Université des sciences et de la technologie Houari Boumediene
Faculté d'Informatique

Rapport Technique Détaillé

Jeu de Stratégie en Java

Projet POO - ISIL A G1 2025/2026

Réalisé par :

SOLTANI Mohamed Elamine

Encadré par :

Dr. ABDELLAHOUM Hamza

Version 1.0.0

Année Universitaire 2025/2026

Janvier 2026

Table des matières

Chapitre 1

Introduction

1.1 Contexte du projet

Ce rapport présente l'analyse technique complète du projet de jeu de stratégie développé en Java dans le cadre du module de Programmation Orientée Objet (POO) de l'ISIL pour l'année académique 2025/2026.

1.2 Objectifs

Les objectifs principaux du projet sont :

— Développer un jeu de stratégie fonctionnel et jouable
— Appliquer les concepts fondamentaux de la POO
— Mettre en œuvre une architecture logicielle propre et extensible
— Respecter les principes SOLID
— Utiliser les collections Java de manière appropriée

1.3 Périmètre

Le projet couvre :

— Un système de gestion de carte procédural
— Un système de ressources dynamique
— Un système de construction de bâtiments
— Un système de recrutement et gestion d'unités
— Un système de combat avec formules de dégâts
— Une interface graphique moderne avec Java Swing
— Une IA basique pour l'adversaire

Chapitre 2

Architecture du système

2.1 Vue d'ensemble

2.1.1 Architecture en couches

Le projet suit une architecture en couches modulaire :

1. **Couche présentation (UI)** : Interface graphique Swing
2. **Couche logique métier** : Règles du jeu, combat, gestion
3. **Couche données** : Structures de données et modèles

2.1.2 Diagramme de packages

La structure du projet est organisée en packages fonctionnels :

```
game/
 main/          - Point d'entrée console
 ui/            - Interface graphique
 map/           - Système de carte
 player/        - Gestion des joueurs
 resource/      - Types de ressources
 unit/          - Hiérarchie des unités
 building/      - Hiérarchie des bâtiments
 combat/        - Système de combat
```

2.2 Principes de conception appliqués

2.2.1 Principes SOLID

Single Responsibility Principle (SRP)

Chaque classe a une responsabilité unique et bien définie :

— `GameMap` : Gestion de la carte uniquement

— `CombatSystem` : Logique de combat uniquement

— `Player` : Gestion des ressources et entités du joueur

Open/Closed Principle (OCP)

Les classes sont ouvertes à l'extension, fermées à la modification :

— Les classes abstraites `Unit` et `Building` permettent d'ajouter de nouveaux types sans modifier le code existant

— Les enums `TileType` et `ResourceType` sont extensibles

Liskov Substitution Principle (LSP)

Les sous-classes peuvent remplacer leurs classes parentes :

— Toute instance de `Soldier`, `Archer` ou `Cavalry` peut être utilisée comme `Unit`

— Tous les bâtiments peuvent être manipulés via la classe `Building`

Interface Segregation Principle (ISP)

Bien que Java Swing impose certaines interfaces, nous avons créé des interfaces spécialisées pour nos besoins.

Dependency Inversion Principle (DIP)

Le code dépend d'abstractions, pas d'implémentations concrètes :

— `Player` manipule des `List<Unit>` et `List<Building>`, pas des types concrets

— Le système de combat accepte des `Unit` abstraites

2.2.2 Patrons de conception utilisés

Factory Pattern (implicite)

La création d'unités et bâtiments suit un pattern factory implicite dans les méthodes de recrutement et construction.

Strategy Pattern

Le système de combat utilise différentes stratégies selon le type d'unité (portée, contre-attaque).

Observer Pattern (via Swing)

L'interface graphique observe les événements utilisateur et met à jour l'affichage en conséquence.

