WARGAME JAVA

Un jeu de stratégie inspiré du monde (imaginaire) du seigneur des anneaux.

GHODBANE Rachid

Licence 3 Informatique – Université Jean Monnet 2025/2026

Table des matières

1 Introduction

Le projet **WARGAME Java** est un jeu de stratégie au tour par tour développé en Java dans le cadre de l'enseignement de Programmation Orientée Objet en Licence 3 Informatique. L'objectif principal est de mettre en pratique les concepts fondamentaux de la POO à travers la conception et l'implémentation d'un jeu complet et fonctionnel.

Le jeu oppose le joueur contrôlant une équipe de héros à une intelligence artificielle gérant une armée de monstres sur une carte hexagonale. Le but est de détruire la base ennemie tout en défendant la sienne, en utilisant différents types d'unités aux caractéristiques variées.

Le code source complet du projet est disponible sur GitHub à l'adresse suivante :

https://github.com/rvsh0x/wargame

2 Analyse du projet

2.1 Architecture générale

Le projet est organisé selon une architecture MVC (Modèle-Vue-Contrôleur) adaptée aux jeux vidéo :

2.1.1 MODÈLE (Logique métier)

- Gestion de la carte et des éléments
- Règles de jeu et mécanique de combat
- Intelligence artificielle des monstres
- Système économique (or et améliorations)

2.1.2 VUE (Interface graphique)

- Fenêtres (Menu, Jeu, Personnages, Classement)
- Panneaux de jeu et d'information
- Menus contextuels (Pause, Sauvegarde, Paramètres)

2.1.3 CONTRÔLEUR (Gestion des événements)

- Gestion des clics souris
- Gestion des touches clavier
- Logique de tour par tour

Note : Le projet contient 29 classes organisées dans le package `wargame`. Les diagrammes UML complets (diagramme de classes, diagramme par couches, architecture simplifiée) sont disponibles en ***Annexe A*** *(pages 17, 18 et 19).*

3 Techniques de POO/Java mises en œuvre

3.1 Héritage

La hiérarchie principale du projet utilise l'héritage pour réutiliser et spécialiser le code :

Element $\rightarrow$ **Soldat** $\rightarrow$ **{Heros, Monstre}**

Avantages :

— Réutilisation du code (combat, déplacement)

— Spécialisation par sous-classe (amélioration pour Heros)

— Maintenance facilitée

Voir exemple de code en Annexe B.1 (page 20).

3.2 Encapsulation

Application systématique du principe d'encapsulation avec :

— Attributs privés ou protégés

— Accesseurs (getters) et mutateurs (setters)

— Visibilité restreinte (package-private pour certains constructeurs)

— Immutabilité pour les types (final)

Voir exemple de code en Annexe B.2 (page 20).

3.3 Polymorphisme

Le polymorphisme est utilisé sous trois formes dans le projet :

— **Polymorphisme d'héritage** : Méthodes abstraites redéfinies dans les sous-classes

— **Polymorphisme d'interface** : Implémentation de contrats (ISoldat, ICarte)

— **Polymorphisme paramétrique** : Utilisation de types génériques (Class< ?>)

Voir exemples de code en Annexe B.3 (page 20).

3.4 Interfaces

Les interfaces définissent des contrats clairs :

— **ISoldat** : Contrat pour tous les soldats

— **ICarte** : Contrat pour la gestion de la carte

Avantages :

— Contrats clairs et explicites

— Flexibilité dans l'implémentation

— Facilite les tests et l'évolution

3.5 Classes abstraites

Deux classes abstraites principales structurent le modèle :

1. **Element** : Base de tous les objets de la carte
   - Constructeurs concrets
   - Méthodes utilitaires (estHeros, estMonstre)
   - Attributs communs (position, carte)
2. **Soldat** : Base de tous les combattants
   - Logique de combat commune
   - Méthodes abstraites pour la spécialisation
   - Implémentation du Template Method pattern

Choix de conception : Classe abstraite si état commun ET comportement partagé, interface si seulement contrat.

3.6 Énumérations

Les énumérations enrichies encapsulent données et comportements :
Avantages :

- Type-safety (sécurité des types)
- Lisibilité accrue
- Centralisation des constantes
- Méthodes statiques de génération (getTypeHAlea, getTypeHUnique)

Voir exemple de code en Annexe B.4 (page 21).

3.7 Exceptions personnalisées

Exception `PasUnMonstre` (inner class dans Carte) pour la validation métier.
Avantages :

- Gestion d'erreurs spécifiques au domaine
- Messages d'erreur clairs et contextuels
- Séparation des erreurs métier des erreurs techniques

Voir exemple de code en Annexe B.5 (page 21).

3.8 Sérialisation

Toutes les classes du modèle implémentent `Serializable` pour la persistance.
Système de sauvegarde complet :

- Classe Sauvegarde avec inner classes Donnees et Metadonnees
- Sauvegarde de l'état complet du jeu
- 4 slots de sauvegarde disponibles
- Affichage des métadonnées (pseudo, temps, date)

Voir exemple de code en Annexe B.6 (page 21).

3.9 Système de grille hexagonale

Un des défis techniques majeurs du projet a été l'implémentation d'une grille hexagonale pour la carte de jeu. Contrairement à une grille carrée classique, les hexagones offrent 6 voisins directs et créent des chemins de déplacement plus naturels. Cette section détaille les fondements mathématiques et algorithmiques de ce système.

3.9.1 Théorie des coordonnées hexagonales

Les grilles hexagonales peuvent être représentées selon plusieurs systèmes de coordonnées. Le projet utilise le système de coordonnées *offset* (décalé) avec alignement axial vertical, où chaque hexagone est identifié par un couple (x, y) :

— $x \in [0, L]$: colonne (largeur de la carte)

— $y \in [0, H]$: ligne (hauteur de la carte)

Dans ce système, les colonnes paires et impaires sont décalées verticalement de $\frac{1}{2}$ unité, créant une asymétrie qui doit être prise en compte dans les calculs de voisinage.

