Fatoumata NDOUR / Elias ECHIKR
Langages Objets Avancés (C++)

Notes de Rendu :

Ce qui a était fait :

Le jeu “Butin” est fonctionnel, se référer au README pour son utilisation.
Le jeu “Dames” et le jeu “Safari” ne sont pas terminés, mais leur implémentation est bien avancée.
Le main de Dames affiche un plateau avec des pièces, mais il n’est pas possible d'interagir avec le plateau. Une fonction auxiliaire afficherPlat() permet de reproduire un affichage basique, voir détail ci-dessous.
Le jeu Safari à un plateau fonctionnel avec SFML, ses méthodes de base (déplacement, constructeur) ont été implémentés, et les pions ont été représentés sous forme d’image, voir détail ci-dessous.
Le diagramme UML se trouve en fin de document.

NB : En cas de problème de compilation, il se peut que les #include “Nom.hpp” ne correspondent pas exactement au nom des fichiers. Normalement ce problème est corrigé mais veuillez nous en excuser si ce n’est pas le cas.

Aspects significatifs commun à tout les jeux :

Le plateau est un vecteur de vecteurs de pointeurs afin d’éviter des copies coûteuses et inutiles.
L’héritage et la factorisation sont évoqués plus bas.
Non utilisation des templates car nous n’avons pas vu d’utilités significatives.

Dames :

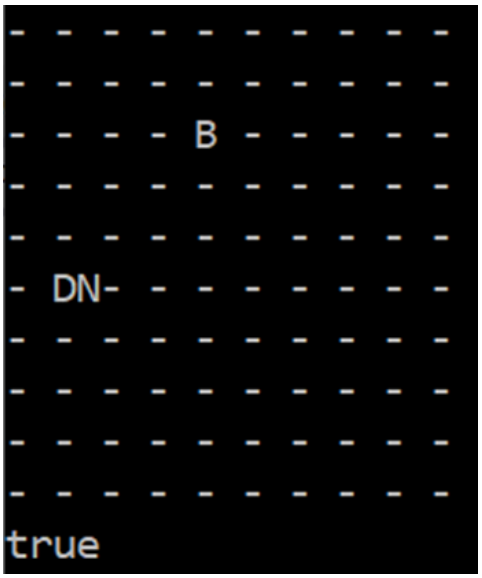
Les méthodes suivantes peuvent être vérifier dans le main, avec un affichage graphique très basique (à l’aide d’une méthode **mat**, **afficherPlat**), on pourra par exemple taper :

```
int main()
{
	Plateau plateau1(10, 10);
	plateau1.ajouterPiece(8, 3, "NOIR", false);
	//plateau1.ajouterPiece(2, 1, "NOIR", true);

	plateau1.afficherPlat();
	cout << boolalpha;
	cout << " " << endl;
	//cout << plateau1.mouvPossible(2,1, 6, 5) << endl;
	plateau1.deplacer(8, 3, 9, 2);
	//cout << plateau1.mouvPossible(3, 3, 5, 5) << endl;
	//cout << plateau1.diagLibre(5, 1, 2, 4) << endl;

	plateau1.afficherPlat();
}
```

pour avoir un aperçu des méthodes présentées ci-dessous. Voici un aperçu de l’affichage basique (sans SFML) :



(Ici B représente un pion blanc, DN une dame noir)

NB : Il peut être utile de désactiver la méthode **PlacerPiece()** du constructeur pour une meilleure visualisation du plateau quand on utilise l'affichage sans SFML.

1.Aspects significatifs :

L'idée était d'utiliser seulement des positions (x_act, y_act) et (x_new, y_new) comment entrée des méthodes afin que pour lier Vue et Modèle, SFML récupère seulement un clic de position (x_act, y_act) et un second de (x_new, y_new).
Toutes les méthodes doivent normalement comporter le mot clé **const** car on ne modifie pas les coordonnées sauf dans **déplacer()**, mais par manque de temps cela n'a pas était possible.

Classe Plateau :

Méthodes :

Constructeur : Initialise un vecteur de taille largeur x longueur , avec des pointeurs nuls partout.
Ensuite on utilise une méthode **placerPiece ()** qui va initialiser des objets pièces (allocation dynamique) et les placer sur le plateau.

