“BoardContext.java”

```
package board.boardalgo;

import board.*;

public class BoardContext {
	private BoardStrategy bs;
	private Board b;
	private Coord coord;

	public BoardContext (BoardStrategy strategy, Board b_, Coord c) {
		bs = strategy;
		b = b_;
		coord = c;
	}

	public String print (int n, boolean slide) {return bs.print(b.readNumber(coord), slide);}
	public void setStrategy (BoardStrategy strategy) {bs = strategy;}
	public void setCoord (Coord coord_) {coord = coord_;}
}
```

“BoardStrategy.java”

```
package board.boardalgo;

import game.GameConstants;

public interface BoardStrategy extends GameConstants {
	String print (int n, boolean onOwnSlide);
}
```

“BoardStrategyDEFAULT.java”

```
package board.boardalgo;

public class BoardStrategyDEFAULT implements BoardStrategy {
	public String print (int n, boolean onOwnSlide) { // n - numéro de pion (1, 2, 3 ou 4)
		return E + "*" + N;
	}
}
```

“BoardStrategyBLUE.java”

```
package board.boardalgo;

public class BoardStrategyBLUE implements BoardStrategy {
	public String print (int n, boolean onOwnSlide) { // n - numéro de pion (1, 2, 3 ou 4)
		if(onOwnSlide) return BK + Integer.toString(n) + N;
		else return B + Integer.toString(n) + N;
	}
}
```

“BoardStrategyGREEN.java”

```
package board.boardalgo;

public class BoardStrategyGREEN implements BoardStrategy {
	public String print (int n, boolean onOwnSlide) { // n - numéro de pion (1, 2, 3 ou 4)
		if(onOwnSlide) return BK + Integer.toString(n) + N;
		else return G + Integer.toString(n) + N;
	}
}
```

“BoardStrategyRED.java”

```
package board.boardalgo;

public class BoardStrategyRED implements BoardStrategy {
	public String print (int n, boolean onOwnSlide) { // n - numéro de pion (1, 2, 3 ou 4)
		if(onOwnSlide) return BK + Integer.toString(n) + N;
		else return R + Integer.toString(n) + N;
	}
}
```

“BoardStrategyYELLOW.java”

```
package board.boardalgo;

public class BoardStrategyYELLOW implements BoardStrategy {
	public String print (int n, boolean onOwnSlide) { // n - numéro de pion (1, 2, 3 ou 4)
		if(onOwnSlide) return BK + Integer.toString(n) + N;
		else return Y + Integer.toString(n) + N;
	}
}
```

“GoHomeContext.java”

```
package board.home;

import board.*;
import player.Player;

public class GoHomeContext {
	private GoHomeStrategy strategy;
	private Coord coord;
	private Board board;
	private Player[] p;

	public GoHomeContext (GoHomeStrategy strategy_, Coord start, Board b, Player[] p_) {
		strategy = strategy_;
		coord = start;
		board = b;
		p = p_;
	}

	public void setStrategy (GoHomeStrategy strategy_) {
		strategy = strategy_;
	}

	public void operation () {
		strategy.algorithm(coord, p, board);
	}
}
```

“GoHomeStrategy.java”

```
package board.home;

import board.*;
import player.Player;
import game.GameConstants;

public interface GoHomeStrategy extends GameConstants {
	void algorithm (Coord start, Player[] p, Board b);
}
```

“GoHomeStrategyDEFAULT.java”

```
package board.home;

import board.*;
import player.Player;

public class GoHomeStrategyDEFAULT implements GoHomeStrategy {
	public void algorithm (Coord start, Player[] p, Board b) {}
}
```

“GoHomeStrategyBLUE.java”

```
package board.home;

import board.*;
import player.Player;

public class GoHomeStrategyBLUE implements GoHomeStrategy {
	public void algorithm (Coord start, Player[] p, Board b) {
		int x = start.getX();

		if(start.equals(BSHORT_START)) {
			for(int i = 0 ; i < 4 ; i++) {
				if(b.getSituation()[x-i][0] != 0) {
					b.goSTART(b.convertIntToPawn(p, b.getSituation()[x-i][0]));
				}
			}
		}
		else if(start.equals(BLONG_START)) {
			for(int i = 0 ; i < 5 ; i++) {
				if(b.getSituation()[x-i][0] != 0) {
					b.goSTART(b.convertIntToPawn(p, b.getSituation()[x-i][0]));
				}
			}
		}
	}
}
```

“GoHomeStrategyGREEN.java”

```
package board.home;

import board.*;
import player.Player;

public class GoHomeStrategyGREEN implements GoHomeStrategy {
	public void algorithm (Coord start, Player[] p, Board b) {
		int x = start.getX();

		if(start.equals(GSHORT_START)) {
			for(int i = 0 ; i < 4 ; i++) {
				if(b.getSituation()[x+i][15] != 0) {
					b.goSTART(b.convertIntToPawn(p, b.getSituation()[x+i][15]));
				}
			}
		}
		else if(start.equals(GLONG_START)){
			for(int i = 0 ; i < 5 ; i++) {
				if(b.getSituation()[x+i][15] != 0) {
					b.goSTART(b.convertIntToPawn(p, b.getSituation()[x+i][15]));
				}
			}
		}
	}
}
```

“GoHomeStrategyRED.java”

```
package board.home;

import board.*;
import player.Player;

public class GoHomeStrategyRED implements GoHomeStrategy {
	public void algorithm (Coord start, Player[] p, Board b) {
		int y = start.getY();

		if(start.equals(RSHORT_START)) {
			for(int i = 0 ; i < 4 ; i++) {
				if(b.getSituation()[15][y-i] != 0) {
					b.goSTART(b.convertIntToPawn(p, b.getSituation()[15][y-i]));
				}
			}
		}
		else if(start.equals(RLONG_START)){
			for(int i = 0 ; i < 5 ; i++) {
				if(b.getSituation()[15][y-i] != 0) {
					b.goSTART(b.convertIntToPawn(p, b.getSituation()[15][y-i]));
				}
			}
		}
	}
}
```

