# Ange Diable C++ 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 C++ permettant de jouer à ce jeu. Mais il n'a pas été écrit en utilisant la toute la puissance de l'orientation objet de C++. 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 C++ existant en annexe seront ré-écrites plus simplement.

## Question 1 (le programme principal)(0,5 pt) un bug…

Le programme principal est situé au début de l'annexe. Que manque-t-il dans ce `main` ? Corriger le `main` en conséquence.

L'instruction Partie * p ; déclare un **pointeur non initialisé**. L'instruction p->initialiser() ; est rejetée.
Il faut faire Partie * p = new Partie(t) ; qui crée un objet de type Partie.

## Question 2 (les classes `Damier` et `Case`)(1 pt) l'affichage du damier

Les classes `Damier` et `Case` sont données en annexe. Si l'utilisateur a choisi une taille de damier égale à 3, dans la méthode `faire()` de la classe `Partie`, donner la sortie de `monDamier->afficher();` au début d'une partie.

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

## Question 3 (modélisation UML)(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 explicitement.** Cependant l'association existe implicitement avec l'attribut 'bouchee' dans la classe Case qui correspond à la présence/absence du diable sur une case.

c) Quelles sont les méthodes redéfinies ?

Les méthodes redéfinies sont les méthodes **jouer() dans les classes AH, DH, AA, DA.**

## Question 4 (spécifications d'interface)(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

j) En une phrase, que ferait le programme si on enlevait le mot-clé `virtual` devant la méthode `jouer()` de la classe `Joueur` ?

**Il bouclerait à l'infini en affichant des damiers initiaux** car la méthode jouer de la classe Joueur serait appelée à la place des méthodes spécialisées.

## Question 5 (diagramme d'instances UML)(1 pt) un diagramme d'instances

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

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

Dessiner le diagramme d'instances UML correspondant.

Diagrammes UML à faire…

## Question 6 (1er aperçu des méthodes `void jouer()`)(0,5 pt) généralités sur la méthode void jouer()

Le corps des méthodes `void jouer()` des classes `AngeHumain`, `AngeAléatoire`, `DiableHumain`, `DiableAléatoire` ont beaucoup de ressemblances et quelques différences. Le but des questions suivantes est de réécrire des méthodes dans les classes descendantes de la classe `Joueur` pour rassembler les traitements ressemblants de ces méthodes et séparer les traitements différents.

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

**Oui**

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

**Dans Joueur**.

## Question 7 (1er aperçu de la méthode `Case * choisirUneCase()`) (2 pts) généralités sur la méthode Case * choisirUneCase()

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

```
void Joueur::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`.

```
Case * Joueur::choisirUneCase() { return NULL; }
```

## Question 8 (la méthode `void effectuerCoupSurCase(Case*)`) (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.

```
void Joueur::effectuerCoupSurCase(Case * c) {;}
void Ange::effectuerCoupSurCase(Case * c) {
  maCase->monAnge = NULL;
  maCase = c;
  c->monAnge = this;
}
void Diable::effectuerCoupSurCase(Case c) {
  c->bouchee = true;
}
```

## Question 9 (la méthode `bool verifier(int, int)`)(2,5 pts) les méthodes void caseInaccessible(Case) et bool verifier(int, int)

Pour simplifier les méthodes `Case * choisirUneCase()` des joueurs « humains », on veut définir, et éventuellement redéfinir dans les classes adéquates, la méthode `bool 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 `bool 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 `bool caseInaccessible(Case*)` dans la classe `Ange`, retournant la valeur de la condition supplémentaire existant dans la méthode `bool verifier(int, int)` de la classe `AngeHumain`.

```
bool Ange::caseInaccessible(Case * c) {
  return (c->distance(maCase) > puissance);
}
```

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

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

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

```
bool Joueur::caseInaccessible(Case * c) { return false; }
```

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

```
bool AngeHumain::verifier(int x, int y) {
  bool 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) {
      cout << "Erreur: case bouchee." << endl;
      ok = false;
    }
    if (c->monAnge!=NULL) {
      cout << "Erreur: case occupee par l'ange." << endl;
      ok = false;
    }
    if (caseInaccessible(c)) {
      cout << "Erreur: case inaccessible." << endl;
      ok = false;
    }
  }
  else ok = false;
  return ok;
}
```

f) Les 2 méthodes `bool 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 (la méthode `void afficherPrompt()`)(0.5 pt) la méthode void afficherPrompt()

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

