Dev4-Projet

Rapport2-remise console

FOUEFACK ATCHEPONG Vivace (54490)

NGOUMENI NKEUDJEU Jeanny (55015)

Table des matières

Chapitre I : Analyse et modélisation

Introduction

Stratego est un jeu de stratégie ou deux joueurs (un joueur avec les pièces rouges et un autre avec les pièces bleues) s'affrontent sur un plateau. Il nous est demandé de concevoir et d'implémenter ce jeu en c++. Pour réaliser ce projet, nous allons définir l'environnement de développement, les outils utilisés ainsi que la modélisation des classes métiers.

I. Environnement de développement et outils utilisés

Pour la réalisation de ce projet, nous allons implémenter l'architecture MVC (Model View Controller) et utiliser :

- L'IDE (Integrated Development Environment) Qt Creator.
- StarUML : est un logiciel de modélisation UML (unified modeling language) qui a permis de mettre sur pieds le diagramme de classe.
- GitLab for esi : outil qui nous faciliterait la collaboration à distance et nous permettrait de gérer nos différentes versions.

II. Modélisation UML

Cette section va traiter la modélisation UML des classes métiers qui vont constituer le cœur du projet. Le diagramme de classes est disponible à la fin de ce document.

a. Classe Player

La classe Player désigne un joueur. Elle est caractérisée par :

- Un attribut Pièce de type vecteur qui va contenir l'ensemble des pièces encore en jeu d'un joueur.
- Un attribut LostPieces de type vecteur qui va contenir l'ensemble des pièces éjectées d'un joueur.
- Un attribut isMyTurn de type booléen qui indique si c'est au tour d'un joueur de joue ou pas.
- Un attribut color qui indique le couleur des pièces d'un joueur.
- Une méthode IsMyPiece (Position position) ->Boolean : cette méthode permet de vérifiée si une des pièces du joueur se trouve à une position reçue en paramètre.

- Une méthode RemovePiece (Position position) ->void : soustrait la pièce située à la position reçue en paramètre de la liste des pièces d'un joueur.
- Une méthode addLostPiece (Piece pièce) ->void : ajout la pièce reçue en paramètre dans le vecteur des pièces éjectées.
- Une méthode emptyLastOccupation (Piece pièce) ->void : permet assurer la gestion des allés et retours successifs d'une pièce. Lorsqu'une nouvelle pièce est déplacée, les anciennes occupations de toutes les autres pièces sont détruites et l'attention est portée uniquement sur la pièce en cours.

b. Classe Piece

Elle désigne une pièce du joueur. Cette pièce possède les méthodes et attributs suivants :

- Un attribut position qui représente la position d'une pièce sur le tableau.
- Un attribut symbole qui désigne le rang de la pièce.
- Un attribut dévoilé de type boolean qui permet de savoir si une pièce est dévoilée ou pas. il est mis à vrai lorsqu'un joueur attaque avec une pièce ou est attaqué par l'adversaire. Par défaut, cet attribut est à faux.
- Un attribut lastOccupations représentant un vecteur d'anciens positions qui permettrait de retenir les différentes positions d'une pièce ce qui contribuerait à la gestion de la contrainte de non-allers-retours (entre deux positions) consécutifs de plus de 3 fois sur une pièce.
- Une Méthode move (Position nextPosition) ->void : déplace une pièce de sa position actuelle vers la position reçue en paramètre.
- Une méthode roundTripCheck () ->boolean : vérifie si la pièce a déjà effectué trois allers-retours entre 2 positions, retourne vrai si c'est le cas et faux dans le cas contraire.
- Une méthode to_stringHidden () -> string : permet de masquer le symbole des pièces non révélées d'un joueur qui n'a pas la main.
- Une méthode addLastOccupation (Position position) -> void : permet d'ajouter l'ancienne position d'une pièce dans son vecteur d'ancienne position.

- Une méthode emptyLastOccup () -> void : vide le vecteur d'anciennes positions d'une pièce.

c. Classe Position

Permet de situer une pièce dans l'espace à deux dimensions du plateau de jeu, par conséquence elle est constituée de :

- Un attribut « x » qui représente l'axe x.
- Un attribut « y » qui représente l'axe y.

Cette classe serait particulièrement utile puisse qu'elle nous permettrait d'assurer la gestion des déplacements.

d. Classe Game

Il s'agit probablement de la classe métier la plus importante puisqu'elle permettrait une communication plus aisée entre le contrôleur et le modèle. On y trouve les attributs et méthodes suivants :

- Un attribut playerOne représentant le joueur rouge.
- Un attribut playerTwo représentant le joueur rouge.
- Une méthode move (Position currentPosition, Position nextPosition, Player player) ->void : déplace la pièce d'un joueur reçu en paramètre de sa position actuelle(currentPosition) vers la position suivante (nextPosition).
- Une méthode gameIsOver () ->boolean : vérifie l'état du jeu. si le jeu est dans un état terminé (le drapeau d'un joueur est capturé ou il n'est plus possible pour un joueur de déplacé un de ses pièces), la méthode retourne vrai ou faux dans le cas contraire.
- Une méthode getWinner () ->player : permet de déterminer qui est le vainqueur à la fin d'une partie.
- Une méthode inWater (Position destination) ->bool : vérifie si l'on se dirige vers l'eau. Si la position de destination d'une pièce contient de l'eau, la méthode retourne vrai ou faux dans le cas contraire.
- Une méthode isGoodMove (Position currentPosition, Position nextPosition) ->bool : vérifie si le déplacement entre la position actuelle d'une pièce et une position de destination est possible.

