Université de Paris

BPO

S2 période C
2020-2021

Rapport de projet
The Game – Le Duel

Slim BEN DAALI
et Anthony ZAKANI
Groupe 103 et 102

TABLE DES MATIERES

I. Présentation du problème
Page 3

II. Structure du code et tests unitaires
Pages 4 à 7

III. Bilan du projet
Page 8

IV. Classes du projet
Application : p. 10
Joueur : p. 11
Partie : p. 15
MainJ : p. 19
PileJ : p. 21
PiocheJ : p. 22
Tests : p. 23

À travers ce rapport, nous vous présenterons les difficultés et défis qui ont rythmé la réalisation du projet de jeu autour du concept du jeu de société « The Game – Le Duel ».

À l'aide du langage Java et des principes de la programmation orientée objet (POO), nous avons pu concevoir un jeu permettant de faire une partie du jeu de société « The Game – Le Duel ».

Présentation du projet

Au cours de la période C du semestre 2, nous avons dû constituer de nouveaux binômes de programmation afin de nous lancer dans un nouveau défi : programmer une version Java du jeu de société de Steffen Benndorf et Reinhard Staupe « ***The Game – Le duel*** ».
Forts de nos acquis pendant le premier semestre, l'objectif était cette fois-ci de changer de mode de pensée. En passant du C++ au Java, nous nous sommes aussi initiés à la **Programmation Orientée Objet (POO)**. Ces nouvelles notions sont au cœur de la programmation en Java et leur compréhension est primordiale pour réussir à écrire un programme cohérent.

Le but du programme est de permettre aux joueurs de faire entre eux une partie de ***The Game*** : il faut donc qu'ils aient à disposition deux piles de cartes, une pioche et un certain nombre de cartes en main pour jouer. C'est donc d'autant plus de concepts à réimaginer puis coder.

Dès les premières semaines, nous nous sommes coordonnés pour travailler en collaboration sur une suite d'outils commune : Github pour le versionnage et la gestion des modifications, IntelliJ pour l'IDE, GitLive puis Code With Me pour le travail en collaboration en temps réel sur le code, SonarLint pour la qualité du code ainsi que Java 1.8 pour version Java ;

Une fois le sujet reçu, nous avons décidé de commencer par mettre en place 3 classes jugées à ce moment principales : le **main,** la classe **Joueur** et la classe **Carte.** Tout au long du projet, nous avons pensé en premier lieu à la structuration de notre code. Cela a mené à tout un tas de changements (la classe **Carte** a par exemple été retirée), signe donc de la réflexion qui a été menée dans la conception du projet. Le respect du principe d'encapsulation était également l'une des clés du projet.

Structure du code

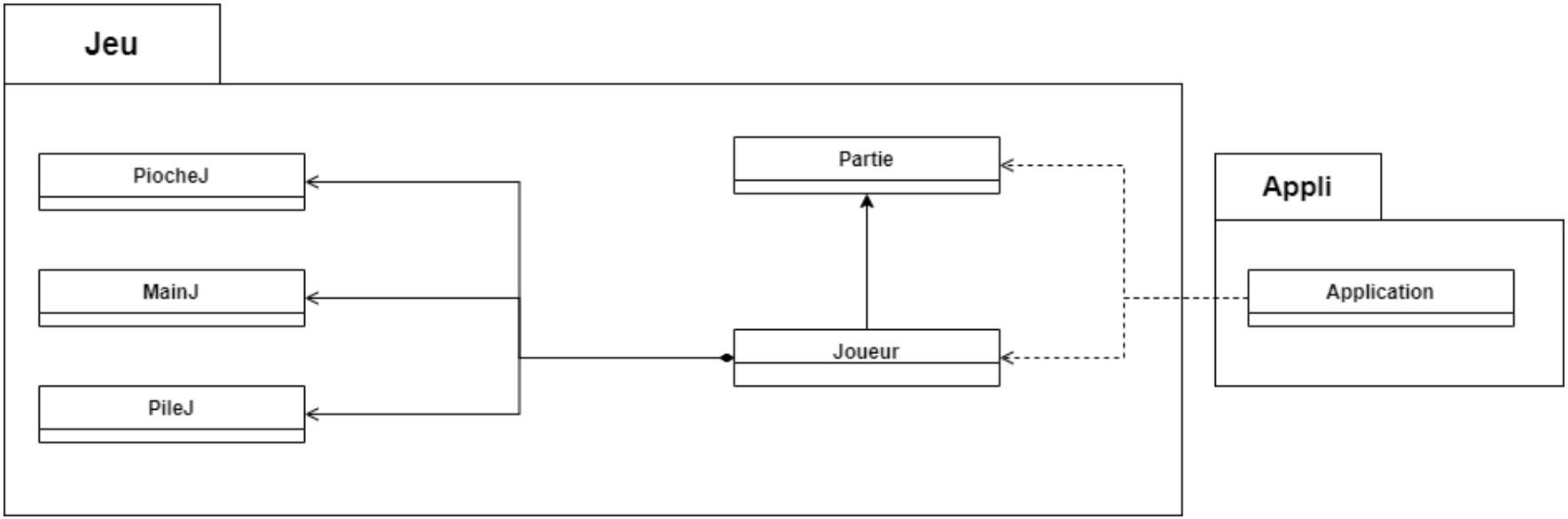
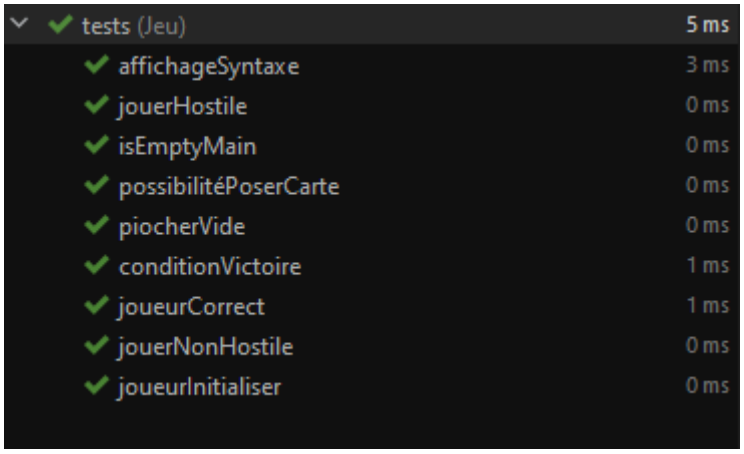


