
          for (j=0; j<t; j++) {
            c = maPartie.monDamier.mesCases[i][j];
            if (!caseInaccessible(c))
            if (!(c.bouchee) && (c.monAnge==null))
```

```
                if (++n == r) return c;
            }
        }
        return null;
    }
    public Case choisirUneCaseParLInterface() {
        int x, y, z;
        boolean ok;
        do {
            System.out.print("x ? ");
            afficherPrompt();
            x = Keyboard.getInt();
            System.out.print("y ? ");
            afficherPrompt();
            y = Keyboard.getInt();
            System.out.println("x = " + x + " y = " + y);
            ok = verifier(x, y);
        } while (!ok);
        return maPartie.monDamier.mesCases[x-1][y-1];
    }
    public boolean verifier(int x, int y) {
        boolean ok;
        if ((x>0) && (y>0) &&
            (x<=maPartie.monDamier.taille) &&
            (y<=maPartie.monDamier.taille) ) {
            ok = true;
            Case c = maPartie.monDamier.mesCases[x-1][y-1];
            if (c.bouchee) {
              System.out.println("Erreur: case bouchee.");
              ok = false;
            }
            if (c.monAnge!=null) {
              System.out.println("Erreur: case occupee par l'ange.");
              ok = false;
            }
            if (caseInaccessible(c)) {
              System.out.println("Erreur: case inaccessible.");
              ok = false;
            }
        }
        else ok = false;
        return ok;
    }
    public void effectuerCoupSurCase(Case c) {
        ;
    }
    public void afficherPrompt() {
        ;
    }
}

class Ange extends Joueur {
    // pas de changement
}

class AngeHumain extends Ange {
        …
    public void afficherPrompt() {
        System.out.print("Ange > ");
    }
    public Case choisirUneCase() {
        return choisirUneCaseParLInterface();
    }
```

```
}

class AngeAleatoire extends Ange {
      …
    public Case choisirUneCase() {
      return choisirUneCaseAleatoirement();
    }
}

class Diable extends Joueur {
    // pas de changement
}

class DiableHumain extends Diable {
      …
    public void afficherPrompt() {
      System.out.print("Diable > ");
    }
    public Case choisirUneCase() {
      return choisirUneCaseParLInterface();
    }
}

class DiableAleatoire extends Diable {
      …
    public Case choisirUneCase() {
      return choisirUneCaseAleatoirement();
    }
}
```

d) Quel sont les inconvénients de cette approche ?

On a deux méthodes « bidon » AngeHumain.choisirUneCase() et DiableHumain.choisirUneCase(). Elles ne servent qu'a appeler Joueur.choisirUneCaseParLInterface(). On a deux méthodes « bidon » AngeAléatoire.choisirUneCase() et DiableAléatoire.choisirUneCase(). Elles ne servent qu'a appeler Joueur.choisirUneCaseAléatoirement(). On a placé choisirUneCaseParLInterface(), vérifier() dans la classe Joueur. Un joueur aléatoire peut donc appeler ces méthodes. On a placé choisirUneCaseAléatoirement() dans la classe Joueur. Un joueur humain peut donc appeler cette méthode.

Question 13 (1 pt) si on avait l'héritage multiple…

a) En supposant que l'héritage multiple est supporté par le langage de programmation, quelles sont les deux classes à rajouter au modèle UML ?

On crée deux nouvelles sous-classes de la classe Joueur : Humain et Aleatoire.

b) Proposer une généralisation UML avec un placement adéquat des méthodes.

On utilise l'héritage multiple : par exemple AngeAléatoire hérite de Ange et de Aléatoire. Analogue pour les 3 autres sous-classes-concrètes. choisirUneCaseAleatoirement(), renommée choisirUneCase(), est placée dans Aléatoire. choisirUneCaseParLInterface() renommée choisirUneCase(), boolean verifier(int, int) et void afficherPrompt() sont placées dans Humain.

c) Quels sont les avantages de cette nouvelle approche ?

Ce nouveau modèle est plus concis, plus simple.

Annexe

```
class Partie {

    public Ange monAnge;
    public Diable monDiable;
    public Damier monDamier;
    public Joueur trait;
    public boolean gagnee;

    public Partie(int t) {
      monDamier = new Damier(t);
      gagnee = false;
    }

    public void faire() {
      System.out.println("Debut de la partie.");
      System.out.println(monDamier);
      do {
          trait.jouer();
          System.out.println(monDamier);
          gagnee = monAnge.jeSuisBloque() || monAnge.jeSuisLibre();
          trait = autreJoueur();
      } while (!gagnee);
      System.out.println("Fin de la partie.");
    }

    public Joueur autreJoueur() {
      if (trait == monAnge) return monDiable;
      else return monAnge;
    }

    public void initialiser() {
      monAnge = new AngeAleatoire(this, 1);
      monDamier.mesCases[monDamier.taille/2]
                        [monDamier.taille/2].monAnge = monAnge;
      monAnge.maCase = monDamier.mesCases[monDamier.taille/2]
                                         [monDamier.taille/2];
      monDiable = new DiableHumain(this);
      trait = monAnge;
    }

    public static void main (String [] args) {
      char r;
      System.out.println("Jeu de l'Ange et du Diable.");
      do {
          System.out.println("quitter          (q)");
          System.out.println("faire une partie (f)");
          r = Keyboard.getChar();
          if (r=='f') {
            System.out.println("\tTaille du damier ? ");
            int t = Keyboard.getInt();
            Partie p;
            p.initialiser();
            p.faire();
          }
      } while (r!='q');
      System.out.println("Au revoir.");
    }
}
```

