THE GAME

LE DUEL

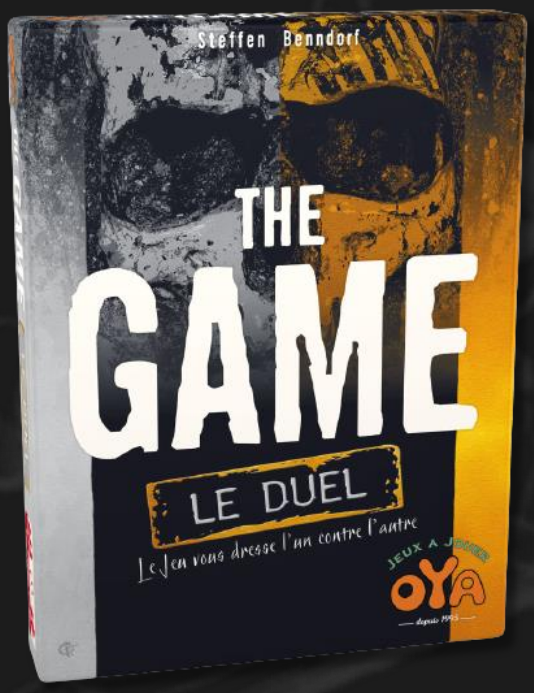
Yannick Li
108

Stefan Radovanovic
107

Rapport De Projet
Base de la programmation orientée objet

Université de Paris

Table des matières



« Le premier jeu d'affrontement où vous devez aider votre adversaire pour gagner. »

Introduction

Présentation

C'est *Steffen Benndorf* créateur de la saga *The Game*, une série de jeu de société basé sur la collaboration qui a eu l'idée du *Duel*, un jeu d'affrontement avec deux joueurs, et 60 cartes chacun. Un affrontement où l'objectif est de se débarrasser de l'entièreté de ses cartes à l'aide de pile ascendante et descendante que chaque joueur possède.

Ce rapport passera en détail chaque étape de conception. Ici, l'objectif du projet étant de programmer ce jeu de carte afin de le numériser et de rendre possible son utilisation sur ordinateur, le langage choisit sera le *java,* un langage de programmation orienté objet.

Pour se faire, nous avons divisé le projet en deux paquetages différents.

- Application
- Composantes

Le paquet *Appli* contient la classe *Application,* contenant le main.

Le paquet *Composantes* contient les classes qui composent le programme à l'image du jeu physique. Ainsi, les classes *PaquetDeCarte*, *Joueur*, *Input* et *Règles* définissent les objets que nous utilisons pour le bon fonctionnement du programme.

Vous trouverez dans ce rapport dans un premier temps le diagramme UML des classes formant notre projet. Puis ensuite, le bilan du projet avec nos retours personnels sur la conception. Le projet se fini par une annexe sur le code java de ces classes ainsi que leurs tests unitaires (chaque classe est suivi de ses tests dédiés) .

Introduction

Aspect Technique

Ce projet a été réalisé avec l'aide de l'*IDE Intellij IDEA*, une plateforme destiné à la programmation java développé par le groupe *JetBrain*.

En parallèle d'*Intellij IDEA*, nous avons utilisé l'extension *Code With Me*. Installé directement sur l'IDE, celui-ci permet de faire du « développement collaboratif », c'est-à-dire qu'il nous a été rendu possible de coder en synchrone nos lignes de code.

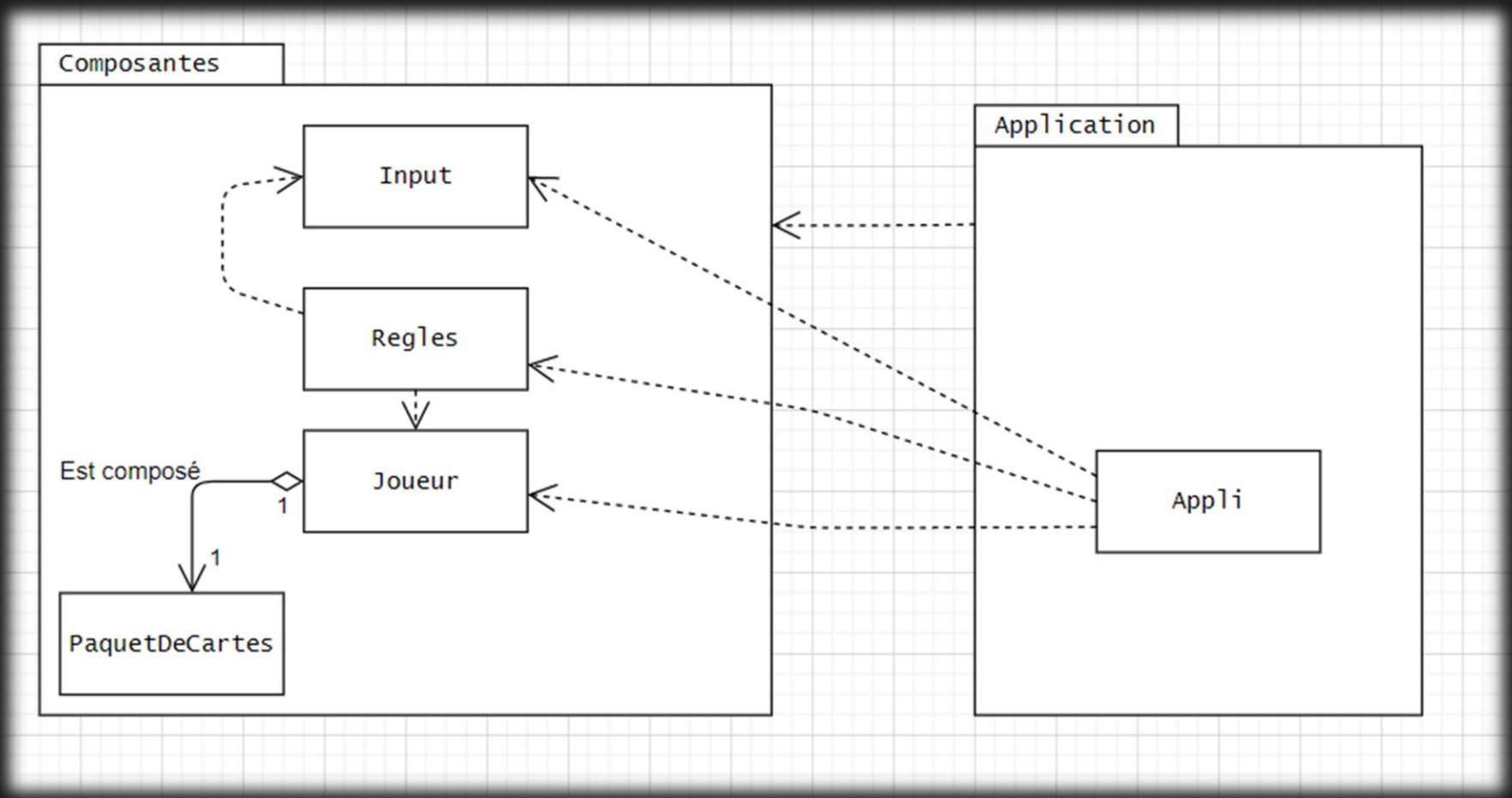
Plus d'info : https://www.jetbrains.com/code-with-me/