“GoHomeStrategyYELLOW.java”

```
package board.home;

import board.*;
import player.Player;

public class GoHomeStrategyYELLOW implements GoHomeStrategy {
	public void algorithm (Coord start, Player[] p, Board b) {
		int y = start.getY();

		if(start.equals(YSHORT_START)) {
			for(int i = 0 ; i < 4 ; i++) {
				if(b.getSituation()[0][y+i] != 0) {
					b.goSTART(b.convertIntToPawn(p, b.getSituation()[0][y+i]));
				}
			}
		}
		else if(start.equals(YLONG_START)){
			for(int i = 0 ; i < 5 ; i++) {
				if(b.getSituation()[15][y-i] != 0) {
					b.goSTART(b.convertIntToPawn(p, b.getSituation()[15][y-i]));
				}
			}
		}
	}
}
```

“GoStartContext.java”

```
package board.start;

import player.Player;
import board.*;

public class GoStartContext {
	private GoStartStrategy strategy;
	private Player[] p;
	private int color;
	private Board b;

	public GoStartContext (GoStartStrategy strategy_, Player[] p_, Board b_, int color_) {
		strategy = strategy_;
		p = p_;
		color = color_;
		b = b_;
	}

	public void setStrategy (GoStartStrategy strategy_) {
		strategy = strategy_;
	}

	public void operation () {
		strategy.algorithm(p, color, b);
	}
}
```

“GoStartStrategy.java”

```
package board.start;

import player.Player;
import board.Board;
import game.GameConstants;

public interface GoStartStrategy extends GameConstants{
	void algorithm (Player[] p, int color, Board b);
}
```

"GoStartStrategyDEFAULT.java"

```
package board.start;

import player.Player;
import board.Board;

public class GoStartStrategyDEFAULT implements GoStartStrategy {
	public void algorithm (Player[] p, int color, Board b) {}
}
```

"GoStartStrategyBLUE.java"

```
package board.start;

import player.Player;
import board.Board;

public class GoStartStrategyBLUE implements GoStartStrategy {
	public void algorithm (Player[] p, int color, Board b) {
		if(!p[1].getSet()[0].getIsActive() && !p[1].getSet()[0].getIsHome()) {
			p[1].getSet()[0].setPos(BSTART);
			p[1].getSet()[0].setIsActive(true);
			b.update(p[1].getSet()[0]);
			b.update(BBASE1, 0);
		}
		else if(!p[1].getSet()[1].getIsActive() && !p[1].getSet()[1].getIsHome()) {
			p[1].getSet()[1].setPos(BSTART);
			p[1].getSet()[1].setIsActive(true);
			b.update(p[1].getSet()[1]);
			b.update(BBASE2, 0);
		}
		else if(!p[1].getSet()[2].getIsActive() && !p[1].getSet()[2].getIsHome()) {
			p[1].getSet()[2].setPos(BSTART);
			p[1].getSet()[2].setIsActive(true);
			b.update(p[1].getSet()[2]);
			b.update(BBASE3, 0);
		}
		else if(!p[1].getSet()[3].getIsActive() && !p[1].getSet()[3].getIsHome()) {
			p[1].getSet()[3].setPos(BSTART);
			p[1].getSet()[3].setIsActive(true);
			b.update(p[1].getSet()[3]);
			b.update(BBASE4, 0);
		}
	}
}
```

“GoStartStrategyGREEN.java”

```
package board.start;

import player.Player;
import board.Board;

public class GoStartStrategyGREEN implements GoStartStrategy {
	public void algorithm (Player[] p, int color, Board b) {
		if(!p[3].getSet()[0].getIsActive() && !p[3].getSet()[0].getIsHome()) {
			p[3].getSet()[0].setPos(GSTART);
			p[3].getSet()[0].setIsActive(true);
			b.update(p[3].getSet()[0]);
			b.update(GBASE1, 0);
		}
		else if(!p[3].getSet()[1].getIsActive() && !p[3].getSet()[1].getIsHome()) {
			p[3].getSet()[1].setPos(GSTART);
			p[3].getSet()[1].setIsActive(true);
			b.update(p[3].getSet()[1]);
			b.update(GBASE2, 0);
		}
		else if(!p[3].getSet()[2].getIsActive() && !p[3].getSet()[2].getIsHome()) {
			p[3].getSet()[2].setPos(GSTART);
			p[3].getSet()[2].setIsActive(true);
			b.update(p[3].getSet()[2]);
			b.update(GBASE3, 0);
		}
		else if(!p[3].getSet()[3].getIsActive() && !p[3].getSet()[3].getIsHome()) {
			p[3].getSet()[3].setPos(GSTART);
			p[3].getSet()[3].setIsActive(true);
			b.update(p[3].getSet()[3]);
			b.update(GBASE4, 0);
		}
	}
}
```

“GoStartStrategyRED.java”

```
package board.start;

import player.Player;
import board.Board;

public class GoStartStrategyRED implements GoStartStrategy {
	public void algorithm (Player[] p, int color, Board b) {
		if(!p[0].getSet()[0].getIsActive() && !p[0].getSet()[0].getIsHome()) {
			p[0].getSet()[0].setPos(RSTART);
			p[0].getSet()[0].setIsActive(true);
			b.update(p[0].getSet()[0]);
			b.update(RBASE1, 0);
		}
		else if(!p[0].getSet()[1].getIsActive() && !p[0].getSet()[1].getIsHome()) {
			p[0].getSet()[1].setPos(RSTART);
			p[0].getSet()[1].setIsActive(true);
			b.update(p[0].getSet()[1]);
			b.update(RBASE2, 0);
		}
		else if(!p[0].getSet()[2].getIsActive() && !p[0].getSet()[2].getIsHome()) {
			p[0].getSet()[2].setPos(RSTART);
			p[0].getSet()[2].setIsActive(true);
			b.update(p[0].getSet()[2]);
			b.update(RBASE3, 0);
		}
		else if(!p[0].getSet()[3].getIsActive() && !p[0].getSet()[3].getIsHome()) {
			p[0].getSet()[3].setPos(RSTART);
			p[0].getSet()[3].setIsActive(true);
			b.update(p[0].getSet()[3]);
			b.update(RBASE4, 0);
		}
	}
}
```