3.9.2 Formalisation mathématique du voisinage

Soit $H(x, y)$ un hexagone à la position (x, y). Le calcul des voisins diffère selon la parité de x :

Cas 1 : Colonne paire ($x \equiv 0 \pmod 2$)
Les 6 voisins de $H(x, y)$ sont définis par :

$$N_{\text{pair}}(x, y) = \{(x-1, y), (x, y-1), (x+1, y), (x+1, y+1), (x, y+1), (x-1, y+1)\} \tag{1}$$

Cas 2 : Colonne impaire ($x \equiv 1 \pmod 2$)
Les 6 voisins de $H(x, y)$ sont définis par :

$$N_{\text{impair}}(x, y) = \{(x-1, y-1), (x, y-1), (x+1, y-1), (x+1, y), (x, y+1), (x-1, y)\} \tag{2}$$

Cette différenciation permet de gérer correctement le décalage vertical entre colonnes adjacentes dans le système offset.

3.9.3 Distance hexagonale

La distance entre deux hexagones $H_1(x_1, y_1)$ et $H_2(x_2, y_2)$ peut être calculée selon différentes métriques. Pour un graphe non pondéré où chaque arête a un poids unitaire, la distance correspond au nombre minimal de déplacements nécessaires, calculable via la distance de Manhattan adaptée aux hexagones :

$$d_{\text{hex}}(H_1, H_2) = \max\left(|x_2 - x_1|, |y_2 - y_1| + \left\lfloor \frac{|x_2 - x_1|}{2} \right\rfloor\right) \tag{3}$$

Toutefois, dans notre contexte de pathfinding avec Dijkstra, la distance est calculée dynamiquement comme le nombre de sauts (edges) dans le graphe, chaque hexagone étant un nœud et chaque adjacence une arête de poids 1.

3.9.4 Complexité du calcul de portée

L'algorithme récursif `posAdjacentePortee()` explore toutes les positions accessibles dans un rayon r. Pour un rayon r, le nombre maximal de positions explorées suit une progression arithmétique basée sur le nombre de voisins par hexagone (6) :

$$|P(r)| \leq 1 + 6 \sum_{i=1}^{r} i = 1 + 3r(r+1) \tag{4}$$

Cette formule découle du fait que chaque niveau i peut contenir jusqu'à $6i$ nouvelles positions. La complexité temporelle est donc $O(r^2)$ et la complexité spatiale $O(r^2)$ pour stocker toutes les positions trouvées.

3.9.5 Implémentation des voisins

La méthode `posAdjacente()` dans la classe Position implémente le calcul des 6 voisins selon la parité de x :

```
public Position[] posAdjacente() {
    int i = 0;
    Position[] res = new Position[6];

    // Definition des decalages selon parite
    int[][] voisinsImpair = {
        {x-1, y-1}, {x, y-1}, {x+1, y-1},
        {x+1, y}, {x, y+1}, {x-1, y}
    };

    int[][] voisinsPair = {
        {x-1, y}, {x, y-1}, {x+1, y},
        {x+1, y+1}, {x, y+1}, {x-1, y+1}
    };

    // Selection selon parite de la colonne
    int[][] voisins = (x % 2 == 0) ? voisinsPair : voisinsImpair;

    // Construction du tableau de voisins valides
    for (int[] voisin : voisins) {
        Position newPos = new Position(voisin[0], voisin[1]);
        if (newPos.estValide()) {
            res[i] = newPos;
            i++;
        }
    }
    return res;
}
```

Listing 1 – Calcul des voisins hexagonaux

3.9.6 Propriétés géométriques et algorithmiques

La grille hexagonale présente plusieurs avantages algorithmiques par rapport à une grille carrée :

1. **Uniformité des distances** : Tous les 6 voisins sont équidistants, contrairement aux 8 voisins d'une grille carrée (4 adjacents + 4 diagonaux plus éloignés). Cette propriété simplifie les calculs de pathfinding.
2. **Directionnalité réduite** : Le nombre de directions possibles (6) est un compromis entre les 4 directions cardinales (limitées) et les 8 directions d'une grille carrée (plus complexes à gérer).
3. **Isotropie améliorée** : Les distances et chemins sont plus uniformes dans toutes les directions, réduisant les biais directionnels dans les algorithmes de recherche de chemin.

3.9.7 Représentation visuelle et rendu

Le rendu graphique utilise un système de coordonnées pixels avec décalage conditionnel basé sur la parité de la colonne. Pour un hexagone de rayon R (en pixels), la position verticale d'affichage est :

$$y_{\text{pixel}} = y \cdot \sqrt{3} \cdot R + \begin{cases} 0 & \text{si } x \equiv 0 \pmod 2 \\ \frac{\sqrt{3} \cdot R}{2} & \text{si } x \equiv 1 \pmod 2 \end{cases} \tag{5}$$

Cette formule garantit que les hexagones sont correctement imbriqués visuellement, chaque hexagone touchant exactement 6 voisins sans chevauchement ni espace.

3.10 Intelligence artificielle avec Dijkstra

L'IA des monstres utilise l'algorithme de Dijkstra pour calculer le chemin optimal vers les héros détectés, créant un comportement intelligent et prévisible. Cette section présente les fondements théoriques, l'analyse de complexité et les détails d'implémentation de cet algorithme classique de pathfinding.

