

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)

Le programme principal est situé dans la classe `Partie`. Que manque-t-il dans ce main ? Corriger le main en conséquence.

Question 2 (les classes `Damier` et `Case`)

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 ?

Question 3 (modélisation UML)

- Dessiner le diagramme de généralisation UML de ce programme.
- 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` ?
- Quelles sont les méthodes redéfinies ?

Question 4 (spécifications d'interface)

- Ce programme permet-il de faire plusieurs parties à la suite ? Quelle est la méthode qui correspond au déroulement d'une partie complète ?
- Peut-on faire jouer deux joueurs humains l'un contre l'autre ?
- Peut-on faire jouer deux joueurs aléatoires l'un contre l'autre ?
- Peut-on faire jouer un joueur humain contre un joueur aléatoire ?
- Peut-on faire jouer deux anges l'un contre l'autre ?
- Peut-on faire jouer deux diables l'un contre l'autre ?
- Un joueur humain peut-il jouer le diable ? l'ange ?
- Un joueur aléatoire peut-il jouer le diable ? l'ange ?
- dessiner un diagramme de séquence UML correspondant à l'interaction de l'utilisateur avec ce programme dans le cas d).
- En une phrase, que ferait le programme si on enlevait le mot-clé `virtual` devant la méthode `jouer()` de la classe `Joueur` ?

Question 5 (diagramme d'instances UML)

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.

Question 6 (1^{er} aperçu des méthodes 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.

- Peut-on dire que les 4 méthodes jouer() font globalement la même chose : d'abord choisir le coup, puis effectuer le coup ?
- On veut que la définition de la méthode void jouer() soit identique pour les 4 classes concrètes de joueur. Dans quelle classe la placer ?

Question 7 (1^{er} aperçu de 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);
}
```

- Peut-on dire que les 4 méthodes Case * choisirUneCase() font la même chose ?
- Peut-on dire que les 2 méthodes Case * choisirUneCase() des classes AngeHumain et DiableHumain font approximativement la même chose ?
- Que les 2 méthodes Case * choisirUneCase() des classes AngeAleatoire et DiableAleatoire font approximativement la même chose ?
- Que les méthodes Case * choisirUneCase() des joueurs « humains » sont très différentes de celles de joueurs « aléatoires » ?
- Donner une définition de la méthode Case * choisirUneCase() de la classe Joueur.

Question 8 (la méthode void effectuerCoupSurCase(Case*))

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

- Peut-on dire que les 4 méthodes void effectuerCoupSurCase(Case*) font la même chose ?
- Peut-on dire que les 2 méthodes void effectuerCoupSurCase(Case*) des classes AngeHumain et AngeAleatoire font approximativement la même chose ?
- Que les 2 méthodes void effectuerCoupSurCase(Case*) des classes DiableHumain et DiableAleatoire font approximativement la même chose ?
- Que les méthodes void effectuerCoupSurCase(Case*) des anges sont différentes de celles des diables ?
- En déduire la définition dans la classe Joueur et les redéfinitions de la méthode void effectuerCoupSurCase(Case*) dans les classes adéquates.

Question 9 (la méthode 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 ».

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

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

Question 10 (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.

Question 11 (la méthode `Case * choisirUneCase()`)

- En utilisant les appels de `afficherPrompt()` et de `verifier(int, int)` donner la définition de `Case * choisirUneCase()` des classes `AngeHumain` et `DiabloHumain`.
- En utilisant l'appel de `caseInaccessible()`, donner la définition de `Case * choisirUneCase()` des classes `AngeAleatoire` et `DiabloAleatoire`.
- Donner la nouvelle description des méthodes des classes C++ descendantes de la classe `Joueur`. On placera correctement les mots-clés `virtual`.

Question 12 (minimiser les répétitions de code)

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 ».

- Afin de supprimer cet inconvénient, une 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 résout-il le problème ?
- De quel outil de modélisation UML a-t-on besoin pour résoudre ce problème ? Cet outil existe-t-il en C++ ?

Question 13 (avec l'héritage multiple en UML)

- On veut faire un diagramme de généralisation utilisant l'héritage multiple ; quelles sont les deux classes à rajouter au modèle UML ?
- On crée deux nouvelles sous-classes de la classe `Joueur` : `Humain` et `Aleatoire`. Proposer une généralisation UML avec un placement adéquat des méthodes.

Question 14 (avec 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,

- sur l'exemple des classes `M`, `A`, `B`, `X`, `AX` et `BX`,
- sur l'exemple de l'ange et du diable.

On explicitera éventuellement le corps de certaines méthodes.

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 rime 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->x==0) || (c->y==0) || (c->x==t-1) || (c->y==t-1) )
                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->x==0) || (c->y==0) || (c->x==t-1) || (c->y==t-1) )
                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;
}

```