Projet BPO

Le Duel

Robin Fauchery
Groupe 112

Sasha Contrepois
Groupe 112

Table des matières

Table des matières 2
Introduction 2
Diagramme de classe 3
Tests 3
Bilan 4
Tests 5
Sources 23

Introduction

Ce programme a été réalisé dans le cadre du projet du module de BPO (Base de la Programmation Orientée Objet) du 2nd semestre de 1ère année d'IUT Informatique. Ce projet nous a permis de mettre en application les concepts basiques de Java étudiés lors de cette période.

Le but de programme est de permettre à deux joueurs de s'affronter dans une partie endiablée de Duel, un jeu de carte :

Les deux joueurs ont tous deux une pioche avec des cartes allant de 2 à 59, et devant eux se trouvent pour chacun deux piles :

- Ascendante qui commence à 1 (ASC)
- Descendante qui commence à 60 (DESC)

Chacun leur tour, ils devront poser au moins deux cartes de la manière suivante :

	Sur soi-même	Sur le joueur ennemi
Pile Ascendante	C > Haut de la pile **OU** C == Haut de la pile - 10	C < Haut de la pile
Pile Descendante	C < Haut de la pile **OU** C == Haut de la pile + 10	C > Haut de la pile

On voit donc apparaître les principaux objets qui vont devoir être représentés sur le diagramme de classe lors de la conception et programmé lors de la réalisation du programme :

- Pile & Type de Pile
- Carte
- Pioche
- Joueur & Type de Joueur
- Action & Type d'Action

Diagramme de classe

Ce diagramme de classe ne représente pas les méthodes et attributs de chaque classe.

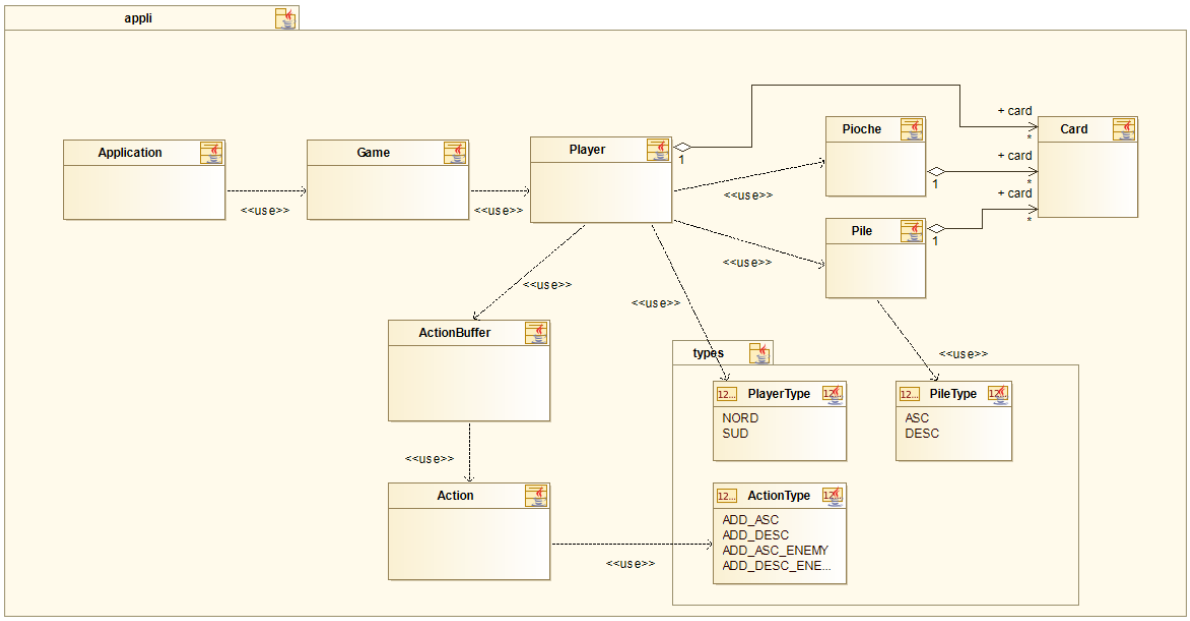
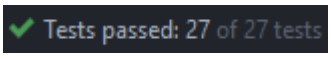


Diagramme réalisé à l'aide du logiciel Open Source Modelio

Tests

Tout au long de ce projet, nous avons suivi une démarche de développement piloté par les tests (TDD). Nous avons donc réalisé nos tests à l'aide de jUnit5 et ces derniers ont été programmé avant de programmer la fonction qu'elle doit tester. Ainsi, cela nous a permis de suivre une démarche de programmation intelligente et produire un code de qualité *(même si c'est à vous de juger cela)*.

Nous avons remarqué qu'il était important, dans cette méthode de développement, de bien documenter les tests effectués. En effet, pour des compliqués, la fonction devient rapidement grande et il est alors facile de s'y perdre et difficile de la relire. Expliquer le test effectué avant est donc une bonne solution pour éviter cela.

Nous avons réalisé au total 27 tests différents, et tous réussissent. 

Pour une meilleure lisibilité, les tests se trouvent à la fin du rapport, à partir de la page 5. Ils sont également présents dans l'archive de rendu, dans le dossier Test.

Bilan

Pour conclure ce rapport, nous pouvons dire que nous sommes fiers de notre réalisation. En effet, ce programme correspond au cahier des charges ou du moins aux caractéristiques définis dans le sujet du projet.

Malgré des passages difficile – *la détection de fin de partie* – ce projet aura été formateur.

En outre, la première étant que la partie de réflexion et conception du programme est essentielle pour un développement intelligent. Prêter du temps à la décomposition du jeu en objets, de manière qu'ils soient le plus possible réutilisables, est alors une majeure partie du travail.

De plus, le développement dirigé par les tests nous aura montré que rédiger des tests unitaires est un vrai métier car cela nécessite beaucoup de réflexion sur tous les cas possibles à gérer, et comment écrire ces tests le plus efficacement possible. Nous avons aussi remarqué qu'il est essentiel d'organiser ces tests car on s'y perd très vite …

Cependant, les tests ont été un outil formidable car cela nous a facilité le débogage en trouvant les erreurs bien plus rapidement qu'en effectuant des tests classiques : simuler *une partie est très chronophage, ce n'est pas une solution viable.*

Même si notre projet est considéré comme terminé, on pourrait dans l'état actuel considérer quelques améliorations. Notamment, nous pourrions améliorer la distinction entre la partie Affichage et « Cœur » du jeu. Nous pourrions aussi améliorer l'utilisation des Exceptions, en ajoutant des PlayerException ou ActionException, permettant de mieux identifier les dans le code.

Il aurait également été possible de modifier la classe Pile pour le rendre plus générique et ainsi faire de la classe Pioche (Pioche du joueur) une classe héritant de Pile.

Enfin, nous pourrions ajouter au jeu une aide permettant de donner au joueur une liste des coups possibles, cette fonction ayant déjà été programmée pour déterminer si le joueur a perdu ou non, cela serait plutôt facile et rapide à mettre en place.

