

```
//*****
//In_Partie.java
//*****
package partie;

public interface In_Partie {

    // ***** METHODE APPLI *****
    /**
     * permet de lancer une partie doit comprendre deux etapes propres a chaque tour
     * qui eux comprendront 1- joueur un coup 2- evaluer l impact de ce coup sur le
     * score des joueurs
     */
    public void lancerPartie();

    public void jouerCoup(Joueur joueurActuel);

    public void evaluerCoup(Joueur joueur1, Joueur joueur2);

}
```

```

//*****
//PartieTicTacToe.java
//*****
package partie;

import interaction.MessagePlacement;
import interaction.Messages_Saisie;
import jeton.*;
import utilitaires.Utills_Grille_Evaluation_Alignement;

public class PartieTicTacToe extends CA_Grille_Partie {

    private int nbrAlign; // nombre de jetons a aligner
    private int[] saisieCellule;

    public PartieTicTacToe() {
        // taille de 3*3 et pointMax =1 et nbrTourMax=9
        super(3, 3);
        nbrAlign = 3;
    }

    public PartieTicTacToe(int choixGrilleLigne, int choixGrilleColonne) {
        super(choixGrilleLigne, choixGrilleColonne);
        nbrAlign = 3;
    }

    public PartieTicTacToe(int choixGrilleLigne, int choixGrilleColonne, int choixNbrAlignements) {
        super(choixGrilleLigne, choixGrilleColonne);
        nbrAlign = choixNbrAlignements;
    }

    @Override
    public void jouerCoup(Joueur joueurActuel) {
        boolean saisieCorrecte = false;

        while (!saisieCorrecte) {
            saisieCellule = Messages_Saisie.saisirCellule(getGrille());
            System.out.println(Messages_Saisie.afficherMessageCellule(joueurActuel, saisieCellule));
            if (estVideCellule(saisieCellule[0], saisieCellule[1]))
                saisieCorrecte = true;
            else
                System.out.println("La case selectionnee est pleine. Veuillez recommencer.\n");
        }
        placerJeton(joueurActuel.getJeton(), saisieCellule[0], saisieCellule[1]);
        System.out.println(MessagePlacement.afficherMessageCoupJoue(joueurActuel, saisieCellule));
    }

    @Override
    public void evaluerCoup(Joueur joueur1, Joueur joueur2) {
        assert (saisieCellule != null); // on oblige le joueur a avoir jouer un coup

        if (Utills_Grille_Evaluation_Alignement.isAlign(saisieCellule[0], saisieCellule[1], nbrAlign, this)) {

            Jeton jetonEvalue = getCellule(saisieCellule[0], saisieCellule[1]);

            if (jetonEvalue.estEgal(joueur1.getJeton())) {
                joueur1.marquerPoint();
                System.out.println(Messages_Saisie.afficherMessageCoupMarquant(joueur1));
            }

            if (jetonEvalue.estEgal(joueur2.getJeton())) {
                joueur2.marquerPoint();
                System.out.println(Messages_Saisie.afficherMessageCoupMarquant(joueur2));
            }
        }
    }
}

```

```

//*****
//Joueur.java
//*****
package partie;

import jeton.Jeton;

public class Joueur {

    private Jeton jeton;
    private int score;
    private static int compteur = 0; // permet de gérer alternativement un joueur
    au JETON_X puis au JETON_O

    /**
     * constructeur de joueur permet d'associer à un joueur un jeton enum
     */
    public Joueur() {
        this.jeton = Jeton.values()[compteur % 2 + 1]; // x doit commencer et JET
ON_X est le [1]
        this.score = 0;
        ++compteur;
    }

    public Joueur(Jeton jeton) {
        assert (!jeton.estVideJeton());
        this.jeton = jeton;
        this.score = 0;
        ++compteur;
    }

    public Jeton getJeton() {
        return jeton;
    }

    public int getScore() {
        return score;
    }

    public void marquerPoint() {
        ++this.score;
    }
}

```

```

//*****
//PartieForme.java
//*****
package partie;

import interaction.Messages_Saisie;
import jeton.*;
import utilitaires.Utils_Grille_Evaluation_Forme;

public class PartieForme extends PartieTicTacToe {

    private Forme forme;
    private int[] saisieCellule;

    public PartieForme() {
        super(7, 7);
        this.forme = new Forme(1);
        this.saisieCellule = new int[2];
    }

    public PartieForme(int choixForme) {
        super(7, 7);
        this.forme = new Forme(choixForme);
        this.saisieCellule = new int[2];
    }

    public PartieForme(int lignes, int colonnes) {
        super(lignes, colonnes);
        assert (lignes >= 3 && colonnes >= 3);
        this.forme = new Forme(1);
        this.saisieCellule = new int[2];
    }

    @Override
    public void evaluerCoup(Joueur joueur1, Joueur joueur2) {
        assert (saisieCellule != null); // on oblige le joueur a avoir jouer un co
up
        if (Utils_Grille_Evaluation_Forme.estCompleteForme(saisieCellule[0], saisi
eCellule[1], this, forme)) {
            // jetonEvaluate dont on evalue l implication dans un alignement av
ec d'autres
            Jeton jetonEvaluate = getCellule(saisieCellule[0], saisieCellule[1]
);
            if (jetonEvaluate.estEgal(joueur1.getJeton())) {
                joueur1.marquerPoint();
                System.out.println(Messages_Saisie.afficherMessageCoupMar
quant(joueur1));
            }
            if (jetonEvaluate.estEgal(joueur2.getJeton())) {
                joueur2.marquerPoint();
                System.out.println(Messages_Saisie.afficherMessageCoupMar
quant(joueur2));
            }
            //
            int[][] coordAFermer = Utils_Grille_Evaluation_Forme.getCoordForm
eComplete(saisieCellule[0], saisieCellule[1], this, forme);
            //
            ouvertsToFermesJetons(coordAFermer);
            afficherGrille();
        }
    }
}

```

```

//*****
//Forme.java
//*****
package partie;

import jeton.Jeton;

public class Forme {

    // il ne sera dessine que des formes symetriques multiaxes
    private int[] distance;
    private int[] orientation;
    private int[][] chemin;
    private int formeNum;
    private String formeStr;
    private int[][] tabGrilleModele;
    private static String[] listFormesDispo = new String[] { "carre", "losange", "croix" };

    /**
     * choixForme 1 pour Carre - 2 pour losange - 3 pour croix
     *
     * @param choixForme 1 pour Carre - 2 pour losange - 3 pour croix
     */
    public Forme(int choixForme) {
        assert (choixForme != 0 && choixForme <= 3);
        switch (choixForme) {
            // niveau de complexite croissant
            case 1:
                // carre
                formeNum = 1;
                formeStr = "carre";
                distance = new int[] { 1, 1, 1, 1 };
                orientation = new int[] { 2, 4, 6, 0 };
                // {Direction.EST,Direction.SUD,Direction.OUEST,Direction.NORD}

                tabGrilleModele = new int[][] { { 0, 0, 1, 1 }, { 0, 1, 1, 0 } };

                break;

            case 2:
                // losange
                formeNum = 2;
                formeStr = "losange";
                distance = new int[] { 1, 1, 1, 1 };
                orientation = new int[] { 1, 3, 5, 7 };
                // {Direction.NORD_EST,Direction.SUD_EST,Direction.SUD_OUEST,Direction.NORD_OUEST}

                // ;
                tabGrilleModele = new int[][] { { 0, 1, 2, 1 }, { 1, 2, 1, 0 } };

                // {ligne} {colonne}

                break;

            case 3:
                // croix
                formeNum = 3;
                formeStr = "croix";
                distance = new int[] { 1, 1, 1, 1, 1 };
                orientation = new int[] { 1, 3, 5, 7, 6 };
                // {Direction.NORD_EST,Direction.SUD_EST,Direction.SUD_OUEST,Direction.NORD_OUEST},
                tabGrilleModele = new int[][] { { 0, 1, 2, 1, 1 }, { 1, 2, 1, 0, 1 } }; // {ligne} {colonne}

                break;

            // creation de la table chemin avec une profondeur et une direction par ligne
            default:
                assert (orientation.length == distance.length);
                chemin = new int[orientation.length][2];
        }
    }
}

```

```

    for (int i = 0; i < orientation.length; ++i) {
        chemin[i][0] = distance[i];
        chemin[i][1] = orientation[i];
    }

    /**
     * transForme renvoie une forme dont les indices sont decale de int
     * decalageIndice
     *
     * @param decalageIndice
     * @return
     */
    public Forme transForme(int decalageIndice) {
        Forme formeTrans = new Forme(getFormeNum());

        decalageIndice %= getNbrPoint();

        int indiceDecale;

        for (int i = 0; i < getNbrPoint(); ++i) {
            indiceDecale = (decalageIndice + i) % getNbrPoint();
            formeTrans.distance[indiceDecale] = distance[i];
            formeTrans.orientation[indiceDecale] = orientation[i];
        }

        for (int i = 0; i < orientation.length; ++i) {
            formeTrans.chemin[i][0] = formeTrans.distance[i];
            formeTrans.chemin[i][1] = formeTrans.orientation[i];
        }

        return formeTrans;
    }

    public String toStringGrilleModele() {
        PartieTicTacToe partie = new PartieTicTacToe();
        for (int i = 0; i < tabGrilleModele[0].length; ++i) {
            partie.placerJeton(Jeton.JETON_X, tabGrilleModele[0][i], tabGrilleModele[1][i]);
        }

        return partie.toStringGrille();
    }

    public static String[] getListFormesDispo() {
        return listFormesDispo;
    }

    public static String toStringFormeDispo() {
        String sFormeDispo = "";
        sFormeDispo += "Les formes disponibles sont : ";
        int indice = 1;
        for (String forme : listFormesDispo) {
            sFormeDispo += "\n<" + indice + "> " + forme;
            ++indice;
        }
        sFormeDispo += ".\n";
        return sFormeDispo;
    }

    public String toStringConsigne() {
        String sConsigne = "";
        sConsigne += "Pour realiser la forme suivante : " + formeStr
            + "\n il faut placer les jetons de la maniere suivante : \n";

        sConsigne += toStringGrilleModele();
        return sConsigne;
    }

    public static String toStringFormeDispoConsigne() {

```

```

        String formeMenu = "";
        formeMenu += toStringFormeDispo();

        for (int i = 1; i <= listFormesDispo.length; ++i) {
            Forme formeExemple = new Forme(i);
            formeMenu += formeExemple.toStringConsigne();
        }
        return formeMenu;
    }

    public String toStringFormeChoisie() {
        String determinantForme;
        if (formeNum == 1) {
            determinantForme = "1e";
        } else {
            determinantForme = "1a";
        }
        String sChoisie = "";
        sChoisie += "La forme choisie du numero " + formeNum + " est " + determinantForme + " " + formeStr + ".\n";
        return sChoisie;
    }

    public int[] getDistance() {
        return distance;
    }

    public int[] getOrientation() {
        return orientation;
    }

    public int[][] getChemin() {
        return chemin;
    }

    public int getFormeNum() {
        return formeNum;
    }

    public String getFormeStr() {
        return formeStr;
    }

    public int[][] getFormeGrille() {
        return tabGrilleModele;
    }

    public int getNbrPoint() {
        return chemin.length;
    }
}

```

```

//*****
//CA_Grille_Partie.java
//*****
package partie;

import interaction.Messages_Saisie;

public abstract class CA_Grille_Partie extends CA_Grille implements In_Partie {

    private Joueur joueur1;
    private Joueur joueur2;
    private int tour;

    public CA_Grille_Partie(int nbrLignes, int nbrColonnes) {
        super(nbrLignes, nbrColonnes);
        joueur1 = new Joueur();
        joueur2 = new Joueur();
        tour = 0;
    }

    public void lancerPartie() {
        afficherGrille();
        // on fait des tours
        while (!estFinie()) {
            ++tour;
            Joueur joueurActuel = (tour % 2 == 0) ? joueur2 : joueur1;

            System.out.println(Messages_Saisie.afficherMessageDebutTour(joueu
rActuel));

            jouerCoup(joueurActuel);
            afficherGrille();
            evaluerCoup(joueur1, joueur2);

            System.out.println(Messages_Saisie.afficherMessageFinTour(joueurA
ctuel));
        }
        // on compte les points
        System.out.println(Messages_Saisie.afficherMessageResultat(joueur1, joueu
r2));
    }

    public abstract boolean estFinie();

    public abstract void jouerCoup(Joueur joueurActuel);

    public abstract void evaluerCoup(Joueur joueur1, Joueur joueur2);

    // ***** getteurs *****
    public int getTour() {
        return tour;
    }

    public int getScoreJ1() {
        return joueur1.getScore();
    }

    public int getScoreJ2() {
        return joueur2.getScore();
    }

}

```

```

//*****
//CA_Grille_Partie_FermetureJeton.java
//*****
package partie;

import jeton.*;
import utilitaires.Utills_Grille_Evaluation_Adjacent;

public abstract class CA_Grille_Partie_FermetureJeton extends CA_Grille_Partie {

    private boolean[][] grilleOuvertureJetons;

    public CA_Grille_Partie_FermetureJeton(int nbrLignes, int nbrColonnes) {
        super(nbrLignes, nbrColonnes);
        grilleOuvertureJetons = new boolean[nbrLignes][nbrColonnes];
        iniGrilleFermeture();
    }

    // ***** GETTEURS *****

    /**
     * Compte le nombre de jeton ouvert
     *
     * @return
     */
    public int getNbrOuvert() {
        int cpt = 0;
        for (int i = 0; i < grilleOuvertureJetons.length; i++) {
            for (int j = 0; j < grilleOuvertureJetons[0].length; j++) {
                if (grilleOuvertureJetons[i][j]) {
                    ++cpt;
                }
            }
        }
        return cpt;
    }

    // ***** METHODE GRILLE *****

    // ***** METHODE GRILLE PERMUTATION *****
    /**
     * permute deux jetons de la grille verifie que les deux jetons electionnes sont
     * dans la grille verifie que les deux jetons sont adjacents verifie que les les
     * cellules sont rempli de JETON Il n est PAS verifie que les deux JETONS a
     * permuter soient ouverts
     *
     * @param ligne1
     * @param colonne1
     * @param colonne1
     * @param ligne2
     */
    public void permutationJeton(int ligne1, int colonne1, int ligne2, int colonne2)
    {
        assert (sontDifferentes(ligne1, colonne1, ligne2, colonne2)); // les jeto
ns doivent etre differents
        assert (ligne1 < getLignes() && ligne1 >= 0); // la cellule doit Ãatre da
ns la grille
        assert (colonne1 < getColonnes() && colonne1 >= 0); // la cellule doit Ã
tre dans la grille
        assert (ligne2 < getLignes() && ligne2 >= 0); // la cellule doit Ãatre da
ns la grille
        assert (colonne2 < getColonnes() && colonne2 >= 0); // la cellule doit Ã
tre dans la grille

        assert (!estVideCellule(ligne1, colonne1)); // la cellule ne doit pas etr
e vide
        assert (!estVideCellule(ligne2, colonne2)); // la cellule ne doit pas etr
e vide

```

```

        assert (Utills_Grille_Evaluation_Adjacent.sontAdjacents(ligne1, colonne1,
ligne2, colonne2, this));

        // permutation jeton
        if (!getCellule(ligne1, colonne1).estEgal(getCellule(ligne2, colonne2)))
        ) {
            Jeton jtemp = getCellule(ligne1, colonne1);
            placerJeton(getCellule(ligne2, colonne2), ligne1, colonne1);
            placerJeton(jtemp, ligne2, colonne2);
        }
        // permutation ouverture
        if ((estOuvert(ligne1, colonne1) && !estOuvert(ligne2, colonne2))
            || (!estOuvert(ligne1, colonne1) && estOuvert(ligne2, col
onne2))) {
            if (estOuvert(ligne1, colonne1)) {
                ouvertToFermeJeton(ligne1, colonne1);
                fermeToOuvertJeton(ligne2, colonne2);
            } else {
                fermeToOuvertJeton(ligne1, colonne1);
                ouvertToFermeJeton(ligne2, colonne2);
            }
        }
    }

    // ***** METHODE GRILLE Fermeture *****
    // les jetons sont au depart ouvert (true)
    private void iniGrilleFermeture() {
        for (boolean[] ligneJeton : grilleOuvertureJetons) {
            for (int i = 0; i < ligneJeton.length; i++) {
                ligneJeton[i] = true;
            }
        }
    }

    /**
     * ferme le jeton (ligne, colonne) fournie
     *
     * @param ligne
     * @param colonne
     */
    public void ouvertToFermeJeton(int ligne, int colonne) {
        assert (ligne < this.grilleOuvertureJetons.length && ligne >= 0); // la c
ellule doit Ãatre dans la grille
        assert (colonne < this.grilleOuvertureJetons[0].length && colonne >= 0);
// la cellule doit Ãatre dans la grille
        assert (grilleOuvertureJetons[ligne][colonne]); // le jeton doit etre ini
tialement ouvert
        assert (!estVideCellule(ligne, colonne));
        grilleOuvertureJetons[ligne][colonne] = false;
    }

    /**
     * ouvre le jeton (ligne, colonne) fournie
     *
     * @param ligne
     * @param colonne
     */
    private void fermeToOuvertJeton(int ligne, int colonne) {
        assert (ligne < this.grilleOuvertureJetons.length && ligne >= 0); // la c
ellule doit Ãatre dans la grille
        assert (colonne < this.grilleOuvertureJetons[0].length && colonne >= 0);
// la cellule doit Ãatre dans la grille
        assert (!grilleOuvertureJetons[ligne][colonne]); // le jeton doit etre in
itialement ferme
        assert (!estVideCellule(ligne, colonne));
        grilleOuvertureJetons[ligne][colonne] = true;
    }

    /**