Diagramme UML des classes de l'application

Tests unitaires



- -> **affichageSyntaxe** teste différents éléments de la syntaxe du jeu (affichage des tours, affichage d'une main) et leur évolution
  - Réponse attendue : réponse différente avant et après test
  - Réponse obtenue : réponse différente avant et après test
- -> **jouerHostile** teste les conséquences du jeu sur la pile d'un adversaire (en termes de statut de jeu et de nombre de cartes piochées)
  - Réponse attendue : le joueur pioche 6 – le nombre de cartes déjà en main en conséquence
  - Réponse obtenue : le joueur pioche 6 – le nombre de cartes déjà en main -> cf. cas de retour 3 dans la fonction remplirMain() obtenu
- -> **jouerNonHostile** teste la réaction du jeu lorsque le joueur ne joue que sur ses piles (en termes de statut du jeu et de nombre de cartes piochées)
  - Réponse attendue : le joueur pioche deux cartes quoi qu'il arrive dans ce cas
  - Réponse obtenue : le joueur pioche deux cartes -> cf. cas de retour 2 pour remplirMain() obtenu
- -> **isEmptyMain** teste si la vérification du contenu de la main retourne bien un statut de main vide lorsque la main est vide
  - Réponse attendue : la réponse à isEmpty() est « faux »
  - Réponse obtenue : la réponse à isEmpty() est « faux »

- -> **possibilitéPoserCarte** teste la possibilité de jouer dès la création des joueurs
  - Réponse attendue : les joueurs peuvent jouer
  - Réponse obtenue : les joueurs peuvent jouer
- -> **piocherVide** teste l'impossibilité pour un joueur de piocher lorsque sa pioche est vide (et donc les mécanismes empêchant le programme de planter si c'est le cas)
  - Réponse attendue : le joueur ne peut pas piocher
  - Réponse obtenue : le joueur pouvait piocher
    - Correction effectuée
  - Réponse après correction : le joueur ne peut pas piocher -> cf. cas implémenté de main vide dans remplirMain()
- -> **joueurCorrect** teste le jeu par un joueur d'une carte qu'il possède
  - Réponse attendue : le joueur peut jouer une carte qu'il possède
  - Réponse obtenue : le joueur peut jouer une carte qu'il possède -> cf. réponse de jouer() avec la carte est « vrai »
- -> **joueurInitialiser** teste la bonne initialisation d'un joueur lors de son instanciation
  - Réponse attendue : l'ensemble des caractéristiques du joueur est initialisée comme attendue (état de la main, de la pioche, cartes 1 et 60 posées respectivement sur les piles ascendantes et descendantes, statuts « jeu hostile » et « gagnant » désactivés).
  - Réponse obtenue : le joueur est correctement initialisé sur l'ensemble des propriétés précitées.
- -> **conditionVictoire** teste les conditions de victoire d'un joueur
  - Réponse attendue : le joueur gagne
  - Réponse obtenue : le joueur gagne

Bilan du projet

Programmer en Java, c'est tout autre chose. De nouvelles notions, de nouveaux challenges, une nouvelle syntaxe et une nouvelle conception de la programmation. Il ne fallait plus seulement concevoir un programme qui marche mais aussi un programme qui respecte les principes d'encapsulation propres à la POO.
Cette nouvelle approche nous a demandé beaucoup de réflexions en cours de projet (nous avons à plusieurs reprises modifié par exemple la portée des classes et de leurs méthodes, pour correspondre avec nos objectifs en terme d'encapsulation).

