Cahier des Charges

Jeu d'échec intelligent

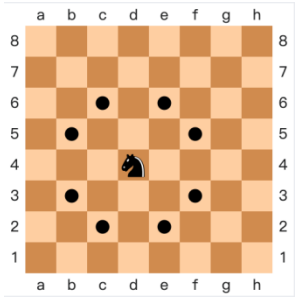
I - Présentation du projet

Dans le cadre de l'UE LIFAP4 Conception et développement d'application, nous avons décider de coder un jeu d'échec en C++. Cette idée du jeu d'échec est plutôt intéressante pour nous. En effet, ce projet de jeu peut être implémenté avec différents niveaux de développements de difficulté différentes : nous pouvons premièrement coder un simple jeu d'échec : des pièces sur un plateau se déplaçant en respectant des contraintes précises et permettant uniquement de jouer à deux joueurs, l'un contre l'autre. Il pourra ensuite nous être possible d'améliorer l'interface et les graphismes qui pourront permettre une expérience plus agréable du jeu, d'ajouter des sons, etc. Par la suite, nous essaierons d'instancier un programme de type d'intelligence artificielle basique. Il pourra jouer les coups légaux sans forcément être un bon joueur, se référer à un dictionnaire d'ouverture pour réaliser des débuts de parties convenables, etc.
Ce projet nous introduira donc possiblement le monde de l'IA et la bibliothèque SFML que nous trouvons intéressante à utiliser.

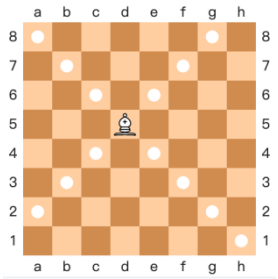
Nous reprendrons les règles de base du jeu d'échec classique. Ces règles seront rédigées et expliquée dans le README.md de notre fichier de projet, mais nous indiquerons ci-dessous les déplacements des pièces :

- Le pion avance de une case, ou deux cases si il est sur sa position de départ, toujours vers l'avant, et sur les deux case en diagonale gauche et droite, toujours vers l'avant.

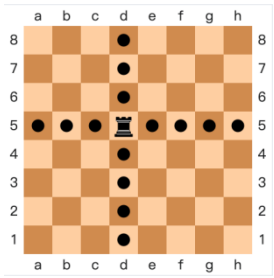
- Le cavalier effectue des déplacement en « L ». Trois cases sur l'axe des abscisses/ordonnées, peu importe le sens, et une case sur l'axe opposé, peu importe le sens.



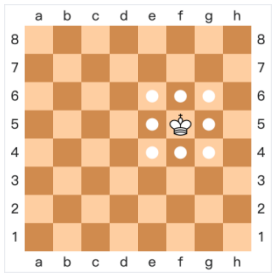
- Le fou se déplace sur les diagonales



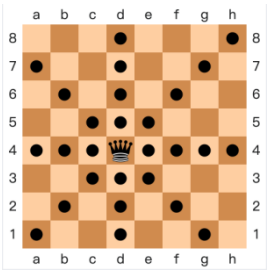
- La tour se déplace sur les axes X et Y.



- Le roi se déplace d'une case par une case

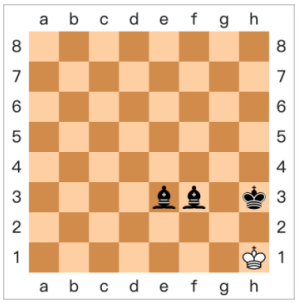


- La dame rassemble les déplacements du fou et de la tour.



- Hormis le cavalier, aucune pièce ne peu passer au dessus d'autres.

Il faut que le roi adverse soit attaqué (échec) et n'ait aucune issue pour se sauver ou se protéger d'une menace sans être attaqué (mat).



II - Description de la demande

Nous souhaitons avoir, à la fin de cette UE un jeu d'échec simple, intuitif et fonctionnel. L'objectif est qu'il soit le plus optimisé possible, aussi bien dans l'allocation et la délocalisation de la mémoire que dans le code.

