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

**INTRODUCTION**

Ce projet de développement d’application s’inscrit dans le cadre de l’Unité d’Enseignement Génie Logiciel et Algorithmes de notre formation (S5).

L’objectif principal est de développer un jeu de Tic Tac Toe en implémentant l’algorithme Minimax, tout en appliquant les connaissances acquises dans les cours de Bases du Génie Logiciel, Algorithmes et Structures de Données et Interfaces Hommes-Machines.

Ce projet vise à concevoir une application permettant à un joueur humain d'affronter m’ordinateur (une AI) utilisant l’algorithme Minimax pour prendre des décisions optimales.

Ce rapport présente les différentes étapes du projet, depuis l’analyse du cahier des charges et la définition des spécifications, jusqu’à l’architecture logicielle, le développement des modules et leur implémentation.

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

Figure : Délimiration environnement-système

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 :**

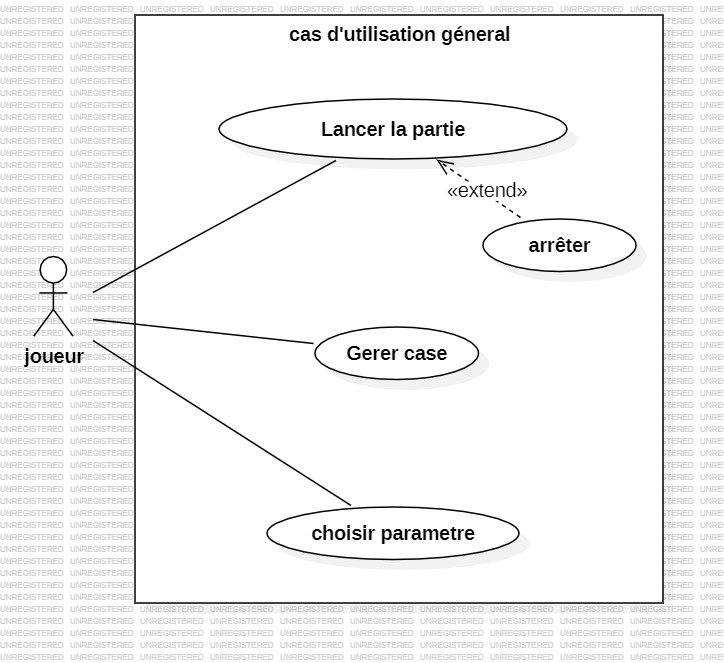


Figure : diagramme de cas d'utilisation

**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

Figure : diagramme de séquence

1. **Architecture du projet et choix des modules**
2. **Architecture du projet**

Nous avons adopté une approche **orientée objet** pour ce développement. Et cela pour plusieurs raisons :

* + - * + Modélisation des entités : L’approche orienté objet permet de représenter les différentes entités du jeu de manière simple. (Joueur, représentant un joueur dans le jeu. La logique du jeu, représentant toutes les règles et la gestion du jeu…)
        + Réutilisabilité : Avec cette approche, nous pouvons facilement modifier le jeu et avoir des bouts de code réutilisable.
        + Encapsulation : L’approche Orientée Objet permet très facilement de cacher les détails de l’implémentation et de contrôle d’accès aux données. Cela nous permet également de pouvoir données accès aux variables par des méthodes getters/setters empêchant des modifications directes.
        + Séparation des responsabilités : Cette approche nous permet de bien séparer les responsabilités entre les classes/entités du jeu rendant le code modulaire et extensible. Chaque classe ayant un rôle bien défini.

L’approche orientée objet nous a permit d’avoir un projet plus modulable et nous a évité également beaucoup de bugs.

TicTacToe/

**|**--- main.py : le fichier contenant la fonction main permettant d’exécuter le projet.

**|**--- gui.py : Interface graphique

**|---** game\_logic.py : Logique du jeu

**|**--- Player/

**|**--- Player.py : Classe de base pour les joueurs

**|**--- PlayerAI.py : Classe pour le joueur IA

**|---** PlayerHumain.py : Classe pour le joueur humain

**|**--- color.py : Gestion des couleurs

Chaque fichier représente une classe qui gère une partie bien définie du projet.

1. **Choix des modules**

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

* + - * + **Modules type abstrait :** Les modules abstraits représente toute entité dans un application. Dans notre cas de jeu de Tic Tac Toe, on trouve dans cette catégorie les modules joueur (***Player***) qui est un module abstrait qui sera hérité par les deux autres modules plus spécifiques : ***PlayerHumain*** représentant le joueur humain et ***PlayerAI*** représentant IA/Algo MinMax/Ordinateur.
        + **Modules type Action :** Les modules d’action représente un ensemble de traitement complexes dans une application. 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), si le match est nul, et gérera toute la logique du Jeu Tic Tac Toe.
        + **Modules Entrées/Sorties :** Les modules entrées/sorties contient l’ensemble des traitements et objets liés aux opérations d’entrées/sorties (cf. cours GL). Dans notre cas, l’interface utilisateur permet aux joueurs de pouvoir interagir avec notre système de la façon la plus fluide possible.

Le choix de l’architecture du projet, l’approche du développement et le choix des modules a été faite pour assurer un code de qualité, favorisant le refactoring, le jeu étant susceptible d’être modifié ou amélioré.

