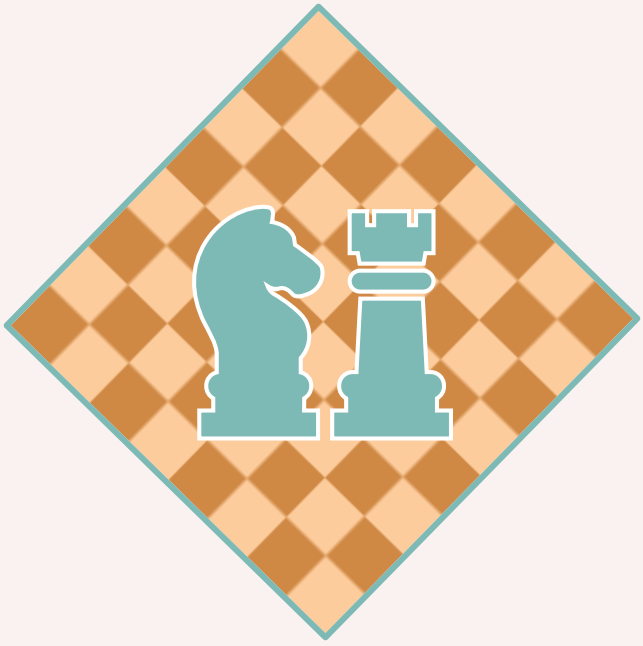
Jeux d'échecs

Projet Java



Zakaria Sellam
Groupe 111

Li Yannick
Groupe 108

Stefan Radovanovic
Groupe 107

Université de Paris

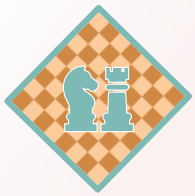


Table des matières

Université de Paris

Introduction

Le projet a été codé par nos soins, nous avons utilisé certaines ressources trouvées sur internet cités dans la documentation.

Le programme prend en charge :

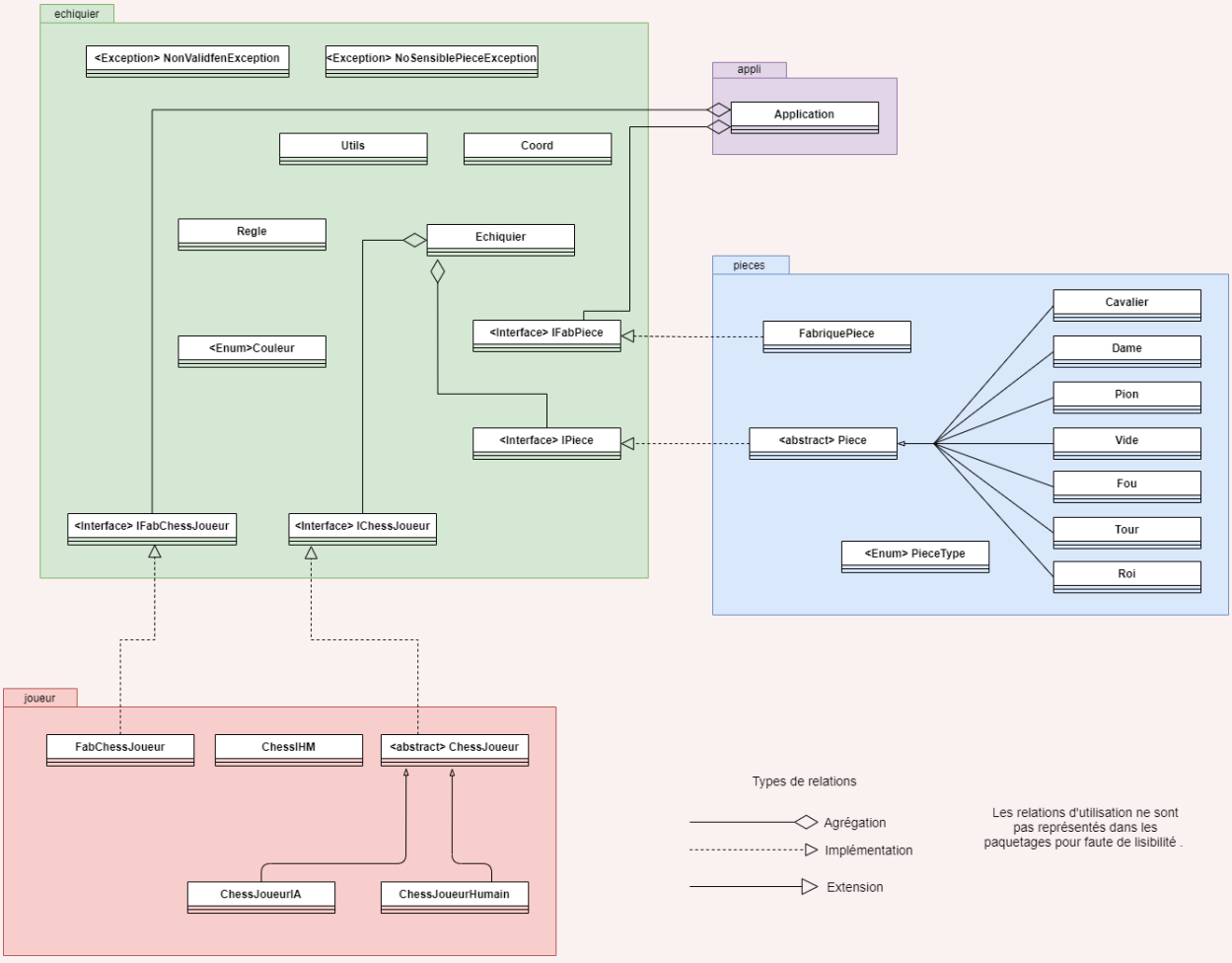
- le déplacement légal des pièces
- la capture des pièces
- Jouer une partie
- Le choix du type de partie (Joueur vs Joueur, Joueur vs IA, IA vs IA)
- Échecs et mat, le pat, certaines nulles (manque de matériel)
- la prise en diagonale et la promotion des pions en dame
- la proposition de nulles

Le programme ne prend pas en charge :

- le rock
- la prise en passant
- La règle des 50 coups
- Nulle par répétition

Répertoire Git du Projet : Yan2708/Chess (github.com)

Diagramme d'architecture



Tests unitaires

Échiquier – 6 à 16

Coord - 6 à 7

Echiquier - 8 à 10

Regle - 11 à 13

Utils - 14 à 16

Pièces – 17 à 28

Cavalier - 17 à 18

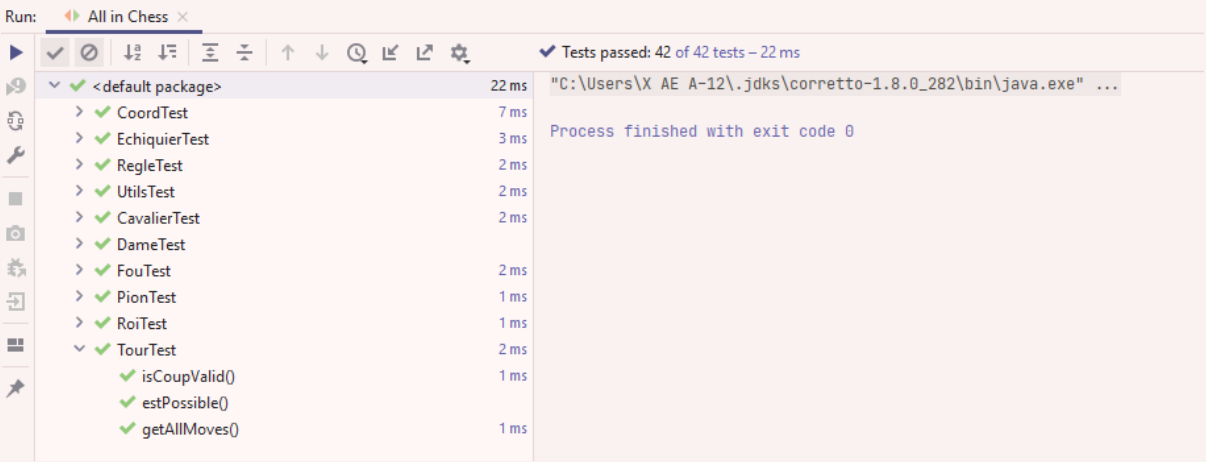
Dame - 19 à 20

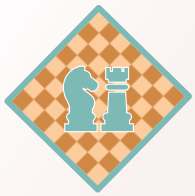
Fou - 21 à 22

Pion - 23 à 24

Roi - 25 à 26

Tour - 27 à 28

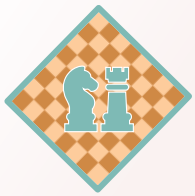




Échiquier

Coord 1/2

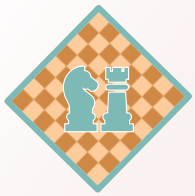
```java
package echiquier;

import echiquier.Coord;
import org.junit.jupiter.api.Test;

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

class CoordTest {

    @Test
    void testAdd(){
        Coord c1 = new Coord(1,1);
        Coord c2 = new Coord(5,-9);
        Coord c3 = new Coord(6,-8);
        c1.add(c2);
        assertEquals(c3,c1);
    }

    @Test
    void testPrimarymove(){
        Coord c1 = new Coord(1,1);
        Coord c2 = new Coord(0,1);
        Coord c3 = new Coord(-1,0);
        assertEquals(c3, Coord.getPrimaryMove(c1,c2));
    }

    @Test
    void testIsStraightPath(){
        Coord c1 = new Coord(1,1);
        Coord c2 = new Coord(0,0);
        Coord c3 = new Coord(2,3);
        Coord c4 = new Coord(5,0);
        assertTrue(Coord.isStraightPath(c2,c1));
        assertTrue(Coord.isStraightPath(c2,c4));
        assertFalse(Coord.isStraightPath(c2,c3));
        assertFalse(Coord.isStraightPath(c3,c1));
        assertTrue(Coord.isStraightPath(c3,c4));
    }
```



Échiquier

Coord 2/2

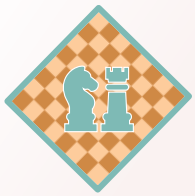
```
    @Test
    void cloneTest(){
        Coord c1 = new Coord(1,1);
        Coord c2 = c1.clone();
        Coord c3 = new Coord(1,1);
        assertEquals(c1,c2);
        assertEquals(c1,c3);
    }

    @Test
    void testToString(){
        Coord c =  new Coord("b7");
        Coord c2 = new Coord(2,0);
        assertEquals(c.toString(),"b7");
        assertEquals(c2.toString(),"a6");
        c.add(c2);
        assertEquals(c.toString(),"b5");
    }

}
```



Échiquier

Echiquier 1/3

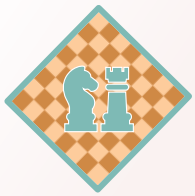
```
package echiquier;

import Joueur.FabChessJoueur;
import org.junit.jupiter.api.Test;
import pieces.FabPiece;

import java.util.LinkedList;
import java.util.List;

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

class EchiquierTest {

    @Test
    void allClassicMoves() {
        Echiquier e = new Echiquier("r7/8/8/3T3P/8/8/8/7R",new FabPiece(), new FabChessJoueur());

        String[] coupPossible = {"d8","d7","d6","a5","b5","c5","e5","f5","g5","d4","d3","d2","d1"};
        LinkedList<Coord> l = e.allClassicMoves(e.getPiece(new Coord("d5")));
        for (String coup : coupPossible)
            assertTrue(l.contains(new Coord(coup)));
        assertFalse(l.contains("h5"));

        String[] coupPossibleRoi = {"a7","b8","b7"};
        l = e.allClassicMoves(e.getPiece(new Coord("a8")));
        assertEquals(l.size(), 3);
        for (String coup : coupPossibleRoi)
            assertTrue(l.contains(new Coord(coup)));
    }

    @Test
    void deplacer() {
        Echiquier e = new Echiquier("r7/8/8/3T3P/8/8/8/7R",new FabPiece(), new FabChessJoueur());
        e.deplacer("a8h4");
        assertEquals("r", e.getPiece(new Coord("h4")).dessiner());
        assertEquals(" ", e.getPiece(new Coord("a8")).dessiner());
    }
```



Échiquier

Echiquier 2/3

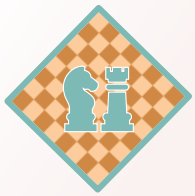
```
@Test
void getPieceFromColor() {
    Echiquier e = new Echiquier("r7/8/8/3T3P/PPPPPPpp/8/8/7R",new FabPiece(), new FabChessJoueur());
    List<IPiece> blanc = e.getPieceFromColor(BLANC);
    List<IPiece> noir = e.getPieceFromColor(NOIR);
    assertEquals(blanc.size(),9);
    assertEquals(noir.size(),3);
    for (IPiece p : blanc) {
        assertEquals(p.getCouleur(), BLANC);
    }
    for (IPiece p : noir) {
        assertEquals(p.getCouleur(), NOIR);
    }
}

@Test
void locateSensiblePiece() {
    Echiquier e = new Echiquier("r7/8/8/3T3P/PPPPPPpp/3R4/8/8",new FabPiece(), new FabChessJoueur());
    assertEquals(e.locateSensiblePiece(BLANC),new Coord("d3"));
    assertEquals(e.locateSensiblePiece(NOIR),new Coord("a8"));
}

@Test
void checkForPromote() {
    Echiquier e = new Echiquier("r6P/PP6/8/3T4/8/3R4/PPPp4/4p3",new FabPiece(), new FabChessJoueur());
    e.checkForPromote(BLANC);
    assertEquals("D", e.getPiece(new Coord("h8")).dessiner());
    assertEquals("P", e.getPiece(new Coord("a7")).dessiner());
    e.checkForPromote(NOIR);
    assertEquals("d", e.getPiece(new Coord("e1")).dessiner());
    assertEquals("p", e.getPiece(new Coord("d2")).dessiner());
    assertEquals("P", e.getPiece(new Coord("c2")).dessiner());
}
```