“GoStartStrategyYELLOW.java”

```
package board.start;

import player.Player;
import board.Board;

public class GoStartStrategyYELLOW implements GoStartStrategy {
	public void algorithm (Player[] p, int color, Board b) {
		if(!p[2].getSet()[0].getIsActive() && !p[2].getSet()[0].getIsHome()) {
			p[2].getSet()[0].setPos(YSTART);
			p[2].getSet()[0].setIsActive(true);
			b.update(p[2].getSet()[0]);
			b.update(YBASE1, 0);
		}
		else if(!p[2].getSet()[1].getIsActive() && !p[2].getSet()[0].getIsHome()) {
			p[2].getSet()[1].setPos(YSTART);
			p[2].getSet()[1].setIsActive(true);
			b.update(p[2].getSet()[1]);
			b.update(YBASE2, 0);
		}
		else if(!p[2].getSet()[2].getIsActive() && !p[2].getSet()[0].getIsHome()) {
			p[2].getSet()[2].setPos(YSTART);
			p[2].getSet()[2].setIsActive(true);
			b.update(p[2].getSet()[2]);
			b.update(YBASE3, 0);
		}
		else if(!p[2].getSet()[3].getIsActive() && !p[2].getSet()[0].getIsHome()) {
			p[2].getSet()[3].setPos(YSTART);
			p[2].getSet()[3].setIsActive(true);
			b.update(p[2].getSet()[3]);
			b.update(YBASE4, 0);
		}
	}
}
```