3.10.1 Formalisation du problème

Le pathfinding sur grille hexagonale peut être modélisé comme un problème de recherche du plus court chemin dans un graphe non orienté $G = (V, E)$ où :

— $V = \{(x, y) \in \mathbb{Z}^2 : 0 \leq x < L, 0 \leq y < H\}$ est l'ensemble des nœuds (positions hexagonales valides)

— $E = \{(u, v) : u, v \in V, u$ et v sont adjacents$\}$ est l'ensemble des arêtes (adjacences hexagonales)

— $w : E \to \mathbb{R}^+$ est la fonction de poids (dans notre cas, $w(e) = 1$ pour toute arête e)

Soit $s \in V$ la position de départ (monstre) et $t \in V$ la position cible (héros). Le problème consiste à trouver un chemin $\pi = (v_0 = s, v_1, \ldots, v_k = t)$ minimisant :

$$w(\pi) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \tag{6}$$

3.10.2 L'algorithme de Dijkstra : théorie et optimalité

L'algorithme de Dijkstra, proposé en 1956 par Edsger W. Dijkstra, est un algorithme glouton qui garantit de trouver le plus court chemin dans un graphe à poids positifs. L'algorithme maintient un ensemble S de nœuds dont la distance minimale depuis s est connue, et une file de priorité Q contenant les nœuds restants.

Invariant principal : Pour tout nœud $v \in S$, $d[v]$ représente la distance minimale réelle depuis s jusqu'à v.

Preuve d'optimalité (résumé) : Supposons qu'il existe un chemin plus court de s à un nœud $u \notin S$. Alors ce chemin doit quitter S à un certain point, disons via l'arête (v, w) où $v \in S$ et $w \notin S$. Mais comme $d[v]$ est optimal et tous les poids sont positifs, l'algorithme aurait déjà exploré cette possibilité, ce qui constitue une contradiction.

3.10.3 Analyse de complexité

Pour notre implémentation utilisant une `PriorityQueue` (tas binaire), la complexité est :

— **Temporelle** : $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$
  — $|V| = L \times H$ opérations d'extraction du minimum (coût $\log |V|$ chacune)
  — $|E| \leq 6|V|$ (chaque hexagone a au plus 6 voisins) opérations de mise à jour de priorité

— **Spatiale** : $O(|V|)$ pour stocker les distances, prédécesseurs et la file de priorité

Dans le pire cas avec $|V| = 450$ (carte 30×15) et $|E| \approx 2700$ (chaque hexagone a en moyenne 6 voisins), l'algorithme effectue environ $450 \times \log_2(450) \approx 3800$ opérations de tas, soit une complexité très raisonnable pour un temps réel dans un jeu au tour par tour.

3.10.4 Adaptations spécifiques au contexte de jeu

Notre implémentation intègre plusieurs adaptations pour le contexte de jeu :

1. **Gestion des obstacles** : Les arêtes vers des hexagones contenant des obstacles sont exclues du graphe (poids effectif ∞).
2. **Évitement des monstres alliés** : Les hexagones occupés par d'autres monstres sont exclus du chemin, évitant les collisions.
3. **Arrêt anticipé** : L'algorithme s'arrête dès que la cible t est atteinte (et non après exploration complète du graphe), optimisant les performances.
4. **Remontée de chemin** : Une fois le chemin optimal trouvé, la méthode `remonterChemin()` reconstitue le chemin complet de t vers s en suivant les prédécesseurs, puis retourne uniquement la première étape depuis s.

3.10.5 Implémentation

Le projet utilise une classe interne `NoeudDijkstra` pour représenter chaque nœud avec sa position, sa distance depuis le départ, et son prédécesseur dans le chemin optimal.

```
public Position dijkstraProchainePas(Position depart, Position cible) {
    Map<String, Integer> distances = new HashMap<>();
    Map<String, Position> precedents = new HashMap<>();

    // File de priorite triee par distance croissante
    PriorityQueue<NoeudDijkstra> file = new PriorityQueue<>(
        Comparator.comparingInt(n -> n.distance)
    );

    // Initialisation
    distances.put(cleDepart, 0);
    file.add(new NoeudDijkstra(depart, 0, null));

    while (!file.isEmpty()) {
        NoeudDijkstra noeudActuel = file.poll();

        if (noeudActuel.position == cible) {
            return remonterChemin(precedents, depart, cible);
        }

        // Explorer les 6 voisins hexagonaux
        for (Position voisin : posActuelle.posAdjacente()) {
            // Calcul du plus court chemin...
        }
    }

    return null;
}
```

Listing 2 – Algorithme de Dijkstra (extrait)

3.10.6 Détails d'implémentation et optimisations

L'implémentation utilise deux structures de données principales pour l'efficacité :

1. **HashMap pour distances et prédécesseurs** : Accès en $O(1)$ amorti pour vérifier et mettre à jour les distances connues.
2. **PriorityQueue (tas binaire)** : Structure optimale pour extraire rapidement le nœud avec la distance minimale, avec complexité $O(\log n)$ pour les opérations d'insertion et d'extraction.

Description algorithmique détaillée :
L'algorithme procède selon les étapes suivantes :

1. **Initialisation** : Créer une file de priorité Q vide, initialiser un dictionnaire $dist[\cdot]$ avec toutes les distances à ∞, et un dictionnaire $pred[\cdot]$ avec tous les prédécesseurs à `null`. Initialiser $dist[s] \leftarrow 0$ et insérer $(s, 0)$ dans Q.
2. **Boucle principale** : Tant que Q n'est pas vide :
   (a) Extraire le nœud u avec la plus petite distance de Q (opération *extract-min*).
   (b) Si $u = t$ (cible atteinte), alors reconstruire le chemin via `remonterChemin()` et retourner la première étape.
   (c) Pour chaque voisin v de u tel que v est valide et non bloqué (pas d'obstacle ni de monstre) :
      i. Calculer $alt \leftarrow dist[u] + w(u, v)$ (dans notre cas, $w(u, v) = 1$).
      ii. Si $alt < dist[v]$ (chemin plus court trouvé), alors :
         A. Mettre à jour $dist[v] \leftarrow alt$ et $pred[v] \leftarrow u$.
         B. Insérer (v, alt) dans Q ou mettre à jour la priorité si v est déjà dans Q.
3. **Résultat** : Si la boucle se termine sans atteindre t, retourner `null` (aucun chemin trouvé).