Échiquier

Echiquier 3/3

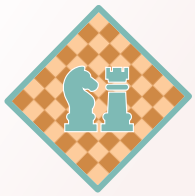
```
@Test
    void testToString() {
        Echiquier e = new Echiquier(new FabPiece(), new FabChessJoueur());
        String board =
                "    a  b  c  d  e  f  g  h    \n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                " 8 | t | c | f | d | r | f | c | t | 8\n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                " 7 | p | p | p | p | p | p | p | p | 7\n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                " 6 |   |   |   |   |   |   |   | 6\n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                " 5 |   |   |   |   |   |   |   | 5\n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                " 4 |   |   |   |   |   |   |   | 4\n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                " 3 |   |   |   |   |   |   |   | 3\n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                " 2 | P | P | P | P | P | P | P | P | 2\n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                " 1 | T | C | F | D | R | F | C | T | 1\n" +
                "   --- --- --- --- --- --- --- ---   \n" +
                "    a  b  c  d  e  f  g  h    \n";
        assertEquals(e.toString(), board);
    }
}
```



Échiquier

Regle 1/3

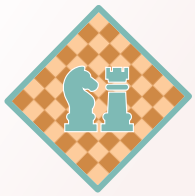
```java
package echiquier;

import Joueur.FabChessJoueur;
import org.junit.jupiter.api.Test;
import pieces.FabPiece;

import java.util.List;

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

class RegleTest {

    @Test
    void checkForMate() {
        // https://en.wikipedia.org/wiki/Checkmate_pattern
        // https://lichess.org/editor
        String[] fens = {"8/4C1pr/8/7T/8/8/8/7R",   //Anastasia's mate
                "6rT/6P1/5R2/8/8/8/8/8",            //Anderssen's mate
                "7r/7T/5C2/8/8/8/8/7R",             //Arabian mate
                "2T3r1/5ppp/8/8/8/8/8/7R",          //Back-rank mate
                "5tr1/6pD/6P1/8/8/8/8/7R",          //Damiano's mate
                "7r/6p1/8/3F3D/8/8/8/7R"};          //Greco's mate
        FabChessJoueur fj = new FabChessJoueur();
        FabPiece fp = new FabPiece();
        for (String fen : fens) {
            Echiquier e = new Echiquier(fen, fp, fj);
            Coord sC = e.locateSensiblePiece(Couleur.NOIR);
            List<IPiece> allys = e.getPieceFromColor(Couleur.NOIR);
            List<IPiece> ennemies = e.getPieceFromColor(Couleur.BLANC);
            assertTrue(Regle.checkForMate(sC, allys, ennemies, e));
        }
    }
```



Échiquier

Regle 2/3

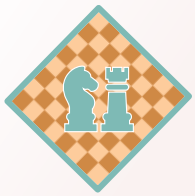
```
@Test
void isStaleMate() {
    // https://chess24.com/en/read/news/7-stalemates-every-chess-player-needs-to-know
    String[] fens = {"r7/P7/1R6/8/8/8/8/8",   //King + rook pawn vs. king (variation)
            "7r/8/6RP/3F4/8/8/8/8",           //The wrong bishop
            "5r2/8/4D2P/8/8/8/8/1R6"};        //Queens StaleMate

    FabChessJoueur fj = new FabChessJoueur();
    FabPiece fp = new FabPiece();
    for (String fen : fens) {
        Echiquier e = new Echiquier(fen, fp, fj);
        Coord sC = e.locateSensiblePiece(Couleur.NOIR);
        List<IPiece> allys = e.getPieceFromColor(Couleur.NOIR);
        List<IPiece> ennemies = e.getPieceFromColor(Couleur.BLANC);
        assertTrue(Regle.isStaleMate(sC, allys, ennemies, e));
    }
}

@Test
void isAttacked() {
    FabChessJoueur fj = new FabChessJoueur();
    FabPiece fp = new FabPiece();
    Echiquier e = new Echiquier("7r/8/8/3P4/8/8/8/R6D", fp, fj);
    List<IPiece> ennemies = e.getPieceFromColor(Couleur.BLANC);
    String[] attaquePion = {"c6", "e6"};
    for (String coup : attaquePion)
        assertTrue(Regle.isAttacked(new Coord(coup), ennemies, e));
    Coord coordRoi = e.locateSensiblePiece(Couleur.NOIR);
    assertTrue(Regle.isAttacked(coordRoi, ennemies, e));
}
```



Échiquier

Regle 2/3

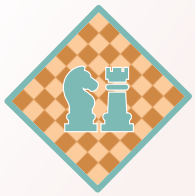
```
@Test
void impossibleMat() {
    String[] fens = {"7r/1f6/8/8/8/7F/8/4R3",   //roi + fou vs roi + fou
            "7r/8/8/8/1c6/7C/8/4R3",            //roi + cavalier vs roi + cavalier
            "8/5r2/8/8/2R5/8/8/8"};         //roi vs roi

    FabChessJoueur fj = new FabChessJoueur();
    FabPiece fp = new FabPiece();
    for (String fen : fens) {
        Echiquier e = new Echiquier(fen, fp, fj);
        List<IPiece> allys = e.getPieceFromColor(Couleur.NOIR);
        List<IPiece> ennemies = e.getPieceFromColor(Couleur.BLANC);
        assertTrue(Regle.impossibleMat(allys, ennemies));
    }
}

@Test
    void voieLibre() {
        FabChessJoueur fj = new FabChessJoueur();
        FabPiece fp = new FabPiece();
        Echiquier e = new Echiquier("2r5/8/6p1/2p5/8/8/2D2T1p/7R", fp, fj);
        IPiece dame = e.getPiece(new Coord("c2"));
        assertFalse(Regle.voieLibre(dame, new Coord("h2"), e));
        assertFalse(Regle.voieLibre(dame, new Coord("c8"), e));
        assertTrue(Regle.voieLibre(dame, new Coord("g6"), e));
    }

    @Test
    void isFinishValid() {
        FabChessJoueur fj = new FabChessJoueur();
        FabPiece fp = new FabPiece();
        Echiquier e = new Echiquier("R6r/8/8/2p5/4P3/8/P1D2P2/2t5", fp, fj);
        IPiece dame = e.getPiece(new Coord("c2"));
        assertTrue(Regle.isFinishValid(dame, new Coord("c1"), e));
        assertTrue(Regle.isFinishValid(dame, new Coord("c5"), e));
        assertTrue(Regle.isFinishValid(dame, new Coord("a4"), e));
        assertFalse(Regle.isFinishValid(dame, new Coord("a2"), e));
        assertFalse(Regle.isFinishValid(dame, new Coord("f2"), e));
        assertFalse(Regle.isFinishValid(dame, new Coord("e42"), e));
    }

}
```



Échiquier

Utils 1/3

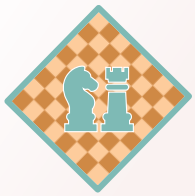
```java
package echiquier;

import Joueur.FabChessJoueur;
import org.junit.jupiter.api.Test;
import pieces.FabPiece;

import java.util.List;

import static echiquier.Couleur.BLANC;
import static echiquier.Couleur.NOIR;
import static org.junit.jupiter.api.Assertions.*;

class UtilsTest {

    @Test
    void getPath() {

        Coord c1 = new Coord("a8");
        Coord c2 = new Coord("a3");
        Coord c3 = new Coord("f3");
        Coord c4 = new Coord("e6");
        String[] ligne = {"a8","a7","a6","a5","a4","a3"};
        String[] diag = {"a8","b7","c6","d5","e4","f3"};

        List<Coord> l = Utils.getPath(c1,c2);
        assertEquals(6,l.size());
        for(String pos : ligne)
            assertTrue(l.contains(new Coord(pos)));

        l = Utils.getPath(c1,c3);
        assertEquals(6,l.size());
        for(String pos : diag)
            assertTrue(l.contains(new Coord(pos)));

        l = Utils.getPath(c1,c4);
        assertEquals(1,l.size());
        assertTrue(l.contains(c4));
    }
```



Échiquier

Utils 2/3

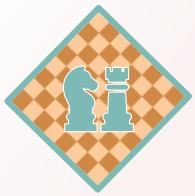
```
@Test
void getAllAttackingPiece() {
    Echiquier e = new Echiquier("r6T/1p6/1PF4p/3T4/8/3R4/8/T7",new FabPiece(), new FabChessJoueur());

    String[] coordAttaquant = { "a1", "h8"};
    List<IPiece> l = Utils.getAllAttackingPiece(e.locateSensiblePiece(NOIR),e.getPieceFromColor(BLANC),e);
    assertEquals(2,l.size());
    for (String coord : coordAttaquant)
        assertTrue(l.contains(e.getPiece(new Coord(coord))));
}

@Test
void allMovesDefendingCheck(){
    FabChessJoueur fj = new FabChessJoueur();
    FabPiece fp = new FabPiece();
    Echiquier e = new Echiquier("3r4/1t6/8/5c2/8/8/8/R2D4", fp, fj);
    Coord sC = e.locateSensiblePiece(NOIR);
    IPiece cavalier = e.getPiece(new Coord("f5"));
    List<IPiece> ennemies = e.getPieceFromColor(BLANC);
    String[] coupCavalier = {"d4","d6"};
    List<Coord> cavalierMoves = Utils.allMovesDefendingCheck(cavalier, sC, ennemies, e);
    assertEquals(2, cavalierMoves.size());
    for(String coup : coupCavalier)
        assertTrue(cavalierMoves.contains(new Coord(coup)));
    IPiece tour = e.getPiece(new Coord("b7"));
    List<Coord> tourMoves = Utils.allMovesDefendingCheck(tour, sC, ennemies, e);
    assertEquals(1, tourMoves.size());
    assertTrue(tourMoves.contains(new Coord("d7")));
}

@Test
void allMovesFromPin() {
    Echiquier e = new Echiquier("r7/1p6/1PF4p/3T4/8/3R4/8/8",new FabPiece(), new FabChessJoueur());
    List<Coord> l = Utils.allMovesFromPin(e.getPiece(new Coord("b7")),e.locateSensiblePiece(NOIR), e);
    assertEquals(1, l.size());
    assertTrue(l.contains(new Coord("c6")));
}
```



Échiquier

Utils 3/3

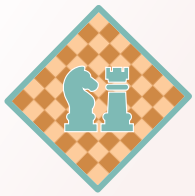
```
    @Test
    void getPningPiece() {
        FabChessJoueur fj = new FabChessJoueur();
        FabPiece fp = new FabPiece();
        Echiquier e = new Echiquier("3r4/2ppp3/8/F5F1/8/8/8/3D3R", fp, fj);
        Coord sC = e.locateSensiblePiece(NOIR);
        IPiece pion1 = e.getPiece(new Coord("c7"));
        assertNotNull(Utils.getPningPiece(pion1.getCoord(), sC, e));
        IPiece pion2 = e.getPiece(new Coord("d7"));
        assertNotNull(Utils.getPningPiece(pion2.getCoord(), sC, e));
        IPiece pion3 = e.getPiece(new Coord("e7"));
        assertNotNull(Utils.getPningPiece(pion3.getCoord(), sC, e));
    }

    @Test
    void getPathToBorder() {
        Coord c1 = new Coord("a4");
        Coord c2 = new Coord(1,0);
        Coord c3 = new Coord(-1,1);
        String[] ligne = { "a4","a3","a2","a1"};
        String[] diag = {"a4","b5","c6","d7","e8"};

        List<Coord> l = Utils.getPathToBorder(c1,c2);
        for (String coord : ligne )
            assertTrue(l.contains(new Coord(coord)));

        l = Utils.getPathToBorder(c1,c3);
        for (String coord : diag )
            assertTrue(l.contains(new Coord(coord)));
    }
}
```



Pièces

Cavalier 1/2

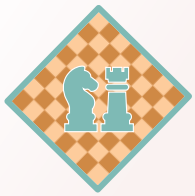
```
package pieces;

import Joueur.FabChessJoueur;
import echiquier.Coord;
import echiquier.Couleur;
import echiquier.Echiquier;
import echiquier.IPiece;
import org.junit.jupiter.api.Test;

import java.util.LinkedList;
import java.util.List;

import static echiquier.Couleur.BLANC;
import static echiquier.Couleur.NOIR;
import static org.junit.jupiter.api.Assertions.*;

class CavalierTest {

    @Test
    void estPossible() {
        Cavalier c = new Cavalier(new Coord("d5"), Couleur.BLANC);

        String[] coordPossible = {"b6","c7","e7","f6","f4","e3","c3","b4"};
        for(String coord : coordPossible)
            assertTrue(c.estPossible(new Coord(coord)));

        String[] coordImpossible = {"a1", "d3", "f7", "b2", "d4"};
        for(String coord : coordImpossible)
            assertFalse(c.estPossible(new Coord(coord)));
    }

    @Test
    void isCoupValid() {
        Cavalier c = new Cavalier(new Coord("d5"), Couleur.BLANC);
        Echiquier e = new Echiquier("8/1D1p4/2r5/3C4/8/8/8/7R",new FabPiece(), new FabChessJoueur());

        String[] coordPossible = {"b6","c7","e7","f6","f4","e3","c3","b4"};
        for(String coord : coordPossible)
            assertTrue(c.isCoupValid(new Coord(coord),e));

        String[] coordImpossible = {"a1", "d3", "f7", "b2", "d4"};
        for(String coord : coordImpossible)
            assertFalse(c.isCoupValid(new Coord(coord),e));
    }
```



Pièces

Cavalier 2/2

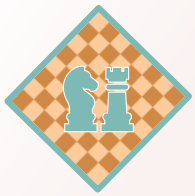
```
    @Test
    void getAllMoves() {
        Echiquier e = new Echiquier("r7/1D1p4/2c5/3C4/8/8/8/7R",new FabPiece(), new
FabChessJoueur());

        String[] coupPossible = {"b6","c7","e7","f6","f4","e3","c3","b4"};
        Coord cR = e.locateSensiblePiece(BLANC);
        List<IPiece> ennemies = e.getPieceFromColor(NOIR);
        IPiece p = e.getPiece(new Coord("d5"));

        LinkedList<Coord> coupCavalier = p.getAllMoves(cR, ennemies, e);
        for(String coup : coupPossible)
            assertTrue(coupCavalier.contains(new Coord(coup)));
    }


}
```



