DOCUMENTATION

Projet 3 Spots Game

GUESBA IYAD GROUPE 111

GHANEMI ABDELHAFID GROUPE 111

Introduction

Le présent document constitue le rapport final du projet de programmation Java, réalisé dans le cadre du cours de développement logiciel du jeu nommé '3 Spots Game' . Ce projet a été entrepris par Abdelhafid Ghanemi et Guesba Iyad, avec pour objectif de concevoir et de mettre en œuvre un jeu de stratégie tour par tour modélisé par un ensemble de classes Java et testé via des tests unitaires.

Le Jeu des 3 Croix se compose d'un petit plateau quadrillé de neuf cases et de trois pions de 2 de long, chacun d'une couleur Bleu ou Rouge, dont l'un est neutre. Le but du jeu est d'atteindre le premier 12 points de victoire tout en s'assurant que l'adversaire atteigne au moins 6 points. Cette condition unique de victoire entraîne une dynamique de jeu où chaque joueur doit constamment évaluer non seulement sa propre progression, mais également celle de son adversaire.

Au cours de la partie, les joueurs sont confrontés à trois questions stratégiques cruciales : comment maximiser leurs points, comment contraindre l'adversaire à marquer des points et dans quelles circonstances il est préférable d'éviter de marquer des points. Ces considérations stratégiques doivent être prises en compte à chaque tour, qui se compose de deux mouvements : le déplacement du pion propre à chaque joueur et celui du pion neutre.

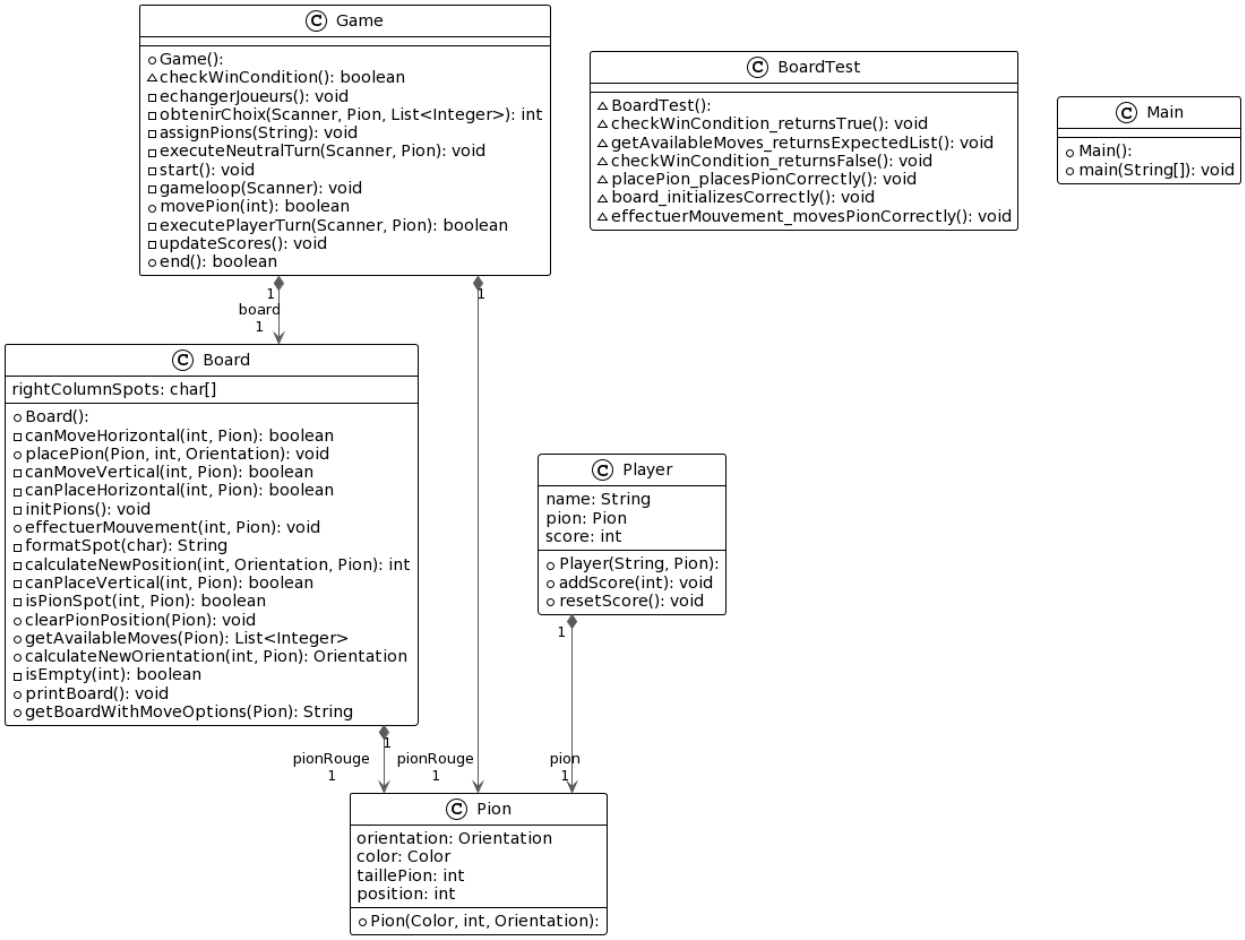
Le déplacement des pions doit recouvrir au moins une nouvelle case à chaque tour, ajoutant ainsi une couche tactique supplémentaire aux décisions prises. La partie se termine quand un joueur atteint le seuil des 12 points.