Concernant le rythme de travail, nous avons très rapidement posé les bases du projet. Dès la deuxième semaine, nos classes étaient posées et dès la mi-février, le jeu était fonctionnel. Les semaines suivantes ont donc été l'occasion de corriger les problèmes de syntaxe, de réfléchir à la structuration du code ou encore à la clarté du code et des algorithmes.
Les deux dernières semaines ont été plus intenses avec une réelle attention portée à la réalisation des tests unitaires, à la répartition en paquetages, classes et méthodes et à la vérification de la syntaxe et de la logique du jeu.

D'un point de vue organisation, l'usage de Github nous a permis de rendre compte en temps réel de l'évolution de notre programme. Cela a été un outil clé pour concilier le projet de BPO avec les autres projets et matières. Autre outil primordial, IntelliJ : l'IDE de JetBrains nous a permis de coder avec facilité : les liens entre fonctions sont clairs, la navigation entre classe/méthode est facilitée, les usages de chaque attribut/méthode est consigné et facile d'accès et surtout les outils de refactorisation de l'IDE nous ont permis de changer rapidement la structure du programme lorsque cela nous semblait nécessaire.

Ce projet était l'occasion pour nous de travailler en synergie pour réussir un objectif commun. Les deux membres d'un binôme doivent coopérer en permanence et surtout débattre car c'est le propre du travail en groupe. Nous avons pu donc apprendre ensemble sur notre manière de conduire ce projet, au fil de nos accords comme de nos désaccords. L'apport des TD/TP et du cours de BPO (et celui de BCO d'un côté) a aussi été l'un des points importants du projet. Nous avons pu y découvrir pléthore de techniques pour pouvoir avancer dans le projet. Ce fut une très bonne première expérience en Java et en POO et nous sommes très content qu'elle ait donné vie un jeu fonctionnel. C'est le principal.

Code source des classes du projet

Application

```java
/**
 * BPO Projet période C : Projet The Game - Le Duel
 * Auteurs : Slim BEN DAALI et Anthony ZAKANI
 *  dans le cadre d'un DUT Informatique au sein de l'Université de Paris
 * Dernière mise à jour : vendredi 12 mars 2021 à 23h40
 */

package appli;

import Jeu.Joueur;
import Jeu.Partie;

/**
 * La classe Application contient le main ainsi que les messages
 * de victoire/défaite.
 * Elle appelle le lancement d'une partie.
 *
 * @author Slim BEN DAALI et Anthony ZAKANI
 */
public class Application {
    public static void main(String[] args) {
        Joueur j = new Joueur("NORD");
        Joueur j2 = new Joueur("SUD");

        Partie.lancerPartie(j,j2);

        if(j.isGagnant()){
            System.out.println("partie finie, " + j.getNom() + " a gagné");
        }else
            System.out.println("partie finie, " + j2.getNom() + " a gagné");
    }
}
```