Enfin, pour le partage et la sauvegarde de notre progression nous avons utiliser un « Service web d'hérbergement et de gestion de développement de logiciels » plus précisément la plateforme *GitHub*.

Le repository : https://github.com/Yan2708/THEGAME.git

Nous reviendrons sur l'aspect technique dans le bilan avec les détails de la conceptions du code.

Diagramme UML



Bilan

Bien que le *Java* soit notre premier langage de programmation orientée objet, nous n'avons pas rencontré énormément d'obstacles durant la programmation. En effet, combiner à l'expérience du *C++* et du *C*, nous étions totalement capables de nous organiser et nous retrouver dans un projet qui nous laisse en totale autonomie (sans Test Unitaires fourni comme en *SDA* ou en *IAP*).

Durant le projet, nous avons pu souligner de quelques inconvénients, ainsi que des avantages auxquels nous avons su tirer le meilleur. Par exemple, au début du projet nous ne savions pas utiliser *Github* alors que maintenant son utilisation constitue un très gros avantage dans notre bagage de compétences, et des exemples comme celui-ci on peut en citer de nombreux autres comme le faite qu'on ait appris et étendu nos dictionnaires de méthodes *standards* (depuis le début de l'année, nous n'utilisions pas de librairie qui nous simplifie le travail, cf. les méthodes comme *qSort* pour le projet de SDA). Concernant les inconvénients du projet nous sommes intimement persuadé que le projet manque cruellement de potentiel au niveau de la manipulation d'objet, car dans notre cas nous manipulons seulement les objets *Joueur* et *PaquetDeCarte*, peut être qu'un projet avec plus de potentielle Objet nous aurait été bénéfique, de plus nous trouvions dommage que les *Td* et *Tp* ne se sont pas attardé sur la gestion et la création d'erreur (*throws, try catch*, etc ...) de notre côté, c'est un aspect de la programmation Java que nous n'avons pas touché sur le projet.
Bien sure quand bien même nous n'avons pas pu tout explorer nous avons appris énormément et si au début du bilan nous disons que nous n'avons pas rencontré énormément d'obstacle il serait faux de dire que le projet était simple, on retient particulièrement l'algorithme qui permet de détecter si un joueur est bloqué et qu'il ne peut plus jouer pour finir une partie, cette méthode dans sa construction s'apparente a une IA (intelligence artificielle) qui joue tout les coups possibles d'un joueur (même s'il nous en faut que deux), nous avons utiliser quelque ruse pour cette méthode mais nous n'en restons pas complètement satisfait. Par exemple, ce type de recherche récursive s'appelle « *arbre binaire* » de recherche et nous sommes persuadés que notre algo est optimisable de ce point de vue.
Pour finir sur une note positive, ce projet était vraiment sympa à coder et assure une bonne introduction au langage *java*.

Annexes

* Tout les tests unitaires sont valides.

Code Source

PaquetDeCartes.java

```java
package Composantes;

import java.util.ArrayList;
import java.util.Random;

/**
 * Le paquet de cartes est une composante essentielle au jeu.
 * il signifie à la fois :
 * - l'ensemble complet des cartes nécessaires pour pratiquer le jeu
 * - mais aussi dans notre cas le jeu de société en lui-même, car c'est le matériel exclusif du jeu.
 * Il s'agit ici de faire une représentation d'un réel paquet de cartes.
 * la méthode utilisée est de faire en sorte de rendre piochable n'importe quelle carte
 * d'une ArrayList aléatoire, à la manière d'un paquet de carte que l'on aurait mélanger.
 *
 * @author      Yannick li
 * @author      Stefan Radovanovic
 * @version     1, 2/27/2021
 * */
public class PaquetDeCartes {

    //Taille du paquet de carte pour un joueur
    private static final int TAILLE_PAQUET_CARTE = 60;

    //Tableau de int représentant le paquet de cartes
    private final ArrayList<Integer> paquetDeCartes = new ArrayList<>(TAILLE_PAQUET_CARTE);


    /**
     * constructeur du paquet de cartes
     */
    public PaquetDeCartes() {
        // 2 methodes;
        // methode realiste : initialiser un paquet avec des valeurs randoms entre 1-60 et unique
        // methode simple : pioche une carte au hasard d'un paquet trié (on prend ca)
        for (int i = 0; i < TAILLE_PAQUET_CARTE; i++) {
            paquetDeCartes.add(i, i + 1);
        }
    }
```