Chapitre 3

Conception détaillée

3.1 Modèle de données

3.1.1 Hiérarchie des unités

```
public abstract class Unit {
    protected String name;
    protected Player owner;
    protected int maxHealth;
    protected int currentHealth;
    protected int attack;
    protected int defense;
    protected int range;           // Portee d'attaque
    protected int movementPoints;  // Points de deplacement par tour
    protected int x, y;            // Position sur la carte
    protected boolean hasActed;    // Si l'unite a deja agi ce tour
    protected Map<ResourceType, Integer> cost;

    public Unit(String name, Player owner, int x, int y) {
        this.name = name;
        this.owner = owner;
        this.x = x;
        this.y = y;
        this.hasActed = false;
        this.cost = new HashMap<>();
        initializeStats();
    }

    protected abstract void initializeStats();

    public int attack(Unit target) {
        if (this.hasActed) {
            return 0;
        }
        int damage = Math.max(1, this.attack - target.defense);
        target.takeDamage(damage);
        this.hasActed = true;
        return damage;
    }

    public void takeDamage(int damage) {
        this.currentHealth -= damage;
        if (this.currentHealth <= 0) {
```

```
            this.currentHealth = 0;
            die();
        }
    }

    protected void die() {
        if (owner != null) {
            owner.removeUnit(this);
        }
    }

    public boolean canAttack(Unit target) {
        int distance = distanceTo(target.getX(), target.getY());
        return !this.hasActed && this.isAlive() &&
               target.isAlive() && distance <= this.range &&
               this.owner != target.owner;
    }
}
```

Listing 3.1 – Classe abstraite Unit (extrait)

Implémentations concrètes :

— `Soldier` : Unité défensive équilibrée

— `Archer` : Unité à distance avec faible défense

— `Cavalry` : Unité mobile rapide

3.1.2 Hiérarchie des bâtiments

```
public abstract class Building {
    protected String name;
    protected Player owner;
    protected int x, y;              // Position sur la carte
    protected int maxHealth;
    protected int currentHealth;
    protected int constructionTime;     // Temps de construction (tours)
    protected int remainingTime;        // Temps restant
    protected boolean isBuilt;          // Si le batiment est termine
    protected Map<ResourceType, Integer> cost;
    protected Map<ResourceType, Integer> production; // Production/tour

    public Building(String name, Player owner, int x, int y) {
        this.name = name;
        this.owner = owner;
        this.x = x;
        this.y = y;
        this.isBuilt = false;
        this.cost = new HashMap<>();
        this.production = new HashMap<>();
        initializeStats();
        this.remainingTime = this.constructionTime;
    }

    protected abstract void initializeStats();
    public abstract void performAction();

    public boolean advanceConstruction() {
```

```
29          if (isBuilt) return true;
30
31          remainingTime--;
32          if (remainingTime <= 0) {
33              isBuilt = true;
34              return true;
35          }
36          return false;
37      }
38
39      public void produceResources() {
40          if (!isBuilt || production.isEmpty()) return;
41
42          for (Map.Entry<ResourceType, Integer> entry : production.
    entrySet()) {
43              owner.addResource(entry.getKey(), entry.getValue());
44          }
45      }
46  }
```

Listing 3.2 – Classe abstraite Building (extrait)

Types de bâtiments :

1. `CommandCenter` : Centre de commandement (objectif)
2. `Farm` : Production de nourriture
3. `Mine` : Production de pierre et or
4. `Sawmill` : Production de bois
5. `TrainingCamp` : Recrutement d'unités

3.1.3 Système de ressources

```
1  public enum ResourceType {
2      GOLD("Or", "\uD83D\uDCB0"),
3      WOOD("Bois", "\uD83E\uDEB5"),
4      STONE("Pierre", "\u26CF\uFE0F"),
5      FOOD("Nourriture", "\uD83C\uDF3E");
6
7      private final String displayName;
8      private final String icon;
9
10     ResourceType(String displayName, String icon) {
11         this.displayName = displayName;
12         this.icon = icon;
13     }
14
15     // Getters...
16 }
```

Listing 3.3 – Enum ResourceType

Gestion avec HashMap :

```
1  public class Player {
2      private final String name;
3      private final Map<ResourceType, Integer> resources;
4      private final List<Unit> units;
```

```
    private final List<Building> buildings;

    public Player(String name) {
        this.name = name;
        this.resources = new HashMap<>();
        this.units = new ArrayList<>();
        this.buildings = new ArrayList<>();
        initializeResources();
    }

    private void initializeResources() {
        resources.put(ResourceType.GOLD, 100);
        resources.put(ResourceType.WOOD, 50);
        resources.put(ResourceType.STONE, 50);
        resources.put(ResourceType.FOOD, 100);
    }

    public void addResource(ResourceType type, int amount) {
        int current = resources.getOrDefault(type, 0);
        resources.put(type, current + amount);
    }

    public boolean hasResources(Map<ResourceType, Integer> costs) {
        for (Map.Entry<ResourceType, Integer> entry : costs.entrySet())
    {
            int available = resources.getOrDefault(entry.getKey(), 0);
            if (available < entry.getValue()) {
                return false;
            }
        }
        return true;
    }

    public boolean payResources(Map<ResourceType, Integer> costs) {
        if (!hasResources(costs)) return false;

        for (Map.Entry<ResourceType, Integer> entry : costs.entrySet())
    {
            removeResource(entry.getKey(), entry.getValue());
        }
        return true;
    }
}
```