Joueur

```java
package Jeu;

import Jeu.Partie.etatCompteur;
import java.util.LinkedList;

/**
 * La classe Joueur représente le joueur et toutes ses caractéristiques.
 * Ses cartes, sa façon de jouer et son état de victoire y sont représentés.
 *
 * @author Slim BEN DAALI et Anthony ZAKANI
 */
public class Joueur {
    /** Représente le nom attribué au joueur */
    private String nomJoueur;
    /** Représente la pile de cartes ascendentes du joueur */
    private PileJ pileAsc;
    /** Représente la pile de cartes descendantes du joueur */
    private PileJ pileDsc;
    /** Représente la main du joueur */
    private MainJ mainJoueur;
    /** Représente la pioche du joueur */
    private PiocheJ piocheJoueur;
    /** Représente l'état de jeu hostile du joueur
     * (si il a joué sur la pile du joueur adverse au tour
     * précédent) */
    private boolean jeuHostile;
    /** Représente l'état de victoire du joueur */
    private boolean gagnant;

    /**
     * @brief Initialise le joueur ainsi que tous ses attributs avec son nom
     * @param nom Le nom du joueur
     */
    public Joueur(String nom){
        this.nomJoueur = nom;
        this.piocheJoueur = new PiocheJ();
        this.mainJoueur = new MainJ(this);
        initialiserPiles();
        this.gagnant = false;
        this.jeuHostile = false;
    }

    /**
     * @brief Retourne le nom du joueur
     * @return le nom du joueur
     */
    public String getNom() {
        return nomJoueur;
    }

    /**
     * @brief Retourne l'etat de l'attribut gagnant
     * @return l'etat de l'attribut gagnant
     */
    public boolean isGagnant(){
        return this.gagnant;
    }

    /**
     * @brief initialise les piles ascendante et descendante
```

```
     */
    private void initialiserPiles(){
        this.pileDsc = new PileJ();
        this.pileAsc = new PileJ();

        this.pileDsc.pushPile(60);
        this.pileAsc.pushPile(1);
    }

    /**
     * @brief Retourne la manière d'afficher les piles
     * @return La manière d'afficher les piles
     */
    private String affichePiles(){
        String a = "^[";
        if (this.pileAsc.showCarte() < 10) {
            a += "0";
        }
        a += String.valueOf(this.pileAsc.showCarte());
        a += "] v[";
        if (this.pileDsc.showCarte() < 10) {
            a += "0";
        }
        a += String.valueOf(this.pileDsc.showCarte());
        a += "]";
        return a;
    }

    /**
     * @brief Retourne la pioche
     * @return la pioche
     */
    PiocheJ getPioche(){
        return this.piocheJoueur;
    }

    /**
     * @brief Retourne la pile ascendante du joueur
     * @return la pile ascendante
     */
    PileJ getPileAsc(){
        return this.pileAsc;
    }

    /**
     * @brief Retourne la pile descendante du joueur
     * @return la pile descendante
     */
    PileJ getPileDsc(){
        return this.pileDsc;
    }

    /**
     * @brief Retourne la main du joueur
     * @return la main
     */
    MainJ getMainJoueur(){ return this.mainJoueur; }

    /**
     * @brief Retourne la manière d'afficher le joueur
     * @return la manière d'afficher le joueur
     */
    public String toString() {
        return nomJoueur + " " +
```

```
                affichePiles() +
                " (m" + mainJoueur.getTailleMain()
                + "p" + piocheJoueur.getNbCartes() + ")";
    }

    /**
     * @brief Modifie la valeur de l'attribut gagnant a vrai
     */
    void setGagnant(){
        this.gagnant = true;
    }

    /**
     * @brief Modifie l'etat de l'attribut jeuHostile à Etat
     * @param Etat l'etat a mettre à l'attribut jeuHostile
     */
    void setJeuHostile(boolean Etat){
        this.jeuHostile = Etat;
    }

    /**
     * @brief Retourne l'etat de l'attribut jeuHostile
     * @return l'état de l'attribut jeuHostile
     */
    boolean getJeuHostile(){
        return this.jeuHostile;
    }

    /**
     * @brief Vérifie si les cartes jouées sont dans la main
     * @param intTab les cartes jouées
     * @return vrai si les cartes sont existantes, sinon faux
     */
    boolean verifCarte(int[] intTab){
        int cartesExistantes = 0;
        for(int cartes : intTab){
            for(int i = 0; i < this.mainJoueur.getTailleMain(); ++i){
                if (cartes == this.mainJoueur.showCarteMain(i)){
                    cartesExistantes++;
                    break;
                }
            }
        }

        return cartesExistantes == intTab.length;
    }

    /**
     * @brief Vérifie si le joueur peut joueur sur ces piles ou les piles de
son adversaire lors de son tour
     * @param J2 L'adversaire
     * @return vrai si le joueur peut jouer sinon faux
     */
    boolean possibiliteJouer(Joueur J2) {
        int cartePossible = 0;
        for (int i = 0; i < this.mainJoueur.getTailleMain(); ++i) {
            if (this.mainJoueur.showCarteMain(i) < this.pileDsc.showCarte()
|| this.mainJoueur.showCarteMain(i) == this.pileDsc.showCarte() + 10) {
                ++cartePossible;
            }
            if (this.mainJoueur.showCarteMain(i) > this.pileAsc.showCarte()
|| this.mainJoueur.showCarteMain(i) == this.pileAsc.showCarte() - 10) {
                ++cartePossible;
            }
```

```
            }
            for (int i = 0; i < this.mainJoueur.getTailleMain(); ++i) {
                if (this.mainJoueur.showCarteMain(i) > J2.pileDsc.showCarte()) {
                    ++cartePossible;
                    break;
                }
                if (this.mainJoueur.showCarteMain(i) < J2.pileAsc.showCarte()) {
                    ++cartePossible;
                    break;
                }
            }
            return cartePossible >= 2;
        }

        /**
         * @brief Vérifie si le joueur a gagné la partie
         * @return vrai si le joueur a gagné sinon faux
         */
        boolean partieGagnee(){
            if(this.mainJoueur.isEmpty() && this.piocheJoueur.isEmpty()){
                this.setGagnant();
                return true;
            }
            return false;
        }

        /**
         * @brief Reprend les cartes posées sur ces piles ou les piles de
    l'adversaire lors du dernier tour
         * @param J2 L'adversaire
         * @param compteur Les lieux ou ont été posées les cartes
         */
        void reprendreCarte(Joueur J2, LinkedList<etatCompteur> compteur){
            for(int i = 0; i < compteur.size(); ++i){
                switch (compteur.get(i)){
                    case ASC:

    this.getMainJoueur().setCarteMain(this.getPileAsc().getCarte());
                        break;
                    case DSC:

    this.getMainJoueur().setCarteMain(this.getPileDsc().getCarte());
                        break;
                    case ASCAD:

    this.getMainJoueur().setCarteMain(J2.getPileAsc().getCarte());
                        break;
                    case DSCAD:

    this.getMainJoueur().setCarteMain(J2.getPileDsc().getCarte());
                        break;
                }
            }
        }

        /**
         * @brief Cherche la carte dans la main
         * @param carteAChercher la carte
         * @return l'index à laquel est la carte
         */
        boolean chercherCarte(int carteAChercher){
            for(int i = 0; i < this.getMainJoueur().getTailleMain(); ++i){
                if(carteAChercher == this.getMainJoueur().showCarteMain(i)){
                    this.getMainJoueur().jouerCarteMain(i);
```

```
                    return true;
                }
            }
            return false;
        }

        /**
         * @brief Remplit la main à partir de la pioche
         * @return le nombre de carte piochée
         */
        int remplirMain(){
            int nbCartesAPiocher;
            boolean aJoueAilleurs = this.getJeuHostile();
            if (aJoueAilleurs){
                nbCartesAPiocher = 6 - this.getMainJoueur().getTailleMain();
            }
            else
                nbCartesAPiocher = 2;
            if( this.getPioche().getNbCartes() >= nbCartesAPiocher) {
                for (int i = 0; i < nbCartesAPiocher; ++i) {

this.getMainJoueur().setCarteMain(this.getPioche().getCartePioche());
                }
            }
            else if (!(this.getPioche().isEmpty())){
                nbCartesAPiocher = this.getPioche().getNbCartes();
                while (!(this.getPioche().isEmpty())) {

this.getMainJoueur().setCarteMain(this.getPioche().getCartePioche());
                }
            }
            else{
                return 0;
            }
            return nbCartesAPiocher;
        }
}
```