Code Source

PaquetDeCartes.java

```java
/**
 * Pioche une carte aleatoirement dans le paquet et la retire du paquet.
 *
 * @return          le numéro de la carte piochée
 * @see             Random#nextInt()
 * @see             ArrayList#remove(int)
 */
public int piocher() {
   assert(!estVide());
   Random rand = new Random();
   int idxAleatoire = rand.nextInt(paquetDeCartes.size());
   int carteAleatoire = paquetDeCartes.get(idxAleatoire);
   paquetDeCartes.remove(idxAleatoire);
   return carteAleatoire;
}


/** Pioche une carte spécifique dans le paquet de carte.
 *
 * @param carte       la carte choisie
 * @return            la carte piochée
 * @see               Integer#valueOf(int)
 * @see               IllegalArgumentException
 **/
public int piocher(int carte) throws IllegalArgumentException{
   if(estVide() && paquetDeCartes.contains(carte))
      throw new IllegalArgumentException("la carte n'est pas dans le paquet de carte !");

   paquetDeCartes.remove(Integer.valueOf(carte));
   return carte;
}


/**
 * Indique si le paquet est vide.
 *
 * @return          la pioche est vide ou non
 */
public boolean estVide()
{
   return paquetDeCartes.size() == 0;
}
```

Code Source

PaquetDeCartes.java

```
    /**
     * Renvoie le dernier indice du paquet de carte.
     *
     * @return                          le dernier indice de l'ArrayList
     */
    public int getLastIdx() {
        return paquetDeCartes.size()-1;
    }


    /**
     * Renvoie le nombre de cartes dans le paquet de cartes.
     *
     * @return                          le nom de cartes
     */
    public int getNbCartes() { return paquetDeCartes.size(); }

}
```

Tests Unitaires

PaquetDeCartesTest.java

```java
package Test;

import Composantes.PaquetDeCartes;
import org.junit.jupiter.api.Test;

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

class PaquetDeCartesTest {

    /**
     * Test la pioche aléatoire dans le paquet de carte.
     * */
    @Test
    public void testPiocheAleatoire()
    {
        PaquetDeCartes p1 = new PaquetDeCartes();
        while(!p1.estVide())
        {
            int carte = p1.piocher();
            assert(carte >=1 && carte < 61);
        }
        assert(p1.estVide());
    }

    /**
     * Test si le paquet est vide.
     * */
    @Test
    public void testEstVide(){
        PaquetDeCartes p1 = new PaquetDeCartes();
        assertFalse(p1.estVide());
        for (int i = 0 ; i<p1.getNbCartes();) {
            p1.piocher();
        }
        assertTrue(p1.estVide());
    }
```

Tests Unitaires

PaquetDeCartesTest.java

```java
/**
 * Test si le paquet est vide.
 * */
@Test
public void testGetNbCartes(){
    PaquetDeCartes p1 = new PaquetDeCartes();
    assertEquals(p1.getNbCartes(), 60);
    p1.piocher();
    assertEquals(p1.getNbCartes(), 59);
}

/**
 * Test le renvoie du dernier index de la liste de carte.
 * */
@Test
public void testGetLastIdx(){
    PaquetDeCartes p1 = new PaquetDeCartes();
    for(int i = 0; i< 60;i++){
        assertEquals(59 - i, p1.getLastIdx());
        p1.piocher();
    }
}

/**
 * Test la pioche specifique (avec une carte donnée).
 * */
@Test
public void testPiocheCarte()
{
    PaquetDeCartes p1 = new PaquetDeCartes();
    int i=1;
    while(!p1.estVide())
    {
        int carte = p1.piocher(i);
        assertTrue(carte >=1 && carte <= 60);
        i++;
    }
    assertTrue(p1.estVide());
}
}
```

Code Source

Joueur.java

```java
package Composantes;

import java.util.ArrayList;
import java.util.Collections;

/**
 * Le joueur est une représentation de l'utilisateur.
 * Il possède un nom ("NORD" ou "SUD").
 * Un jeu de cartes.
 * Une pile ascendante et descendante (la base).
 * Un paquet de cartes, la pioche.
 * Il s'agit ici de manœuvrer le joueur en fonction de l'utilisateur et de ce qu'il décide.
 *
 * @author      Yannick li
 * @author      Stefan Radovanovic
 * @version     1, 2/27/2021
 */
public class Joueur {

    public static final int NB_CARTES_MAX = 6;  // nombre de cartes maximum dans le jeu


    public final String nom;  // nom du joueur

    public ArrayList<Integer> jeu;  // jeu du joueur

    private int ascendant, descendant;   // base du joueur

    private final PaquetDeCartes pioche;  // pioche du joueur
```

Code Source

Joueur.java

```
/**
 * Constructeur du joueur
 *
 * @param n           le nom du joueur, sous une chaîne de caractères.
 */
public Joueur(String n) {
   nom = n;
   pioche = new PaquetDeCartes();
   // la pile ascendante est initialisée à 1 et la descendante à 60
   ascendant = pioche.piocher(1);
   descendant = pioche.piocher(60);
   jeu = new ArrayList<>();
   for(int i=0; i < NB_CARTES_MAX ; i++) // pioche le jeu du joueur
      piocherCarte();

}


/**
 * Constructeur d'un clone de joueur.
 * Ce constructeur permet de recopier les données d'un joueur
 * afin de les manipuler sans incidence avec le vrai Joueur (l'utilisateur).
 *
 * @param j           le joueur à cloner.
 */
private Joueur(Joueur j) {   // clone utilisé que pour la vérification des coups
   jeu = new ArrayList<>();

   jeu.addAll(j.jeu);    //  duplication en profondeur d'une arrayList
   ascendant = j.ascendant;
   descendant = j.descendant;
   pioche = new PaquetDeCartes();  //   pas besoin de paquet de carte
   nom = "CLONE";     //   pas besoin de nom
}


/**
 * crée un clone du joueur
 *
 * @return           le clone du joueur
 */
public Joueur clone() {
   return new Joueur(this);
}
```

Code Source

Joueur.java

```java
/**
 * getter de la pile ascendante
 *
 * @return la valeur de la pile
 * */
public int getAscendant() {
    return ascendant;
}


/**
 * setter de la pile ascendante
 *
 * @param ascendant         la nouvelle valeur de la pile
 * */
public void setAscendant(int ascendant) {
    this.ascendant = ascendant;
}


/**
 * getter de la pile descendante
 *
 * @return la valeur de la pile
 * */
public int getDescendant() {
    return descendant;
}


/**
 * setter de la pile descendante
 *
 * @param descendant        la nouvelle valeur de la pile
 * */
public void setDescendant(int descendant) {
    this.descendant = descendant;
}
```