```



```

    * ferme la table de jeton (ligne, colonne) fournie
    *
    * @param coordCible
    */
    public void ouvertsToFermesJetons(int[][] coordCible) {
        assert (coordCible != null);
        for (int i = 0; i < coordCible.length; i++) {
            ouvertToFermeJeton(coordCible[i][0], coordCible[i][1]);
        }
    }

    public boolean estOuvert(int ligne, int colonne) {
        assert (ligne < this.grilleOuvertureJetons.length && ligne >= 0); // la c
        // la cellule doit être dans la grille
        assert (colonne < this.grilleOuvertureJetons[0].length && colonne >= 0);
        // la cellule doit être dans la grille
        return grilleOuvertureJetons[ligne][colonne];
    }

    // ***** METHODE GRILLE AFFICHAGE *****

    /**
     * toStringGrille avec les jetons fermes
     *
     * @return une chaine de caractères contenant l'état de la grille
     */
    public String toStringGrilleFerme() {
        String sGrille = " "; // decalage pour les noms de lignes en dizaines

        // ligne des indices de colonnes
        for (int j = 1; j <= getColonnes(); ++j)
            if (j < 10) {
                sGrille += " " + " " + " " + j;
            } else {
                sGrille += " " + " " + j;
            }
        sGrille += "\n";

        // il faut d'abord parcourir les reference de ligne de jeton pour acceder
        // jetons
        for (int ligne = 1; ligne <= getLignes(); ++ligne) {
            if (ligne < 10) {
                sGrille += " " + ligne;
            } else {
                sGrille += ligne;
            }
            for (int colonne = 0; colonne < getColonnes(); ++colonne) {
                sGrille += " " + toStringJetonOuF((ligne - 1), colonne);
            }
            sGrille += "\n";
        }
        return sGrille;
    }

    /**
     * Affiche en system out la String du ToString avec les jetons fermes
     */
    @Override
    public void afficherGrille() {
        System.out.println(this.toStringGrilleFerme());
    }

    /**
     * renvoie l equivalent du symbole ferme pour le jeton donne
     */
    public Character getSymboleJetonFerme(Jeton jeton) {
        if (jeton == Jeton.JETON_X) {
            return 'x';
        }
    }

```

```

    }
    if (jeton == Jeton.JETON_O) {
        return 'o';
    } else {
        return ' ';
    }
}

/**
 * renvoie le symbole d un jeton ouvert ou ferme en fonction de la table
 * grilleOuvertureJetons qui comprend tous les jetons fermes
 *
 * @param ligne
 * @param colonne
 * @return
 */
public Character getSymboleJetonOuF(int ligne, int colonne) {
    assert (ligne < this.grilleOuvertureJetons.length && ligne >= 0); // la c
    // la cellule doit être dans la grille
    assert (colonne < this.grilleOuvertureJetons[0].length && colonne >= 0);
    // la cellule doit être dans la grille

    if (estOuvert(ligne, colonne)) {
        return getCellule(ligne, colonne).getSymbole();
    } else {
        return getSymboleJetonFerme(getCellule(ligne, colonne));
    }
}

public String toStringJetonOuF(int ligne, int colonne) {
    return "" + '[' + getSymboleJetonOuF(ligne, colonne) + ']; // "" shortc
    // ut to cast from char to string
}
}

```

```

//*****
//PartieMorpion.java
//*****
package partie;

import java.util.EnumSet;

import direction.Direction;
import interaction.MessagePlacement;
import interaction.Messages_Saisie;
import jeton.*;
import utilitaires.Utills_Grille_Evaluation_Adjacent;
import utilitaires.Utills_Grille_Evaluation_Alignement;

public class PartieMorpion extends CA_Grille_Partie_FermetureJeton {

    private int nbrAlign;
    private int[] saisieCellule;

    public PartieMorpion(int nbrLignes, int nbrColonnes, int nbrAlign) {
        super(nbrLignes, nbrColonnes);
        this.saisieCellule = new int[2];
        this.nbrAlign = nbrAlign;
        int choixNbrAlignMax = (nbrColonnes >= nbrLignes) ? nbrLignes : nbrColonnes;

        assert (nbrAlign <= choixNbrAlignMax); // ce nombre ne doit pas être plus grand que le nombre de colonnes ou de lignes de votre grille
    }

    public PartieMorpion(int nbrLignes, int nbrColonnes) {
        super(nbrLignes, nbrColonnes);
        this.saisieCellule = new int[2];
        this.nbrAlign = 3;
        int choixNbrAlignMax = (nbrColonnes >= nbrLignes) ? nbrLignes : nbrColonnes;

        assert (nbrAlign <= choixNbrAlignMax); // ce nombre ne doit pas être plus grand que le nombre de colonnes ou de lignes de votre grille
    }

    public PartieMorpion() {
        super(5, 6);
        this.saisieCellule = new int[2];
        this.nbrAlign = 3;
    }

    // ***** METHODE CA *****

    @Override
    public void jouerCoup(Joueur joueurActuel) {
        boolean saisieCorrecte = false;

        while (!saisieCorrecte) {
            saisieCellule = Messages_Saisie.saisirCellule(getGrille());
            System.out.println(Messages_Saisie.afficherMessageCellule(joueurActuel, saisieCellule));
            if (estVideCellule(saisieCellule[0], saisieCellule[1])) {

                if (estVideGrille()) {
                    saisieCorrecte = true;
                } else {
                    if (Utills_Grille_Evaluation_Adjacent.existeAdjacent(saisieCellule[0], saisieCellule[1], this)) {
                        saisieCorrecte = true;
                    } else
                        System.out.println(

```

```

"La case selectionnee ne
comporte pas de jeton adjacent. Veuillez recommencer.\n");
        }
    } else
        System.out.println("La case selectionnee est pleine. Veuillez recommencer.\n");
    }
    placerJeton(joueurActuel.getJeton(), saisieCellule[0], saisieCellule[1]);
    System.out.println(MessagePlacement.afficherMessageCoupJoue(joueurActuel,
saisieCellule));
}

@Override
public void evaluerCoup(Joueur joueur1, Joueur joueur2) {
    evaluerCoupAlignOuvert(joueur1, joueur2, this.saisieCellule);
}

@Override
public boolean estFinie() {
    return estPleineGrille();
}

// ***** EN AVAL DU COUP GAGNANT *****

/**
 * fermer les jeton selon un axe (continue) de longueur profondeur d orientation
 * suivant oneDirection mais ne ferme pas le jeton de depart (coord
 * ligne,colonne) Ne continue de fermer que si les jetons evalues ne sont pas
 * vide ne sont pas deja ferme sont les memes (axe continue) et renvoie le
 * nombre de fermeture de jetons realisees
 */
public int fermerAxeJetons1D(int ligne, int colonne, int profondeur, Direction direction) {
    assert (ligne < getLignes() && ligne >= 0); // la cellule doit être dans la grille
    assert (colonne < getColonnes() && colonne >= 0); // la cellule doit être dans la grille
    assert (profondeur >= 1);

    int nbrJetonFermes = 0;
    int coeffProfondeur = 1;
    boolean valide = true;
    while (coeffProfondeur <= profondeur && this.existeNextCellule(ligne, colonne, coeffProfondeur, direction)
        && valide) {
        int[] coordCible = coordNextJeton(ligne, colonne, coeffProfondeur, direction);

        int ligneCible = coordCible[0];
        int colonneCible = coordCible[1];
        if (getSymboleJetonOuf(ligneCible, colonneCible) == getSymboleJetonOuf(ligne, colonne)) {
            ouvertToFermeJeton(ligneCible, colonneCible);
            ++nbrJetonFermes;
        } else {
            valide = false;
        }
        ++coeffProfondeur;
    }
    return nbrJetonFermes;
}

/**

```

```

* ferme des jetons aprÃs ils ont ete trouves dans un alignement, ferme d abord
* dans une direction (nord au sud sens horaire) puis si le nombre de jeton a
* fermer n a pas ete atteint ferme des jetons dans la direction opposÃe (nord
* au sud sens anti horaire) il faut que le jeton evalue soit ouvert
*
* @param ligne      du jeton model a fermer
* @param colonne    du jeton model a fermer
* @param profondeur nombre de jetons que l on souhaite fermer (qui sont
*                  implique dans un alignement)
*/
public void fermeAlignementXD(int ligne, int colonne, int profondeur) {
    assert (ligne < getLignes() && ligne >= 0); // la cellule doit Ãtre dans
la grille
    assert (colonne < getColonnes() && colonne >= 0); // la cellule doit Ãtr
e dans la grille
    assert (!estVideCellule(ligne, colonne)); // la cellule evaluÃe ne doit
pas etre vide
    assert (profondeur >= 2);

    assert (isDirectAvecAlignOouF(ligne, colonne, profondeur)); // on s assure
qu il y ait deja un alignement
    assert (estOuvert(ligne, colonne)); // il faut que le jeton evalue soit ou
vert sinon on va tenter de fermer des
/
    Direction direction = getAllDirectAlignOouF(ligne, colonne, profondeur)[0
]; // axe dans lequel la fermeture va

// se realiser
    assert (direction != null);

    int resteJeton = profondeur - 1; // compte le nombre de jetons qu il rest
e a fermer il faut que le dernier jeton

    // fermeture d un premier sens de la direction
    resteJeton -= fermerAxeJetons1D(ligne, colonne, profondeur, direction);
    // fermeture du sens oppose de la direction
    if (resteJeton >= 1) {
        resteJeton -= fermerAxeJetons1D(ligne, colonne, resteJeton, direc
tion.inverser());
    }
    assert (resteJeton == 0);

    // fermeture du premier jeton en dernier car il sert de modele a
    // fermerAxeJetons1D
    ouvertToFermeJeton(ligne, colonne);
}

// ***** EVALUATION ALIGNEMENT OUVERT OU FERME *****

/**
* renvoie une chaine de symbole de jetons OUVERT OU FERME obtenus dans une
* direction donnee de taille inferieure ou egale a la profondeur (tant que la
* projection est dans la grille) a partir d une case de la grille (ligne,
* colonne) Attention la case de depart n est pas comprise dans la chaine
*
* @param ligne
* @param colonne
* @param profondeur
* @param direction
* @param grille
* @return
*/
public String getLigneJetonOouF(int ligne, int colonne, int profondeur, Direction
direction) {
    assert (ligne < this.getLignes() && ligne >= 0); // la cellule doit Ãtre
dans la grille
    assert (colonne < this.getColonnes() && colonne >= 0); // la cellule doit

```

```

Ãtre dans la grille
    assert (profondeur > 0);

    String aligneCible = "";
    int coeffProfondeur = 1;

    while (coeffProfondeur <= profondeur && this.existeNextCellule(ligne, col
onne, coeffProfondeur, direction)) {
        int colonneCible = coeffProfondeur * direction.getDcolonne() + co
lonne;

        int ligneCible = coeffProfondeur * direction.getDligne() + ligne;
        aligneCible += getSymboleJetonOouF(ligneCible, colonneCible);
        ++coeffProfondeur;
    }
    return aligneCible;
}

/**
* alignement OUVERT OU FERME pour UNE Direction donnee ET son Inversee
*
* @param ligne      de la cellule observÃe
* @param colonne    de la cellule observÃe
* @param profondeur est le nombre de cellule observÃes au max qui sont alignÃe
s
*
* dans grille doit etre >=2
* @param direction et direction opposÃe vers laquelle observer un alignement
* @return si un alignement a ÃtÃ trouvÃ
*/
public boolean appartientAlignOouF(int ligne, int colonne, int profondeur, Direct
ion direction) {
    assert (ligne < getLignes() && ligne >= 0); // la cellule doit Ãtre dans
la grille
    assert (colonne < getColonnes() && colonne >= 0); // la cellule doit Ãtr
e dans la grille
    assert (!estVideCellule(ligne, colonne)); // la cellule evaluÃe ne doit
pas etre vide
    assert (profondeur >= 2);

    String aligneEvalue = "";
    for (int i = 1; i <= profondeur; ++i) {
        aligneEvalue += getSymboleJetonOouF(ligne, colonne);
    }

    String aligneCible = "";
    String inverse = getLigneJetonOouF(ligne, colonne, profondeur, direction.
inverser());

    inverse = new StringBuilder(inverse).reverse().toString();
    aligneCible += inverse;
    aligneCible += getSymboleJetonOouF(ligne, colonne);
    aligneCible += getLigneJetonOouF(ligne, colonne, profondeur, direction);

    return aligneCible.contains(aligneEvalue);
}

/**
* alignement OUVERT OU FERME pour TOUTES les Directions disponibles le nombre
* de direction pour laquelle un alignement a ete trouvÃ
*
* @param ligne
* @param colonne
* @param profondeur
* @return le nombre de direction/orientation qui ont ÃtÃ trouvÃs avec
*         alignementCellule dans toutes les directions
*/
public int nbrDirectAvecAlignOouF(int ligne, int colonne, int profondeur) {
    assert (ligne < getLignes() && ligne >= 0); // la cellule doit Ãtre dans
la grille
    assert (colonne < getColonnes() && colonne >= 0); // la cellule doit Ãtr
e dans la grille

```

```

        assert (!estVideCellule(ligne, colonne)); // la cellule évaluée ne doit
pas etre vide
        assert (profondeur >= 2);

        int alignement = 0;

        for (Direction oneDirection : EnumSet.range(Direction.NORD, Direction.SUD
_EST)) {
            if (appartientAlignOouF(ligne, colonne, profondeur, oneDirection)
) {
                ++alignement;
            }
        }
        return alignement;
    }

    /**
     * existe t il une direction pour laquelle un alignement OUVERT OU FERME de
     * taille profondeur a ete trouve ?
     *
     * @param ligne
     * @param colonne
     * @param profondeur
     * @param grille
     * @return
     */
    public boolean isDirectAvecAlignOouF(int ligne, int colonne, int profondeur) {
        assert (ligne < getLignes() && ligne >= 0); // la cellule doit être dans
la grille
        assert (colonne < getColonnes() && colonne >= 0); // la cellule doit être
e dans la grille
        assert (!estVideCellule(ligne, colonne)); // la cellule évaluée ne doit
pas etre vide
        assert (profondeur >= 2);

        for (Direction oneDirection : EnumSet.range(Direction.NORD, Direction.SUD
_EST)) {
            if (appartientAlignOouF(ligne, colonne, profondeur, oneDirection)
) {
                return true;
            }
        }
        return false;
    }

    /**
     * AVANT appel de cette fonction il devra avoir ete verifie qu il avait des
     * alignements renvoie les directions (droites et inverses) pour lesquelles un
     * alignement OUVERT a ete trouve
     *
     * @param ligne
     * @param colonne
     * @param profondeur
     * @return table des directions pour lesquelles un alignement a ete trouve
     */
    public Direction[] getAllDirectAlignOouF(int ligne, int colonne, int profondeur)
    {
        assert (ligne < getLignes() && ligne >= 0); // la cellule doit être dans
la grille
        assert (colonne < getColonnes() && colonne >= 0); // la cellule doit être
e dans la grille
        assert (!estVideCellule(ligne, colonne)); // la cellule évaluée ne doit
pas etre vide
        assert (profondeur >= 2);
        assert (isDirectAvecAlignOouF(ligne, colonne, profondeur));

        Direction[] tableDirect = new Direction[nbrDirectAvecAlignOouF(ligne, col
onne, profondeur)];
        int indice = 0;

```

```

T))
        for (Direction direction : EnumSet.range(Direction.NORD, Direction.SUD_ES
        if (appartientAlignOouF(ligne, colonne, profondeur, direction)) {
            tableDirect[indice] = direction;
            ++indice;
        }

        return tableDirect;
    }

    // ***** COORDONNEES A FERMEES *****

    /**
     * donne la longueur de l axe (continue) de longueur <= profondeur d orientation
     * suivant oneDirection mais NE FERME AUCUN JETON Ne continue d evaluer que si
     * les jetons ne sont pas vide ne sont pas deja ferme sont les memes (axe
     * continue) sans prendre en compte le jeton de depart (ligne,colonne)
     *
     * @param ligne
     * @param colonne
     * @param profondeur si egale a 0 la cellule fermee sera uniquement la
     * cellule[ligne][colonne]
     * @param direction orientation de l axe de fermeture des jetons
     * @return renvoie le nombre de jetons appartenant a un axe pour une direcion
     * donne sans prendre en compte le jeton de depart
     */
    public int getLongueurAxeJetons1D(int ligne, int colonne, int profondeur, Directi
on direction) {
        assert (ligne < getLignes() && ligne >= 0); // la cellule doit être dans
la grille
        assert (colonne < getColonnes() && colonne >= 0); // la cellule doit être
e dans la grille
        assert (profondeur >= 1);

        int nbrJetonFermes = 0;
        int coeffProfondeur = 1;
        boolean valide = true;
        while (coeffProfondeur <= profondeur && this.existeNextCellule(ligne, col
onne, coeffProfondeur, direction)
            && valide) {
            int[] coordCible = coordNextJeton(ligne, colonne, coeffProfondeur
, direction);
            int ligneCible = coordCible[0];
            int colonneCible = coordCible[1];
            if (getSymboleJetonOouF(ligneCible, colonneCible) == getSymboleJetonOouF(ligne, colonne)) {
                ++nbrJetonFermes;
            } else {
                valide = false;
            }
            ++coeffProfondeur;
        }
        return nbrJetonFermes;
    }

    /**
     * getLongueurAxeJetons1D donne les coord des jeton ouvert a ferme pour
     * direction donner selon un axe (continue) de longueur profondeur d orientation
     * suivant oneDirection mais ne prend pas en compte le jeton de depart (coord
     * ligne,colonne) Ne continue de fermer que si les jetons evalue ne sont pas
     * vide ne sont pas deja ferme sont les memes (axe continue)
     *
     * @param ligne
     * @param colonne
     * @param profondeur
     * @param direction orientation de l axe de fermeture des jetons
     * @return renvoie les coordonnees
     */
    public int[][] getCoordAlignJetons1DOouF(int ligne, int colonne, int profondeur,

