

# Mini-projet Programmation Objet (Echiquier)

1. Objectif.....	1
1.1 Présentation.....	2
1.2 Interface.....	2
2. Niveau de rendu attendu.....	3
2.1 Fonctionnalité Base (5 pts) .....	3
2.2 Fonctionnalité Prise (+1 pt) .....	4
2.3 Fonctionnalité Pions (+1 pt).....	4
2.4 Fonctionnalité Obstacle (+2 pt) .....	4
2.5 Fonctionnalité Prise en passant (+2 pt) .....	4
2.6 Fonctionnalité Echec (+3 pts) .....	4
2.7 Fonctionnalité Roque (+2 pts) .....	4
2.8 Fonctionnalité Promotion (+1 pt) .....	4
2.9 Fonctionnalité Mat (+3 pts) .....	4
2.10 Fonctionnalité Pat (+3 pts) .....	4
3. Contraintes.....	5
4. Modélisation .....	7
4.1 La classe Square .....	7
4.2 La classe Piece et ses sous-classes.....	8
4.3 La classe Echiquier .....	8
4.4 La classe Jeu .....	10

## 1. Objectif

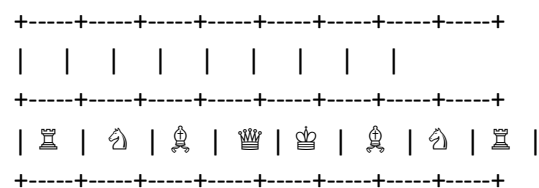
Proposer une interface simple, en mode texte, permettant de représenter des mouvements de pièces d'échecs.

Cet exercice a pour objet de démontrer les notions suivantes:

- héritage
- polymorphisme
- méthodes virtuelles

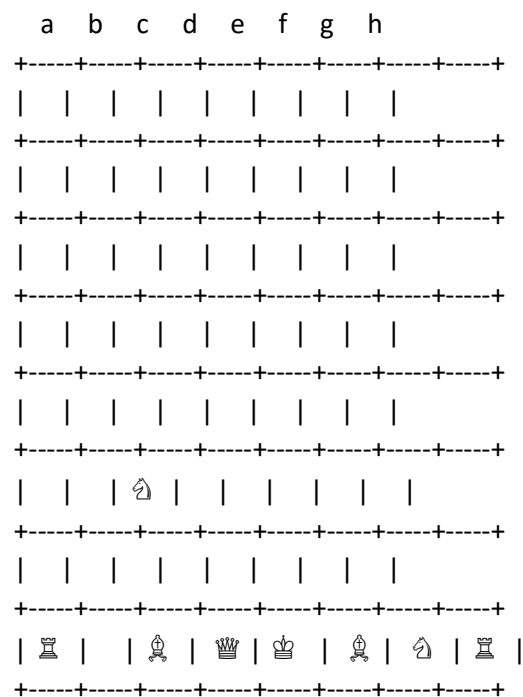
On souhaite pouvoir accomplir les actions telles que présentées dans le programme principal suivant. Ce programme propose comme seule interface à l'utilisateur la saisie d'un mouvement en notation algébrique, comme par exemple *b1c3* qui désigne le mouvement de la pièce initialement située sur la case *b1*) (c-a-d en colonne *b*, rangée *1*) vers la case *c3*.

## 1.2 Interface



Coup (eg. a1a8) ? b1c3

-> déplacement de ♘ b1c3



Coup (eg. a1a8) ?

## 2. Niveau de rendu attendu

Selon le temps consacré, vous pouvez raffiner les fonctionnalités de cette interface jusqu'à un niveau pleinement fonctionnel (ce qui n'est pas si simple). Voici différents niveaux de fonctionnalités que vous pouvez atteindre.