“Board.java”

```
package board;

import deck.*;
import game.GameConstants;
import game.cardalgo.*;
import piece.*;
import player.Player;
import board.boardalgo.*;
import board.home.*;
import board.start.*;

/** Context strategy :
* 0 - normal (NORMAL) BoardStrategyNORMAL
* 1 - rouge (RED) BoardStrategyRED
* 2 - bleu (BLUE) BoardStrategyBLUE
* 3 - jaune (YELLOW) BoardStrategyYELLOW
* 4 - vert (GREEN) BoardStrategyGREEN
*/

public class Board implements BoardParams, GameConstants {
	private BoardContext colorContext;
	private BoardStrategy colorStrategy;

	private PickContext algoContext;

	private GoHomeContext goHomeContext;
	private GoHomeStrategy goHomeStrategy;

	private GoStartContext goStartContext;
	private GoStartStrategy goStartStrategy;

	private CardFactory cardFactory = new CardFactoryDown();
	private UpDeck dropDeck = UpDeck.getInstance();
	private DownDeck pickDeck = cardFactory.createDeck();
	private Card picked;

	private int[][] situation;
	private int turn;

	public Board () {
		Init();
	}
```

```
private void Init () {
	// initialisation de situation et visual
	situation = new int[16][16];
	for(int i = 0 ; i < 16 ; i++) {
		for(int j = 0 ; j < 16 ; j++) {
			situation[i][j] = EMPTY;
		}
	}
	// par défaut context est configuré à NORMAL
	colorContext = new BoardContext(new BoardStrategyDEFAULT(), this, new Coord(0, 0));
	// mélange les cartes de pickDeck
	mixPickDeck();
}


// ** Fonctions de contrôle **

// détermine si des pions friendly sont actifs sur le board
public boolean isActiveOnBoard (int color) {return countActive(color) > 0;}

// compte le nombre de pions actifs friendly
private int countActive (int color) {
	int n = 0; // compteur

	for(int i = 0 ; i <= 15 ; i++) {
		if(situation[0][i] != 0) {
			if(readColor(new Coord(0, i)) == color && readActive(new Coord(0, i)) == 1) n++;
		}
	}
	for(int i = 1 ; i < 15 ; i++) {
		if(situation[i][15] != 0) {
			if(readColor(new Coord(i, 15)) == color && readActive(new Coord(i, 15)) == 1) n++;
		}
	}
	for(int i = 15 ; i >= 0 ; i--) {
		if(situation[15][i] != 0) {
			if(readColor(new Coord(15, i)) == color && readActive(new Coord(15, i)) == 1) n++;
		}
	}
	for(int i = 14 ; i > 0 ; i--) {
		if(situation[i][0] != 0) {
			if(readColor(new Coord(i, 0)) == color && readActive(new Coord(i, 0)) == 1) n++;
		}
	}
	return n;
}
```

```
// envoie un pion à HOME
public boolean sendHOME (Player player, Pawn o) {
	Coord old = new Coord(DEFAULT);
	old.setCoord(o.getPos());
	o.setPos(INFINITE);
	o.setIsHome(true);
	o.setIsActive(false);
	update(o);
	update(old, 0);
	if(player.countHome() == 4) return true;
	return false;
}

// envoie un pion dans START (selon Pawn)
public void goSTART (Pawn o) {
	// reset l'ancienne position à 0
	update(o.getPos(), 0);
	// envoie à START
	switch(o.getColor()) {
		case 1:	switch(o.getNumber()) {
				case 1:	o.setIsActive(false);
					o.setPos(RBASE1);
					update(o);
					break;
				case 2:	o.setIsActive(false);
					o.setPos(RBASE2);
					update(o);
					break;
				case 3:	o.setIsActive(false);
					o.setPos(RBASE3);
					update(o);
					break;
				case 4:	o.setIsActive(false);
					o.setPos(RBASE4);
					update(o);
					break;
			}
			break;
		case 2:	switch(o.getNumber()) {
				case 1:	o.setIsActive(false);
					o.setPos(BBASE1);
					update(o);
					break;
				case 2:	o.setIsActive(false);
					o.setPos(BBASE2);
					update(o);
					break;
```

```
                case 3:  o.setIsActive(false);
                         o.setPos(BBASE3);
                         update(o);
                         break;
                case 4:  o.setIsActive(false);
                         o.setPos(BBASE4);
                         update(o);
                         break;
            }
            break;
case 3:     switch(o.getNumber()) {
                case 1:  o.setIsActive(false);
                         o.setPos(YBASE1);
                         update(o);
                         break;
                case 2:  o.setIsActive(false);
                         o.setPos(YBASE2);
                         update(o);
                         break;
                case 3:  o.setIsActive(false);
                         o.setPos(YBASE3);
                         update(o);
                         break;
                case 4:  o.setIsActive(false);
                         o.setPos(YBASE4);
                         update(o);
                         break;
            }
            break;
case 4:     switch(o.getNumber()) {
                case 1:  o.setIsActive(false);
                         o.setPos(GBASE1);
                         update(o);
                         break;
                case 2:  o.setIsActive(false);
                         o.setPos(GBASE2);
                         update(o);
                         break;
                case 3:  o.setIsActive(false);
                         o.setPos(GBASE3);
                         update(o);
                         break;
                case 4:  o.setIsActive(false);
                         o.setPos(GBASE4);
                         update(o);
                         break;
            }
```

```
                    break;
        }
        o.setCount(0);
    }

    // si sur slide, envoie les pions occupants à START - point d'entrée pour stratégie GoHome
    public void goSTART(Player[] p, Coord start) {
        goHomeContext = new GoHomeContext(setContextStrategy3(start), start, this, p);
        goHomeContext.operation();
    }

    // retourne situation
    public int[][] getSituation () {return situation;}

    // détermine si un pion dans un tableau est sur SA case départ
    public boolean isOnOwnStart (Pawn[] pset, int color) {
        boolean ok = false;

        for(Pawn o : pset) {
            if(isOnOwnStart2(o, color)) {
                ok = true;
                break;
            }
        }
        return ok;
    }

    // détermine si un pion est sur SA case départ (surdéfinition)
    public boolean isOnOwnStart2 (Pawn o, int color) {
        switch(color) {
            case 1:   if(o.getPos().equals(RSTART)) return true;
                      break;
            case 2:   if(o.getPos().equals(BSTART)) return true;
                      break;
            case 3:   if(o.getPos().equals(YSTART)) return true;
                      break;
            case 4:   if(o.getPos().equals(GSTART)) return true;