Code Source

Joueur.java

```java
/**
 * Renvoie le nombre de cartes dans la pioche
 *
 * @return              le nombre de cartes dans la pioche
 */
public int getNbPioche() {
    return pioche.getNbCartes();
}


/**
 * Verifie si la pioche du joueur est vide
 *
 * @return              si la pioche est vide ou non
 */
public boolean piocheEstVide() {
    return pioche.estVide();
}


/**
 *  Pioche une carte dans la pioche et la rajoute dans le jeu du joueur.
 */
public void piocherCarte(){
    if(!jeuEstPlein() && !piocheEstVide()){
        jeu.add(pioche.piocher());
    }
    Collections.sort(jeu);
}


/**
 * Vérifie si le jeu du joueur est vide
 *
 * @return              si le jeu est vide ou non
 */
public boolean jeuEstVide() {
    return jeu.isEmpty();
}
```

Code Source

Joueur.java

```java
/**
 * Vérifie si le jeu du joueur est plein.
 *
 * @return              si le jeu est plein ou non.
 */
public boolean jeuEstPlein() {
    return jeu.size() == NB_CARTES_MAX;
}


/**
 * Vérifie si la carte jouée du joueur est dans sa main.
 *
 * @param carte         la carte du joueur.
 * @return              la carte est présente ou non.
 */
public boolean estDansLeJeu(int carte) {
    return jeu.contains(carte);
}


/**
 * Pose une carte donnée dans une base donnée.
 *
 * @param carte         la carte donné.
 * @param base          la base dans la quelle il faut jouer.
 * */
public void poserCarte(int carte, char base) {
    if (base == '^')
        ascendant = carte;
    else if (base == 'v')
        descendant = carte;
}
```

Code Source

Joueur.java

```java
/**
 * pose une carte donnée sur sa base.
 *
 * @param carte                 la carte à poser.
 * @param base                  la base où l'on va poser la carte.
 */
public void jouerCarte(int carte, char base) {
    assert(estDansLeJeu(carte));
    this.jeu.remove((Integer) carte);
    this.poserCarte(carte, base);
}


/**
 * pose une carte donnée sur la base du joueur adverse
 *
 * @param carte                 la carte à poser
 * @param base                  la base où l'on va poser la carte
 * @param j                     le joueur à qui on pose une carte dans sa base
 */
public void jouerCarte(int carte, char base, Joueur j) {
    assert(estDansLeJeu(carte));
    this.jeu.remove(((Integer) carte));
    j.poserCarte(carte, base);
}


/**
 * Créer une chaîne de caractères comportant l'ensemble du jeu d'un joueur.
 *
 * @return                      la chaîne de caractères
 */
public String afficherJeu(){
    StringBuilder sb = new StringBuilder();
    sb.append("cartes ").append(nom).append(" { ");
    for(Integer carte : jeu){
        sb.append(String.format("%02d", carte)).append(" ");
    }
    sb.append("}");
    return sb.toString();
}
```

Code Source

Joueur.java

```
    /**
     * Créer une chaîne de caractères comportant le nom du joueur, sa base,
     * le nombre de carte de son jeu ainsi que le nombre de carte dans sa pioche.
     *
     * @return                          la chaîne de caractères
     */
    @Override
    public String toString() {
        return nom + " ^["+ String.format("%02d", ascendant) + "]"
                + " v[" + String.format("%02d", descendant) + "]"
                + " (m" + jeu.size() + "p" + (pioche.getLastIdx() + 1) + ")";
    }

}
```

Tests Unitaires

JoueurTest.java

```java
package Test;

import Composantes.Joueur;
import org.junit.jupiter.api.Test;

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

// test la classe joueur
class JoueurTest {

    /**
     * Test si une carte est dans le jeu.
     * */
    @Test
    public void testEstDansLeJeu() {
        Joueur NORD = new Joueur("NORD");
        NORD.jeu.set(0, 21);
        NORD.jeu.set(1, 45);
        NORD.jeu.set(2, 52);
        NORD.jeu.set(3, 13);
        NORD.jeu.set(4, 9);
        NORD.jeu.set(5, 12);
        assertTrue(NORD.estDansLeJeu(21));
        assertTrue(NORD.estDansLeJeu(45));
        assertTrue(NORD.estDansLeJeu(52));
        assertTrue(NORD.estDansLeJeu(13));
        assertTrue(NORD.estDansLeJeu(9));
        assertTrue(NORD.estDansLeJeu(12));
        assertFalse(NORD.estDansLeJeu(1));
        assertFalse(NORD.estDansLeJeu(60));
        assertFalse(NORD.estDansLeJeu(19));
    }
```

Tests Unitaires

JoueurTest.java

```java
/**
 * Test de la pioche de carte.
 * */
@Test
public void testPiocherCarte(){
    Joueur NORD = new Joueur("NORD");
    NORD.piocherCarte();;
    assertTrue(NORD.jeu.size()==6);
    NORD.jeu.clear();
    NORD.piocherCarte();
    assertTrue(NORD.jeu.size()==1);
    NORD.piocherCarte();
    assertTrue(NORD.jeu.size()==2);
}

/**
 * Test du getter de nombre de cartes.
 * */
@Test
public void testGetNbPioche(){
    Joueur NORD = new Joueur("NORD");
    assertEquals(NORD.getNbPioche(), 52);
    NORD.jeu.clear();
    while(!NORD.jeuEstPlein())
        NORD.piocherCarte();
    assertEquals(NORD.getNbPioche(), 46);
    while(NORD.getNbPioche()!=0){
        NORD.jeu.clear();
        NORD.piocherCarte();
    }
    assertEquals(NORD.getNbPioche(), 0);
}
```