Listing 3.4 – Gestion des ressources dans Player

3.1.4 Système de carte

```
public enum TileType {
    GRASS("Herbe", new Color(102, 204, 102), true, 0, 1),
    FOREST("For t", new Color(34, 139, 34), true, 0, 1.5),
    MOUNTAIN("Montagne", new Color(139, 137, 137), false, 2, 2),
    WATER("Eau", new Color(65, 105, 225), false, 0, 0),
    DESERT("D sert", new Color(244, 164, 96), false, 0, 1.2);

    private final String name;
    private final Color color;
```

```
    private final boolean buildable;
    private final int defenseBonus;
    private final double movementCost;

    // Constructor et getters...
}
```

Listing 3.5 – Enum TileType

```
public class GameMap {
    private int width;
    private int height;
    private Tile[][] tiles;

    public GameMap(int width, int height) {
        this.width = width;
        this.height = height;
        this.tiles = new Tile[height][width];
        generateMap();
    }

    private void generateMap() {
        Random rand = new Random();
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                TileType type = generateTileType(rand);
                tiles[y][x] = new Tile(x, y, type);
            }
        }
    }

    private TileType generateTileType(Random rand) {
        int val = rand.nextInt(100);
        if (val < 50) return TileType.GRASS;
        else if (val < 70) return TileType.FOREST;
        else if (val < 85) return TileType.MOUNTAIN;
        else if (val < 95) return TileType.WATER;
        else return TileType.DESERT;
    }

    // Autres m thodes...
}
```

Listing 3.6 – Classe GameMap

3.2 Système de combat

3.2.1 Formule de calcul des dégâts

Le système de combat implémente une formule complexe avec plusieurs facteurs :

```
public class CombatSystem {
    private final Random random;
    private final GameMap map;

    public CombatSystem(GameMap map) {
        this.map = map;
```

```
        this.random = new Random();
    }

    public boolean performAttack(Unit attacker, Unit defender) {
        if (!canAttack(attacker, defender)) {
            return false;
        }

        // Calcul des degats de base
        int baseDamage = calculateDamage(attacker, defender);

        // Application du bonus de terrain
        int terrainBonus = getTerrainDefenseBonus(defender);
        int finalDamage = Math.max(1, baseDamage - terrainBonus);

        // Chance de coup critique (10%)
        if (random.nextInt(100) < 10) {
            finalDamage = (int)(finalDamage * 1.5);
            System.out.println("COUP CRITIQUE !");
        }

        // Application des degats
        defender.takeDamage(finalDamage);
        attacker.setHasActed(true);

        // Contre-attaque si le defenseur survit et est a portee
        if (defender.isAlive() && canCounterAttack(defender, attacker))
    {
            performCounterAttack(defender, attacker);
        }

        return true;
    }

    private int calculateDamage(Unit attacker, Unit defender) {
        int baseDamage = attacker.getAttack() - defender.getDefense();

        // Ajout d'un facteur aleatoire (-20% a +20%)
        double randomFactor = 0.8 + (random.nextDouble() * 0.4);
        int damage = (int)(baseDamage * randomFactor);

        return Math.max(1, damage);
    }

    private void performCounterAttack(Unit defender, Unit attacker) {
        int counterDamage = calculateDamage(defender, attacker);
        counterDamage = counterDamage / 2; // 50% des degats
        attacker.takeDamage(counterDamage);
    }
}
```

Listing 3.7 – Classe CombatSystem (extrait principal)