Annexe 1 : Tests

--ActionBufferTest.java--

```java
package appli;

import org.junit.jupiter.api.Test;

import appli.types.*;

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

class ActionBufferTest {

    @Test

    void add() {

        Player player = new Player(PlayerType.NORD);
        Player enemy = new Player(PlayerType.SUD);

        ActionBuffer actionBuffer = new ActionBuffer(player, enemy);

        Action action = new Action(ActionType.ADD_ASC, new Card(9));
        player.addCardInHand(new Card(9));
        assertDoesNotThrow(() -> actionBuffer.add(action));
        assertThrows(IllegalArgumentException.class ,() -> actionBuffer.add(action));

        // Test de l'encapsulation
        // action est déjà ajoutée dans l'ActionBuffer, avec une carte d'une valeur 9
        action.setCard(new Card(5));
        assertNotEquals(new Card(5), actionBuffer.getActions().get(0).getCard());

        // On essaye d'ajouter des actions non valides
        Action illegalAction = new Action(ActionType.ADD_ASC, new Card(4));
        assertThrows(Exception.class, () -> actionBuffer.add(illegalAction));
        illegalAction.setType(ActionType.ADD_ASC_ENEMY);
        assertThrows(Exception.class, () -> actionBuffer.add(illegalAction));
        illegalAction.setCard(null);
        assertThrows(Exception.class, () -> actionBuffer.add(illegalAction));

    }

    @Test
    void canAdd() {

        Player p;
        Player e;
        ActionBuffer actionBuffer;
        Action action;

        /*
         * On vérifie que l'on ne peut pas ajouter une
         * action qui n'est pas valide
         */
        p = new Player(PlayerType.SUD);
        e = new Player(PlayerType.NORD);
        actionBuffer = new ActionBuffer(p, e);

        action = new Action(null, null);
        assertFalse(actionBuffer.canAdd(action));
```

```
action = new Action(null, new Card(20));
assertFalse(actionBuffer.canAdd(action));

/*
 * On vérifie que l'on ne peut pas ajouter deux fois la même carte
 * sur une pile ennemie
 */
p = new Player(PlayerType.SUD);
e = new Player(PlayerType.NORD);
actionBuffer = new ActionBuffer(p, e);

e.getPileAsc().add(new Card(31));
p.addCardInHand(new Card(20));

action = new Action(ActionType.ADD_ASC_ENEMY, new Card(20));
assertTrue(action.isValid(p, e));
assertTrue(actionBuffer.canAdd(action));
actionBuffer.add(action);
assertFalse(actionBuffer.canAdd(action));

action.setCard(new Card(18));

assertFalse(actionBuffer.canAdd(action));

/*
 * On vérifie que l'on ne peut pas placer la même carte
 * deux fois sur une même pile
 */
p = new Player(PlayerType.SUD);
e = new Player(PlayerType.NORD);
actionBuffer = new ActionBuffer(p, e);
action = new Action(ActionType.ADD_ASC, new Card(10));

p.addCardInHand(new Card(10));

assertTrue(actionBuffer.canAdd(action));
actionBuffer.add(action);
assertFalse(actionBuffer.canAdd(action));

/*
 * On vérifie que l'on ne peut pas placer la même
 * carte sur la pile ASC et DESC (ET ennemi)
 */
p = new Player(PlayerType.SUD);
e = new Player(PlayerType.NORD);
ActionBuffer actionBuffer1 = new ActionBuffer(p, e);

Action addDesc = new Action(ActionType.ADD_DESC, new Card(30));
Action addAsc = new Action(ActionType.ADD_ASC, new Card(30));
Action addEDesc = new Action(ActionType.ADD_ASC_ENEMY, new Card(30));
Action addEAsc = new Action(ActionType.ADD_ASC_ENEMY, new Card(30));
p.addCardInHand(new Card(30));

assertTrue(actionBuffer1.canAdd(addAsc));
assertDoesNotThrow(() -> actionBuffer1.add(addAsc));

assertFalse(actionBuffer1.canAdd(addDesc));
assertThrows(IllegalArgumentException.class ,() -> actionBuffer1.add(addDesc));
```

```
        assertFalse(actionBuffer1.canAdd(addEDesc));
        assertThrows(IllegalArgumentException.class ,() -> actionBuffer1.add(addEDesc));

        assertFalse(actionBuffer1.canAdd(addEAsc));
        assertThrows(IllegalArgumentException.class ,() -> actionBuffer1.add(addEAsc));

        /*
         * On vérifie les règles d'ajout +10 et -10
         *
         * (Scénario issu d'un bug)
         * NORD : ^[32] v[48] (m6p45)
         * cartes NORD { 13 19 34 40 53 58 }
         *
         * Impossible de faire : 58v 53v
         */
        p = new Player(PlayerType.SUD);
        e = new Player(PlayerType.NORD);
        ActionBuffer actionBuffer2 = new ActionBuffer(p, e);
        p.addCardInHand(new Card(53));
        p.addCardInHand(new Card(58));
        p.getPileDesc().add(new Card(48));

        Action action1 = new Action(ActionType.ADD_DESC, new Card(58));
        Action action2 = new Action(ActionType.ADD_DESC, new Card(53));
        assertTrue(actionBuffer2.canAdd(action1));
        assertDoesNotThrow(() -> actionBuffer2.add(action1));

        assertNotEquals(action1.getCard(), action2.getCard());

        assertTrue(actionBuffer2.canAdd(action2));
        assertDoesNotThrow(() -> actionBuffer2.add(action2));

    }

    @Test
    void clear() {

        Player player = new Player(PlayerType.NORD);
        Player enemy = new Player(PlayerType.SUD);

        ActionBuffer actionBuffer = new ActionBuffer(player, enemy);
        Action action = new Action(ActionType.ADD_ASC, new Card(9));
        player.addCardInHand(new Card(9));
        actionBuffer.add(action);

        enemy.getPileAsc().add(new Card(31));
        player.addCardInHand(new Card(20));
        Action action2 = new Action(ActionType.ADD_ASC_ENEMY, new Card(20));
        actionBuffer.add(action2);

        assertEquals(2, actionBuffer.getActions().size());

        actionBuffer.clear();

        assertEquals(0, actionBuffer.getActions().size());

    }

    @Test
```

```
    void containsEnemyAction(){

        Player player = new Player(PlayerType.NORD);
        Player enemy = new Player(PlayerType.SUD);

        ActionBuffer actionBuffer = new ActionBuffer(player, enemy);

        assertFalse(actionBuffer.containsEnemyAction());

        player.addCardInHand(new Card(5));
        enemy.getPileAsc().add(new Card(10));

        Action action = new Action(ActionType.ADD_ASC_ENEMY, new Card(5));
        actionBuffer.add(action);

        assertTrue(actionBuffer.containsEnemyAction());

    }

}
```

--ActionTest.java--

```java
package appli;

import appli.types.ActionType;
import appli.types.PileType;
import appli.types.PlayerType;
import org.junit.jupiter.api.Test;

import java.util.HashMap;

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

class ActionTest {

    @Test
    void constructorByString(){

        HashMap<String, Boolean> tests = new HashMap<>();
        tests.put("erreur de saisie", false);
        tests.put("1^^'", false);
        tests.put("09^^", false);
        tests.put("", false);
        tests.put("05^", true);
        tests.put("45v", true);
        tests.put("45^'", true);

        for(String key : tests.keySet()){

            if(tests.get(key))
                assertDoesNotThrow(() -> new Action(key));
            else
                assertThrows(Exception.class ,() -> new Action(key));

        }
    }

    @Test
    void isEnemyTarget(){

        Action action = new Action(ActionType.ADD_ASC_ENEMY, new Card(5));
        assertTrue(action.isEnemyTarget());

        action.setType(ActionType.ADD_DESC_ENEMY);
        assertTrue(action.isEnemyTarget());

        action.setType(ActionType.ADD_ASC);
        assertFalse(action.isEnemyTarget());

        action.setType(ActionType.ADD_DESC);
        assertFalse(action.isEnemyTarget());

    }
```

```java
@Test
    void isValid(){
        Player p;
        Player e;
        Action action;
        ActionBuffer actionBuffer;
        /*
         * On vérifie que les actions comportant des
         * types et cartes nulles sont non valides
         */
        p = new Player(PlayerType.SUD);
        e = new Player(PlayerType.NORD);
        p.addCardInHand(new Card(5));
        p.addCardInHand(new Card(60));

        action = new Action(null, null);
        assertFalse(action.isValid(p, e));
        for(ActionType type : ActionType.values()) {
            action = new Action(type, null);
            assertFalse(action.isValid(p, e));
        }
        action = new Action(null, new Card(60));
        assertFalse(action.isValid(p, e));

        /*
         * On vérifie que l'on ne peut pas ajouter une
         * carte que l'on ne possède pas
         */
        p = new Player(PlayerType.SUD);
        e = new Player(PlayerType.NORD);

        action = new Action(ActionType.ADD_ASC, new Card(5));
        assertFalse(action.isValid(p, e));
        p.addCardInHand(new Card(5));
        assertTrue(action.isValid(p, e));
        /*
         * On vérifie que l'on ne peut pas ajouter :
         * - une carte < sur la pile ASC : (La carte 20 sur la carte 25)
         * - une carte > sur la pile DESC : (La carte 40 sur la carte 35)
         * Mais que l'inverse est possible
         */
        p = new Player(PlayerType.SUD);
        e = new Player(PlayerType.NORD);
        p.addCardInHand(new Card(20));
        p.addCardInHand(new Card(40));
        p.getPileAsc().add(new Card(25));
        p.getPileDesc().add(new Card(35));

        action = new Action(ActionType.ADD_ASC, new Card(20));
        assertFalse(action.isValid(p, e));

        action = new Action(ActionType.ADD_DESC, new Card(40));
        assertFalse(action.isValid(p, e));

        action = new Action(ActionType.ADD_DESC, new Card(20));
        assertTrue(action.isValid(p, e));

        action = new Action(ActionType.ADD_ASC, new Card(40));
        assertTrue(action.isValid(p, e));
```

```
        /*
         * On vérifie les règles d'ajout +10 et -10
         *
         * (Scénario issu d'un bug)
         * NORD : ^[32] v[48] (m6p45)
         * cartes NORD { 13 19 34 40 53 58 }
         *
         * Impossible de faire : 58v 53v
         */
        p = new Player(PlayerType.SUD);
        e = new Player(PlayerType.NORD);
        p.addCardInHand(new Card(58));
        p.getPileDesc().add(new Card(48));

        action = new Action(ActionType.ADD_DESC, new Card(58));
        assertTrue(action.isValid(p, e));

        /*
         * On vérifie les règles d'ajout sur la pile adverse :
         * - ASC : Doit être inférieur
         * - DESC : Doit être supérieur
         */
        p = new Player(PlayerType.SUD);
        e = new Player(PlayerType.NORD);
        p.addCardInHand(new Card(40));
        p.addCardInHand(new Card(30));
        p.addCardInHand(new Card(25));
        p.addCardInHand(new Card(20));

        action = new Action(ActionType.ADD_ASC_ENEMY, new Card(20));
        assertFalse(action.isValid(p, e));
        e.getPileAsc().add(new Card(25));
        assertTrue(action.isValid(p, e));

        action = new Action(ActionType.ADD_DESC_ENEMY, new Card(40));
        assertFalse(action.isValid(p, e));
        e.getPileDesc().add(new Card(30));
        assertTrue(action.isValid(p, e));

    }

    // Getters & Setters ==========----------

    @Test
    public void getTargetPile(){

        Action action = new Action(ActionType.ADD_ASC, new Card(6));
        assertEquals(PileType.ASC, action.getTargetPile());

        action = new Action(null, new Card(6));
        assertNull(action.getTargetPile());

    }

}
```

