Fixelles Alban 57343
Sabouri Abdessamad 53960

Baba Is You

Rapport

1.Table des matières

Table des matières

Dev4 Année académique
2022-2023

2.Partie Métier

a) Structure

Parmi les trois choix de structure possibles, nous avons décidé de choisir la structure QtCreator pour plusieurs raisons.

Simplicité : La syntaxe de la structure .pro est plus simple et plus intuitive que celle de CMake.

L'utilisation dans QtCreator : Comme on utilise Qt Creator comme IDE, il est plus facile de configurer et d'utiliser une structure .pro, car l'IDE est conçu pour travailler avec ce format.

Facilement personnalisable : La structure .pro est facilement personnalisable, ce qui permet de l'adapter à nos besoins spécifiques. Nous pouvons ajouter des options de compilation personnalisées, définir des variables d'environnement et des cibles de construction personnalisées.

b) Description générale

Nous allons implémenter notre projet en MVC avec le pattern observater observable.

Le « Controller » fait le lien entre la « View » et le « Model ».

La « View » servira à afficher en console ainsi que l'affichage final.

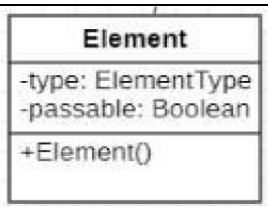
Le « Model » reprend toutes les fonctions qu'il faudra implémenter.

Le design pattern Observateur est un modèle de programmation qui permet à plusieurs objets de recevoir une notification lorsqu'un événement se produit dans un autre objet. Les objets observateurs s'inscrivent auprès du sujet pour recevoir les notifications, formant une relation un à plusieurs. Cela permet de synchroniser les objets et de minimiser les dépendances entre les parties de l'application.

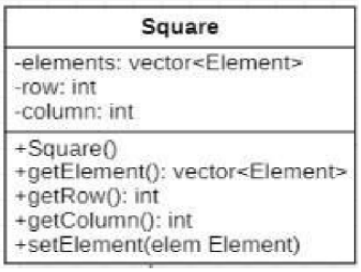
c) Description détaillée des fonctions et attributs

La documentation se trouve dans le diagramme UML ainsi que dans les headers, ici se trouve l'explication de nos choix[1].

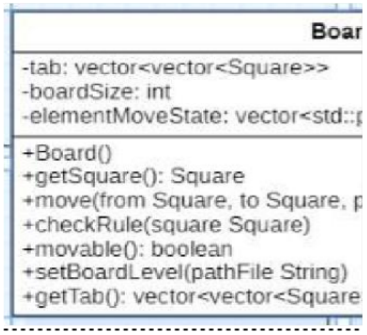
Fixelles Alban 57343
Sabouri Abdessamad 53960



Nous avons choisi que chaque élément du jeu serait un pion sur une case et qu'il serait mobile ou non en fonction des règles qui leur sont appliquées.



Ce sont les cases du plateau. Nous aurons besoins de getters.

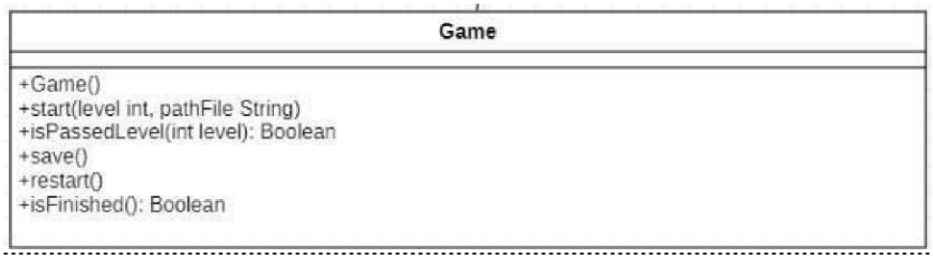


Le board va contenir toutes les cases du jeu.

La méthode move() servira à bouger les pions de case en case.

La méthode checkRule() sert à vérifier s'il y a une règle verticale ou horizontale.

La méthode movable() sert à savoir si « qqch » est mobile.



La méthode start() sert à lancer les différents niveaux.

[1] Il y a une différence entre la justification et la documentation. Si les méthodes ne sont pas décrites ici, c'est qu'elles le sont dans la documentation.