3.2.2 Résultat de combat

Le système de combat inclut également des fonctionnalités avancées :

```
public boolean canAttack(Unit attacker, Unit defender) {
```

```
    if (attacker == null || defender == null) return false;
    if (!attacker.isAlive() || !defender.isAlive()) return false;
    if (attacker.hasActed()) return false;
    if (attacker.getOwner() == defender.getOwner()) return false;

    // Verification de la portee
    int distance = attacker.distanceTo(defender.getX(), defender.getY())
    ;
    if (distance > attacker.getRange()) return false;

    return true;
}

// Simulation de combat pour l'IA
public Unit simulateCombat(Unit unit1, Unit unit2) {
    int health1 = unit1.getCurrentHealth();
    int health2 = unit2.getCurrentHealth();

    while (health1 > 0 && health2 > 0) {
        // Unit1 attaque
        int damage1 = calculateDamage(unit1, unit2);
        health2 -= damage1;

        if (health2 <= 0) return unit1;

        // Unit2 contre-attaque
        int damage2 = calculateDamage(unit2, unit1);
        health1 -= damage2;
    }

    return health1 > 0 ? unit1 : unit2;
}

// Deplacement avec verification
public boolean moveUnit(Unit unit, int targetX, int targetY) {
    if (!map.isValidPosition(targetX, targetY)) return false;

    Tile targetTile = map.getTile(targetX, targetY);
    if (targetTile == null || !targetTile.isAccessible()) return false;

    int distance = map.getDistance(unit.getX(), unit.getY(),
                                   targetX, targetY);
    if (distance > unit.getMovementPoints()) return false;
    if (unit.hasActed()) return false;

    // Deplacement
    unit.moveTo(targetX, targetY);
    return true;
}
```

Listing 3.8 – Simulation et verification de combat

Chapitre 4

Interface utilisateur

4.1 Architecture UI

4.1.1 Composants principaux

1. **ModernMainMenuFrame** : Menu principal
2. **ModernGameFrame** : Fenêtre de jeu principale
3. **ModernGamePanel** : Affichage de la carte
4. **ModernActionPanel** : Panneau d'actions (gauche)
5. **ModernInfoPanel** : Panneau d'informations (droite)
6. **NotificationPanel** : Système de notifications

4.1.2 Design pattern MVC

L'interface suit le pattern Model-View-Controller :

— **Model** : Classes métier (Player, Unit, Building, GameMap)

— **View** : Composants Swing (Panels, Frames)

— **Controller** : Gestionnaires d'événements dans GameFrame

4.1.3 Système de notifications

```
public class NotificationPanel extends JPanel {
    private List<Notification> notifications = new ArrayList<>();
    private Timer animationTimer;

    public NotificationPanel() {
        setLayout(null);
        setOpaque(false);

        // Timer pour animer et supprimer les notifications
        animationTimer = new Timer(50, e -> {
            List<Notification> toRemove = new ArrayList<>();

            for (Notification notif : notifications) {
                notif.update();
                if (notif.shouldRemove()) {
                    toRemove.add(notif);
```

```
                }
            }

            notifications.removeAll(toRemove);
            repaint();
        });
        animationTimer.start();
    }

    public void addInfo(String message) {
        addNotification(message, NotificationType.INFO);
    }

    public void addSuccess(String message) {
        addNotification(message, NotificationType.SUCCESS);
    }

    public void addCombat(String message) {
        addNotification(message, NotificationType.COMBAT);
    }

    private void addNotification(String message, NotificationType type)
    {
        Notification notif = new Notification(message, type);
        notifications.add(notif);
        repositionNotifications();
        repaint();
    }

    enum NotificationType {
        INFO(new Color(33, 150, 243), "INFO"),
        SUCCESS(new Color(76, 175, 80), "OK"),
        WARNING(new Color(255, 152, 0), "ATTENTION"),
        COMBAT(new Color(244, 67, 54), "COMBAT"),
        DEATH(new Color(139, 0, 0), "MORT");

        final Color color;
        final String prefix;
    }
}
```

Listing 4.1 – NotificationPanel avec animations

4.2 Rendu graphique

4.2.1 Affichage de la carte

```
public class GameMap {
    private final int width;
    private final int height;
    private final Tile[][] tiles;
    private final Random random;

    public GameMap(int width, int height) {
        this.width = width;
        this.height = height;
```

```
10          this.tiles = new Tile[height][width];
11          this.random = new Random();
12          generateMap();
13      }
14
15      private void generateMap() {
16          for (int y = 0; y < height; y++) {
17              for (int x = 0; x < width; x++) {
18                  TileType type = generateTileType(x, y);
19                  tiles[y][x] = new Tile(x, y, type);
20              }
21          }
22      }
23
24      private TileType generateTileType(int x, int y) {
25          int rand = random.nextInt(100);
26
27          // 50% Herbe (terrain de base)
28          if (rand < 50) {
29              return TileType.GRASS;
30          }
31          // 20% Foret
32          else if (rand < 70) {
33              return TileType.FOREST;
34          }
35          // 15% Montagne
36          else if (rand < 85) {
37              return TileType.MOUNTAIN;
38          }
39          // 10% Eau
40          else if (rand < 95) {
41              return TileType.WATER;
42          }
43          // 5% Desert
44          else {
45              return TileType.DESERT;
46          }
47      }
48
49      public int getDistance(int x1, int y1, int x2, int y2) {
50          return Math.abs(x2 - x1) + Math.abs(y2 - y1);
51      }
52  }
```