Ce projet nous a permis de mettre en pratique nos connaissances en programmation orientée objet, notamment l'utilisation de l'encapsulation, ainsi que l'importance des tests unitaires dans le processus de développement logiciel. Nous avons également appris à organiser notre code en paquetages pour une meilleure lisibilité et maintenabilité.

La conception et le développement de ce jeu ont représenté une opportunité d'appliquer des algorithmes de recherche et de résolution de problèmes, ainsi que de gérer les interactions entre l'utilisateur et le système. En dépit des défis rencontrés, le projet a été une expérience d'apprentissage enrichissante, favorisant à la fois le travail d'équipe et l'approfondissement de nos compétences techniques.

Nous vous invitons à parcourir ce dossier pour découvrir les détails de notre réalisation, de la conception initiale à l'implémentation finale, en passant par les phases de test et d'évaluation du produit logiciel.

Diagramme UML



Code Java des Tests Unitaires

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class BoardTest {

    // Teste si la méthode printBoard s'exécute sans lever d'exception
    @Test
    void printBoard_printsCorrectly() {
        Board board = new Board();
        assertDoesNotThrow(() -> board.printBoard());
    }

    // Teste si la méthode getBoardWithMoveOptions retourne les options
correctes
    @Test
    void getBoardWithMoveOptions_returnsCorrectOptions() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        assertDoesNotThrow(() -> board.getBoardWithMoveOptions(pion));
    }

    // Teste si la méthode formatSpot formate correctement les spots
    @Test
    void formatSpot_formatsCorrectly() {
        Board board = new Board();
        assertEquals(" ", board.formatSpot('O'));
        assertEquals("R", board.formatSpot('R'));
        assertEquals("B", board.formatSpot('B'));
        assertEquals("W", board.formatSpot('W'));
    }

    // Teste si la méthode isPionSpot retourne correctement si une case est un
spot de pion
    @Test
    void isPionSpot_returnsCorrectly() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        assertFalse(board.isPionSpot(0, pion));
        assertTrue(board.isPionSpot(1, pion));
    }

    // Teste si la méthode canMoveHorizontal retourne correctement si un pion
peut se déplacer horizontalement
    @Test
    void canMoveHorizontal_returnsCorrectly() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
```

```
        assertFalse(board.canMoveHorizontal(2, pion));
        assertTrue(board.canMoveHorizontal(1, pion));
    }

    // Teste si la méthode canMoveVertical retourne correctement si un pion
peut se déplacer verticalement
    @Test
    void canMoveVertical_returnsCorrectly() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.VERTICAL);
        assertFalse(board.canMoveVertical(0, pion));
        assertTrue(board.canMoveVertical(4, pion));
    }

    // Teste si la méthode placePion place correctement un pion
    @Test
    void placePion_placesPionCorrectly() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        board.placePion(pion, 3, Pion.Orientation.HORIZONTAL);
        assertEquals(3, pion.getPosition());
    }

    // Teste si la méthode effectuerMouvement déplace correctement un pion
    @Test
    void effectuerMouvement_movesPionCorrectly() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        board.effectuerMouvement(0, pion);
        assertEquals(0, pion.getPosition());
    }

    // Teste si la méthode calculateNewOrientation retourne correctement la
nouvelle orientation
    @Test
    void calculateNewOrientation_returnsCorrectOrientation() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        assertEquals(Pion.Orientation.HORIZONTAL,
board.calculateNewOrientation(0, pion));
    }

    // Teste si la méthode calculateNewPosition retourne correctement la
nouvelle position
    @Test
    void calculateNewPosition_returnsCorrectPosition() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        assertEquals(0, board.calculateNewPosition(0,
Pion.Orientation.HORIZONTAL, pion));
    }

    // Teste si la méthode canPlaceHorizontal retourne correctement si un pion
peut être placé horizontalement
    @Test
    void canPlaceHorizontal_returnsCorrectly() {
        Board board = new Board();
```

```
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        assertFalse(board.canPlaceHorizontal(2, pion));
        assertTrue(board.canPlaceHorizontal(1, pion));
    }

    // Teste si la méthode canPlaceVertical retourne correctement si un pion
peut être placé verticalement
    @Test
    void canPlaceVertical_returnsCorrectly() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.VERTICAL);
        assertFalse(board.canPlaceVertical(0, pion));
        assertTrue(board.canPlaceVertical(4, pion));
    }

    // Teste si la méthode isEmpty retourne correctement si une case est vide
    @Test
    void isEmpty_returnsCorrectly() {
        Board board = new Board();
        assertTrue(board.isEmpty(0));
        assertFalse(board.isEmpty(1));
    }

    // Teste si la méthode clearPionPosition efface correctement la position
d'un pion
    @Test
    void clearPionPosition_clearsCorrectly() {
        Board board = new Board();
        Pion pion = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        board.clearPionPosition(pion);
        assertTrue(board.isEmpty(1));
    }
}
```

Certains tests n’ont pas aboutis vous trouverez une explication plus detaillé a la Page 24 lors du bilan du projet !

Code Java Complet du Projet

CLASS « MAIN »