--ApplicationTest.java--

```java
package appli;

import appli.types.ActionType;
import appli.types.PlayerType;
import org.junit.jupiter.api.Test;

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

class ApplicationTest {

    /*
     * Cette classe permet de simuler différents scénarios particuliers,
     * difficiles a traiter.
     */
    @Test
    public void testScenarios(){

        /*
         *  NORD ^[24] v[46] (m6p45)
         *  SUD ^[25] v[52] (m6p46)
         *  cartes NORD { 10 13 30 52 57 58 }
         *
         *  Action : 58v' 30^
         *
         *  Le bug est sûrement lié à un problème de caractère lorsqu'il a été tapé
         */
        Player p = new Player(PlayerType.NORD);
        Player e = new Player(PlayerType.SUD);

        p.getPileAsc().add(new Card(24));
        p.getPileDesc().add(new Card(46));
        e.getPileAsc().add(new Card(25));
        e.getPileDesc().add(new Card(52));

        int[] val = { 10, 13, 30, 52, 57, 58 };

        for(int v : val)
            p.addCardInHand(new Card(v));

        ActionBuffer buffer = new ActionBuffer(p,e);
        Action a1 = new Action(ActionType.ADD_DESC_ENEMY, new Card(58));
        Action a2 = new Action(ActionType.ADD_ASC, new Card(30));

        assertTrue(buffer.canAdd(a1));
        assertDoesNotThrow(() -> buffer.add(a1));

        assertTrue(buffer.canAdd(a2));
        assertDoesNotThrow(() -> buffer.add(a2));

        assertDoesNotThrow(() -> p.executeActionBuffer(buffer));

        assertEquals(new Card(30), p.getPileAsc().getTop());
        assertEquals(new Card(58), e.getPileDesc().getTop());

    }

}
```

--CardTest.java--

```java
package appli;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

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

class CardTest {

    @Test
    public void testConstructor(){
        assertThrows(IllegalArgumentException.class, () -> new Card(0));
        assertThrows(IllegalArgumentException.class, () -> new Card(61));
        assertThrows(IllegalArgumentException.class, () -> new Card(-60));

        assertDoesNotThrow(() -> new Card(5));
        assertDoesNotThrow(() -> new Card(1));
        assertDoesNotThrow(() -> new Card(60));
    }

    @Test
    @DisplayName("Constructeur de Carte par copie")
    public void testConstructorCopy(){
        Card card = new Card(5);
        Card copy = new Card(card);

        assertFalse(card == copy);
        assertEquals(card, copy);
    }

    @Test
    public void testToString(){
        Card c = new Card(5);
        assertEquals("05", c.toString());

        c = new Card(50);
        assertEquals("50", c.toString());
    }

    @Test
    public void testEquals(){
        Card c1 = new Card(5);
        Card c2 = new Card(5);
        Card c3 = new Card(3);

        assertEquals(c1, c1);
        assertNotEquals(null, c1);
        assertNotEquals("05", c1);

        assertEquals(c1, c2);
        assertEquals(c2, c1);

        assertNotEquals(c1, c3);
        assertNotEquals(c3, c1);
    }

}
```

--GameTest.java--

```java
package appli;

import appli.types.PlayerType;
import org.junit.jupiter.api.Test;

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

class GameTest {

    @Test
    public void testConstructor(){

        assertThrows(IllegalArgumentException.class, () -> new Game(new Player(), new
Player(PlayerType.SUD)));
        assertThrows(IllegalArgumentException.class, () -> new Game(new
Player(PlayerType.SUD), new Player(PlayerType.SUD)));
        assertThrows(IllegalArgumentException.class, () -> new Game(new
Player(PlayerType.NORD), new Player(PlayerType.NORD)));

        assertDoesNotThrow(() -> new Game(new Player(PlayerType.NORD), new
Player(PlayerType.SUD)));

    }

}
```

--PiocheTest.java--

```java
package appli;

import org.junit.jupiter.api.Test;

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

class PiocheTest {

    @Test
    public void testDrawCard(){
        Pioche pioche = new Pioche();
        assertDoesNotThrow(pioche::drawCard);
    }

    @Test
    public void testIsEmpty(){
        Pioche pioche = new Pioche();

        assertFalse(pioche.isEmpty());

        for (int i = 2; i <= 59; i++)
            assertDoesNotThrow(pioche::drawCard);

        assertTrue(pioche.isEmpty());
    }

}
```

--PileTest.java--

```java
package appli;

import appli.types.PileType;
import org.junit.jupiter.api.Test;

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

class PileTest {

    @Test
    public void testCanAdd(){

        Pile pileAsc = new Pile(PileType.ASC);
        assertTrue(pileAsc.canAdd(new Card(2)));
        assertTrue(pileAsc.canAdd(new Card(40)));
        pileAsc.add(new Card(40));
        assertFalse(pileAsc.canAdd(new Card(35)));
        assertTrue(pileAsc.canAdd(new Card(41)));

        Pile pileDesc = new Pile(PileType.DESC);
        assertTrue(pileDesc.canAdd(new Card(58)));
        pileDesc.add(new Card(58));
        assertTrue(pileDesc.canAdd(new Card(53)));
        assertFalse(pileDesc.canAdd(new Card(60)));

        /*
         * On vérifie la règle de +10 & -10
         */
        pileAsc = new Pile(PileType.ASC);
        pileDesc = new Pile(PileType.DESC);

        // +10
        pileDesc.add(new Card(48));
        assertFalse(pileDesc.canAdd(new Card(55)));
        assertTrue(pileDesc.canAdd(new Card(58)));
    }

    @Test
    public void testAdd(){

        Pile pileAsc = new Pile(PileType.ASC);
        assertTrue(pileAsc.canAdd(new Card(2)));
        assertTrue(pileAsc.canAdd(new Card(40)));

        Pile pileDesc = new Pile(PileType.DESC);
        assertTrue(pileDesc.canAdd(new Card(55)));
        assertFalse(pileDesc.canAdd(new Card(60)));

    }

}
```