Nous souhaiterions aussi rendre notre jeu le plus élégant et agréable que possible, et si les contraintes de temps et notre niveau de programmation nous le permettent, implémenter une intelligence artificielle permettant, avec un niveau relatif, de jouer seul face à la machine.

Pour réaliser ceci, il nous faudra coder :

- un menu principal permettant de :
- -s'inscrire, qui permettra au joueur inscrit d'avoir les résultats de ses anciennes parties, stockés dans un fichier texte par exemple.

  - choisir son adversaire (réel ou machine)
  - choisir sur mode de jeu (30, 10 min, blitz (5 min), 1 min, sans temps limite)
  - choisir le niveau de difficulté de l'IA (suivant ce que l'on à réalisé)
  - choisir le mode de sélection d'attribution de la couleur et donc du trait (attribution aléatoire, blanc ou noir)
  - un bouton « Commencer la partie » permettant de lancer la partie après avoir choisi les paramètres de celle-ci
  - Création de situations précises : choix de position pour chaque pièce à la guise l'utilisateur : mode situation

- Une fois la partie lancée, le plateau de jeu est initialisé et l'ensemble des pièces devra être affiché. Nous sélectionnerons une image aux bonnes dimension à la fois du plateau et des différentes pièces.

- Jouabilité via la souris ou le pavé tactile pour les ordinateurs portables.
- Modéliser des flèches sur l'échiquier pour aider le joueur à calculer ses coups et montrer ce qu'il souhaite faire. (un clique droit puis relier deux cases entre elles permettent de tracer une flèche entre ces cases).
- Utilisation de sons lorsque l'on déplace, mange des pièces, échec / mat du roi.
- Pour le déplacement des pièces : un clic sur une pièce mettra la pièce concernée valeur (surbrillance ou/et grossissement x1,05, etc) et les cases sur lesquelles elle peut se déplacer seront signalées par un indicateur.
- Différentes valeur d'importance pour les pièces : 1 pour le pion, 3 pour le fou et le cavalier, 5 pour la tour et 7 pour la dame.

Une version graphique grâce à SFML et une version texte dans le terminal seront réalisées. La version texte consistera à afficher notre application avec des caractère dans le terminal et de représenter par exemple les pièces avec des lettres.

Notre cahier des charges a un but prévisionnel, des modifications, ajouts, retrait de certaines fonctionnalités seront sans aucun doute effectuées. Ce cahier des charges sera donc modifié et mis à jour régulièrement en fonction de l'évolution des différentes contraintes.

III - Contraintes

Nous devrons rendre ce projet le 16 mai 2021, nous avons donc un temps limité mais malgré tout suffisant pour réaliser la plupart des fonctionnalités importantes.

Pour la réalisation de ce projet, nous utiliserons une méthode de production qui nous a été présentée en cours : la méthode AGILE.

Nous implémenterons cette application avec le langage C++.
Nous utiliserons la librairie SFML pour manipuler les images (plateau, pièces) ajouter des sons au mouvement des pièces.
Afin de rendre notre code clair et compréhensible par tous, nous utiliserons Doxygen qui permettra de générer une documentation simple et efficace. Cette documentation sera mise à jour régulièrement.

Avant de coder notre jeu, pour choisir une direction et savoir par quoi commencer, nous réaliserons un diagramme des classes UML qui nous permettra de connaître le rôle, le nombre et l'importance de chacune des classes de notre projet.

Nous insistons sur le fait que les exigences et les fonctionnalités peuvent varier en fonction de l'avancement du projet et du temps restant avant le rendu final.

IV - Déroulement du projet

Tâche 0 :
- Étude du projet et de ses besoins
- Réalisation d'un diagramme de Gantt pour les différentes tâches à effectuer et les dates définissant leur période de réalisation
- Réalisation d'un diagramme des classes UML.