```

```

Direction direction) {
    assert (ligne < getLignes() && ligne >= 0); // la cellule doit être dans
    la grille
    assert (colonne < getColonnes() && colonne >= 0); // la cellule doit être
    e dans la grille
    assert (profondeur >= 2);
    assert (isDirectAvecAlignOouF(ligne, colonne, profondeur));

    int[][] coordJetonContinu = new int[getLongueurAxeJetons1D(ligne, colonne
, profondeur, direction)][2];

    // coord des jetons dans une direction
    int nbrJetonFermes = 0;
    int coeffProfondeur = 1;
    boolean valide = true;
    while (coeffProfondeur <= profondeur && this.existeNextCellule(ligne, col
onne, coeffProfondeur, direction)
        && valide) {
        int[] coordCible = coordNextJeton(ligne, colonne, coeffProfondeur
, direction);
        int ligneCible = coordCible[0];
        int colonneCible = coordCible[1];
        if (getSymboleJetonOouF(ligneCible, colonneCible) == getSymboleJe
tonOouF(ligne, colonne)) {
            coordJetonContinu[nbrJetonFermes][0] = ligneCible;
            coordJetonContinu[nbrJetonFermes][1] = colonneCible;
            ++nbrJetonFermes;
        } else {
            valide = false;
        }
        ++coeffProfondeur;
    }

    return coordJetonContinu;
}

/**
 * donne les coord des jeton ouvert a fermer pour la premiere direction ou un
 * alignemet a ete trouver selon un axe (continue) de longueur profondeur d
 * orientation suivant oneDirection mais ne prend pas en compte le jeton de
 * depart (coord ligne,colonne) Ne continue de fermer que si les jetons evalue
 * ne sont pas vide ne sont pas deja ferme sont les memes (axe continue)
 */
@param ligne
@param colonne
@param profondeur
@param direction orientation de l axe d observation des jetons
@return renvoie les coordonnees
*/
public int[][] getCoordAlignJetonsXDOouF(int ligne, int colonne, int profondeur)
{
    assert (ligne < getLignes() && ligne >= 0); // la cellule doit être dans
    la grille
    assert (colonne < getColonnes() && colonne >= 0); // la cellule doit être
    e dans la grille
    assert (profondeur >= 2);
    assert (isDirectAvecAlignOouF(ligne, colonne, profondeur));

    Direction direction = getAllDirectAlignOouF(ligne, colonne, profondeur)[0];

    int[][] coordJetonContinu = new int[profondeur][2];

    // coord des jetons dans une direction
    int[][] coordJetonAferme1D = getCoordAlignJetons1DOouF(ligne, colonne, pr
ofondeur, direction);

    for (int i = 0; i < coordJetonAferme1D.length; i++) {
        coordJetonContinu[i][0] = coordJetonAferme1D[i][0];

```

```

        coordJetonContinu[i][1] = coordJetonAferme1D[i][1];
    }
    // coord du jeton central
    coordJetonContinu[coordJetonAferme1D.length][0] = ligne;
    coordJetonContinu[coordJetonAferme1D.length][1] = colonne;

    // coord des jetons dans une direction inverse si besoin
    if (coordJetonAferme1D.length < profondeur - 1) {
        int[][] coordJetonAferme1DI = getCoordAlignJetons1DOouF(ligne, co
lonne, profondeur, direction.inverser());
        for (int i = coordJetonAferme1D.length + 1; i < coordJetonContinu
.length; i++) {
            coordJetonContinu[i][0] = coordJetonAferme1DI[i - coordJe
tonAferme1D.length - 1][0];
            coordJetonContinu[i][1] = coordJetonAferme1DI[i - coordJe
tonAferme1D.length - 1][1];
        }
    }

    return coordJetonContinu;
}

// ***** EVALUATION ALIGNEMENT OUVERT *****

public void evaluerCoupAlignOuvert(Joueur joueur1, Joueur joueur2, int[] saisieCe
llule) {
    assert (joueur1 != null && joueur2 != null && saisieCellule != null && jo
ueur1 != joueur2);
    if (Utils_Grille_Evaluation_Alignement.isAlign(saisieCellule[0], saisieCe
llule[1], nbrAlign, this)) {
        if (isDirectAvecAlignOouF(saisieCellule[0], saisieCellule[1], nbr
Align)) {
            Jeton jetonEvalue = getCellule(saisieCellule[0], saisieCe
llule[1]);

            if (jetonEvalue.estEgal(joueur1.getJeton())) {
                joueur1.marquerPoint();
                System.out.println(Messages_Saisie.afficherMessag
eCoupMarquant(joueur1));
            }
            if (jetonEvalue.estEgal(joueur2.getJeton())) {
                joueur2.marquerPoint();
                System.out.println(Messages_Saisie.afficherMessag
eCoupMarquant(joueur1));
            }
            fermeAlignementXD(saisieCellule[0], saisieCellule[1], nbr
Align);
            afficherGrille();
        }
    }
}

```

```

//*****
//CA_Grille.java
//*****
package partie;

import direction.Direction;
import jeton.Jeton;

public abstract class CA_Grille implements In_Grille {

    private Jeton[][] grille;

    // ***** METHODE GRILLE *****

    // ***** METHODE GRILLE CONSTRUCTEUR *****
    public CA_Grille(int nbrLignes, int nbrColonnes) {
        assert (nbrLignes > 0 && nbrColonnes > 0);
        this.grille = new Jeton[nbrLignes][nbrColonnes];
        this.viderGrille(); // initialisation
    }

    public CA_Grille() {
        this.grille = new Jeton[3][3];
        this.viderGrille(); // initialisation
    }

    /**
     * viderGrille permet de mettre/initialiser tout les jetons de la grille a
     * JETON_VIDE
     */
    private void viderGrille() {
        for (Jeton[] ligneJeton : this.grille) {
            for (int i = 0; i < ligneJeton.length; i++) {
                ligneJeton[i] = Jeton.JETON_VIDE;
            }
        }
    }

    // ***** METHODE GRILLE GETTEURS *****
    public int getColonnes() {
        return this.grille[0].length;
    }

    public int getLignes() {
        return this.grille.length;
    }

    public int getNbrCellules() {
        return this.grille[0].length * this.grille.length;
    }

    public Jeton[][] getGrille() {
        return grille;
    }

    /**
     *
     * @param ligne de la cellule indique le 0 compte
     * @param colonne de la cellule indique le 0 compte
     * @return un JETON (contenant un symbole X ou O ou x ou o et un boolean pour
     * indiquer l ouverture
     */
    public Jeton getCellule(int ligne, int colonne) {
        assert (ligne < this.grille.length && ligne >= 0); // la cellule doit Ãªtre dans la grille
        assert (colonne < this.grille[0].length && colonne >= 0); // la cellule doit Ãªtre dans la grille
        return this.grille[ligne][colonne];
    }
}

```

```

// ***** METHODE GRILLE PLACEMENT JETON *****
/**
 * place un jeton dans la grille La cellule ciblée peut être vide
 *
 * @param jeton à placer (seuls JETON_X ou JETON_O sont autorisés)
 * @param ligneCible de la cellule de la grille le 0 compte
 * @param colonneCible de la cellule de la grille le 0 compte
 */
public void placerJeton(Jeton jeton, int ligneCible, int colonneCible) {
    assert (ligneCible < this.grille.length && ligneCible >= 0); // la cellule doit être dans la grille
    assert (colonneCible < this.grille[0].length && colonneCible >= 0); // la cellule doit être dans la grille
    this.grille[ligneCible][colonneCible] = jeton;
}

// ***** METHODE GRILLE EVALUATION *****

// ***** METHODE GRILLE EVALUATION EST VIDE *****
/**
 * estVideCellule
 *
 * @param ligne de la cellule de la grille le 0 compte
 * @param colonne de la cellule de la grille le 0 compte
 * @return la cellule est elle vide ?
 */
public boolean estVideCellule(int ligne, int colonne) {
    assert (ligne < this.grille.length && ligne >= 0); // la cellule doit être dans la grille
    assert (colonne < this.grille[0].length && colonne >= 0); // la cellule doit être dans la grille
    return getCellule(ligne, colonne).estVideJeton();
}

/**
 * toutes les cellules de la grille sont elles vides ?
 *
 * @return
 */
public boolean estVideGrille() {
    boolean estVide = true;
    for (Jeton[] ligneJeton : this.grille) {
        for (int i = 0; i < ligneJeton.length; i++) {
            if (!ligneJeton[i].estVideJeton()) {
                estVide = false;
            }
        }
    }
    return estVide;
}

/**
 * estPleineGrille
 *
 * @return toutes les cellules de la grille contiennent elles JETON_VIDE ?
 */
public boolean estPleineGrille() {
    boolean estPleine = true;
    for (Jeton[] ligneJeton : this.grille) {
        for (int i = 0; i < ligneJeton.length; i++) {
            if (ligneJeton[i].estVideJeton()) {
                estPleine = false;
            }
        }
    }
    return estPleine;
}

```

```

// ***** METHODE GRILLE EVALUATION JETON *****
/**
 * renvoie le nombre de jeton observe dans une grille
 *
 * @param jetonEvaluate evaluate
 * @return le nombre de jeton observe dans une grille
 */
public int getNbrJeton(Jeton jetonEvaluate) {
    int nbr = 0;
    for (int i = 0; i < grille.length; i++) {
        for (int j = 0; j < grille[0].length; j++) {
            if (getCellule(i, j).estEgal(jetonEvaluate)) {
                ++nbr;
            }
        }
    }
    return nbr;
}

// ***** METHODE GRILLE EVALUATION CELLULE *****
/**
 * Les cellules sont elles de coordonnees differentes
 *
 * @param ligne1
 * @param colonne1
 * @param ligne2
 * @param colonne2
 * @return
 */
public boolean sontDifferentes(int ligne1, int colonne1, int ligne2, int colonne2) {
    return (ligne1 != ligne2 || colonne1 != colonne2);
}

// ***** METHODE GRILLE EVALUATION GRILLE *****
/**
 * comparaison de grille utilisable pour s assurer que des grilles generees
 * alÃ©atoirement sont differentes
 *
 * @param grille1 de jeton
 * @param grilleCible de jeton
 * @return true si grille1 comporte au moins un jeton different de grille2
 */
public boolean estEgaleGrille(Jeton[][] grille2) {
    // comparaison de taille
    if ((this.grille.length != grille2.length) || (this.grille[0].length != grille2[0].length)) {
        return false;
    }
    // comparaison des cellules
    for (int i = 0; i < this.getLignes(); ++i) {
        for (int j = 0; j < this.getColonnes(); ++j) {
            if (!grille[i][j].estEgal(grille2[i][j]))
                return false;
        }
    }
    return true;
}

// ***** METHODE GRILLE AFFICHAGE *****
/**
 * toString
 *
 * @return une chaine de caractÃ©re contenant l'etat de la grille
 */
public String toStringGrille() {
    String sGrille = " "; // decalage pour les noms de lignes en dizaines

```

```

// ligne des indices de colonnes
for (int j = 1; j <= getColonnes(); ++j)
    if (j < 10) {
        sGrille += " " + " " + " " + j;
    } else
        sGrille += " " + " " + j;
sGrille += "\n";

// il faut d'abord parcourir les reference de ligne de jeton pour acceder
// jetons
for (int ligne = 1; ligne <= getLignes(); ++ligne) {
    if (ligne < 10) {
        sGrille += " " + ligne;
    } else
        sGrille += ligne;
    for (int colonne = 0; colonne < getColonnes(); ++colonne) {
        sGrille += " " + getCellule((ligne - 1), colonne).toString();
    }
    sGrille += "\n";
}
return sGrille;
}

public String toStringGrille() {
    String sGrille = " "; // decalage pour les noms de lignes en dizaines
    int ligne = 0;

    // ligne des indices de colonnes
    for (int i = 1; i <= this.grille[0].length; ++i)
        if (i < 10) {
            sGrille += " " + " " + " " + i;
        } else
            sGrille += " " + " " + i;
    sGrille += "\n";
    ++ligne;

    // il faut d'abord parcourir les reference de ligne de jeton pour acceder
    // jetons
    for (Jeton[] ligneJeton : grille) {
        if (ligne < 10) {
            sGrille += " " + ligne;
        } else
            sGrille += ligne;
        for (int i = 0; i < ligneJeton.length; i++) {
            sGrille += " " + ligneJeton[i].toString();
        }
        sGrille += "\n";
        ++ligne;
    }
    return sGrille;
}

/**
 * Affiche en system out la String du ToString
 */
public void afficherGrille() {
    System.out.println(this.toStringGrille());
}

// ***** METHODE GRILLE GET_NEXT_CELLULE *****
/**
 * coordNextJeton permet de savoir quelles sont les coordonnees (ligne,colonne)
 * du jeton image cÃ©le jeton contenu dans la cellule projete depuis la
 * cellule de la grille a ligne,colonne vers la direction donnee a la
 * profondeur/distance donnee Le jeton peut etre vide Pas de limite de
 * profondeur

```

```

    *
    * @param ligne
    * @param colonne
    * @param profondeur
    * @param direction
    * @return
    */
    public int[] coordNextJeton(int ligne, int colonne, int profondeur, Direction direction) {
        la grille
        e dans la grille
        assert (ligne < getLignes() && ligne >= 0); // la cellule doit Ãatre dans
        assert (colonne < getColonnes() && colonne >= 0); // la cellule doit Ãatre
        assert (existeNextCellule(ligne, colonne, profondeur, direction));

        int[] coord = new int[2];
        int ligneCible = profondeur * direction.getDligne() + ligne;
        int colonneCible = profondeur * direction.getDcolonne() + colonne;

        coord[0] = ligneCible;
        coord[1] = colonneCible;
        return coord;
    }

    /**
     * Pour les elements donnees, existeNextCellule permet de savoir si la cellule
     * image est comprise dans la grille Pas d indication de la nature du jeton Pas
     * de limite de profondeur cellule image cad : cÃ d le jeton contenu dans la
     * cellule projetee depuis la cellule de la grille a ligne,colonne vers la
     * direction donnee a la profondeur/distance donnee
     */
    * @param ligne
    * @param colonne
    * @param profondeur
    * @param direction
    * @return
    */
    public boolean existeNextCellule(int ligne, int colonne, int profondeur, Direction direction) {
        la grille
        e dans la grille
        assert (ligne < getLignes() && ligne >= 0); // la cellule doit Ãatre dans
        assert (colonne < getColonnes() && colonne >= 0); // la cellule doit Ãatre

        boolean existe = false;

        int ligneCible = profondeur * direction.getDligne() + ligne;
        int colonneCible = profondeur * direction.getDcolonne() + colonne;

        if (ligneCible < getLignes() && ligneCible >= 0 && colonneCible < getColonnes() && colonneCible >= 0) {
            existe = true;
        }
        return existe;
    }

    /**
     * getNextJeton permet d obtenir le jeton image cÃ d le jeton contenu dans la
     * cellule projetee depuis la cellule de la grille a ligne,colonne vers la
     * direction donnee a la profondeur/distance donnee
     */
    * @param ligne
    * @param colonne
    * @param profondeur
    * @param direction
    * @return
    */
    public Jeton getNextJeton(int ligne, int colonne, int profondeur, Direction direction) {
        la grille
        e dans la grille
        assert (ligne < getLignes() && ligne >= 0); // la cellule doit Ãatre dans
        assert (colonne < getColonnes() && colonne >= 0); // la cellule doit Ãatre
        assert (existeNextCellule(ligne, colonne, profondeur, direction));

        int[] coord = coordNextJeton(ligne, colonne, profondeur, direction);

        return getCellule(coord[0], coord[1]);
    }

```



```

//*****
//In_Grille.java
//*****
package partie;

import jeton.Jeton;

public interface In_Grille {

    // ***** METHODE GRILLE *****

    // ***** METHODE GRILLE GETTEURS *****
    /**
     *
     * @return nombre de Colonnes de la grille
     */
    public int getColonnes();

    /**
     *
     * @return nombre de lignes de la grille
     */
    public int getLignes();

    /**
     *
     * @return nombre de cellules de la grille
     */
    public int getNbrCellules();

    public Jeton[][] getGrille();

    /**
     *
     * @param ligne de la cellule indique le 0 compte
     * @param colonne de la cellule indique le 0 compte
     * @return un JETON (contenant un symbole X ou O ou x ou o et un boolean pour
     * indiquer l ouverture
     */
    public Jeton getCellule(int ligne, int colonne);

    // ***** METHODE GRILLE EVALUATION *****

    // ***** METHODE GRILLE EVALUATION EST VIDE *****
    /**
     *
     * estVideCellule
     *
     * @param ligne de la cellule de la grille le 0 compte
     * @param colonne de la cellule de la grille le 0 compte
     * @return la cellule est elle vide ?
     */
    public boolean estVideCellule(int ligne, int colonne);

    /**
     *
     * toutes les cellules de la grille sont ils vides ?
     *
     * @return
     */
    public boolean estVideGrille();

    /**
     *
     * estPleineGrille
     *
     * @return toutes les cellules de la grille contiennent elles JETON_VIDE ?
     */
    public boolean estPleineGrille();

    // ***** METHODE GRILLE EVALUATION JETON *****