Pièces

Dame 1/2

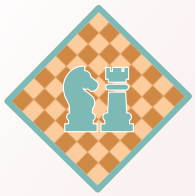
```
package pieces;

import Joueur.FabChessJoueur;
import echiquier.Coord;
import echiquier.Couleur;
import echiquier.Echiquier;
import echiquier.IPiece;
import org.junit.jupiter.api.Test;

import java.util.LinkedList;
import java.util.List;

import static echiquier.Couleur.BLANC;
import static echiquier.Couleur.NOIR;
import static org.junit.jupiter.api.Assertions.*;

class DameTest {

    @Test
    void estPossible() {
        Dame d = new Dame(new Coord("d5"), Couleur.BLANC);

        String[] coordPossible = {"d1","d2","d6","d8","a5","g5","f7","a8","c4","f3"};
        for(String coord : coordPossible)
            assertTrue(d.estPossible(new Coord(coord)));

        String[] coordImpossible = {"f6", "g6", "b4", "e1", "g4"};
        for(String coord : coordImpossible)
            assertFalse(d.estPossible(new Coord(coord)));
    }

    @Test
    void isCoupValid() {
        Dame d = new Dame(new Coord("a8"), Couleur.NOIR);

        Echiquier e = new Echiquier("d3p3/p7/2p5/8/8/8/8/r6R",new FabPiece(), new FabChessJoueur());

        String[] coordPossible = {"b8","c8","d8","b7"};
        for(String coord : coordPossible)
            assertTrue(d.isCoupValid(new Coord(coord),e));

        String[] coordImpossible = {"a1", "d3", "f7", "b2", "d4"};
        for(String coord : coordImpossible)
            assertFalse(d.isCoupValid(new Coord(coord),e));
    }
```



Pièces

Dame 2/2

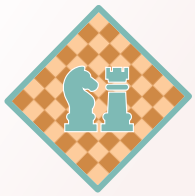
```
    @Test
    void getAllMoves() {
        Echiquier e = new Echiquier("d3p3/p7/2p5/8/8/8/8/r6R",new FabPiece(), new FabChessJoueur());

        String[] coupPossible = {"b8","c8","d8","b7"};
        Coord cR = e.locateSensiblePiece(BLANC);
        List<IPiece> ennemies = e.getPieceFromColor(NOIR);
        IPiece p = e.getPiece(new Coord("a8"));

        LinkedList<Coord> coupDame = p.getAllMoves(cR, ennemies, e);
        for(String coup : coupPossible)
            assertTrue(coupDame.contains(new Coord(coup)));
    }


}
```



Pièces

Fou 1/2

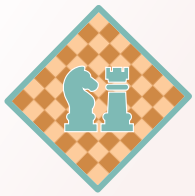
```
package pieces;

import Joueur.FabChessJoueur;
import echiquier.Coord;
import echiquier.Couleur;
import echiquier.Echiquier;
import echiquier.IPiece;
import org.junit.jupiter.api.Test;

import java.util.LinkedList;
import java.util.List;

import static echiquier.Couleur.BLANC;
import static echiquier.Couleur.NOIR;
import static org.junit.jupiter.api.Assertions.*;

class FouTest {

    @Test
    void estPossible() {
        Fou f = new Fou(new Coord("d5"), Couleur.BLANC);

        String[] coordPossible = {"b7","e6","f3","a2"};
        for(String coord : coordPossible)
            assertTrue(f.estPossible(new Coord(coord)));

        String[] coordImpossible = {"f6", "d4", "b4", "e1", "g4"};
        for(String coord : coordImpossible)
            assertFalse(f.estPossible(new Coord(coord)));
    }


    @Test
    void isCoupValid() {
        Fou f = new Fou(new Coord("a8"), Couleur.NOIR);

        Echiquier e = new Echiquier("f3p3/p7/8/8/4p3/8/8/r6R",new FabPiece(), new FabChessJoueur());
        String[] coordPossible = {"b7","c6","d5"};
        for(String coord : coordPossible)
            assertTrue(f.isCoupValid(new Coord(coord),e));

        String[] coordImpossible = {"e4", "f3", "f7", "b2", "d4"};
        for(String coord : coordImpossible)
            assertFalse(f.isCoupValid(new Coord(coord),e));
    }
```



Pièces

Fou 2/2

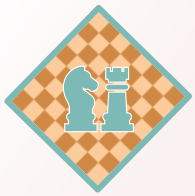
```
    @Test
    void getAllMoves() {
        Echiquier e = new Echiquier("f3p3/p7/8/8/4p3/8/8/r6R",new FabPiece(), new FabChessJoueur());

        String[] coupPossible = {"b7","c6","d5"};
        Coord cR = e.locateSensiblePiece(BLANC);
        List<IPiece> ennemies = e.getPieceFromColor(NOIR);
        IPiece p = e.getPiece(new Coord("a8"));

        LinkedList<Coord> coupDame = p.getAllMoves(cR, ennemies, e);
        for(String coup : coupPossible)
            assertTrue(coupDame.contains(new Coord(coup)));
    }

}
```



Pièces

Pion 1/2

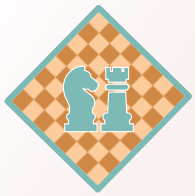
```
package pieces;


import Joueur.FabChessJoueur;
import echiquier.Coord;
import echiquier.Echiquier;
import echiquier.IPiece;
import org.junit.jupiter.api.Test;

import java.util.LinkedList;
import java.util.List;

import static echiquier.Couleur.BLANC;
import static echiquier.Couleur.NOIR;
import static org.junit.jupiter.api.Assertions.*;

class PionTest {

    @Test
    void estPossible() {
        Pion p = new Pion(new Coord("b7"), NOIR);
        String[] coordPossible = {"b5","b6","a6","c6"};
        for(String coord : coordPossible)
            assertTrue(p.estPossible(new Coord(coord)));

        String[] coordImpossible = {"b4","a5","c5","a8","b8","c8"};
        for(String coord : coordImpossible)
            assertFalse(p.estPossible(new Coord(coord)));

        Pion p2 = new Pion(new Coord("b2"), BLANC);
        String[] coordPossible2 = {"b3","b4","a3","c3"};
        for(String coord : coordPossible2)
            assertTrue(p2.estPossible(new Coord(coord)));

        String[] coordImpossible2 = {"b5","a4","c4","a1","b1","c1"};
        for(String coord : coordImpossible2)
            assertFalse(p2.estPossible(new Coord(coord)));
    }
```



Pièces

Pion 2/2

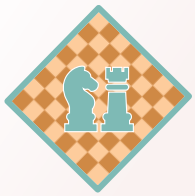
```
@Test
void isCoupValid() {
    Pion p = new Pion(new Coord("b7"), NOIR);
    Echiquier e = new Echiquier("8/1p6/P1p5/8/4p3/8/8/r6R",new FabPiece(), new FabChessJoueur());
    String[] coordPossible = {"b5","b6","a6"};
    for(String coord : coordPossible)
        assertTrue(p.isCoupValid(new Coord(coord),e));

    String[] coordImpossible = {"c6", "b4","a5","c5","a8","b8","c8"};
    for(String coord : coordImpossible)
        assertFalse(p.isCoupValid(new Coord(coord),e));
}

@Test
void autoPromote(){
    Pion pion = new Pion(new Coord("a8"),BLANC);
    IPiece p = pion.autoPromote();
    assertEquals(p.dessiner(), "D");
    Pion pion2 = new Pion(new Coord("a8"),NOIR);
    IPiece p2 = pion2.autoPromote();
    assertEquals(p2.dessiner(), "d");
}

@Test
void getAllMoves() {
    Echiquier e = new Echiquier("8/1p6/P1p5/8/4p3/8/8/r6R",new FabPiece(), new FabChessJoueur());

    String[] coupPossible = {"b5","b6","a6"};
    Coord cR = e.locateSensiblePiece(NOIR);
    List<IPiece> ennemies = e.getPieceFromColor(BLANC);
    IPiece p = e.getPiece(new Coord("b7"));

    LinkedList<Coord> coupDame = p.getAllMoves(cR, ennemies, e);
    for(String coup : coupPossible)
        assertTrue(coupDame.contains(new Coord(coup)));
}

}
```



Pièces

Roi 1/2

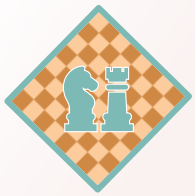
```
package pieces;

import Joueur.FabChessJoueur;
import echiquier.Coord;
import echiquier.Echiquier;
import org.junit.jupiter.api.Test;
import echiquier.*;

import java.util.LinkedList;
import java.util.List;

import static echiquier.Couleur.*;

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


class RoiTest {

    @Test
    void estPossible() {
        Roi r = new Roi(new Coord("d5"), BLANC);
        String[] coordPossible = {"d4","c4","c5","c6","d6","e6","e5","e4"};
        for(String coord : coordPossible)
            assertTrue(r.estPossible(new Coord(coord)));

        String[] coordImpossible = {"a1", "d3", "f7", "b2", "e3"};
        for(String coord : coordImpossible)
            assertFalse(r.estPossible(new Coord(coord)));
    }
```



Pièces

Roi 2/2

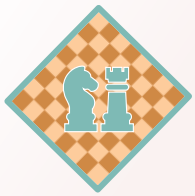
```
@Test
void isCoupValid() {
    Roi r = new Roi(new Coord("c6"), NOIR);
    Echiquier e = new Echiquier("8/1D1p4/2r5/1P1p4/8/8/8/7R",new FabPiece(), new FabChessJoueur());

    String[] coordPossible = {"c5","b5","b6","b7","c7","d6"};
    for(String coord : coordPossible)
        assertTrue(r.isCoupValid(new Coord(coord), e));

    String[] coordImpossible = {"d7", "d5"};
    for(String coord : coordImpossible)
        assertFalse(r.isCoupValid(new Coord(coord), e));
}

@Test
void getAllMoves() {
    Echiquier e = new Echiquier("8/1D1p4/2r5/1P1p4/8/8/8/7R",new FabPiece(), new FabChessJoueur());

    String[] coupPossible = {"b7", "d6", "c5"};
    Coord cR = e.locateSensiblePiece(NOIR);
    List<IPiece> ennemies = e.getPieceFromColor(BLANC);
    IPiece p = e.getPiece(new Coord("c6"));

    LinkedList<Coord> coupRoi = p.getAllMoves(cR, ennemies, e);
    for(String coup : coupPossible)
        assertTrue(coupRoi.contains(new Coord(coup)));
}
```



Pièces

Tour 1/2

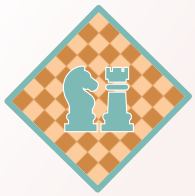
```
package pieces;

import Joueur.FabChessJoueur;
import echiquier.Coord;
import echiquier.Echiquier;
import org.junit.jupiter.api.Test;
import echiquier.*;

import java.util.LinkedList;
import java.util.List;

import static echiquier.Couleur.*;

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

class TourTest {

    @Test
    void estPossible() {
        Tour t = new Tour(new Coord("a3"),Couleur.NOIR);
        String[] coordPossible = {"a5", "a1", "b3", "a2", "c3", "d3"};
        for(String coord : coordPossible)
            assertTrue(t.estPossible(new Coord(coord)));
    }

    @Test
    void isCoupValid() {
        Echiquier e = new Echiquier("T7/8/8/8/8/r6R/8/8",new FabPiece(), new FabChessJoueur());
        String[] coupPossible = {"b8, c8, d8, e8, f8, g8, h8"};
        IPiece p = e.getPiece(new Coord("A8"));
        for(String c: coupPossible) {
            assertTrue(p.isCoupValid(new Coord(c), e));
        }
    }
```



Pièces

Tour 2/2

```
    @Test
    void getAllMoves(){
        Echiquier e = new Echiquier("T7/PPPPPPPP/8/8/8/r6R/8/8",new FabPiece(), new
FabChessJoueur());
        String[] coupPossible = {"b8, c8, d8, e8, f8, g8, h8"};
        Coord cR = e.locateSensiblePiece(BLANC);
        List<IPiece> ennemies = e.getPieceFromColor(NOIR);
        IPiece p = e.getPiece(new Coord("A8"));

        LinkedList<Coord> coupTour = p.getAllMoves(cR, ennemies, e);
        for(String c: coupPossible){
            assertTrue(coupTour.contains(new Coord(c)));
        }
    }


}
```



Codes Sources

Echiquier – 30 à 58

IChessJoueur – 30

IFabChessJoueur - 31

IFabPiece - 32

IPiece – 33 à 36

Coord – 37 à 41

Echiquier – 42 à 49

Utils – 50 à 53

Regle – 54 à 57

Couleur – 58

Pieces – 59 à 74

Piece – 59 à 61

FabPiece _ 62

Pion – 63 à 65

Roi – 66 à 69

Dame – 70

Tour - 71

Fou - 72

Cavalier – 73

Vide - 74

Joueur – 75 à 83

ChessJoueur - 75

ChessIHM – 76 à 78

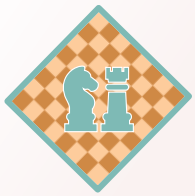
ChessJoueurIA – 79 à 81

ChessJoueurHumain - 82

FabChessJoueur - 83

Appli – 84 à 86

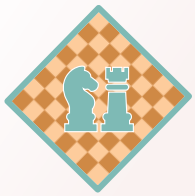
Application – 84 à 86



Echiquier

IChessJoueur

```java
package echiquier;

import java.util.List;

/**
 * interface definissant les joueurs d'un echiquier.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public interface IChessJoueur {

   /**
    * Renvoie le coup du joueur,
    * si c'est un Humain le coup se fera avec l'invite de commande
    * mais avec une IA celle-ci a besoin de l'etat de l'echiquier.
    * @param e l'echiquier
    * @param allys les pieces alliés
    * @param enemies les pieces ennemies
    * @param sC la coordonnée sensible de l'allié
    * @return le coup du joueur.
    */
   String getCoup(Echiquier e, List<IPiece> allys, List<IPiece> enemies, Coord sC);

   /** Renvoie la couleur de la pièce */
   Couleur getCouleur();

   /** Renvoie une chaine de caractères du coup au format "c3c4" */
   String coupToString(String coup);

   /** demande au joueur s'il accept la nulle (une Ia ne refuse jamais) */
   boolean acceptDraw();
}
```