```
class Damier {
    public Case [][] mesCases;
    public int taille;

    public Damier(int t) {
      taille = t;
      mesCases = new Case [taille][taille];
      int i, j;
      for (i=0; i<taille; i++) {
          for (j=0; j<taille; j++) {
            mesCases[i][j] = new Case(i, j, this);
          }
      }
    }
    public String toString() {
      String s = "";
      int i, j;
      s = s + "   ";
      for (i=1; i<=taille; i++) s = s + " " + i + " ";
      s = s + "\n";
      for (i=0; i<taille; i++) {
          s = s + " " + (i+1) + " ";
          for (j=0; j<taille; j++) {
            s = s + mesCases[i][j].toString();
          }
          s = s + " " + (i+1) + " ";
          s = s + "\n";
      }
      s = s + "   ";
      for (i=1; i<=taille; i++) s = s + " " + i + " ";
      s = s + "\n";
      return s;
    }
}

class Case {
    public Damier monDamier;
    public int x;
    public int y;
    public Ange monAnge;
    public boolean bouchee;

    public Case(int a, int b, Damier d) {
      monDamier = d;
      x = a;
      y = b;
      monAnge = null;
      bouchee = false;
    }
    public String toString() {
      String s;
      if (monAnge != null) s = ">A<";
      else if (bouchee) s = " @ ";
      else s = " + ";
      return s;
    }
    public int distance(Case c) {
      return max(abs(x - c.x), abs(y - c.y));
    }
}
```

```
class Joueur {

    public Partie maPartie;

    public Joueur(Partie p) {
      maPartie = p ;
    }

    public void jouer() {
      ;
    }
}


class Ange extends Joueur {

    public int puissance;

    public Case maCase;

    public Ange(Partie pa, int pui) {
      super(pa) ;
      puissance = pui ;
    }

    // retourne false si deplacement sur case vide possible, true sinon
    public boolean jeSuisBloque() {
      boolean uneCaseLibre = false;
      int t = maPartie.monDamier.taille;
      int i, j;
      for (i=0; i<t; i++) {
          for (j=0; j<t; j++) {
            Case c = maPartie.monDamier.mesCases[i][j];
            if ( maCase.distance(c) <= puissance )
                if (!(c.bouchee) && (c.monAnge==null)) uneCaseLibre = true;
          }
      }
      return !uneCaseLibre;
    }

    // retourne true ange sur case du bord du damier, false sinon
    public boolean jeSuisLibre() {
      if ((maCase.x==0) || (maCase.y==0)) return true;
      int t = maPartie.monDamier.taille;
      if ((maCase.x==t-1) || (maCase.y==t-1)) return true;
      return false;
    }

}


class Diable extends Joueur {

    public Diable(Partie p) {
      super(p) ;
    }

}
```

```
class AngeHumain extends Ange {

    public AngeHumain(Partie pa, int pui) {
       super(pa, pui);
    }

    // La méthode JOUER demande a l'utilisateur de taper une valeur
    // de x et de y designant une case.
    // Elle verifie si x et y correspondent bien a une case du damier.
    // Tant que les valeurs de x et y ne sont pas correctes,
    // l'utilisateur doit retaper ces valeurs.
    // Quand le coup est choisi (tapé et vérifié), le programme effectue le
    // coup (l'ange quitte l'ancienne case et arrive sur la nouvelle case).

    public void jouer() {

       int x, y;
       int t = maPartie.monDamier.taille ;
       boolean ok;

       // l'utilisateur choisit un coup

       do {

           // l'utilisateur tape un coup

           System.out.print("Ange, x > ");  // affichage d'un prompt
           x = Keyboard.getInt();
           System.out.print("Ange, y > ");  // affichage d'un prompt
           y = Keyboard.getInt();
           System.out.println("x = " + x + " y = " + y);

           // le programme verifie le coup

           if ((x>0) && (y>0) && (x<=t) && (y<=t) ) {
              ok = true;
              Case c = maPartie.monDamier.mesCases[x-1][y-1];
              if (c.bouchee) {
                  System.out.println("Erreur: case bouchee.");
                  ok = false;
              }
              if (c.monAnge!=null) {
                  System.out.println("Erreur: case occupee par l'ange.");
                  ok = false;
              }

              // la case est-elle inaccessible a l'ange ?

              if (c.distance(maCase) > puissance) {
                  System.out.println("Erreur: case inaccessible.");
                  ok = false;
              }
           }
           else ok = false;

       } while (!ok);