```

```

    /**
     * renvoie le nombre de jeton observe dans une grille
     *
     * @param jetonEvaluate evaluate
     * @return le nombre de jeton observe dans une grille
     */
    public int getNbrJeton(Jeton jetonEvaluate);

    // ***** METHODE GRILLE EVALUATION CELLULE *****
    /**
     * Les cellules sont elles de coordonnees differentes
     *
     * @param ligne1
     * @param colonne1
     * @param ligne2
     * @param colonne2
     * @return
     */
    public boolean sontDifferentes(int ligne1, int colonne1, int ligne2, int colonne2
    );

    // ***** METHODE GRILLE EVALUATION GRILLE *****
    /**
     * comparaison de grille utilisable pour s assurer que des grilles generees
     * alÃ©atoirement sont differentes
     *
     * @param grille1 de jeton
     * @param grilleCible de jeton
     * @return true si grille1 comporte au moins un jeton different de grille2
     */
    public boolean estEgaleGrille(Jeton[][] grille2);

    // ***** METHODE GRILLE PLACEMENT JETON *****
    /**
     * place un jeton dans la grille La cellule ciblee peut etre vide
     *
     * @param jeton Ã placer (seuls JETON_X ou JETON_O sont autorisÃ©s)
     * @param ligneCible de la cellule de la grille le 0 compte
     * @param colonneCible de la cellule de la grille le 0 compte
     */
    public void placerJeton(Jeton jeton, int ligneCible, int colonneCible);

    // ***** METHODE GRILLE AFFICHAGE *****
    /**
     * toString
     *
     * @return une chaine de caractÃ©re contenant l'etat de la grille
     */
    public String toStringGrille();

    /**
     * Affiche en system out la String du ToString
     */
    public void afficherGrille();
}

```

```

//*****
//PartiePermutation.java
//*****
package partie;

import java.util.Collections;
import java.util.LinkedList;
import java.util.Random;

import interaction.MessagePermutation;
import interaction.Messages_Saisie;
import jeton.*;
import utilitaires.Utills_Grille_Evaluation_Adjacent;

public class PartiePermutation extends PartieMorpion {

    private int nbrAlign;
    private int[] saisieCellule;
    private int[] saisieCellule2;

    // la partie durera tant qu'il y aura des jetons a l'aligner pour un des deux
    // joueurs

    public PartiePermutation(int nbrLignes, int nbrColonnes, int nbrAlign) {
        super(nbrLignes, nbrColonnes);
        remplirAleaGrille();
        this.saisieCellule = new int[2];
        this.saisieCellule2 = new int[2];

        this.nbrAlign = nbrAlign;
        int choixNbrAlignMax = (nbrColonnes >= nbrLignes) ? nbrLignes : nbrColonnes;

        assert (nbrAlign <= choixNbrAlignMax);
        // ce nombre ne doit pas être plus grand que le nombre de colonnes ou de
        // lignes de votre grille
    }

    public PartiePermutation(int nbrLignes, int nbrColonnes) {
        super(nbrLignes, nbrColonnes);
        remplirAleaGrille();
        this.saisieCellule = new int[2];
        this.saisieCellule2 = new int[2];

        this.nbrAlign = 3;
        int choixNbrAlignMax = (nbrColonnes >= nbrLignes) ? nbrLignes : nbrColonnes;

        assert (nbrAlign <= choixNbrAlignMax);
        // ce nombre ne doit pas être plus grand que le nombre de colonnes ou de
        // lignes de votre grille
    }

    public PartiePermutation() {
        super(5, 6);
        remplirAleaGrille();
        this.saisieCellule = new int[2];
        this.saisieCellule2 = new int[2];
    }

    // ***** METHODE GRILLE *****
    // ***** METHODE GRILLE REMPLISSAGE ALEATOIRE *****
    /**
     * remplissage aléatoire avec JEON caractères ouverts, à partir d'une liste de
     * JETON finie de taille égale à celle de la grille tirage aléatoire sans remise
     * de la liste de JETON dans chacune des cellules si le nombre de jeton est
     * impair le dernier jeton sera déterminé de manière aléatoire Ne pourra être
     * appelée que si la grille est vide
     */
    private void remplirAleaGrille() {

```

```

        assert (this.estVideGrille());
        // initialisation de la liste des jetons
        LinkedList<Jeton> listeJetons = new LinkedList<Jeton>();
        // Linked list car accès terminaux constant

        // initialisation de la liste des jetons
        if (getNbrCellules() % 2 == 0) {
            for (int i = 0; i < getNbrCellules(); ++i) {
                listeJetons.addLast(Jeton.values()[i % 2 + 1]);
            }
        } else {
            Random r = new Random();
            int valeur = r.nextInt(2) + 1; // valeur entre 1 et 2
            listeJetons.addLast(Jeton.values()[valeur]);
            for (int i = 1; i < getNbrCellules(); ++i) { // on ne commence pl
                listeJetons.addLast(Jeton.values()[i % 2 + 1]);
            }
        }

        // brassage de la liste des jetons
        // https://www.tutorialspoint.com/java/util/collections_shuffle.htm
        Collections.shuffle(listeJetons);

        // insertion de la liste de jetons dans la grille
        // boucle for each pour la Linked list qui travaille difficilement avec d
        // indices
        for (Jeton[] ligneJeton : this.getGrille()) {
            for (int i = 0; i < ligneJeton.length; i++) {
                ligneJeton[i] = listeJetons.getFirst();
                listeJetons.removeFirst();
            }
        }

        @Override
        public void jouerCoup(Joueur joueurActuel) {
            boolean saisieCorrecteJouerCoup = false;

            while (!saisieCorrecteJouerCoup) {
                System.out.println("Vous allez choisir les deux cases pour permut
                ation.\n");

                saisieCellule = Messages_Saisie.saisirCellule(getGrille());
                System.out.println(Messages_Saisie.afficherMessageCellule(joueurA
                ctuel, saisieCellule));

                saisieCellule2 = Messages_Saisie.saisirCellule(getGrille());
                System.out.println(Messages_Saisie.afficherMessageCellule(joueurA
                ctuel, saisieCellule2));

                // les jetons doivent être de cases différentes
                if (sontDifferentes(saisieCellule[0], saisieCellule[1], saisieCel
                lule2[0], saisieCellule2[1])) {
                    // les jetons doivent être adjacents
                    if (Utills_Grille_Evaluation_Adjacent.sontAdjacents(saisie
                    Cellule[0], saisieCellule[1],
                        saisieCellule2[0], saisieCellule2[1], thi
                    s)) {
                        saisieCorrecteJouerCoup = true;
                    } else
                        System.out.println(
                            "La case sélectionnée n'est pas a
                            djacente a la première case sélectionnée. Veuillez recommencer la saisie des deux cellule
                            s.\n");
                    } else
                        System.out.println(
                            "La case sélectionnée est identique a la
                            première case sélectionnée. Veuillez recommencer la saisie des deux cellules.\n");

```

```

        }
        permutationJeton(saisieCellule[0], saisieCellule[1], saisieCellule2[0], s
aisieCellule2[1]);
        System.out.println(MessagePermutation.afficherMessageCoupJoue(joueurActue
l, saisieCellule, saisieCellule2));
    }

    /**
     * on ne peut pas ganger d alignements avec des jetons qui sont deja fermes
     * comme on peut modifier les jetons fermes il faut limiter les jetons evalues
     * au jeton ouverts
     */
    @Override
    public void evaluerCoup(Joueur joueur1, Joueur joueur2) {

        assert (saisieCellule != null); // on oblige le joueur a avoir jouer un co
up
        assert (saisieCellule2 != null); // on oblige le joueur a avoir jouer un c
oup

        evaluerCoupAlignOuvert(joueur1, joueur2, saisieCellule);
        if (estOuvert(saisieCellule2[0], saisieCellule2[1])) {
            evaluerCoupAlignOuvert(joueur1, joueur2, saisieCellule2);
        }
    }

    @Override
    public boolean estFinie() {
        return (getScoreJ1() >= pointMaxPermut(getLignes(), getColonnes(), nbrAli
gn)
                || getScoreJ2() >= pointMaxPermut(getLignes(), getColonne
s(), nbrAlign));
    }

    public static int pointMaxPermut(int ligne, int colonne, int align) {
        assert (ligne > 0 && colonne > 0 && align > 0);
        int pointMax = (ligne * colonne) - ((ligne * colonne) % 2);
        pointMax /= 2;
        pointMax /= align;
        return pointMax;
    }
}

```

```

package utilitaires;

import partie.*;
import direction.Direction;

public class Utils_Grille_Evaluation_Adjacent {

    // ***** METHODE GRILLE *****
    // ***** METHODE GRILLE ADJACENT JETON *****

    /**
     * existe il dans les cellules voisines de la cellule donnee [ligne,colonne] des
     * jetons non vides la cellule peut etre vide mais doit etre dans la grille
     *
     * @param ligne
     * @param colonne
     * @return
     */
    public static boolean existeAdjacent(int ligne, int colonne, CA_Grille grille) {
        assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit Ãªtre dans la grille
        assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit Ãªtre dans la grille

        for (Direction oneDirection : Direction.values())
            if (grille.existeNextCellule(ligne, colonne, 1, oneDirection)) {
                if (!grille.getNextJeton(ligne, colonne, 1, oneDirection).estVideJeton()) {
                    return true;
                }
            }
        return false;
    }

    /**
     * les cellules donnees sont elles adjacents doivent etre des cellules
     * differentes Il n est pas verifie que les cellules comprennent des jetons non
     * vides
     *
     * @param ligne1
     * @param colonne1
     * @param colonne2
     * @param ligne2
     * @return
     */
    public static boolean sontAdjacents(int ligne1, int colonne1, int ligne2, int colonne2, CA_Grille grille) {
        assert (ligne1 != ligne2 || colonne1 != colonne2); // les jetons doivent etre differents

        assert (ligne1 < grille.getLignes() && ligne1 >= 0); // la cellule doit Ãªtre dans la grille
        assert (colonne1 < grille.getColonnes() && colonne1 >= 0); // la cellule doit Ãªtre dans la grille
        assert (ligne2 < grille.getLignes() && ligne2 >= 0); // la cellule doit Ãªtre dans la grille
        assert (colonne2 < grille.getColonnes() && colonne2 >= 0); // la cellule doit Ãªtre dans la grille

        boolean adjacent = false;

        if ((Math.abs(ligne1 - ligne2) <= 1) && (Math.abs(colonne1 - colonne2) <= 1)) {
            adjacent = true;
        }

        return adjacent;
    }
}

```

```

package utilitaires;

import direction.Direction;
import jeton.Jeton;
import partie.*;

public class Utils_Grille_Evaluation_Forme {

    /**
     * existeForme permet de savoir si les cellules de toute une forme Ã partir d un
     * point donnÃ© (en haut a gauche de la forme) sont inclus dans la grille
     *
     * mais pas de savoir si elle existe juste de savoir si elle est comprises dans
     * la grille Ã partir du point en haut a gauche
     *
     * @param ligneOrigine
     * @param colonneOrigine
     * @param forme
     * @return
     */
    public static boolean existeForme(int ligne, int colonne, CA_Grille grille, Forme
forme) {
        boolean existe = true;
        // le premier point de la forme est evalue en coordonne [ligne,colonne]

        for (int i = 0; i < forme.getNbrPoint(); ++i) {
            // on obtient les parametres de la projection (directionCible et
            // profondeurCible) pour parvenir Ã la cellule suivante
            Direction directionCible = Direction.values()[forme.getOrientatio
n()[i]];

            int profondeurCible = forme.getDistance()[i];

            // on verifie que la cellule cible existe
            if (!grille.existeNextCellule(ligne, colonne, profondeurCible, di
rectionCible)) {
                return false; // si il y a au moins un point de la forme
                // renvoie false
            } else {
                // si elle existe on extrait ses coordonnees pout les reu
                // tiliser dans la boucle
                int[] coordCible = grille.coordNextJeton(ligne, colonne,
profondeurCible, directionCible);
                ligne = coordCible[0];
                colonne = coordCible[1];
            }
        }
        return existe;
    }

    /**
     * getCoordForme donne un tableau de coordonne (ligne colonne) permettant d
     * identifier les cellules impliquees dans la realisation de la forme donnee
     * mais ne donne la forme que pour le point donne grille[ligne,colonne] etant un
     * point le point en haut a gauche de la forme
     *
     * @param ligneOrigine
     * @param colonneOrigine
     * @param forme
     * @return unt table contenant x coordonnes (donc une table de table a deux
     * dimensions)
     */
    public static int[][] getCoordForme(int ligne, int colonne, CA_Grille grille, For
me forme) {
        assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit Ãt
re dans la grille
        assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule do
it Ãtre dans la grille
        assert (Utils_Grille_Evaluation_Forme.existeForme(ligne, colonne, grille,

```

```

forme));

        int[][] coord = new int[forme.getNbrPoint()][2];

        // le premier point de la forme est evalue en coordonne [ligne,colonne]
        coord[0][0] = ligne;
        coord[0][1] = colonne;

        for (int i = 1; i < forme.getNbrPoint(); ++i) {
            // on obtient les parametres de la projection (directionCible et
            // profondeurCible) pour parvenir Ã la cellule suivante
            Direction directionCible = Direction.values()[forme.getOrientatio
n()[i - 1]];

            int profondeurCible = forme.getDistance()[i - 1];
            coord[i] = grille.coordNextJeton(ligne, colonne, profondeurCible,
directionCible);

            ligne = coord[i][0];
            colonne = coord[i][1];
        }
        return coord;
    }

    /**
     * permet de renvoyer le contenu des cellules (cad jetons) appartenant Ã une
     * seule forme Ã partir d un point donnÃ©e dans une chaine de caractere
     *
     * @param ligne du point d ancrage de la forme
     * @param colonne du point d ancrage de la forme
     * @param forme evaluee
     * @return
     */
    public static String getJetonForme(int ligne, int colonne, CA_Grille grille, Form
e forme) {
        assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit Ãt
re dans la grille
        assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule do
it Ãtre dans la grille
        assert (Utils_Grille_Evaluation_Forme.existeForme(ligne, colonne, grille,
forme));

        int[][] coordDesJetons = Utils_Grille_Evaluation_Forme.getCoordForme(lign
e, colonne, grille, forme);

        String sJeton = "";

        for (int i = 0; i < coordDesJetons.length; ++i) {
            sJeton += grille.getCellule(coordDesJetons[i][0], coordDesJetons[
i][1]).getSymbole();
        }

        return sJeton;
    }

    /**
     * permet de renvoyer le contenu des cellules (cad jetons) (dans une chaine de
     * caractere) appartenant Ã plusieurs formes derivees d une seule (forme donnee)
     * les formes sont derivees Ã partir de forme.transForme() qui renvoie une forme
     * dont les indices sont decalé de int decalageIndice permettant ainsi les
     * jetons de toutes les cellules existantes incluses dans une forme pour lequel
     * le point donnee est considere succesivement comme chaqu'un des points de la
     * forme
     *
     * @param ligne du point d ancrage de la forme
     * @param colonne du point d ancrage de la forme
     * @param forme evaluee d origine
     * @return
     */
    public static String getJetonFormeAll(int ligne, int colonne, CA_Grille grille, F
orme forme) {