La méthode isPassedLevel() c'est à savoir si un niveau est terminé pour pouvoir lancer le niveau suivant.
La méthode save() sert à sauvegarder l'état du jeu pour pouvoir le relancer quand on revient sur le programme après l'avoir quitté.

La méthode restart() sert à savoir c'est une partie peut être encore jouée ou non en fonction de l'état du joueur.

La méthode isFinished() sert à savoir si 2 éléments sont sur la même position h notamment pour savoir si le niveau est gagné ou non. Ce sera la comparaison entre deux pions et la méthode est donc différente de isPassedLevel()

3.Partie Console

Nous avons apporté plusieurs modifications et ajouts aux classes existantes, ainsi qu'à de nouvelles classes, pour la partie console du jeu. Ces modifications visaient à améliorer la structure et à intégrer des concepts de conception de logiciels.

Tout d'abord, nous avons introduit les classes Controller et View pour mettre en œuvre le modèle de conception MVC (Modèle-Vue-Contrôleur). Cette approche permet de séparer clairement les responsabilités du jeu, en isolant la logique métier dans le modèle (Game) et en gérant l'interaction avec l'utilisateur via la vue (View) et le contrôleur (Controller).

En outre, nous avons également appliqué le modèle de conception Observateur-Observé. La classe View agit en tant qu'observateur, écoutant les notifications émises par la classe Game, qui agit en tant qu'objet observé. Cela permet à la vue d'être informée des changements d'état du jeu et de mettre à jour l'affichage en conséquence.

En termes de modifications spécifiques apportées aux classes, nous avons apporté des ajustements significatifs aux classes Element_Is_Kill, Element_You, Element_Sink, Element_Push, Element_Win, Element_Stop et Game. Ces classes sont responsables de la logique des éléments du jeu et ont été modifiées pour incorporer de nouveaux attributs et méthodes, ainsi que pour ajuster ou supprimer certains existants.

Une autre classe importante que nous avons ajoutée est BoardLoader, qui a pour fonction de charger le plateau de jeu à partir d'un fichier texte. Elle lit le fichier et crée une structure de

données en fonction de son contenu, en utilisant les classes Board, Square, et Element pour représenter la grille et ses éléments respectivement.

Ainsi, la structure globale du jeu est la suivante : Game (modèle) est lié à BoardLoader, qui à son tour est lié à Board. Le plateau est composé de carrés (Square) qui contiennent des éléments (Element) dans un format vectoriel approprié.

Ces modifications et ajouts ont permis d'améliorer la conception du jeu en rendant le code plus modulaire, maintenable et extensible.

Classe BoardLoader :

Cette classe sert à nous crée un **Board** en lisant des fichiers txt qui représentent les niveaux du jeu. La classe **BoardLoader** lit un fichier de plateau de jeu, crée un vector d'objets de type Square et remplit chaque Square avec un ou plusieurs objets de type Element.

existSquare : vérifie si une Position donnée est déjà occupée par un Square dans un vector d'objets Square.
sortByPosition : est utilisée pour trier un vector d'objets Square en fonction de leur variable membre Position.
createBoardSquares : crée un vector 2D d'objets Square, en utilisant les variables boardRows et boardCols pour déterminer les dimensions du board.
updateWithEmptySquares : Met à jour le vecteur 2D avec des carrés vides pour remplir le tableau en fonction du niveau courant.
switchSymbol : prend une chaîne représentant un élément et renvoie une chaîne correspondante
qui sera utilisée pour afficher l'élément sur le plateau de jeu.
elementPtr remplit des pointeurs vers des objets Element en fonction de leur nom de membre variable. Ces pointeurs sont utilisés pour mettre à jour les objets Square avec les objets Element appropriés.

