Rapport projet 22 Pommes

Table des matières

introduction:

Pour réaliser ce projet, j'ai effectué beaucoup de changement au fil de temps.

Au début, je travaillais avec une structure de cmake mais j'avais beaucoup de problème pour créer des sous dossiers et j'ai donc changé de structure et j'ai choisi la structure qmake car plus simple à utiliser

Le pattern choisi est le Modèle Vue Contrôleur (MVC). :

- La partie métier va contenir les éléments du jeu et la logique du jeu.
- La partie vue servira d'affichage en console.
- La partie contrôleur va faire le lien entre la vue et le modèle

Ps : j'ai fait une partie console car j'ai vu que dans les exigences du professeur c'était écrit qu'il fallait 3 dossiers dans le projet.

Métier:

Les fonctions, les classes et les attributs se trouve dans le diagramme UML[1] et dans les headers du sous projet métier.

Classe Tile :

La tuile sera la classe mère de la classe Apple et de la classe Picker qui sera totalement

Abstraire dont les méthodes devra être ''override'' à travers les classes enfants.

Cette classe correspond également à la case du plateau de jeu.

Attribut :

Color color : La tuile aura une couleur soit rouge soit verte ou none.

Bool isApple : c'est un booléen qui va déterminer si la tuile est de type Apple(pomme)

Bool isPicker : c'est un booléen qui va déterminer si la tuile est de type Picker(cueilleur)

String name : nom de la tuile

Int value : valeur de la tuile

Bool faceUp : booléen pour déterminer si la tuile est face visible.

Méthode :

Getter des attributs.

flipfaceUp() : la tuile sera face visible.

Constructeur

Le constructeur va construire la tuile et la faceUp sera initialisé à false au début de la partie.

Rem : Au début, il n'y avait pas d'attribut isPicker et isApple mais un attribut isMoveable qui était à ''true'' pour le cueilleur et ''false'' pour les pommes mais avec du recul je me

suis dit que si je voulais ajouter un nouveau type de tuile ce serait plus difficile à implémenter par la suite j'ai donc opté à mettre des « détermination de type d'objet

(eg. bool isPicker)».

Classe Apple :

La classe va contenir les méthodes virtuelles de la classe Tile et va être implémenté dans le header de la classe Apple.

Classe qui correspond à l'élément d'une pomme.

Constructeur

Apple::Apple(int value, Color color) : Tile(color, true, false ,value){this.name =Apple::GetName()}

L'attribut isApple sera à true

L'attribut isPicker sera à false car ce n'est pas un cueilleur.

Le nom va varier en fonction de la couleur de la pomme soit ''RA '' pour RED APPLE ou soit ''GA'' pour GREEN APPLE.

Classe Picker :

La classe va contenir les méthodes virtuelles de la classe Tile et va être implémenté dans le header de la classe Picker.

Classe qui correspond à l'élément d'un cueilleur.

Constructeur

Picker::Picker(): Tile(Color::NONE, false, true ,0){ this->name = Picker::getName();}

L'attribut isApple sera à « false » car ce n'est pas une pomme

L'attribut isPicker sera à « true »

L'attribut value sera 0 car il n'a pas de valeur

L'attribut color sera NONE.

Le nom sera ''PI''.

Classe Color :

La classe color sera en réalité une énumération qui sera NONE GREEN ROUGE.

Classe Bag :

La classe Bag sera le panier d'un joueur dans le jeu qui sera soit rouge ou soit vert

Attribut :

Color color : couleur du panier

int getCapacity : capacité du panier

Méthode :

Getter des attributs

void setCapacity(int value) : mettre dans le panier la valeur passé en paramètre.

void setCapacityAt0() : remettre le panier à 0 si les joueurs veulent rejouer une partie.

bool isFull : vérifie si la capacité est rempli en d'autre terme si la capacité est égale à 11

Constructeur :

Bag::Bag(Color color) : color(color), capacity(0){}

initialise la capacité à 0 et initialise la couleur du panier avec le paramètre color

Classe Player :

La classe Player correspond à un joueur.

Attribut :

Int id : identifiant du joueur

std::vector<std::shared_ptr<Bag>> bags : Les paniers du joueur sera un vecteur de shared_ptr de Bag j'ai choisi un pointeur intelligent pour qu'il libère automatiquement la mémoire.

Int score : score du joueur

Méthode :

Getter et setter

Constructeur :

créer les paniers du joueur et initialise son score à 0.

Classe Position :

La classe position va nous permettre d'indiquer une position sur le plateau de jeu.

Attribut :

int row : la ligne du plateau

int col : la colonne du plateau

Méthode :

Getter des attributs

Constructeur :

Position::Position(int x,int y) : row(x),col(y) {}

La row sera initialisé avec le x passé en paramètre .