```

```

        assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit Ãªtre dans la grille
        assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit Ãªtre dans la grille

        String sJetonAll = "";

        for (int i = 0; i < forme.getNbrPoint(); ++i) {
            Forme formeTemp = forme.transForme(i);
            if (Utils_Grille_Evaluation_Forme.existeForme(ligne, colonne, grille, formeTemp)) {
                sJetonAll += Utils_Grille_Evaluation_Forme.getJetonForme(ligne, colonne, grille, formeTemp);
            }
            sJetonAll += ",";
        }

        return sJetonAll;
    }

    /**
     * permet d evaluer si un point complete une forme
     *
     * @param ligne
     * @param colonne
     * @param forme
     * @return
     */
    public static boolean estCompleteForme(int ligne, int colonne, CA_Grille grille, Forme forme) {
        assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit Ãªtre dans la grille
        assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit Ãªtre dans la grille
        assert (!grille.estVideCellule(ligne, colonne)); // la cellule evaluee ne doit pas etre vide

        // quelle chaine devrait on avoir pour que la forme soit complete
        Jeton jetonCible = grille.getCellule(ligne, colonne);
        String formeCible = "";
        for (int i = 1; i <= forme.getNbrPoint(); ++i) {
            formeCible += jetonCible.getSymbole();
        }

        // quelle sont les formes derivees dans lesquelles sont impliquees la
        // cellule[ligne,colonne] observe
        // pour le template donnee (forme)
        String formeEvaluee = Utils_Grille_Evaluation_Forme.getJetonFormeAll(ligne, colonne, grille, forme);

        // comparaison des chaines
        return formeEvaluee.contains(formeCible);
    }

    /**
     * si un point complete une forme permet d obtenir les coordonnees des jetons
     * emme si le jeton complete plusieurs forme a la fois il n y aura qu une seule
     * forme complete renvoye
     *
     * @param ligne du point d ancrage de la forme
     * @param colonne du point d ancrage de la forme
     * @param forme evaluee d origine
     * @return table
     */
    public static int[][] getCoordFormeComplete(int ligne, int colonne, CA_Grille grille, Forme forme) {
        assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit Ãªtre dans la grille
        assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit Ãªtre dans la grille

        assert (estCompleteForme(ligne, colonne, grille, forme));

        // quelle chaine devrait on avoir pour que la forme soit complete
        Jeton jetonCible = grille.getCellule(ligne, colonne);
        String formeCible = "";
        for (int i = 1; i <= forme.getNbrPoint(); ++i) {
            formeCible += jetonCible.getSymbole();
        }

        int[][] coordFormeComplete = new int[forme.getNbrPoint()][2];

        // quelle sont les formes derivees dans lesquelles sont impliquees la
        // cellule[ligne,colonne] observe
        // pour le template donnee (formeCible)
        for (int i = 0; i < forme.getNbrPoint(); ++i) {
            Forme formeTemp = forme.transForme(i);
            if (existeForme(ligne, colonne, grille, formeTemp)) {
                String sFormeTemp = getJetonForme(ligne, colonne, grille, formeTemp);

                if (formeCible.equals(sFormeTemp)) {
                    coordFormeComplete = getCoordForme(ligne, colonne, grille, formeTemp);

                    return coordFormeComplete;
                }
            }
        }

        return coordFormeComplete;
    }

```

```

package utilitaires;

import java.util.EnumSet;

import direction.Direction;
import jeton.Jeton;
import partie.CA_Grille;

public class Utils_Grille_Evaluation_Alignement {

    /**
     * renvoie une chaine de symbole de jetons obtenus dans une direction donnee de
     * taille inferieure ou egale a la profondeur (tant que la projection est dans
     * la grille) a partir d'une case de la grille (ligne, colonne) Attention la
     * case de depart n est pas comprise dans la chaine
     *
     * @param ligne
     * @param colonne
     * @param profondeur
     * @param direction
     * @param grille
     * @return
     */
    public static String getLigneJeton(int ligne, int colonne, int profondeur, Direction direction, CA_Grille grille) {
        assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit être dans la grille
        assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit être dans la grille
        assert (profondeur > 0);

        // aligneCible ligne de jeton observée dans la direction donnée
        String aligneCible = "";
        int coeffProfondeur = 1;

        while (coeffProfondeur <= profondeur && grille.existeNextCellule(ligne, colonne, coeffProfondeur, direction)) {
            int colonneCible = coeffProfondeur * direction.getDcolonne() + colonne;
            int ligneCible = coeffProfondeur * direction.getDligne() + ligne;
            aligneCible += grille.getCellule(ligneCible, colonneCible).getSymbole();
            ++coeffProfondeur;
        }
        return aligneCible;
    }

    /**
     * alignement pour UNE Direction donnee ET son Inversee
     *
     * @param ligne de la cellule observée
     * @param colonne de la cellule observée
     * @param profondeur est le nombre de cellule observées au max qui sont alignées
     *
     * dans grille doit etre >=2
     * @param direction et direction opposée vers laquelle observer un alignement
     * @return si un alignement a été trouvé
     */
    public static boolean isDirectInAlign(int ligne, int colonne, int profondeur, Direction direction, CA_Grille grille) {
        assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit être dans la grille
        assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit être dans la grille
        assert (!grille.estVideCellule(ligne, colonne)); // la cellule évaluée ne doit pas etre vide
        assert (profondeur >= 2);
    }

```

```

// jetonEvalue dont on evalue l implication dans un alignement avec d'autres
Jeton jetonEvalue = grille.getCellule(ligne, colonne);

// aligneEvalue ligne de jeton que le joueur souhaiterait avoir à partir de
// jetonEvalue
String aligneEvalue = "";
for (int i = 1; i <= profondeur; ++i) {
    // aligneEvalue ligne de jeton que le joueur souhaiterait avoir à partir de
    // jetonEvalue
    aligneEvalue += jetonEvalue.getSymbole();
}

// aligneCible ligne de jeton observée dans les directions données
String aligneCible = "";
String inverse = getLigneJeton(ligne, colonne, profondeur, direction.inverse(), grille);
inverse = new StringBuilder(inverse).reverse().toString();
aligneCible += inverse;
aligneCible += jetonEvalue.getSymbole();
aligneCible += getLigneJeton(ligne, colonne, profondeur, direction, grille);

// comparaison des chaines
return aligneCible.contains(aligneEvalue);
}

/**
 * alignement pour TOUTES les Directions disponibles le nombre de direction pour laquelle un alignement a ete trouvé
 *
 * @param ligne
 * @param colonne
 * @param profondeur
 * @return le nombre de direction/orientation qui ont été trouvées avec alignementCellule dans toutes les directions
 */
public static int nbrDirectAvecAlign(int ligne, int colonne, int profondeur, CA_Grille grille) {
    assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit être dans la grille
    assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit être dans la grille
    assert (!grille.estVideCellule(ligne, colonne)); // la cellule évaluée ne doit pas etre vide
    assert (profondeur >= 2);

    int alignement = 0;

    for (Direction oneDirection : EnumSet.range(Direction.NORD, Direction.SUD _EST)) {
        if (isDirectInAlign(ligne, colonne, profondeur, oneDirection, grille)) {
            ++alignement;
        }
    }
    return alignement;
}

/**
 * existe t il une direction pour laquelle un alignement de taille profondeur a ete trouve ?
 *
 * @param ligne
 * @param colonne
 * @param profondeur
 * @param grille

```

```

        * @return
        */
        public static boolean isAlign(int ligne, int colonne, int profondeur, CA_Grille grille) {
            assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit Ãªtre dans la grille
            assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit Ãªtre dans la grille
            assert (!grille.estVideCellule(ligne, colonne)); // la cellule évaluée ne doit pas être vide
            assert (profondeur >= 2);

            for (Direction oneDirection : EnumSet.range(Direction.NORD, Direction.SUD_EST)) {
                if (isDirectInAlign(ligne, colonne, profondeur, oneDirection, grille)) {
                    return true;
                }
            }
            return false;
        }

        /**
         * AVANT appel de cette fonction il devra avoir été vérifié qu'il avait des
         * alignements renvoie les directions (droites et inverses) pour lesquelles un
         * alignement a été trouvé
         *
         * @param ligne
         * @param colonne
         * @param profondeur
         * @return table des directions pour lesquelles un alignement a été trouvé
         */
        public static Direction[] getDirectAlign(int ligne, int colonne, int profondeur, CA_Grille grille) {
            assert (ligne < grille.getLignes() && ligne >= 0); // la cellule doit Ãªtre dans la grille
            assert (colonne < grille.getColonnes() && colonne >= 0); // la cellule doit Ãªtre dans la grille
            assert (!grille.estVideCellule(ligne, colonne)); // la cellule évaluée ne doit pas être vide
            assert (profondeur >= 2);
            assert (isAlign(ligne, colonne, profondeur, grille));

            Direction[] tableDirect = new Direction[nbrDirectAvecAlign(ligne, colonne, profondeur, grille)];
            int indice = 0;
            for (Direction direction : EnumSet.range(Direction.NORD, Direction.SUD_EST)) {
                if (isDirectInAlign(ligne, colonne, profondeur, direction, grille)) {
                    tableDirect[indice] = direction;
                    ++indice;
                }
            }
            return tableDirect;
        }
    }
}

```



```

//*****
//Appli.java
//*****
package appli;

import partie.*;

public class Appli {

    /**
     * Veuillez taper dans les arguments des param tres du lancement de
     * l application : < 0 > Pour la selection d un jeu via un menu interacti
f. < 1
     * > Pour jouer au TicTacToe. < 2 > Pour jouer au Morpion. < 3 > Pour jouer au
     * TicTacToe extension Forme. < 4 > Pour jouer au TicTacToe extension
     * Permutation.
     *
     * @param args
     */
    public static void main(String[] args) {

        InPartie partie = null;

        for (String s : args) {

            int choixJeu = Integer.parseInt(s);
            int choixGrilleLigne;
            int choixGrilleColonne;
            int choixNbrAlignements = 3;
            int choixForme = 1;

            switch (choixJeu) {
                case 1:
                    choixGrilleLigne = 3;
                    choixGrilleColonne = 3;
                    System.out.println("La partie de TicTacToe va commencer,
preparez-vous !\n");
                    partie = new PartieTicTacToe(choixGrilleLigne, choixGrill
eColonne, choixNbrAlignements);
                    break;
                case 2:
                    choixGrilleLigne = 5;
                    choixGrilleColonne = 6;
                    System.out.println("La partie de Morpion va commencer, pr
eparez-vous !\n");
                    partie = new PartieMorpion(choixGrilleLigne, choixGrilleC
olonne, choixNbrAlignements);
                    break;
                case 3:
                    choixGrilleLigne = 12;
                    choixGrilleColonne = 12;
                    // forme choix
                    System.out.println("Determinons la forme qui permettra d
e marquer un point.");
                    System.out.println(Forme.toStringFormeDispoConsigne());
                    System.out.println("Veuillez choisir une forme pas son in
dice, entre 1 et "
                                + Forme.getListFormesDispo().length + " c
ompris.");
                    choixForme = Menu.setChoix(1, Forme.getListFormesDispo().
length);
                    Forme formeChoisie = new Forme(choixForme);
                    System.out.println(formeChoisie.toStringFormeChoisie());
                    System.out.println("La partie de TicTacToe Forme va comme
ncer, preparez-vous !\n");
                    partie = new PartieForme(choixForme);
                    break;
                case 4:
                    choixGrilleLigne = 5;

```

```

                    choixGrilleColonne = 6;
                    System.out.println("La partie de TicTacToe Permutation va
commencer, preparez-vous !\n");
                    partie = new PartiePermutation(choixGrilleLigne, choixGri
lleColonne, choixNbrAlignements);
                    break;
                default:
                    Menu.affichageMenuPrincipal();
                    break;
            }
        }
        if (partie != null) {
            partie.lancerPartie();
        }
    }
}

```

```

//*****
//Menu.java
//*****
package appli;

import java.util.Scanner;

import partie.CA_Grille_Partie;
import partie.Forme;
import partie.PartieForme;
import partie.PartieMorpion;
import partie.PartiePermutation;
import partie.PartieTicTacToe;

public class Menu {

    /**
     * Securise la selection du choix de la partie ou Securise la selection de la
     * taille de la grille
     *
     * @param borneMin choixMinimale autorsÃ©
     * @param borneMax choixMaximum autorsÃ©
     */
    public static int setChoix(int borneMin, int borneMax) {
        boolean saisieCorrecte = false;
        int nombreChoisie = borneMin;
        while (!saisieCorrecte) {

            Scanner sc = new Scanner(System.in);
            System.out.println("Veuillez choisir un nombre entre " + borneMin
                + " et " + borneMax + " compris.\nPuis appuyez sur \'Entree\'.\n");
            String saisie = sc.nextLine();
            System.out.println("Vous avez tape : " + saisie + ".\n");

            try {
                if (saisie.matches("\\s*\\d+\\s*")) {
                    nombreChoisie = Integer.parseInt(saisie);
                    if (nombreChoisie >= borneMin && nombreChoisie <=
                        borneMax) {
                        // sortie de la boucle
                        saisieCorrecte = true;
                    } else
                        System.out.println("La saisie est incorre
                            cte, recommencez.\n");
                } else
                    System.out.println("La saisie est incorrecte, rec
                        ommencez.\n");
            } catch (java.lang.NumberFormatException e1) {
                System.out.println("Le format est invalide, recommencez.\
                    n");
            }
        }
        return nombreChoisie;
    }

    public static void affichageMenuPrincipal() {
        System.out.println("Bienvenue dans le jeu TicTacToe et ses variantes.\n\n
            ");
        System.out.println("<<< MENU PRINCIPAL >>>\n");
        System.out.println("Veuillez taper :");
        System.out.println("< 1 > Pour afficher les rÃˆgles.");
        System.out.println("< 2 > Pour afficher des informations sur ce projet.");
        ;
        System.out.println("< 3 > Pour jouer.");
        System.out.println("< 4 > Pour quitter.");

        int choix;

```

```

        choix = setChoix(1, 4);

        switch (choix) {
        case 1:
            System.out.println("<<< REGLES >>>\n");
            affichageMenuRegles();
            break;
        case 2:
            System.out.println("<<< INFORMATIONS >>>\n");
            affichageMenuInfo();
            break;
        case 3:
            System.out.println("<<< JOUEZ ! >>>\n");
            affichageMenuJeu();
            break;
        case 4:
            System.out.println("Au revoir.\n");
            break;
        default:
            System.out.println("Choix de menu invalide.\n");
            break;
        }
        if (choix != 4) {
            System.out.println("Retour au Menu Principal.\n");
            affichageMenuPrincipal();
        }
    }

    private static void affichageMenuRegles() {
        System.out.println("\nLe Tic-Tac-Toe.");
        System.out.println("Le Tic-Tac-Toe, aussi appelÃ© Morpion et Oxo en Belgi
            que, est un jeu de rÃ©flexion se\n"
                + "pratiquant Ã  deux joueurs au tour par tour dont le bu
                t est de crÃ©er le premier un alignement. Nous\n"
                + "commentÃ©ons par le jeu dans sa forme la plus simple.\n"
                + "Le jeu se joue sur une grille de 3 Ã  3. Deux joueu
                rs s'affrontent. Ils doivent remplir chacun Ã  leur tour une case de la grille avec le symbole qui le
                ur est attribuÃ© : O ou X. Le gagnant est celui qui\n"
                + "arrive le premier Ã  aligner trois symboles identiques
                , horizontalement, verticalement ou en diagonale.\n"
                + "La partie est nulle si toutes les cases sont occupÃ©es
                et qu'aucun joueur n'a obtenu un alignement. Il\n"
                + "est coutume de laisser le joueur jouant X effectuer le
                premier coup de la partie.");
        System.out.println("\nLe Morpion.");
        System.out.println("Dans le jeu de Morpion, les grilles ont une taille quelco
            nque. Les rÃˆgles du jeu sont modifiÃ©es\n"
                + "comme suit.\n"
                + "La partie ne se termine plus au
                premier alignement mais continue en alternant les coups des\n"
                + "deux joueurs jusqu'Ã  ce que t
                outes les cases soient occupÃ©es.\n"
                + "Un joueur ne peut poser un symb
                ole que sur une case Ã©tant adjacente (horizontalement, verti-
                calement ou en diagonale) Ã  une case
                occupÃ©e. Au premier coup, le placement est libre.\n"
                + "Un mÃªme symbole ne peut compte
                r que pour un alignement. DÃ©s qu'un alignement est formÃ©,\n"
                + "les symboles qui le composent ne peuve
                nt plus concourir Ã  la rÃ©alisation d'un autre alignement.\n"
                + "Ces symboles sont dits Ã  trois fermÃ©s.
                Les symboles non encore fermÃ©s sont dits Ã  trois ouverts.\n"
                + "En fin de partie, le joueur aya
                nt fait le plus d'alignements gagne. La partie est nulle en cas\n"
                + "d'Ã©galitÃ©.");
        System.out.println("\nLe Tic-Tac-Toe extension forme.");
    }

```

```

        System.out.println("Ce nâ\200\231est plus des alignements quâ\200\231il faut faire mais des formes particuliÃres (une croix par\n"
            + "exemple).");
        System.out.println("\nLe Tic-Tac-Toe extension permutation.");
        System.out.println("La grille est initialement remplie alÃatoirement dâ\200\231autant de X que de O (plus un X ou un O choisi\n"
            + "lui aussi alÃatoirement si le nombre de cases est impair). Un coup ne consiste plus Ã dÃposer\n"
            + "un symbole mais Ã permuter un X avec un O. Les symboles permutÃs peuvent Ãtre ouverts ou\n"
            + "fermÃs. Un point est remportÃ si cette permutation conduit Ã rÃaliser une forme particuliÃre\n"
            + "de symboles ouverts (un alignement de 3 symboles, par exemple). Le joueur qui remporte le\n"
            + "point ne dÃpend pas de qui joue le coup mais du symbole composant la forme. Si câ\200\231est un X\n"
            + "(resp. O), le point est remportÃ par le joueur X (resp. O). Ainsi, une mÃme permutation peut\n"
            + "conduire Ã augmenter le score des deux joueurs.");
    }

    private static void affichageMenuInfo() {
        System.out.println("Auteur : Adrien JALLAIS - adrien.jallais@protonmail.com\n");
        System.out.println("Etablissement : IUT Paris Descartes.\n");
        System.out.println("Diplome prepare : DUT Annee Speciale.\n");
        System.out.println("Sujet propose par : POITRENAUD Denis.\n");
        System.out.println("Lien GitHub : < https://github.com/Naedri/OOP-TicTacToe.git > \n");
        System.out.println("Version 1.0.\n");
    }

    private static void affichageMenuJeu() {
        int choixJeu;

        System.out.println("Veuillez taper :");
        System.out.println("< 0 > Pour revenir au Menu Principal");
        System.out.println("< 1 > Pour jouer au TicTacToe.");
        System.out.println("< 2 > Pour jouer au Morpion.");
        System.out.println("< 3 > Pour jouer au TicTacToe extension Forme.");
        System.out.println("< 4 > Pour jouer au TicTacToe extension Permutation.");
    };

    choixJeu = setChoix(0, 4);

    if (choixJeu != 0) {
        int choixGrilleLigne;
        int choixGrilleColonne;
        int choixNbrAlignements = 3;
        int choixForme = 1;
        Forme formeChoisie = null;

        if (choixJeu == 3) {
            // taille grille
            choixGrilleLigne = 12;
            choixGrilleColonne = 12;

            System.out.println("La grille sera composee de " + choixGrilleLigne + " lignes et " + choixGrilleColonne + " colonnes.");

            // forme choix
            System.out.println("Determinons la forme qui permettra de marquer un point.");

            System.out.println(Forme.toStringFormeDispoConsigne());
            System.out.println("Veuillez choisir une forme pas son indice, entre 1 et "