Déplacer: Vérifie si un mouvement est possible avec **MouvPossibles()**, si c'est le cas met à jour le plateau à l'aide des méthodes **Prendre**() et en déplaçant la pièce.

Prendre: S'occupe du déplacement de la pièce en mettant à jour le plateau.

Mouv_possibles : La méthode la plus lourde, on aurait souhaité la rendre plus optimale avec l'utilisation de switch ou d'enum, et en utilisant des fonctions annexes (par exemple vérifier qu'un mouvement est possible) mais ce ne fut pas possible à cause de notre gestion

du temps.
Indication sur la méthode Mouv_possibles ():
1. D'abord on vérifie que l'entrée est bien dans le plateau (ie : les positions A et B sont ds le plateau)
2. Ensuite on vérifie qu'il y a une pièce à déplacer à la position A (x_act / y_act)
3. Et que l'endroit ou on veut déplacer la pièce est vide
4. On regarde si on a une dame → logique déplacement des dames
5. Ou si on a pion → logique déplacement des pions

Ens_des_mouvs :
Destructeur: Visite chaque case du plateau et réalloue la mémoire alloué dynamiquement par new Pièce grâce à delete.
prisePossiblePiece () : Booléen qui renvoie true si une prise est possible avec un pion, essentiellement elle vérifie qu'il y a une pièce adverse et que la case adjacente est vide.
prisePossibleDame () : Équivalente mais avec les dames. Utillise la méthode diag_libre pour vérifier que l'espace est vide avant la prise.

diagLibre () : Renvoie vraie si aucune pièce n'est présente entre la position actuelle et la case avant la nouvelle position.

prendrePiece () / PrendreDame () : Permet de delete les pièces lors d'une prise.

Attributs :
m_ fait référence à variable membre.
m_plateau : Vecteur de Vecteur pour représenter matrice / plateau
m_largeur / m_longueur : Utile pour initialiser le vecteur.

Classe Piece :

friend classe de plateau car on veut accéder aux attributs de pièces dans le plateau

Méthodes :
Constructeur:
afficher : Utile pour vérifier la création
mat : Utile pour l'affichage sans SFML.
Getters était utile seulement pour le main avec SFML.

Attributs :
m_position pair <int,int> : Cet attribut n'a pas été utilisé finalement.
m_couleur string : Couleur de la pièce
m_dame bool : Permet de savoir si on est une dame (True) ou un pion (False).

Classe Joueur :
(Voir ci-dessous)

2. Problèmes connus / Extension pas implémenter:

L'implémentation du plateau SFML n'était pas aisée, mais le principal problème rencontrés fut d'implémenter la règle de "Prise obligatoire" et ce problème n'a pas pu être réglé.
A chaque tour le joueur **doit déplacer la pièce qui maximise les prises,** or cela implique pour chaque pièce de simuler leurs déplacements, ce qui implique un coût considérable en mémoire.
L'idée était d'utiliser un algorithme de recherche en profondeur avec FILE (possiblement MinMax) pour connaître le chemin qui maximise les prises..
Il fallait prendre en compte la gestion des copies, la gestion de la mémoire en utilisant pushback du vecteur (pas optimal / NB on aurait pu réserver une certaines mémoires égale au nombres maximales de prise disons toutes les pièces adverses).

Voici un squelette de l'algorithme DFS :

Pour chaque pièce, vérifier s'il y a une prise possible :
    Création d'un nouveau plateau avec le déplacement de la prise Pi.
    Ajouter Pi à un vecteur StockagePlateauAvecPrise
    Garder en mémoire le chemin pour la prise
Ensuite appeler la fonction par récurrence sur les éléments de StockagePlateauAvecPrise tout en gardant un compteur qui compte le nmb de prises.
A la fin, renvoyer le chemin qui maximise le nmb de prises.

Butin :

1.Aspects significatifs :

Classe Plateau (Grille) :

Méthodes :
Constructeur :
getTaille : getter pour obtenir la taille, // en mode protégé
get_gcase : getter renvoie le plateau avec les pions placés

Placer pion: Méthode utiliser exclusivement avec le constructeur (même logique que précédemment)

retirer : Permet de retirer le pion à la coordonnées (x,y) avec delete, on a alors rencontré un problème de segment fault *(voir détail section 2. Problèmes)*.

peutSauter : Si un saut est possible, la méthode renvoie la pièce qu'on peut sauter.
Cette méthode vérifie dans chaque direction si la case d'après est vide et si on a une pièce présente sur la case intermédiaire.

sauter: Récupère la coordonnée du pointeur → Si ce n'est pas nullptr c'est qu'on peut sauter un pièce, dans ce cas on va mettre à jour la position via la méthode **déplacerPiece**.

calculValeurPion : Cette méthode permet le calcul la valeur totale des pièces sur le plateau, c'est une fonction intermédiaire ayant pour but de calculer le score final du joueur gagnant.