La col sera initialisé avec le y passé en paramètre.

Classe Board :

La classe Board sera composé d'un vector 2D contenant des shared_ptr de Tile

J'ai choisi ce conteneur car il est dynamique donc sa taille va être redimensionner automatiquement et c'est le plus facile à utiliser. Elle contient des shared_ptr de tile qui vont libérer automatiquement la mémoire.

Attributs :

const int height : longueur du plateau

const int width : largeur du plateau

Méthodes :

Getter des attributs height et width

bool isInside(Position&pos) : retourne vrai si la position est compris dans les bornes du plateau.

bool canBeput(Position&pos) : vérifie si le cueilleur peut être placé à la position donnée.

Bool checkPickerMov(Position&pos) : vérifie les déplacements de la tuile cueilleur sachant que le cueilleur peut juste se déplacer sur sa ligne ou sur sa colonne.

Position getPickerPosition() : retourne la position de la tuile du cueilleur se trouvant dans le Plateau sinon (-1,-1).

Rem : Un attribut Position dans la classe tuile aurait été mieux pour avoir la position du cueilleur sans parcourir tout le plateau et donc avoir une complexité plus grande $O(n^2)$.

Shared_ptr<Tile>& getTile(Position& pos) : retourne la référence d'une tuile a la position passée en paramètre.

void put(std::shared_ptr<Tile>& tile, Position& pos) : mettre la tuile à la position donné

Constructeur :

Board::Board(int longeur, int largeur) : height(longeur), width(largeur), tiles(height, std::vector<std::shared_ptr<Tile>>(width, nullptr)){}

Va créer un plateau de tuile vide de taille height x width.

Classe Deck :

Cette classe représente les cartes du jeu. Va contenir 12 pommes rouges ,12 pommes vertes et une carte cueilleur.

Attributs :

vector<shared_ptr<Tile>> cards : un vecteur de shared pointeur de tuiles .

Méthodes :

Shuffle() : mélanger les cartes a l'aide de la méthode std ::shuffle se trouvant dans la librairie random

vector<std::shared_ptr<Tile>> getCardsFaceDown() : retourne les cartes face caché.

vector<std::shared_ptr<Tile>> getCardsFaceUp () : retourne les cartes face visible.

int getNbCards() : retourne le nombre de cartes.

Constructeur :

Deck::Deck(bool gameMode){}

Va construire un deck en fonction du paramètre gameMode.

Si gameMode = 1 alors il va créer un deck en commençant par les pommes puis va mélanger les cartes ensuite il va placer le cueilleur à la fin du vecteur.

Sinon il va directement tout mettre puis mélanger les cartes.

Classe State :

La classe state est une énumération contenant l'état du jeu

NOT_STARTED,

GAME_MODE_NORMAL,

GAME_MODE_ALLONGE,

PLACE_PICKER,

TURN_END,

GAME_OVER

Lors de la création du jeu, il sera à l'état NOT_STARTED puis en fonction du mode de jeu il va créer le plateau soit en mode normal ou soit en mode allongé puis il va mettre l'état à PLACE_PICKER puis grâce à l'action putPicker il va mettre l'état à TURN_END si ce n'est pas GAME_OVER sinon il passe à l'état de GAME_OVER.

Classe Model :

Cette classe représente la logique. Elle va reprendre les méthodes des classes précédemment citées et vont être réadapté.

Attribut :

Players: un vecteur de pointeur shared de Player

int curentPlayer : joueur courant

Deck deck : cartes du jeu

State state : état de jeu

State gameMode : état de mode de jeu qui peut être omis car on ne l'utilise pas vraiment

pickerTile : pointeur shared de la tuile cueilleur

Méthode :

Getter et setter du model.

Int getWinner() : retourne le numéro du vainqueur si le joueur courant atteint 11 pommes rouge dans son panier rouge et 11 pommes vertes dans son panier vert il a gagné. Si le joueur courant dépasse 11 pommes dans un de ses paniers le vainqueur sera le joueur suivant sinon retourne -1 en cas d'égalité

void start(bool mode) : instancie le jeu selon le mode de jeu si le mode = 0 alors il va créer un plateau de 5 x 5 puis il va placer les cartes dans le plateau et va déterminer le joueur courant au hasard grâce à la méthode std::shuffle() de la librairie <random>.

Si le mode =1 alors il va faire la même chose sauf qu'il va créer un plateau de 3x8 mais en réalité ça sera un plateau de 3x9 dont la dernière colonne sera réservée au cueilleur(pickerTile) afin de déterminer le commencement du jeu.