Classe Game :

bool saveLevel() : cette méthode permet d'enregistrer l'état actuel d'un niveau et renvoie false si l'enregistrement ne s'est pas fait.

bool loadLevel() :cette méthode permet de recharger une niveau enregistré au préalable et renvoie false si ça n'a pas réussi.

bool isRule(std::vector<Element*> elemsToCheck) : cette méthode vérifie si une règle est présente parmi les éléments d'une case, elle renvoie true si une règle se trouve dans ce vecteur.

bool isConnector(std::vector<Element*> elemsToCheck) : cette méthode vérifie si un 'is' est présent parmi les éléments d'une case, elle renvoie true si oui.

void whichRule(Element a,std::vector<Element> rule) : permet de savoir quelle règle il faut désappliquer et applique la bonne règle en fonction de 'rule'.

void DisAllRules(Element a) : permet de désactiver toutes les règles pour un élément si cet élément ne forme aucune règle.

void disapplyRule() : parcours le board et vérifie les règles qui ne sont pas/plus formées pour les désactiver.

void checkRule() : comme on connait la position des 'is' sur le board, regarde au-dessus et en dessous, à gauche et à droite pour voir si une règle est formée, si elle l'est, on l'applique.

bool checkPush(Square* square, Direction dir) : cette méthode permet de vérifier si il y a un ou plusieurs éléments déplaçables dans la direction vers laquelle on veut évoluer. Elle vérifie également si dans cette direction se trouve un bord ou un élément stop, ce qui empêcherait l'avancement des éléments. Cette méthode est récursive.

void updateIsPos() : cette méthode permet de mettre à jour la position des 'is' dans le vecteur qui contient leurs copies afin d'avoir constamment leur position connue.

Les Classes enfants de la classe Element :

Sous-classe Element_Is_Kill :

void apply(Element elem, BoardLoader* boardLoader) , si un élément 'you' est sur la même case qu'un élément 'kill', on supprime l'élément 'you' et on met à jour la liste des éléments déplaçables.

Sous-classe Element_Is_Sink :

void apply(Element elem, BoardLoader* boardLoader) , si un élément 'you' OU 'push' se trouve sur la même case que l'élément sink, on supprime les deux éléments.

Sous-classe Element_Is_You :

void apply(Element elem, BoardLoader* boardLoader) , cette méthode, en fonction de l'élément concerné (qu'on connait grâce au élément_text), permet de mettre les élément déplaçables à jour. Ce sont ces éléments qui sont bougés dans la méthode 'move' de la classe Game.
void disapply(Element elem, BoardLoader* boardLoader) , cette méthode permet de supprimer la liste d'éléments déplaçables. **Sous-classe Element_Is_Win :** void apply(Element elem, BoardLoader* boardLoader) , cette méthode permet de vérifier si un élément 'you' est sur la même case qu'un élément 'win' et incrémente le niveau de jeu si c'est le cas.

Sous-classe Element_Is_Stop :

void apply(Element elem, BoardLoader* boardLoader) , cette méthode permet de mettre les éléments à 'stop' et dans les différentes méthodes de mouvement dans la classe Game, si un élément est 'stop', le mouvement ne se fait pas.

void disapply(Element elem, BoardLoader* boardLoader) , on enlève le fait que ces éléments sont 'stop'.

Sous-classe Element_Is_Push :

void apply(Element elem, BoardLoader* boardLoader) , cette méthode permet de mettre les éléments à 'push' et dans les différentes méthodes de mouvement dans la classe Game, si un élément est 'push', le mouvement se fait.
void disapply(Element elem, BoardLoader* boardLoader) , on enlève le fait que ces éléments sont 'push'.

4.Partie GUI

Dans cette partie du code, nous avons implémenté une interface graphique (GUI) pour le jeu "Baba Is You". Voici une explication des différents éléments utilisés et des concepts mis en œuvre :

Classe MainWindow :

Cette classe représente la fenêtre principale de l'application GUI.
Elle hérite de la classe QMainWindow, ce qui en fait une fenêtre principale.
Elle implémente également l'interface Observer, ce qui lui permet d'observer les changements dans le jeu et de réagir en conséquence.

Fixelles Alban 57343
Sabouri Abdessamad 53960

Classe ControllerGUI :

Cette classe est le contrôleur du jeu, responsable de la logique du jeu et de la gestion des interactions avec l'interface graphique.
Elle hérite de la classe QWidget, ce qui en fait un widget de l'interface utilisateur.
Elle utilise des signaux et des slots pour communiquer avec la classe MainWindow et réagir aux événements utilisateur.