Cette description formelle capture l'essence de l'algorithme de Dijkstra tout en restant accessible sans nécessiter de package spécialisé pour les algorithmes.

3.10.7 Comparaison avec d'autres algorithmes de pathfinding

Dijkstra a été choisi plutôt que A* pour plusieurs raisons :

— **Simplicité** : Aucune heuristique à concevoir ou calibrer.

— **Optimalité garantie** : Dans un graphe non pondéré, Dijkstra est aussi efficace qu'A* et garantit toujours l'optimalité.

— **Performance suffisante** : Pour des cartes de taille raisonnable (30×15 = 450 nœuds) et un contexte de jeu au tour par tour, la différence de performance avec A* est négligeable.

Si une optimisation future était nécessaire, A* avec une heuristique de distance hexagonale pourrait réduire le nombre de nœuds explorés, mais au prix d'une complexité algorithmique accrue.

Résultat : L'implémentation permet aux monstres de poursuivre efficacement les héros détectés en calculant des chemins optimaux contournant les obstacles, créant une expérience de jeu dynamique et stratégique avec un comportement IA prévisible et performant.

4 Présentation du résultat

4.1 Fonctionnalités principales

4.1.1 Système de jeu

1. **Carte hexagonale personnalisable** avec obstacles variés
   - Taille configurable (défaut : 30×15)
   - Obstacles : Rochers, Forêts, Eau
   - Style de map paramétrable
2. **Deux équipes avec types d'unités variés**
   HÉROS (joueur) :
   - HUMAIN : Équilibré (PV :40, ATK :10, TIR :2, VUE :3)
   - NAIN : Tank (PV :80, ATK :50, TIR :0, VUE :1)
   - ELF : Archer (PV :70, ATK :40, TIR :6, VUE :5)
   - HOBBIT : Éclaireur (PV :20, ATK :20, TIR :2, VUE :3)
   - BASE : Objectif à défendre

   MONSTRES (IA) :
   - TROLL : Boss (PV :100, ATK :30, Récompense :100 or)
   - ORC : Soldat (PV :40, ATK :10, Récompense :30 or)
   - GOBELIN : Éclaireur (PV :20, ATK :5, Récompense :10 or)
   - BASE : Objectif à détruire
3. **Mécanique de combat**
   - Combat au corps à corps si adjacents (dégâts aléatoires 0-max)
   - Combat à distance sinon (dégâts fixes)
   - Riposte si survie et portée suffisante
   - Repos pour regagner 5 PV (max initial)
4. **Système économique**
   - Gain d'or en tuant des monstres
   - Amélioration des héros : 50 or par niveau
   - Bonus de +25% sur toutes les stats + soin complet
   - BASE non améliorable
5. **Brouillard de guerre**
   - Zones non explorées masquées (hexagones gris 70% opacité)
   - Révélation selon portée visuelle des héros
   - Mise à jour dynamique
6. **Intelligence artificielle**
   - Algorithme de Dijkstra pour se rapprocher des héros détectés
   - Détection dans un rayon de 5 hexagones
   - Déplacement aléatoire si aucun héros détecté
7. **Conditions de victoire/défaite**
   - Victoire : Destruction de la base ennemie
   - Défaite : Tous les héros morts OU base détruite

4.2 Interface graphique

L'interface graphique du jeu a été conçue pour offrir une expérience utilisateur intuitive et immersive avec un thème médiéval cohérent.

Composants principaux :

— **Menu principal** : Saisie pseudo, boutons Jouer/Sauvegardes/Paramètres/Historique/Personnages

— **Fenêtre de jeu** : Carte hexagonale avec brouillard de guerre, panneau latéral complet

— **Panneau latéral** : Or, liste des héros (portraits, stats, barres de vie), mini-carte tactique, infos temps réel

— **Menus contextuels** : Pause (ESC), Sauvegarde (4 slots), Écran de fin, Personnages, Historique

Style visuel :

— Thème médiéval avec palette marron foncé (RGB : 80, 50, 30)

— Texte blanc, effets de survol, dialogues customisés

— Musique médiévale d'ambiance

Pour une présentation visuelle détaillée de l'interface, voir Annexe E : Captures d'écran (page 22).

4.3 Système de jeu

4.3.1 Tour de jeu

Phase du joueur :

1. Sélection d'un héros (clic)
2. Affichage des positions accessibles
3. Actions possibles : Déplacement, Attaque, Repos, Amélioration
4. Fin du tour (touche F)

Phase de l'IA :

1. Traitement automatique de tous les monstres
2. Détection des héros (rayon 5)
3. Déplacement intelligent (Dijkstra) ou aléatoire
4. Attaque si héros adjacent
5. Mise à jour de l'affichage

4.3.2 Sauvegarde/Chargement

La sérialisation complète permet de sauvegarder :

— Positions de tous les éléments

— PV actuels de tous les soldats

— Niveau d'amélioration des héros

— Or accumulé

— Temps de jeu et tour actuel

5 Organisation du travail

5.1 Répartition des tâches

Ce projet a été développé individuellement avec les phases suivantes :

Phase	Période	Temps
PHASE 1 - Conception	Semaines 1-2	15h
PHASE 2 - Modèle	Semaines 3-4	30h
PHASE 3 - Interface	Semaines 5-7	35h
PHASE 4 - Fonctionnalités avancées	Semaines 8-9	35h
PHASE 5 - Documentation	Semaine 10	10h
TOTAL	**10 semaines**	**125h**

TABLE 1 – Répartition du temps de développement

6 Ressources utilisées

6.1 Bibliothèques Java

- `java.awt.*` : Interface graphique de base
- `javax.swing.*` : Composants GUI avancés
- `java.io.*` : Sérialisation et fichiers
- `javax.sound.sampled.*` : Gestion audio
- `java.util.*` : Collections et utilitaires