Apres l'appel à cette méthode il va placer l'état de jeu à PLACE_PICKER.

void PutPicker(Position &pos) :placer le cueilleur à la position donnée en paramètre et va placer la pomme dans son sac et va « set » la capacité du sac grâce à la méthode addAppleInBag.L'ancienne position sera vide. Ensuite, Il va passer à l'état suivant mais pour se faire il va regarder si le jeu est fini ou pas si oui il va aller à l'état GAME_OVER sinon à TURN_END et ça sera au joueur suivant de jouer.

void nextPlayer() : cette méthode va mettre a jour le joueur courant pour qu'il passe au joueur suivant.

Void AddAppleInBag(shared_ptr<Tile>& tile): met une pomme dans un sac à sa couleur correspondante (e.g une pomme rouge dans le sac rouge).

Void resetBagCapaciry() : mettre la capacité des sacs à 0 lorsque le jeu sera terminé.

Constructeur :

```
Model::Model()
{
    state = State::NOT_STARTED;
    players.push_back(std::make_shared<Player>(0));
    players.push_back(std::make_shared<Player>(1));
```

```
  this->curentPlayer =0;
}
```

Initialise l'état de jeu à NOT_STARTED.

Le joueur courant sera à 0.

Mettre les joueurs dans le vecteur.

Classe View :

La partie vue va afficher les informations des joueurs, demander une position, demander un mode de jeu au joueur courant et afficher le plateau du jeu.

Classe Controller :

Le contrôleur va mettre en lien la vue et le model

Attribut :

Model & model : référence de la classe Model

View& view : référence de la classe View

Rem : Choix de référence pour ne pas créer un copie de l'objet.

Méthode :

void play(): commence la partie avec toutes les information du jeu et la logique du jeu. Utilisation d'un while(true) pour que le programme continue de tourner jusqu'à GAME_OVER et va demander au joueur s'il veut faire une autre partie s'il dit non le programme s'arrête grâce au return exit (0).

Constructeur :

```
Controller::Controller(Model &model , View &view) : model(model) ,view(view){}
```

Model sera initialisé avec le model passé en paramètre et la view sera également initialiser avec la view passée en paramètre.

Gui

Dans ce sous dossier, il y a les différentes classes pour l'implémentation de l'interface graphique du jeu « 22 Pommes ».

Classe Model :

La classe Model a subit quelques modifications au niveau du constructeur.

```
Model::Model(int scorePlayer1,int scorePlayer2)
1{
2  players.push_back(std::make_shared<Player>(1));
 3 players.at(0)->setScore(scorePlayer1);
```

```
4 players.push_back(std::make_shared<Player>(2));
 5 players.at(1)->setScore(scorePlayer2);
  6 this->curentPlayer =0;
7}
```

J'ai mis un setScore à la ligne 3 et 5 afin de sauvegarder le score du joueur en fin de jeu dans la classe PlayAgain et qui va également transmettre les données à la classe GameModeDiag.

Rem : Au début, de l'implémentation, j'ai constaté que le score était toujours remis à 0 lorsque je voulais rejouer une partie donc j'ai soit voulu créer une classe qui va stocker les statistique du joueur ou soit le faire ainsi. J'ai choisi la deuxième option car c'est beaucoup plus rapide.

Modification de certaines méthode qui faisaient planter l'exécutable à cause de la gestion des exceptions donc j'ai remplacé ''throw'' par des return false.

Classe State :

Modification de la classe énumération State.

Suppression du NOT_STARTED ,MODE_NORMAL,MODE_ALLONGE

Classe ControllerGUi :

Dans la partie Gui, la classe Controller a été remplacé par ControllerGui.

Elle hérite de QObject.

Cette classe reste le contrôleur du jeu, responsable à la logique du jeu

Elle va envoyer des signaux à la MainWindow pour gérer la logique du jeu.

Attribut :

Model*model : pointeur brut de la classe Model

Signaux

void error() :Lorsqu'une erreur aura lieu il va émettre un signal a la mainWindow.

void signalNextPlayer() :Emet un signal lorsque l'état sera TURN_END

void signalGameOver() :Emet un signal si l'état est à GAME_OVER

Slot :

*void play(Position pos) :*similaire à la version Controller de la partie métier mais il n'y a plus d'état NOT_STARTED et il n'y a plus d'affichage. Il va juste émettre des signaux pour l'envoyer à la mainWindow lorsqu'un évènement survient.

Lorsqu'un utilisateur va appuyer sur un bouton à une position donné en paramètre, il va vérifier si le cueilleur peut être placer si oui il va mettre le cueilleur à l'endroit désigné puis il va vérifier l'état du signal si c'est GAME_OVER, il va émettre signalGameOver () et il va

exécuter le slot displayWinner se trouvant dans la MainWindow sinon il va émettre signalNextPlayer et va exécuter le slot displayPlayerWithBag se trouvant dans la même classe. Si le cueilleur ne peut pas être placé il va émettre un signal d'erreur.