- Une méthode nextCaseMove (Position currentPosition, Position nextPosition, Player & player, Player & opponent) -> void : fait véritablement le déplacement lorsque toutes les conditions sont réunies.
- Une méthode canMinerMove (Position currentPosition, Position nextPosition, Player & player, Player & opponent) ->bool : retourne vraie si toutes les conditions sont réunies pour le déplacement d'un démineur donc la position actuelle et la position de destination sont prises en paramètre, ou faux dans le cas contraire.
- Une méthode canStillMove (Player player) -> bool : vérifie si le joueur reçu en paramètre peut encore déplacer une de ses pièces.

Conclusion

Bien souvent en informatique, la phase d'analyse est précédée de celle de l'implémentation. Cette phase d'analyse est une nécessité pour le bon déroulement du projet étant donné qu'elle permet l'étude des données et des traitements à effectuer lors de l'implémentation.

Chapitre II : Implémentation console

Introduction

Ici, nous allons traiter l'implémentation des classes, des méthodes du jeu en C++ ainsi que la mise en place d'une interface console. Dans cette partie, seul les algorithmes non triviaux et d'autres points pertinent seront abordés.

a. Modifications apportées à la remise 1

Certaines modifications ont été apportés à la remise précédente conformément aux remarques et suggestions fait lors de la présentation de la première remise. Les plus importantes sont :

- La classe Board a été supprimée parce qu'elle n'apportait aucune plus-value au projet si ce n'est éventuellement de conceptualiser à proprement parler le plateau. En l'occurrence, les interactions avec une éventuelle Classe auraient été limitées à la comparaison des positions du board et des pièces, d'où le choix de directement utilisée les positions des pièces.
- L'ajout d'un vecteur de pièce dans la classe player afin de stocker les pièces éjectées.
- L'ajout des méthodes inWater, canMinerMove, nextCaseMove et isGoodMove afin de désengorger la méthode move de la classe Game.

b. Classe contrôleur

cette partie gère la logique de la prise de décisions. C'est en quelque sorte l'intermédiaire entre le modèle et la vue. Il va demander à la vue d'interagir avec les utilisateurs puis les données fournir sont envoyées au modèle qui va faire les calculs nécessaires. Le contrôleur analyser, prendre des décisions et renvoyer les données nécessaires pour l'affichage à la vue. Cette classe possède une seule méthode :

- la méthode Start () -> void : elle permet de lancer le jeu.

c. Classe view

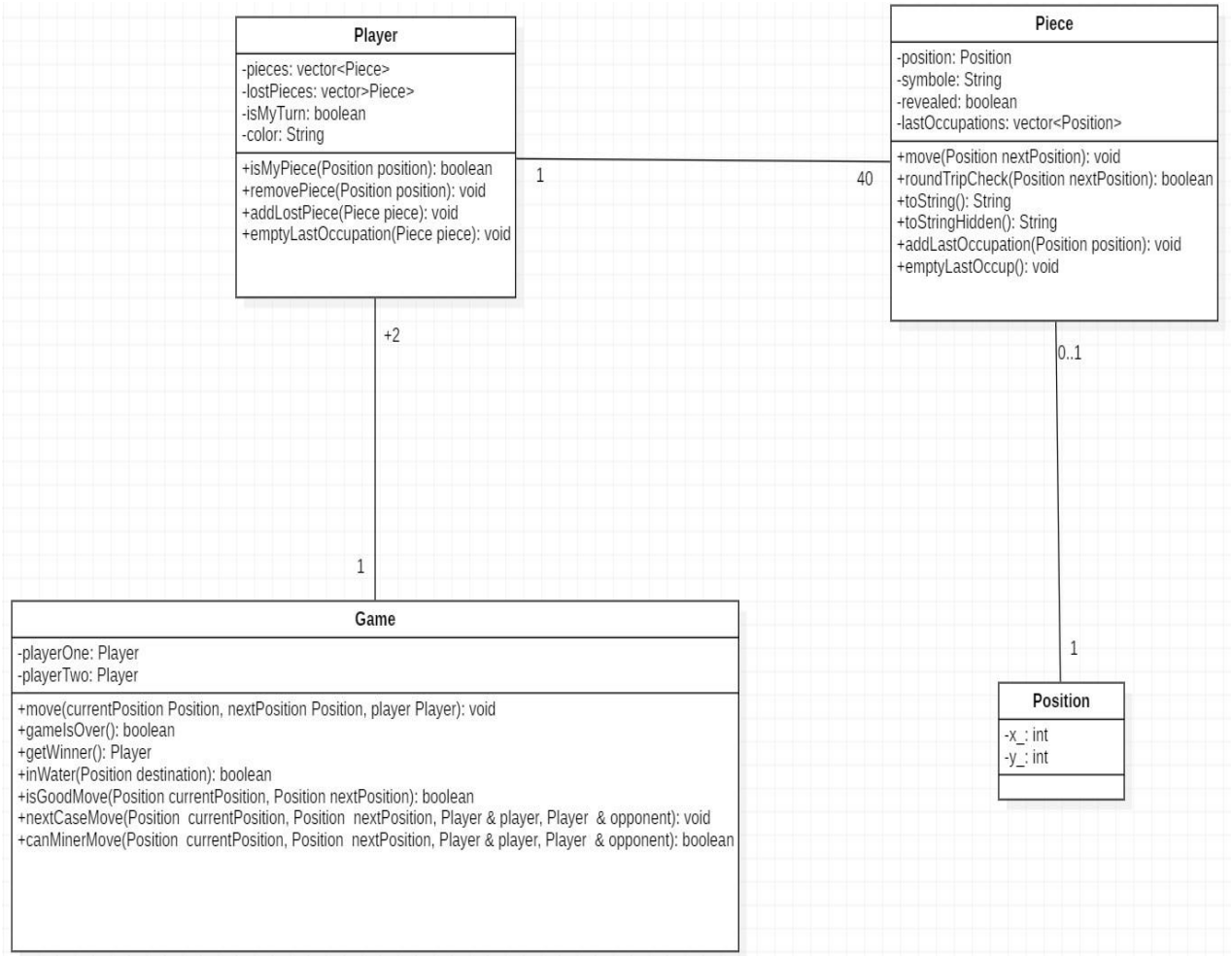
Cette classe traite tout ce qui est affichage console. Elle contient la logique nécessaire pour afficher les données provenant du modèle ainsi que les différentes interactions avec les utilisateurs. Elle contient principalement les méthodes suivantes :