```
void AngeHumain::afficherPrompt() {
  cout << "Ange > ";
}
void DiableHumain::afficherPrompt() {
  cout << "Diable > ";
}
```

## Question 11 (la méthode `Case * choisirUneCase()`) (2 pts) la nouvelle définition de Case choisirUneCase()

a) En utilisant les appels de `afficherPrompt()` et de `verifier(int, int)` donner la définition de `Case * choisirUneCase()` des classes `AngeHumain` et `DiableHumain`.

```
Case * AngeHumain::choisirUneCase () {
      int x, y;
      do {
          afficherPrompt(); cout << "x ? " << endl; cin >> x;
          afficherPrompt(); cout << "y ? " << endl; cin >> y;
      } while (!verifier(x, y));
      return maPartie->monDamier->mesCases[x-1][y-1];
    }
}
```

b) En utilisant l'appel de `caseInaccessible()`, donner la définition de `Case * choisirUneCase()` des classes `AngeAleatoire` et `DiableAleatoire`.

```
Case * AngeAleatoire::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 C++ descendantes de la classe `Joueur`. On placera correctement les mots-clés `virtual`.

```
class Joueur { public:
    void jouer();
    virtual Case * choisirUneCase();
    virtual bool caseInaccessible(Case *);
    virtual void effectuerCoupSurCase(Case *);
};
void Joueur::jouer() {
  Case * c = choisirUneCase();
  effectuerCoupSurCase(c);
}
Case * Joueur::choisirUneCase() { return NULL; }
bool Joueur::caseInaccessible(Case * c) { return false; }
void Joueur::effectuerCoupSurCase(Case * c) { ; }

class Ange : public Joueur { public:
    bool caseInaccessible(Case *);
    void effectuerCoupSurCase(Case *);
};
class AngeHumain : public Ange { public:
  void afficherPrompt();
  Case * choisirUneCase();
  bool verifier(int, int);
};
class AngeAleatoire : public Ange { public:
  Case * choisirUneCase();
  bool caseInaccessible(Case *);
};
class Diable : public Joueur { public :
   void effectuerCoupSurCase(Case *) {
}
class DiableHumain : public Diable { public :
  void afficherPrompt() ;
  Case * choisirUneCase () ;
  bool verifier(int, int) ;
} ;
class DiableAleatoire : public Diable { public :
       Case * choisirUneCase () ;
} ;
```

## Question 12 (minimiser les répétitions de code) (1 pt) l'héritage multiple serait bien utile…

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 ». Idem pour `verifier(int, int)` 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 ?

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 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.**

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

**L'héritage multiple. Oui.**

## Question 13 (avec l'héritage multiple en UML) (1 pt) avec l'héritage multiple

a) On veut faire un diagramme de généralisation utilisant l'héritage multiple ; quelles sont les deux classes à rajouter au modèle UML ?

**Aléatoire et Humain.**

b) On crée deux nouvelles sous-classes de la classe `Joueur` : `Humain` et `Aleatoire`. 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. **Case * choisirUneCase() est placée dans Aléatoire et dans Humain**.
**verifier(int, int) et afficherPrompt() sont placées dans Humain**.

## Question 14 (avec l'héritage multiple en C++) (1 pt) l'héritage multiple en C++

En C++, l'intégration de l'héritage multiple avec la redéfinition de méthodes pose le problème suivant : si la classe `M` a trois sous classes `A`,`B` et `X` ; si `AX` est sous-classe de `A` et de `X`, si `BX` est sous-classe de `B` et de `X` ; si les méthodes virtuelles `M::m()`, `A::m()` et `B::m()` sont définies, si la méthode `X::x()` appelle la méthode `m()` pour un objet de la classe `AX`, alors malheureusement la méthode `M::m()` peut être éventuellement appelée au lieu de la méthode `A::m()`.