Un barème indicatif est donné pour chaque niveau: c'est le nombre de points **maximum** sous l'hypothèse que le code est de bonne qualité et implémente correctement les fonctionnalités du niveau. Après le niveau *Base*, chaque niveau de fonctionnalité est presque indépendant mais l'ordre

### 2.1 Fonctionnalité Base (5 pts)

Au ce niveau, vous proposez une preuve de concept. Le niveau 1 est atteint quand on a

- un affichage de l'échiquier et des pièces
- que l'utilisateur peut saisir un coup ordinaire (pas les roques)
- que le programme gère l'alternance des coups blanc/noir
- le programme se termine par /quit

L'interface vérifie les contraintes suivantes pour un déplacement :

- vérifier que les coordonnées saisies sont dans l'échiquier,
- vérifier la légalité géométrique du mouvement de la pièce manipulée : en diagonale pour le fou, en ligne ou en colonne pour la tour, en ligne ou en colonne ou en diagonale pour la dame, et le pion avance de 1 ou 2 cases (contraintes supplémentaires dans un niveau ultérieur).
- vérifier que la case d'origine contient bien une pièce.

Quand l'une de ces contraintes n'est pas respectée, l'échiquier devra resté inchangé et un message d'erreur pertinent affiché.

A ce stade, on ne demande pas de vérifier qu'un obstacle se situe sur la trajectoire, ou que la case d'arrivée soit vide ou une pièce opposée.

## 2.2 Fonctionnalité Prise (+1 pt)

- Dans un déplacement, si la case de destination est occupée par une pièce de couleur opposée, c'est une prise, mais est un mouvement illégal si la case est occupée par une pièce de même couleur.

## 2.3 Fonctionnalité Pions (+1 pt)

- Le mouvement des pions est conforme aux règles suivantes : il peut avancer de 2 cases à son premier coup, 1 case autrement, et prendre en diagonale vers l'avant.

## 2.4 Fonctionnalité Obstacle (+2 pt)

- On vérifie la légalité des déplacements en vérifiant en plus qu'une pièce ne passe pas au-dessus d'une autre (concerne Tour, Dame, Fou).

## 2.5 Fonctionnalité Prise en passant (+2 pt)

- On ajoute la prise en passant du pion.

## 2.6 Fonctionnalité Echec (+3 pts)

- On ajoute la détection de l'échec. Après un mouvement, on signale dans l'affichage qu'un roi est en échec le cas échéant. On vérifie aussi qu'un mouvement ne met pas le roi en échec.

## 2.7 Fonctionnalité Roque (+2 pts)

- On ajoute le mouvement du roque, petit roque noté O-O, et grand roque noté O-O-O. On pourra se contenter de vérifier uniquement que le roi et la tour impliquée n'ont jamais bougé, sans vérifier qu'une des cases du roque est sous la menace d'une pièce adverse.

## 2.8 Fonctionnalité Promotion (+1 pt)

- On ajoute la promotion du pion arrivé sur la rangée 8 pour les blancs, 1 pour les noirs. On pose alors la question en mode texte, en quelle pièce doit être promu le pion. L'utilisateur tape un caractère parmi {Q,R,B,K} pour respectivement promouvoir en Reine, Tour, Fou, Cavalier.

## 2.9 Fonctionnalité Mat (+3 pts)

- On détecte l'échec et mat et on arrête la partie.

## 2.10 Fonctionnalité Pat (+3 pts)

- On détecte le pat et on arrête la partie. On vérifiera en priorité qu'aucune pièce du joueur à qui c'est le tour ne peut bouger légalement. Pour être complet, on peut implémenter la règle des 3 positions répétées successivement, et les 50 coups joués sans prise.

### 3. Contraintes

- Votre programme doit lire les coups entrés sur l'entrée standard unix.
- Les coups seront notés en notation algébrique comme indiqué au début (par exemple a2a4 pour faire avancer le pion blanc gauche de 2 cases). Voici un exemple d'extrait de code qui permet de vérifier la validité d'une entrée.

































- `#include <regex>`

```
bool saisie_correcte(string const & cmd) {  
    regex movmtpattern("[a-h][1-8][a-h][1-8]");  
    return regex_match(cmd,movmtpattern);  
}
```

```
bool saisie_correcte_petitroque(string const & cmd) {  
    regex movmtpattern("(O|o|0)-(O|o|0)");  
    return regex_match(cmd,movmtpattern);  
}
```

- Dans un objectif de test, votre programme terminera en affichant sur stdout la position finale du jeu sous une forme canonique : on affiche le contenu de chaque case en parcourant l'échiquier dans l'ordre (a1,b1, ..., h1, a2, b2, ..., h2, ..., a8, ..., h8). Le contenu de la case est codifié par les lettres suivantes :
  - un caractère w ou b pour la couleur, respectivement blanche ou noire,
  - un caractère pour la pièce: Q pour Reine (Queen), K pour Roi (King), B pour fou (Bishop), N pour cavalier (kNight), R pour Tour (Rook), et P pour pion (Pawn).
  - on n'affiche rien si la case est vide, et chaque case est séparée par une virgule (aucun espace).