```

```

            + Forme.getListFormesDispo().length + " compris.");
        } else {
            // taille grille
            System.out.println("Determinons la taille de la grille sur laquelle vous allez jouer.");
            System.out.println("Combien de lignes souhaitez vous ?");
            System.out.println("Veuillez choisir un nombre entre 3 et 12 compris");
            choixGrilleLigne = setChoix(3, 12);
            System.out.println("Combien de colonnes souhaitez vous ?");
            System.out.println("Veuillez choisir un nombre entre 3 et 12 compris");
            choixGrilleColonne = setChoix(3, 12);

            System.out.println("La grille sera composee de " + choixGrilleLigne + " lignes et " + choixGrilleColonne + " colonnes.");

            // taille alignement
            System.out.println("Determinons la taille de l'alignement qui permettra de marquer un point.");
            System.out.println("Combien de jetons alignes souhaitez vous ?");
            System.out.println("Veuillez choisir un nombre entre 3 et 5 compris.");
            System.out.println("Attention ce nombre ne doit pas Ãtre plus grand que le nombre de colonnes ou de lignes de votre grille.");
            int choixNbrAlignMax = (choixGrilleColonne >= choixGrilleLigne) ? choixGrilleColonne : choixGrilleLigne;
            choixNbrAlignMax = (choixNbrAlignMax > 5) ? 5 : choixNbrAlignMax;
            choixNbrAlignements = setChoix(3, choixNbrAlignMax);

            System.out.println("Le nombre d alignement choisi est de " + choixNbrAlignements + " jetons.");
        }

        CA_Grille_Partie partie = null;

        switch (choixJeu) {
            case 1:
                System.out.println("La partie de TicTacToe va commencer, preparez-vous !\n");
                partie = new PartieTicTacToe(choixGrilleLigne, choixGrilleColonne, choixNbrAlignements);
                break;
            case 2:
                System.out.println("La partie de Morpion va commencer, preparez-vous !\n");
                partie = new PartieMorpion(choixGrilleLigne, choixGrilleColonne, choixNbrAlignements);
                break;
            case 3:
                System.out.println("La partie de TicTacToe Forme va commencer, preparez-vous !\n");
                partie = new PartieForme(choixForme);
                break;
            case 4:
                System.out.println("La partie de TicTacToe Permutation va commencer, preparez-vous !\n");
                partie = new PartiePermutation(choixGrilleLigne, choixGrilleColonne, choixNbrAlignements);
                break;
        }
    }
}

```

```
lleColonne, choixNbrAlignements);  
                break;  
            }  
            partie.lancerPartie();  
        }  
    }  
}
```

```

//*****
//Jeton.java
//*****
package jeton;

public enum Jeton {

    JETON_VIDE(' '), JETON_X('X'), JETON_O('O');

    private char symbole;

    // Jeton enum
    private Jeton(char jeton) {
        this.symbole = jeton;
    }

    public boolean estEgal(Jeton jetonCible) {
        return (this.symbole == jetonCible.symbole);
    }

    // jeton caractere
    public char getSymbole() {
        return this.symbole;
    }

    public boolean estVideJeton() {
        return this.equals(JETON_VIDE);
    }

    // toString
    public String toString() {
        return "" + '[' + this.symbole + ']'; // "" shortcut to cast from char to
string
    }
}

```

```

//*****
//Direction.java
//*****
package direction;

public enum Direction {
    NORD(-1, 0), NORD_EST(-1, 1), EST(0, 1), SUD_EST(1, 1), SUD(1, 0), SUD_OUEST(1, -
1), OUEST(0, -1),
    NORD_OUEST(-1, -1);

    // DEPLACEMENT(ligne,colonne)
    // enum index commence a 0

    // dÃ©placement relatif de la direction
    private final int dligne, dcolonne;

    private Direction(int deplacementLigne, int deplacementColonne) {
        this.dcolonne = deplacementColonne;
        this.dligne = deplacementLigne;
    }

    /**
     * Retourne le dÃ©placement sur l'axe des x correspondant Ã  la direction
     * courante.
     *
     * @return le dÃ©placement en x
     */
    public int getDcolonne() {
        return dcolonne;
    }

    /**
     * Retourne le dÃ©placement sur l'axe des y correspondant Ã  la direction
     * courante.
     *
     * @return le dÃ©placement en y
     */
    public int getDligne() {
        return dligne;
    }

    /**
     * Retourne une direction inverse Ã  la direction courante.
     *
     * @return la direction opposÃ©e
     */
    public Direction inverser() {
        return Direction.values()[ (this.ordinal() + 4) % 8];
    }
}

```

```

//*****
//Text_Permutation.java
//*****
package tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import jeton.Jeton;
import partie.PartieMorpion;

class Test_Permutation {

    @Test
    void testgrillePermutation() {
        System.out.println("testgrillePermutation EN COURS \n");

        Jeton jx = Jeton.JETON_X;
        Jeton jo = Jeton.JETON_O;

        PartieMorpion grille = new PartieMorpion(3, 3);
        grille.placerJeton(jx, 0, 0);
        grille.placerJeton(jx, 2, 0);
        grille.placerJeton(jx, 2, 1);
        grille.placerJeton(jo, 1, 0);
        grille.placerJeton(jo, 1, 1);
        grille.placerJeton(jo, 1, 2);
        grille.placerJeton(jo, 2, 2);

        grille.placerJeton(jo, 0, 1);
        grille.placerJeton(jx, 0, 2);

        PartieMorpion grille2 = new PartieMorpion(3, 3);
        grille2.placerJeton(jx, 0, 0);
        grille2.placerJeton(jx, 2, 0);
        grille2.placerJeton(jx, 2, 1);
        grille2.placerJeton(jo, 1, 0);
        grille2.placerJeton(jo, 1, 1);
        grille2.placerJeton(jo, 1, 2);
        grille2.placerJeton(jo, 2, 2);

        grille2.placerJeton(jx, 0, 1);
        grille2.placerJeton(jo, 0, 2);
        grille.permutationJeton(0, 1, 0, 2);

        assertTrue(grille2.estEgaleGrille(grille.getGrille()));

        System.out.println("test Permutation FAIT \n");
    }
}

```

```

//*****
//Text_Forme.java
//*****
package tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import jeton.Jeton;
import partie.Forme;
import partie.PartieForme;
import utilitaires.Utills_Grille_Evaluation_Forme;

class Test_Forme {

    @Test
    void testPartieForme() {
        System.out.println("testPartieForme EN COURS \n");

        PartieForme tour = new PartieForme(3, 3);
        Jeton jx = Jeton.JETON_X;
        Jeton jo = Jeton.JETON_O;
        tour.placerJeton(jo, 0, 0);
        tour.placerJeton(jx, 2, 0);
        tour.placerJeton(jx, 0, 2);
        tour.placerJeton(jx, 2, 1);
        tour.placerJeton(jo, 0, 1);
        tour.placerJeton(jo, 1, 0);
        tour.placerJeton(jo, 1, 1);
        tour.placerJeton(jx, 1, 2);
        tour.placerJeton(jo, 2, 2);

        Forme carre = new Forme(1);
        Forme losange = new Forme(2);
        Forme croix = new Forme(3);
        assertEquals(4, carre.getNbrPoint());
        assertEquals(4, losange.getNbrPoint());
        assertEquals(5, croix.getNbrPoint());

        assertTrue(Utills_Grille_Evaluation_Forme.existeForme(0, 0, tour, carre));
        assertTrue(Utills_Grille_Evaluation_Forme.existeForme(1, 1, tour, carre));
        assertFalse(Utills_Grille_Evaluation_Forme.existeForme(2, 2, tour, carre));

        ;
        ;
        ;
        ;
        ;
        ;
        ;

        // getCoordForme
        int[][] coordTest = new int[][] { { 0, 0 }, { 0, 1 }, { 1, 1 }, { 1, 0 }
    }; // tour du carre dans le sens des

        // aiguilles d un
        e montre

        int[][] coordTest1 = new int[carre.getNbrPoint()][2];
        coordTest1 = Utills_Grille_Evaluation_Forme.getCoordForme(0, 0, tour, carr
e);

        assertEquals(coordTest, coordTest1);

        // getJetonForme

```

```

String jetonCarre = Utills_Grille_Evaluation_Forme.getJetonForme(0, 0, tou
r, carre);

String StrCarre = "O000";

assertEquals("O000", jetonCarre);
;
tour.afficherGrille();
tour.placerJeton(jo, 0, 2);
tour.placerJeton(jx, 0, 1);
jetonCarre = Utills_Grille_Evaluation_Forme.getJetonForme(0, 0, tour, carr
e);

StrCarre = "OX00";
assertEquals(jetonCarre, StrCarre);
jetonCarre = Utills_Grille_Evaluation_Forme.getJetonForme(0, 1, tour, carr
e);

StrCarre = "XOX0";
assertEquals(jetonCarre, StrCarre);

// getJetonFormeAll
tour = new PartieForme(3, 3);
tour.placerJeton(jo, 0, 0);
tour.placerJeton(jx, 2, 0);
tour.placerJeton(jx, 0, 2);
tour.placerJeton(jx, 2, 1);

System.out.println(Utills_Grille_Evaluation_Forme.getJetonFormeAll(1, 1, t
our, carre));
tour.afficherGrille();

tour.placerJeton(jo, 2, 2);

System.out.println(Utills_Grille_Evaluation_Forme.getJetonFormeAll(1, 1, t
our, carre));
tour.afficherGrille();

tour.placerJeton(jo, 1, 1);

System.out.println(Utills_Grille_Evaluation_Forme.getJetonFormeAll(1, 1, t
our, carre));
tour.afficherGrille();

// estCompleteForme
tour = new PartieForme(3, 3);
tour.placerJeton(jo, 0, 0);
tour.placerJeton(jx, 2, 0);
tour.placerJeton(jx, 0, 2);
tour.placerJeton(jx, 2, 1);
tour.placerJeton(jo, 0, 1);
tour.placerJeton(jo, 1, 0);
tour.placerJeton(jo, 1, 1);
tour.placerJeton(jx, 1, 2);
tour.placerJeton(jo, 2, 2);
tour.afficherGrille();

assertTrue(Utills_Grille_Evaluation_Forme.estCompleteForme(0, 0, tour, car
re));
assertTrue(Utills_Grille_Evaluation_Forme.estCompleteForme(1, 1, tour, car
re));
assertTrue(Utills_Grille_Evaluation_Forme.estCompleteForme(0, 1, tour, car
re));
assertTrue(Utills_Grille_Evaluation_Forme.estCompleteForme(1, 0, tour, car
re));

assertFalse(Utills_Grille_Evaluation_Forme.estCompleteForme(0, 2, tour, ca
rre));
assertFalse(Utills_Grille_Evaluation_Forme.estCompleteForme(1, 2, tour, ca
rre));
assertFalse(Utills_Grille_Evaluation_Forme.estCompleteForme(2, 2, tour, ca
rre));

```



```
        rre));
        rre));
        rre));

        System.out.println("testPartieForme FAIT \n");
    }
}
```

```

//*****
//Text_Grille.java
//*****
package tests;

import static org.junit.jupiter.api.Assertions.*;

import java.util.EnumSet;

import org.junit.jupiter.api.Test;

import direction.Direction;
import jeton.Jeton;
import partie.CA_Grille;
import partie.CA_Grille_Partie;
import partie.CA_Grille_Partie_FermetureJeton;
import partie.PartieMorpion;
import partie.PartiePermutation;
import partie.PartieTicTacToe;
import utilitaires.Utills_Grille_Evaluation_Adjacent;

class Test_Grille {

    CA_Grille_Partie_FermetureJeton grilleMorpion = new PartieMorpion();
    CA_Grille_Partie grilleTicTacToe = new PartieTicTacToe();

    @Test
    void testJeton() {
        System.out.println("Test jetons.");
        Jeton jo = Jeton.JETON_O;
        Jeton jo2 = Jeton.JETON_O;
        Jeton jo3 = Jeton.JETON_O;
        grilleMorpion.placerJeton(jo, 0, 0);
        grilleMorpion.placerJeton(jo, 1, 1);
        grilleMorpion.placerJeton(jo, 2, 2);

        Jeton jx = Jeton.JETON_X;
        assertTrue(jo.estEgal(jo2));
        assertTrue(jo.estEgal(jo));
        assertFalse(jo.estEgal(jo3));

        assertEquals('O', jo.getSymbole());
        assertEquals('O', jo2.getSymbole());
        assertEquals('X', jx.getSymbole());

        assertFalse(jo.estVideJeton());

        assertEquals('O', jo2.getSymbole());
        assertTrue(jo2.estEgal(jo3));
    }

    @Test
    void testGrillePlacerJeton() {
        System.out.println("Test placer Jeton.");

        PartieTicTacToe grille = new PartieTicTacToe();

        assertTrue(grille.estVideGrille());
        assertFalse(grille.estPleineGrille());
        assertEquals(3, grille.getLignes());
        assertEquals(3, grille.getColonnes());

        Jeton jo = Jeton.JETON_O;
        Jeton jx = Jeton.JETON_X;

        assertTrue(grille.estVideCellule(0, 0));
        grille.placerJeton(jo, 0, 0);
        assertFalse(grille.estVideCellule(0, 0));
        assertFalse(grille.estVideGrille());
    }

```

```

        assertFalse(grille.estPleineGrille());
        grille.placerJeton(jo, 1, 1);
        grille.placerJeton(jo, 2, 2); // on commence a 0

        grille = new PartieTicTacToe();
        for (int i = 0; i < grille.getLignes(); ++i) {
            for (int j = 0; j < grille.getColonnes(); ++j) {
                grille.placerJeton(jo, i, j);
            }
        }
        assertTrue(grille.estPleineGrille());
        assertFalse(grille.estVideGrille());

        grille = new PartieTicTacToe();
        grille.placerJeton(jx, 2, 2);
        grille.placerJeton(jo, 0, 0);
        assertTrue(grille.getCellule(2, 2).estEgal(jx));
        assertTrue(grille.getCellule(0, 0).estEgal(jo));
        // assertTrue(grille.getCellule(3,2).estEgal(jx)); //ne doit pas marcher car
        // 3 est hors grille

        grille = new PartieTicTacToe(5, 4);
        assertFalse(grille.estPleineGrille());
        assertEquals(5, grille.getLignes());
        assertEquals(4, grille.getColonnes());

        grille.placerJeton(jx, 0, 0);
        grille.placerJeton(jo, 0, 1);

        assertTrue(grille.getCellule(0, 0).estEgal(jx));
        assertTrue(grille.getCellule(0, 1).estEgal(jo));

        grille.placerJeton(jo, 0, 2);
        grille.placerJeton(jx, 1, 0);
        grille.placerJeton(jo, 1, 1);
        grille.placerJeton(jo, 1, 2);
        grille.placerJeton(jx, 2, 0);
        grille.placerJeton(jo, 2, 1);
        grille.placerJeton(jo, 2, 2);

        assertFalse(grille.estPleineGrille());
    }

    @Test
    void testGrilleAdjacent() {
        CA_Grille_Partie_FermetureJeton grille = new PartieMorpion(6, 7);
        assertFalse(grille.estPleineGrille());

        System.out.println("Test sans jeton.");

        for (int i = 0; i < grille.getLignes(); ++i) {
            for (int j = 0; j < grille.getColonnes(); ++j) {
                assertFalse(Utills_Grille_Evaluation_Adjacent.existeAdjacent(i, j, grille));
                // System.out.println("Il n existe pas de cellule non vide a
                // adjacente pour la cellule "+ (i+1)+"ligne "+(j+1)+"colonne.");
            }
        }

        System.out.println("Test avec un jeton O.");

        Jeton jo = Jeton.JETON_O;

        grille.placerJeton(jo, 0, 0);
        assertTrue(Utills_Grille_Evaluation_Adjacent.existeAdjacent(0, 1, grille));
    }

    ;

    assertTrue(Utills_Grille_Evaluation_Adjacent.existeAdjacent(1, 1, grille))
    ;