- Une méthode displayScore (Game game) ->void : permet d'afficher le score de chacun des joueurs du game.
- Une méthode displayWinner (Game game) -> : permet d'indiquer qui est le vainqueur lorsque le jeu est terminé.
- Une méthode displayBoard (Game & game) ->void : c'est l'une des méthodes les plus importante du jeu étant donné que c'est elle qui contient la logique nécessaire pour l'affichage des données du modèle à travers le game reçu en paramètre.
- Une méthode askCurrentPosition () -> Position : demande la position de la pièce qu'un joueur souhaite déplacer et retourne cette position.
- Une méthode askNextPosition () -> Position : demande la position de destination d'une pièce et la retourne.
- Une méthode askPosition () -> Position : demande la position de la pièce qu'un joueur souhaite déplacer et retourne cette position s'il existe une pièce à cette position sinon l'opération est répétée jusqu'à ce que la position soit fournie.
- Une méthode symbolContaint (string s) ->bool : vérifier si un symbole reçu en paramètre fait partie des symboles valables du jeu.
- Une méthode symboleList () -> map<string, int > : la méthode fournir la liste des symboles et leurs tailles maximales. la clé serait le symbole et la valeur serait la taille.
- Une méthode bluePiecePositioning () -> vector<Piece> : lit la stratégie du joueur bleu dans un fichier et retourne un vecteur.
- Une méthode redPiecePositioning () -> vector<Piece> : lit la stratégie du joueur rouge dans un fichier et retourne un vecteur.
- Une méthode askPositioningStyle () ->int : demande le type de stratégie à adoptée.

Conclusion

Ce chapitre était consacré à l'implémentions des actions définies lors de la phase d'analyse, à l'implémentation des actions de la vue et l'implémentation d'un contrôleur pour coordonner l'ensemble de ces actions. Nous obtenons ainsi une version console fonctionnel qui nous servira de base pour la version graphique.

Annexe


Player
-pieces: vector<Piece>
-lostPieces: vector>Piece>
-isMyTurn: boolean
-color: String
+isMyPiece(Position position): boolean
+removePiece(Position position): void
+addLostPiece(Piece piece): void
+emptyLastOccupation(Piece piece): void
1
40
Piece
-position: Position
-symbole: String
-revealed: boolean
-lastOccupations: vector<Position>
+move(Position nextPosition): void
+roundTripCheck(Position nextPosition): boolean
+toString(): String
+toStringHidden(): String
+addLastOccupation(Position position): void
+emptyLastOccup(): void
+2
1
0..1
1
Game
-playerOne: Player
-playerTwo: Player
+move(currentPosition Position, nextPosition Position, player Player): void
+gameIsOver(): boolean
+getWinner(): Player
+inWater(Position destination): boolean
+isGoodMove(Position currentPosition, Position nextPosition): boolean
+nextCaseMove(Position currentPosition, Position nextPosition, Player & player, Player & opponent): void
+canMinerMove(Position currentPosition, Position nextPosition, Player & player, Player & opponent): boolean
Position
-x_: int
-y_: int