Par exemple, la position finale suivante:

	a	b	c	d	e	f	g	h
8								
7								
6								
5								
4								
3								
2								
1								

2. Noir -> (eg. d2d4) ? /quit

est codée par :

wR,wN,wB,wQ,wK,wB,,wR,wP,wP,wP,,wP,wP,wP,wP,,,,,wN,,,,,wP,,,,,,bP,,,,,,bP,bP,bP,,  
bP,bP,bP,bP,bR,bN,bB,bQ,bK,bB,bN,bR,

Voici un exemple de fonction qui peut implémenter cela (utilise la classe Square conseillée mais peut être adapté facilement. Cette fonction appelle une fonction pgn\_piece\_name() chargée de rendre le symbole utilisé dans la notation PGN.

```
string Echiquier::canonical_position() const {
    string output;
    for (size_t row(1); row<=8; row++){
        for (char col('a'); col<='h'; col++) {
            Square square(col+to_string(row));
            if (!est_case_vide(square))
                // get pieces with their PGN names,
                // true -> with P for pawns, true -> w/b for colors.
                output += pgn_piece_name(get_piece(square)->to_string(),true,true);
            output += ",";
        }
    }
    return output;
}
```

La fonction pgn\_piece\_name() est la suivante. Elle pourrait être simplifiée mais cette version admet des paramètres qui permettent plus de flexibilité : afficher les pions (oui ou non), faire précéder la pièce de sa couleur (oui ou non).

```
string Echiquier::pgn_piece_name(string const name, bool view_pawn, bool view_color)
const {

    string psymb;
    if (name=="\u2656") psymb="R"; // Rook W
    else if (name=="\u2658") psymb="N"; // Knight W
    else if (name=="\u2657") psymb="B"; // Bishop W
    else if (name=="\u2655") psymb="Q"; // Queen W
    else if (name=="\u2654") psymb="K"; // King W
    else if (name.rfind("\u2659",0)==0 && view_pawn) psymb= "P"; // Pawn W
    if (psymb.size()>0) { // one of the white piece has been found
        if (view_color)
            return "w"+psymb;
        else
            return psymb;
    }
    if (name=="\u265C") psymb= "R"; // Rook B
    else if (name=="\u265E") psymb= "N"; // Knight B
    else if (name=="\u265D") psymb= "B"; // Bishop B
    else if (name=="\u265B") psymb= "Q"; // Queen B
```

```

else if (name=="\u265A") psymb= "K"; // King B
else if (name.rfind("\u265F",0)==0 && view_pawn) psymb= "P"; // Pawn B
if (psymb.size()>0) { // one of the black piece has been found
    if (view_color)
        return "b"+psymb;
    else
        return psymb;
}
else return "";
}

```

Le fait de lire sur l'entrée standard facilite les tests. Vous pouvez avoir un fichier contenant des coups, par exemple le fichier suivant partie.txt e2e4

```

e7e5
g1f3
b8c6
d1d2
d2d3
g8f6
c1g5
f8e7
/quit

```

que vous pouvez jouer par la commande suivante, en supposant que votre exécutable s'appelle echecs :

```
$ cat partie.txt | ./echecs
```

Les projets rendus seront testés de cette manière.

## 4. Modélisation

Voici une description d'une conception possible. Toute autre conception peut être envisagée. Cependant, les explications suivantes sont inter-dépendantes, c-a-d qu'une classe proposée dépend souvent d'une autre, et attention à ne pas prendre une partie seulement des propositions sans comprendre comment elles s'articulent avec le reste.

### 4.1 La classe Square

- Pour manipuler les coordonnées, il peut être intéressant d'avoir une classe représentant une **case** de l'échiquier (la classe ne s'appelle pas **case** car c'est un mot clé de C/C++). Cette classe n'est pas essentielle mais simplifie le nombre de paramètres à chaque fois qu'on appelle des fonctions manipulant des coordonnées (ligne,colonne).
- Etant donné la notation algébrique proposée à l'utilisateur, cette classe peut permettre des conversions entre coordonnées en notation algébrique et indices de tableau. Par exemple un constructeur Square("a1") pourra avantageusement convertir la notation *a1* en ligne=0 et colonne=0. On pourra aussi y ajouter des méthodes facilitant l'affichage, par exemple to\_string() renvoyant "a1" pour une coordonnées (0,0).