6.2 Documentation et tutoriels

- Documentation officielle Oracle Java SE : https://docs.oracle.com/en/java/javase/17/
- Java Swing Tutorial : https://docs.oracle.com/javase/tutorial/uiswing/
- Cours "Programmation Orientée Objets (POO - Java)" de Mathias Géry, Licence 3 Informatique, Université Jean Monnet, 2025
- OpenClassrooms - Cours Java
- Stack Overflow pour résolution de problèmes spécifiques
- GeeksforGeeks - Dijkstra's Algorithm

6.3 Assets et ressources graphiques

- Carte et personnages : Création personnelle / Génération par IA (GPT-5.1)
- Musique : "Medieval Waltz" (licence Creative Commons)
- Icônes et textures : Open source

7 Conclusion

Le projet **WARGAME Java** démontre la mise en œuvre complète des concepts de Programmation Orientée Objet dans le cadre d'un jeu de stratégie fonctionnel et jouable.

L'application met en pratique de manière approfondie les principes de la POO :

— Héritage et polymorphisme (hiérarchie Element $\rightarrow$ Soldat $\rightarrow$ Heros/Monstre)

— Encapsulation rigoureuse des données

— Interfaces (ISoldat, ICarte) définissant des contrats clairs

— Énumérations enrichies (TypesH, TypesM) avec logique métier

— Exceptions personnalisées et sérialisation

L'architecture MVC adoptée facilite la maintenance et l'évolution du code. Le système de jeu complet (combat, amélioration, économie, IA Dijkstra, brouillard de guerre) offre une expérience stratégique cohérente avec une interface graphique intuitive.

Ce projet a permis de comprendre concrètement comment une bonne conception orientée objet permet d'ajouter des fonctionnalités complexes sans bouleverser l'architecture existante. Les concepts appris seront réutilisables dans de futurs projets de développement.

7.1 Améliorations possibles

En termes de perspectives d'évolution, ce projet pourrait être enrichi avec :

— Un éditeur de niveaux personnalisés

— Un système de compétences actives pour les héros

— Un mode multijoueur

— Des graphismes plus élaborés (particules, animations avancées)

— Un système de progression (campagne, achievements)

— Une IA plus avancée (stratégies coordonnées, retraite tactique)

Annexe A : Diagrammes UML

Le projet contient plusieurs diagrammes UML de complexité croissante pour faciliter la compréhension de l'architecture.

A.1 - Diagramme de classes complet

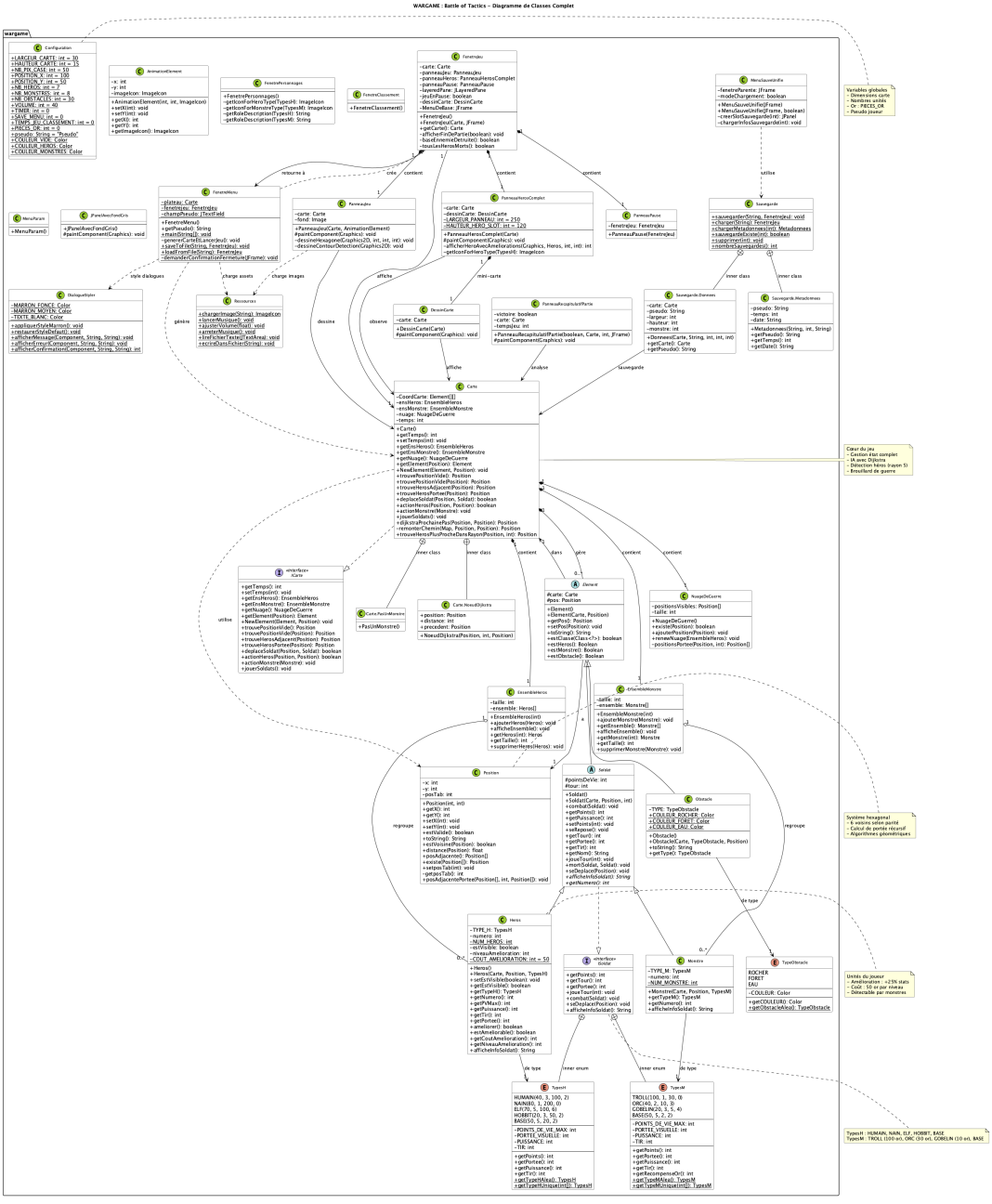

FIGURE 1 – Diagramme UML complet des 29 classes du projet

Note : Ce diagramme est très détaillé. Pour une vue plus synthétique, consulter les sections A.2 et A.3.

