Une image contenant Police, logo, Graphique, texte

Description générée automatiquement

**Rapport du projet d’application**

**Travail réalisé en binôme par :**

* Christ Chadrak MVOUNGOU
* Radia MERABTENE

**Groupe :**

**TD :** A **TP :** 1

**Matières évaluées :**

* Génie Logiciel
* Algorithmique
* Interface Homme-Machine

**Année universitaire :** 2024 -2025

1. **Partie génie Logicielle :**
2. **Réalisation de la délimitation système :**

Nous avons adopté le développement en modules, notre application serait un système à utilisateur unique qui est le joueur adversaire de l’IA lançant la partie du jeu.

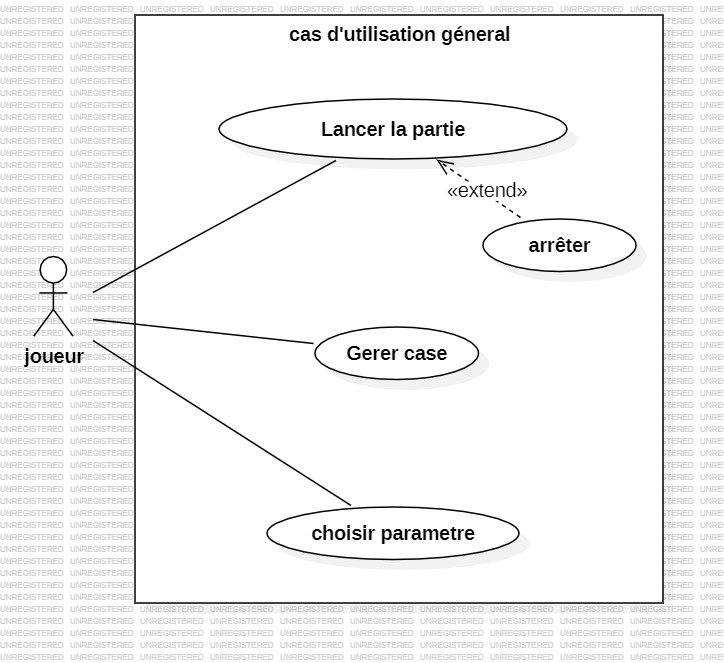
Une image contenant cercle, outil, conception

Description générée automatiquement

1. **Modélisation UML :**

Pour la modélisation nous avons fait le choix de faire deux diagrammes le diagramme des cas d’utilisation et le diagramme de séquences.

1. **Diagramme des cas d’utilisation :**



**Scenario du diagramme des cas d’utilisation :**

Acteurs :

* Principal : Le joueur humain.

Précondition :

* Une partie n’est pas déjà lancée.

Scénario :

* Le joueur ouvre le jeu, il est dirigé vers la page d’acceuil où il y’aura deux boutons :
  + - Lancer une partie.
    - Paramètre.
    - Quitter (ceci pour améliorer l’interaction de l’humain avec l’interface).

1. Premier cas : le joueur clique sur le bouton jouer directement , il aura donc la grille par défaut et commence directement à jouer en considérant les paramètres par défaut( grille 3 x 3, sans cases grisées , couleur par défaut ).
2. Deuxième cas :Le joueur choisit d’abord les paramètres, il valide et la grille se lance selon ses paramètres et il commence à jouer.

Il va alors jusqu’au bout de la partie.

Cas alternatif aux deux cas précédents :

Arrêter la partie : le joueur peut arrêter la partie quand il veut, et le jeu est terminé il est donc redirigé vers l’accueil.

1. **Diagramme de séquence :**

Une image contenant capture d’écran, motif, ligne, art

Description générée automatiquement

1. **Les choix des modules :**

**La structure générale du projet :**

**Tic-tac-toe** : le dossier racine du projet

- Player : un sous dossier contenant les classes palyers (IA, Humain et la classe abstraite player).

Les autres fichiers classes :

Game\_logic.py

Gui.py : le fichier de l’interface du jeu.

Main.py : le fichier contenant la fonction main permettant d’executer le projet.

**Les classes :**

Nous avons adopté le développement en classe en utilisant la programmation objet et ces concepts : abstraction , encapsulation, héritage.