                      break;
        }
        return false;
    }

    // détermine si une case est libre
    public boolean isFree (Coord coord) {return situation[coord.getX()][coord.getY()] == 0;}
```

```
// détermine si une case départ est occupée par même couleur
public boolean isStartFree (int color) {
	switch(color) {
		case 1:	if(readParams(RSTART) != 0) {
				if(readColor(RSTART) != 1) return true;
			}
			break;
		case 2:	if(readParams(BSTART) != 0) {
				if(readColor(BSTART) != 2) return true;
			}
			break;
		case 3:	if(readParams(YSTART) != 0) {
				if(readColor(YSTART) != 3) return true;
			}
			break;
		case 4:	if(readParams(GSTART) != 0) {
				if(readColor(GSTART) != 4) return true;
			}
			break;
		default:	return false;
	}
	return false;
}

// détermine si une case est occupée par pion de même couleur
public boolean isFriendly (Pawn o, Coord coord) {
	if(!isFree(coord)) {
		return readColor(coord) == o.getColor();
	}
	return false;
}

// détermine si une case est occupée par un adversaire
public boolean isOpponent (Pawn o, Coord dest) {
	if(!isFree(dest)) {
		return readColor(dest) != o.getColor();
	}
	return false;
}

// détermine si coordonnée est un "friendly slide"
public boolean isFriendlySlide (Pawn o, Coord dest) {
	switch(o.getColor()) {
		case 1:	if(dest.equals(RSHORT_START) || dest.equals(RLONG_START)) return true;
			break;
		case 2:	if(dest.equals(BSHORT_START) || dest.equals(BLONG_START)) return true;
			break;
```

```
            case 3:   if(dest.equals(YSHORT_START) || dest.equals(YLONG_START)) return true;
                      break;
            case 4:   if(dest.equals(GSHORT_START) || dest.equals(GLONG_START)) return true;
                      break;
        }
        return false;
    }

    // détermine si coordonnée est un début de slide quelconque
    public boolean isOnSlide(Coord coord) {
        return (coord.equals(RSHORT_START) || coord.equals(RLONG_START) ||
        coord.equals(BSHORT_START) || coord.equals(BLONG_START) ||
        coord.equals(YSHORT_START) || coord.equals(YLONG_START) ||
        coord.equals(GSHORT_START) || coord.equals(GLONG_START));
    }

    // détermine si coordonnée est un SAFETY
    public boolean isSafety (Coord coord, int color) {
        switch(color) {
            case 1:   return coord.equals(RSAFETY1) || coord.equals(RSAFETY2) || coord.equals(RSAFETY3) ||
                      coord.equals(RSAFETY4) || coord.equals(RSAFETY5);
            case 2:   return coord.equals(BSAFETY1) || coord.equals(BSAFETY2) || coord.equals(BSAFETY3) ||
                      coord.equals(BSAFETY4) || coord.equals(BSAFETY5);
            case 3:   return coord.equals(YSAFETY1) || coord.equals(YSAFETY2) || coord.equals(YSAFETY3) ||
                      coord.equals(YSAFETY4) || coord.equals(YSAFETY5);
            case 4:   return coord.equals(GSAFETY1) || coord.equals(GSAFETY2) || coord.equals(GSAFETY3) ||
                      coord.equals(GSAFETY4) || coord.equals(GSAFETY5);
        }
        return false;
    }

    // place un pion sur la case départ - point d'entrée pour la stratégie GoStart
    public void getOnStart (Player[] p, int color) {
        goStartContext = new GoStartContext(setContextStrategy4(color), p, this, turn);
        goStartContext.operation();
    }

    // déplacement du pion (point d'entrée pour la stratégie algo)
    public void movePawn () {algoContext.operation();}

    // assigne la couleur en cours pour prochain joueur
    public void setTurn () {
        turn++;
        if(turn > 4) turn = 1;
    }
```

```
// idem (surdéfinition)
public void setTurn (int n) {turn = n;}

// renvoie le tour
public int getTurn () {return turn;}

// retourne coordonnées d'un pion selon couleur et numéro
public Coord getCoordFromColNum (Player[] p, int color, int number) {
	int n = (color * 1000) + (number * 100) + 10;
	for(int i = 0 ; i < 16 ; i++) {
		for(int j = 0; j < 16 ; j++) {
			if(situation[i][j] == n) return new Coord(i, j);
		}
	}
	return new Coord(); // x = -1 et y = -1
}

// retourne cas de départ selon la couleur
public Coord getStartingPos (int color) {
	Coord coord = new Coord();

	switch(color) {
		case 1:	coord.setCoord(RSTART);
			break;
		case 2:	coord.setCoord(BSTART);
			break;
		case 3:	coord.setCoord(YSTART);
			break;
		case 4:	coord.setCoord(GSTART);
			break;
		default:	break;
	}
	return coord;
}

// retourne la couleur de la slide selon coordonnée
public int getSlideByCoord (Coord coord) {
	int x = coord.getX();
	int y = coord.getY();

	if(coord.equals(RSAFETY1) || coord.equals(RSAFETY2) || coord.equals(RSAFETY3) ||
	coord.equals(RSAFETY4) || coord.equals(RSAFETY5)) return 1;
	else if(coord.equals(BSAFETY1) || coord.equals(BSAFETY2) || coord.equals(BSAFETY3) ||
	coord.equals(BSAFETY4) || coord.equals(BSAFETY5)) return 2;
	else if(coord.equals(YSAFETY1) || coord.equals(YSAFETY2) || coord.equals(YSAFETY3) ||
	coord.equals(YSAFETY4) || coord.equals(YSAFETY5)) return 3;
	else if(coord.equals(GSAFETY1) || coord.equals(GSAFETY2) || coord.equals(GSAFETY3) ||
```

```
        coord.equals(GSAFETY4) || coord.equals(GSAFETY5)) return 4;
        else if(x == 15) {
            if((y > 1 && y < 7) || (y > 10 && y < 15)) return 1;
        }
        else if(y == 0) {
            if((x > 1 && x < 7) || (x > 10 && x < 15)) return 2;
        }
        else if(x == 0) {
            if((y > 0 && y < 5) || (y > 8 && y < 14)) return 3;
        }
        else if(y == 15) {
            if((x > 0 && x < 5) || (x > 8 && x < 14)) return 4;
        }
        return 0;
    }

    // détermine si une coordonnée se trouve sur une slide de même couleur
    public boolean isCoordOnOwnSlide (Coord coord) {
        return readColor(coord) == getSlideByCoord(coord);
    }

    // ** Définitions des stratégies **
    // configure la stratégie de couleur
    private BoardStrategy setContextStrategy1 (Coord coord) {
        switch(readColor(coord)) {
            case 0:   colorStrategy = new BoardStrategyDEFAULT();
                      break;
            case 1:   colorStrategy = new BoardStrategyRED();
                      break;
            case 2:   colorStrategy = new BoardStrategyBLUE();
                      break;
            case 3:   colorStrategy = new BoardStrategyYELLOW();
                      break;
            case 4:   colorStrategy = new BoardStrategyGREEN();
                      break;