Partie

```
package Jeu;

import java.util.LinkedList;
import java.util.Scanner;

/**
 * La classe Partie contient des méthodes permettant le déroulement d'une
partie
 * de "The Game - Le Duel".
 * Elle contient ce qui est nécessaire au bon déroulement d'une partie du jeu
de
 * société.
 *
 * @author Slim BEN DAALI et Anthony ZAKANI
 */
public class Partie {
    enum etatCompteur{ASC, DSC, ASCAD, DSCAD}

    /**
     * @brief Joue la carte sur ca pile ou la pile de l'adversaire
     * @param carteJouee La carte jouée
```

```
     * @param Carte La carte jouée plus la pile sur laquel jouée
     * @param J Le joueur qui joue
     * @param J2 l'adversaire
     * @param compteur la pile sur laquelle la carte a été jouée
     * @return vrai si la carte a pu être jouée
     */
    protected static boolean jouer(int carteJouee, String Carte, Joueur J,
Joueur J2, LinkedList<etatCompteur> compteur){
        if(Carte.length() == 4){
            if(Carte.charAt(3) == '\''){
                switch (Carte.charAt(2)){
                    case 'v' :
                        if(carteJouee > J2.getPileDsc().showCarte()){
                            if (!J.chercherCarte(carteJouee))
                                return false;
                            J2.getPileDsc().pushPile(carteJouee);
                            compteur.add(etatCompteur.DSCAD);
                            J.setJeuHostile(true);
                            return true;
                        }
                        break;
                    case '^' :
                        if(carteJouee < J2.getPileAsc().showCarte()){
                            if(!(J.chercherCarte(carteJouee)))
                                return false;
                            J2.getPileAsc().pushPile(carteJouee);
                            compteur.add(etatCompteur.ASCAD);
                            J.setJeuHostile(true);
                            return true;
                        }
                        break;
                    default : return false;
                }
            }
            else
                return false;
        }
        else{
            try {
                switch (Carte.charAt(2)) {
                    case 'v':
                        if (carteJouee < J.getPileDsc().showCarte() ||
carteJouee == (J.getPileDsc().showCarte() + 10)) {
                            if (!(J.chercherCarte(carteJouee)))
                                return false;
                            J.getPileDsc().pushPile(carteJouee);
                            compteur.add(etatCompteur.DSC);
                            return true;
                        }
                        break;
                    case '^':
                        if (carteJouee > J.getPileAsc().showCarte() ||
carteJouee == (J.getPileAsc().showCarte() - 10)) {
                            if (!(J.chercherCarte(carteJouee)))
                                return false;
                            J.getPileAsc().pushPile(carteJouee);
                            compteur.add(etatCompteur.ASC);
                            return true;
                        }
                        break;
                    default:
                        return false;
                }
            }catch(StringIndexOutOfBoundsException e){
```

```
                return false;
            }
        }
        return false;
    }

    /**
     * @brief Effectue les opérations nécessaires au déroulement d'un tour
     * @param J désigne le joueur qui joue le tour
     * @param J2 désigne le joueur contre qui le tour est joué
     * @return vrai si le tour suivant doit se produire
     *         faux si le tour actuel est censé être le dernier
     */
    static boolean tour(Joueur J, Joueur J2){
        J.getMainJoueur().rangerMain();
        J2.getMainJoueur().rangerMain();
        // séparer pour rendre flexible ?
        if(J.getNom().equals("NORD")) {
            System.out.println(J.toString());
            System.out.println(J2.toString());
        }
        else{
            System.out.println(J2.toString());
            System.out.println(J.toString());
        }

        System.out.println(J.getMainJoueur().toString(J));
        int nbEssais = 0;
        int nbCartePoser = 0;
        int nbCartesPiochees = 0;
        boolean coupValide = false;

        if(!J.possibiliteJouer(J2)){
            J2.setGagnant();
            return false; // la partie est finie -> le jeu s'arrêtera avant
le prochain tour
        }

        do {
            if (nbEssais == 0){
                System.out.print("> ");
            }
            else
                System.out.print("#> ");

            nbEssais++;

            Scanner sc = new Scanner(System.in);
            String s = sc.nextLine();
            String[] tab = s.split("\\s+");

            if (tab.length < 2){
                continue;
            }

            int[] intTab = new int[tab.length];
            try {
                for (int i = 0; i < tab.length; ++i) {
                    // try sinon catch sur la conversion en int (qui pourra
sortir de la boucle et forcer l'utilisateur
                    // a recommencer sa saisie)
                    // try this
                    intTab[i] =
Integer.parseInt(String.valueOf(tab[i].charAt(0)));
```

```
                    intTab[i] *= 10;
                    intTab[i] +=
Integer.parseInt(String.valueOf(tab[i].charAt(1)));
                    // or if failed, go outside and retry
                }
            }catch(NumberFormatException | StringIndexOutOfBoundsException
a){
                continue;
            }

            if (!J.verifCarte(intTab)){
                continue;
            }
            LinkedList<etatCompteur> Compteur = new LinkedList<>();
            nbCartePoser = 0;
            for (int i = 0; i < intTab.length; ++i) {
                if (jouer(intTab[i], tab[i], J, J2, Compteur)) {
                    ++nbCartePoser;
                }
            }

            int nbCarteAd = 0;

            for(int i = 0; i < Compteur.size(); ++i){
                if(Compteur.get(i).equals(etatCompteur.ASCAD) ||
Compteur.get(i).equals(etatCompteur.DSCAD)){
                    ++nbCarteAd;
                }
            }

            if(nbCartePoser == intTab.length && nbCarteAd <= 1){
                coupValide = true;
            }
            else {
                J.reprendreCarte(J2, Compteur);
            }

            // pouvoir revenir en arrière sur les cartes en cas de jeu faux
pour cause de cartes mal posées
        }while (!coupValide);

        if (!J.partieGagnee()) {
            if (!J.getPioche().isEmpty())
                nbCartesPiochees = J.remplirMain();
            J.setJeuHostile(false);
            System.out.println(nbCartePoser + " cartes posées, " +
nbCartesPiochees + " cartes piochées");

            return true; // la partie n'est pas finie -> le jeu continue

        } else {
            // voir plus haut, séparer pour rendre flexible !
            if (J2.getNom().equals("NORD")) {
                System.out.println(J2.toString());
                System.out.println(J.toString());
            } else {
                System.out.println(J.toString());
                System.out.println(J2.toString());
            }

            System.out.println(J2.getMainJoueur().toString(J2));

            return false; // la partie est finie -> le jeu s'arrêtera avant
le prochain tour
```

```
        }
    }

    /**
     * @brief Permet l'alternance des tours entre les deux joueurs
     * @param joueur1 le premier joueur à jouer
     * @param joueur2 le second joueur à jouer
     */
    public static void lancerPartie(Joueur joueur1, Joueur joueur2){
        for(;;) {
            //Nord joue
            if (joueur1.getMainJoueur().getTailleMain() >= 2) { // si sa main
est en possibilité de jouer
                if (!tour(joueur1, joueur2)) { // si l'état du tour donne
faux (partie perdue) -> le jeu s'interrompt
                    break;
                }
            } else {
                break;
            }

            //Sud joue
            if (joueur2.getMainJoueur().getTailleMain() >= 2) { // si sa main
est en possibilité de jouer
                if (!tour(joueur2, joueur1)) { // si l'état du tour donne
faux (partie perdue) -> le jeu s'interrompt
                    break;
                }
            } else {
                break;
            }
        } // Tant que l'on est pas sorti par un break
    }
}
```