Tâche 1 :
- Implémentation des classes Piece et Frame

Tâche 2 :
- Implémentation des héritages de la classe Piece (Pawn, King, Queen, etc.)

Tâche 3 :
- Utilisation de SFML pour intégrer des images et texture du plateau et des pièces

Tâche 4 :
- Implémentation des classes secondaires : MovePiece, Configuration, Game, Player, Display et DisplayText

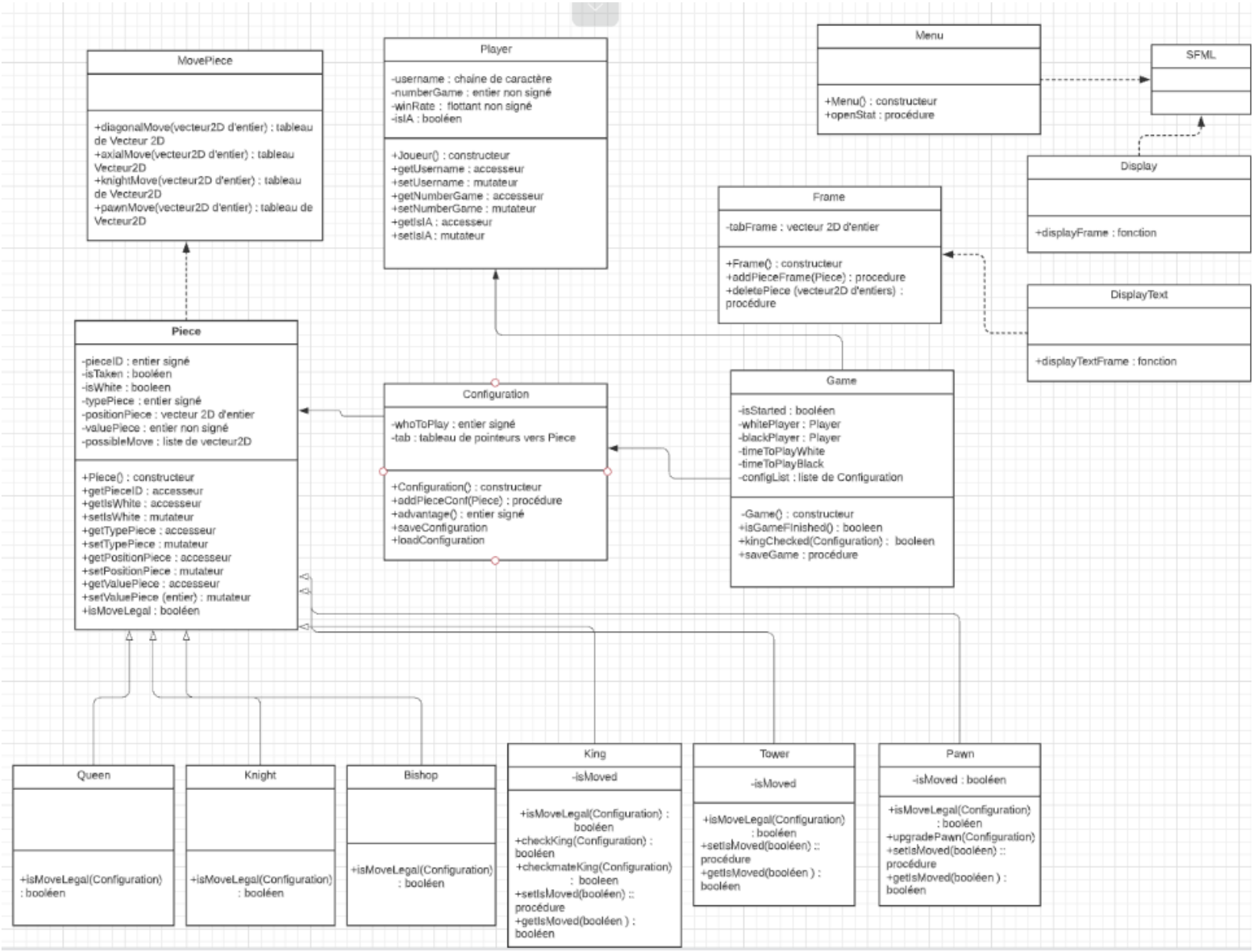
Tâche 5 :
- Utilisation de SFML pour la gestion des évènements souris et clavier