Notre code s’organise en classes qui s’organisent en catégories :

Modules abstraits : on trouve dans cette catégorie les modules joueur (Player) qui est un module abstrait qui sera hérité par les deux autres modules plus spécifiques : PlayerHuman qui représente le joueur humain avec comme méthodes des méthodes permettant de gérer les attributs de cette classe.

Et une méthode pour initier le jeu en utilisant le module GameLogic.

PlayerIA : qui représentera l’ordinateur et qui implémentera l’algorithme minmax pour trouver la position la plus avantageuse à jouer pour l’ordinateur.

Modules Actions : ici on trouve principalement le module représentant la logique du jeu qui se nomme GameLogic, ce module contiendra des fonctions pour vérifier si un des joueurs a remporté la partie (en vérifiant en colonnes en lignes et en diagonales) ou si le match est nul.

Et d’autres fonctions tel la réinitialisation de la partie, jouer une case, création de la grille , redimension de la taille de la grille selon les demandes de l’utilisateur.

Modules Interfaces :

Ici nous mettons le module créant l’interface du jeu.

**Pourquoi ce choix de développement ?**

Utiliser le concept des classes a été favorisé pour une meilleure structuration du code, le jeu impliquant des interactions entre les différents modules et l’interfaces cette méthode d’organisation se montre efficace.

Le choix aussi est confirmé pour assurer des rafactoring sur le code d’une manière plus simple, le jeu étant susceptible de vivre des modifications pour par exemple inclure de nouvelles fonctionnalités telles l’ajout de la fonctionnalité cases grisées (en pyramide ou en random), ceci se fera facilement et efficacement avec ce type de développement en classes.

1. **Convention de codage :**

Pour les conventions de codages nous avons adoptées les mêmes conventions vues en cours à savoir :

**Les variables** : chaque identifiant de variable commencera par la lettre majuscule désignant son type, puis le trigramme correspondant à la classe à qui elle appartient.

**Les types**: chaque type commencera par la lettre T suivi du nom en minuscule.

**Les noms de fonctions**: si la fonction appartient à une classe elle commencera donc par Le trinôme de cette classe sinon elle aura son propre nom commençant par une majuscule.

**Voici le code d’un module modèle suivant cette convention de codage**.

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

La classe se nomme **TPlayer :** le T pour designer que c’est un nouveau type, suivi par un nom significatif pour la classe qui est donc player pour designer que cette classe va représenter des objets joueurs.

Dans le constructeur nous avons les attributs :

**cPLRName**: donc on précède d’abord par le type « c » pour dire que cet attribut est de type chaine de caractères , suivi par le trinôme de la classe player et enfin un nom significatif de l’attribut.

bPLRIsAI : le b pour dire que l’attribut est de type booléen , puis le trinôme de la casse PLR et en fin un mot significatif IsAI : on comprendra donc que cet attribut est un booléen qui confirme si l’objet player instancié est un humain ou IA.

1. **Tests** **unitaires :**

**Les tests unitaires :**

Pour tester notre code nous avons décidé d’analyser la fonction **GLIcheck\_column** : qui va évaluer si une colonne est remplie de façon avantageuse par rapport à un joueur( ce joueur est specifié selon sa couleur passé en paramètre) c’est-à-dire si la partie est remportée par un joueur en analysant les colonnes de la grille du jeu.

Fonction choisie : GLIchech\_column.

Signature de la fonction : def GLIcheck\_column(self, iRow: int, iCol: int, oColor: TColor) -> TPlayer | None:

Les variables influentes sur la partie : iRow : l’indice de la colonne à verifier.

iCol : l’indice de la ligne à verifier.

oColor : la couleur du joueur

1. Variable iRow : l’indice de ligne
2. **Valeur :**

**Domaine moyen :** 0 ≤ iRow≤ taille\_max de la grille (3 ou 4 ou 5) -1 **(A)**

Car l’évaluation des indices commencent à partir de zéro.

**Domaine limite :** iRow ≥taille\_max de la grille (3 ou 4 ou 5). **(B)**

1. **Signes :**

iRow est positif **(C)**

iRow est negatif **(D)**

1. Variable iCol : l’indice de la colonne :
2. **Valeur :**

**Domaine moyen :** 0 ≤ iCol≤ taille\_max de la grille (3 ou 4 ou 5) -1 **(E)**

Car l’évaluation des indices commencent à partir de zéro.

