# Rapport de Projet Personnel : Conception d'un Moteur de Jeu Puissance 4 en Java

Matthieu Comparetto

Juin - Juillet 2025

# Table des matières

**1 Introduction** **3**

**2 Architecture Logicielle** **3**
2.1 Le "Cœur" du Système : `Arbitre.java` . . . . . 3
2.2 Flexibilité et Paramétrage . . . . . 3
2.3 Abstraction et Polymorphisme . . . . . 4
2.4 Encapsulation . . . . . 5

**3 Implémentation Technique** **5**
3.1 L'Algorithme de Victoire (`Direction.java`) . . . . . 5
3.2 Gestion des Exceptions et Robustesse . . . . . 5

**4 Conclusion** **6**

# 1 Introduction

Désireux de mettre à l'épreuve les compétences acquises durant ma première année de cycle ingénieur, j'ai entrepris la réalisation autonome d'un moteur de jeu Puissance 4 en Java. Ce projet personnel vise à valider ma maîtrise de la Programmation Orientée Objet à travers un cas concret, tout en développant une logique logicielle robuste, évolutive et capable de gérer des interactions utilisateurs complexes.

# 2 Architecture Logicielle

L'architecture globale du projet repose sur une séparation stricte des responsabilités (Modèle-Vue-Contrôleur simplifié) afin de garantir la clarté et la maintenance du code.

L'entrée du programme se fait via la classe `Menu.java`, qui gère l'interface utilisateur en ligne de commande. Ce menu offre une expérience fluide et sécurisée : grâce à une boucle de contrôle active, le programme est immunisé contre les erreurs de saisie (lettres à la place de chiffres, choix inexistants), assurant une exécution sans interruption.

Les fonctionnalités proposées sont :

— Joueur vs Ordinateur (IA aléatoire)
— Joueur vs Joueur (Local)
— Paramétrage du jeu (Dimensions, règles)
— Crédits et gestion de sortie

## 2.1 Le "Cœur" du Système : `Arbitre.java`

La classe `Arbitre.java` agit comme le chef d'orchestre du jeu. C'est elle qui détient la logique du déroulement d'une partie. Elle instancie la grille, gère l'alternance des tours et interroge la classe `Direction` pour vérifier les conditions de victoire après chaque coup.

Sa force réside dans son utilisation du polymorphisme : l'arbitre manipule des objets de type `Joueur`, sans avoir besoin de savoir s'il s'agit d'un humain ou d'une intelligence artificielle.

## 2.2 Flexibilité et Paramétrage

Contrairement à une approche rigide, j'ai choisi de ne pas fixer les dimensions de la grille (`Nb_ligne`, `Nb_Colonne`) ni le nombre de jetons à aligner (`nbAlign`) comme des constantes immuables. En utilisant des variables