## 4.2 La classe Piece et ses sous-classes

- Cette classe est abstraite. En effet, les seules entités réellement manipulées seront les Tour, Cavaliers, Roi, ... . Toutes ses méthodes ne seront pas forcément abstraites. Par exemple, une méthode utile pourrait être affiche() affichant le nom de la pièce.
- Avec cette classe Piece viennent donc autant de sous-classes que de variantes de pièce, avec leur implémentation propre à leur comportement. Par exemple, on pourrait avoir une méthode: est\_mouvement\_legal(case\_origine,case\_destination) qui indiquera si ce mouvement de pièce est légal : en diagonale pour un fou, en ligne droite pour une tour, ...
- Il est suggéré de mémoriser comme attribut de pièce, entre autres:
  - son nom (pour lequel on peut utiliser l'UTF-8 : le caractère \u2656 affichera ♖), et éventuellement un identifiant unique.
  - sa couleur
  - sa position
- Des méthodes accesseurs viennent naturellement pour exploiter ses informations: get\_pos() pour récupérer la position courante de la pièce, get\_couleur(), to\_string() pour afficher le nom de la pièce,...

## 4.3 La classe Echiquier

- Cette classe représente le stockage des pièces créées. On peut proposer un stockage dans un tableau 8x8 dont chaque case pointe vers une pièce allouée en mémoire.
- Piece \* echiquier[8][8];

La mise en oeuvre de méthodes polymorphes sur les pièces de l'échiquier nécessite de stocker des pointeurs. On peut stocker dans ce tableau tout type de pièce héritant de Piece, comme par exemple l'adresse que renverrait un new Tour(...) (en supposant la classe Tour définie comme sous-classe de Piece). Quand une case est vide, on peut stocker le pointeur nul (nullptr).

- Un objet de la classe Echiquier devrait être le seul autorisé à faire les manipulations sur le tableau echiquier . Par exemple, le déplacement *a1b2*, impliquant les cases echiquier[0][0] (origine) et echiquier[1][1] (destination) donnera en fin de compte lieu à copier le pointeur se trouvant en echiquier[0][0], en echiquier[1][1], puis à mettre nullptr en echiquier[0][0] qui devient case libre après le déplacement. Pour cela, on pourra penser aux méthodes publiques:
  - deplace( ...)
  - pose\_piece(...) permettant de placer les pièces au départ.
- Le **constructeur** devrait initialiser l'echiquier à la position de départ. Pour cela, on peut confier à ce constructeur les rôles de :
  - créer l'échiquier comme un tableau 8x8 capable de stocker des pointeurs vers des pièces
  - créer les instances des différentes pièces du jeu (qui ne seront de toute façon créés qu'une fois si on n'implémente la promotion du pion).

L'exemple ci-dessous montre ce que pourrait être l'initialisation. Notez qu'on utilise dans cet exemple des vecteurs piecesb, piecesn, pionsb, pionsn (attributs de la classe) pour stocker les pointeurs vers les pièces créées pour les copier ensuite dans le tableau echiquier. En effet, cela peut être pratique de parcourir les pièces par catégorie. Mais cela n'est pas strictement nécessaire et chaque pointeur vers une pièce pourrait être directement stocké dans le tableau. Par exemple



```
echiquier[0][0] = new Tour (Blanc, "\u2656 ", Square(0,0));
```