Destructeur: Création d'un destructeur personnalisé car utilisation d'allocation dynamique, utilisation de virtual car héritage utilisé.

Attributs :

m_plateau : Vecteur de vecteur de pointeurs.
m_taille : Ici on a pris en compte que largeur = longueur, et on à conserver un seul attribut.

Classe Piece (*pion*) :

Méthodes :

Constructeur: Crée une pièce à partir d'une couleur, d'une posit, et d'une valeur (NB : On aurait pu associer directement une valeur à couleur dans notre constructeur).
Destructeur : Utilisation du destructeur par défaut car pas d'allocation dynamique.
Deplacer : Méthode qui permet de mettre à jour les coordonnés

Attributs :

m_position : Position de la pièce
m_couleur : Couleur (représenté par nom dans la classe Héritage) de la pièce

Classe Joueur :

Méthodes :

Constructeur: Prend un nom et un score, et on initialise a_le_tour = false. Ensuite on initialise un vecteur avec une taille de 64 car dans le pire des cas le joueur ne peut pas capturer plus de 64 pièces.

retirerPieceJaune : Au début du tour, chaque joueur retire un pion jaune.

ChoisirPionJaune : Depuis le Plateau de jeu on récupère la pièce à la position (x,y)

jouerTour : Représente ce que fait le joueur à son tour, renvoie un booléen s'il à réussi à jouer.

a_Le_Tour : La méthode permet de connaître si le Joueur a le tour de jouer.

Getters :
getNom : permet de récupérer le nom du joueur déclaré en privé
getScore : c'est pour avoir accès au score du joueur
getPieceCapturer : renvoie le tableau de pièce capturé par joueur

Destructeur : Permet de libérer la mémoire de tout ce qui à été alloué.

Attributs :

nom : nom du joueur
score : score obtenu à chaque étape du jeu
pièce capturer : l'ensemble des pièces capturées par le **joueur** lors d'une partie

2. Problèmes connus / Extension pas implémenter:

Problèmes connus :

Segmentation fault lié au Plateau :
Après avoir supprimé une pièce du **vecteur de plateau** , on rencontre un problème si on veut déplacer une autre pièce à la case qui contenait la pièce supprimée → le programme se terminait automatiquement avec un *segmentation fault* . On a résolu le problème en ajoutant dans la fonction qui déplace une pièce une ligne qui met la case à la valeur nullptr après le **delete** de cette case.

Affichage du plateau :
Au début, on avait vraiment du mal à comprendre comment la boucle while du window peut tracer et mettre à jour le plateau . On pensait à implémenter des lignes de codes dans le bit de régler ce problème mais on s'est rendu compte qu'une méthode similaire était d'or et déjà inclus dans la librairie SFML. Ensuite il fallait juste mettre la partie du jeu dans la boucle et définir les codes effacent / retracent la grille en dehors de la boucle.

Les templates dans la classe Joueur :
On a voulu créer une classe mère joueur qui factorise les parties communes aux classes Joueurs des différents jeux en utilisant les templates.
Pour ce faire on a déclaré la classe Joueur comme une classe template <typename T> et après on a fait hériter chaque classe joueur avec son type T = classe de la pièce associée à chaque jeux.
Mais finalement cette partie n'est pas opérationnelle car un problème de références non définies apparaît et il n'a pas pu être résolu.

les Héritages :
Le problème persistait entre plateau et pièce mais en incluant les classes mères dans le Makefile on à pu régler les problèmes de références.

Dans le main :
On voulait faire en sorte que tant que le joueur n'a pas fini de capturer les pièces, l' interface graphique ne s'actualise pas directement c'est-à-dire laisse une trace des mouvements effectués. Ces détails n'ont pu être terminés.

Safari

1 . Aspects significatifs

classe Plateau

Méthodes:
→ Les méthodes sont semblables au jeu “Butin”.