Tests Unitaires

JoueurTest.java

```java
/**
 * Test si le jeu est vide.
 * */
@Test
public void testJeuEstVide() {
    Joueur NORD = new Joueur("NORD");
    assertFalse(NORD.jeuEstVide());
    NORD.jeu.clear();
    assertTrue(NORD.jeuEstVide());
}

/**
 * Test si le jeu est plein.
 * */
@Test
public void testJeuEstPlein() {
    Joueur NORD = new Joueur("NORD");
    assertTrue(NORD.jeuEstPlein());
    NORD.jeu.clear();
    assertFalse(NORD.jeuEstPlein());
}

/**
 * Test l'action de jouer une carte.
 * */
@Test
public void testJouerCarte() {
    Joueur NORD = new Joueur("NORD");
    Joueur SUD = new Joueur("SUD");
    NORD.jeu.set(0, 23);
    NORD.jeu.set(1,34);
    NORD.jouerCarte(23, 'v');
    assertEquals(NORD.getDescendant(), 23);
    NORD.jouerCarte(34, '^', SUD);
    assertEquals(SUD.getAscendant(), 34);
}
}
```

Code Source

Input.java

```java
package Composantes;


/**
 * Les inputs sont les entrées données à notre programme, celles-ci sont indépendantes du programme.
 * Par conséquent, ces données ne peuvent être traitées par de simples assertions et necessite
 * un traitement particulier quant à leurs traitements.
 * Ici, il s'agit de pouvoir récupérer les inputs de l'utilisateur et pouvoir les manipuler
 * sans affecter son expérience (avec des crashs ou assertions levées par exemple).
 * (Classe Static)
 *
 * @author      Yannick li
 * @author      Stefan Radovanovic
 * @version     1, 2/27/2021
 * */
public class Input {


    /**
     * Décompose les différents coups d'un joueur.
     * Cette méthode utilise la méthode Split de la classe String qui permet de séparer
     * une chaîne de caractères en fragment stocké par la suite dans un tableau de String,
     * ici on sépare avec un regex nous permettant d'obtenir séparément chaque coup de l'utilisateur ("\\s+").
     *
     * @param UsersLine     le String à décomposer (l'entrée de l'uttilisateur).
     * @return              un tableau de String contenant tous les coups du joueur
     * @see                 String#split(String)
     */
    public static String[] decomposer(String UsersLine) {
        return UsersLine.split("\\s+");
    }
```

Code Source

Input.java

```java
/**
 * Vérifie le format d'un coup de l'utilisateur (non la sémantique).
 * Un coup doit avoir pour ses deux premiers caractère des entiers.
 * le troisième caractère peut être "^" ou un "v" pour les bases ascendantes et descendantes
 * enfin, si l'utilisateur joue son coup chez l'adversaire
 * le quatrième caractère doit être "\'" (une apostrophe)
 *
 * @param s             la chaine de caractères à vérifier.
 * @return              le coup est jouable ou non.
 * @see                 Character#isDigit(char)
 * */
private static boolean inputChecker(String s) {
    if(!(s.length()==4 || s.length()==3))
        return false;
    if(!(Character.isDigit(s.charAt(0)) && Character.isDigit(s.charAt(1))))
        return false;
    if(!(s.charAt(2)=='^' || s.charAt(2) == 'v'))
        return false;
    if(s.length()==4)
        return s.charAt(3) == '\'';
    return true;
}


/**
 * Verifie si la syntaxe des coups est correcte.
 * Cette méthode utilise les méthodes précédente pour vérifier l'intégralité des coups de l'utilisateur.
 * Elle verifie egalement s'il y a bien au moins deux coups.
 *
 * @param coups         les coups du utilisateur, stockés dans un tableau de String.
 * @return              les coups ont une syntaxe correcte ou non.
 * */
public static boolean isSyntaxValid(String[] coups){
    if(coups.length < 2)
        return false;
    for (String coup: coups) {
        if(!Input.inputChecker(coup))
            return false;
    }
    return true;
}
```

Code Source

Input.java

```
    /**
     * Retourne la carte qui est joué dans la chaine de caractères.
     * Cette methode est à utiliser qu'après avoir vérifié que le coup est jouable.
     * De plus, elle utilise la méthode substring pour faire une sous chaîne de caractère
     * contenant les caractères allant jusqu'à un certain indice.
     *
     * @param coup          le coup à jouer, sous la forme de String.
     * @return              la valeur de la carte.
     * @see                 String#substring(int)
     * @see                 Integer#parseInt(String)
     */
    public static int getCarte(String coup) {
        return Integer.parseInt(coup.substring(0, 2), 10); //   substring creer une sous chaine de caracteres
                                                           //   parseInt renvoie un entier en base 10 (radix)
    }


    /**
     * Retourne dans quelle base le coup va être joué.
     * Cette méthode est à utiliser qu'après avoir vérifié que le coup est jouable
     *
     * @param coup          le coup à jouer, sous la forme de String.
     * @return              le caractère représentant la base ("^" ou "v").
     */
    public static char getBase(String coup) {
        return coup.charAt(2);
    }

}
```

Tests Unitaires

InputTest.java

```java
package Test;

import Composantes.Input;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class InputTest {

    /**
     * Test du décomposeur de ligne.
     * */
    @Test
    public void testDecomposer() {
        String [] s = Input.decomposer("12v 39^ 46v' 59^");
        assertEquals(s[0], "12v");
        assertEquals(s[1], "39^");
        assertEquals(s[2], "46v'");
        assertEquals(s[3], "59^");
        assertEquals(s.length, 4);
    }

    /**
     * Test la syntaxe d'une entrée de joueur.
     * */
    @Test
    public void testIsSyntaxValid() {
        assertTrue(Input.isSyntaxValid(Input.decomposer("12v 15^")));
        assertTrue(Input.isSyntaxValid(Input.decomposer("12v 15v'")));
        assertTrue(Input.isSyntaxValid(Input.decomposer("12v' 15v' 81v 35v")));
        assertFalse(Input.isSyntaxValid(Input.decomposer("une erreur de saisie")));
        assertFalse(Input.isSyntaxValid(Input.decomposer("uneLonguePhrase")));
        assertFalse(Input.isSyntaxValid(Input.decomposer("12v")));
        assertFalse(Input.isSyntaxValid(Input.decomposer("deux elements")));
        assertFalse(Input.isSyntaxValid(Input.decomposer("12 12v")));
        assertFalse(Input.isSyntaxValid(Input.decomposer("12' 32v")));
        assertFalse(Input.isSyntaxValid(Input.decomposer("")));   // une saisie vide
    }
```