Listing 4.2 – Generation procedurale de la carte

Chapitre 5

IA Ennemie

5.1 Améliorations possibles

5.1.1 IA avancée

— Système de prise de décision avec arbres de comportement
— Évaluation stratégique des positions
— Gestion tactique des formations d'unités
— Planification à long terme des constructions

5.1.2 Pathfinding

— Implémentation de l'algorithme A* pour le déplacement
— Gestion des obstacles et du terrain
— Optimisation des trajets multiples

Chapitre 6

Collections Java

6.1 Utilisation des collections

6.1.1 HashMap pour les ressources

```
private HashMap<ResourceType, Integer> resources;

public Player(String name, int startX, int startY) {
    this.resources = new HashMap<>();
    resources.put(ResourceType.GOLD, 100);
    resources.put(ResourceType.WOOD, 50);
    resources.put(ResourceType.STONE, 50);
    resources.put(ResourceType.FOOD, 50);
}
```

Listing 6.1 – Gestion des ressources

Avantages :

- Accès en O(1) aux ressources
- Typage fort avec enum
- Facilité d'ajout de nouvelles ressources

6.1.2 ArrayList pour les entités

```
private ArrayList<Unit> units;
private ArrayList<Building> buildings;

public void addUnit(Unit unit) {
    units.add(unit);
}

public void removeDeadUnits() {
    units.removeIf(unit -> !unit.isAlive());
}

public List<Unit> getUnits() {
    return Collections.unmodifiableList(units);
}
```

Listing 6.2 – Listes d'unités et bâtiments

Avantages :

— Accès indexé rapide
— Itération efficace
— Support des lambdas et streams
— Collections immuables pour l'encapsulation

6.1.3 List pour les notifications

```
private List<Notification> notifications = new ArrayList<>();

public void updateNotifications() {
    notifications.removeIf(Notification::shouldRemove);
}
```

Listing 6.3 – File de notifications

Chapitre 7

Tests et validation

7.1 Tests fonctionnels

7.1.1 Tests unitaires

Bien que des tests unitaires formels n'aient pas été implémentés avec JUnit, les fonctionnalités ont été testées manuellement :

Fonctionnalité	**Test**	**Résultat**
Construction bâtiment	Coût, placement	OK
Recrutement unité	Coût, prérequis	OK
Déplacement unité	Distance, obstacles	OK
Combat	Dégâts, mort	OK
Production ressources	Valeurs par tour	OK
Victoire/Défaite	Conditions	OK

TABLE 7.1 – Résultats des tests fonctionnels

7.1.2 Tests d'intégration

Les tests d'intégration ont vérifié :

— Interaction entre systèmes (combat + carte)

— Cohérence des ressources

— Synchronisation UI-logique

— Tour par tour complet

7.2 Validation du cahier des charges

Exigence	**Statut**
Carte procédurale avec types de terrain	
Système de ressources (4 types)	
5 types de bâtiments minimum	
3 types d'unités minimum	
Système de combat avec formule	
Interface graphique moderne	
Architecture POO propre	
Collections Java utilisées	
Principes SOLID respectés	
Code organisé en packages	
Mode solo contre IA	

Table 7.2 – Validation des exigences

Chapitre 8

Extensions futures

8.1 Fonctionnalités à ajouter

8.1.1 Priorité haute

— **Sauvegarde/Chargement** : Sérialisation des parties
— **Multijoueur local** : Mode à 2 joueurs sur le même PC
— **Niveaux de difficulté** : Facile, Normal, Difficile pour l'IA
— **Tutorial** : Guide interactif pour nouveaux joueurs

