

SAÉ 2.2

Joueur artificiel de Puissance 4

Rapport de modélisation et conception objet

Chauderon Enzo – Diaz Matthieu – Alexandre Gabriel

Introduction

Ce rapport présente la modélisation orientée objet du jeu Puissance 4 avec un joueur artificiel dans le cadre de la SAÉ 2.2. Il décrit les différentes classes utilisées pour le jeu et l'intelligence artificielle, leurs relations ainsi que les principales méthodes utilisées.

L'objectif est de produire une analyse orientée objet du problème, sans implémenter le code, en justifiant les choix de conception. Notre objectif est aussi de proposer une architecture assez flexible pour permettre différentes variantes du jeu, comme des tailles de grille différentes ou un autre nombre de pions à aligner.

1. Synthèse rédactionnelle sur les classes

1.1 Combien de classes et pourquoi

La modélisation que nous proposons repose sur trois classes principales :

- **Puissance4** : classe parente abstraite qui regroupe les attributs et méthodes communs aux classes Partie et IA, afin d'éviter la redondance.
- **Partie** : classe fille qui gère le déroulement du jeu, les interactions avec les joueurs et l'état de la partie.
- **IA** : classe fille qui encapsule toute la logique de l'intelligence artificielle, notamment l'algorithme minimax et l'heuristique d'évaluation.

Ce découpage en trois classes permet de séparer les rôles de chaque partie du programme afin de garder une structure plus claire.

La classe Puissance4 permet d'éviter de dupliquer les attributs communs dans Partie et IA, notamment la grille de jeu et les informations sur les dimensions du plateau. Cette architecture permet aussi de séparer la logique du jeu classique de celle de l'intelligence artificielle, ce qui rend le programme plus simple à modifier ou à compléter plus tard.

1.2 Attributs de chaque classe

Classe Puissance4 (protected)

Ces attributs sont accessibles par les classes filles via le mot-clé protected :

Attribut	Type	Rôle
nbLigne	int	Nombre de lignes de la grille
nbColonne	int	Nombre de colonnes de la grille
grille	char[nbLigne][nbColonne]	Le plateau de jeu. On utilise char car c'est le type le plus léger permettant de représenter 3 états (0 = vide, 1 = joueur 1, 2 = joueur 2)
colonnePleine	bool[nbColonne]	Indique les colonnes pleines et empêche de placer un pion sur cette colonne et donc simplifie les calculs
nbPionAligner	int	Nombre de pions à aligner pour gagner. Cet attribut permet de rendre le programme adaptable a différentes variantes du jeu.
tourJoueur	bool	Joueur actuel (true = J1, false = J2)

Classe Partie (protected)

Attribut	Type	Rôle
joueurIA	IA	Objet gérant le joueur IA s'il est utilisé. Il est initialisé à NULL lorsqu'aucun joueur artificiel n'est actif.
partieTerminee	bool	Indique si la partie est finie

Classe IA (protected)

Attribut	Type	Rôle
profondeurMax	int	Profondeur maximale de l'arbre de recherche minimax. Plus cette valeur est élevée, plus l'IA anticipe loin, mais plus le temps de calcul augmente.
difficulte	int	Associe un nombre (comme 1 ou 2) pour la difficulté de l'IA Idiot ou Avancé, qui aura un impact direct sur l'heuristique et le niveau de profondeur à calculer

1.3 Redondances et complémentarités

Sans la classe Puissance4, les attributs nbLigne, nbColonne, grille et colonnePleine devraient être dupliqués dans Partie et IA.

Or l'IA a besoin d'accéder à la grille pour simuler des coups et évaluer les positions. L'héritage commun évite cette redondance et centralise les structures de données importantes.

Les méthodes de vérification de victoire :

- calculGagnant
- calculGagnantLignes
- calculGagnantColonnes
- calculGagnantDiagonales

sont placées dans Puissance4 car elles sont utilisées à la fois par Partie (pour savoir si un joueur a gagné) et par IA (pour évaluer une position simulée). Cela permet aussi d'éviter des différences entre les calculs faits pendant une vraie partie et ceux réalisés pendant les simulations de l'IA.