Tests Unitaires

InputTest.java

```
    /**
     * test la recuperation de la carte d'un coup
     * */
    @Test
    public void testGetCarte() {
        assertEquals(Input.getCarte("55v"), 55);
        assertEquals(Input.getCarte("09^'"), 9);
        assertEquals(Input.getCarte("95v'"), 95);
        assertEquals(Input.getCarte("12^"), 12);
    }

    /**
     * test la recuperation de base d'un coup
     * */
    @Test
    void testGetBase() {
        assertEquals(Input.getBase("55v'"), 'v');
        assertEquals(Input.getBase("55^'"), '^');
        assertEquals(Input.getBase("55^"), '^');
        assertEquals(Input.getBase("55v"), 'v');
    }
}
```

Code Source

Regles.java

```java
package Composantes;
/**
 * Les règles sont un ensemble d'instructions établi pour conditionner le bon déroulement d'un jeu.
 * Ils établissent les facultés et les contraintes qui sont présentées
 * à chaque joueur et doivent être clairement énoncées à chacun d'eux avant le début de la partie.
 * Il s'agit ici de réguler la sémantique des coups de l'uttilisateur.
 * (Classe Static)
 *
 * @author                  Yannick li
 * @author                  Stefan Radovanovic
 * @version 1, 2/27/2021
 * */
public class Regles {


    /**
     * Joue les coups d'un joueur et pioche selon le coup.
     *
     * @param coups             les coups du joueurs, sous la forme d'un tableau de String
     * @param j1                le joueur qui joue les coups.
     * @param j2                le joueur qui recoit (ou non) les coups.
     * */
    public static void jouerCoups(String[] coups,Joueur j1, Joueur j2) {
        boolean coupAdv = false;
        for (String coup: coups) {
            if(isCampEnnemie(coup)) {
                j1.jouerCarte(Input.getCarte(coup), Input.getBase(coup), j2);
                coupAdv = true;
            }
            else j1.jouerCarte(Input.getCarte(coup), Input.getBase(coup));
        }
        regleDePioche(coupAdv, j1); // pioche les cartes selon les coups jouées
    }
```

Code Source

Regles.java

```java
/**
 *  Fait piocher au joueur les cartes selon si il a joué chez l'adversaire ou non.
 *  - 2 cartes si aucun coup n'a été jouée chez l'adversaire
 *  - rempli la main si un coup a été jouée chez l'adversaire
 *
 * @param jouerAd                   true si le joueur joue chez l'adversaire
 * @param j                         le joueur qui doit piocher
 */
private static void regleDePioche(boolean jouerAd, Joueur j) {
    if (jouerAd) {              //pioche jusqu'à que sa main soit pleine
        while (!j.jeuEstPlein() && !j.piocheEstVide())
            j.piocherCarte();
    }
    else                // n'a joué que sur ses bases, pioche 2 cartes
        for (int i = 0; i < 2; i++)
            if (!j.jeuEstPlein() && !j.piocheEstVide())
                j.piocherCarte();
}
```

Code Source

Regles.java

```java
/**
 * Vérifie si la carte jouée du joueur est posable sur une base donnée.
 * Conformément aux règles du jeu, un joueur peux poser sur :
 * - sa base ascendante, si la carte est supérieure ou est égale à la dizaine du dessous.
 * - sa base descendante, si la carte est inférieur ou est égale à la dizaine du dessus.
 * - la base ascendante adverse, si la carte est inférieur.
 * - la base descendante adverse, si la carte est supérieur.
 *
 * @param carte                 la carte du joueur
 * @param base                  la base choisie ( 'v' pour descendante et '^' pour ascendante)
 * @param poseur                le joueur qui pose sa carte
 * @param receveur              le joueur qui reçoit la carte
 * @return                      si la carte est posable ou non
 */
private static boolean estPosable(int carte, char base , Joueur poseur, Joueur receveur){
    if(!poseur.estDansLeJeu(carte)) //  si la carte fait partie du jeu ou non
        return false;
    if(receveur.equals(poseur)) {   //  si le joueur pose dans son camp
        if( base == 'v' )
            return receveur.getDescendant() > carte || (receveur.getDescendant() + 10 == carte);    //  dizaine au dessus
        else if( base == '^')
            return receveur.getAscendant() < carte || (receveur.getAscendant() - 10) == carte;    //  dizaine en dessous
    }
    else {  // camp adverse
        if( base == 'v')
            return receveur.getDescendant() < carte;
        else if( base == '^')
            return receveur.getAscendant() > carte;
    }
    return false;
}
```

Code Source

Regles.java

```java
/**
 * Vérifie si les coups du joueur sont jouable (la sémantique).
 * /!\ A utiliser en partant du principe que la syntaxe est bonne.
 * /!\ la méthode simule et applique les coups à des clones.
 * cad qu'il faut mettre en paramètre des clones.
 *
 * @param coups                 les coups du joueurs
 * @param j1Bis                 le joueur qui joue les coups
 * @param j2Bis                 le joueur adverse
 * @return                      si les coups sont valides ou non
 * @see                         Joueur#clone()
 * @see                         Input#isSyntaxValid(String[])
 * */
public static boolean areCoupsValid(String[] coups, Joueur j1Bis, Joueur j2Bis) {

    boolean coupAdv = false; // vérifie si un coups a été joué chez l'adversaire

    for(String coup : coups) {
      int carte = Input.getCarte(coup);
      char base = Input.getBase(coup);

      if(!j1Bis.estDansLeJeu(carte))
         return false;

      Joueur receveur = isCampEnnemie(coup) ? j2Bis : j1Bis; // le joueur qui reçoit la carte

      if(estPosable(carte, base, j1Bis, receveur)) {
         if(isCampEnnemie(coup)) {
            if(coupAdv)    // il est possible de jouer qu'une fois chez l'adversaire
               return false;
            j1Bis.jouerCarte(carte, base, receveur);
            coupAdv = true;

         }
         else j1Bis.jouerCarte(carte, base);
      } else return false;
    }
    return true;
}
```