En utilisant des vecteurs par catégorie, le code peut ressembler à cela:

```
// constructeur
Echiquier::Echiquier ()
{

    alloc_mem_echiquier(); // --> alloue un tableau équivalent à un Piece *[8][8]
        // en initialisant les cases à nullptr
        // et alloue des vecteurs piecesb, piecesn, pionsb
    // Constructeur (Couleur,nom_affiché, case)
    piecesb[0] = new Tour (Blanc, "\u2656 ", Square(0,0));
    piecesb[1] = new Cavalier(Blanc, "\u2658 ", Square(0,1));
    piecesb[2] = new Fou (Blanc, "\u2657 ", Square(0,2));
    piecesb[3] = new Dame (Blanc, "\u2655 ", Square(0,3));
    piecesb[4] = new Roi (Blanc, "\u2654 ", Square(0,4));
    piecesb[5] = new Fou (Blanc, "\u2657 ", Square(0,5));
    piecesb[6] = new Cavalier(Blanc, "\u2658 ", Square(0,6));
    piecesb[7] = new Tour (Blanc, "\u2656 ", Square(0,7));
    piecesn[0] = new Tour (Noir, "\u265C ", Square(7,0));
    piecesn[1] = new Cavalier(Noir, "\u265E ", Square(7,1));
    piecesn[2] = new Fou (Noir, "\u265D ", Square(7,2));
    piecesn[3] = new Dame (Noir, "\u265B ", Square(7,3));
    piecesn[4] = new Roi (Noir, "\u265A ", Square(7,4));
    piecesn[5] = new Fou (Noir, "\u265D ", Square(7,5));
    piecesn[6] = new Cavalier(Noir, "\u265E ", Square(7,6));
    piecesn[7] = new Tour (Noir, "\u265C ", Square(7,7));

    // allocation des pions
    for (unsigned char i(0);i<NBCOL;i++) {
        pionsb[i] = new Pion(Blanc, "\u2659 ", Square(1,i));
        pionsn[i] = new Pion(Noir, "\u265F ", Square(6,i));
    }
    // Pose des pieces en position initiale
    // pose des pieces blanches
    for (unsigned char i(0);i<NBCOL;i++)
        // met à jour le tableau echiquier, à la case donnée par
        // la position courante de la pièce obtenue avec
        // piecesb[i]->get_pos(),
        // avec le pointeur vers la pièce (piecesb[i])
        pose_piece(piecesb[i],piecesb[i]->get_pos());

    // puis pose des pièces noires, pions blancs, pions noirs
    // ....
}
```

- Enfin, la classe doit permettre de visualiser l'échiquier (le plus important pour l'utilisateur). L'affichage montré en exemple est produit par l'extrait de code suivant:

```
void Echiquier::affiche () const {

    string space5 = string(5, ' ');
    cout << endl;
    cout << "  a  b  c  d  e  f  g  h  " << endl;
    cout << " +---+---+---+---+---+---+---+---+ " << endl;
    for (int i(NBCOL-1);i>=0;i--) {
        cout << i+1 << " "; // numérotation ligne dans affichage
        for (int j(0);j<NBCOL;j++) {
            cout << "|";
            if (echiquier[i][j]) {
                cout << "\u0020\u0020"; //U+0020 est un espace utf-8 taille police
                echiquier[i][j]-> affiche();
                cout << "\u0020" << " ";
            }
            else
                cout << space5; // 2 ascii spaces
        }
        cout << "|\n +---+---+---+---+---+---+---+---+ ";
        cout << endl;
    }
}
```

#### 4.4 La classe Jeu

On propose d'instancier un objet de classe Jeu pour interagir avec l'utilisateur.

- Cet objet possède un attribut **échiquier** (un objet de classe Echiquier).
- La classe Jeu propose essentiellement 2 méthodes publiques
  - la gestion d'un coup : saisie du coup, puis toute la "machinerie" que ça implique derrière.
  - afficher l'échiquier (pour que l'utilisateur se rende compte de la nouvelle position)

- Ainsi, l'interaction avec un programme principal pourrait se résumer à :

```
int main() {
    Jeu monjeu;

    // boucle de jeu, s'arrete a la fin de la partie
    bool stop(false);
    do {
        monjeu.affiche();
        stop = monjeu.coup();
    } while (!stop);
}
```

- La méthode la plus compliquée de cette classe Jeu sera la gestion du déplacement suite à la saisie d'un coup. Si on appelle cette méthode deplace, elle devra avoir comme paramètre la position d'origine de la pièce et sa position d'arrivée, et éventuellement la couleur du joueur si vous implémentez l'alternance des coups blancs/noirs.
- La méthode deplace travaillera sur son attribut echiquier en passant cet echiquier aux méthodes publiques de la classe Echiquier pour qu'elles le modifient selon ses instructions.

Author: Stéphane Genaud

Created: 2021-04-09 Fri 16:31