--PlayerTest.java--

```java
package appli;

import appli.types.ActionType;
import appli.types.PlayerType;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

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

class PlayerTest {

    @Test
    @DisplayName("Vérification qu'un joueur peut jouer")
    void canPlay() {

        Player p = new Player(PlayerType.NORD);
        Player e = new Player(PlayerType.SUD);

        assertFalse(p.canPlay(e));

        p.addCardInHand(new Card(5));

        // Rappel, on peut jouer que si l'on peut poser 2 cartes
        assertFalse(p.canPlay(e));

        p.addCardInHand(new Card(55));

        assertTrue(p.canPlay(e));

        /*
         * Verifie que dans le cas suivant un joueur
         * peut jouer
         */
        p = new Player(PlayerType.NORD);
        e = new Player(PlayerType.SUD);

        p.getPileAsc().add(new Card(59));
        p.getPileDesc().add(new Card(40));

        p.addCardInHand(new Card(50));
        p.addCardInHand(new Card(48));
        p.addCardInHand(new Card(46));

        assertTrue(p.canPlay(e));

    }

    // Test Draw Card ====================

    @Test
    @DisplayName("Pioche une carte")
    public void testDrawCard(){

        Player p = new Player();

        assertEquals(0, p.getHand().size());
```

```
        // Drawing 1 card
        assertDoesNotThrow(() -> p.drawCard(1));
        assertEquals(1, p.getHand().size());

        // Drawing 57 cards from it's drawing pile
        assertDoesNotThrow(() -> p.drawCard(57));
        assertEquals(58, p.getHand().size());

        assertThrows(Exception.class, () -> p.drawCard(1));

    }

    @Test
    @DisplayName("Pioche automatiquement à la fin du tour")
    public void testDrawEndroundCard() {

        Player p1 = new Player(PlayerType.NORD);
        Player e1 = new Player(PlayerType.SUD);

        assertEquals(0, p1.getHand().size());

        p1.addCardInHand(new Card(2));
        p1.addCardInHand(new Card(50));
        p1.addCardInHand(new Card(58));

        /*
         * Pioche 2 cartes à la fin d'un tour
         * lorsque 3 cartes ont été posées
         * sans avoir posé de carte sur la pile ennemie
         */
        ActionBuffer actionBuffer = new ActionBuffer(p1, e1);
        assertDoesNotThrow(() -> actionBuffer.add(new Action(ActionType.ADD_ASC, new
Card(2))));
        assertDoesNotThrow(() -> actionBuffer.add(new Action(ActionType.ADD_DESC, new
Card(58))));
        assertDoesNotThrow(() -> actionBuffer.add(new Action(ActionType.ADD_DESC, new
Card(50))));
        assertDoesNotThrow(() -> p1.executeActionBuffer(actionBuffer));

        assertDoesNotThrow(() -> assertEquals(2, p1.drawEndRoundCards(actionBuffer)));
        assertEquals(2, p1.getHand().size());

        /*
         * La partie est à l'état suivant :
         *  NORD ^[01] v[50] m5p1
         *  SUD  ^[05] v[60] ----
         *
         * Si il pose 3 cartes, dont une sur une pile ennemie
         * il piochera 4 cartes pour remonter à 6
         */
        Player p2 = new Player(PlayerType.NORD);
        Player e2 = new Player(PlayerType.SUD);
        ActionBuffer actionBuffer2 = new ActionBuffer(p2, e2);

        p2.addCardInHand(new Card(1));
        p2.addCardInHand(new Card(2));
        p2.addCardInHand(new Card(3));
        p2.addCardInHand(new Card(4));
        p2.addCardInHand(new Card(45));
```

```
        assertEquals(5, p2.getHand().size());

        e2.getPileAsc().add(new Card(5));

        assertDoesNotThrow(() -> actionBuffer2.add(new Action(ActionType.ADD_ASC_ENEMY,
new Card(3))));
        assertDoesNotThrow(() -> actionBuffer2.add(new Action(ActionType.ADD_ASC, new
Card(4))));
        assertDoesNotThrow(() -> actionBuffer2.add(new Action(ActionType.ADD_DESC, new
Card(45))));

        assertDoesNotThrow(() -> p2.executeActionBuffer(actionBuffer2));

        assertDoesNotThrow(() -> assertEquals(4, p2.drawEndRoundCards(actionBuffer2)));
        assertEquals(6, p2.getHand().size());

        /*
         * La partie est à l'état suivant :
         * NORD ^[01] v[60] m2p1
         * SUD  ^[01] v[60] ----
         *
         * Si il reste 1 carte dans sa pioche, il piochera au maximum 1 carte
         */
        Player p3 = new Player(PlayerType.NORD);
        Player e3 = new Player(PlayerType.SUD);
        ActionBuffer actionBuffer3 = new ActionBuffer(p3, e3);

        // On laisse une seule carte dans la pioche
        while (p3.getPioche().size() != 1){
            try {
                p3.getPioche().drawCard();
            }catch (Exception exception) {
                exception.printStackTrace();
            }
        }
        assertEquals(1, p3.getPioche().size());

        p3.addCardInHand(new Card(5));
        p3.addCardInHand(new Card(6));

        assertDoesNotThrow(() -> actionBuffer3.add(new Action(ActionType.ADD_ASC, new
Card(5))));
        assertDoesNotThrow(() -> actionBuffer3.add(new Action(ActionType.ADD_ASC, new
Card(6))));
        assertDoesNotThrow(() -> p3.executeActionBuffer(actionBuffer3));
        assertDoesNotThrow(() -> assertEquals(1, p3.drawEndRoundCards(actionBuffer3)));
        assertEquals(1, p3.getHand().size());

        /*
         * La partie est à l'état suivant :
         * NORD ^[01] v[60] m4p0
         * SUD  ^[03] v[60] ----
         *
         * Il n'y a plus de cartes dans la pioche
         * A la fin du tour, il ne piochera donc pas de carte
         */
        Player p4 = new Player(PlayerType.NORD);
        Player e4 = new Player(PlayerType.SUD);
```

```
        ActionBuffer actionBuffer4 = new ActionBuffer(p4, e4);

        // On vide la pioche
        while (!p4.getPioche().isEmpty()){
            try {
                p4.getPioche().drawCard();
            }catch (Exception exception) {
                exception.printStackTrace();
            }
        }

        p4.addCardInHand(new Card(1));
        p4.addCardInHand(new Card(2));
        p4.addCardInHand(new Card(7));
        p4.addCardInHand(new Card(9));

        e4.getPileAsc().add(new Card(3));

        assertDoesNotThrow(() -> actionBuffer4.add(new Action(ActionType.ADD_ASC_ENEMY,
new Card(2))));
        assertDoesNotThrow(() -> actionBuffer4.add(new Action(ActionType.ADD_ASC, new
Card(7))));
        assertDoesNotThrow(() -> actionBuffer4.add(new Action(ActionType.ADD_ASC, new
Card(9))));
        assertDoesNotThrow(() -> p4.executeActionBuffer(actionBuffer4));
        assertDoesNotThrow(() -> assertEquals(0, p4.drawEndRoundCards(actionBuffer4)));
        assertEquals(1, p4.getHand().size());

    }

    // Test action ==========----------

    @Test
    @DisplayName("Exécution d'une action par un joueur")
    void execute(){

        Player p = new Player(PlayerType.NORD);
        Player e = new Player(PlayerType.SUD);

        Action action = new Action(ActionType.ADD_ASC, new Card(5));
        assertThrows(Exception.class, () -> p.executeAction(action, e));
        p.addCardInHand(new Card(5));
        assertEquals(1, p.getHand().size());

        assertDoesNotThrow(() -> p.executeAction(action, e));

        assertEquals(new Card(5), p.getPileAsc().getTop());
        assertFalse(p.hasCardInHand(action.getCard()));
        assertEquals(0, p.getHand().size());

        Action action1 = new Action(ActionType.ADD_ASC_ENEMY, null);
        assertThrows(Exception.class, () -> p.executeAction(action1, e));
    }

    @Test
    @DisplayName("Execution de toutes les actions d'un ActionBuffer")
    void executeAll(){

        Player player = new Player(PlayerType.NORD);
```

```
    Player enemy = new Player(PlayerType.SUD);
    ActionBuffer actionBuffer = new ActionBuffer(player, enemy);

    Action action1 = new Action(ActionType.ADD_ASC, new Card(5));
    player.addCardInHand(new Card(5));
    actionBuffer.add(action1);

    player.addCardInHand(new Card(6));
    enemy.getPileAsc().add(new Card(10));
    Action action2 = new Action(ActionType.ADD_ASC_ENEMY, new Card(6));
    actionBuffer.add(action2);

    assertDoesNotThrow(() -> player.executeActionBuffer(actionBuffer));

    assertEquals(5, player.getPileAsc().getTop().getValue());
    assertEquals(6, enemy.getPileAsc().getTop().getValue());

}

// Hand ==========----------

@Test
@DisplayName("Ajoute une carte en main")
public void addCardInHand(){

    Player p = new Player(null);
    p.addCardInHand(new Card(42));

    assertEquals(1, p.getHand().size());
    assertEquals(new Card(42), p.getHand().get(0));

    /*
     * On vérifie que les cartes sont bien triées à l'ajout de nouvelles cartes
     */
    p.addCardInHand(new Card(5));
    assertEquals(new Card(5), p.getHand().get(0));
    assertEquals(new Card(42), p.getHand().get(1));

    p.addCardInHand(new Card(59));
    assertEquals(new Card(5), p.getHand().get(0));
    assertEquals(new Card(42), p.getHand().get(1));
    assertEquals(new Card(59), p.getHand().get(2));

}

@Test
@DisplayName("Enlève une carte de la main")
public void removeCardFromHand(){

    Player p = new Player(null);

    // Tries to remove a card the player do not own
    assertThrows(Exception.class ,() -> p.removeCardFromHand(new Card(42)));

    p.addCardInHand(new Card(42));

    // Tries to remove a card the player owns
    assertDoesNotThrow(() -> p.removeCardFromHand(new Card(42)));
```

```
        assertEquals(0, p.getHand().size());

    }

    @Test
    @DisplayName("Vérification de la présence d'une carte en main")
    public void hasCardInHand(){

        Player p = new Player(null);
        assertFalse(p.hasCardInHand(new Card(5)));
        assertFalse(p.hasCardInHand(new Card(1)));

        p.addCardInHand(new Card(42));

        assertTrue(p.hasCardInHand(new Card(42)));
        assertFalse(p.hasCardInHand(new Card(41)));

    }

    @Test
    @DisplayName("Fonction de tri des cartes")
    public void sortCardsInHand(){

        Player p = new Player(PlayerType.SUD);
        p.getHand().add(new Card(8));
        p.getHand().add(new Card(55));
        p.getHand().add(new Card(2));

        assertEquals(8, p.getHand().get(0).getValue());
        assertEquals(55, p.getHand().get(1).getValue());
        assertEquals(2, p.getHand().get(2).getValue());

        p.sortCardsInHand();

        assertEquals(2, p.getHand().get(0).getValue());
        assertEquals(8, p.getHand().get(1).getValue());
        assertEquals(55, p.getHand().get(2).getValue());

    }
}
```