L'explication est la suivante. Lorsque l'on est dans `AX`, la méthode `m()` n'est pas définie explicitement et le C++ doit chercher quelle méthode `m()` appeler. Pour trouver la bonne méthode `m()`, le C++ a deux possibilités : remonter l'arbre de généralisation allant de `AX` à `M` *en passant par* `A` ou bien *en passant par* `X`. Dans notre exemple, comme l'appel de `m()` a lieu dans la classe `X`, le C++ passe par `X` et, une fois arrivé en `M`, il appelle la méthode `M::m()`. Il *ne redescend pas* l'arbre de généralisation vers `A` ou `B` pour trouver une redéfinition plus spécialisée de la méthode `m()`.

Proposer une généralisation UML contenant les méthodes adéquates et résolvant ce problème,

a) sur l'exemple des classes `M`, `A`, `B`, `X`, `AX` et `BX`,

b) sur l'exemple de l'ange et du diable.

On explicitera éventuellement le corps de certaines méthodes.

C++ appellent toujours les méthodes concrètes si celles-ci sont redéfinies, **donc l'idée est de réécrire systématiquement les méthodes concrètes et que celles-ci appellent explicitement la bonne méthode abstraite en une ligne**.
dans les methodes AngeHumain : :choisirUneCase et DiableHumain : :choisirUneCase, il y aura uniquement l'appel explicite a Humain : : choisirUneCase, rien de plus.

Sur l'exemple de l'ange et du diable la solution est :

```
class Joueur { public:
    void jouer();
    virtual Case * choisirUneCase();
    virtual bool caseInaccessible(Case *);
    virtual void effectuerCoupSurCase(Case *);
 };

void Joueur::jouer() {
  Case * c = choisirUneCase();
  effectuerCoupSurCase(c);
}
```

```
Case * Joueur::choisirUneCase() {
  return NULL;
}

bool Joueur::caseInaccessible(Case * c) {
  return false;
}

void Joueur::effectuerCoupSurCase(Case * c) {
}

class Ange : public Joueur { public:
    bool caseInaccessible(Case *);
    void effectuerCoupSurCase(Case *);
};


bool Ange::caseInaccessible(Case * c) {
  return (c->distance(maCase) > puissance);
}

void Ange::effectuerCoupSurCase(Case * c) {
  maCase->monAnge = NULL;
  maCase = c;
  c->monAnge = this;
}

class Diable : public Joueur { public:
  void effectuerCoupSurCase(Case *);
  bool caseInaccessible(Case *);
};

void Diable::effectuerCoupSurCase(Case * c) {
  c->bouchee = true;
}

bool Diable::caseInaccessible(Case * c) {
  return false;
}

class Humain : public Joueur { public:
  Humain(Partie *);
  Case * choisirUneCase();
  bool verifier(int, int);
  virtual void afficherPrompt();
};

Humain::Humain(Partie * p) : Joueur(p) {
}

Case * Humain::choisirUneCase() {
  int x, y;
  do {
    afficherPrompt(); cout << "x ? "; cin >> x;
    afficherPrompt(); cout << "y ? "; cin >> y;
  } while (!verifier(x, y));
  return maPartie->monDamier->mesCases[x-1][y-1];
}

bool Humain::verifier(int x, int y) {
  bool 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) {
      cout << "Erreur: case bouchee." << endl;
      ok = false;
    }
    if (c->monAnge!=NULL) {
      cout << "Erreur: case occupee par l'ange." << endl;
      ok = false;
    }
    if (caseInaccessible(c)) {
      cout << "Erreur: case inaccessible." << endl;
      ok = false;
    }
  }
  else ok = false;
  return ok;
}

void Humain::afficherPrompt() {
  cout << "Bug" << endl;
}

class Aleatoire : public Joueur { public:
  Aleatoire(Partie *);
  Case * choisirUneCase();
};

Aleatoire::Aleatoire(Partie * p) : Joueur(p) {
}

Case * Aleatoire::choisirUneCase() {
  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;
}

class AngeHumain : public Ange, public Humain { public:
  AngeHumain(Partie *, int);
  void afficherPrompt();
  Case * choisirUneCase();
```

```
  bool caseInaccessible(Case *);
};

AngeHumain::AngeHumain(Partie * pa, int pui) : Ange(pa, pui), Humain(pa) {}

void AngeHumain::afficherPrompt() {
  cout << "Ange > ";
}

Case * AngeHumain::choisirUneCase() {
  return Humain::choisirUneCase();
}

bool AngeHumain::caseInaccessible(Case * c) {
  return Ange::caseInaccessible(c);
}

class AngeAleatoire : public Ange, public Aleatoire { public:
  AngeAleatoire(Partie *, int);
  Case * choisirUneCase();
  bool caseInaccessible(Case *);
};

AngeAleatoire::AngeAleatoire(Partie * pa, int pui) : Aleatoire(pa),
Ange(pa, pui) {}

Case * AngeAleatoire::choisirUneCase() {
  return Aleatoire::choisirUneCase();
}

bool AngeAleatoire::caseInaccessible(Case * c) {
  return Ange::caseInaccessible(c);
}

class DiableHumain : public Diable, public Humain { public:
  DiableHumain(Partie *);
  void afficherPrompt();
  Case * choisirUneCase();
  bool caseInaccessible(Case *);
};

DiableHumain::DiableHumain(Partie * p) : Diable(p), Humain(p) {}

void DiableHumain::afficherPrompt() {
  cout << "Diable > ";
}

Case * DiableHumain::choisirUneCase() {
  return Humain::choisirUneCase();
}

bool DiableHumain::caseInaccessible(Case * c) {
  return false;
}

class DiableAleatoire : public Diable, public Aleatoire { public:
  DiableAleatoire(Partie *);
  Case * choisirUneCase();
  bool caseInaccessible(Case *);
};

DiableAleatoire::DiableAleatoire(Partie * p) : Diable(p), Aleatoire(p) {}
```

```
Case * DiableAleatoire::choisirUneCase() {
  return Aleatoire::choisirUneCase();
}

bool DiableAleatoire::caseInaccessible(Case * c) {
  return Diable::caseInaccessible(c);
}
```