```java
/**
 * Classe principale de l'application.
 */
public class Main {
    /**
     * Point d'entrée de l'application.
     *
     * @param args Les arguments de la ligne de commande.
     */
    public static void main(String[] args) {
        // Création d'une nouvelle instance de jeu
        Game game = new Game();
    }
}
```

CLASS PION

```java
/**
 * Classe représentant un pion dans le jeu.
 */
public class Pion {
    /**
     * Enumération représentant les couleurs possibles d'un pion.
     */
    public enum Color {
        RED('R'),
        BLUE('B'),
        WHITE('W'),
        SPECIAL('O');

        private final char symbol;

        /**
         * Constructeur de l'énumération Color.
         * @param symbol Le symbole représentant la couleur.
         */
        Color(char symbol) {
            this.symbol = symbol;
```

```
        }

        /**
         * Méthode pour obtenir le symbole de la couleur.
         * @return Le symbole de la couleur.
         */
        public char getSymbol() {
            return this.symbol;
        }
    }

    /**
     * Enumération représentant les orientations possibles d'un pion.
     */
    public enum Orientation {
        HORIZONTAL,
        VERTICAL
    }

    private Color color;
    private int position;
    private Orientation orientation;
    private final int taillePion = 2;

    /**
     * Constructeur de la classe Pion.
     * @param color La couleur du pion.
     * @param position La position du pion sur le plateau.
     * @param orientation L'orientation du pion.
     */
    public Pion(Color color, int position, Orientation orientation) {
        this.color = color;
        this.position = position;
        this.orientation = orientation;
    }

    /**
     * Méthode pour obtenir la couleur du pion.
     * @return La couleur du pion.
     */
    public Color getColor() {
        return color;
    }

    /**
     * Méthode pour définir l'orientation du pion.
     * @param orientation La nouvelle orientation du pion.
     */
    public void setOrientation(Orientation orientation) {
        this.orientation = orientation;
    }

    /**
     * Méthode pour obtenir la position du pion.
     * @return La position du pion.
     */
    public int getPosition() {
```

```
        return position;
    }

    /**
     * Méthode pour définir la position du pion.
     * @param position La nouvelle position du pion.
     */
    public void setPosition(int position) {
        this.position = position;
    }

    /**
     * Méthode pour obtenir l'orientation du pion.
     * @return L'orientation du pion.
     */
    public Orientation getOrientation() {
        return orientation;
    }
}
```

CLASS « PLAYER »

```
public class Player {
    private String name;
    private int score;
    private Pion pion;

    public Player(String name, Pion pion) {
        this.name = name;
        this.pion = pion;
        this.score = 0;
    }

    public String getName() {
        return name;
    }

    public int getScore() {
        return score;
    }
```

```
    public Pion getPion() {
        return pion;
    }

    public void addScore(int score) {
        this.score += score;
    }

    public void resetScore() {
        this.score = 0;
    }
}
```

CLASS « BOARD »

```
import java.util.*;

/**
 * Classe représentant le plateau de jeu.
 */
public class Board {
    // Le plateau de jeu est représenté par un tableau de 9 cases. Chaque case
est soit vide (O) soit contient un pion (R, B, W).
    private char[] spots;

    // Initialisation des pions sur le plateau
    private Pion pionRouge, pionBleu, pionBlanc;

    /**
     * Constructeur pour initialiser le plateau de jeu et les pions sur le
plateau.
     */
    public Board() {
        // Initialiser le plateau de jeu avec des cases vides
        this.spots = new char[9];
        // Remplir le tableau avec des cases vides
        Arrays.fill(this.spots, 'O');
        // Initialiser les pions sur le plateau
        initPions();
    }

    /**
     * Initialise les pions sur le plateau de jeu en leurs donnant une
position et une orientation initiale.
     * Utilise la méthode placePion pour placer les pions sur le plateau qui
met à jour la position et l'orientation du pion.
     */
    private void initPions() {
        // Création des pions avec une couleur, une position et une
orientation
```

```
        pionRouge = new Pion(Pion.Color.RED, 1, Pion.Orientation.HORIZONTAL);
        pionBleu = new Pion(Pion.Color.WHITE, 4, Pion.Orientation.HORIZONTAL);
        pionBlanc = new Pion(Pion.Color.BLUE, 7, Pion.Orientation.HORIZONTAL);
        // Placement des pions sur le plateau
        placePion(pionRouge, pionRouge.getPosition(),
pionRouge.getOrientation());
        placePion(pionBlanc, pionBlanc.getPosition(),
pionBlanc.getOrientation());
        placePion(pionBleu, pionBleu.getPosition(),
pionBleu.getOrientation());
    }

    /**
     * Affiche le plateau de jeu.
     */
    public void printBoard() {
        // Nous utilisons une boucle for pour parcourir le tableau de cases et
afficher le plateau de jeu
        // Car le plateau de jeu est représenté par un tableau de 9 cases
        // Un premier for qui affiche ligne par ligne le plateau de jeu
        // premier for pour afficher la ligne complete du plateau de jeu
        // deuxième for pour construire les cases du plateau de jeu
        // nous mettons les valeurs du tableau dans des cases du plateau de
jeu
        System.out.println("* * * * * * * * * * * * *");

        for (int i = 0; i < spots.length; i += 3) {

            System.out.println("*       *       *       *");
            System.out.println("*   " + formatSpot(spots[i]) + "   *   " +
formatSpot(spots[i + 1]) + "   *   " + formatSpot(spots[i + 2]) + "   *");
            System.out.println("*       *       *       *");
            if (i < 6) System.out.println("* * * * * * * * * * * * *");
        }
        System.out.println("* * * * * * * * * * * * *");
    }