A.2 - Diagramme par couches

Ce diagramme présente l'architecture du projet organisée en 4 couches logiques selon le modèle MVC (Modèle-Vue-Contrôleur) :

— **Présentation** : Interface utilisateur (fenêtres, panneaux, menus)

— **Logique métier** : Modèle de jeu (carte, soldats, combat, IA)

— **Données** : État du jeu (position, configuration, obstacles)

— **Services** : Utilitaires (sauvegarde, ressources, gestion de l'or)

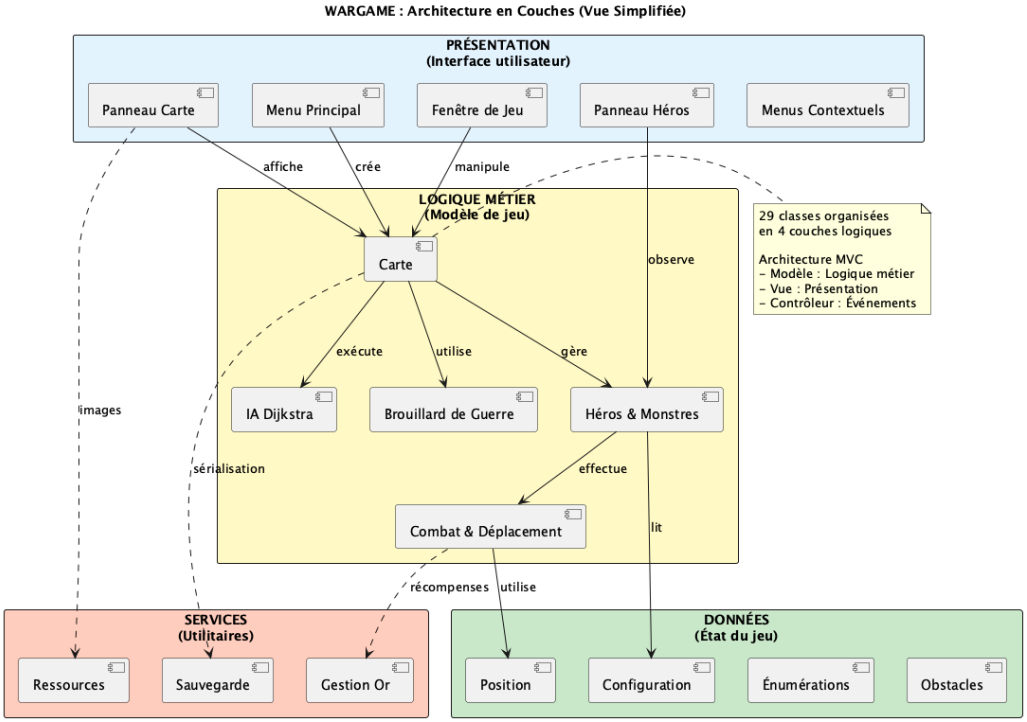

FIGURE 2 – Architecture en couches du projet

Ce diagramme permet de comprendre rapidement l'organisation globale et les flux de communication entre les différentes couches du système.

A.3 - Diagramme d'architecture simplifiée

Ce diagramme présente une vue intermédiaire avec 17 classes principales organisées en 3 packages :

— **MODÈLE** : Logique métier (Element, Soldat, Heros, Monstre, Carte, etc.)

— **VUE** : Interface graphique (fenêtres et panneaux principaux)

— **UTILITAIRES** : Classes de support (Configuration, Ressources, Sauvegarde)