# Annexe

```
void main () {
  cout << "Jeu de l'Ange et du Diable." << endl;
  char r; int t;
  do {
    cout << "quitter             (q)" << endl;
    cout << "faire une partie    (f)" << endl;
    cin >> r;
    if (r=='f') {
      cout << "\tTaille du damier ? " << endl;
      cin >> t;
      Partie * p;
      p->initialiser();
      p->faire();
    }
  } while (r!='q');
  cout << "Au revoir." << endl;
}

class Partie { public:
  Ange * monAnge;
  Diable * monDiable;
  Damier * monDamier;
  Joueur * trait;
  bool gagnee;
  Partie(int t);
  void faire();
  Joueur * autreJoueur();
  void initialiser();
};

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

void Partie::faire() {
  cout << "Debut de la partie." << endl;
  monDamier->afficher();
  do {
    trait->jouer();
    monDamier->afficher();
    gagnee = monAnge->jeSuisBloque() || monAnge->jeSuisLibre();
    trait = autreJoueur();
  } while (!gagnee);
  cout << "Fin de la partie." << endl;
}

Joueur * Partie::autreJoueur() {
  if (trait == monAnge) return monDiable;
  else return monAnge;
}

void Partie::initialiser() {
  char r;
  cout << "\tAnge Humain ou Aleatoire ? (h/a)" << endl;
  cin >> r;
  if (r=='h') monAnge = new AngeHumain(this, 1);
  if (r=='a') 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];
  cout << "\tDiable Humain ou Aleatoire ? (h/a)" << endl;
  cin >> r;
  if (r=='h') monDiable = new DiableHumain(this);
  if (r=='a') monDiable = new DiableAleatoire(this);
  trait = monAnge;
}
```

```
#define TAILLEMAX 101

class Damier { public:
  Case * mesCases[TAILLEMAX][TAILLEMAX];
  int taille;
  Damier(int);
  void afficher();
};

Damier::Damier(int t) {
  taille = t;
  int i, j;
  for (i=0; i<taille; i++) {
    for (j=0; j<taille; j++) {
      mesCases[i][j] = new Case(i, j, this);
    }
  }
}

void Damier::afficher() {
  int i, j;
  cout << "   ";
  for (i=1; i<=taille; i++) cout << " " << i << " ";
  cout << endl;
  for (i=0; i<taille; i++) {
    cout << " " << (i+1) << " ";
    for (j=0; j<taille; j++) {
      mesCases[i][j]->afficher();
    }
    cout << " " << (i+1) << " " << endl;
  }
  cout << "   ";
  for (i=1; i<=taille; i++) cout << " " << i << " ";
  cout << endl;
}

class Case { public:
  Damier * monDamier;
  bool bouchee;
  Ange * monAnge;
  int x;
  int y;

  Case(int, int, Damier *);
  void afficher();
  int distance(Case *);

};

Case::Case(int a, int b, Damier * d) {
  monDamier = d;
  x = a;
  y = b;
  monAnge = NULL;
  bouchee = false;
}

void Case::afficher() {
  if (monAnge != NULL) cout << ">A<";
  else if (bouchee) cout << " @ ";
  else cout << " + ";
}

int Case::distance(Case * c) {
 return max( abs(x - c->x), abs(y - c->y));
}
```

```
class Joueur { public:
    Partie * maPartie;
    Joueur(Partie *);
    virtual void jouer();
};

Joueur::Joueur(Partie * p) {
  maPartie = p;
}

void Joueur::jouer() {
  ;
}
```

```
class Ange : public Joueur { public:
    int puissance;
    Case * maCase;
    Ange(Partie *, int);
    bool jeSuisBloque();
    bool jeSuisLibre();
};

Ange::Ange(Partie * pa, int pui) : Joueur(pa) {
  puissance = pui;
}

bool Ange::jeSuisBloque() {
  bool uneCaseLibre = false;
  int t = maPartie->monDamier->taille;
  for (int i=0; i<t; i++) {