    /**
     * Affiche le plateau de jeu avec les options de mouvement.
     * Utilise une liste pour stocker les mouvements possibles que nous
obtenons avec la méthode getAvailableMoves.
     * Parcourt le tableau de cases et affiche le plateau de jeu avec les
options de mouvement.
     * Vérifie si la position est vide ou contient un pion.
     * Si la position est vide nous affichons un espace sinon nous affichons
la valeur de la position.
     * @param pionToMove Le pion à déplacer.
     * @return Une chaîne de caractères représentant le plateau de jeu avec
les options de mouvement.
     */
    public String getBoardWithMoveOptions(Pion pionToMove) {
        List<Integer> availableMoves = getAvailableMoves(pionToMove);
        String[] displaySpots = new String[spots.length];
        for (int i = 0; i < spots.length; i++) {
            if (i == 2 || i == 5 || i == 8) { // Colonnes les plus à droite
                displaySpots[i] = spots[i] == 'O' ? "O" :
String.valueOf(spots[i]);
```

```
            } else {
                displaySpots[i] = spots[i] == 'O' ? " " :
String.valueOf(spots[i]);
            }
        }

        // Nettoyer la position initiale et la position adjacente si le pion
est horizontal.
        int startPosition = pionToMove.getPosition();
        if (pionToMove.getOrientation() == Pion.Orientation.HORIZONTAL &&
startPosition % 3 < 2) {
            displaySpots[startPosition + 1] = displaySpots[startPosition] = "
";
        }

        // Ajouter les options de mouvement avec des numéros directement basés
sur leur ordre dans la liste.
        int moveNumber = 1;
        for (int move : availableMoves) {
            if (move < displaySpots.length) { // Assure que le numéro de
mouvement est dans la plage du tableau
                displaySpots[move] = Integer.toString(moveNumber++);
            }
        }

        // Construction et retour de la chaîne de caractères représentant le
plateau de jeu
        StringBuilder boardDisplay = new StringBuilder();
        boardDisplay.append("* * * * * * * * * * * * * *\n");
        for (int i = 0; i < displaySpots.length; i += 3) {
            boardDisplay.append("*       *       *       *\n"); // Lignes
vides pour la séparation
            boardDisplay.append("*   ")
                    .append(displaySpots[i]).append("   *   ")
                    .append(displaySpots[i+1]).append("   *   ")
                    .append(displaySpots[i+2]).append("   *\n");
            boardDisplay.append("*       *       *       *\n");
            if (i < 6) { // Séparateurs entre les lignes de spots, sauf pour
la dernière ligne
                boardDisplay.append("* * * * * * * * * * * * * *\n");
            }
        }
        boardDisplay.append("* * * * * * * * * * * * * *");

        return boardDisplay.toString();
    }

    /**
     * Formate les cases du plateau de jeu.
     * @param spot La case à formater.
     * @return Une chaîne de caractères représentant la case formatée.
     */
    public String formatSpot(char spot) {
        return spot == 'O' ? " " : String.valueOf(spot);
    }

    /**
```

```
     * Vérifie si la position est un spot de pion.
     * @param position La position à vérifier.
     * @param pion Le pion à vérifier.
     * @return Un booléen indiquant si la position est un spot de pion.
     */
    public boolean isPionSpot(int position, Pion pion) {
        return spots[position] == pion.getColor().getSymbol();
    }

    /**
     * Obtient les spots de la colonne la plus à droite.
     * Utilisé pour mettre à jour les scores et l'affichage du plateau de jeu
'O' pour les spots vides.
     * @return Un tableau de caractères représentant les spots de la colonne
la plus à droite.
     */
    public char[] getRightColumnSpots() {
        return new char[] {spots[2], spots[5], spots[8]};
    }

    /**
     * Obtient les mouvements possibles pour un pion donné.
     * Utilise une liste pour stocker les mouvements possibles.
     * Parcourt le tableau de cases et obtient les mouvements possibles.
     * Utilise les méthodes canMoveHorizontal et canMoveVertical pour vérifier
les mouvements possibles.
     * @param pion Le pion pour lequel obtenir les mouvements possibles.
     * @return Une liste d'entiers représentant les mouvements possibles.
     */
    public List<Integer> getAvailableMoves(Pion pion) {
        List<Integer> availableMoves = new ArrayList<>();

        // Mouvements horizontaux possibles pour des positions qui ne sont pas
en fin de ligne
        for (int i = 0; i < spots.length; i++) {
            if (i % 3 < 2) { // 0 ou 1 ou 3 ou 4 ou 6 ou 7 (donc pas 2, 5, ou
8)
                if (canMoveHorizontal(i, pion)) {
                    availableMoves.add(i);
                }
            }
        }