Annexe 2 : Sources

--ActionType.java--

```java
package appli.types;

public enum ActionType {

    /**
     * Ces types définissent les 4 actions possibles dans le jeu
     */

    ADD_ASC, ADD_DESC, ADD_ASC_ENEMY, ADD_DESC_ENEMY

}
```

--PileType.java--

```java
package appli.types;

public enum PileType {

    ASC, DESC

}
```

--PlayerType.java--

```java
package appli.types;

public enum PlayerType {

    NORD, SUD

}
```

--Card.java--

```java
package appli;

public class Card {

    private Integer value;

    /**
     * Constructeur
     * @param value Valeur de la carte (1 <= valeur <= 60)
     * @throws IllegalArgumentException Valeur incorrecte
     */
    public Card(Integer value) throws IllegalArgumentException{

        if(value < 1 || value > 60)
            throw new IllegalArgumentException("La valeur d'une carte appartient à
l'intervalle [1,60]");

        this.value = value;

    }
```

```java
    /**
     * Constructeur par copie
     */
    public Card(Card card){
        this(card.getValue());
    }

    /**
     * Retourne la carte sous la forme de deux digits
     * @return string Carte sous le format de deux nombres
     */
    @Override
    public String toString(){

        return String.format("%02d", this.getValue());

    }

    @Override
    public boolean equals(Object obj){

        if(obj == this)
            return true;
        if(obj == null)
            return false;
        if(!(obj instanceof Card))
            return false;

        Card c = (Card) obj;
        return (this.getValue().equals(c.getValue()));

    }

    /**
     * Vérifie si la carte est plus petite qu'une autre
     * @param c Carte avec laquelle on compare
     * @return True : La carte est plus petite, False sinon
     */
    public boolean isSmallerThan(Card c){

        return (this.getValue() < c.getValue());

    }

    // Getters and Setter ==========----------
    public Integer getValue() {
        return value;
    }

}
```

--Pile.java--

```java
package appli;

import appli.types.PileType;
import java.util.ArrayList;

public class Pile {

    private PileType pileType;
    private ArrayList<Card> cards;

    public Pile(PileType type){
        cards = new ArrayList<>();
        this.pileType = type;

        if(this.pileType.equals(PileType.ASC))
            cards.add(new Card(1));
        else if(this.pileType.equals(PileType.DESC))
            cards.add(new Card(60));
    }

    /**
     * Constructeur par copie
     * @param pile
     */
    public Pile(Pile pile){
        this.pileType = pile.pileType;
        this.cards = new ArrayList<>(pile.cards);
    }

    /**
     * Vérifie si une carte peut être ajoutée à la pile (ascendante ou descendante)
     * @param card Carte à vérifier
     * @return True : On peut l'ajouter, False sinon
     */
    public boolean canAdd(Card card){

        if(this.getPileType().equals(PileType.ASC)){
            if(this.getTop().isSmallerThan(card))
                return true;

            return (card.getValue() == (this.getTop().getValue() - 10));
        }else{
            if(card.isSmallerThan(this.getTop()))
                return true;

            return (card.getValue() == (this.getTop().getValue() + 10));
        }

    }
```

```java
    /**
     * Vérifie si une carte énemie peut être ajoutée à la pile
     * @param card Carte à vérifier
     * @return True : On peut l'ajouter, False sinon
     */
    public boolean canAddEnemy(Card card){

        return ((this.getPileType().equals(PileType.ASC) && card.getValue() <
this.getCards().get(this.getCards().size()-1).getValue())
                || this.getPileType().equals(PileType.DESC) && card.getValue() >
this.getCards().get(this.getCards().size()-1).getValue());

    }

    /**
     * Ajoute une carte à la pile
     * @param card Carte à ajouter (La carte doit pouvoir être ajoutée à la pile)
     * @see Pile#canAdd(Card)
     */
    public void add(Card card){
        if(this.canAdd(card))
            this.cards.add(card.copy());
    }

    /**
     * Ajoute une carte à la pile
     * @param card Carte à ajouter (La carte doit pouvoir être ajoutée à la pile)
     * @see Pile#canAdd(Card)
     */
    public void addEnemy(Card card){
        if(this.canAddEnemy(card))
            this.cards.add(card);
    }

    /**
     * Affiche la pile selon le standard du projet
     */
    @Override
    public String toString(){

        StringBuilder stringBuilder = new StringBuilder();

        if(this.getPileType().equals(PileType.ASC))
            stringBuilder.append("^");
        else
            stringBuilder.append("v");

        stringBuilder.append("[");
        stringBuilder.append(this.cards.get(this.cards.size()-1).toString());
        stringBuilder.append("]");

        return stringBuilder.toString();

    }
```

```
 // Getters and Setter ====================
    public PileType getPileType(){this.pileType; }

    public ArrayList<Card> getCards(){this.cards;}

    public Card getTop(){
        if(this.cards.size() == 0)
            return null;
        return this.cards.get(this.cards.size() - 1);
    }
}
```