MainJ

```
package Jeu;

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

/**
 * La classe MainJ représente la main (les cartes que peut jouer un joueur).
 * Y sont représentés sa contenance et les manières de la manipuler.
 *
 * @author Slim BEN DAALI et Anthony ZAKANI
 */
class MainJ {
    /** Représente une liste chaînée d'entiers correspondants
     * aux cartes de la main */
    private LinkedList<Integer> main;

    /**
     * @brief Initialise la main en la remplissant de six cartes de la pioche
     * @param J Le joueur sur lequel on
     */
    MainJ(Joueur J){
        this.main = new LinkedList<>();
        for(int i = 0; i < 6; ++i) {
            this.main.add(J.getPioche().getCartePioche());
        }
```

```
}

/**
 * @brief Range la main dans l'ordre croissant
 */
void rangerMain(){
    Collections.sort(this.main);
}

/**
 * @brief Retourne la manière d'afficher la main du joueur
 * @param J le joueur
 * @return la manière d'afficher la main
 */
public String toString(Joueur J){
    StringBuilder a = new StringBuilder("cartes " + J.getNom() + " { ");
    for(int i = 0; i < this.main.size(); ++i){
        if (this.main.get(i) < 10) {
            a.append("0");
        }
        a.append(this.main.get(i));
        a.append(" ");
    }
    a.append("}");
    return a.toString();
}

/**
 * @brief Supprime la carte de la main
 * @param idx L'index de la carte dans la main
 */
void jouerCarteMain(int idx){
    this.main.remove(idx);
}

/**
 * @brief Ajoute la carte dans la main
 * @param carte la carte
 */
void setCarteMain(int carte){
    if(this.main.size() < 6) {
        this.main.add(carte);
    }
}

/**
 * @brief Retourne le numéro de la carte à l'index dans la main
 * @param idx L'index
 * @return La carte
 */
int showCarteMain(int idx){
    return this.main.get(idx);
}

/**
 * @brief Retourne la taille de la main
 * @return la taille de la main
 */
int getTailleMain(){
    return this.main.size();
}

/**
 * @brief Retourne si la carte est vide
```

```
     * @return vrai si la carte est vide sinon faux
     */
    boolean isEmpty(){
        return this.main.isEmpty();
    }
}
```