            default:  break;
        }
        return colorStrategy;
    }

    // configure la stratégie pour goHome
    private GoHomeStrategy setContextStrategy3 (Coord start) {
        int n = 0;
        if(start.equals(RSHORT_START) || start.equals(RLONG_START)) n = 1;
        else if(start.equals(BSHORT_START) || start.equals(BLONG_START)) n = 2;
        else if(start.equals(YSHORT_START) || start.equals(YLONG_START)) n = 3;
        else if(start.equals(GSHORT_START) || start.equals(GLONG_START)) n = 4;
```

```
        else n = 5;

        switch(n) {
            case 1:   goHomeStrategy = new GoHomeStrategyRED();
                      break;
            case 2:   goHomeStrategy = new GoHomeStrategyBLUE();
                      break;
            case 3:   goHomeStrategy = new GoHomeStrategyYELLOW();
                      break;
            case 4:   goHomeStrategy = new GoHomeStrategyGREEN();
                      break;
            case 5:   goHomeStrategy = new GoHomeStrategyDEFAULT();
        }
        return goHomeStrategy;
    }

    // configure la stratégie pour goStart
    private GoStartStrategy setContextStrategy4 (int color_) {
        switch(color_) {
            case 1:   goStartStrategy = new GoStartStrategyRED();
                      break;
            case 2:   goStartStrategy = new GoStartStrategyBLUE();
                      break;
            case 3:   goStartStrategy = new GoStartStrategyYELLOW();
                      break;
            case 4:   goStartStrategy = new GoStartStrategyGREEN();
                      break;
            default:  goStartStrategy = new GoStartStrategyDEFAULT();
        }
        return goStartStrategy;
    }

    // ** Implémentation de l'interface BoardParams **

    // lecture des paramètres
    public int readParams (Coord o) {return situation[o.getX()][o.getY()];}

    public int readColor (Coord o) {
        return readParams(o) / 1000;
    }

    public int readNumber (Coord o) {
        return (readParams(o) - (readColor(o) * 1000)) / 100;
    }
```

```
public int readActive (Coord o) {
    return (readParams(o) - (readColor(o) * 1000) - (readNumber(o) * 100)) / 10;
}

public int readHome (Coord o) {
    return readParams(o) - (readColor(o) * 1000) - (readNumber(o) * 100) - (readActive(o) * 10);
}

// modification des paramètres
public void update (Pawn pawn) {
    situation[pawn.getPos().getX()][pawn.getPos().getY()] = convertPawnToInt(pawn);
}

// injection de paramètres par coordonnée et valeur
public void update (Coord coord, int value) {
    situation[coord.getX()][coord.getY()] = value;
}

// renvoie les paramètre en int
public int convertPawnToInt (Pawn pawn) {
    int active, home;
    if(pawn.getIsActive()) active = 1;
    else active = 0;
    if(pawn.getIsHome()) home = 1;
    else home = 0;

    return pawn.getColor() * 1000 + pawn.getNumber() * 100 + active * 10 + home;
}

// renvoie un pion selon params (et Player[])
public Pawn convertIntToPawn (Player[] p, int params) {
    PawnFactory factory = new PawnFactory();
    Pawn o = factory.createPawn(0, 0, new Coord(DEFAULT));
    int color = params / 1000;
    int number = (params - (color * 1000)) / 100;

    switch(color) {
        case 1:   switch(number) {
                      case 1:   return p[0].getSet()[0];
                      case 2:   return p[0].getSet()[1];
                      case 3:   return p[0].getSet()[2];
                      case 4:   return p[0].getSet()[3];
                  }
```

```
            case 2:   switch(number) {
                        case 1:   return p[1].getSet()[0];
                        case 2:   return p[1].getSet()[1];
                        case 3:   return p[1].getSet()[2];
                        case 4:   return p[1].getSet()[3];
                      }
            case 3:   switch(number) {
                        case 1:   return p[2].getSet()[0];
                        case 2:   return p[2].getSet()[1];
                        case 3:   return p[2].getSet()[2];
                        case 4:   return p[2].getSet()[3];
                      }
            case 4:   switch(number) {
                        case 1:   return p[3].getSet()[0];
                        case 2:   return p[3].getSet()[1];
                        case 3:   return p[3].getSet()[2];
                        case 4:   return p[3].getSet()[3];
                      }
        }
        return o;
}

// renvoie un pion selon coordonnées
public Pawn getPawnFromCoord (Player[] p, Coord coord) {
        return convertIntToPawn(p, readParams(coord));
}

// pige une carte de pickDeck
public void pickCard () {
        if(!isPickEmpty()) {
                picked = pickDeck.getCard();
                dropDeck.add(picked);
        }
        // sinon transfère, shuffle et pick
        else {
                pickDeck.transfer(dropDeck.getList());
                dropDeck.getList().clear();
                pickDeck.mixDeck();
                picked = pickDeck.getCard();
        }
}

// mélange les cartes
private void mixPickDeck () {pickDeck.mixDeck();}
```

```
// insère carte pigée dans dropDeck
public void dropCard () {
	dropDeck.add(picked);
	pickDeck.remove();
}

// vérifie si pickDeck vide
public boolean isPickEmpty () {return pickDeck.isEmpty();}

// retourne la carte pigée (... peut avoir un bug ...)
public Card getPickedCard () {return picked;}

// remet un JOKER dans le dropDeck
public void dropJoker (Card joker) {dropDeck.add(joker);}

// configure le contexte des cartes
public void setPickContext (PickContext context_) {algoContext = context_;}

// ** Fonctions pour l'affichage **
// affichage du board
public void displayBoard (Player[] p, int turn) {
	System.out.print(M + "\t\t\t\t\tTour courant : ");
	switch(turn) {
		case 1:	System.out.println(R + "ROUGE" + N);
			break;
		case 2:	System.out.println(B + "BLEU" + N);
			break;
		case 3:	System.out.println(Y + "JAUNE" + N);
			break;
		case 4:	System.out.println(G + "VERT" + N);
			break;
		default:	System.out.println(E + "DEFAULT" + N);
		break;
	}
	// ligne 0
	{
	System.out.println();
	space(34);
	print(new Coord(0, 0), false); space(1); // case ordinaire
	YellowBG(); print(YSHORT_START, isCoordOnOwnSlide(YSHORT_START)); YellowBG(); space(1); // début de SHORT
	SLIDE jaune
	YellowBG(); print(new Coord(0, 2), isCoordOnOwnSlide(new Coord(0, 2))); YellowBG(); space(1); // SHORT SLIDE