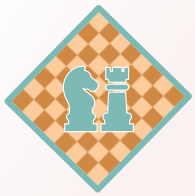


Echiquier

IFabChessJoueur

```java
package echiquier;

/**
 * fabrique de joueur
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public interface IFabChessJoueur {

   /**
    * Crée un joueur selon le type donnée
    * 'H' pour joueur Humain
    * 'I' pour jouer IA
    * @param type le type de joueur
    * @param couleur la couleur associé au joueur
    * @return le joueur
    */
   IChessJoueur getJoueur(char type, Couleur couleur);
}
```

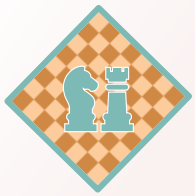
```java
package echiquier;

/**
 * fabrique de piece.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public interface IFabPiece {

    /**
     * Crée une pièce selon le type donné
     * 'D' pour dame, 'R' pour roi etc
     * @param type le caractère représentant le type de pièce voulu
     * @param c la coordonnée associée a la piece
     * @return la pièce
     */
    IPiece getPiece(char type, Coord c);

}
```



Echiquier

IPiece 1/4

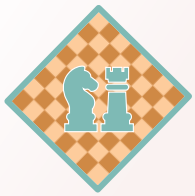
```java
package echiquier;

import java.util.LinkedList;
import java.util.List;

/**
 * Interface definissant les elements de l'echiquier, ici des pieces.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public interface IPiece {

   /** Met à jour la position de la pièce selon la position donnée en paramètre */
   void newPos(Coord c);

   /** Renvoie la couleur de la pièce */
   Couleur getCouleur();

   /**
    * Pour une position donnée, calcule si le déplacement est possible pour la pièce
    * @param c@return si le deplacement est possible pour la pièce ou non
    */
   boolean estPossible(Coord c);

   /**
    * Renvoie le type de la pièce avec comme difference BLANC en MAJUSCULE, NOIR en minuscule
    * @return la représentation de la pièce
    */
   String dessiner();

   /** Renvoie la coordonnée */
   Coord getCoord();
```



Echiquier

IPiece 2/4

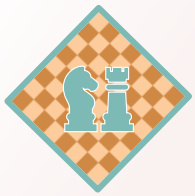
```
/**
 * Verifie si pour une coordonné d'arrivée la piece peut effectué le deplacement
 * dans l'echiquier.
 * verifie si la voie est libre, le deplacement est possible etc...
 * @param cF coordonné d'arrivée
 * @param e l'échiquier
 * @return le coup est valide
 */
boolean isCoupValid(Coord cF, Echiquier e);

/** Vérifie si la piece est sensible*/
boolean estSensible();

/** Vérifie si la case est représenté par une pièce vide */
boolean estVide();

/** Renvoie la piece chargé de promouvoir la piece a etre promu*/
IPiece autoPromote();

/**
 * Verifie si une piece peut se deplacer sur une coordonnée donnée.
 * à la difference de isCoupValid() cette methode ne verifie pas
 * si l'arrivé d'une piece vers la coordonnée est correct.
 * @param c coordonné d'arrivée
 * @param e L'échiquier
 * @return  Si la piéce est attaquable
 */
boolean peutAttaquer(Coord c, Echiquier e);
```



Echiquier

IPiece 3/4

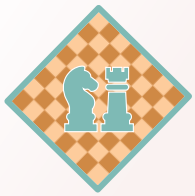
```java
/**
 * Verifie si pour une coordonné d'arrivée la piece peut effectué le deplacement
 * dans l'echiquier.
 * verifie si la voie est libre, le deplacement est possible etc...
 * @param cF coordonné d'arrivée
 * @param e l'échiquier
 * @return le coup est valide
 */
boolean isCoupValid(Coord cF, Echiquier e);

/** Vérifie si la piece est sensible*/
boolean estSensible();

/** Vérifie si la case est représenté par une pièce vide */
boolean estVide();

/** Renvoie la piece chargé de promouvoir la piece a etre promu*/
IPiece autoPromote();

/**
 * Verifie si une piece peut se deplacer sur une coordonnée donnée.
 * à la difference de isCoupValid() cette methode ne verifie pas
 * si l'arrivé d'une piece vers la coordonnée est correct.
 * @param c coordonné d'arrivée
 * @param e L'échiquier
 * @return  Si la piéce est attaquable
 */
boolean peutAttaquer(Coord c, Echiquier e);
```



Echiquier

IPiece 4/4

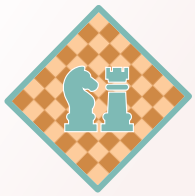
```
    /**
     * Renvoie tout les mouvements possible de la piece sur l'echiquier.
     * @param sC la coordonnée sensible de l'allié
     * @param ennemies Les pièces enemies
     * @param e l'echiquier
     * @return tout les movements possible
     */
    LinkedList<Coord> getAllMoves(Coord sC, List<IPiece> ennemies, Echiquier e);

    /** Renvoie la capacité d'une piece a tenir une fin de partie (quand il ne reste qu'elle et les rois)*/
    boolean canHoldEndGame();
}
```



Echiquier

Coord 1/5

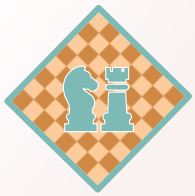
```java
package echiquier;

import static java.lang.Math.abs;

/**
 * permet de manipuler des coordonnées (x,y)
 * notation :
 * cS = cStart ----> coordonnées de depart
 * cF = cFinal ----> coordonnées d'arrivé
 * sC = sensibleCoord ----> coordonnées sensible
 * cP = coordPiece ----> coordonnées de la Piece.
 * etc...
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Coord implements Cloneable{

   /* X represente les lignes et y les colonnes d'un tableau 2d */
   private int x,y; // A DEMANDER SI C'EST CORRECT

   public int getX() {
      return x;
   }

   public int getY() {
      return y;
   }

   /** Constructeur de coordonnée*/
   public Coord(int x, int y) {
      this.x = x;
      this.y = y;
   }
```



Echiquier

Coord 2/5

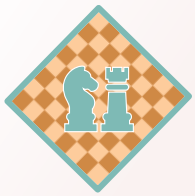
```java
/**
 * Constructeur de coordonnée a partir d'une String
 * "b2" devient (6, 1), les lignes et colonnes sont inversés
 * - pour arriver d'une lettre majuscule à un chiffre il faut le soustraire par 65
 * - pour arriver d'un nombre en char a un Int il faut le soustaire par 48, puis l'inverser
 * selon les lignes de l'echiquier.
 **/
public Coord(String c){
   this(abs((c.charAt(1) - 48) - Echiquier.LIGNE), Character.toUpperCase(c.charAt(0)) - 65);
}

/**
 * Ajoute les valeurs d'une coordonnée à une autre
 * @param c la coordonnée a ajouter (souvent un mouvement primaire)
 */
public void add(Coord c){
   this.x += c.x;
   this.y += c.y;
}

/**
 * Renvoie si le chemin est droit ou non
 * (diagonale, horizontale, verticale)
 * @param cS coordonnées de depart
 * @param cF coordonnées d'arrivé
 * @return le chemin est horizontale, verticale ou diagonale
 */
public static boolean isStraightPath(Coord cS, Coord cF){
   int difX = abs(cS.x - cF.x);
   int difY = abs(cS.y - cF.y);
   return (difX == 0 ||
         difY == 0 ||
         difX == difY);
}
```



Echiquier

Coord 3/5

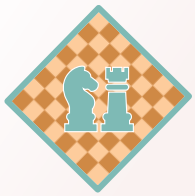
```java
/**
 * Constructeur de coordonnée a partir d'une String
 * "b2" devient (6, 1), les lignes et colonnes sont inversés
 * - pour arriver d'une lettre majuscule à un chiffre il faut le soustraire par 65
 * - pour arriver d'un nombre en char a un Int il faut le soustaire par 48, puis l'inverser
 * selon les lignes de l'echiquier.
 **/
public Coord(String c){
   this(abs((c.charAt(1) - 48) - Echiquier.LIGNE), Character.toUpperCase(c.charAt(0)) - 65);
}

/**
 * Ajoute les valeurs d'une coordonnée à une autre
 * @param c la coordonnée a ajouter (souvent un mouvement primaire)
 */
public void add(Coord c){
   this.x += c.x;
   this.y += c.y;
}

/**
 * Renvoie si le chemin est droit ou non
 * (diagonale, horizontale, verticale)
 * @param cS coordonnées de depart
 * @param cF coordonnées d'arrivé
 * @return le chemin est horizontale, verticale ou diagonale
 */
public static boolean isStraightPath(Coord cS, Coord cF){
   int difX = abs(cS.x - cF.x);
   int difY = abs(cS.y - cF.y);
   return (difX == 0 ||
         difY == 0 ||
         difX == difY);
}
```



Echiquier

Coord 4/5

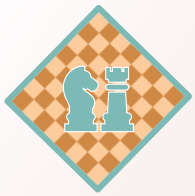
```java
/**
 * Renvoie le mouvement primaire entre deux points.
 * EST(1,0),NORD_EST(1,1),NORD(0,1),NORD_OUEST(-1,1),OUEST (-1,0),SUD_OUEST(-1,-1),SUD(0,-1),SUD_EST(1,-1)
 * @param cS coordonnées de la case de depart
 * @param cF coordonnées de la case d'arrivé
 * @return le mouvement primaire dans un tableau
 * */
public static Coord getPrimaryMove(Coord cS, Coord cF){
   int x = (cF.x - cS.x);
   int y = (cF.y - cS.y);
   if(abs(x)>1)
      x=x/ abs(x);
   if(abs(y)>1)
      y=y/ abs(y);
   return new Coord(x,y);
}

/**
 * Méthode de comparaison d'une coordonnée à un objet.
 * lorsque des methodes comme Contains compare deux objets elle compare les adresses
 * de ces objets en memoire. Dans notre cas deux coordonnées peuvent etre identique
 * sans avoir la meme adresse, alors on redefinie la method Equals pour ne pas
 * uttiliser celle par defaut.
 * @param o l'objet avec laquelle on veut comparer
 * @return si les objets sont identiques
 */
@Override
public boolean equals(Object o) {
   if (this == o) return true;
   if (o == null || getClass() != o.getClass()) return false;
   Coord coord = (Coord) o;
   return x == coord.x && y == coord.y;
}
```



Echiquier

Coord 5/5

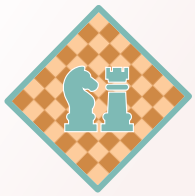
```
    /** revoie le clone d'une coordonnée */
    @Override
    public Coord clone() {
        return new Coord(this.x, this.y);
    }

    /**
     * la coordonnée en format conventionelle
     * une coordonnée 0,1 devient "b8" (les lignes et les colonnes sont inversé)
     * car dans la table Ascii les lettres commenece à la decimal 97
     * Enfin le numero de ligne est inversé par rapport au ligne d'un tableau
     * (la ligne 1 du tableau devient la ligne 7 sur l'echiquier, 3 -> 5, 2 -> 6 etc...)
     * */
    @Override
    public String toString(){
        return (char)(y + 97)+""+abs(x - Echiquier.LIGNE);
    }
}
```



Echiquier

Echiquier 1/8

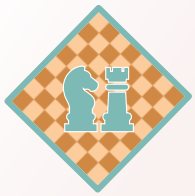
```java
package echiquier;

import java.util.LinkedList;

/**
 * Le corps du projet.
 * fonctione grace a la notation Forsyth-Edwards.
 * il doit contenir deux pieces sensible (2 rois par defaut).
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Echiquier {

   /* Un echiquier est constitué de 8 colonnes et 8 lignes soit 64 cases */
   public static final int LIGNE = 8, COLONNE = 8;

   /* La notation Forsyth-Edwards sert à noter la position des pièces sur l'échiquier.
    * pour l'uttiliser on commence par la premiere ligne est on note l'initial de la piece
    * que l'on veut a la position donnée.
    * exemple : tcfdrct/ donne une tour en A8, un cavalier en B8, un fou en C8 etc
    * le " / " delimite les lignes et les nombres donnent les cases vides.
    */
   private static final String BasicFen = "tcfdrfct/pppppppp/8/8/8/8/PPPPPPPP/TCFDRFCT"; //fen generique

   /* les joueurs des pieces Noirs et Blances */
   private IChessJoueur j1, j2;

   /* l'échiquier est représenté par un tableau 2d de pieces */
   private IPiece[][] echiquier;

   /* la fabrique de piece est sauvegardé en attribut pour pouvoir l'utiliser en dehors du constructeur */
   private final IFabPiece fabrique;
```



Echiquier

Echiquier 2/8

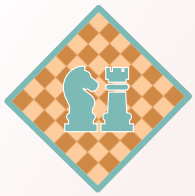
```
/* l'échiquier doit etre composer de deux Rois (blanc et noir) */
static class NoSensiblePieceException extends Exception{}

/**  */
static class NonValidFenException extends Exception{}

/**
 * constructeur de l'échiquier
 * @param fPiece la fabrique d'objet pièce
 * @param fen l'enregistrement fen
 * @param mode le mode chosi pour jouer
 * @param fJoueur fabrique de Joueur
 */
public Echiquier(IFabPiece fPiece, String fen, String mode, IFabChessJoueur fJoueur) {
   setJoueur(mode, fJoueur);
   echiquier = new IPiece[LIGNE][COLONNE ];
   this.fabrique = fPiece;
   try{
      fillBoard(fPiece, fen);
      SensibleError();          //un echiquier doit contenir deux rois
   }catch (NonValidFenException  err){
      throw new IllegalArgumentException("La fen n'est pas correct");
   }catch (NoSensiblePieceException err){
      throw new IllegalArgumentException("Il manque deux pieces sensible à la fen");
   }
}

/** constructeur de l'échiquier avec l'enregistrement fen generique */
public Echiquier(IFabPiece fabrique, String mode, IFabChessJoueur fJoueur){
   this(fabrique, BasicFen, mode, fJoueur);
}

/** constructeur par defaut d'un echequier */
public Echiquier(IFabPiece fabrique, IFabChessJoueur fJoueur){
   this(fabrique, BasicFen, "pp", fJoueur);
}

/** constructeur de l'échiquier avec le mode par default (pour les test) */
public Echiquier(String fen, IFabPiece fabrique, IFabChessJoueur fJoueur){
   this(fabrique, fen, "pp",fJoueur);
}
```