Tâche 6 :
- Implémentation de la classe Menu. Conception de l'interface graphique du jeu

Tâche 7 :
- Conception d'une IA basique pour permettre de "jouer" contre l'ordinateur
- Intégration de l'IA au jeu

Tâche 8 :
- Préparation de la soutenance

Semaines / Tâches	14/03	21/03	28/03	04/04	11/04	18/04	25/04	02/05	09/05
Tâche 0									
Tâche 1									
Tâche 2									
Tâche 3									
Tâche 4									
Tâche 5									
Tâche 6									
Tâche 7									
Tâche 8									


MovePiece
+diagonalMove(vecteur2D d'entier) : tableau de Vecteur 2D
+axialMove(vecteur2D d'entier) : tableau Vecteur2D
+knightMove(vecteur2D d'entier) : tableau de Vecteur2D
+pawnMove(vecteur2D d'entier) : tableau de Vecteur2D
Player
-username : chaine de caractère
-numberGame : entier non signé
-winRate : flottant non signé
-isIA : booléen
+Joueur() : constructeur
+getUsername : accesseur
+setUsername : mutateur
+getNumberGame : accesseur
+setNumberGame : mutateur
+getIsIA : accesseur
+setIsIA : mutateur
Menu
+Menu() : constructeur
+openStat : procédure
SFML
Display
+displayFrame : fonction
Frame
-tabFrame : vecteur 2D d'entier
+Frame() : constructeur
+addPieceFrame(Piece) : procedure
+deletePiece (vecteur2D d'entiers) : procédure
DisplayText
+displayTextFrame : fonction
Piece
-pieceID : entier signé
-isTaken : booléen
-isWhite : booleen
-typePiece : entier signé
-positionPiece : vecteur 2D d'entier
-valuePiece : entier non signé
-possibleMove : liste de vecteur2D
+Piece() : constructeur
+getPieceID : accesseur
+getIsWhite : accesseur
+setIsWhite : mutateur
+getTypePiece : accesseur
+setTypePiece : mutateur
+getPositionPiece : accesseur
+setPositionPiece : mutateur
+getValuePiece : accesseur
+setValuePiece (entier) : mutateur
+isMoveLegal : booléen
Configuration
-whoToPlay : entier signé
-tab : tableau de pointeurs vers Piece
+Configuration() : constructeur
+addPieceConf(Piece) : procédure
+advantage() : entier signé
+saveConfiguration
+loadConfiguration
Game
-isStarted : booléen
-whitePlayer : Player
-blackPlayer : Player
-timeToPlayWhite
-timeToPlayBlack
-configList : liste de Configuration
-Game() : constructeur
+isGameFinished() : booleen
+kingChecked(Configuration) : booleen
+saveGame : procédure
Queen
+isMoveLegal(Configuration) : booléen
Knight
+isMoveLegal(Configuration) : booléen
Bishop
+isMoveLegal(Configuration) : booléen
King
-isMoved
+isMoveLegal(Configuration) : booléen
+checkKing(Configuration) : booléen
+checkmateKing(Configuration) : booleen
+setIsMoved(booléen) :: procédure
+getIsMoved(booléen) : booléen
Tower
-isMoved
+isMoveLegal(Configuration) : booléen
+setIsMoved(booléen) :: procédure
+getIsMoved(booléen) : booléen
Pawn
-isMoved : booléen
+isMoveLegal(Configuration) : booléen
+upgradePawn(Configuration)
+setIsMoved(booléen) :: procédure
+getIsMoved(booléen) : booléen