        // Mouvements verticaux possibles pour des positions qui ne sont pas
dans la première ligne
        for (int i = 3; i < spots.length; i++) {
            if (canMoveVertical(i, pion)) {
                availableMoves.add(i); // La position - 3 représente le
mouvement vertical vers le haut
            }
        }

        return availableMoves;
    }

    /**
     * Vérifie si un pion peut se déplacer horizontalement à une position
```

```
donnée.
     * Exclut les mouvements à partir de la dernière colonne car ils ne sont
pas possibles grâce à la vérification position % 3 == 2.
     * Vérifie si la case actuelle est vide et si la case directement à droite
est vide ou contient le pion actuel.
     * Si la case actuelle est vide et la suivante est vide ou contient le
pion actuel alors le déplacement horizontal est possible.
     * Sinon le déplacement horizontal n'est pas possible.
     * @param position La position à vérifier.
     * @param pion Le pion à vérifier.
     * @return Un booléen indiquant si le pion peut se déplacer
horizontalement à la position donnée.
     */
    public boolean canMoveHorizontal(int position, Pion pion) {
        if (position % 3 == 2) return false; // Empêche de vérifier à
l'extrême droite.

        // Vérifie si la case actuelle est vide et si la case directement à
droite est vide ou contient le pion actuel.
        char currentSpot = spots[position];
        char nextSpot = spots[position + 1];

        // Pour un déplacement horizontal valide, la position actuelle doit
être vide et la suivante doit être vide ou contenir le pion actuel.
        return currentSpot == 'O' && (nextSpot == 'O' || nextSpot ==
pion.getColor().getSymbol());
    }

    /**
     * Méthode pour vérifier si un pion peut se déplacer verticalement à une
position donnée.
     * Nous excluons les mouvements à partir de la première ligne car ils ne
sont pas possibles.
     * Nous utilisons une vérification pour savoir si la position actuelle est
vide et si la case au-dessus est vide ou contient le pion actuel.
     * Si la position actuelle est vide et la case au-dessus est vide ou
contient le pion actuel alors le déplacement vertical est possible.
     * Sinon le déplacement vertical n'est pas possible.
     * @param position La position à vérifier.
     * @param pion Le pion à vérifier.
     * @return Un booléen indiquant si le pion peut se déplacer verticalement
à la position donnée.
     */
    public boolean canMoveVertical(int position, Pion pion) {
        if (position < 3) return false;

        char currentSpot = spots[position];
        char nextSpot = spots[position - 3];

        return currentSpot == 'O' && (nextSpot == 'O' || nextSpot ==
pion.getColor().getSymbol());
    }

    /**
     * Méthode pour placer un pion à une nouvelle position et orientation.
     * Nous utilisons la méthode clearPionPosition pour effacer l'ancienne
position du pion.
```

```
     * Nous utilisons une vérification pour savoir si le placement ne déborde
pas sur une nouvelle ligne.
     * Nous utilisons une vérification pour savoir si le placement ne déborde
pas en bas du plateau.
     * Nous utilisons les méthodes isEmpty et isPionSpot pour vérifier la
validité du placement.
     * Alors nous mettons à jour la position et l'orientation du pion avec les
nouvelles valeurs.
     * @param pion Le pion à placer.
     * @param nouvellePosition La nouvelle position du pion.
     * @param nouvelleOrientation La nouvelle orientation du pion.
     */
    public void placePion(Pion pion, int nouvellePosition, Pion.Orientation
nouvelleOrientation) {
        clearPionPosition(pion);

        int deuxiemeCase;

        if (nouvelleOrientation == Pion.Orientation.HORIZONTAL) {
            if (nouvellePosition % 3 < 2) {
                deuxiemeCase = nouvellePosition + 1;
            } else {
                return;
            }
        } else {
            if (nouvellePosition >= 3) {
                deuxiemeCase = nouvellePosition - 3;
            } else {
                return;
            }
        }
        if ((isEmpty(nouvellePosition) || isPionSpot(nouvellePosition, pion))
&&
                (isEmpty(deuxiemeCase) || isPionSpot(deuxiemeCase, pion))) {
            spots[nouvellePosition] = pion.getColor().getSymbol();
            if (nouvelleOrientation == Pion.Orientation.HORIZONTAL &&
nouvellePosition % 3 < 2) {
                spots[deuxiemeCase] = pion.getColor().getSymbol();
            } else if (nouvelleOrientation == Pion.Orientation.VERTICAL &&
nouvellePosition >= 3) {
                spots[deuxiemeCase] = pion.getColor().getSymbol();
            }
        } else {
            return;
        }
        pion.setPosition(nouvellePosition);
        pion.setOrientation(nouvelleOrientation);
    }

    /**
     * Méthode pour effectuer un mouvement.
     * Nous utilisons les méthodes calculateNewOrientation et
calculateNewPosition pour obtenir la nouvelle position et orientation.
     * Nous utilisons les méthodes canMoveHorizontal et canMoveVertical pour
vérifier la validité du mouvement.
     * Nous utilisons la méthode placePion pour placer le pion à la nouvelle
position et orientation.
```