Echiquier

Echiquier 3/8

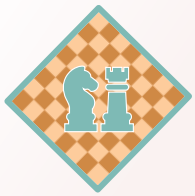
```java
/**
 * setter des jouers de l'echiquier selon le mode.
 * si le mode par defaut est humain contre humain (raison du switch redondant)
 * @param mode le mode
 * @param fJoueur la fabrique de Joueur
 */
private void setJoueur(String mode, IFabChessJoueur fJoueur){
   switch (mode) {
      case "pi":  j1 = fJoueur.getJoueur('H', Couleur.BLANC);
                  j2 = fJoueur.getJoueur('I', Couleur.NOIR);
                  break;
      case "ii":  j1 = fJoueur.getJoueur('I', Couleur.BLANC);
                  j2 = fJoueur.getJoueur('I', Couleur.NOIR);
                  break;
      default:    j1 = fJoueur.getJoueur('H', Couleur.BLANC);
                  j2 = fJoueur.getJoueur('H', Couleur.NOIR);
   }
}

/**
 * rempli le l'echiquier de piece selon la fen en parametre
 * @param fabrique la fabrique d'objet pièce
 * @param fen l'enregistrement fen
 * @throws ArrayIndexOutOfBoundsException si la fen n'est pas correct
 */
private void fillBoard(IFabPiece fabrique, String fen)
      throws NonValidFenException {
   String[] splittedFen = fen.split("/"); // le fen est divisé pour n'avoir qu'un tableau de ligne

   for (int lg = 0; lg < splittedFen.length; lg++) {
      String s = splittedFen[lg];

      // https://stackoverflow.com/a/18590949
      String sequence = (s.matches(".*\\d.*")) ? ReformatFenSequence(s) : s;

      if (sequence.length() != LIGNE) throw new NonValidFenException();

      for (int cl = 0; cl < COLONNE; cl++)
         echiquier[lg][cl] = fabrique.getPiece(sequence.charAt(cl), new Coord(lg, cl));
   }
}
```



Echiquier

Echiquier 4/8

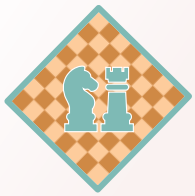
```java
/**
 * lorsque la sequence d'un enregistrement fen possede un entier il faut le remplacer par
 * l'initial "V" pour Piece VIDE, donc "drt3pp" devient "drtVVVpp"
 * @param str la sequence à formater
 * @return la sequence formatée
 */
private String ReformatFenSequence(String str){
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < str.length(); i++) {
        Character c = str.charAt(i);
        if(Character.isDigit(c)){
            for(int y = 0; y < c - 48; y++)   // les entiers commence à 48 dans la table ascii
                sb.append("V");
            continue;
        }
        sb.append(c);
    }
    return sb.toString();
}

/**
 * renvoie la liste de toutes les coordonnées atteignables dans l'échiquier par une piece donnée.
 * @param p la piece
 * @return la liste des coordonnées atteignable
 */
public LinkedList<Coord> allClassicMoves(IPiece p){
    LinkedList<Coord> allClassicMoves = new LinkedList<>();
    for(IPiece[] ligne : echiquier) {
        for (IPiece piece : ligne) {
            Coord c = piece.getCoord();
            if((p.isCoupValid(c, this)))
                allClassicMoves.add(c);
        }
    }
    return allClassicMoves;
}
```



Echiquier

Echiquier 5/8

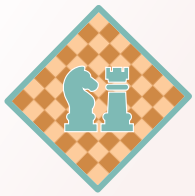
```java
/**
 * Renvoie une liste comportant toutes les pieces de l'echiquier
 * de couleur demandée.
 * @param couleur la couleur demandée
 * @return la liste de pieces
 */
public LinkedList<IPiece> getPieceFromColor(Couleur couleur){
   LinkedList<IPiece> pieces = new LinkedList<>();
   for(IPiece[] ligne : echiquier)
      for(IPiece p: ligne){
         if(Couleur.isRightColor(p, couleur))
            pieces.add(p);
      }
   return pieces;
}

/** Renvoie la pièce aux coordonnées en param */
public IPiece getPiece(Coord c){
   return echiquier[c.getX()][c.getY()];
}

/** renvoie le joueur selon sa couleur */
public IChessJoueur getJoueur(Couleur c){
   switch (c){
      case BLANC : return j1;
      case NOIR : return j2;
      default: return null;
   }
}

/** Vérifie si une case aux coordonnées est vide */
public boolean estVide(Coord c){
   return echiquier[c.getX()][c.getY()].estVide();
}
```



Echiquier

Echiquier 6/8

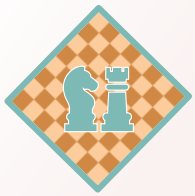
```java
/**
 * Renvoie une liste comportant toutes les pieces de l'echiquier
 * de couleur demandée.
 * @param couleur la couleur demandée
 * @return la liste de pieces
 */
public LinkedList<IPiece> getPieceFromColor(Couleur couleur){
   LinkedList<IPiece> pieces = new LinkedList<>();
   for(IPiece[] ligne : echiquier)
      for(IPiece p: ligne){
         if(Couleur.isRightColor(p, couleur))
            pieces.add(p);
      }
   return pieces;
}

/** Renvoie la pièce aux coordonnées en param */
public IPiece getPiece(Coord c){
   return echiquier[c.getX()][c.getY()];
}

/** renvoie le joueur selon sa couleur */
public IChessJoueur getJoueur(Couleur c){
   switch (c){
      case BLANC : return j1;
      case NOIR : return j2;
      default: return null;
   }
}

/** Vérifie si une case aux coordonnées est vide */
public boolean estVide(Coord c){
   return echiquier[c.getX()][c.getY()].estVide();
}
```



Echiquier

Echiquier 7/8

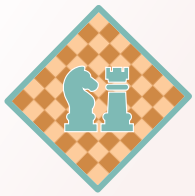
```java
/** Renvoie si une coordonnée est en dehors de l'echiquier ou non */
public static boolean inBound(Coord c){
   int x = c.getX(), y = c.getY();
   return (x >= 0 && x < LIGNE) && (y >= 0 && y < COLONNE);
}

/**
 * Recherche une piece sensible sur l'echiquier selon la couleur
 * @param couleur la couleur du roi (NOIR ou BLANC)
 * @return la coordonnées, null si aucune piece n'a été trouvé
 */
public Coord locateSensiblePiece(Couleur couleur) {
   for(IPiece[] ligne : echiquier)
      for(IPiece p : ligne)
         if(p.estSensible() && Couleur.isRightColor(p, couleur))
            return p.getCoord();
   return null;
}

/**
 * renvoie une exception si il y a pas de roi de couleur donné dans l'echiquier
 * @throws NoSensiblePieceException aucun roi trouvé
 */
private void SensibleError() throws NoSensiblePieceException {
   if(locateSensiblePiece(Couleur.BLANC) == null &&
      locateSensiblePiece(Couleur.NOIR) == null)
      throw new NoSensiblePieceException();
}
```



Echiquier

Echiquier 8/8

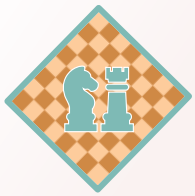
```
    /**
     * vérifie si une piece peut etre promu
     * https://www.apprendre-les-echecs-24h.com/blog/debuter-aux-echecs/promotion-aux-echecs/
     * @param couleur la couleur des pièces à vérifier
     */
    public void checkForPromote(Couleur couleur){
        int x = couleur == Couleur.BLANC ? 0 : LIGNE - 1, y = 0;
        for (IPiece p : echiquier[x])
            echiquier[x][y++] = p.autoPromote();
    }

    /** Créer une chaîne de caractères comportant l'ensemble de l'échiquier. */
    @Override
    public String toString() {
        String SAUT = "    --- --- --- --- --- --- --- ---   \n"; // delimite les lignes
        String LETTRE = "     a   b   c   d   e   f   g   h    \n"; // symbolise les colonnes
        int compteur = COLONNE; // compteur de de ligne
        StringBuilder sb = new StringBuilder(LETTRE + SAUT);

        for(IPiece[] ligne : echiquier){
            sb.append(" ").append(compteur).append(" |");

            for(IPiece p: ligne)
                sb.append(" ").append(p.dessiner()).append(" |");

            sb.append(" ").append(compteur--).append("\n").append(SAUT);
        }

        sb.append(LETTRE);
        return sb.toString();
    }
}
```



Echiquier

Utils 1/4

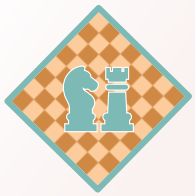
```java
package echiquier;

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

import static echiquier.Couleur.*;

/**
 * classe regroupant les utilitaires d'un echiquier.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Utils {

   /**
    * Renvoie le chemin de coordonnées entre deux coordonnées (Une liste de coord).
    * @param cS la coordonnée de départ
    * @param cF la coordonnée d'arrivée
    * @return les coordonnées de la droite entre les deux coords.
    */
   public static LinkedList<Coord> getPath(Coord cS, Coord cF){
      LinkedList<Coord> tile = new LinkedList<>(Collections.singletonList(cF));
      Coord cClone = cS.clone();
      if(!Coord.isStraightPath(cS, cF))       //  si le chemin n'est pas verticale ou horizontale
         return tile;                         //  il y a pas de chemin de case attaqué.

      Coord pM = Coord.getPrimaryMove(cS, cF);

      while(!cClone.equals(cF)) {
         tile.add(cClone.clone());
         cClone.add(pM);
      }

      return tile;
   }
```



Echiquier

Utils 2/4

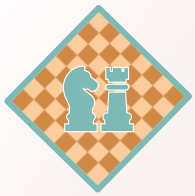
```java
/**
 * Retourne toutes les pièces attaquant une coordonnées.
 * @param c la coordonnée
 * @param pieces les pièces enemies
 * @param e l'echiquier
 * @return une liste des pièces attaquantes
 */
public static LinkedList<IPiece> getAllAttackingPiece(Coord c, List<IPiece> pieces, Echiquier e){
   LinkedList<IPiece> cPiece = new LinkedList<>();
   for(IPiece p : pieces)
      if(p.peutAttaquer(c, e))
         cPiece.add(p);
   return cPiece;
}

/**
 * Retourne les déplacements possibles d'une pièce pour défendre une coordonnée sensible
 * @param p la pièce
 * @param sC la coordonnée sensible
 * @param ennemies les pièces ennemies
 * @param e l'echiquier
 * @return les coordonnées possibles de la pièce pour défendre
 */
public static LinkedList<Coord> allMovesDefendingCheck(IPiece p, Coord sC, List<IPiece> ennemies, Echiquier e){
   LinkedList<Coord> moves = new LinkedList<>();
   LinkedList<IPiece> attackingPiece = getAllAttackingPiece(sC, ennemies, e);

   if(attackingPiece.size() != 1)    //s'il y a plusieurs piece attaquante
      return moves;                  //une piece ne peut pas defendre

   Coord cF = attackingPiece.get(0).getCoord();
   LinkedList<Coord> path = getPath(sC, cF);   //le chemin entre la piece sensible
                                               // et la piece attaquante
   return allValidMovesFromPath(p, path, e);
}
```



Echiquier

Utils 3/4

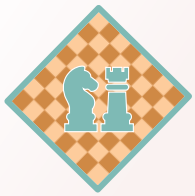
```java
/**
 * Retourne les coordonnées de déplacement possibles pour une pièce clouée
 * @param p la pièce clouée
 * @param sC la coordonné sensible qui donne lieu au clouage
 * @param e l'echiquier
 * @return la liste des coordonnées
 */
public static LinkedList<Coord> allMovesFromPin(IPiece p, Coord sC, Echiquier e){
   LinkedList<Coord> moves = new LinkedList<>();
   Coord cS = p.getCoord();
   IPiece pningPiece = getPningPiece(cS, sC, e); //la piece clouante

   if(pningPiece == null)
      return moves;

   List<Coord> path = getPath(cS, pningPiece.getCoord());
   return allValidMovesFromPath(p, path, e);
}

/**
 * Renvoie pour un chemin de coordonnée donné les coordonnées que la piece
 * peut acceder.
 * @param p la piece
 * @param path le chemin
 * @param e l'echiquier
 * @return les coordonnées accessible
 */
private static LinkedList<Coord> allValidMovesFromPath(IPiece p, List<Coord> path, Echiquier e){
   LinkedList<Coord> moves = new LinkedList<>();
   for(Coord c : path)
      if(p.isCoupValid(c, e))
         moves.add(c);
   return moves;
}
```