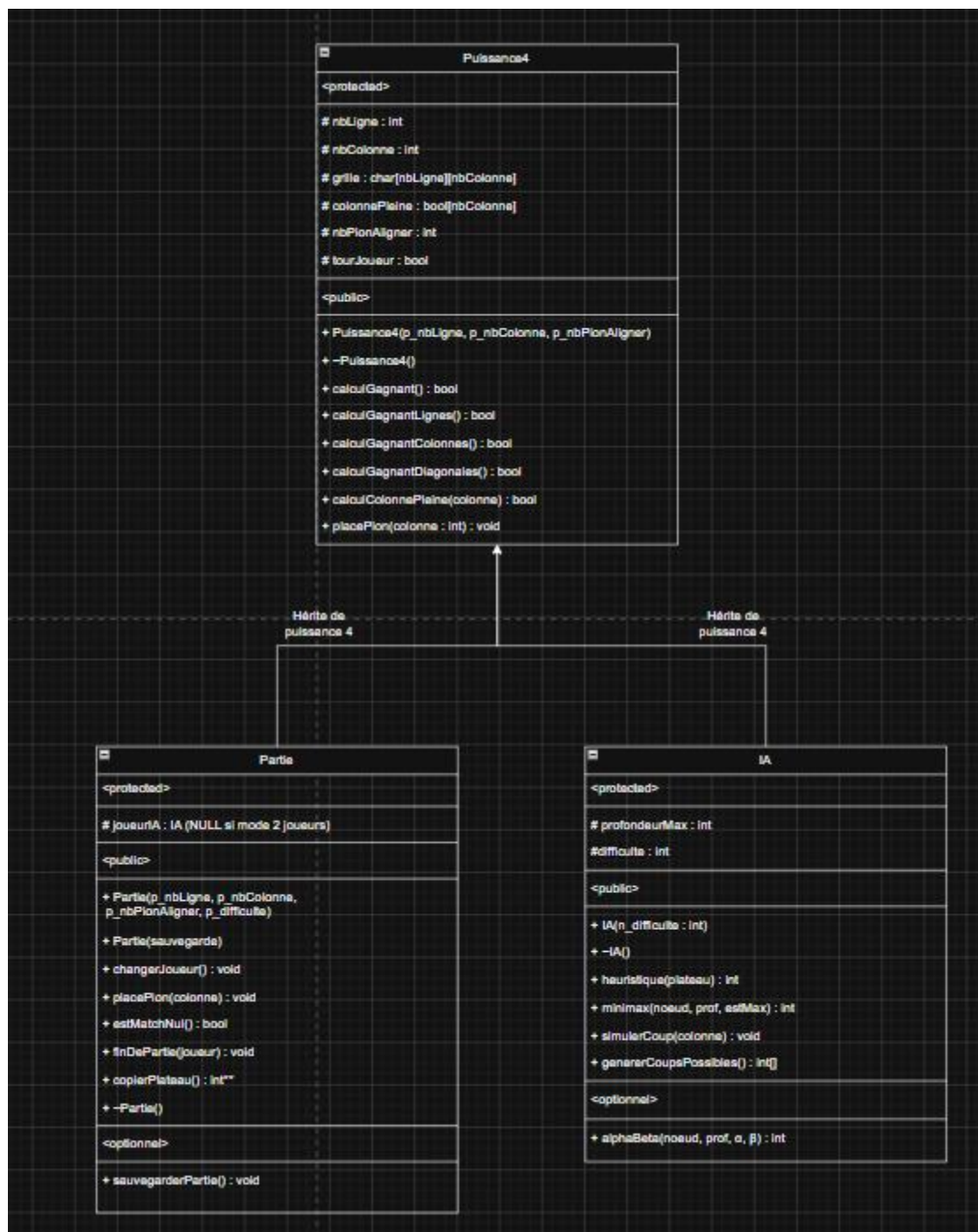
2. Diagramme de classes

2.1 Relations entre les classes

Les relations entre les classes sont les suivantes :

Relation	Description
Héritage Partie -> Puissance4	Partie hérite de Puissance4 pour accéder aux attributs communs du jeu sans les redéfinir. Elle y ajoute la gestion du déroulement de la partie.
Héritage IA -> Puissance4	IA hérite de Puissance4 pour pouvoir simuler des coups sur la grille et calculer des scores sans maintenir sa propre copie des structures de données.
Association Partie -> IA	Partie contient un attribut joueurIA de type IA. Cette relation représente le fait qu'une partie peut être jouée contre une intelligence artificielle.