```
     * @param choix Le choix de mouvement.
     * @param pion Le pion à déplacer.
     */
    public void effectuerMouvement(int choix, Pion pion) {
        Pion.Orientation desiredOrientation = calculateNewOrientation(choix,
pion);
        int nouvellePosition = calculateNewPosition(choix, desiredOrientation,
pion);

        boolean movePossible = (desiredOrientation ==
Pion.Orientation.HORIZONTAL && canMoveHorizontal(nouvellePosition, pion)) ||
                (desiredOrientation == Pion.Orientation.VERTICAL &&
canMoveVertical(nouvellePosition, pion));
        if (movePossible) {
            placePion(pion, nouvellePosition, desiredOrientation);
        }
    }

    /**
     * Méthode pour calculer la nouvelle position basée sur le choix et
l'orientation du pion.
     * Nous utilisons une vérification pour savoir si le choix est valide.
     * Nous utilisons une vérification pour savoir si le choix est horizontal
ou vertical.
     * Nous utilisons les méthodes canPlaceHorizontal et canPlaceVertical pour
vérifier la validité du choix.
     * Nous utilisons la méthode isEmpty pour vérifier si la position est
vide.
     * Nous mettons à jour la nouvelle position avec la valeur du choix.
     * @param choix Le choix de position.
     * @param orientation L'orientation du pion.
     * @param pion Le pion à déplacer.
     * @return La nouvelle position calculée.
     */
    public int calculateNewPosition(int choix, Pion.Orientation orientation,
Pion pion) {
        int newPosition = choix;
        return newPosition;
    }

    /**
     * Méthode pour vérifier si un pion peut être placé horizontalement à une
position donnée.
     * Assurez-vous que la position n'est pas dans la dernière colonne du
plateau et que les deux cases consécutives sont vides ou contiennent le pion
actuel.
     * @param position La position à vérifier.
     * @param pion Le pion à vérifier.
     * @return Un booléen indiquant si le pion peut être placé horizontalement
à la position donnée.
     */
    public boolean canPlaceHorizontal(int position, Pion pion) {
        return position % 3 < 2 && (isEmpty(position) || isPionSpot(position,
pion)) && (isEmpty(position + 1) || isPionSpot(position + 1, pion));
    }

    /**
```

```
     * Méthode pour vérifier si un pion peut être placé verticalement à une
position donnée.
     * @param position La position à vérifier.
     * @param pion Le pion à vérifier.
     * @return Un booléen indiquant si le pion peut être placé verticalement à
la position donnée.
     */
    public boolean canPlaceVertical(int position, Pion pion) {
        return position >= 3 && (isEmpty(position - 3) || isPionSpot(position
- 3, pion)) && (isEmpty(position) || isPionSpot(position, pion));
    }

    /**
     * Méthode pour vérifier si une position est vide.
     * @param position La position à vérifier.
     * @return Un booléen indiquant si la position est vide.
     */
    public boolean isEmpty(int position) {
        return spots[position] == 'O';
    }

    /**
     * Méthode pour effacer la position du pion.
     * Nous utilisons une vérification pour savoir si l'orientation est
horizontale.
     * Nous utilisons une vérification pour savoir si l'orientation est
verticale.
     * Nous utilisons la méthode isEmpty pour vérifier si la position est
vide.
     * Nous mettons à jour la position avec la valeur 'O'.
     * @param pion Le pion dont la position doit être effacée.
     */
    public void clearPionPosition(Pion pion) {
        int position = pion.getPosition();
        spots[position] = 'O';
        if (pion.getOrientation() == Pion.Orientation.HORIZONTAL && position %
3 < 2) {
            spots[position + 1] = 'O';
        } else if (pion.getOrientation() == Pion.Orientation.VERTICAL &&
position >= 3) {
            spots[position - 3] = 'O';
        }
    }

    /**
     * Méthode pour calculer la nouvelle orientation basée sur le choix et
l'orientation du pion.
     * Nous utilisons une vérification pour savoir si le choix est valide.
     * Nous utilisons les méthodes canPlaceHorizontal et canPlaceVertical pour
vérifier la validité du choix.
     * Nous mettons à jour la nouvelle orientation avec la valeur de
l'orientation actuelle du pion.
     * @param choix Le choix d'orientation.
     * @param pion Le pion dont l'orientation doit être calculée.
     * @return La nouvelle orientation calculée.
     */
    public Pion.Orientation calculateNewOrientation(int choix, Pion pion) {
```

```
        boolean horizontalPossible = canPlaceHorizontal(choix, pion);
        boolean verticalPossible = canPlaceVertical(choix, pion);

        Pion.Orientation newOrientation;
        if (horizontalPossible && !verticalPossible) {
            newOrientation = Pion.Orientation.HORIZONTAL;
        } else {
            newOrientation = Pion.Orientation.VERTICAL;
        }

        return newOrientation;
    }


}
```

CLASS « GAME »

```
import java.util.List;
import java.util.Scanner;

/**
 * Classe représentant le jeu.
 */
public class Game {
    private Board board;
    private Pion pionRouge, pionBleu, pionBlanc;
    private Pion pionJoueur1, pionJoueur2;
    private int scoreJoueur1 = 0, scoreJoueur2 = 0;