    for (int 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;
}

bool Ange::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 : public Joueur { public:
  Diable(Partie *);
};

Diable::Diable(Partie * p) : Joueur(p) {
}
```

```
class AngeHumain : public Ange { public:
  AngeHumain(Partie *, int);
  void jouer();
};

AngeHumain::AngeHumain(Partie * pa, int pui) : Ange(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).

void AngeHumain::jouer() {

  int x, y;
  bool ok;

  // l'utilisateur choisit un coup

  do {

    // l'utilisateur tape un coup

    cout << "Ange > x ? ";
    cin >> x;
    cout << "Ange > y ? ";
    cin >> y;

    // le programme verifie le coup

    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) {
      cout << "Erreur: case bouchee." << endl; ok = false;
      }
      if (c->monAnge!=NULL) {
      cout << "Erreur: case occupee par l'ange." << endl; ok = false;
      }

      // la case est-elle inaccessible a l'ange ?
      if (c->distance(maCase) > puissance) {
      cout << "Erreur: case inaccessible." << endl; 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 : public Diable { public:
  DiableHumain(Partie *);
  void jouer();
};


DiableHumain::DiableHumain(Partie * p) : Diable(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).

void DiableHumain::jouer() {

  int x, y;
  bool ok;

  // l'utilisateur choisit un coup

  do {

    // l'utilisateur tape un coup

    cout << "Diable > x ? ";
    cin >> x;
    cout << "Diable > y ? ";
    cin >> y;

    // le programme verifie le coup

    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) {
      cout << "Erreur: case bouchee." << endl; ok = false;
      }
      if (c->monAnge!=NULL) {
      cout << "Erreur: case occupee par l'ange." << endl; 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 : public Ange { public:
  AngeAleatoire(Partie *, int);
  void jouer();
};


AngeAleatoire::AngeAleatoire(Partie * pa, int pui) : Ange(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).

void AngeAleatoire::jouer() {

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

  // le programme choisit un coup

  n = 0;  // on compte le nombre de coups possibles.
  for (i=0; i<t; i++) {
    for (j=0; j<t; j++) {
      Case * c = maPartie->monDamier->mesCases[i][j];
      if ((c->distance(maCase)<=puissance) && !(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++) {
      Case * c = maPartie->monDamier->mesCases[i][j];
      if ((c->distance(maCase)<=puissance) && !(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 : public Diable { public:
  DiableAleatoire(Partie *);
  void jouer();
};

DiableAleatoire::DiableAleatoire(Partie * p) : Diable(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).

void DiableAleatoire::jouer() {
  int x = 0;
  int y = 0;
  int t = maPartie->monDamier->taille;
  int i, j, n, r;

  // le programme choisit un coup

  n = 0;  // on compte le nombre de coups possibles.
  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; // on selectionne un coup aleatoire dans les coups possibles.
  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;
}


class Alea { public:
  static int engendrer(int);
};

int Alea::engendrer(int n) {
  int a = 1 + random() % n;
  return a;
}
```