	jaune
	YellowBG(); print(new Coord(0, 3), isCoordOnOwnSlide(new Coord(0, 3))); YellowBG(); space(1); // SHORT SLIDE
	jaune
	YellowBG(); print(YSTART, isCoordOnOwnSlide(YSTART)); space(1); // fin de SHORT SLIDE jaune
	print(new Coord(0, 5), false); space(1); // case ordinaire
```

```
print(new Coord(0, 6), false); space(1); // case ordinaire
print(new Coord(0, 7), false); space(1); // case ordinaire
print(new Coord(0, 8), false); space(1); // case ordinaire
YellowBG(); print(YLONG_START, isCoordOnOwnSlide(YLONG_START)); YellowBG(); space(1); // début de LONG SLIDE jaune
YellowBG(); print(new Coord(0, 10), isCoordOnOwnSlide(new Coord(0, 10))); YellowBG(); space(1); // LONG SLIDE jaune
YellowBG(); print(new Coord(0, 11), isCoordOnOwnSlide(new Coord(0, 11))); YellowBG(); space(1); // LONG SLIDE jaune
YellowBG(); print(new Coord(0, 12), isCoordOnOwnSlide(new Coord(0, 12))); YellowBG(); space(1); // LONG SLIDE jaune
YellowBG(); print(YLONG_END, isCoordOnOwnSlide(YLONG_END)); space(1); // fin de LONG SLIDE jaune
print(new Coord(0, 14), false); space(1); // case ordinaire
print(new Coord(0, 15), false); // case ordinaire
System.out.println();
}
// ligne 1
{
space(34);
print(new Coord(1, 0), false); space(3); // case ordinaire
YellowBG(); print(YSAFETY1, isCoordOnOwnSlide(YSAFETY1)); space(25); // SAFETY jaune
GreenBG(); print(GSHORT_START, isCoordOnOwnSlide(GSHORT_START)); // début de SHORT SLIDE verte
System.out.println();
}
// ligne 2
{
space(34);
BlueBG(); print(BLONG_END, isCoordOnOwnSlide(BLONG_END)); space(3); // fin de LONG SLIDE bleue
YellowBG(); print(YSAFETY2, isCoordOnOwnSlide(YSAFETY2)); space(3); // SAFETY jaune
print(YBASE4, false); space(1); // START jaune
print(YBASE3, false); space(5); // START jaune
printH(GREEN, p); space(3); // HOME vert
GreenBG(); print(GSAFETY5, isCoordOnOwnSlide(GSAFETY5)); GreenBG(); space(1); // SAFETY vert
GreenBG(); print(GSAFETY4, isCoordOnOwnSlide(GSAFETY4)); GreenBG(); space(1); // SAFETY vert
GreenBG(); print(GSAFETY3, isCoordOnOwnSlide(GSAFETY3)); GreenBG(); space(1); // SAFETY vert
GreenBG(); print(GSAFETY2, isCoordOnOwnSlide(GSAFETY2)); GreenBG(); space(1); // SAFETY vert
GreenBG(); print(GSAFETY1, isCoordOnOwnSlide(GSAFETY1)); GreenBG(); space(1); // SAFETY vert
GreenBG(); print(new Coord(2, 15), isCoordOnOwnSlide(new Coord(2, 15))); // LONG SLIDE verte
System.out.println();
}
// ligne 3
{
space(34);
BlueBG(); print(new Coord(3, 0), isCoordOnOwnSlide(new Coord(3, 0))); space(3); // LONG SLIDE bleue
YellowBG(); print(YSAFETY3, isCoordOnOwnSlide(YSAFETY3)); space(3); // SAFETY jaune
print(YBASE2, false); space(1); // START jaune
print(YBASE1, false); space(19); // START jaune
```

```
GreenBG(); print(new Coord(3, 15), isCoordOnOwnSlide(new Coord(3, 15))); // SHORT SLIDE verte
System.out.println();
}
// ligne 4
{
space(34);
BlueBG(); print(new Coord(4, 0), isCoordOnOwnSlide(new Coord(4, 0))); space(3); // LONG SLIDE bleue
YellowBG(); print(YSAFETY4, isCoordOnOwnSlide(YSAFETY4)); space(19); // SAFETY jaune
print(GBASE2, false); space(1); // START vert
print(GBASE4, false); space(3); // START vert
GreenBG(); print(GSTART, isCoordOnOwnSlide(GSTART)); // fin de SHORT SLIDE verte
System.out.println();
}
// ligne 5
{
space(34);
BlueBG(); print(new Coord(5, 0), isCoordOnOwnSlide(new Coord(5, 0))); space(3); // LONG SLIDE bleue
YellowBG(); print(YSAFETY5, isCoordOnOwnSlide(YSAFETY5)); space(7); // SAFETY jaune
deck(7, 1); space(5); // DECK UP
print(GBASE1, false); space(1); // START vert
print(GBASE3, false); space(3); // START vert
print(new Coord(5, 15), false); // case ordinaire
System.out.println();
}
// ligne 6
{
space(34);
BlueBG(); print(BLONG_START, isCoordOnOwnSlide(BLONG_START)); space(11); // début LONG SLIDE bleue
deck(1, 1); space(1); // DECK UP
printD1(turn); space(1); // digit 1 DECK UP
printD2(turn); space(1); // digit 2 DECK UP
deck(1, 1); space(11); // DECK
print(new Coord(6, 15), false); // case ordinaire
System.out.println();
}
// ligne 7
{
space(34);
print(new Coord(7, 0),false); space(3); // case ordinaire
System.out.print(Y);
printH(YELLOW, p); space(7); // HOME jaune
deck(7, 1); space(11); // DECK
print(new Coord(7, 15), false); // case ordinaire
System.out.println();
}
// ligne 8
{
```

```
space(34);
print(new Coord(8, 0), false); space(11); // case ordinaire
deck(7, 4); space(7); // DECK
printH(RED, p); space(3); // HOME rouge
print(new Coord(8, 15), false); // case ordinaire
System.out.println();
}
// ligne 9
{
space(34);
print(new Coord(9, 0), false); space(11); // case ordinaire
deck(1, 4); System.out.print("\033[1;95mSORRY\033[0m"); // DECK
deck(1, 4); space(11); // DECK
GreenBG(); print(GLONG_START, isCoordOnOwnSlide(GLONG_START)); // début de LONG SLIDE verte
System.out.println();
}
// ligne 10
{
space(34);
print(new Coord(10, 0), false); space(3); // case ordinaire
print(BBASE3, false); space(1); // START bleu
print(BBASE1, false); space(5); // START bleu
deck(7, 4); space(7); // DECK
RedBG(); print(RSAFETY5, isCoordOnOwnSlide(RSAFETY5)); space(3); // SAFETY rouge
GreenBG(); print(new Coord(10, 15), isCoordOnOwnSlide(new Coord(10, 15))); // LONG SLIDE verte
System.out.println();
}
// ligne 11
{
space(34);
BlueBG(); print(BSTART, isCoordOnOwnSlide(BSTART)); space(3); // fin de SHORT SLIDE bleue
print(BBASE4, false); space(1); // START bleu
print(BBASE2, false); space(19); // START bleu
RedBG(); print(RSAFETY4, isCoordOnOwnSlide(RSAFETY4));space(3); // SAFETY rouge
GreenBG(); print(new Coord(11, 15), isCoordOnOwnSlide(new Coord(11, 15))); // LONG SLIDE verte
System.out.println();
}
// ligne 12
{
space(34);
BlueBG(); print(new Coord(12, 0), isCoordOnOwnSlide(new Coord(12, 0))); space(19); // SHORT SLIDE bleue
print(RBASE1, false); space(1); // START rouge
print(RBASE2, false); space(3); // START rouge
RedBG(); print(RSAFETY3, isCoordOnOwnSlide(RSAFETY3)); space(3); // SAFETY rouge
GreenBG(); print(new Coord(12, 15), isCoordOnOwnSlide(new Coord(12, 15))); // LONG SLIDE verte
System.out.println();
}
```

```
// ligne 13
{
space(34);
BlueBG(); print(new Coord(13, 0), isCoordOnOwnSlide(new Coord(13, 0))); BlueBG(); space(1); // SHORT SLIDE
bleue
BlueBG(); print(BSAFETY1, isCoordOnOwnSlide(BSAFETY1)); BlueBG(); space(1); // SAFETY bleu
BlueBG(); print(BSAFETY2, isCoordOnOwnSlide(BSAFETY2)); BlueBG(); space(1); // SAFETY bleu
BlueBG(); print(BSAFETY3, isCoordOnOwnSlide(BSAFETY3)); BlueBG(); space(1); // SAFETY bleu