--Pioche.java--

```
package appli;

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

public class Pioche {

    private ArrayList<Card> cardList;

    public Pioche(){
        this.cardList = new ArrayList<>();

        for(int i = 2; i <= 59; i++)
            this.cardList.add(new Card(i));
    }

    /**
     * Prend la carte en haut de la pioche
     * @pre La pioche ne doit pas être vide
     */
    public Card drawCard() throws Exception{

        Random random = new Random();

        if(!this.isEmpty()){
            int index = random.nextInt(this.cardList.size());

            Card card = this.cardList.get(index);
            this.cardList.remove(index);

            return card;
        }else
            throw new Exception("La pioche est vide !");
    }

    /**
     * Permet de savoir si la pioche est vide ou non
     * @return True : La pioche est vide, False : elle contient encore des cartes
     */
    public boolean isEmpty() {
        return this.cardList.isEmpty();
    }

    /**
     * Retourne la taille de la pioche
```

```
     * @return Taille de la pioche
     */
    public int size(){ return this.cardList.size(); }

    // Getters and Setter ==========----------

    /**
     * Permet d'obtenir la liste des cartes de la pioche
     * @return Liste des cartes
     */
    public ArrayList<Card> getCardList() {return this.cardList;}

}
```

--Action.java--

```
package appli;

import appli.types.ActionType;
import appli.types.PileType;

public class Action {

    private ActionType actionType;
    private Card card;

    /**
     * Une action est la description d'un mouvement effectué par un joueur
     * Cette action peut ou pas être valide
     *
     * Une action valide indique qu'avec la configuration donnée, cette carte peut être
posée
     * Si plusieurs cartes sont valides, l'ordre reste a valider par l'ActionBuffer
     * @param type Type d'action
     * @param card Carte utilisé dans l'action
     */
    public Action(ActionType type, Card card){
        this.actionType = type;

        if(card != null)
            this.card = new Card(card);
    }

    /**
     * Constructeur par copie
     */
    public Action(Action action){
        this(action.getType(), action.getCard());
    }

    /**
     * Créé une action à partir d'une chaîne de caractères (Parse string)
     * @param action Action demandé à faire par l'utilisateur
     * @throws IllegalArgumentException Erreur si la chaîne de caractère est incorrecte
     */
    public Action(String action) throws IllegalArgumentException {

        // On vient d'abord vérifier que la taille est correcte
```

```
        if (action.length() < 3 || action.length() > 4)
            throw new IllegalArgumentException("Taille de la chaîne invalide.");

        // On vérifie que si il y a un 4e caractère, il s'agît de '
        if (action.length() == 4 && action.charAt(3) != '\'')
            throw new IllegalArgumentException("Le dernier caractère est incorrect.");

        // On vient maintenant récupérer le nombre contenu dans la première partie de
l'action
        int cardValue = 0;
        try {
            cardValue = Integer.parseInt(action.substring(0, 2));
        } catch (Exception e) {
            throw new IllegalArgumentException("La valeur de la carte doit être de format
\"00\".");
        }

        // On récupère le type d'action
        ActionType type;

        // On vérifie que le caractère désignant la pile est correct
        if (action.charAt(2) == '^') {
            if (action.length() == 4)
                type = ActionType.ADD_ASC_ENEMY;
            else
                type = ActionType.ADD_ASC;
        } else if (action.charAt(2) == 'v') {
            if (action.length() == 4)
                type = ActionType.ADD_DESC_ENEMY;
            else
                type = ActionType.ADD_DESC;
        } else
            throw new IllegalArgumentException("Aucune pile n'est sélectionnée.");

        // Si tout est correct on construit l'action
        this.actionType = type;
        this.card = new Card(cardValue);

    }

    /**
     * Vérifie si une action vise ou non la pile du joueur énemie
     * @return True si l'action est envers un joueur énemie, False sinon
     */
    public boolean isEnemyTarget(){

        if(this.actionType != null)
            return (this.actionType.equals(ActionType.ADD_ASC_ENEMY) ||
this.actionType.equals(ActionType.ADD_DESC_ENEMY));

        return false;

    }
```

```java
    /**
     * Vérifie qu'une action est valide sans prendre en compte un ActionBuffer
     * @param player Le joueur qui exécute l'action
     */
    public boolean isValid(Player player, Player enemy){

        // On vérifie que le type d'action est correct
        if(this.getType() == null || this.getCard() == null)
            return false;

        // On vérifie que le joueur possède la carte
        if(!player.hasCardInHand(this.getCard()))
            return false;

        // Si l'action est vers l'ennemi, on vérifie que la carte peut être posée
        if(this.isEnemyTarget())
            return enemy.getPile(this.getTargetPile()).canAddEnemy(this.getCard());

        // Sinon
        return player.getPile(this.getTargetPile()).canAdd(this.getCard());

    }

    public boolean isValid(Player player, Player enemy, ActionBuffer actionBuffer){

        // Si on vérifie la validité de l'action sans actionBuffer
        // ou si l'actionBuffer est vide, cela revient à faire sans
        // ou si l'actionBuffer ne contient qu'une action vers un ennemi et que l'action
n'est pas vers un ennemi
        if(actionBuffer == null
                || actionBuffer.getActions().size() == 0
                || (actionBuffer.getActions().size() == 1 &&
actionBuffer.getActions().get(0).isEnemyTarget()) && !this.isEnemyTarget())
            return this.isValid(player, enemy);

        if(this.getType() == null || this.getCard() == null)
            return false;

        if(!player.hasCardInHand(this.getCard()))
            return false;

        // Si l'action est vers l'ennemi, on vérifie que la carte peut être posée
        if(this.isEnemyTarget() &&
!enemy.getPile(this.getTargetPile()).canAddEnemy(this.getCard()))
            return false;

        // Vérification concernant l'action buffer ======================

        // On vérifie si le joueur n'a pas déjà utilisé la carte
        if(actionBuffer.useCard(this.getCard()))
            return false;

        // Si il s'agit d'une action sur sa pile, on vérifie qu'on peut l'ajouter sur la
pile temporaire
        if(this.getType().equals(ActionType.ADD_ASC) &&
actionBuffer.getTmpAsc().canAdd(this.getCard()))
            return true;
```

```
if(this.getType().equals(ActionType.ADD_DESC) &&
actionBuffer.getTmpDesc().canAdd(this.getCard()))
            return true;

        // Si il s'agit d'une action sur une pile adverse, on vérifie qu'il n'existe pas
déjà une action vers l'ennemi
        if(this.isEnemyTarget()) {
            for (Action a : actionBuffer.getActions())
                if (a.isEnemyTarget())
                    return false;

            return true;
        }

        return false;

    }

    // Getters & Setters ==========----------

    /**
     * Vérifie si une action vise ou non la pile du joueur énemie
     * @return True si l'action est envers un joueur énemie, False sinon
     */
    public PileType getTargetPile(){

        if(this.actionType == null)
            return null;

        if(this.actionType.equals(ActionType.ADD_ASC) ||
this.actionType.equals(ActionType.ADD_ASC_ENEMY))
            return PileType.ASC;
        else
            return PileType.DESC;

    }

    public ActionType getType() { return actionType; }

    public void setType(ActionType actionType) {

        if(actionType == null)
            throw new IllegalArgumentException("Le type d'action ne peut pas être nul");

        this.actionType = actionType;
    }

    public Card getCard() { return card; }

    public void setCard(Card card) {
        this.card = card;
    }
}
```

```java
package appli;

import appli.types.ActionType;
import appli.types.PileType;

import java.util.ArrayList;

public class ActionBuffer {

    private Player player;
    private Player enemy;

    private ArrayList<Action> actions;

    private Pile tmpAsc;
    private Pile tmpDesc;

    /**
     * Un ActionBuffer est une liste d'actions VALIDES en attente d'exécutions
     *
     * Son rôle et d'assurer :
     * - la non redondance des actions (utilisation de d'une même carte à plusieurs reprises)
     * - d'exécuter les actions dans le bon ordre (et donc la validité des entrées des joueurs)
     *
     * @param player Joueur courant
     * @param enemy ennemi du joueur courant
     */
    public ActionBuffer(Player player, Player enemy){

        this.actions = new ArrayList<>();
        this.player = player;
        this.enemy = enemy;

        this.tmpAsc = new Pile(PileType.ASC);
        this.tmpDesc = new Pile(PileType.DESC);

    }

    /**
     * Constructeur par copie
     */
    public ActionBuffer(ActionBuffer buffer){

        this.actions = new ArrayList<>(buffer.getActions());
        this.player = buffer.player;
        this.enemy = buffer.enemy;

        this.tmpAsc = new Pile(buffer.getTmpAsc());
        this.tmpDesc = new Pile(buffer.getTmpDesc());

    }
```

```java
/**
 * Ajoute une action au buffer
 * @param action Action à ajouter
 */
public void add(Action action) throws IllegalArgumentException{

    if(this.canAdd(action)){

        Action actionCopy = new Action(action);

        this.actions.add(actionCopy);

        if(actionCopy.getType().equals(ActionType.ADD_ASC))
            tmpAsc.add(actionCopy.getCard());
        else if(actionCopy.getType().equals(ActionType.ADD_DESC))
            tmpDesc.add(actionCopy.getCard());

    }else
        throw new IllegalArgumentException("L'action n'est pas valide");

}

/**
 * Vérifie si une action peut être ajoutée au buffer
 * @param action Action que l'on veut vérifier
 */
public boolean canAdd(Action action){

    return action.isValid(this.player, this.enemy, this);

}

/**
 * Efface les actions contenues dans le buffer
 */
public void clear(){
    this.actions = new ArrayList<>();
    this.tmpAsc = new Pile(PileType.ASC);
    this.tmpDesc = new Pile(PileType.DESC);
}

/**
 * Vérifie si l'action buffer contient une action envers le joueur énemi
 * @return True si elle contient, False sinon
 */
public boolean containsEnemyAction(){

    for(Action action : this.getActions()){

        if(action.isEnemyTarget())
            return true;

    }

    return false;

}
```

```java
    /**
     * Vérifie si une action dans le buffer utilise une carte
     * @param card Carte à vérifier
     * @return True : La carte est utilisée, False sinon
     */
    public boolean useCard(Card card){

        for(Action a : this.getActions()) {
            if (a.getCard().equals(card))
                return true;
        }

        return false;

    }

    @Override
    public String toString(){

        StringBuilder str = new StringBuilder("=>");

        for(Action a : this.getActions()){

            str.append(" ");
            str.append(a.getCard().toString());

            switch (a.getType()) {
                case ADD_ASC:
                    str.append("^");
                    break;
                case ADD_DESC:
                    str.append("v");
                    break;
                case ADD_ASC_ENEMY:
                    str.append("^'");
                    break;
                case ADD_DESC_ENEMY:
                    str.append("v'");
                    break;
            }

        }

        return str.toString();

    }

    // Getters & Setter ==========----------
    public ArrayList<Action> getActions(){ return this.actions; }

    public Pile getTmpAsc(){ return tmpAsc; }

    public Pile getTmpDesc(){ return tmpDesc; }

    public Player getEnemy(){ return enemy; }

}
```