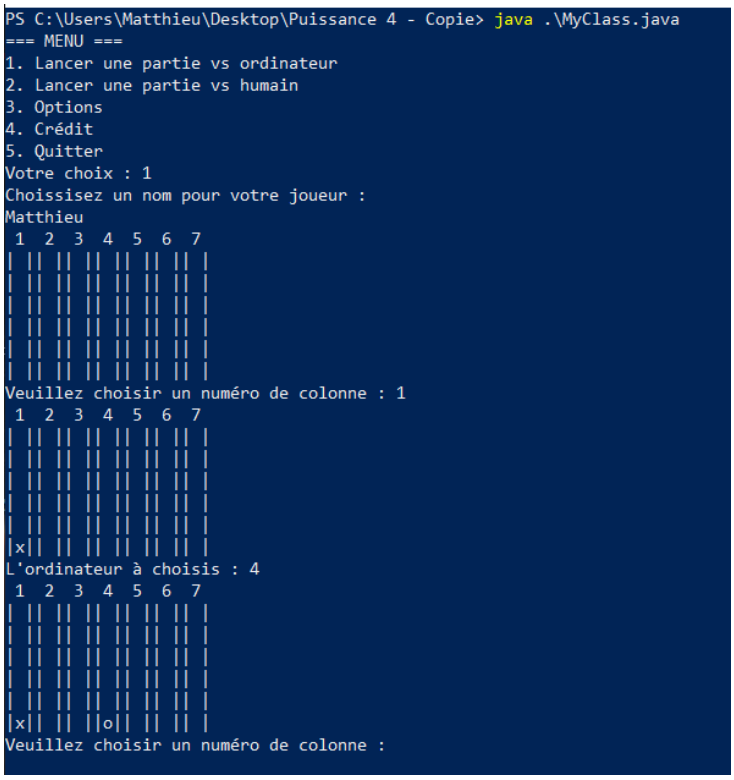


FIGURE 1 – Interface textuelle du menu principal

statiques modifiables via des *setters* dans les classes `Grille` et `Direction`, l'utilisateur peut transformer le traditionnel "Puissance 4" en "Puissance 5" sur une grille géante, offrant ainsi une modularité totale au moteur de jeu.

## 2.3 Abstraction et Polymorphisme

Le projet définit une interface `Joueur.java`, imposant un contrat strict pour toute entité souhaitant participer à la partie. Deux implémentations ont été réalisées :

- **JoueurHumain :** Récupère les entrées via le scanner et gère la validation des colonnes.
- **JoueurOrdinateur :** Utilise une logique de décision automatisée (actuellement basée sur un choix aléatoire valide).

Cette structure permet d'injecter facilement une IA avancée (algorithme Minimax) à l'avenir, illustrant le principe de "ouvert à l'extension, fermé à la modification".

### 2.4 Encapsulation

La classe `Grille.java` protège l'état interne du jeu. Le tableau de données est `private`, forçant toute interaction à passer par des méthodes sécurisées. Par exemple, `remplirCase` vérifie l'état de la cellule avant toute modification, garantissant l'intégrité des règles du jeu contre toute manipulation externe erronée.

## 3 Implémentation Technique

### 3.1 L'Algorithme de Victoire (`Direction.java`)

La détection de la victoire représentait le défi algorithmique majeur de ce projet. Plutôt que de parcourir l'intégralité de la grille à chaque tour (opération coûteuse), j'ai opté pour une vérification locale autour du dernier jeton posé.

L'algorithme utilise la méthode `versDirection` de manière récursive. Pour chaque axe (Horizontal, Vertical, Diagonales), il explore les deux directions opposées (ex : Nord et Sud) et additionne les jetons identiques consécutifs. Si le total atteint ou dépasse `nbAlign`, la victoire est proclamée. Cette approche est à la fois élégante, performante et s'adapte automatiquement à n'importe quel changement du nombre de jetons requis pour gagner.

### 3.2 Gestion des Exceptions et Robustesse

L'utilisation des exceptions personnalisées (`extends Exception`) a été un choix délibéré pour gérer les flux d'erreurs de manière "propre" et professionnelle. Plutôt que d'utiliser des codes d'erreur numériques (souvent sources de confusion), le programme lance :

- `GrilleException` : Si l'utilisateur tente de viser une colonne hors des limites.
- `ColonnePException` : Si la colonne choisie est déjà saturée.

Ce mécanisme permet de séparer la logique de détection d'erreur (dans `Grille` ou `Joueur`) de la logique de traitement (dans le `Menu`). Lorsqu'une exception est capturée, le programme affiche un message explicatif et sollicite une nouvelle saisie sans jamais planter, offrant une expérience utilisateur robuste.

# 4 Conclusion

Ce projet m'a permis de transformer des concepts théoriques de Programmation objet en une application logicielle concrète et fonctionnelle. La séparation des responsabilités et l'usage de patterns comme le polymorphisme et l'encapsulation offrent une base solide. Ce moteur est désormais prêt pour des évolutions majeures, telles que l'ajout d'une interface graphique moderne ou d'une intelligence artificielle compétitive.