    /**
     * Constructeur de la classe Game.
     * Initialise le plateau et les pions à leurs positions de départ.
     */
    public Game() {
        this.board = new Board();
        this.pionRouge = new Pion(Pion.Color.RED, 1,
Pion.Orientation.HORIZONTAL);
        this.pionBlanc = new Pion(Pion.Color.WHITE, 4,
Pion.Orientation.HORIZONTAL);
        this.pionBleu = new Pion(Pion.Color.BLUE, 7,
Pion.Orientation.HORIZONTAL);
        start();
    }

    /**
     * Méthode pour démarrer le jeu.
     * Demande aux joueurs de choisir une couleur et lance la boucle de jeu.
     */
```

```java
    public void start() {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Joueur 1, choisissez une couleur (R pour Rouge, B
pour Bleu): ");
        String couleur = scanner.nextLine().toUpperCase();
        while (!couleur.equals("R") && !couleur.equals("B")) {
            System.out.println("Choix de couleur invalide. Veuillez
réessayer.");
            couleur = scanner.nextLine().toUpperCase();
        }
        assignPions(couleur);
        gameloop(scanner);
        scanner.close();
    }

    /**
     * Méthode pour assigner les pions aux joueurs en fonction de leur choix
de couleur.
     * @param couleur La couleur choisie par le joueur 1.
     */
    private void assignPions(String couleur) {
        switch (couleur) {
            case "R":
                this.pionJoueur1 = this.pionRouge;
                this.pionJoueur2 = this.pionBleu;
                break;
            case "B":
                this.pionJoueur1 = this.pionBleu;
                this.pionJoueur2 = this.pionRouge;
                break;
            default:
                System.out.println("Choix de couleur invalide.");
                return;
        }
        this.board.placePion(this.pionJoueur1, this.pionJoueur1.getPosition(),
this.pionJoueur1.getOrientation());
        this.board.placePion(this.pionBlanc, this.pionBlanc.getPosition(),
this.pionBlanc.getOrientation());
        this.board.placePion(this.pionJoueur2, this.pionJoueur2.getPosition(),
this.pionJoueur2.getOrientation());
    }

    /**
     * Méthode représentant la boucle principale du jeu.
     * @param scanner L'objet Scanner utilisé pour lire les entrées des
joueurs.
     */
    private void gameloop(Scanner scanner) {
        boolean gameOver = false;
        while (!gameOver) {
            if (!executePlayerTurn(scanner, pionJoueur1)) {
                gameOver = true;
                continue;
            }
            executeNeutralTurn(scanner, pionBlanc);
            gameOver = checkWinCondition();
            if (gameOver) break;
```

```
            echangerJoueurs();
        }
    }

    /**
     * Méthode pour exécuter le tour d'un joueur.
     * @param scanner L'objet Scanner utilisé pour lire l'entrée du joueur.
     * @param currentPlayerPion Le pion du joueur courant.
     * @return Un booléen indiquant si le tour du joueur a été exécuté avec
succès.
     */
    private boolean executePlayerTurn(Scanner scanner, Pion currentPlayerPion)
{
        System.out.println("Déplacements possibles pour " +
currentPlayerPion.getColor() + ":");
        System.out.println(board.getBoardWithMoveOptions(currentPlayerPion));
        List<Integer> playerMoves =
this.board.getAvailableMoves(currentPlayerPion);
        if (playerMoves.isEmpty()) {
            System.out.println("Aucun déplacement possible pour " +
currentPlayerPion.getColor() + ". Match nul.");
            return false;
        }
        int playerChoice = obtenirChoix(scanner, currentPlayerPion,
playerMoves);
        this.board.effectuerMouvement(playerChoice, currentPlayerPion);
        updateScores();
        return true;
    }

    /**
     * Méthode pour exécuter le tour du pion neutre.
     * @param scanner L'objet Scanner utilisé pour lire l'entrée du joueur.
     * @param neutralPion Le pion neutre.
     */
    private void executeNeutralTurn(Scanner scanner, Pion neutralPion) {
        System.out.println("Déplacements possibles pour le pion neutre :");
        System.out.println(board.getBoardWithMoveOptions(neutralPion));
        List<Integer> neutralMoves =
this.board.getAvailableMoves(neutralPion);
        if (!neutralMoves.isEmpty()) {
            int neutralChoice = obtenirChoix(scanner, neutralPion,
neutralMoves);
            this.board.effectuerMouvement(neutralChoice, neutralPion);
        }
    }

    /**
     * Méthode pour mettre à jour les scores des joueurs.
     */
    private void updateScores() {
        scoreJoueur1 = 0;
        scoreJoueur2 = 0;

        char[] rightColumnSpots = board.getRightColumnSpots();

        for (char spot : rightColumnSpots) {
```

```
            if (spot == pionJoueur1.getColor().getSymbol()) {
                scoreJoueur1++;
            } else if (spot == pionJoueur2.getColor().getSymbol()) {
                scoreJoueur2++;
            }
        }

        System.out.println("Score Joueur 1: " + scoreJoueur1 + " - Score
Joueur 2: " + scoreJoueur2);
        board.printBoard();
    }

    /**
     * Méthode pour obtenir le choix du joueur.
     * @param scanner L'objet Scanner utilisé pour lire l'entrée du joueur.
     * @param pion Le pion du joueur courant.
     * @param mouvementsPossibles La liste des mouvements possibles pour le
pion.
     * @return Le choix du joueur.
     */
    private int obtenirChoix(Scanner scanner, Pion pion, List<Integer>
mouvementsPossibles) {
        int choixIndex = -1;
        do {
            System.out.println("Entrez le numéro de votre choix :");
            if (scanner.hasNextInt()) {
                int choix = scanner.nextInt();
                if (choix > 0 && choix <= mouvementsPossibles.size()) {
                    choixIndex = mouvementsPossibles.get(choix - 1);
                    break;
                }
            } else {
                scanner.next();
            }