Echiquier

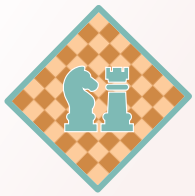
Utils 4/4

```
    /**
     * Retourne la pièce susceptible d'attaquer une coordonnée sensible,
     * si pièce testé n'est plus la.
     * Clouage en francais ou Pin en anglais.
     * @param cS la position de la pièce
     * @param sC la coordonnée sensible de l'allié
     * @param e l'echiquier
     * @return la pièce qui cloue ou rien
     */
    public static IPiece getPningPiece(Coord cS, Coord sC, Echiquier e){
        Couleur oppositeColor = getOpposite(e.getPiece(cS));
        List<Coord> pathToBorder = getPathToBorder(cS, Coord.getPrimaryMove(sC, cS));
        for(Coord c : pathToBorder){
            IPiece piece = e.getPiece(c);

            if(isRightColor(piece, oppositeColor) &&          // si dans ce chemin il y a une piece
                    piece.isCoupValid(cS, e) &&               // si la piece peut prendre la piece clouée
                    piece.estPossible(sC))                    // et le roi juste apres.
                return piece;
        }
        return null;   //aucune piece clouante
    }

    /**
     * Renvoie une liste de coordonnées allant d'une position donnée à la limite de l'echiquier.
     * Cette methode fonction grace au mouvement primaire pour retrouver la coordonnée
     * delimitant la fin de l'echiquier.
     * @param cS coord d'arrivé
     * @param cP mouvement primaire
     * @return les coordonnées
     */
    public static LinkedList<Coord> getPathToBorder(Coord cS, Coord cP) {
        Coord cF = cS.clone();  // clone pour manipuler

        // on applique le mouvement primaire tant que la limite de l'echiquier n'est pas atteinte
        while(Echiquier.inBound(cF)){
            cF.add(cP);    //add = addition de coordonnées avec un mvnt primaire
        }
        cF.add(new Coord(-(cP.getX()), -(cP.getY()) ) );  //on revient d'un en arriere car on est hors limite (boucle while)
        return getPath(cS, cF);
    }

}
```



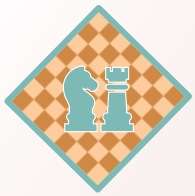
Echiquier
Regle 1/4

```java
package echiquier;

import java.util.LinkedList;
import java.util.List;

/**
 * classe regroupant les regles mis en place par les echecs.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 * */
public class Regle {

   /**
    * Verifie si pour une couleur donnée le roi est en echec et mat.
    * Le roi doit etre en echec et s'il n'y a aucun coup possible
    * alors la condition est verifié.
    * @param sC la coordonnée sensible de l'allié
    * @param allys toutes les pieces alliés
    * @param ennemies toutes les pieces de l'ennemi
    * @param e l'echiquier
    * @return si le roi est en échec et mat
    */
   public static boolean checkForMate(Coord sC, List<IPiece> allys, List<IPiece> ennemies, Echiquier e){
      if(!isAttacked(sC, ennemies, e))
         return false;

      return getAllPossibleMoves(sC, allys, ennemies, e).isEmpty();
   }

   /**
    * Verifie si il y a une egalité par pat dans l'etat actuel de l'echiquier.
    * le pat est une regle qui reconnait comme etant une egalité le cas ou le joueur
    * actuel n'a pas de coup possible parmis toutes ces pieces
    * sans que le roi soit en echec.
    * @param sC la coordonnée sensible de l'allié
    * @param allys les pieces alliées
    * @param ennemies les pieces ennemies
    * @param e l'echiquier
    * @return l'egalité par pat est detecté
    */
   public static boolean isStaleMate(Coord sC, List<IPiece> allys, List<IPiece> ennemies, Echiquier e){
      if(isAttacked(sC, ennemies, e))
         return false;

      return getAllPossibleMoves(sC, allys, ennemies, e).isEmpty();
   }
```



Echiquier
Regle 2/4

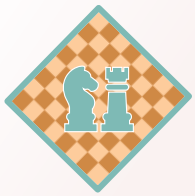
```java
package echiquier;

import java.util.LinkedList;
import java.util.List;

/**
 * classe regroupant les regles mis en place par les echecs.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 * */
public class Regle {

   /**
    * Verifie si pour une couleur donnée le roi est en echec et mat.
    * Le roi doit etre en echec et s'il n'y a aucun coup possible
    * alors la condition est verifié.
    * @param sC la coordonnée sensible de l'allié
    * @param allys toutes les pieces alliés
    * @param ennemies toutes les pieces de l'ennemi
    * @param e l'echiquier
    * @return si le roi est en échec et mat
    */
   public static boolean checkForMate(Coord sC, List<IPiece> allys, List<IPiece> ennemies, Echiquier e){
      if(!isAttacked(sC, ennemies, e))
         return false;

      return getAllPossibleMoves(sC, allys, ennemies, e).isEmpty();
   }

   /**
    * Verifie si il y a une egalité par pat dans l'etat actuel de l'echiquier.
    * le pat est une regle qui reconnait comme etant une egalité le cas ou le joueur
    * actuel n'a pas de coup possible parmis toutes ces pieces
    * sans que le roi soit en echec.
    * @param sC la coordonnée sensible de l'allié
    * @param allys les pieces alliées
    * @param ennemies les pieces ennemies
    * @param e l'echiquier
    * @return l'egalité par pat est detecté
    */
   public static boolean isStaleMate(Coord sC, List<IPiece> allys, List<IPiece> ennemies, Echiquier e){
      if(isAttacked(sC, ennemies, e))
         return false;

      return getAllPossibleMoves(sC, allys, ennemies, e).isEmpty();
   }
```



Echiquier

Regle 3/4

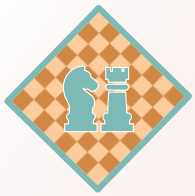
```java
/**
 * Renvoie tout les movememements possibles d'un groupe de piece.
 * on ne cherche pas a acceder aux mouvement mais a les denombrer
 * @param sC la coordonnée sensible de l'allié
 * @param allys les pieces alliées
 * @param ennemies les pieces ennemies
 * @param e l'echiquier
 * @return tout les movememements possibles d'un groupe de piece
 */
private static LinkedList<Coord> getAllPossibleMoves(Coord sC, List<IPiece> allys, List<IPiece> ennemies,
                                    Echiquier e){
   LinkedList<Coord> allPossibleMoves = new LinkedList<>();

   for(IPiece p: allys)
      allPossibleMoves.addAll(p.getAllMoves(sC, ennemies, e));

   return allPossibleMoves;
}

/**
 * verifie avec si la coordonnée sensible est attaquée par les pieces adverses.
 * @param sC les coordonnées du roi
 * @param ennemies les pièces présentes sur l'échiquier
 * @param e l'echiquier
 * @return si une pièce(s) menace(s) la coordonnée sensible
 * @see Utils#getAllAttackingPiece(Coord, java.util.List, Echiquier)
 */
public static boolean isAttacked(Coord sC, List<IPiece> ennemies, Echiquier e){
   return !Utils.getAllAttackingPiece(sC, ennemies, e).isEmpty();
}
```



Echiquier

Regle 4/4

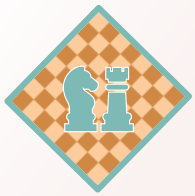
```java
    /**
     * verifie si le materiel present sur l'echiquier est suffisant pour continuer une partie.
     * https://www.chess.com/terms/draw-chess#:~:text=A%20draw%20occurs%20in%20chess,player%20wins%20half%20a%20point.
     * @param allys       les pieces alliées
     * @param ennemies    les pieces ennemies
     * @return            si le matériel des joueurs est insuffisant pour mettre l'un l'autre en echec et mat
     */
    public static boolean impossibleMat(List<IPiece> allys, List<IPiece> ennemies){
        LinkedList<IPiece> allPieces = new LinkedList<>(ennemies);
        allPieces.addAll(allys);
        allPieces.removeIf(p -> !p.canHoldEndGame());

        return allPieces.isEmpty();
    }

    /**
     * Vérifie si le chemin entre 2 points n'a pas d'obstacle (pas l'arrivé)
     * @param cF coordonnées de la case d'arrivé
     * @param p la pièce à deplacer
     * @param e l'echiquier
     * @return la voie est libre
     */
    public static boolean voieLibre(IPiece p, Coord cF, Echiquier e){
        Coord cS = p.getCoord();
        LinkedList<Coord> path = Utils.getPath(cS, cF);
        path.removeIf(c -> c.equals(cF) || c.equals(cS) || e.estVide(c));
        return path.isEmpty();
    }

    /**
     * Vérifie si la l'arrivé d'une piece sur une case est valide.
     * @param p la pièce à deplacer
     * @param c coordonnées de la case
     * @param e l'echiquier
     * @return l'arrivée est valide
     */
    public static boolean isFinishValid(IPiece p, Coord c, Echiquier e){
        IPiece pA = e.getPiece(c);
        return !Couleur.areSameColor(p, pA) && !pA.estSensible();
    }
}
```



Echiquier

Couleur

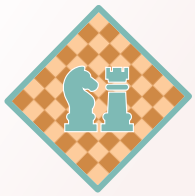
```java
package echiquier;

/**
 * les couleurs que peut attribuer l'echiquier a ces elements.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public enum Couleur {
   NOIR,
   BLANC,
   VIDE // Attribution de la couleur VIDE uniquement pour les cases vides de l'échiquier
   ;

   /** verifie que la couleur entre 2 pieces est identique*/
   public static boolean areSameColor(IPiece p1, IPiece p2){
      return p1.getCouleur() == p2.getCouleur();
   }

   /** verifie que la couleur d'une piece est celle souhaité*/
   public static boolean isRightColor(IPiece p, Couleur couleur){
      return p.getCouleur() == couleur;
   }

   /** verifie que 2 pieces sont opposés (NOIR vs BLANC)*/
   public static boolean areOpposite(IPiece p1, IPiece p2) {
      return p2.getCouleur().equals(getOpposite(p1));
   }

   /** renvoie la couleur opposé à la piece (a redefinir si ajout de Couleur)*/
   public static Couleur getOpposite(IPiece p1) {
      return p1.getCouleur().equals(BLANC) ? NOIR : BLANC;
   }
}
```



Pieces

Piece 1/3

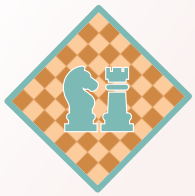
```java
package pieces;

import echiquier.Coord;
import echiquier.*;

import java.util.LinkedList;
import java.util.List;

import static echiquier.Couleur.BLANC;

/**
 * {@inheritDoc}
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 * */
public abstract class Piece implements IPiece
{
   /** Position de la pièce */
   protected Coord coord;

   /** Couleur de la pièce */
   private final Couleur couleur;

   /** Constructeur d'une pièce */
   public Piece(Couleur c, Coord coord) {
      newPos(coord);
      couleur=c;
   }

   /** {@inheritDoc} */
   @Override
   public void newPos(Coord coord){
      this.coord = coord;
   }

   /** {@inheritDoc} */
   @Override
   public final Couleur getCouleur(){
      return this.couleur;
   }

   /** {@inheritDoc} */
   @Override
   public abstract boolean estPossible(Coord c);
```



Pieces

Piece 2/3

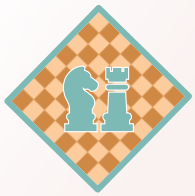
```java
/** Renvoie le symbole de la pièce */
public abstract String getSymbole();

/** {@inheritDoc} */
@Override
public final String dessiner(){
   return (couleur == BLANC) ? getSymbole() : getSymbole().toLowerCase() ;
}

/** {@inheritDoc} */
@Override
public final Coord getCoord() {
   return coord;
}

/** {@inheritDoc} */
@Override
public boolean isCoupValid(Coord cF, Echiquier e){
   return this.estPossible(cF) && Regle.voieLibre(this, cF, e) && Regle.isFinishValid(this, cF, e);
}

/** {@inheritDoc} */
@Override
public boolean estSensible(){
   return false;
}

/** {@inheritDoc} */
@Override
public boolean estVide(){
   return false;
}

/** {@inheritDoc} */
@Override
public IPiece autoPromote(){ return this;}

/** {@inheritDoc} */
public boolean peutAttaquer(Coord c, Echiquier e){
   return estPossible(c) && Regle.voieLibre(this, c, e);
}
```



Pieces

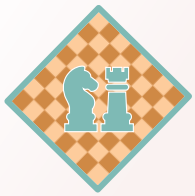
Piece 3/3

```
    /** {@inheritDoc} */
    @Override
    public LinkedList<Coord> getAllMoves(Coord sC, List<IPiece> ennemies, Echiquier e){
        boolean sCAttacked = Regle.isAttacked(sC, ennemies, e);

        if(isPinned(sC, e) && ! sCAttacked){
            return Utils.allMovesFromPin(this, sC, e);
        }

        if(sCAttacked)
            return Utils.allMovesDefendingCheck(this, sC, ennemies, e);

        return e.allClassicMoves(this);
    }

    /**
     * Retourne si la pièce est clouée
     * @param sC la coordonnée sensible de l'allié
     * @param e l'echiquier
     * @return si la pièce est clouée ou non
     */
    protected boolean isPinned(Coord sC, Echiquier e){
        if(!Coord.isStraightPath(coord, sC)||
            !Regle.voieLibre(this, sC, e))
            return false;

        return !(Utils.getPningPiece(coord, sC, e) == null);
    }

    /** {@inheritDoc} */
    @Override
    public boolean canHoldEndGame(){
        return true;
    }
}
```



Pieces
FabPiece

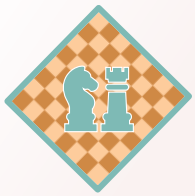
```java
package pieces;

import echiquier.Coord;
import echiquier.Couleur;
import echiquier.IFabPiece;
import echiquier.IPiece;

/**
 * {@inheritDoc}
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class FabPiece implements IFabPiece {

   /**
    * {@inheritDoc}
    */
   public IPiece getPiece(char type, Coord c) {
      Couleur color = Character.isUpperCase(type) ? Couleur.BLANC : Couleur.NOIR;
      switch(Character.toUpperCase(type)){
         case 'F':
            return new Fou(c, color);
         case 'R':
            return new Roi(c, color);
         case 'D':
            return new Dame(c, color);
         case 'P':
            return new Pion(c, color);
         case 'T':
            return new Tour(c, color);
         case 'C':
            return new Cavalier(c, color);
         default: return new Vide(c, Couleur.VIDE);
      }
   }
}
```