Constructeur :

```
ControllerGui::ControllerGui(Model*model,QObject *parent) :
    QObject(parent),
    ui(new Ui::ControllerGui),model(model){}
```

Initialise le model avec le model passé en paramètre.

Classe MainWindow :

La fenêtre principale du mode de jeu sélectionné. Il y aura tous les éléments du jeu visibles comme le plateau de jeu, les informations des joueurs, un label indiquant le premier joueur.

Attribut :

Voir header

Méthodes :

voir header

Slot :

void On_helpbutton() : lorsque l'utilisateur va appuyer sur le bouton help un signal sera émis et va ouvrir la fenêtre d'aide.

Void displayError() : Lorsque le contrôleur va signaler une erreur de placement du cueilleur il va ouvrir une fenêtre de dialogue de type de critical (QMessageBox).

Void displayWinner() : Lorsque le contrôleur va émettre signalGameOver , il affiche le vainqueur de la partie ,son score et va demander au joueur s'il veut rejouer ou quitter la partie en ouvrant la fenêtre askPlayAgain.

void displayPlayerWithBag() : lorsque le contrôleur va émettre un signalNextPlayer il va mettre à jour les informations des joueurs.

Constructeur :

Le constructeur va créer les données des joueurs, créer le plateau de boutons, mettre la mise en forme de la fenêtre comme le background, titre du mode de jeu et initialisé les éléments nécessaires a la création d'une interface graphique comme les QLabel,QPushButtons etc.

Classe GameModeDial :

La classe gameModeDial est la fenêtre d'accueil permettant à l'utilisateur de choisir son mode de jeu, de lire les règles du jeu ou de quitter l'application. Pour la réalisation de cette classe j'ai créé des boutons à partir du formulaire Ui.

Attribut :

Int scorePlayer1 : score du joueur 1

Int scorePlayer2 : score du joueur 2

Slot :

void action_quit() : quitte l'application si l'utilisateur appuie sur le bouton quitter

void action_normalClicked() : ouvre la mainWindow d'un jeu en mode normal s'il appuie sur le bouton version normal.

void action_versionClicked() : ouvre la mainWindow d'un jeu mode allongé s'il appuie sur le bouton version allonge.

void action_howToPlay() : ouvre la fenêtre d'aide si l'utilisateur appuie sur le bouton aide

Constructeur :

Initialise le score du joueur1 avec le scorePlayer1 en paramètre

Initialise le score du joueur2 avec le scorePlayer2 en paramètre

Mise en forme de la fenêtre comme setWindowTitle qui va setup le titre de la fenêtre,

setFixedSize pour définir la taille de la fenêtre etc.

Connection entre les signaux et les slots pour rendre les boutons fonctionnels

Classe HowToPlay:

Cette classe va juste contenir les règles du jeu et pour la modélisation de cette classe, j'ai dû créer fichier Word ensuite j'ai écrit les règles puis converti en format png. Ensuite, je l'ai mis comme background.

Classe PlayAgain :

La classe PlayAgain est la fenêtre qui va être afficher en fin de partie, Elle va demander a l'utilisateur s'il veut rejouer une partie ou quitter s'il veut fermer l'application. S'il veut rejouer il va ouvrir la classe GameModeDia pour qu'il puisse sélectionner un mode de jeu

Attribut :

Int scorePlayer1 : score du joueur 1

Int scorePlayer2 : score du joueur 2

Slot :

void action_quit() : quitte l'application et ferme toutes les fenêtres si l'utilisateur appuie sur le bouton quitte

void on_playAgain() : ouvre la fenêtre d'accueil si il appuie sur le bouton rejouer

Constructeur :

Initialise le score du joueur1 avec le scorePlayer1 en paramètre

Initialise le score du joueur2 avec le scorePlayer2 en paramètre

Mise en forme de la fenêtre comme setWindowTitle qui va setup le titre de la fenêtre,

setFixedSize pour définir la taille de la fenêtre etc.

Connection entre les signaux et les slots pour rendre les boutons fonctionnels

rem : c'est grâce à cette classe que le score des joueurs sont sauvegarder à la fin de la partie.

Conclusion

La réalisation de ce projet m'a pris un temps fou mais ce fut une bonne expérience.

Ce projet m'a offert une occasion d'apprentissage inestimable rien qu'avec les recherches et les vidéos que j'ai dû regarder sur YouTube afin de comprendre les outils d'une interface graphique Qt.

Je suis fier du travail que j'ai fourni.