Signaux et slots :

Les signaux et les slots sont un mécanisme de communication entre les objets dans Qt.
Dans notre code, nous utilisons les signaux pour envoyer des informations de la classe MainWindow vers la classe ControllerGUI.
Les signaux sont émis lorsque l'utilisateur interagit avec l'interface graphique, par exemple en appuyant sur une touche.
Les slots sont des fonctions dans la classe ControllerGUI qui réagissent aux signaux et effectuent des actions en conséquence, comme déplacer un personnage dans le jeu.

Rôle des deux classes Game & BoardLoader dans gui:

La classe Game est la classe principale qui représente le jeu "Baba Is You".
Elle est utilisée dans le contrôleur et l'interface graphique pour gérer la logique du jeu, les règles et les mises à jour de l'état du jeu.
La classe BoardLoader est responsable du chargement des niveaux du jeu à partir de fichiers.
Elle est utilisée pour initialiser le jeu avec un niveau spécifique et charger les éléments du tableau.

Utilisation de QLabel et de QTableWidget :

QLabel est utilisé pour afficher des images dans l'interface graphique, représentant les différents éléments du jeu.
QTableWidget est utilisé pour afficher la grille du jeu, où chaque cellule peut contenir un QLabel pour représenter un élément du jeu.

Save et Load :

Tout au long du jeu il est possible de sauvegarder et de charger un niveau. On peut sauvegarder un niveau, quitter le programme et obtenir l'ancien était du jeu en chargeant la sauvegarde. Nous avons décider d'implémenter comme suit, un niveau sauvegardé est supprimé quand le niveau est réussi. Si on essaie de charger un niveau vide, un message d'erreur apparaît.

La mise en œuvre de l'Observateur/Observé :

On a créé une interface Observer avec une méthode Update(Subject*) pour permettre aux observateurs de recevoir des notifications de changements.

La classe MainWindow hérite de l'interface Observer et implémente la méthode Update(Subject*) pour réagir aux changements dans l'objet observé.

On enregistre l'objet MainWindow en tant qu'observateur de l'objet Game en appelant game.Attach(this->view), ce qui permet à MainWindow d'être notifié des changements.
Après chaque modification de l'état du jeu, on appelle game.NotifyObservers() pour informer tous les observateurs du changement.
Lorsque game.NotifyObservers() est appelé, chaque observateur enregistré (dans ce cas, MainWindow) reçoit l'appel de sa méthode Update(Subject*), ce qui leur permet de mettre à jour leur affichage en fonction de l'état actuel du jeu.
En utilisant ce modèle Observer/Observable, on a séparé la logique de jeu (Game) de l'interface utilisateur (MainWindow). L'interface utilisateur réagit aux changements du jeu et met à jour son affichage en conséquence, tandis que le jeu fonctionne indépendamment de l'interface utilisateur. Cela permet une conception modulaire et extensible, favorisant une meilleure séparation des préoccupations.

5.Conclusion

Nous souhaitons conclure notre rapport sur le projet "Baba Is You" en soulignant les points clés que nous avons abordés.

Nous avons utilisé la structure QtCreator pour sa simplicité et sa compatibilité avec notre IDE. La syntaxe du fichier .pro est intuitive, ce qui facilite la personnalisation du projet. Pour la partie métier du jeu, nous avons adopté l'architecture MVC avec le pattern Observateur-Observé. Cela nous a permis de séparer clairement les responsabilités et d'améliorer la synchronisation entre les composants.
Dans la partie console, nous avons apporté des modifications pour améliorer la structure et intégrer des concepts de conception logicielle. Nous avons introduit les classes Controller et View pour implémenter le modèle MVC, et utilisé le pattern Observateur-Observé pour réduire les dépendances.
Nous avons également développé une interface graphique (GUI) en utilisant la classe MainWindow et la classe ControllerGUI. Nous avons utilisé des signaux et des slots pour faciliter la communication et la réactivité de l'interface.

En résumé, nous avons exploré et appliqué des concepts de conception logicielle tels que MVC, le pattern Observateur-Observé et l'utilisation de structures de données appropriées. Nous avons mis l'accent sur la modularité, la maintenabilité et l'extensibilité du code. Nous sommes ouverts à vos commentaires et suggestions pour améliorer notre travail.