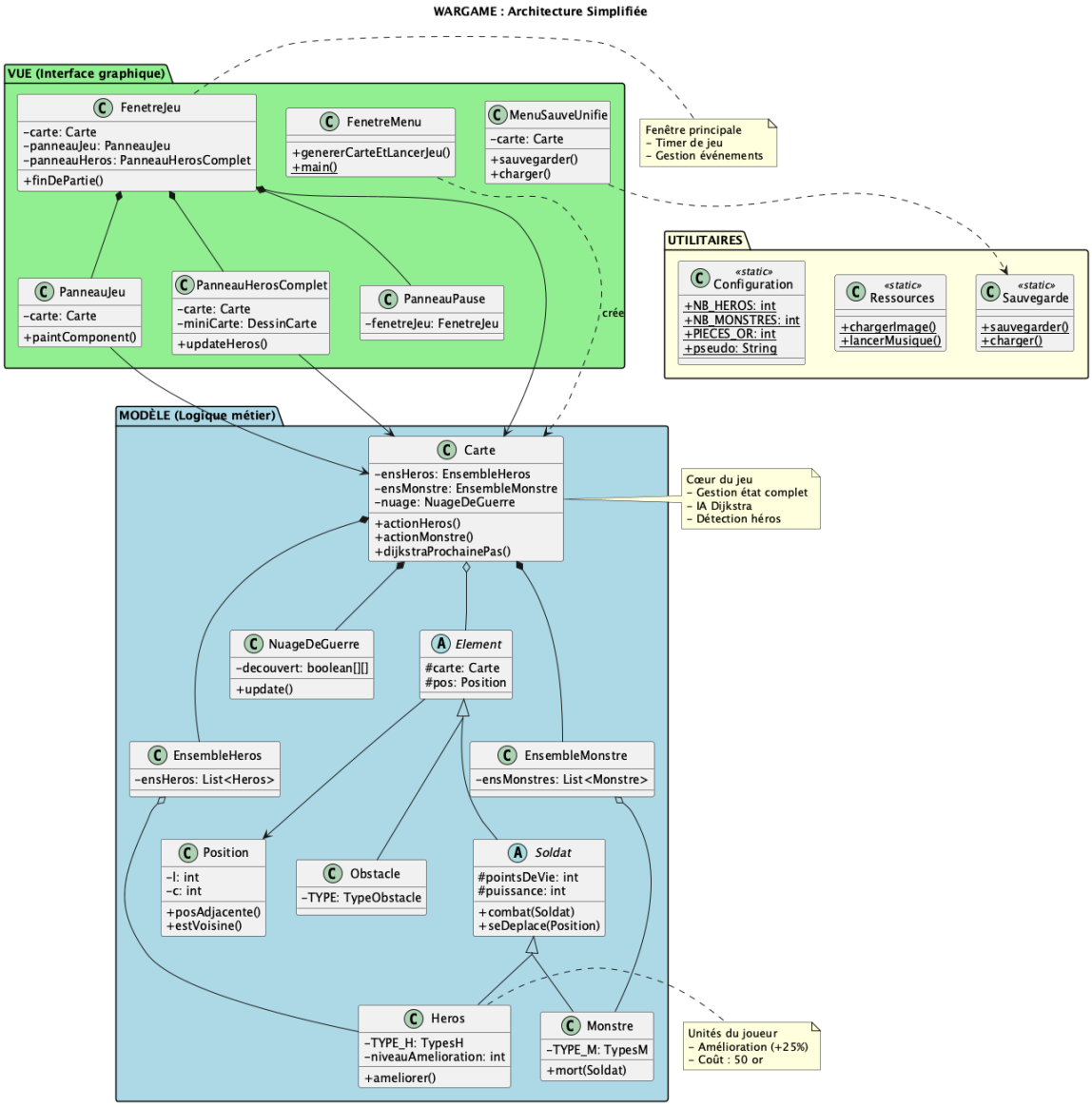

FIGURE 3 – Architecture simplifiée avec les 17 classes principales

Ce diagramme offre un bon équilibre entre lisibilité et exhaustivité, montrant les classes essentielles avec leurs attributs et méthodes clés.

Annexe B : Exemples de code

Cette annexe présente des exemples de code illustrant les techniques de POO/Java décrites dans la section 3.

B.1 - Héritage

```java
public abstract class Soldat extends Element implements ISoldat {
    protected int pointsDeVie;

    public void combat(Soldat soldat) {
        // Logique commune a tous les soldats
    }
}

public class Heros extends Soldat {
    private int niveauAmelioration;

    public boolean ameliorer() {
        // Logique specifique aux heros
    }
}
```

Listing 3 – Exemple d'héritage avec Soldat

B.2 - Encapsulation

```java
public class Heros extends Soldat {
    private final TypesH TYPE_H;  // Immuable
    private int niveauAmelioration = 0;  // Modifiable

    public TypesH getTypeH() { return TYPE_H; }
    public int getNiveauAmelioration() { return niveauAmelioration; }
}
```

Listing 4 – Exemple d'encapsulation

B.3 - Polymorphisme

Polymorphisme d'héritage :

```java
public abstract class Soldat {
    public abstract String afficheInfoSoldat();
}

public class Heros extends Soldat {
    public String afficheInfoSoldat() {
        return TYPE_H + " " + numero + " | PV: " + pointsDeVie;
    }
}
```

Listing 5 – Méthode abstraite redéfinie

Polymorphisme d'interface :

```
public interface ISoldat {
    void combat(Soldat soldat);
    void seDeplace(Position newPos);
}
```

Listing 6 – Implémentation d'interface

Polymorphisme paramétrique :

```
public boolean estClasse(Class<?> c) {
    return this.getClass().equals(c);
}
```

Listing 7 – Utilisation de types génériques

B.4 - Énumérations

```
public enum TypesH {
    HUMAIN(40, 3, 10, 2),
    NAIN(80, 1, 50, 0),
    ELF(70, 5, 40, 6);

    private final int POINTS_DE_VIE_MAX, PORTEE_VISUELLE;

    public static TypesH getTypeHAlea() {
        return values()[(int)(Math.random()*values().length-1)];
    }
}
```

Listing 8 – Énumération TypesH avec attributs et méthodes

B.5 - Exceptions personnalisées

```
public static class PasUnMonstre extends Exception {
    public PasUnMonstre() {
        super("L'element n'est pas un monstre");
    }
}
```

Listing 9 – Exception personnalisée

B.6 - Sérialisation

```
public class Carte implements ICarte, Serializable {
    private static final long serialVersionUID = 1L;
}
```

Listing 10 – Implémentation de Serializable

Annexe C : Captures d'écran de l'interface

Cette annexe présente les captures d'écran de l'interface du jeu. Toutes les images sont disponibles dans le dossier `interface/`.