        } while (true);

        return choixIndex;
    }

    /**
     * Méthode pour vérifier si les conditions de victoire sont remplies.
     * @return Un booléen indiquant si les conditions de victoire sont
remplies.
     */
    boolean checkWinCondition() {
        if (this.scoreJoueur1 >= 12) {
            if (this.scoreJoueur2 < 6) {
                System.out.println("Le joueur 2 gagne car le joueur 1 a
atteint 12 points mais le joueur 2 a moins de 6 points.");
                return true;
            } else {
                System.out.println("Le joueur 1 gagne.");
                return true;
            }
        } else if (this.scoreJoueur2 >= 12) {
            if (this.scoreJoueur1 < 6) {
```

```
                System.out.println("Le joueur 1 gagne car le joueur 2 a
atteint 12 points mais le joueur 1 a moins de 6 points.");
                return true;
            } else {
                System.out.println("Le joueur 2 gagne.");
                return true;
            }
        }
        return false;
    }

    /**
     * Méthode pour échanger les joueurs.
     */
    private void echangerJoueurs() {
        Pion temp = this.pionJoueur1;
        this.pionJoueur1 = this.pionJoueur2;
        this.pionJoueur2 = temp;
    }

    /**
     * Méthode pour terminer le jeu.
     * @return Un booléen indiquant que le jeu est terminé.
     */
    public boolean end() {
        System.out.println("Fin du jeu.");
        return true;
    }

    /**
     * Méthode pour déplacer un pion.
     * @param choix Le choix de mouvement.
     * @return Un booléen indiquant si le mouvement a été effectué avec
succès.
     */
    public boolean movePion(int choix) {
        return true;
    }
}
```

BILAN DU PROJET

Dans le cadre de notre projet, nous avons eu l'occasion de développer un jeu de plateau exigeant une réflexion approfondie sur la logique de programmation et des décisions de conception cruciales. Tout au long de ce projet, nous avons été confrontés à une série de défis qui ont non seulement testé notre capacité à résoudre des problèmes complexes, mais ont également mis en évidence l'importance d'une planification et d'une exécution méthodiques.

Dès le début, notre ambition était de créer un jeu à la fois simple dans ses règles mais riche en stratégies. Nous avons d'abord opté pour une représentation du plateau de jeu sous forme d'une matrice bidimensionnelle, une approche qui nous semblait intuitive et pratique pour gérer l'espace de jeu. Cette première tentative nous a cependant rapidement confrontés à des limites, notamment en termes de manipulation et d'affichage des mouvements possibles des pièces. La gestion des états et des interactions des pièces dans un tel espace s'est avérée plus complexe que prévu, nous obligeant à reconsidérer notre stratégie de développement.

Face à la complexité croissante de notre code et à la difficulté de maintenir une clarté dans la structure du projet, nous avons pris la décision difficile mais nécessaire de recommencer à zéro. Cette réinitialisation a été l'occasion de repenser notre approche, en mettant un accent particulier sur la sélection judicieuse des structures de données et sur la simplification de la logique de jeu.

Un des enjeux majeurs auquel nous avons dû faire face concernait l'élaboration d'un algorithme efficace pour déterminer les mouvements possibles des pièces sur le plateau. Cette tâche impliquait de tenir compte non seulement des règles spécifiques au mouvement de chaque pièce, mais aussi des interactions potentielles avec les autres pièces, incluant les blocages et les possibilités de capture. La mise au point de cet algorithme a nécessité une compréhension fine des mécanismes du jeu et une capacité à transcrire ces principes en un code précis et performant.

Un autre défi inattendu a émergé lors de notre tentative de finaliser l'affichage du plateau pour des situations où un mouvement peut s'effectuer à la fois horizontalement et verticalement. Initialement, nous avions utilisé une `List<Integer>` pour représenter les mouvements possibles. Néanmoins, pour gérer efficacement les cas de double position, il est apparu essentiel d'adopter une structure de données plus sophistiquée, telle qu'une `HashMap<Integer, String>`. Ce choix nous a permis d'attribuer de manière unique un identifiant de mouvement à chaque case du

plateau. Cependant, malgré cette avancée, des difficultés persistent dans l'affichage, illustrant la complexité de concilier les exigences fonctionnelles et les contraintes techniques.

En conclusion, ce projet a été une expérience d'apprentissage précieuse, nous poussant à repenser nos méthodes de travail et à apprendre de nos erreurs. Bien que nous ayons été contraints de réinitialiser notre projet à plusieurs reprises, ces décisions difficiles nous ont permis de progresser et d'améliorer notre approche. Les difficultés rencontrées, notamment en matière de tests unitaires et de gestion des mouvements des pièces, soulignent l'importance d'une réflexion continue sur la conception et l'optimisation du code. Les enseignements tirés de ce projet, malgré les obstacles, enrichiront sans aucun doute notre parcours académique et professionnel.