**Domaine limite :** iRow ≥taille\_max de la grille (3 ou 4 ou 5). **(F)**

1. **Signes :**

iRow est positif **(G)**

iRow est negatif **(H)**

1. Variable oColor : la couleur du joueur  :
2. **Valeur :**

**Domaine moyen :** Ocolor = { Rouge, Vert, Jaune , Bleu} **(I)**

**Domaine limite :** Ocolor ne prend aucune couleur ou une couleur non sité dans le domaine moyen **(J)**

**L’énumération des jeux de tests :**

**A+C+E+G+I**: exemple de test : dans le cas ou la grille a pour dimension 3 \*3 :

iRow = 2

iCol= 1

Ocolor= Rouge

**B+C+E+G+J :**

iRow= 3

iCol= 2

Ocolor = noir

Erreur car l’utilisateur utilise une couleur en dehors des couleurs permises.

Observation :

Pour plus de sécurité : on doit typer la valeur de icol et irow a un unsigned int au lieu de int pour éviter des valeurs négatives dans les indices.

Ocolor : créer une énumération des couleurs possibles pour faciliter les choix des joueurs et éviter une couleur non prises en compte par l’interface.

1. **Partie Algorithmique :**
2. **L’Algorithme MinMax :**

Nous avons décidé d’implémenter la stratégie du minmax dans la stratégie du jeu pour l’IA, ainsi nous allons rajouter un niveau de difficulté du jeu pour l’humain, car avec le minmax l’ordinateur cochera la case qui minimisera les chances de l’adversaire de remporter la partie.

**Comment s’organise le minmax dans le code :**

Pour que le jooeur IA joue son tour il utilise 3 fonctions (PLRjouer,PLRMinmax, PLRget\_possible\_moves)

Comment l’IA utilise ces trois fonctions pour déterminer le best move ?

Quand le tour de jouer est à l’IA ce dernier va appeler la fonction PLRMinmax (notre algorihtme minmax principal) cette fonction avant de de générer le meilleur coup du jeu doit d’abord appeler :

La fonction PLRget\_possible\_moves : pour énumérer tous les cas possibles à partir de l’état actuel de la grille du jeu jusqu’à un niveau final c’est-à-dire la ou un des joueurs remporte la partie .

Et à la fin renvois une liste contenant tous les coups possibles c’est-à-dire toutes les cases qui restent cliquable pour le joueur car elles sont vides.

La fonction PLRsimulate\_move : cette dernière va créer une grille en imaginant les cas possibles avec l’ajout d’une des cases simulées par la fonction précédentes (c’est-à-dire elle va générer une grille qui prend pour base la grille de l’état actuel du jeu en remplissant les cases différentes jusqu’à remplissage complet de la grille ) elle formera ainsi un cas possible de jeu.

La fonction Minmax : va maintenant s’appuyer sur ces deux fonctions et leurs résultats et va attribuer des scores pour distinguer les cases favorables et les moins favorables pour l’IA (elle attribue pour cela la valeur 10 quand la case maximise les chances de l’IA de gagner sinon -10 si l’avantage est donné à l’humain).

Cette fonction va aller jusqu’à la profondeur maximale passée en paramètre pour permettre à la fin de récupérer le score le plus avantageux ainsi que les coordonnées de la case qui doit etre jouée par l’IA qui donc serait passé à la fonction PLRjouer pour que l’IA coche cette case dans la grille.

A noter que pour augmenter la difficulté di jeu on doit incrémenter la valeur de la profondeur ce qui ajoutera de la puissance sur l’algorithme de minmax provoquant ainsi plus de chance à l’IA de remporter la partie.

**Premier prototype:**

La fonction PLRget\_possible\_moves : identification des cases vides restantes.

**Fonction PLRget\_possible\_moves:**

Créer une liste vide pour contenir toutes les cases vides restantes

Parcours avec une boucle les lignes de la grille :

Parcours avec une boucle les colonnes de la grille :

Si la case à la i éme ligne et la j éme colonne est vide :

l’ajouter à la liste des moves

retourner la liste des moves

La fonction PLRsimulate\_move : simulation de coups .