PileJ

```
package Jeu;

import java.util.LinkedList;

/**
 * La classe PileJ représente la pile de cartes sur laquelle un joueur peut
jouer.
 * Y sont représentés sa contenance et les manières de la manipuler.
 *
 * @author Slim BEN DAALI et Anthony ZAKANI
 */
class PileJ {
    /** Représente la liste chaînée d'entiers correspondant aux
     * cartes dans la pile du joueur */
    private LinkedList<Integer> pile;

    /**
     * @brief Initialise la pile
     */
    PileJ(){
        this.pile = new LinkedList<>();
    }

    /**
     * @brief Ajoute la carte à la pile
     * @param carte la carte
     */
    void pushPile(int carte){ this.pile.push(carte); }

    /**
     * @brief Retourne le numéro de la carte
     * @return le numéro de la carte
     */
    int showCarte(){ return this.pile.peek(); }

    /**
     * brief Retourne le numéro de la carte et la supprime
     * @return le numéro de la carte
     */
    int getCarte(){
        int carte = this.pile.peek();
        this.pile.pop();
        return carte;
    }
}
```

PiocheJ

```java
package Jeu;

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

/**
 * La classe PiocheJ représente la pioche qu'un joueur peut utiliser pour piocher ses cartes.
 * Y sont représentés sa contenance et les manières de la manip
 */
class PiocheJ {
    /** Représente la liste chaînée d'entiers correspondant
     * aux cartes dans la pioche du joueur */
    private LinkedList<Integer> pilePioche;

    /**
     * @brief Initialise la pioche
     */
    PiocheJ(){
        this.pilePioche = new LinkedList<>();
        initialiserPioche();
    }

    /**
     * Remplie la pioche de 58 cartes
     */
    private void initialiserPioche(){
        for (int i = 2; i <= 59; ++i){
            this.pilePioche.push(i);
        }
        Collections.shuffle(pilePioche);
    }

    /**
     * @brief Retourne le nombre de carte dans la pioche
     * @return le nombre de carte dans la pioche
     */
    int getNbCartes(){
        return this.pilePioche.size();
    }

    /**
     * @brief Retourne si la pioche est vide
     * @return vrai si la pioche est vide sinon faux
     */
    boolean isEmpty(){
        return this.pilePioche.size() == 0;
    }

    /**
     * @brief Retourne le numéro de la carte en tête et la supprime
     * @return le numéro de la carte en tête
     */
    int getCartePioche() {
        int carteTiree = this.pilePioche.peek();
        this.pilePioche.pop();
        return carteTiree;
    }
}
```