--Player.java--

```java
package appli;

import appli.types.ActionType;
import appli.types.PileType;
import appli.types.PlayerType;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.Scanner;

public class Player {

    private PlayerType playerType;
    private Pioche pioche;
    private Pile pileAsc;
    private Pile pileDesc;
    private ArrayList<Card> hand;

    public Player(PlayerType playerType){
        this.playerType = playerType;
        this.pioche = new Pioche();
        this.pileAsc = new Pile(PileType.ASC);
        this.pileDesc = new Pile(PileType.DESC);
        this.hand = new ArrayList<>();
    }

    public Player(){
        this.pioche = new Pioche();
        this.pileAsc = new Pile(PileType.ASC);
        this.pileDesc = new Pile(PileType.DESC);
        this.hand = new ArrayList<>();
    }

    /**
     * Vérifie si un joueur peut jouer (i.e. poser au moins 2 cartes)
     * @return True : Le joueur peut jouer, False : Le joueur ne peut pas jouer
     */
    public boolean canPlay(Player enemy) {

        return (this.getActionsUsingCards(2, enemy).size() > 0);

    }
```

```java
/**
 * Retourne, pour un joueur et une liste préalable d'actions
 * les coups faisables avec les cartes restantes
 * @param actionBuffer Buffer des actions effectuées
 * @return Liste des coups possibles
 */
public ArrayList<ActionBuffer> getPossibleActions(ActionBuffer actionBuffer){

    ArrayList<ActionBuffer> buffers = new ArrayList<>();
    for(Card card : this.getHand()){

        if(!actionBuffer.useCard(card)){
            for(ActionType type : ActionType.values()){

                Action nextAction = new Action(type, card);

                if(actionBuffer.canAdd(nextAction)){
                    ActionBuffer newBuffer = new ActionBuffer(actionBuffer);
                    newBuffer.add(nextAction);
                    buffers.add(newBuffer);
                }
            }
        }
    }
    return buffers;
}

/**
 * Retourne une liste de combinaisons d'actions permettant
 * d'utiliser un certain nombre de cartes de la main du
 * (Cette fonction permet de déterminer si un joueur peu jouer ou non)
 * @param cardNumber Nombre de carte devant être utilisées
 * @see Player#getPossibleActions(ActionBuffer)
 */
public ArrayList<ActionBuffer> getActionsUsingCards(int cardNumber, Player enemy){

    ArrayList<ActionBuffer> bufferList = new ArrayList<>();
    bufferList.add(new ActionBuffer(this, enemy));

    for(int i = 0; i < cardNumber; i++){

        ArrayList<ActionBuffer> tmpList = new ArrayList<>();

        for(ActionBuffer buffer : bufferList){

            tmpList.addAll(this.getPossibleActions(buffer));

        }

        bufferList = new ArrayList<>(tmpList);

    }

    return bufferList;

}
```

```java
/**
     * Gère les coups du joueur
     * @param enemy Joueur ennemi
     */
    public void playTurn(Player enemy){

        boolean correctActions = false;
        boolean error = false;

        ActionBuffer actionBuffer = new ActionBuffer(this, enemy);
        while(!correctActions){

            ArrayList<String> actionStrings = this.typeActions(error);
            actionBuffer.clear();
            error = false;

            if(actionStrings.isEmpty() || actionStrings.size() < 2){
                error = true;
                continue;
            }

            for(String actionString : actionStrings){

                try {

                    Action action = new Action(actionString);

                    if(actionBuffer.canAdd(action)) {
                        actionBuffer.add(action);
                    }else {
                        error = true;
                        break;
                    }

                }catch (IllegalArgumentException e){
                    error = true;
                    break;
                }

            }

            if(!error)
                correctActions = true;

        }

        try{

            this.executeActionBuffer(actionBuffer);
            int drawnCards = this.drawEndRoundCards(actionBuffer);

            System.out.println(actionBuffer.getActions().size() + " cartes posées, " +
drawnCards + " cartes piochées");

        }catch (Exception e){
            e.printStackTrace();
        }

    }
```

```java
    /**
     * Demande les coups que veut jouer le joueur
     */
    public ArrayList<String> typeActions(boolean previousError){

        if(previousError)
            System.out.print("#");

        System.out.print("> ");
        ArrayList<String> actionStrings = new ArrayList<>();
        Scanner scanner1 = new Scanner(System.in);

        if(scanner1.hasNextLine()){
            Scanner scanner2 = new Scanner(scanner1.nextLine());
            while(scanner2.hasNext())
                actionStrings.add(scanner2.next());
        }
        return actionStrings;
    }

    // Draw Card ==========----------
    /**
     * Fait piocher un certain nombre de carte au joueur et trie son paquet
     * @param n Le nombre de cartes à piocher (Si > au nbr de cartes dans la pioche,
renvoie une exception)
     */
    public void drawCard(int n) throws Exception{

        if(n > this.getPioche().size())
            throw new Exception("Il n'y a pas assez de cartes dans la pioche !");
        else{
            for(int i = 0; i < n; i++)
                this.addCardInHand(this.getPioche().drawCard());
        }

    }

    /**
     * Pioche le nombre de carte nécessaire
     * @param actionBuffer Action buffer contenant les actions exécutées
     */
    public int drawEndRoundCards(ActionBuffer actionBuffer) throws Exception {

        int drawnCards = 2;

        if(actionBuffer.containsEnemyAction())
            drawnCards = 6 - this.getHand().size();

        if(drawnCards > this.getPioche().size())
            drawnCards = this.getPioche().size();

        if(drawnCards > 0)
            this.drawCard(drawnCards);

        return drawnCards;
    }
```

```java
// Actions ==========----------

    /**
     * Execute l'action
     * @param action Action a exécuter
     * @param enemy Joueur énemi
     * @throws Exception L'action n'a pas pu s'exécuter correctement
     */
    public void executeAction(Action action, Player enemy) throws Exception{
        this.removeCardFromHand(action.getCard());

        if(action.isEnemyTarget())
            enemy.getPile(action.getTargetPile()).addEnemy(action.getCard());
        else
            this.getPile(action.getTargetPile()).add(action.getCard());
    }

    /**
     * Execute l'action
     * @param buffer Ensemble des actions à exécuter
     * @throws Exception L'action n'a pas pu s'exécuter correctement
     */
    public void executeActionBuffer(ActionBuffer buffer) throws Exception{
        for(Action action : buffer.getActions())
            this.executeAction(action, buffer.getEnemy());
    }

    // Hand ==========----------

    /**
     * Ajoute une carte dans la main du joueur & ordonne la liste de cartes
     * @param card Carte à ajouter
     */
    public void addCardInHand(Card card){
        if(!this.hasCardInHand(card)) {
            this.hand.add(card);
            this.sortCardsInHand();
        }
    }

    /**
     * Enlève une carte dans la main du joueur
     * @param card Carte à enlever
     *             La carte doit être présente dans la main du joueur
     */
    public void removeCardFromHand(Card card) throws Exception{

        if(!this.hasCardInHand(card))
            throw new Exception("Le joueur ne possède pas cette carte.");
        else{
            this.hand.remove(card);
            for(int i = 0; i < this.hand.size(); i++){
                if(this.hand.get(i).equals(card)) {
                    this.hand.remove(i);
                    break;
                }
            }
        }
    }
```

```java
    /**
     * Vérifie si un joueur possède une carte dans sa main
     * @param card Carte à vérifier
     * @return True : il possède cette carte dans sa main, False sinon
     */
    public boolean hasCardInHand(Card card){
        for(Card c : this.getHand())
            if(c.equals(card))
                return true;

        return false;
    }

    /**
     * Tri les cartes en main dans l'ordre croissant
     */
    public void sortCardsInHand(){
        this.hand.sort(new Comparator<Card>() {
            @Override
            public int compare(Card o1, Card o2) {
                return o1.getValue() - o2.getValue();
            }
        });
    }

    // Display ==========----------
    /**
     * Affiche les cartes en main
     */
    public String handToString(){

        StringBuilder sb = new StringBuilder("cartes ");
        sb.append(this.getType());
        sb.append(" { ");

        for(Card card : this.hand) {
            sb.append(card.toString());
            sb.append(" ");
        }

        sb.append("}");

        return sb.toString();
    }

    /**
     * Affiche les piles, le nombre de cartes en main et le nombre de cartes dans la
pioche
     */
    @Override
    public String toString(){
        StringBuilder sb = new StringBuilder();
        sb.append(this.getDisplayName());
        sb.append(" ");
        sb.append(this.getPileAsc());
        sb.append(" ");
        sb.append(this.getPileDesc());
        sb.append(" ");
```

```
        sb.append("(m");
        sb.append(this.getHand().size());
        sb.append("p");
        sb.append(this.getPioche().size());
        sb.append(")");

        return sb.toString();
    }

    /**
     * Retourne le nom du joueur avec des espaces pour conserver l'alignement dans
l'affichage
     * @return Nom d'affichage
     */
    private String getDisplayName() {

        // On récupère la taille du nom le plus grand (permet d'adapter les noms
facilement)
        int size = 0;
        for (PlayerType type : PlayerType.values()){
            if (type.toString().length() > size)
                size = type.toString().length();
        }

        if(size == this.getType().toString().length())
            return this.getType().toString();

        StringBuilder sb = new StringBuilder(this.getType().toString());
        for(int i = 0; i < size-this.getType().toString().length(); i++)
            sb.append(" ");

        return sb.toString();
    }

    // Getters and Setters ==========----------

    public PlayerType getType() { return playerType; }
    public Pioche getPioche() { return pioche; }
    public Pile getPileAsc() { return pileAsc; }
    public Pile getPileDesc() { return pileDesc; }

    /**
     * Retourne la pile en fonction du type de pile
     * @param type Type de la pile
     */
    public Pile getPile(PileType type){
        if(type.equals(PileType.ASC))
            return this.getPileAsc();
        else if(type.equals(PileType.DESC))
            return this.getPileDesc();

        return null;
    }

    public ArrayList<Card> getHand() { return hand; }

}
```