C.1 - Menu principal

FIGURE 4 – Menu principal avec saisie de pseudo et boutons d'accès

C.2 - Fenêtre de jeu

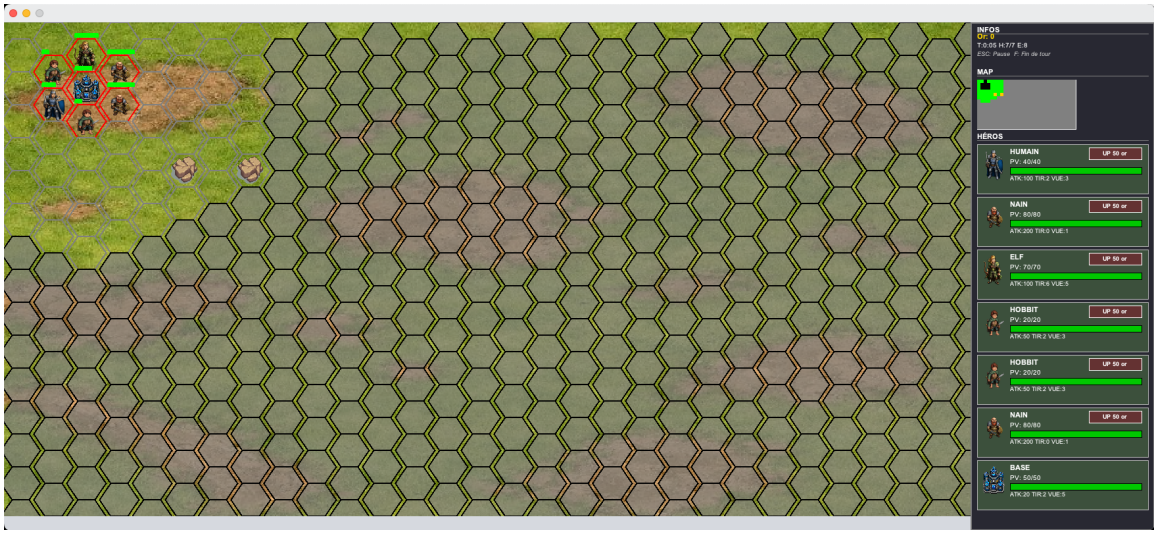

FIGURE 5 – Fenêtre de jeu avec carte hexagonale et panneau latéral

C.3 - Panneau latéral (détail)

FIGURE 6 – Panneau latéral : or, héros, mini-carte, infos

C.4 - Menu pause

FIGURE 7 – Menu pause accessible via ESC

C.5 - Menu sauvegarde

FIGURE 8 – Menu de sauvegarde avec 4 slots

C.6 - Menu chargement

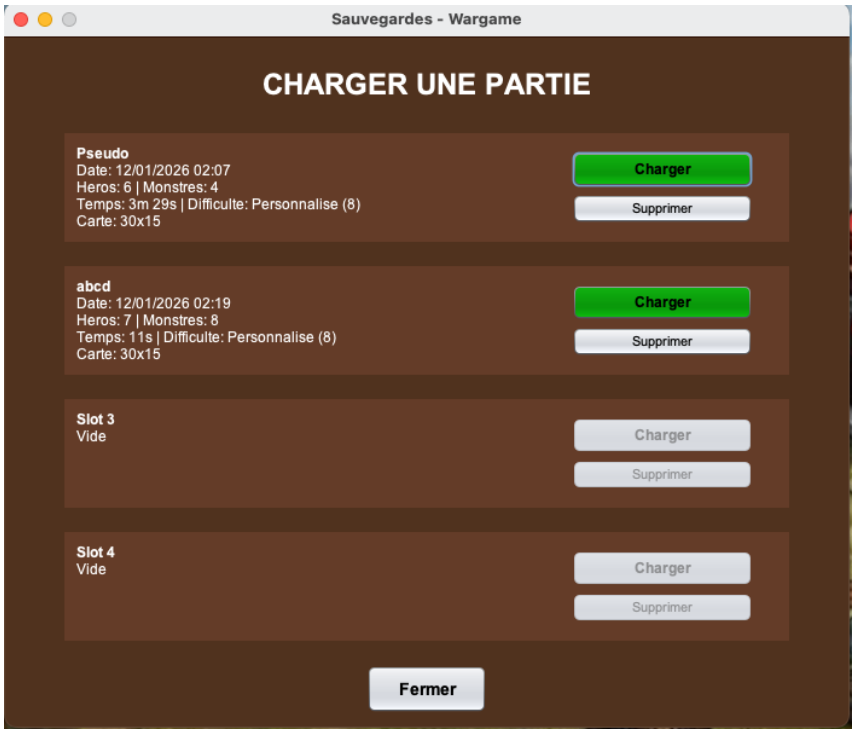

FIGURE 9 – Menu de chargement des parties

C.7 - Menu paramètres

FIGURE 10 – Menu paramètres : volume, difficulté, style de map

C.8 - Fenêtre personnages

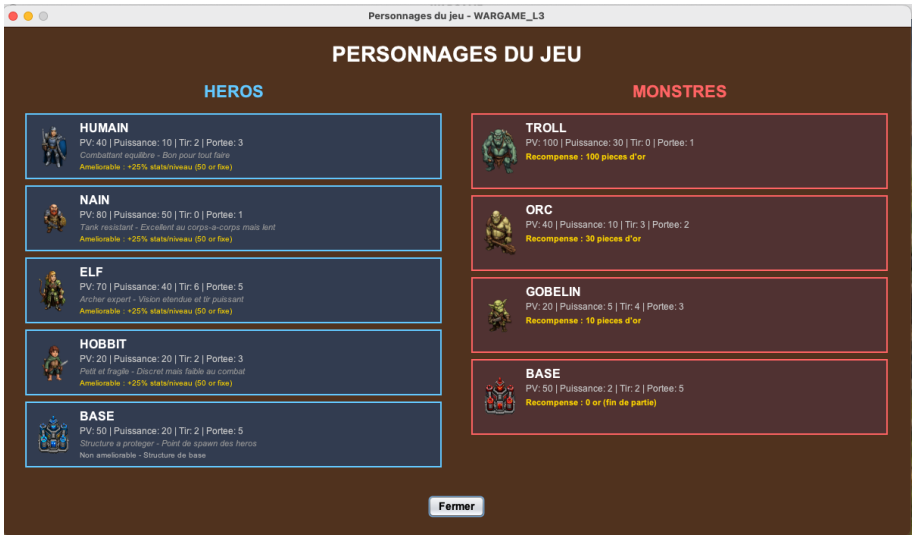

FIGURE 11 – Fenêtre des personnages : caractéristiques des héros et monstres