Tests

```java
package Jeu;

import org.junit.Test;
import org.junit.Assert;
import java.util.LinkedList;

public class tests {

    @Test
    public void piocherVide(){
        // Vérifier que piocher si la pioche est vide est impossible.
        Joueur J = new Joueur("Cobaye");
        int nbCartes = J.getPioche().getNbCartes();
        // vider la pioche
        for (int i = 0 ; i < nbCartes ; ++i){
            J.getPioche().getCartePioche();
        }
        Assert.assertEquals(0, J.remplirMain());
    }

    @Test
    public void jouerHostile(){
        // Vérifier que le statut de jeu hostile s'applique correctement et
produit une piochage approprié
        Joueur J = new Joueur("Cobaye");
        J.setJeuHostile(true);
        for(int i = 0; i < 3; ++i) {

J.getMainJoueur().jouerCarteMain(J.getMainJoueur().getTailleMain() - 1);
        }
        Assert.assertEquals(3, J.remplirMain());
    }

    @Test
    public void jouerNonHostile(){
        // Vérifier que le statut de jeu non hostile s'applique correctement
et produit une piochage approrié
        Joueur J = new Joueur("Cobaye");
        J.setJeuHostile(false);
        for(int i = 0; i < 3; ++i) {

J.getMainJoueur().jouerCarteMain(J.getMainJoueur().getTailleMain() - 1);
        }

        Assert.assertEquals(2, J.remplirMain());
    }

    @Test
    public void joueurInitialiser(){
        //Vérifie que le joueur est bien initialiser
        Joueur J = new Joueur("Cobaye");
        Assert.assertEquals(6, J.getMainJoueur().getTailleMain());
        Assert.assertEquals(52, J.getPioche().getNbCartes());
        Assert.assertEquals(60, J.getPileDsc().showCarte());
        Assert.assertEquals(1, J.getPileAsc().showCarte());
        Assert.assertEquals(false, J.getJeuHostile());
        Assert.assertEquals(false, J.isGagnant());
    }
```

```
    @Test
    public void affichageSyntaxe(){
        // Vérifie si l'état de la main change entre deux "tours"
        Joueur J1 = new Joueur("NORD");
        Assert.assertEquals("NORD ^[01] v[60] (m6p52)", J1.toString());
        String avantTest_PILES = J1.toString();
        String avantTest_MAIN = J1.getMainJoueur().toString(J1);
        for(int i = 0; i < 3; ++i){

J1.getMainJoueur().jouerCarteMain(J1.getMainJoueur().getTailleMain() - 1);
        }
        String apresTest_MAIN = J1.getMainJoueur().toString(J1);
        String apresTest_PILES = J1.toString();

        Assert.assertNotEquals(avantTest_MAIN,apresTest_MAIN);
        Assert.assertNotEquals(avantTest_PILES,apresTest_PILES);
    }

    @Test
    public void isEmptyMain(){
        //Vérifie que la fonction isEmpty fonctionne bien
        Joueur J = new Joueur("Cobaye");
        int nbCarte = J.getMainJoueur().getTailleMain();
        for(int i = 0; i < nbCarte; ++i){

J.getMainJoueur().jouerCarteMain(J.getMainJoueur().getTailleMain() - 1);
        }
        Assert.assertTrue(J.getMainJoueur().isEmpty());
    }

    @Test
    public void joueurCorrect(){
        //Verifie que le joueur jouer
        Joueur J = new Joueur("Cobaye");
        Joueur J2 = new Joueur("Cobaye");
        J.getPileAsc().pushPile(21);
        J.getMainJoueur().jouerCarteMain(5);
        J.getMainJoueur().setCarteMain(11);
        LinkedList<Partie.etatCompteur> compteur = new LinkedList<>();
        Assert.assertTrue(Partie.jouer(11, "11^", J, J2, compteur));
    }

    @Test
    public void possibilitéPoserCarte(){
        //Verifie que la méthode possibilitéJouerfonctionne bien
        Joueur J = new Joueur("Cobaye");
        Joueur J2 = new Joueur("Cobaye");
        Assert.assertTrue(J.possibiliteJouer(J2));
    }

    @Test
    public void conditionVictoire(){
        //Verifie que les conditions de la victoire sont bien implémentées
        Joueur J = new Joueur("Cobaye");
        Joueur J2 = new Joueur("Cobaye2");
        int nbCartes = J.getPioche().getNbCartes();
        for (int i = 0 ; i < nbCartes ; ++i){
            J.getPioche().getCartePioche();
        }
        int nbCartesMain = J.getMainJoueur().getTailleMain();
        for (int i = 0 ; i < nbCartesMain ; ++i){

J.getMainJoueur().jouerCarteMain(J.getMainJoueur().getTailleMain()-1);
        }
```

```
            Assert.assertTrue(J.partieGagnee());
        }
    }
```