Pieces

Pion 1/3

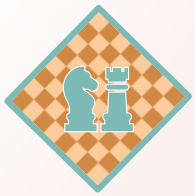
```java
package pieces;

import echiquier.Coord;
import echiquier.Couleur;
import echiquier.Echiquier;
import echiquier.IPiece;

import static java.lang.Math.abs;
import static echiquier.Couleur.*;

/**
 * C'est la pièce la moins mobile du jeu et pour cette raison la moins forte.
 * Depuis sa position d'origine, le pion peut avancer d'une ou deux cases.
 * Par la suite, le pion avance d'une seule case à la fois, sans changer de colonne
 * et seulement vers une case vide.
 * Le pion ne peut ni reculer, ni prendre vers l'arrière ou le côté.
 * Le pion prend en diagonale vers l'avant.
 * Le pion peut prendre en avançant d'une case en diagonale n'importe
 * quelle pièce adverse qui s'y trouverait.
 * /!\ la prise en passant n'est pas codé
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Pion extends Piece{

    /** les 2 coordonnées en x possible de départ d'un pion
     * (pour les test, lorsque la fen est personnalisé, le pion
     * n'a pas le droit d'avancer sur 2 deux s'il n'est pas a une
     * position de depart*/
    private static final int START1 = 1, START2 = 6;

    /** si la pièce a bougée */
    private boolean firstMove;

    /** la direction que le pion doit suivre */
    private int forward;

    /**
     * Constructeur d'un pion
     * @see Piece#Piece(Couleur, Coord)
     */
    public Pion(Coord coord, Couleur c) {
        super(c, coord);
        firstMove = true;
        this.forward = c == BLANC ? -1 : 1;
    }
```



Pieces

Pion 2/3

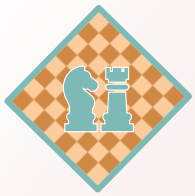
```java
/** le pion a fait son premier pas*/
@Override
public void newPos(Coord coord) {
   super.newPos(coord);
   firstMove = false;
}

/** {@inheritDoc} */
@Override
public boolean estPossible(Coord c) {
   int varX = c.getX() - coord.getX();
   int varY = abs(coord.getY()-c.getY());
   return (varY == 1 || varY == 0) && varX == forward ||
         (isFirstMove() && varX == (2*forward) && varY == 0);
}

/** Calcule si c'est le premier mouvement du pion
 *
 * @return si la pièce a bougée ou non
 */
private boolean isFirstMove(){
   int x = this.coord.getX();
   return (x == START1 || x == START2) && firstMove;
}

/** {@inheritDoc} */
@Override
public String getSymbole() {
   return "P";
}
```



Pieces

Pion 3/3

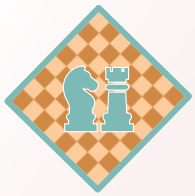
```
    /**
     * le pion gère aussi la prise en diagonale
     * {@inheritDoc}
     */
    @Override
    public boolean isCoupValid(Coord cF, Echiquier e) {
        IPiece p = e.getPiece(cF);
        return super.isCoupValid(cF, e) &&
                (isPriseEnDiag(cF) ? areOpposite(this, p) && !p.estVide() : p.estVide());
    }

    /**
     * si le mouvement d'un pion resulte en un coup diagonale
     * @param c la coordonnées d'arrivée
     * @return le coup est une prise diagonale
     */
    private boolean isPriseEnDiag(Coord c){
        int diffX = abs(c.getX() - coord.getX());
        int diffY = abs(c.getY() - coord.getY());
        return diffX == 1 && diffY == 1;
    }

    /** le pion est promouvable */
    @Override
    public IPiece autoPromote() {
        return new Dame(this.coord, this.getCouleur());
    }

    /**
     * gère la prise en diagonale
     * {@inheritDoc}
     */
    @Override
    public boolean peutAttaquer(Coord c, Echiquier e) {
        return super.peutAttaquer(c, e) && this.isPriseEnDiag(c);
    }
}
```



Pieces

Roi 1/4

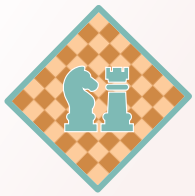
```java
package pieces;

import echiquier.Coord;
import echiquier.Couleur;
import echiquier.Echiquier;
import echiquier.IPiece;
import echiquier.Utils;

import java.util.LinkedList;
import java.util.List;

import static echiquier.Regle.isAttacked;
import static java.lang.Math.abs;

/**
 * Le roi ( ♔, ♚) est la pièce clé du jeu d'échecs.
 * Si le roi d'un joueur est menacé de capture au prochain coup de façon imparable,
 * il est dit échec et mat et le joueur concerné perd la partie.
 * Le roi se déplace d'une case dans n'importe quelle direction.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Roi extends Piece{

   /** Constructeur d'un roi */
   public Roi(Coord coord, Couleur c) {
      super(c, coord);
   }

   /** {@inheritDoc} */
   @Override
   public boolean estPossible(Coord c) {
      int varX = abs(coord.getX()-c.getX());
      int varY = abs(coord.getY()-c.getY());
      if(varX == 0 && varY == 0 ) // si la pièce fait du sur place
         return false;
      // le déplacement est valide seulement si le roi se déplace dans un rayon de une case
      return (varX == 1 || varX == 0) && (varY == 0 || varY == 1);
   }
```



Pieces

Roi 2/4

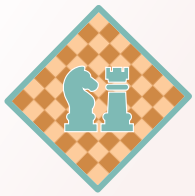
```java
/** {@inheritDoc} */
@Override
public String getSymbole() {
   return "R";
}

/** le roi est sensible au attaque */
@Override
public boolean estSensible() {
   return true;
}

/**
 * il faut retirer au mouvement du roi chaque coordonnées susceptible
 * d'etre attaqué.
 * */
@Override
public LinkedList<Coord> getAllMoves(Coord sC, List<IPiece> ennemies, Echiquier e) {
   LinkedList<Coord> moves = e.allClassicMoves(this);
   LinkedList<IPiece> checkingPieces = Utils.getAllAttackingPiece(coord, ennemies, e);
   moves.removeIf(c -> isAttacked(c, ennemies, e));
   moves.removeIf(c -> isAttackedPath(c, checkingPieces));
   return moves;
}

/**
 * Verifie si une coordonnée est dans le chemin d'une piece qui attaque le roi
 * @param attackingPiece les pieces ennemies
 * @return liste de cases possiblement attaqué protégé par le roi (car en travers du chemin)
 */
private boolean isAttackedPath(Coord c, List<IPiece> attackingPiece){
   for(IPiece p : attackingPiece) {
      Coord cS = p.getCoord().clone();
      if(!Coord.isStraightPath(coord, cS))
         continue;

      Coord cP = Coord.getPrimaryMove(cS, coord);
      cS.add(cP); //on ajoute une premiere fois
      if(Utils.getPathToBorder(cS, cP).contains(c))
         return true;
   }
   return false;
}
```



Pieces

Roi 3/4

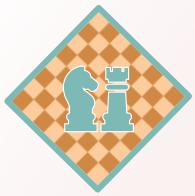
```java
/** {@inheritDoc} */
@Override
public String getSymbole() {
    return "R";
}

/** le roi est sensible au attaque */
@Override
public boolean estSensible() {
    return true;
}

/**
 * il faut retirer au mouvement du roi chaque coordonnées susceptible
 * d'etre attaqué.
 * */
@Override
public LinkedList<Coord> getAllMoves(Coord sC, List<IPiece> ennemies, Echiquier e) {
    LinkedList<Coord> moves = e.allClassicMoves(this);
    LinkedList<IPiece> checkingPieces = Utils.getAllAttackingPiece(coord, ennemies, e);
    moves.removeIf(c -> isAttacked(c, ennemies, e));
    moves.removeIf(c -> isAttackedPath(c, checkingPieces));
    return moves;
}
```



Pieces

Roi 4/4

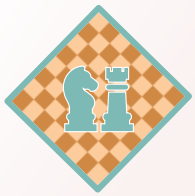
```java
/**
 * Verifie si une coordonnée est dans le chemin d'une piece qui attaque le roi
 * @param attackingPiece les pieces ennemies
 * @return liste de cases possiblement attaqué protégé par le roi (car en travers du chemin)
 */
private boolean isAttackedPath(Coord c, List<IPiece> attackingPiece){
   for(IPiece p : attackingPiece) {
      Coord cS = p.getCoord().clone();
      if(!Coord.isStraightPath(coord, cS))
         continue;

      Coord cP = Coord.getPrimaryMove(cS, coord);
      cS.add(cP); //on ajoute une premiere fois
      if(Utils.getPathToBorder(cS, cP).contains(c))
         return true;
   }
   return false;
}

 /** le roi ne peut pas etre cloué */
   @Override
   protected boolean isPinned(Coord sC, Echiquier e) {
      return false;
   }

   /** le roi ne peut pas mater */
   @Override
   public boolean canHoldEndGame() {
      return false;
   }
}
```



Pieces

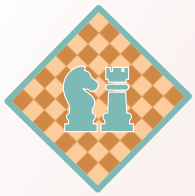
Dame

```java
package pieces;

import echiquier.Coord;
import echiquier.Couleur;

import static java.lang.Math.abs;

/**
 * La dame est la pièce la plus puissante du jeu.
 * capable de se mouvoir en ligne droite, verticalement, horizontalement,
 * et diagonalement, sur un nombre quelconque de cases inoccupées.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Dame  extends Piece{
   /** Constructeur d'une dame */
   public Dame(Coord coord, Couleur c) {
      super(c, coord);
   }

   /** {@inheritDoc} */
   @Override
   public boolean estPossible(Coord c) {
      int varX = abs(coord.getX()-c.getX());
      int varY = abs(coord.getY()-c.getY());
      return (varY >= 1 && varY==varX) || (varX > 0 && varY == 0) ||
            (varY > 0 && varX == 0);
   }

   /** {@inheritDoc} */
   @Override
   public String getSymbole() {
      return "D" ;
   }
}
```



Pieces
Tour

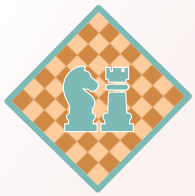
```java
package pieces;

import echiquier.Coord;
import echiquier.Couleur;

import static java.lang.Math.abs;

/**
 * Represente les Tours dans un jeu d'echec.
 * La tour peut se déplacer horizontalement ou verticalement.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Tour extends Piece{
   /** Constructeur d'une tour */
   public Tour(Coord coord, Couleur c) {
      super(c, coord);
   }

   /** {@inheritDoc} */
   @Override
   public boolean estPossible(Coord c) {
      int varX = abs(coord.getX()-c.getX());
      int varY = abs(coord.getY()-c.getY());
      return (varX > 0 && varY == 0) || (varY > 0 && varX == 0);
   }

   /** {@inheritDoc} */
   @Override
   public String getSymbole() {
      return "T";
   }
}
```



Pieces

Fou

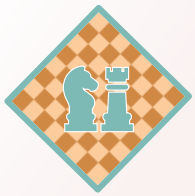
```java
package pieces;

import echiquier.Coord;
import echiquier.Couleur;

import static java.lang.Math.abs;

/**
 * represente le fou dans les echecs
 * il se deplace uniquement en diagonale.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Fou extends Piece{

   /** Constructeur d'un fou */
   public Fou(Coord coord, Couleur c) {
      super(c, coord);
   }

   /** {@inheritDoc} */
   @Override
   public boolean estPossible(Coord c) {
      int varX = abs(coord.getX()-c.getX());
      int varY = abs(coord.getY()-c.getY());
      return (varY >= 1 && varX==varY);
   }

   /** {@inheritDoc} */
   @Override
   public String getSymbole() {
      return "F";
   }

   /** le fou ne peut pas mater */
   @Override
   public boolean canHoldEndGame() {
      return false;
   }
}
```



Pieces

Cavalier

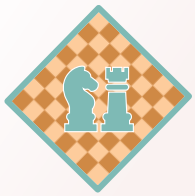
```java
package pieces;

import echiquier.Coord;
import echiquier.Couleur;

import static java.lang.Math.abs;

/**
 * cavalier ou le cheval dans le jeu d'echec.
 * Le déplacement du cavalier est original.
 * Il se déplace en L, c'est-à-dire de deux cases dans une direction
 * combinées avec une case perpendiculairement.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Cavalier extends Piece{

    /** Constructeur d'un cavalier */
    public Cavalier(Coord coord, Couleur c) {
        super(c, coord);
    }

    /** {@inheritDoc} */
    @Override
    public boolean estPossible(Coord c) {
        int varX = abs(coord.getX()-c.getX());
        int varY = abs(coord.getY()-c.getY());
        return (varX==1 && varY==2)||(varX==2 && varY==1);
    }

    /** {@inheritDoc} */
    @Override
    public String getSymbole() {
        return "C";
    }

    /** le cavalier ne peut pas mater*/
    @Override
    public boolean canHoldEndGame() {
        return false;
    }
}
```



Pieces

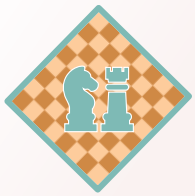
Vide

```java
package pieces;

import echiquier.Coord;
import echiquier.Couleur;

/**
 * Classe representant les pieces vides.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Vide extends Piece{

   /** Constructeur d'une pièce vide */
   public Vide(Coord coord, Couleur c) {
      super(c, coord);
   }

   /** {@inheritDoc} */
   @Override
   public boolean estPossible(Coord c) {
      return false;
   }

   /** {@inheritDoc} */
   @Override
   public String getSymbole() {
      return " ";
   }

   /** La piece vide est effectivement vide*/
   @Override
   public boolean estVide() {
      return true;
   }
}
```



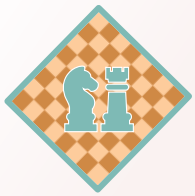
Joueur
ChessJoueur

```java
package Joueur;

import echiquier.Coord;
import echiquier.Couleur;
import echiquier.Echiquier;
import echiquier.IChessJoueur;
import echiquier.IPiece;

import java.util.List;

/**
 * {@inheritDoc}
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 * */
public abstract class ChessJoueur implements IChessJoueur {
   /** la couleur du joueur */
   protected Couleur couleur;

   /** Constructeur de joueur */
   public ChessJoueur(Couleur couleur){
      this.couleur = couleur;
   }

   /** {@inheritDoc} */
   public final Couleur getCouleur(){
      return couleur;
   }

   /** {@inheritDoc} */
   public abstract String getCoup(Echiquier e, List<IPiece> allys, List<IPiece> enemies, Coord sC);

   /** {@inheritDoc} */
   public abstract String coupToString(String coup);

   /** {@inheritDoc} */
   public abstract boolean acceptDraw();
}
```