1. **Convention de codage :**

Nous avons adopté 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. Une image contenant texte, capture d’écran, logiciel, Police

          Description générée automatiquement

Figure : Exemple de code avec convention de nommage décrite

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**: La variable commence par le type « **c** » pour chaine de caractère, suivi du trigramme de la classe Player (**PLR)** et enfin un nom significatif de l’attribut.

**bPLRIsAI :** le « **b** » pour booléen , puis le trigramme 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 spécifié 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 influençant cette fonction :

* + - * + iRow : l’indice de la ligne à vérifier.
        + iCol : l’indice de la colonne à vérifier.
        + oColor : la couleur du joueur

1. **Variable iRow :**
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 négatif 🡪 **(D)**

1. **Variable iCol :**
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 :**
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** (pour des langages de programmation ayant ce type) au lieu d’un type **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 prise en compte par l’interface.

1. **PARTIE ALGORITHMIQUE :**
2. **L’Algorithme MinMax :**

***Rappel :*** L'algorithme **Minimax** est une méthode utilisée pour prendre des décisions optimales. Dans notre cas, nous l’utilisons pour que l’adversaire du joueur humain puisse prendre des meilleures décisions.

* + - * + **Principe :**

Nous utilisons l’algorithme de minmax pour trouver le meilleur coup pour le joueur AI/Ordinateur en simulant tous les coups possibles, tout en supposant que l'adversaire (le joueur humain) joue de manière optimale.

L'algorithme explore un arbre de décisions, où chaque nœud représente un état du jeu.

* **Min** : Correspond aux tours de l'adversaire, qui cherche à minimiser le score du joueur.
* **Max** : Correspond aux tours du joueur AI, qui cherche à maximiser son score.
  + - * + **Étapes :**

1. **Exploration de l'arbre** : L'algorithme simule tous les mouvements possibles jusqu'à une certaine profondeur. Nous avons utilisé la profondeur dans ce jeu pour gérer le niveau de jeu. Etant donné que plus la profondeur de l’arbre est élevée, plus le joueur analyse les meilleurs coups possibles.
2. **Évaluation des états** : Chaque état du jeu est évalué donnant +10 au joueur pour un bon coup possible et un -10 pour un état où le joueur est dans la bonne situation de gagner sinon un 0 pour un match nul. Cela indiquera donc à l’AI qu’il n’y a pas trop d’intérêt de considérer cet état.
3. **Propagation des scores** : A la fin de l’exploration, arrivée au niveau des feuilles, le joueur AI remontent dans l’arbre en faisant la somme des scores.
4. **Décision finale** : Le joueur AI peut ainsi choisir le mouvement associé au meilleur score.

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

Pour que le joueur IA joue son tour il utilise quatres fonctions (PLRjouer, PLRMinmax, PLRget\_possible\_moves et simulated\_game)

**Roles de ces functions :**

* + - * + La fonction PLRget\_possibles\_moves : Fais un passage sur l’état du jeu et enregistre tous les coups/mouvements possibles.
        + La fonction PLRsimulate\_move : A partir des coups/mouvements possibles, le joueur AI va simuler un coup et elle formera ainsi un cas possible de jeu.
        + La fonction PLRMinMax : Cette fonction se rappelle récursivement avec la profondeur qui s’incrémente 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 sera jouée par l’IA.
        + La fonction PLRjouer : Peut maintenant utiliser le bon coup retourné par la fonction Minmax pour jouer.

**Stratégie d’implémentation de niveau de difficulté.**

Nous savons que plus la profondeur est élevée, plus le joueur AI prend la décision la plus optimale, cela nous permet donc d’incrémenter la profondeur du jeu à 1 à chaque fois que le joueur Humain qui gagne la partie.

---------------------------------- IMAGE A INSERER

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

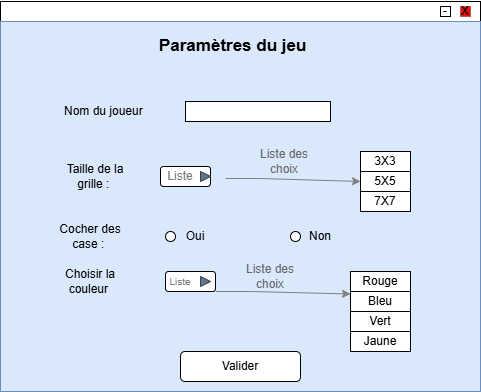
**Quitter :** pour éteindre tout le jeu, plus facile que d’aller sur la barre de la fenêtre.



**La page des paramètres du jeu :**

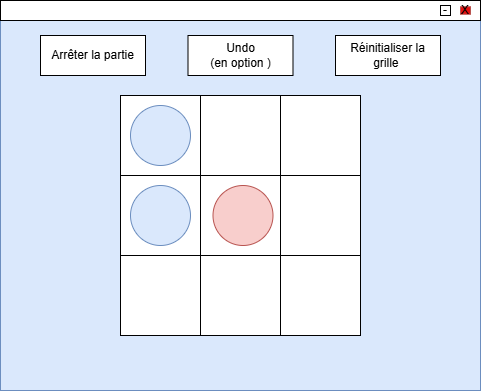
Pour le choix des symboles, nous partons sur des jetons pour lesquelles le joueur aura le droit de choisir la couleur. Les couleurs disponibles seront sous forme d’une liste déroulée.

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.