DIAGRAMME :



2.2 Justification des choix relationnels

Le choix de l'héritage plutôt que la composition pour relier IA et Partie à Puissance4 se justifie par le fait que ces deux classes utilisent directement les mêmes données principales du jeu.

Elles partagent notamment :

- la grille de jeu,
- les dimensions du plateau,
- les méthodes de vérification de victoire,
- les méthodes de vérification des colonnes.

Une architecture par composition avec une classe Plateau indépendante aurait également été possible. Dans cette architecture alternative, la classe Partie aurait contenu un objet Plateau et la classe IA aurait utilisé ce même objet pour effectuer ses simulations.

Cette solution était plus proche d'une architecture orientée objet "classique". Cependant, nous avons préféré utiliser l'héritage afin de simplifier l'accès aux données communes dans le cadre de cette SAÉ.

La relation d'association entre Partie et IA représente le fait qu'une partie peut optionnellement inclure un joueur IA. Cette relation est plus souple qu'un héritage direct car une partie sans IA ne crée pas d'objet IA inutile.

3. Description des méthodes importantes

3.1 Méthodes de la classe Puissance4

Ces méthodes sont partagées entre Partie et IA :

calculGagnant() : bool :

Vérifie si un joueur a gagné en appelant les trois méthodes suivantes.

calculGagnantLignes() : bool :

Parcourt toutes les lignes horizontales et vérifie si nbPionAligner pions identiques sont alignés.

calculGagnantColonnes() : bool :

Même vérification sur les colonnes verticales.

calculGagnantDiagonales() : bool :

Vérifie les diagonales. Séparée en méthode propre pour ne pas surcharger calculGagnant.

calculColonnePleine(colonne) : bool :

Retourne true si la colonne est pleine. Met à jour le tableau colonnePleine si besoin.

3.2 Méthodes de la classe Partie

- **Partie(p_nbligne, p_nbColonne, p_nbPionAligner, p_difficulte)**
Constructeur par défaut. Initialise la grille avec des cases vides, met tourJoueur à true et initialise joueurIA à NULL.
- **Partie(sauvegarde)**
Constructeur de reprise. Charge l'état d'une partie sauvegardée depuis une base de données SQL.
- **placePion(colonne) : void**
Méthode centrale du jeu.

Elle insère un pion dans la colonne donnée, puis :

- appelle calculGagnant pour vérifier la victoire
- appelle calculColonnePleine pour mettre à jour l'état de la colonne
- appelle estMatchNul pour vérifier si la grille est pleine
- appelle changerJoueur pour passer la main

La vérification que la colonne n'est pas pleine est réalisée avant l'appel de cette méthode.

- **changerJoueur() : void**
Inverse tourJoueur.

Si joueurIA n'est pas NULL et que c'est au tour de l'IA, la méthode appelle joueurIA.simulerCoup().

- **estMatchNul() : bool**
Vérifie si toutes les colonnes sont pleines sans qu'il y ait de gagnant.
- **finDePartie(joueur) : void**
Termine la partie et annonce le gagnant ou le match nul.

- **copierPlateau() : Partie**
Retourne une copie complète de l'état du jeu.

Cette méthode est utilisée par l'IA pour simuler des coups sans modifier la partie réelle.

- **~Partie()**
Destructeur libérant les ressources mémoire allouées dynamiquement.

3.3 Méthodes de la classe IA

Ces méthodes fonctionnent ensemble pour permettre à l'IA de choisir le meilleur coup :

- **IA(n_difficulte)**
Constructeur de la classe IA et initialisation de la difficulté.
- **simulerCoup(colonne) : void**
Appelle copierPlateau() pour obtenir une copie de la grille, puis simule un coup sur cette copie sans modifier la partie réelle.

Cette méthode est utilisée en boucle par minimax afin d'explorer les différentes branches de l'arbre de recherche.

- **heuristique(plateau) : int**
Évalue la qualité d'une position de jeu en parcourant toutes les fenêtres possibles de taille N :
 - horizontales
 - verticales
 - diagonales

Chaque fenêtre reçoit un score selon :

- le nombre de pions de l'IA
- le nombre de pions adverses
- le nombre de cases vides

Dans notre implémentation, les scores positifs représentent des positions favorables à l'IA tandis que les scores négatifs représentent des positions favorables au joueur humain.