```

```

        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(1, 0, grille)) ;
;
        // (i==0 && j==0)
        for (int i = 0; i < grille.getLignes(); ++i) {
            for (int j = 0; j < grille.getColonnes(); ++j) {
                if ((i == 0 && j == 1) || (i == 1 && j == 1) || (i == 1 &
& j == 0)) {
                    assertTrue(Utils_Grille_Evaluation_Adjacent.exist
eAdjacent(i, j, grille));
//
                    System.out.println("Il existe une cellule non vid
e adjacente pour la cellule "+ (i+1)+"ligne "+(j+1)+"colonne.");
                } else {
                    assertFalse(Utils_Grille_Evaluation_Adjacent.exis
teAdjacent(i, j, grille));
//
                    System.out.println("Il n existe pas de cellule no
n vide adjacente pour la cellule "+ (i+1)+"ligne "+(j+1)+"colonne.");
                }
            }
        }
        System.out.println("Test avec un jeton X.");
        grille = new PartieMorpion(6, 7);
        assertFalse(grille.estPleineGrille());
        Jeton jx = Jeton.JETON_X;
        grille.placerJeton(jx, 0, 0);
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(0, 1, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(1, 1, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(1, 0, grille))
;
        // (i==0 && j==0)
        for (int i = 0; i < grille.getLignes(); ++i) {
            for (int j = 0; j < grille.getColonnes(); ++j) {
                if ((i == 0 && j == 1) || (i == 1 && j == 1) || (i == 1 &
& j == 0)) {
                    assertTrue(Utils_Grille_Evaluation_Adjacent.exist
eAdjacent(i, j, grille));
//
                    System.out.println("Il existe une cellule non vid
e adjacente pour la cellule "+ (i+1)+"ligne "+(j+1)+"colonne.");
                } else {
                    assertFalse(Utils_Grille_Evaluation_Adjacent.exis
teAdjacent(i, j, grille));
//
                    System.out.println("Il n existe pas de cellule no
n vide adjacente pour la cellule "+ (i+1) +"ligne "+(j+1)+"colonne.");
                }
            }
        }
        System.out.println("Test avec deux jetons.");
        grille = new PartieMorpion(6, 6);
        grille.placerJeton(jo, 0, 0);
        grille.placerJeton(jx, 2, 2);
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(1, 1, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(1, 2, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(1, 3, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(2, 3, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(3, 3, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(3, 2, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(3, 1, grille))
;
        assertTrue(Utils_Grille_Evaluation_Adjacent.existeAdjacent(2, 1, grille))
;
        for (int i = 0; i < grille.getLignes(); ++i) {
            System.out.println("Il existe une cellule non vide adjacente pour
la cellule "+ (i+1)+"ligne "+5+"colonne.");
            System.out.println("Il existe une cellule non vide adjacente pour
la cellule "+ (i+1) +"ligne "+6+"colonne.");
            System.out.println("Il existe une cellule non vide adjacente pour
la cellule "+ 5+"ligne "+(i+1)+"colonne.");
            System.out.println("Il existe une cellule non vide adjacente pour
la cellule "+ 6+"ligne "+(i+1)+"colonne.");
        }
        assertFalse(Utils_Grille_Evaluation_Adjacent.existeAdjacent(4, i,
grille));
        assertFalse(Utils_Grille_Evaluation_Adjacent.existeAdjacent(5, i,
grille));
        assertFalse(Utils_Grille_Evaluation_Adjacent.existeAdjacent(i, 4,
grille));
        assertFalse(Utils_Grille_Evaluation_Adjacent.existeAdjacent(i, 5,
grille));
    }
    assertTrue(Utils_Grille_Evaluation_Adjacent.sontAdjacents(0, 0, 1, 1, gri
lle));
    assertFalse(Utils_Grille_Evaluation_Adjacent.sontAdjacents(0, 0, 2, 2, gr
ille));
    assertTrue(Utils_Grille_Evaluation_Adjacent.sontAdjacents(1, 1, 0, 0, gri
lle));
    assertFalse(Utils_Grille_Evaluation_Adjacent.sontAdjacents(2, 2, 0, 0, gr
ille));
    assertTrue(Utils_Grille_Evaluation_Adjacent.sontAdjacents(1, 1, 2, 2, gri
lle));
    assertTrue(Utils_Grille_Evaluation_Adjacent.sontAdjacents(2, 2, 1, 1, gri
lle));
    assertFalse(Utils_Grille_Evaluation_Adjacent.sontAdjacents(2, 2, 0, 0, gr
ille));
    assertTrue(Utils_Grille_Evaluation_Adjacent.sontAdjacents(1, 2, 1, 1, gri
lle));
    assertFalse(Utils_Grille_Evaluation_Adjacent.sontAdjacents(1, 3, 1, 1, gr
ille));
}
@Test
void testGrilleRemplissageAleatoire() {
    System.out.println("Test remplissage aleatoire.");
    System.out.println("remplissage aleatoire 1");
    CA_Grille grille1 = new PartiePermutation(6, 7);
    CA_Grille grille2 = new PartiePermutation(6, 7);
    CA_Grille grille3 = new PartiePermutation(6, 7);
    grille1.afficherGrille();
    System.out.println("remplissage aleatoire 2");
    grille2.afficherGrille();
    System.out.println("remplissage aleatoire 3");
    grille3.afficherGrille();
    assertFalse(grille1.estEgaleGrille(grille2.getGrille()));
    assertFalse(grille1.estEgaleGrille(grille3.getGrille()));
    assertFalse(grille2.estEgaleGrille(grille3.getGrille()));
}

```

```

}

@Test
void testGrillePermutationJeton() {
    System.out.println("Test permutation jeton.");

    Jeton jx = Jeton.JETON_X;
    Jeton jo = Jeton.JETON_O;

    PartiePermutation partie = new PartiePermutation(4, 4);
    PartiePermutation partieC = new PartiePermutation(4, 4);

    PartiePermutation grille = new PartiePermutation();

    partie.placerJeton(jo, 0, 0);
    partie.placerJeton(jx, 1, 1);
    partie.placerJeton(jo, 2, 2);
    partie.placerJeton(jo, 1, 2);

    partieC.placerJeton(jo, 0, 0);
    partieC.placerJeton(jx, 1, 1);
    partieC.placerJeton(jo, 2, 2);
    partieC.placerJeton(jo, 1, 2);

    grille.permutationJeton(1, 1, 2, 2);
    assertFalse(partie.estEgaleGrille(partieC.getGrille()));
    grille.permutationJeton(1, 1, 2, 2);
    assertFalse(partie.estEgaleGrille(partieC.getGrille()));

    grille.permutationJeton(0, 0, 1, 1);
    assertFalse(partie.estEgaleGrille(partieC.getGrille()));
    grille.permutationJeton(0, 0, 1, 1);
    assertFalse(partie.estEgaleGrille(partieC.getGrille()));

    grille.permutationJeton(1, 2, 1, 1);
    assertFalse(partie.estEgaleGrille(partieC.getGrille()));
    grille.permutationJeton(1, 1, 1, 2);
    assertFalse(partie.estEgaleGrille(partieC.getGrille()));
}

//
// @Test
// void testGrilleFermetureJeton(){
//     System.out.println("Test fermeture jeton.");
//
//     Jeton jx = Jeton.JETON_X ;
//     Jeton jo = Jeton.JETON_O ;
//
//     PartieMorpion grille = new PartieMorpion(4,4);
//
//     grille.placerJeton(jo, 0, 2);
//     grille.placerJeton(jx, 1, 0);
//     grille.placerJeton(jo, 1, 1);
//     grille.placerJeton(jo, 1, 2);
//     grille.placerJeton(jx, 2, 0);
//     grille.placerJeton(jo, 2, 1);
//     grille.placerJeton(jo, 2, 2);
//     grille.placerJeton(jo, 2, 3);
//
//     grille.ouvertToFermeJeton(1, 2);
//
//     assertTrue(grille.getCellule(1, 1).estOuvert());
//     assertFalse(grille.getCellule(1, 2).estOuvert());
//
// }

@Test
void testGrille() {
    CA_Grille grille = new PartieTicTacToe(4, 4);

    Jeton jx = Jeton.JETON_X;

```

```

    Jeton jo = Jeton.JETON_O;
    grille.placerJeton(jo, 0, 2);
    grille.placerJeton(jx, 1, 0);
    grille.placerJeton(jo, 1, 1);
    grille.placerJeton(jo, 1, 2);
    grille.placerJeton(jx, 2, 0);
    grille.placerJeton(jo, 2, 1);
    grille.placerJeton(jo, 2, 2);
    grille.placerJeton(jo, 2, 3);
    assertEquals(6, grille.getNbrJeton(jo));
    assertEquals(2, grille.getNbrJeton(jx));
    grille.placerJeton(jo, 3, 0);
    grille.placerJeton(jx, 0, 3);
    grille.placerJeton(jx, 3, 3);
    assertEquals(7, grille.getNbrJeton(jo));
    assertEquals(4, grille.getNbrJeton(jx));
    grille.placerJeton(jx, 1, 3);
    assertEquals(7, grille.getNbrJeton(jo));
    assertEquals(5, grille.getNbrJeton(jx));
}

@Test
void testGrilleDifferentesCellules() {
    System.out.println("Test Differentes Cellules.");

    Jeton jx = Jeton.JETON_X;
    Jeton jo = Jeton.JETON_O;

    CA_Grille grille = new PartieTicTacToe(4, 4);

    grille.placerJeton(jo, 0, 2);
    grille.placerJeton(jx, 1, 0);
    grille.placerJeton(jo, 1, 1);
    grille.placerJeton(jo, 1, 2);
    grille.placerJeton(jx, 2, 0);
    grille.placerJeton(jo, 2, 1);

    assertFalse(grille.sontDifferentes(0, 1, 0, 1));
    assertFalse(grille.sontDifferentes(1, 1, 1, 1));
    assertFalse(grille.sontDifferentes(0, 0, 0, 0));
    assertFalse(grille.sontDifferentes(1, 0, 1, 0));
    assertFalse(grille.sontDifferentes(2, 2, 2, 2));
    assertFalse(grille.sontDifferentes(1, 2, 1, 2));
    assertFalse(grille.sontDifferentes(0, 2, 0, 2));

    assertTrue(grille.sontDifferentes(1, 0, 0, 1));
    assertTrue(grille.sontDifferentes(0, 1, 1, 0));
    assertTrue(grille.sontDifferentes(1, 0, 0, 1));
    assertTrue(grille.sontDifferentes(0, 1, 1, 0));
    assertTrue(grille.sontDifferentes(1, 2, 0, 1));
    assertTrue(grille.sontDifferentes(1, 2, 2, 1));
}

@Test
void testGrilleDeplacement() {
    PartieTicTacToe grille = new PartieTicTacToe(5, 5);
    grille.placerJeton(Jeton.JETON_X, 2, 2);
    grille.afficherGrille();

    assertTrue(grille.existeNextCellule(0, 0, 1, Direction.SUD));
    assertTrue(grille.existeNextCellule(0, 0, 2, Direction.SUD));
    assertTrue(grille.existeNextCellule(0, 0, 3, Direction.SUD));
    assertTrue(grille.existeNextCellule(0, 0, 4, Direction.SUD));
    assertFalse(grille.existeNextCellule(0, 0, 5, Direction.SUD));
    assertFalse(grille.existeNextCellule(0, 0, 6, Direction.SUD));

    for (Direction oneDirection : EnumSet.range(Direction.NORD, Direction.NOR
D_OUEST)) {

```

```

        assertTrue(grille.existeNextCellule(1, 1, 1, oneDirection));
    }

    for (Direction oneDirection : EnumSet.range(Direction.NORD, Direction.NOR
D_OUEST)) {
        assertTrue(grille.existeNextCellule(2, 2, 2, oneDirection));
    }

    for (Direction oneDirection : EnumSet.range(Direction.NORD, Direction.NOR
D_OUEST)) {
        assertFalse(grille.existeNextCellule(2, 2, 3, oneDirection));
    }

    int[] coordTest;

    coordTest = new int[] { 1, 0 };
    assertEquals(coordTest, grille.coordNextJeton(0, 0, 1, Direction.SUD
));

    coordTest = new int[] { 2, 0 };
    assertEquals(coordTest, grille.coordNextJeton(0, 0, 2, Direction.SUD
));

    coordTest = new int[] { 3, 0 };
    assertEquals(coordTest, grille.coordNextJeton(0, 0, 3, Direction.SUD
));

    coordTest = new int[] { 2, 4 };
    assertEquals(coordTest, grille.coordNextJeton(2, 2, 2, Direction.EST
));

    coordTest = new int[] { 2, 0 };
    assertEquals(coordTest, grille.coordNextJeton(2, 2, 2, Direction.OUE
ST));

    coordTest = new int[] { 0, 0 };
    assertEquals(coordTest, grille.coordNextJeton(2, 2, 2, Direction.NOR
D_OUEST));

    coordTest = new int[] { 4, 0 };
    assertEquals(coordTest, grille.coordNextJeton(2, 2, 2, Direction.SUD
_OUEST));
    }
}

```

```

//*****
//Test_forme.java
//*****
package tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import partie.Forme;

class Test_forme {

    @Test
    void test() {

        int[] distance;
        int[] orientation;

        Forme forme;

        // carre
        forme = new Forme(1);
        distance = new int[] { 1, 1, 1, 1 };
        orientation = new int[] { 2, 4, 6, 0 };
        assertEquals(distance, forme.transForme(0).getDistance());
        assertEquals(orientation, forme.transForme(0).getOrientation());

        orientation = new int[] { 0, 2, 4, 6 };
        assertEquals(distance, forme.transForme(1).getDistance());
        assertEquals(orientation, forme.transForme(1).getOrientation());

        orientation = new int[] { 0, 2, 4, 6 };
        assertEquals(distance, forme.transForme(5).getDistance());
        assertEquals(orientation, forme.transForme(5).getOrientation());

    }

}

```

```

//*****
//Text_TicTacToe.java
//*****
package tests;

```

```

import static org.junit.jupiter.api.Assertions.*;

```

```

import org.junit.jupiter.api.Test;

```

```

import direction.Direction;
import jeton.Jeton;
import partie.PartieTicTacToe;
import utilitaires.Utills_Grille_Evaluation_Alignement;

```

```

class Test_TicTacToe {

```

```

    @Test

```

```

    void testPartieTicTacToe() {
        System.out.println("testPartieTicTacToe EN COURS \n");

```

```

        PartieTicTacToe grille = new PartieTicTacToe();
        Jeton jo = Jeton.JETON_0;

```

```

        assertFalse(grille.estPleineGrille());
        assertEquals(3, grille.getLignes());
        assertEquals(3, grille.getColonnes());

```

```

        assertTrue(grille.estVideCellule(0, 0));
        grille.placerJeton(jo, 0, 0);
        assertFalse(grille.estVideCellule(0, 0));
        grille.placerJeton(jo, 1, 1);
        grille.placerJeton(jo, 2, 2); // on commence a 0

```

```

        // j0 sans autres jetons

```

```

        assertTrue(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, Di
rection.NORD_OUEST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.NORD, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.NORD_EST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.EST, grille));
        assertTrue(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, Di
rection.SUD_EST, grille)); // attention

```

```

        // c est

```

```

        // true

```

```

        // parce que

```

```

        // la

```

```

        // fonction

```

```

        // regarde dans les deux sens

```

```

        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.SUD, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.SUD_OUEST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.OUEST, grille));
        assertTrue(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, Di
rection.NORD_OUEST, grille));

```

```

// ajout autres jetons
Jeton jx = Jeton.JETON_X;
grille.placerJeton(jx, 2, 0);
grille.placerJeton(jx, 2, 1);

```

```

// j0 avec autres jetons

```

```

        assertTrue(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, Di
rection.NORD_OUEST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.NORD, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.NORD_EST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.EST, grille));
        assertTrue(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, Di
rection.SUD_EST, grille)); // attention

```

```

        // c est

```

```

        // true

```

```

        // parce que

```

```

        // la

```

```

        // fonction

```

```

        // regarde dans les deux sens

```

```

        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.SUD, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.SUD_OUEST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, D
irection.OUEST, grille));
        assertTrue(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 2, 3, Di
rection.NORD_OUEST, grille));

```

```

        // jx avec jo mais ne marque pas de points

```

```

        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.NORD_OUEST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.NORD, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.NORD_EST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.EST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.SUD_EST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.SUD, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.SUD_OUEST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.OUEST, grille));
        assertFalse(Utills_Grille_Evaluation_Alignement.isDirectInAlign(2, 1, 3, D
irection.NORD_OUEST, grille));

```

```

        // multiple alignements

```

```

        // alignements en diagonal

```

```

        assertEquals(0, Utills_Grille_Evaluation_Alignement.nbrDirectAvecAlign(2,
0, 3, grille));
        assertEquals(1, Utills_Grille_Evaluation_Alignement.nbrDirectAvecAlign(2,
2, 3, grille));