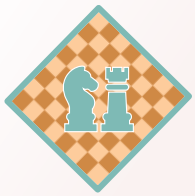
Joueur
ChessIHM 1/3

```java
package Joueur;

import echiquier.Coord;
import echiquier.*;

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

/**
 * Interface Homme Machine permettant que recuperer les input de l'utilisateur
 * puis les passer par un multitude de filtre.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class chessIHM {

   /** Seul 3 modes de jeu sont acceptés.*/
   private static boolean isValid(String mode){
      switch (mode) {
         case "pp": //joueur contre joueur
         case "pi": // joueur contre ordi
         case "ii": // ordi contre ordi
            return true;
         default:return false;
      }
   }

   /** Renvoie le mode choisi par l'utilisateur */
   public static String getMode(){
      Scanner sc = new Scanner(System.in);
      System.out.print("Votre mode : > ");
      String mode = sc.nextLine();
      while(!isValid(mode)){
         System.out.print("#> ");
         mode = sc.nextLine();
      }
      return mode;
   }
```



Joueur
ChessIHM 2/3

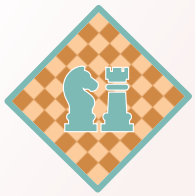
```java
/**
 * Renvoie le coup de l'utilisateur.
 * Celui-ci passe par deux verification :
 * - la syntax
 * - la semantique
 * La methode prend en compte la nulle et l'abandon
 * @param j le joueur
 * @param e l'echiquier
 * @return  le coup valide
 */
public static String getCoup(IChessJoueur j, Coord sC, List<IPiece> ennemies, Echiquier e){
   Scanner sc = new Scanner(System.in);
   String coup = sc.nextLine();

   while(!(coup.equals("nulle") || coup.equals("abandon")) &&
         !(syntaxValid(coup) && semanticValid(coup, j, sC, ennemies, e))){
      System.out.print("#> ");
      coup = sc.nextLine();
   }
   return coup;
}

/**
 * Vérifie si la sémantique du coup est valide.
 * A utiliser après inputValid.
 * @param coup  le coup
 * @param j     le joueur
 * @param e     l'echiquier
 * @return      le coup est correcte semantiquement
 */
private static boolean semanticValid(String coup, IChessJoueur j, Coord sC,
                        List<IPiece> ennemies, Echiquier e){
   int mid = coup.length() / 2;
   Coord cS = new Coord(coup.substring(0,mid));
   Coord cF = new Coord(coup.substring(mid));
   if(!Echiquier.inBound(cS) && !Echiquier.inBound(cF))
      return false;
   IPiece p = e.getPiece(cS);
   return Couleur.isRightColor(p, j.getCouleur()) &&
         p.getAllMoves(sC, ennemies, e).contains(cF);
}
```



Joueur

ChessIHM 3/3

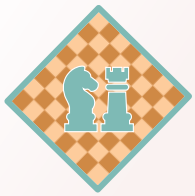
```
    /** Si l'input respecte le format d'un coup ("a2a3", b4c2", un pattern identique a chaque coup)*/
    private static boolean syntaxValid(String coup){
        if(coup.length() != 4)
            return false;

        return Character.isLetter(coup.charAt(0)) &&
                Character.isDigit(coup.charAt(1)) &&
                Character.isLetter(coup.charAt(2)) &&
                Character.isDigit(coup.charAt(3));
    }

    /**
     * Renvoie si un joueur propose accepte la proposition de nulle de son adversaire.
     * @param j le joueur adverse
     * @return  le joueur adverse accepte la nulle
     */
    public static boolean acceptDraw(IChessJoueur j) {
        Scanner sc = new Scanner(System.in);
        System.out.print("les " + j.getCouleur() + "s, acceptez vous la nulle ? (O/N) : > ");
        String answer = sc.nextLine();
        while (true) {
            switch (answer) {
                case "O": return true;
                case "N": return false;
            }
            System.out.print("#> ");
            answer = sc.nextLine();
        }
    }
}
```



Joueur

ChessJoueurIA 1/3

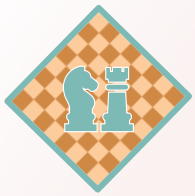
```
package Joueur;

import echiquier.Coord;
import echiquier.Couleur;
import echiquier.Echiquier;
import echiquier.IPiece;

import java.util.HashMap;
import java.util.List;
import java.util.Random;

/**
 * Cette clase represente une Ia pour les echecs sans intelligence, ces coups sont calculés de facon
 * aléatoire.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class ChessJoueurIA extends ChessJoueur {

    /**
     * {@inheritDoc}
     */
    public ChessJoueurIA(Couleur couleur) {
        super(couleur);
    }
```



Joueur

ChessJoueurIA 2/3

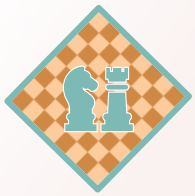
```java
/**
 * {@inheritDoc}
 * l'ordinateur cherche un coup valide et le renvoie
 */
@Override
public String getCoup(Echiquier e, List<IPiece> allys, List<IPiece> enemies, Coord sC) {
    pause();

    HashMap<IPiece, List<Coord>> allPossibleMoves = new HashMap<>();
    for(IPiece p : allys){
        allPossibleMoves.put(p, p.getAllMoves(sC, enemies, e));
    }

    Random rand = new Random();

    IPiece choosenP = allys.get(rand.nextInt(allys.size()));

    int cmpt = 0;   //securité contre les boucles infinis
    // choisis une piece au hasard, si celle-ci n'a pas
    // de mouvement disponible, l'ia continue.
    while(allPossibleMoves.get(choosenP).isEmpty()) {
        choosenP = allys.get(rand.nextInt(allys.size()));
        if(cmpt == 100)
            return "abandon";
        cmpt++;
    }


    Coord cS = choosenP.getCoord();

    List<Coord> allMoves = allPossibleMoves.get(choosenP);
    Coord cF = allMoves.get(rand.nextInt(allMoves.size()));

    return cS.toString() + cF.toString();
}
```



Joueur

ChessJoueurIA 3/3

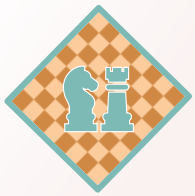
```
    /** il y a une pause quand l'IA joue */
    private void pause() {
        try{
            Thread.sleep(50);
        } catch (Exception ignored){ }
    }

    /** renvoie un chaine de caractere du coup de l'Ia (pour un affichage clair)
     * @param coup le coup de l'Ia
     * */
    @Override
    public String coupToString(String coup) {
        return "> " + coup;
    }

    /** l'Ia accepte toujours la nulle*/
    @Override
    public boolean acceptDraw() {
        return true;
    }
}
```



Joueur

ChessJoueurHumain

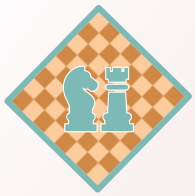
```java
package Joueur;

import echiquier.Coord;
import echiquier.Couleur;
import echiquier.Echiquier;
import echiquier.IPiece;

import java.util.List;

/**
 * Cette classe represente un joueur d'echec Humain.
 * celui-ci s'appuie sur une IHM pour communiquer avec le programme.
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class ChessJoueurHumain extends ChessJoueur {

    /** Constructeur d'un joueur humain */
    public ChessJoueurHumain(Couleur couleur) {
        super(couleur);
    }

    /**
     * Un joueur Humain doit faire appel a une ihm pour donner son coup
     * @see chessIHM
     * */
    @Override
    public String getCoup(Echiquier e, List<IPiece> allys, List<IPiece> ennemies, Coord sC) {
        return chessIHM.getCoup(this, sC, ennemies, e);
    }

    /** le coup est deja affiché en invite de commande*/
    @Override
    public String coupToString(String coup) {
        return "";
    }

    /**
     * fais appel à une ihm pour accepter la proposition de nulle
     * @see chessIHM
     */
    @Override
    public boolean acceptDraw() {
        return chessIHM.acceptDraw(this);
    }
}
```



Joueur

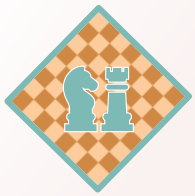
FabChessJoueur

```
package Joueur;

import echiquier.Couleur;
import echiquier.IFabChessJoueur;
import echiquier.IChessJoueur;

/**
 * {@inheritDoc}
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class FabChessJoueur implements IFabChessJoueur {

   /**
    * {@inheritDoc}
    */
   @Override
   public IChessJoueur getJoueur(char type, Couleur couleur) {
      if (type == 'I') {
         return new ChessJoueurIA(couleur);
      }
      return new ChessJoueurHumain(couleur);
   }
}
```



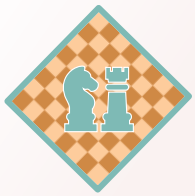
Appli
Application 1/3

```
package appli;

import Joueur.FabChessJoueur;

import Joueur.chessIHM;
import echiquier.*;
import pieces.*;

import java.util.List;

import static echiquier.Couleur.*;

/**
 * @author Stefan Radovanovic, Yannick Li, Zakaria Sellam
 */
public class Application {

   /** message de debut de partie*/
   private static final String START = "mode de jeux : \n" +
         "- Player vs Player (\"pp\") \n" +
         "- Plaver vs Ia (\"pi\") \n" +
         "- Ia vs Ia (\"ii\") \n\n" +
         "Règles : \n" +
         "- pour proposer un match nul : \"nulle\" \n" +
         "- pour abandonner : \"abandon\"";

   /** reference de joueur et de couleur*/
   private static IChessJoueur actif, passif;
   private static Couleur cActif, cPassif;

   /** message de fin de partie*/
   private static String endGameMessage;

   /** le dernier coup joué*/
   private static String coup;
```



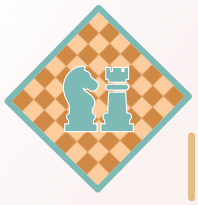
Appli
Application 2/3

```
/** changement de tour, le joueur passif devient actif et pourra jouer*/
   private static void swithJoueur(){
      IChessJoueur tmp = actif;
      actif = passif;
      passif = tmp;

      cActif = actif.getCouleur();
      cPassif = passif.getCouleur();
   }

/** regroupement de toutes les conditions de fin de partie dans les echecs*/
public static boolean partieEnd(Coord sC, List<IPiece> allys, List<IPiece> ennemies, Echiquier e){
   if(Regle.isStaleMate(sC, allys, ennemies, e)) {
      endGameMessage = "Egalité par pat";
      return true;
   }
   if(Regle.checkForMate(sC, allys, ennemies, e)){
      endGameMessage ="Les "+ actif.getCouleur() + "S ont perdu";
      return true;
   }
   if(Regle.impossibleMat(allys, ennemies)){
      endGameMessage = "NULLE";
      return true;
   }
   return false;
}

/** gestion de l'abandon et de la proposition de nulle*/
private static boolean isEndByPlayer(Echiquier e, List<IPiece> allys, List<IPiece> enemies, Coord sC){
   switch (coup){
      case"nulle":
         if(passif.acceptDraw()){
            endGameMessage = "match nul par accord !";
            return true;
         }
         System.out.print("votre demande de nulle n'a pas été accepté, jouez : > ");
         coup = actif.getCoup(e, allys, enemies, sC);
         return false;
      case"abandon" :
         endGameMessage = "les " + cPassif + "s gagnent par forfait !";
         return true;
      default: return false;
   }
}
```



Appli
Application 3/3

```
    public static void main(String[] args) {
        System.out.println(START);
        String mode = chessIHM.getMode();
        Echiquier e = new Echiquier(new FabPiece(),mode, new FabChessJoueur());

        actif = e.getJoueur(BLANC);
        passif = e.getJoueur(NOIR);

        cActif = actif.getCouleur();
        cPassif = passif.getCouleur();

        System.out.println(e.toString());

        while(true){
            Coord sC = e.locateSensiblePiece(cActif);
            List<IPiece> piecesActif = e.getPieceFromColor(cActif);
            List<IPiece> piecesPassif = e.getPieceFromColor(cPassif);

            if(partieEnd(sC, piecesActif, piecesPassif, e))
                break;

            System.out.print(cActif + " joue ");
            coup = actif.getCoup(e, piecesActif, piecesPassif, sC);

            if(isEndByPlayer(e, piecesActif, piecesPassif, sC))
                break;

            System.out.println(actif.coupToString(coup));

            e.deplacer(coup);

            e.checkForPromote(cActif);

            System.out.println(e.toString());

            swithJoueur();
        }

        System.out.println(endGameMessage);
    }

}
```



Intégrations d'autres règles

Les règles de déplacements de pièces comme le rook peuvent être implémentés en modifiant la classe des pièces concernées.

Les règles de Jeu comme la prise en passant ou la nulle par répétition nécessite que l'on stocke les coups joués lors de la partie (création d'une classe partie).

Pour augmenter l'intelligence des ordinateurs, on pourrait intégrer un système de points à la capture où il choisira le coup rapportant le plus de points, ce qui nécessiterai que le programme attribue des points de captures de pièces, de mise en échec, de clouage de pièce, etc...

Les règles comme la règle des 50 coups nécessite de programmer un algorithme de recherche d'échecs et mat qui pourrait aussi être utile pour augmenter le niveau des ordinateurs.

Bilan

Nous avons eu quelques problèmes au début à faire du code générique mais nous avons fait du mieux que nous avons pu. Il a été difficile de programmer les règles du jeu d'échecs car le jeu est très complexe, il fallait donc réfléchir à des moyens de trouver les pièces qui menacent le roi ainsi que ceux qui peuvent le défendre.

Nous avons beaucoup réfléchi aux dépendances de notre programme et nous sommes plutôt fier de ceux-ci. Le projet a été très instructif pour apprendre à faire les classes abstraites, les interfaces ainsi que leurs implémentations.

Il reste encore beaucoup de règles que nous n'avons pas essayé ou réussi à implémenter, ainsi que la détection de nulles spécifiques. Il serait peut-être plus intéressant de faire ce jeux d'échecs mais avec une interface graphique pour rendre le résultat plus visuel.