Attributs:
→ Mêmes attributs que les autres plateaux.

Classe Piece

Méthodes:
constructeur: Méthodes héritées de la **super classe** Pièce .

Attributs:
Mêmes attributs que les autres sauf le booléen est_capturer qui détermine si l’animal est pris en capture.

Classe Joueur:

Méthodes:
constructeur:
get_nom: renvoie le nom du joueur
get pièces: renvoie les pièces du joueur
placer barrière: placer les barrières sur le plateau il est fait par le joueur
placer_pièces:placer les pièces(animaux) sur le plateau il est fait par le joueur aussi
jouer tour : dans cette partie on décrit ce que fait exactement le joueur à son tour.

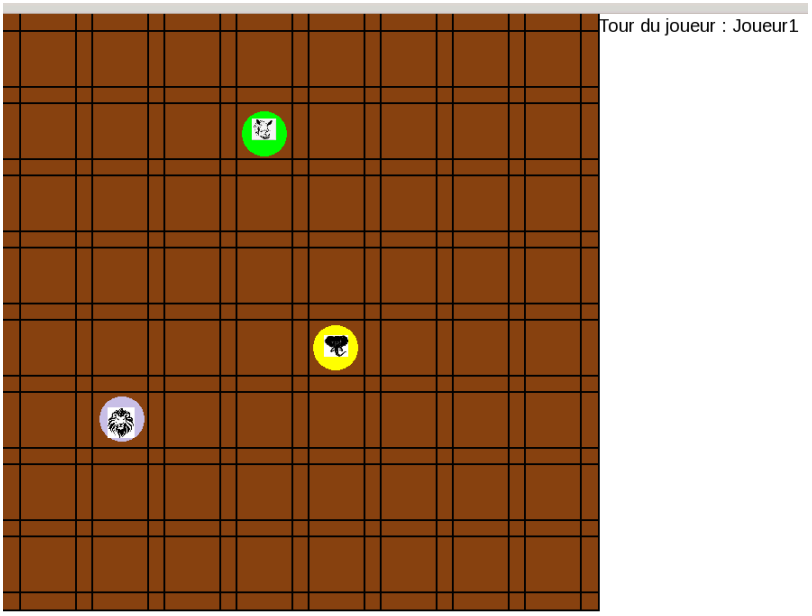
Attributs:
nom: nom du joueur
tableau de pièces: qui lui permet de jouer.

2. Problèmes connus / Extension pas implémenter:

Problèmes connus :

Héritage : On a les mêmes problèmes que dans butin puisqu’on le faisait en même temps.