Code Source

Regles.java

```java
/**
 * Methode récursive qui verifie si il y a au moins 2 combinaisons de cartes possible pour un joueur.
 * L'algorithme simule une partie avec les combinaisons courantes (base, carte en main etc).
 * /!\ la methode simule et applique les coups à des clones.
 *
 * @param j1                le joueur qui doit jouer
 * @param j2                le 2ème joueur
 * @param nb                le nombre de coups possibles (deux)
 * @param coupAdv           si l'on a déjà joué dans chez l'adversaire ou non
 * @return                  si la partie continue ou non
 * @see                     Joueur#clone()
 * */
public static boolean partieContinue(Joueur j1, Joueur j2, int nb, boolean coupAdv){
    if(nb>=2)  //  si il y a au moins 2 coups à jouer
        return true;
    for(int carte : j1.jeu){
        // A chaque carte jouée
        Joueur j1Bis = j1.clone();
        Joueur j2Bis = j2.clone();
        //  la vérification commence en jouant les coups possible chez le joueur courant
        if(estPosable(carte, 'v', j1Bis, j1Bis)) {
                j1Bis.jouerCarte(carte, 'v');
                nb++;
            if (partieContinue(j1Bis, j2Bis, nb, coupAdv))
                return true;
        }
        if(estPosable(carte, '^', j1Bis, j1Bis)) {
            j1Bis.jouerCarte(carte, '^');
            nb++;
            if (partieContinue(j1Bis, j2Bis, nb, coupAdv))
                return true;
        }
```

Code Source

Regles.java

```
        //  verification chez le joueur adverse
        if(!coupAdv){ //  si l'on a pas déjà joué chez l'adversaire
            if(estPosable(carte, 'v', j1Bis, j2Bis)) {
                j1Bis.jouerCarte(carte, 'v', j2Bis);
                nb++;
                coupAdv=true;
                if (partieContinue(j1Bis, j2Bis, nb, coupAdv))
                    return true;
                else
                    coupAdv=false;
                    nb--;
            }
            if(estPosable(carte, '^', j1Bis, j2Bis)) {
                j1Bis.jouerCarte(carte, '^', j2Bis);
                nb++;
                coupAdv=true;
                if (partieContinue(j1Bis, j2Bis, nb, coupAdv))
                    return true;
                else
                    coupAdv=false;
                    nb--;
            }
        }
    }
    return false;
}
```

Code Source

Regles.java

```
    /**
     * Verifie si un coup joué est destiné au joueur adverse (à utiliser après estPosable)
     *
     * @return              si le coup est pour le joueur adverse ou non
     * */
    private static boolean isCampEnnemie(String coup) {
        return coup.length() == 4; // si un coup constitue une chaine de 4 caracteres alors il doit etre
                                   // destiné à l'adversaire car il comprend une apostrophe. exemple : 34v' (4 char)
    }


    /**
     *  Verifie si le joueur a posé la totalité de ses cartes.
     *
     * @return              si il n'y a plus de carte à jouer
     * */
    public static boolean partieFinie(Joueur j){
        return (j.jeuEstVide() && j.getNbPioche()==0);
    }

}
```

Tests Unitaires

ReglesTest.java

```java
package Test;

import Composantes.Joueur;
import Composantes.Regles;
import Composantes.Input;
import org.junit.jupiter.api.Test;

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

class ReglesTest {

    /**
     * test la possibilité de poser une carte sur une base adverse ou non
     */
    @Test
    public void testJouerCoups() {
        Joueur NORD = new Joueur("NORD");
        Joueur SUD = new Joueur("SUD");
        SUD.jeu.set(0, 12);
        SUD.jeu.set(1, 39);
        SUD.jeu.set(2, 46);
        SUD.jeu.set(3, 59);
        Regles.jouerCoups(Input.decomposer("12v 39^ 46v' 59^"), SUD, NORD);
        assertEquals(SUD.getDescendant(), 12);
        assertEquals(SUD.getAscendant(), 59);
        assertEquals(NORD.getDescendant(), 46);
    }
```

Tests Unitaires

ReglesTest.java

```java
/**
 * test la verification semantique des coups
 * la syntaxe est bonne, on test seulement la semantique
 *
 * */
@Test
public void testAreCoupsValid() {
    Joueur NORD = new Joueur("NORD");
    Joueur SUD = new Joueur("SUD");

    NORD.setAscendant(30);      //  les bases sont mis à 30 pour une meilleur manipulation
    NORD.setDescendant(30);     //
    SUD.setAscendant(30);       //
    SUD.setDescendant(30);      //

    SUD.jeu.set(0, 12);
    SUD.jeu.set(1, 39);
    SUD.jeu.set(2, 46);
    SUD.jeu.set(3, 59);
    SUD.jeu.set(4, 22);
    SUD.jeu.set(5, 14);

    assertTrue(Regles.areCoupsValid(Input.decomposer("12v 39^ 46^"), SUD.clone(), NORD.clone()));
    assertTrue(Regles.areCoupsValid(Input.decomposer("14v 12v"), SUD.clone(), NORD.clone()));
    assertTrue(Regles.areCoupsValid(Input.decomposer("12^' 39^ 46^"), SUD.clone(), NORD.clone()));
    assertTrue(Regles.areCoupsValid(Input.decomposer("12v 39v' 46^"), SUD.clone(), NORD.clone()));

    assertFalse(Regles.areCoupsValid(Input.decomposer("12^ 39v 46^'"), SUD.clone(), NORD.clone()));
    assertFalse(Regles.areCoupsValid(Input.decomposer("14^ 22v'"), SUD.clone(), NORD.clone()));
    assertFalse(Regles.areCoupsValid(Input.decomposer("19^ 39v' 46^"), SUD.clone(), NORD.clone()));
    assertFalse(Regles.areCoupsValid(Input.decomposer("14v' 39v 46^"), SUD.clone(), NORD.clone()));

}

/**
 * test la detection de fin de partie pour un joueur
 * */
@Test
public void testPartieFinie() {
    Joueur NORD = new Joueur("NORD");
    assertFalse(Regles.partieFinie(NORD));
    NORD.jeu.clear();
```