       // le programme effectue le coup

       maCase.monAnge = null;
       maCase = maPartie.monDamier.mesCases[x-1][y-1];
       maPartie.monDamier.mesCases[x-1][y-1].monAnge = this;
    }
}
```

```
class DiableHumain extends Diable {

    public DiableHumain(Partie p) {
       super(p);
    }

    // La méthode JOUER demande a l'utilisateur de taper une valeur
    // de x et de y designant une case.
    // Elle verifie si x et y correspondent bien a une case du damier.
    // Tant que les valeurs de x et y ne sont pas correctes,
    // l'utilisateur doit retaper ces valeurs.
    // Quand le coup est choisi (tapé et vérifié), le programme effectue le
    // coup (il bouche une case).

    public void jouer() {

       int x, y;
       int t = maPartie.monDamier.taille;
       boolean ok;

       // l'utilisateur choisit un coup

       do {

           // l'utilisateur tape un coup

           System.out.print("Diable, x > ");  // affichage d'un prompt
           x = Keyboard.getInt();
           System.out.print("Diable, y > ");  // affichage d'un prompt
           y = Keyboard.getInt();
           System.out.println("x = " + x + " y = " + y);

           // le programme verifie le coup

           if ((x>0) && (y>0) && (x<=t) && (y<=t) ) {
              ok = true;
              Case c = maPartie.monDamier.mesCases[x-1][y-1];
              if (c.bouchee) {
                  System.out.println("Erreur: case bouchee.");
                  ok = false;
              }
              if (c.monAnge!=null) {
                  System.out.println("Erreur: case occupee par l'ange.");
                  ok = false;
              }

              // aucune case n'est inaccessible au diable.

           }
           else ok = false;
       } while (!ok);

       // le programme effectue le coup

       maPartie.monDamier.mesCases[x-1][y-1].bouchee = true;
    }
}
```

```
class AngeAleatoire extends Ange {

    public AngeAleatoire(Partie pa, int pui) {
       super(pa, pui);
    }

    // La méthode JOUER n'interagit pas avec l'utilisateur.
    // Pour choisir le coup, elle compte le nombre 'n' de coups possibles,
    // puis elle tire aleatoirement un nombre 'r' compris entre 1 et 'n'
    // et enfin elle retrouve la case correspondant au rieme coup.
    // Quand le coup est choisi, le programme effectue le coup
    // (l'ange quitte l'ancienne case et arrive sur la nouvelle case).

    public void jouer() {

       int t = maPartie.monDamier.taille;
       int x, y, i, j, n, r;
       x = y = n = 0;

       // le programme choisit un coup

       for (i=0; i<t; i++) {
           for (j=0; j<t; j++) {
              Case c = maPartie.monDamier.mesCases[i][j];
              if (c.distance(maCase) <= puissance)
              if (!(c.bouchee) && (c.monAnge==null)) n++;
           }
       }
       r = Alea.engendrer(n);
       n = 0;
       for (i=0; i<t; i++) {
           for (j=0; j<t; j++) {
              Case c = maPartie.monDamier.mesCases[i][j];
              if (c.distance(maCase) <= puissance)
              if (!(c.bouchee) && (c.monAnge==null))
              if (++n == r) {
                  x = c.x + 1;
                  y = c.y + 1;
              }
           }
       }

       // le programme effectue le coup

       maCase.monAnge = null;
       maCase = maPartie.monDamier.mesCases[x-1][y-1];
       maPartie.monDamier.mesCases[x-1][y-1].monAnge = this;
    }
}
```

```
class DiableAleatoire extends Diable {

    public DiableAleatoire(Partie p) {
       super(p);
    }

    // La méthode JOUER n'interagit pas avec l'utilisateur.
    // Pour choisir le coup, elle compte le nombre 'n' de coups possibles,
    // puis elle tire aleatoirement un nombre 'r' compris entre 1 et 'n'
    // et enfin elle retrouve la case correspondant au rieme coup.
    // Quand le coup est choisi, le programme effectue le coup
    // (il bouche une case).

    public void jouer() {

       int t = maPartie.monDamier.taille;
       int x, y, i, j, n, r;
       x = y = n = 0;

       // le programme choisit un coup

       for (i=0; i<t; i++) {
           for (j=0; j<t; j++) {
              Case c = maPartie.monDamier.mesCases[i][j];
              if (!(c.bouchee) && (c.monAnge==null)) n++;
           }
       }
       r = Alea.engendrer(n);
       n = 0;
       for (i=0; i<t; i++) {
           for (j=0; j<t; j++) {
              Case c = maPartie.monDamier.mesCases[i][j];
              if (!(c.bouchee) && (c.monAnge==null))
              if (++n == r) {
                  x = c.x + 1;
                  y = c.y + 1;
              }
           }
       }

       // le programme effectue le coup

       maPartie.monDamier.mesCases[x-1][y-1].bouchee = true;
    }
}

import java.util.*;


class Alea {

    // retourne un nombre entier compris entre 1 et n

    public static int engendrer(int n) {
       Random r = new Random();
       int a = Math.abs(r.nextInt()) % n + 1;
       System.out.println("n = " + n + " a = " + a);
       return a;
    }
}
```