```

```

        // alignements horizontaux

```

```
        grille.placerJeton(jo, 1, 2);
        grille.placerJeton(jo, 1, 0);
        assertEquals(0, Utils_Grille_Evaluation_Alignement.nbrDirectAvecAlign(2,
0, 3, grille));
        assertEquals(0, Utils_Grille_Evaluation_Alignement.nbrDirectAvecAlign(2,
1, 3, grille));
        assertEquals(1, Utils_Grille_Evaluation_Alignement.nbrDirectAvecAlign(2,
2, 3, grille));
        assertEquals(1, Utils_Grille_Evaluation_Alignement.nbrDirectAvecAlign(1,
2, 3, grille));

        System.out.println("testPartieTicTacToe FAIT \n");
    }
}
```



```

//*****
//Text_Saisie.java
//*****
package tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import interaction.Messages_Saisie;

class Test_Saisie {

    @Test
    void test() {
        String s1 = "1-2";
        String s10 = "01-0002";
        String s1S = "1 - 2";

        String s1F = "22";
        String s2F = "1-2-3";
        String s3F = "1 - &";
        String s4F = "-1-2-";
        String s5F = "a-b";
        String s6F = "-1-2";

        assertTrue(Messages_Saisie.estValideSaisie(s1));
        assertTrue(Messages_Saisie.estValideSaisie(s10));
        assertTrue(Messages_Saisie.estValideSaisie(s1S));

        assertFalse(Messages_Saisie.estValideSaisie(s1F));
        assertFalse(Messages_Saisie.estValideSaisie(s2F));
        assertFalse(Messages_Saisie.estValideSaisie(s3F));
        assertFalse(Messages_Saisie.estValideSaisie(s4F));
        assertFalse(Messages_Saisie.estValideSaisie(s5F));
        assertFalse(Messages_Saisie.estValideSaisie(s6F));

        int s1T[] = { 1, 2 };
        int s1TR[] = Messages_Saisie.extraireCelluleSaisie(s1);
        assertEquals(s1T[0], Messages_Saisie.extraireCelluleSaisie(s1)[0]);
        assertEquals(s1T[1], Messages_Saisie.extraireCelluleSaisie(s1)[1]);
        assertEquals(s1T[0], s1TR[0]);
        assertEquals(s1T[1], s1TR[1]);
    }
}

```

```

//*****
//Text_Morpion.java
//*****
package tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import direction.Direction;
import jeton.Jeton;
import partie.PartieMorpion;
import utilitaires.Utills_Grille_Evaluation_Alignement;

class Test_Morpion {

    @Test
    void testPartieMorpion() {
        System.out.println("testPartieMorpion EN COURS \n");

        Jeton jx = Jeton.JETON_X;
        Jeton jo = Jeton.JETON_O;

        PartieMorpion tour = new PartieMorpion();
        PartieMorpion grille = tour;
        // nbrAlignementXD

        tour.placerJeton(jo, 1, 2);
        tour.placerJeton(jo, 1, 0);
        tour.placerJeton(jo, 0, 0);
        tour.placerJeton(jx, 1, 1);

        tour.afficherGrille();
        String stro = tour.getLigneJetonOouF(1, 3, 3, Direction.OUEST);
        System.out.println(stro);
        assertEquals("OXO", stro);
        assertEquals('X', tour.getSymboleJetonOouF(1, 1));

        tour.ouvertToFermeJeton(1, 1);
        stro = tour.getLigneJetonOouF(1, 3, 3, Direction.OUEST);
        assertEquals("OxO", stro);
        assertEquals('x', tour.getSymboleJetonOouF(1, 1));

        tour.placerJeton(jx, 3, 0);
        assertEquals('X', tour.getSymboleJetonOouF(3, 0));
        tour.ouvertToFermeJeton(3, 0);
        assertEquals('x', tour.getSymboleJetonOouF(3, 0));

        tour.placerJeton(jo, 3, 1);
        assertEquals('O', tour.getSymboleJetonOouF(3, 1));
        tour.ouvertToFermeJeton(3, 1);
        assertEquals('o', tour.getSymboleJetonOouF(3, 1));

        tour.afficherGrille();
        tour.placerJeton(jo, 0, 1);
        tour.placerJeton(jo, 0, 2);
        tour.afficherGrille();
        assertEquals('O', tour.getSymboleJetonOouF(0, 0));

        tour.fermeAlignementXD(0, 2, 3);

        tour.afficherGrille();

        assertEquals(2, Utills_Grille_Evaluation_Alignement.nbrDirectAvecAlign(1,
2, 2, tour));
        //
        les inverses
        Direction[] directions = new Direction[] { Direction.NORD, Direction.SUD_
EST };

```

```

        assertEquals(directions, Utills_Grille_Evaluation_Alignement.getDirec
tAlign(1, 2, 2, tour));
        grille.placerJeton(jo, 2, 2);
        assertEquals(Direction.NORD, Utills_Grille_Evaluation_Alignement.getDirect
Align(1, 2, 2, tour)[0]); // pas sud

        // mais nord

        // et pas

        // est car

        // nord est
        // le premier a faire un alignement

        grille.placerJeton(jx, 2, 0);
        grille.placerJeton(jx, 2, 1);

        assertEquals(1, Utills_Grille_Evaluation_Alignement.nbrDirectAvecAlign(1,
2, 3, tour));
        assertEquals(2, Utills_Grille_Evaluation_Alignement.nbrDirectAvecAlign(1,
2, 2, tour));
        assertEquals(Direction.NORD, Utills_Grille_Evaluation_Alignement.getDirect
Align(1, 2, 2, tour)[0]); // pas sud

        // mais nord

        // et pas

        // est car

        // nord est
        // le premier a faire un alignement

        // fermeAlignementXD avec 3
        grille = new PartieMorpion(6, 6);
        grille.placerJeton(jx, 0, 1);
        grille.placerJeton(jx, 0, 2);
        grille.placerJeton(jx, 0, 3);
        grille.placerJeton(jx, 0, 4);
        grille.placerJeton(jo, 1, 1);
        grille.placerJeton(jx, 1, 2);
        grille.fermeAlignementXD(0, 2, 3);

        PartieMorpion grilleFerme = new PartieMorpion(6, 6);
        grilleFerme.placerJeton(jx, 0, 1);
        grilleFerme.placerJeton(jx, 0, 2);
        grilleFerme.placerJeton(jx, 0, 3);
        grilleFerme.placerJeton(jx, 0, 4);
        grilleFerme.placerJeton(jo, 1, 1);
        grilleFerme.placerJeton(jx, 1, 2);
        grilleFerme.ouvertToFermeJeton(0, 2);
        grilleFerme.ouvertToFermeJeton(0, 3);
        grilleFerme.ouvertToFermeJeton(0, 4);

        assertTrue(grille.estEgaleGrille(grilleFerme.getGrille()));
        assertTrue(grilleFerme.estEgaleGrille(grille.getGrille()));

        System.out.println("fermerAlignement XD avec 3");
        grille.afficherGrille();
        grilleFerme.afficherGrille();

```

```

//
    fermeAlignementXD avec 4
    grille = new PartieMorpion(6, 6);
    grille.placerJeton(jx, 0, 1);
    grille.placerJeton(jx, 0, 2);
    grille.placerJeton(jx, 0, 3);
    grille.placerJeton(jx, 0, 4);
    grille.placerJeton(jo, 1, 1);
    grille.placerJeton(jx, 1, 2);
    grille.afficherGrille();
    grille.getLigneJetonOouF(0, 2, 4, Direction.EST);
    grille.appartientAlignOouF(0, 2, 4, Direction.EST);
    grille.isDirectAvecAlignOouF(0, 2, 4);
    grille.fermeAlignementXD(0, 2, 4);

    grilleFerme = new PartieMorpion(6, 6);
    grilleFerme.placerJeton(jx, 0, 1);
    grilleFerme.placerJeton(jx, 0, 2);
    grilleFerme.placerJeton(jx, 0, 3);
    grilleFerme.placerJeton(jx, 0, 4);
    grilleFerme.placerJeton(jo, 1, 1);
    grilleFerme.placerJeton(jx, 1, 2);
    grilleFerme.ouvertToFermeJeton(0, 1);
    grilleFerme.ouvertToFermeJeton(0, 2);
    grilleFerme.ouvertToFermeJeton(0, 3);
    grilleFerme.ouvertToFermeJeton(0, 4);

    assertTrue(grille.estEgaleGrille(grilleFerme.getGrille()));
    assertTrue(grilleFerme.estEgaleGrille(grille.getGrille()));

//

    System.out.println("fermerAlignement XD avec 4");
    grille.afficherGrille();
    grilleFerme.afficherGrille();

    // fermer avec Coordonnees
    // a partir de getCoordAlignJetonsXDOouF
    grille = new PartieMorpion(3, 3);
    grille.placerJeton(jx, 0, 0);
    grille.placerJeton(jx, 2, 0);
    grille.placerJeton(jx, 2, 1);
    grille.placerJeton(jo, 1, 0);
    grille.placerJeton(jo, 1, 1);
    grille.placerJeton(jo, 1, 2);
    grille.placerJeton(jo, 2, 2);

    grille.placerJeton(jo, 0, 1);
    grille.placerJeton(jx, 0, 2);

    int[][] jetonsAFermerEvalue = grille.getCoordAlignJetonsXDOouF(0, 2, 3);
    int[][] jetonsAFermerExpected = new int[][] { { 0, 2 }, { 1, 2 }, { 2, 2 } };

    assertEquals(jetonsAFermerExpected, jetonsAFermerEvalue);

    System.out.println("fermerAlignement XD avec coord");

    System.out.println("testPartieMorpion FAIT \n");
}

}

```

```

//*****
//MessagePermutation.java
//*****
package interaction;

import partie.Joueur;

public class MessagePermutation {

    /**
     * Message indiquant que "le joueur X/O a choisi de permute son jeton dans la
     * cellule ligne ..
     *
     * @param j          joueur ayant realiser la permutation
     * @param celluleCoord1 premier cellule
     * @param celluleCoord2 seconde cellule
     * @return string a afficher
     */
    public static String afficherMessageCoupJoue(Joueur j, int[] celluleCoord1, int[]
celluleCoord2) {
        assert (j != null);
        assert (celluleCoord1 != null && celluleCoord2 != null); // on s assure q
ue setSaisieCellule a ete appelee
        int saisieLigne1 = celluleCoord1[0] + 1;
        int saisieColonne1 = celluleCoord1[1] + 1;
        int saisieLigne2 = celluleCoord2[0] + 1;
        int saisieColonne2 = celluleCoord2[1] + 1;
        return "Le joueur " + j.getJeton().getSymbole() + " a choisi de permuter
la cellule ligne [" + saisieLigne1
                + "] colonne [" + saisieColonne1 + "], avec la cellule li
gne [" + saisieLigne2 + "] colonne ["
                + saisieColonne2 + "].\n";
    }
}

```

```

//*****
//MessagePlacement.java
//*****

package interaction;

import partie.Joueur;

public class MessagePlacement {

    /**
     * Message indiquant que "le joueur X/O a choisi de placer son jeton dans la
     * cellule ligne ..
     *
     * @param j          joueur dont le symbole sera utilise
     * @param celluleCoord1 tab de coordonnees de la cellule = * @return string a
     *                  afficher
     */
    public static String afficherMessageCoupJoue(Joueur j, int[] celluleCoord1) {
        assert (j != null);
        assert (celluleCoord1 != null); // on s assure que setSaisieCellule a ete
appelée
        int saisieLigne1 = celluleCoord1[0] + 1;
        int saisieColonne1 = celluleCoord1[1] + 1;
        return "Le joueur " + j.getJeton().getSymbole() + " a choisi de placer so
n jeton dans la cellule ligne ["
            + saisieLigne1 + "] colonne [" + saisieColonne1 + "].\n";
    }
}

```

```

//*****
//Messages_Saisie.java
//*****
package interaction;

import java.util.Scanner;

import jeton.*;
import partie.Joueur;

public class Messages_Saisie {

    /**
     * saisieCellule input du joueur sous format de table Ã 1 colonne 2 lignes
     * messageResultat victoire ou non (cÃ d "Continuez la partie!" ou "Le Joueur X a
     * gagnÃ la partie") messageTour indique qui doit jouer messagePermutation quel
     * cellule de la grille ont ÃtÃ permutÃes
     */

    /**
     * saisieCellule sous forme de int[] saisieCellule[0] : ligne saisieCellule[1] :
     * colonne definie la cellule saisie par le joueur securise le choix de la
     * cellule par le joueur (l entrÃe ) ne verifie pas que la cellule est pleine ou
     * non
     */
    public static int[] saisirCellule(Jeton[][] grille) {
        boolean saisieCorrecte = false;
        int[] cellule = new int[2];

        while (!saisieCorrecte) {

            Scanner sc = new Scanner(System.in);
            System.out.println(
                "Veuillez choisir une case,\nex: pour la case de
la ligne 1 a la colonne 2, tapez : \n1-2\npuis appuyez sur \'Entree\'.\n");
            String saisie = sc.nextLine();
            System.out.println("Vous avez tape : " + saisie + ".\n");

            try {
                if (estValideSaisie(saisie)) {
                    cellule = extraitCelluleSaisie(saisie);
                    // il faut tout de meme REverifier que la dimensi
                    // decrementation pour s adapter Ã 1 aff
                    if (cellule.length == 2) {
                        --cellule[0];
                        --cellule[1];
                        if (cellule[0] >= 0 && cellule[0] < grill
                            && cellule[1] < grille[0]
                                // sortie de la boucle
                                saisieCorrecte = true;
                    } else
                        System.out.println("Il faut chois
ir une ligne et une colonne de la grille, recommencez.\n");
                } else
                    System.out.println("La saisie est incorrecte, rec
ommencez.\n");
            }

            catch (java.lang.NumberFormatException el) {
                System.out.println("Le format est invalide, recommencez.\n");
            }
        }
        return cellule;
    }
}

```

```

}

public static boolean estValideSaisie(String saisie) {
    // format : entier[espaces]-[espaces]entier (ligne puis colonne)
    // \s Matches the whitespace. Equivalent to [\t\n\r\f].
    // \d Matches the digits. Equivalent to [0-9].
    // re* Matches 0 or more occurrences of the preceding expression (e)
    // re+ Matches 1 or more of the previous thing (e).
    return saisie.matches("\\s*\\d+\\s*-\\s*\\d+\\s*");
}

public static int[] extraitCelluleSaisie(String saisie) {
    assert (estValideSaisie(saisie));

    String[] split = saisie.split("-"); // https://stackoverflow.com/a/985726

    int[] coordonnees = new int[split.length];
    for (int i = 0; i < coordonnees.length; ++i)
        coordonnees[i] = Integer.parseInt(split[i]);
    return coordonnees;
}

// PROF VERSION
public static void analyseSaisie(String s) {
    if (s.matches("\\s*\\d+\\s*-\\s*\\d+\\s*")) {
        Scanner sc = new Scanner(s);
        sc.useDelimiter("\\s*-\\s*");
        int ligne = sc.nextInt();
        int colonne = sc.nextInt();
        System.out.println("ligne = " + ligne + ", colonne = " + colonne);
        sc.close();
    } else
        System.out.println("Coup invalide");
}

/**
 * messageCellule sous forme string indiquant ce que le joueur a selectionne
 * comme cellule doit etre appele Ã la suite de saisirCellule()
 *
 * @return Le joueur X (resp O) a saisie la cellule ligne [i], colonne[j].
 */
public static String afficherMessageCellule(Joueur j, int[] celluleCoord) {
    assert (j != null);
    assert (celluleCoord != null); // on s assure que setSaisieCellule a ete
    appelee

    int saisieLigne = celluleCoord[0] + 1;
    int saisieColonne = celluleCoord[1] + 1;
    return "Le joueur " + j.getJeton().getSymbole() + " a choisi la cellule l
igne [" + saisieLigne + ", colonne [" + saisieColonne + "].\n";
}

/**
 * messageResultat indiquant victoire ou non il faut qu elle soit appelee aprÃs
 * un coup joue (cÃ d "C est au joueur suivant" ou "Le Joueur X a gagnÃ la
 * partie")
 *
 * @param m match en cours
 * @param joueurActuel joueur qui vient de jouer
 * @return
 */
public static String afficherMessageFinTour(Joueur joueurActuel) {
    assert (joueurActuel != null);
    return "Le joueur " + joueurActuel.getJeton().getSymbole() + " a termine
son tour.\n";
}
}

```

```

/**
 * messageResultat indiquant victoire ou non il faut qu elle soit appelee aprÃs
 * un coup joue (cÃ d "C est au joueur suivant" ou "Le Joueur X a gagnÃ la
 * partie")
 *
 * @param m          match en cours
 * @param joueurActuel joueur qui vient de jouer
 * @return
 */
public static String afficherMessageResultat(Joueur joueurActuel, Joueur joueurAutre) {
    assert (joueurActuel != null && joueurAutre != null);
    assert (joueurActuel != joueurAutre);
    String messageResultat = "";

    if (joueurActuel.getScore() == joueurAutre.getScore()) {
        messageResultat = "C est un match nul.\n";
    } else {
        if (joueurActuel.getScore() > joueurAutre.getScore()) {
            messageResultat = "C est un match victorieux pour le joueur " + joueurActuel.getJeton().getSymbole() + ".\n";
        }
        if (joueurActuel.getScore() < joueurAutre.getScore()) {
            messageResultat = "C est un match victorieux pour le joueur " + joueurAutre.getJeton().getSymbole() + ".\n";
        }
    }
    messageResultat += "Le joueur " + joueurActuel.getJeton().getSymbole() + " a marque " + joueurActuel.getScore() + " points.\n";
    messageResultat += "Le joueur " + joueurAutre.getJeton().getSymbole() + " a marque " + joueurAutre.getScore() + " points.\n";

    return messageResultat;
}

/**
 * messageTour
 *
 * @param j joueur dont le tour commence
 * @return
 */
public static String afficherMessageDebutTour(Joueur j) {
    assert (j != null);
    return "C'est au joueur " + j.getJeton().getSymbole() + " de jouer.\n";
}

public static String afficherMessageCoupMarquant(Joueur joueur) {
    assert (joueur != null);
    return "Le joueur " + joueur.getJeton().getSymbole() + " a marque un point.\n";
}
}

```