8.1.2 Priorité moyenne

— **Plus d'unités** : Mage, Catapulte, Espion
— **Plus de bâtiments** : Tour de guet, Mur, Temple
— **Technologies** : Arbre de recherche
— **Événements aléatoires** : Tempêtes, découvertes
— **Campagne** : Série de missions avec scénario

8.1.3 Priorité basse

— **Multijoueur en ligne** : Mode réseau
— **Éditeur de carte** : Création de cartes personnalisées
— **Mods** : Support de modifications communautaires
— **Statistiques** : Historique des parties, achievements

8.2 Améliorations techniques

8.2.1 Architecture

— Migration vers pattern MVC plus strict
— Implémentation d'un système d'événements
— Séparation logique/présentation plus nette
— Ajout d'interfaces pour plus de flexibilité

8.2.2 Tests

— Suite de tests unitaires JUnit
— Tests d'intégration automatisés
— Tests de charge (grandes cartes)
— Couverture de code à 80%+

8.2.3 Performance

— Implémentation de A* pour pathfinding
— Optimisation du rendu (dirty rectangles)
— Threading pour l'IA
— Profiling et optimisation mémoire

Chapitre 9

Guide du développeur

9.1 Structure du code

9.1.1 Conventions de nommage

- **Classes** : PascalCase (`GameMap`, `CombatSystem`)
- **Méthodes** : camelCase (`performAction`, `canAfford`)
- **Constantes** : UPPER_SNAKE_CASE (`TILE_SIZE`, `CRIT_CHANCE`)
- **Variables** : camelCase (`selectedUnit`, `resources`)

9.1.2 Organisation des fichiers

```
src/game/
 main/GameLauncher.java
 ui/
    ModernMainMenuFrame.java
    ModernGameFrame.java
    ModernGamePanel.java
    ModernInfoPanel.java
    ModernActionPanel.java
    NotificationPanel.java
 map/
    TileType.java
    Tile.java
    GameMap.java
 player/
    Player.java
 resource/
    ResourceType.java
 unit/
    Unit.java
    Soldier.java
    Archer.java
    Cavalry.java
 building/
    Building.java
```

```
    CommandCenter.java
    Farm.java
    Mine.java
    Sawmill.java
    TrainingCamp.java
 combat/
     CombatSystem.java
```

Chapitre 10

Conclusion

10.1 Objectifs atteints

Le projet a réussi à :

— Implémenter un jeu de stratégie complet et jouable
— Appliquer tous les concepts POO demandés
— Créer une architecture modulaire et extensible
— Respecter les principes SOLID
— Utiliser efficacement les collections Java
— Développer une interface graphique moderne

10.2 Compétences développées

Ce projet a permis de développer :

— Maîtrise approfondie de la POO en Java
— Conception d'architecture logicielle
— Développement d'interfaces graphiques avec Swing
— Gestion de projets complexes
— Résolution de problèmes algorithmiques
— Documentation technique

10.3 Perspectives

Le jeu constitue une base solide pour de nombreuses extensions. L'architecture modulaire permet d'ajouter facilement :

— De nouvelles unités et bâtiments
— Des mécaniques de gameplay supplémentaires
— Un mode multijoueur
— Une campagne narrative

Le code source est organisé, documenté et prêt pour une maintenance et une évolution futures.

Annexe A

Glossaire

POO Programmation Orientée Objet
SOLID Ensemble de 5 principes de conception orientée objet
MVC Model-View-Controller, pattern architectural
Swing Bibliothèque Java pour interfaces graphiques
HashMap Structure de données clé-valeur avec accès en O(1)
ArrayList Liste dynamique avec accès indexé
IA Intelligence Artificielle
GUI Graphical User Interface (Interface Graphique)

Annexe B

Bibliographie

1. Java Documentation - Oracle (https://docs.oracle.com/javase/)
2. Java Swing Tutorial

Annexe C

Annexes

C.1 Commandes de compilation

C.1.1 Compilation complète

```
cd src
javac game/**/*.java
```

C.1.2 Exécution

```
java game.ui.ModernMainMenuFrame
```

C.2 Configuration requise

C.2.1 Système minimum

— OS : Windows 7+, macOS 10.12+, Linux
— Java : JDK 11 ou supérieur
— RAM : 256 MB
— Résolution : 1280×720

C.2.2 Système recommandé

— OS : Windows 10+, macOS 12+, Linux (Ubuntu 20.04+)
— Java : JDK 17 ou supérieur
— RAM : 512 MB
— Résolution : 1920×1080