Tests Unitaires

ReglesTest.java

```
    for (int i = 0; i < NORD.getNbPioche();) {
        NORD.piocherCarte();
        NORD.jeu.clear();
    }
    assertTrue(Regles.partieFinie(NORD));
}

/**
 * test de la fonction partieContinue
 * */
@Test
public void testPartieContinue(){
    Joueur NORD = new Joueur("NORD");
    Joueur SUD = new Joueur("SUD");
    assertTrue(Regles.partieContinue(NORD.clone(), SUD.clone(), 0,false));
    // avec des bases injouables
    NORD.setAscendant(61);
    NORD.setDescendant(0);
    SUD.setAscendant(61);
    SUD.setDescendant(0);
    assertFalse(Regles.partieContinue(NORD.clone(), SUD.clone(), 0, false));
    // avec un jeu vide
    NORD.jeu.clear();
    assertFalse(Regles.partieContinue(NORD.clone(), SUD.clone(), 0, false));

    SUD.jeu.clear();
    NORD.jeu.add(0,59);
    NORD.jeu.add(1,58);
    NORD.setAscendant(57);
    NORD.setDescendant(2);
    SUD.setAscendant(1);
    SUD.setDescendant(60);
    assertTrue(Regles.partieContinue(NORD.clone(), SUD.clone(), 0,false));
    NORD.setAscendant(58);
    assertFalse(Regles.partieContinue(NORD.clone(), SUD.clone(), 0,false));
```

Tests Unitaires

ReglesTest.java

```
        //cas spécifique
        NORD.setAscendant(49);
        NORD.setDescendant(2);
        SUD.setAscendant(53);
        SUD.setDescendant(9);
        SUD.jeu.add(0,34);
        SUD.jeu.add(1,44);
        SUD.jeu.add(2,38);
        SUD.jeu.add(3,6);
        SUD.jeu.add(4,41);
        SUD.jeu.add(5,39);
        assertTrue(Regles.partieContinue(SUD.clone(), NORD.clone(), 0, false));


    }
}
```

Code Source

Application.java

```java
package Appli;

import Composantes.Joueur;
import Composantes.Regles;
import Composantes.Input;
import java.util.Scanner;

/**
 * L'application agence chacune de nos composantes
 * et permet de jouer au jeu The Game - le Duel.
 *
 * @author      Yannick li
 * @author      Stefan Radovanovic
 * @version     1, 2/27/2021
 * */
public class Application {

    /**
     * Récupère l'entrée de l'uttilisateur, son coup.
     * Cette méthode utilise la classe Scanner qui couplé à un flux comme system.in
     * permet d'extraire les informations données qui sont ensuite retourné dans un String.
     * "> " est affiché pour correspondre aux normes d'affichages.
     *
     * @return          un String contenant les coups du joueur.
     * @see             Scanner
     * */
    private static String getUsersLine(Scanner sc) {
        System.out.print("> ");
        return sc.nextLine();
    }


    /**
     * Affiche les informations des joueurs NORD et SUD ainsi que la main du joueur courant.
     *
     * @param NORD      le joueur NORD
     * @param SUD       le joueur SUD
     * @param courant   le joueur courant
     * */
    private static void showGame(Joueur NORD, Joueur SUD, Joueur courant) {
        System.out.println(NORD);
        System.out.println(SUD);
        System.out.println(courant.afficherJeu());
    }
```

Code Source

Application.java

```
public static void main(String[] args) {
    Joueur NORD = new Joueur("NORD");
    Joueur SUD = new Joueur("SUD");

    Joueur courant = NORD;      //  Références des objets joueurs
    Joueur passif = SUD;        //

    @SuppressWarnings("resource")
    Scanner sc = new Scanner(System.in);

    while(Regles.partieContinue(courant.clone(), passif.clone(), 0,false)) {

        showGame(NORD, SUD, courant);

        String[] coups = Input.decomposer(getUsersLine(sc));

        while(!(Input.isSyntaxValid(coups) &&
                Regles.areCoupsValid(coups, courant.clone(), passif.clone()))){

            System.out.print("#");
            coups = Input.decomposer(getUsersLine(sc));
        }

        int nbCarteAvantCoup = courant.jeu.size();  //  save
        Regles.jouerCoups(coups, courant, passif);

        //  calcul permettant d'obtenir le nombre de carte piochée.
        int nbCartePiochee = courant.jeu.size() - (nbCarteAvantCoup - coups.length);

        System.out.println(coups.length + " cartes posées, " + nbCartePiochee + " cartes piochées");

        //  permutation
        Joueur tmp = passif;
        passif = courant;
        courant = tmp;

        if(Regles.partieFinie(passif))
            break;  //  la partie se finit avant de laisser le prochain joueur jouer

    }
    showGame(NORD, SUD, courant);

    System.out.println("partie finie, " + passif.nom + " a gagné");
    sc.close();
}
}
```

Mise en avant

On a voulu mettre en avant notre algorithme de détection de fin de partie donc on le met en fin d'annexe :

```
NORD ^[59] v[05] (m6p39)
SUD ^[41] v[04] (m2p44)
cartes NORD { 15 20 26 33 40 42 }
> |
```

Non la partie n'est pas finie ☺ (" 15v 20^' " par exemple)

```
2 cartes posées, 2 cartes piochées
NORD ^[55] v[13] (m2p48)
SUD ^[46] v[22] (m2p48)
cartes NORD { 16 18 }
partie finie, SUD a gagné
```

Partie finie en trois rounds