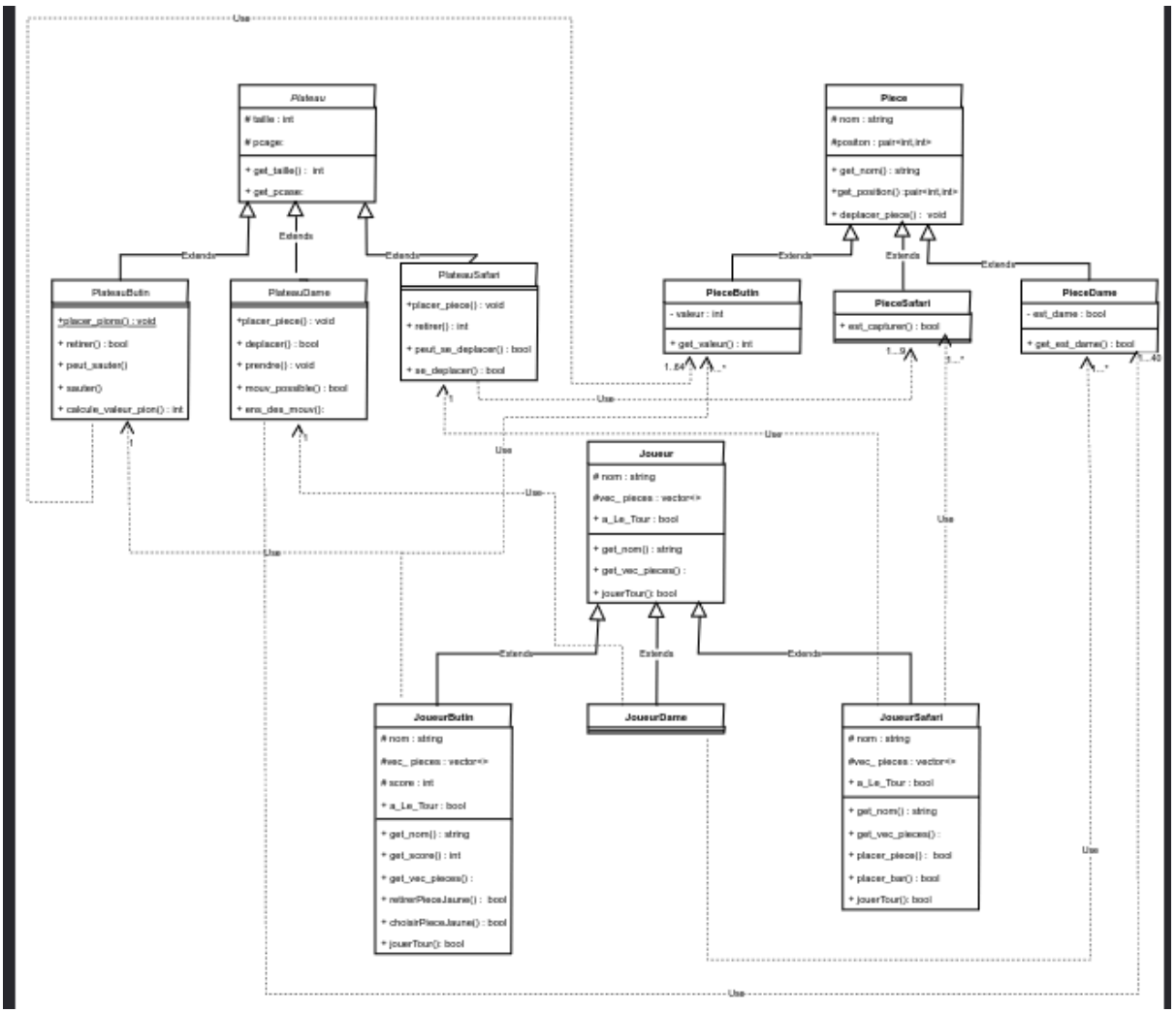
On a pas pu utiliser le plateau sous forme de case avec des barrières en forme rectangulaire car on arrivait pas à obtenir les coordonnées de certaines cases. Nous avons alors abandonné le plateau ci-dessous et réutilisé le plateau de butin. (voir README)



Voilà l'image qu'on avait au début mais par manque de temps on a pas pu résoudre les erreurs pour l'utiliser .

On a pas pu aussi bien implémenter la fonction jouertour de la classe joueur par manque de temps.

Références : Diagrammes UML.


Use
Plateau
taille : int
pcase:
+ get_taille() : int
+ get_pcase:
Piece
nom : string
#position : pair<int,int>
+ get_nom() : string
+get_position() :pair<int,int>
+ deplacer_piece() : void
Extends
PlateauSutin
+placer_pions() : void
+ retirer() : bool
+ peut_sauter()
+ sauter()
+ calcule_valeur_pion() : int
PlateauDame
+placer_piece() : void
+ deplacer() : bool
+ prendre() : void
+ mouv_possible() : bool
+ ens_des_mouv():
PlateauSafari
+placer_piece() : void
+ retirer() : int
+ peut_se_deplacer() : bool
+ se_deplacer() : bool
PieceSutin
- valeur : int
+ get_valeur() : int
PieceSafari
+ est_capturer() : bool
PieceDame
- est_dame : bool
+ get_est_dame() : bool
1..64
1..*
1..9
1..40
Joueur
nom : string
#vec_ pieces : vector<>
+ a_Le_Tour : bool
+ get_nom() : string
+ get_vec_pieces() :
+ jouerTour(): bool
JoueurSutin
nom : string
#vec_ pieces : vector<>
score : int
+ a_Le_Tour : bool
+ get_nom() : string
+ get_score() : int
+ get_vec_pieces() :
+ retirerPieceJaune() : bool
+ choisirPieceJaune() : bool
+ jouerTour(): bool
JoueurDame
JoueurSafari
nom : string
#vec_ pieces : vector<>
+ a_Le_Tour : bool
+ get_nom() : string
+ get_vec_pieces() :
+ placer_piece() : bool
+ placer_bar() : bool
+ jouerTour(): bool