Ce choix reste arbitraire : il serait également possible d'utiliser l'inverse en adaptant simplement les comparaisons effectuées dans l'algorithme minimax.

Exemples de fenêtres évaluées :

Situation	Exemple	Explication	Score possible
Position neutre	[][][][]	Aucun joueur n'a d'avantage.	0
Début d'alignement IA	[X][X][][]	L'IA commence à construire un alignement.	+100
Position très avantageuse pour l'IA	[X][X][X][]	L'IA est à un coup de gagner.	+1000
Victoire IA	[X][X][X][X]	Alignement complet de l'IA.	+100000
Menace adverse	[O][O][O][]	Le joueur humain peut gagner au prochain tour si l'IA ne bloque pas. Le score est volontairement élevé afin de forcer l'IA à défendre.	-5000
Contrôle du centre	[][][X][]	Les positions proches du centre offrent plus de possibilités d'alignements.	Bonus supplémentaire
Alignement bloqué	[X][X][O][]	L'alignement de l'IA est bloqué par un pion adverse.	+10

Le principe est le même pour les colonnes et les diagonales, cependant ceci reste une proposition d'heuristique parmi des centaines de possibilités. De plus les scores attribués restent principalement expérimentaux et pourront être ajustés pendant les phases de test afin d'obtenir un comportement plus efficace de l'IA.

- **minimax(noeud, profondeur, estMax) : int**
Algorithme de recherche arborescente.

Il explore récursivement tous les coups possibles jusqu'à la profondeur maximale.

Lorsque profondeur = 0, il appelle heuristique afin d'évaluer la position.

Sinon, il alterne entre :

- maximisation du score pour l'IA ;
- minimisation du score pour le joueur adverse

Le temps de calcul augmente rapidement avec la profondeur de recherche car chaque coup possible crée plusieurs nouvelles branches.

Une victoire rapide peut également recevoir un meilleur score qu'une victoire plus lointaine afin d'encourager l'IA à terminer la partie le plus vite possible.

- **genererCoupsPossibles() : int[]**
Retourne la liste des colonnes jouables

Cette méthode permet à minimax et alphaBeta de ne parcourir que les coups valides.

- **alphaBeta(noeud, profondeur, alpha, beta) : int**
Optimisation de minimax par élagage alpha-bêta.

Cette méthode évite d'explorer certaines branches dont le résultat ne peut plus influencer la décision finale.

Elle réduit donc fortement le temps de calcul.

- **~IA()**
Destructeur de la classe IA.

3.4 Coopération des méthodes IA avec le jeu

Lorsque c'est au tour de l'IA, la méthode changerJoueur de Partie appelle simulerCoup. Celle-ci utilise copierPlateau pour travailler sur une copie, puis lance minimax. Minimax explore l'arbre en appelant simulerCoup pour chaque coup possible et heuristique pour évaluer les feuilles de l'arbre. Le meilleur coup trouvé est retourné à Partie qui appelle alors placePion avec ce coup

À chaque niveau de l'arbre, minimax vérifie aussi calculGagnant (hérité de Puissance4) pour détecter une victoire immédiate et arrêter l'exploration de cette branche.

3.5 Méthodes optionnelles

alphaBeta(noeud, profondeur, alpha, beta) : int : optimisation de minimax par élagage alpha-bêta. Permet d'éviter d'explorer des branches dont le résultat ne peut pas influencer la décision finale, réduisant significativement le temps de calcul

sauvegarderPartie() : void : enregistre l'état actuel de la partie dans une base de données SQL pour permettre une reprise ultérieure via le constructeur Partie(fichierSQL)

Conclusion

Cette modélisation en trois classes permet une séparation claire des responsabilités :

- Puissance4 centralise les données communes
- Partie gère le déroulement du jeu
- IA encapsule toute la logique algorithmique de l'intelligence artificielle

L'héritage entre Puissance4, Partie et IA évite la redondance des attributs et des méthodes de vérification de victoire.

L'association entre Partie et IA permet une conception flexible où le mode IA est optionnel

Cette architecture facilite également l'ajout futur de fonctionnalités comme :

- la sauvegarde de partie
- l'optimisation par alpha-bêta
- des heuristiques plus avancées
- différentes tailles de plateau
- Et encore plus peut être