**Fonction PLRsimulate\_move :**

Créer une copie de la grille actuelle

Si le tour de joeur est à l’humain :

Modifier la couleur de la case {ixj] en ajoutant la couleur de l’humain

Si le tour est à l’IA :

Modifier la couleur de la case {ixj} en ajoutant la couleur de l’IA

Donner la main à l’autre joueur pour jouer son coup

Retourner la grille créée en copie

Fonction PLRminimax : la fonction principale du minmax :

**Fonction PLRminimax :**

Si le joueur actuel a gangé :

Retourner une valeur +10 et ne rien faire

Si le joueur adversaire a gangé ( l’humain) :

Retourner une valeur -10 et ne rien faire

Si le jeu est terminé (Match null ou profondeur max atteinte) :

Retourner une valeur 0 et ne rien faire

récupérer les cas possibles avec l’appel de la fonction PLRget\_possible\_moves

Si le tour est à l’IA :

Initialiser best\_move et best \_score

Pour chaque cas de case vide possible :

Simuler une grille avec l’appel de la fonction PLRsimulate\_move

Faire un appel récursif sur la fonction PLRminmax avec profondeur -1

Récupérer les valeurs du minmax

Si la valeur du score > best\_score alors :

Best\_score 🡨 score

Modifier best\_move à prendre les coordonné de la case stimulant ce score

Retourner best\_score et best\_move

Si le joueur est humain (minimiser) :

Initialiser best\_move et best \_score

Pour chaque cas de case vide possible :

Pour chaque cas de case vide possible :

Simuler une grille avec l’appel de la fonction PLRsimulate\_move

Faire un appel récursif sur la fonction PLRminmax avec profondeur -1

Récupérer les valeurs du minmax

Si la valeur du score < best\_score alors :

Best\_score 🡨 score

Modifier best\_move à prendre les coordonné de la

case stimulant ce score

Retourner best\_score et best\_move

**LDA du minmax : sur papier :**

1. **Partie IHM :**

**Interface :**

**La page d’acceuil :**

Quand l’utilisateur/joueur clique sur l’icone du jeu il tombera sur cette fenêtre ou deux boutons essentiels se trouveront :

**Lancer la partie** : si le joueur clique dessus une partie du jeu se lancera avec des paramètres par défaut notamment une grille de jeu 3\*3.

**Paramètre**: si le joueur clique dessus un petit formulaire lui permettant de choisir ses paramètres de jeu s’affichera

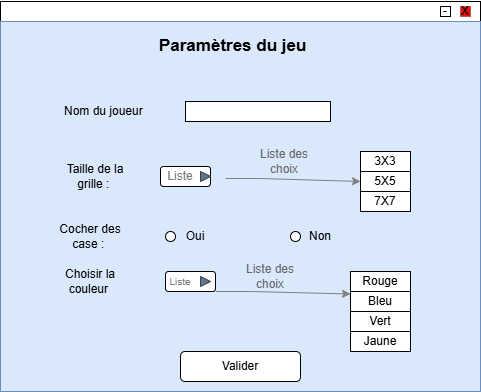
**Quitter :** pour éteindre tout le jeu, plus facile que d’aller sur la barre de la fenêtre.



**La page des paramètres du jeu :**

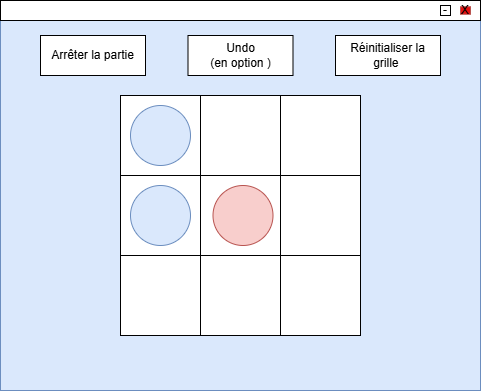
Pour le choix des symboles, nous partons sur des jetons pour lesquelles le joueur aura le droit de choisir la couleur. Les couleurs disponibles seront sous forme d’une liste déroulée.

Nous avons également laissé le choix à l’utilisateur de choisir la dimension de sa grille

****

**La grille du jeu :**

D’abord un menu qui va permettre de terminer la partie , un undo( en option ) pour décocher une case (la dernièrement cliquée) et puis un bouton pour réinitialiser une partie et permette de recommencer à nouveau de jouer.



**Pour l’option :**

Nous avons décidé d’aller avec la fonction undo qui va permettre à l’utilisateur de revenir sur le dernier choix de case effectué.