--Game.java--

```java
package appli;

import appli.types.PlayerType;

public class Game {

    private Player[] players; // Liste de taille 2 : [NORD,SUD]
    private Player currentPlayer;
    boolean finished = false;

    /**
     * Constructeur de Game
     * @param player1 Joueur 1
     * @param player2 Joueur 2
     * Les deux joueurs doivent avoir un type différent (NORD et SUD)
     */
    public Game(Player player1, Player player2){

        this.players = new Player[2];

        if(player1.getType() == null || player2.getType() == null)
            throw new IllegalArgumentException("Les joueurs doivent avoir un type
défini");
        else if(player1.getType().equals(player2.getType()))
            throw new IllegalArgumentException("Les joueurs doivent être d'un type
différent l'un de l'autre");

        if(player1.getType().equals(PlayerType.NORD)){
            this.players[0] = player1;
            this.players[1] = player2;
        }else{
            this.players[0] = player2;
            this.players[1] = player1;
        }

        for(Player p : this.players){
            try {
                p.drawCard(6);
            }catch (Exception e){
                e.printStackTrace();
            }
        }

        this.currentPlayer = players[0];

    }
```

```java
/**
 * Lance la boucle de jeu
 */
public void play(){
    try {
        while(!isFinished()){

            for(Player p : players)
                System.out.println(p.toString());

            System.out.println(currentPlayer.handToString());
            currentPlayer.playTurn(getEnemy(currentPlayer));
            nextPlayer(); // On passe au joueur suivant
        }

        for(Player p : players)
            System.out.println(p.toString());

        System.out.println(currentPlayer.handToString());
        nextPlayer(); // On place le gagnant dans la position courante

    }catch (Exception exception) {
        exception.printStackTrace();
    }
}

/**
 * Change le joueur courant
 */
public void nextPlayer() throws Exception{
    this.currentPlayer = this.getEnemy(this.currentPlayer);
}
/**
 * Vérifie si la partie est terminée ou non
 * @return True si la partie est terminée, False sinon
 */
public boolean isFinished() throws Exception{
    return !currentPlayer.canPlay(getEnemy(currentPlayer));
}

/**
 * Retourne le joueur adverse
 * @param p Joueur dont on veut obtenir l'énemi
 * @return Joueur adverse
 */
private Player getEnemy(Player p) throws Exception{
    if(p == null)
        throw new IllegalArgumentException("Le joueur doit être différent de null");

    if(this.players.length != 2)
        throw new Exception("La partie ne contient pas deux joueurs");

    if(this.players[0] == p)
        return this.players[1];

    return this.players[0];

}
```

```java
    /**
     * Retourne le gagnant de la partie
     * @return Gagnant de la partie
     */
    public Player getWinner(){ return currentPlayer; }


}
```

--Application.java--

```java
package appli;

import appli.types.PlayerType;

public class Application {

    public static void main(String[] args){

        Game game = new Game(new Player(PlayerType.NORD), new Player(PlayerType.SUD));

        game.play();

        try {
            Player winner = game.getWinner();
            System.out.println("partie finie, " + winner.getType() + " a gagné");
        }catch (Exception e){
            e.printStackTrace();
        }

    }

}
```