BlueBG(); print(BSAFETY4, isCoordOnOwnSlide(BSAFETY4)); BlueBG(); space(1); // SAFETY bleu
BlueBG(); print(BSAFETY5, isCoordOnOwnSlide(BSAFETY5)); space(3); // SAFETY bleu
printH(BLUE, p); space(5); // HOME bleu
print(RBASE3, false); space(1); // START rouge
print(RBASE4, false); space(3); // START rouge
RedBG(); print(RSAFETY2, true); space(3); // SAFETY rouge
GreenBG(); print(GLONG_END, isCoordOnOwnSlide(GLONG_END)); // fin de LONG SLIDE verte
System.out.println();
}
// ligne 14
{
space(34);
BlueBG(); print(BSHORT_START, isCoordOnOwnSlide(BSHORT_START)); space(25); // début de SHORT SLIDE bleue
RedBG(); print(RSAFETY1, isCoordOnOwnSlide(RSAFETY1)); space(3); // SAFETY rouge
print(new Coord(14, 15), false); // case ordinaire
System.out.println();
}
// ligne 15
{
space(34);
print(new Coord(15, 0), false); space(1); // case ordinaire
print(new Coord(15, 1), false); space(1); // case ordinaire
RedBG(); print(RLONG_END, isCoordOnOwnSlide(RLONG_END)); RedBG(); space(1); // début de LONG SLIDE rouge
RedBG(); print(new Coord(15, 3), isCoordOnOwnSlide(new Coord(15, 3))); RedBG(); space(1); // LONG SLIDE rouge
RedBG(); print(new Coord(15, 4), isCoordOnOwnSlide(new Coord(15, 4))); RedBG(); space(1); // LONG SLIDE rouge
RedBG(); print(new Coord(15, 5), isCoordOnOwnSlide(new Coord(15, 5))); RedBG(); space(1); // LONG SLIDE rouge
RedBG(); print(RLONG_START, isCoordOnOwnSlide(RLONG_START)); space(1); // fin de LONG SLIDE rouge
print(new Coord(15, 7), false); space(1); // case ordinaire
print(new Coord(15, 8), false); space(1); // case ordinaire
print(new Coord(15, 9), false); space(1); // case ordinaire
print(new Coord(15, 10), false); space(1); // case ordinaire
RedBG(); print(RSTART, isCoordOnOwnSlide(RSTART)); RedBG(); space(1); // fin de SHORT SLIDE rouge
RedBG(); print(new Coord(15, 12), isCoordOnOwnSlide(new Coord(15, 12))); RedBG(); space(1); // SHORT SLIDE
rouge
RedBG(); print(new Coord(15, 13), isCoordOnOwnSlide(new Coord(15, 13))); RedBG(); space(1); // SHORT SLIDE
rouge
RedBG(); print(RSHORT_START, isCoordOnOwnSlide(RSHORT_START)); space(1); // début de SHORT SLIDE rouge
print(new Coord(15, 15), false); space(1); // case ordinaire
System.out.println();
```

```
        System.out.println();
        }
}

public void displaySituation () {
        for(int i = 0 ; i < 16 ; i++) {
                for(int j = 0 ; j < 16 ; j++) {
                        System.out.print(situation[i][j] + " ### ");
                }
                System.out.println();
        }
}

// affiche n espaces
public void space (int n) {
        for(int i = 0 ; i < n ; i++) System.out.print(" ");
}

// affiche le contour des decks
public void deck (int n, int color) {
        for(int i = 0 ; i < n ; i++) {
                if(color == GREEN) System.out.print(GG + " " + N);
                else if(color == RED) System.out.print(RR + " " + N);
                else System.out.print(N + "*" + N);
        }
}

// affiche le numéro du pion et sa couleur (ou * pour non-occupé) -point d'entrée pour stratégie BoardStrategy
public void print (Coord coord, boolean ownSlide) {
        colorContext.setCoord(coord);
        colorContext.setStrategy(setContextStrategy1(coord));
        System.out.print(colorContext.print(readNumber(coord), ownSlide)); // coord est 0,0
}

// affiche le premier digit de dropDeck
private void printD1 (int color) {
        if(picked == null) System.out.print(N + " ");
        else {
                int n = picked.getNumber();
                String number = Integer.toString(n);
                if(n >= 10) {
                        switch(color) {
                                case 1:   System.out.print(N + R + number.charAt(0) + N);
                                          break;
                                case 2:   System.out.print(N + B + number.charAt(0) + N);
                                          break;
                                case 3:   System.out.print(N + Y + number.charAt(0) + N);
```

```
                        break;
                case 4:  System.out.print(N + B + number.charAt(0) + N);
                        break;
                default:  System.out.print(N + " ");
            }
        }
        else System.out.print(N + " ");
    }
}

// afficher le deuxième digit de dropDeck
private void printD2 (int color) {
    if(picked == null) System.out.print(N + " ");
    else {
        int n = picked.getNumber();
        String number = Integer.toString(n);
        if(n < 10) {
            switch(color) {
                case 1:  System.out.print(N + R + number.charAt(0) + N);
                        break;
                case 2:  System.out.print(N + B + number.charAt(0) + N);
                        break;
                case 3:  System.out.print(N + Y + number.charAt(0) + N);
                        break;
                case 4:  System.out.print(N + G + number.charAt(0) + N);
                        break;
                default:  System.out.print(N + " ");
            }
        }
        else {
            switch(color) {
                case 1:  System.out.print(N + R + number.charAt(1) + N);
                        break;
                case 2:  System.out.print(N + B + number.charAt(1) + N);
                        break;
                case 3:  System.out.print(N + Y + number.charAt(1) + N);
                        break;
                case 4:  System.out.print(N + G + number.charAt(1) + N);
                        break;
                default:  System.out.print(N + " ");
            }
        }
    }
}
```

```
    // affiche les backgrounds
    private void RedBG () {System.out.print(RR);}
    private void BlueBG () {System.out.print(BB);}
    private void YellowBG () {System.out.print(YY);}
    private void GreenBG () {System.out.print(GG);}

    // affiche le nombre de pions dans HOME
    private void printH (int color, Player[] p) {
        if(color == RED) {
            System.out.print(R + pullHomeCount(RED, p) + N);
        }
        else if(color == BLUE) {
            System.out.print(B + pullHomeCount(BLUE, p) + N);
        }
        else if(color == YELLOW) {
            System.out.print(Y + pullHomeCount(YELLOW, p) + N);
        }
        else if(color == GREEN) {
            System.out.print(G + pullHomeCount(GREEN, p) + N);
        }
    }

    // retourne nombre de pions dans HOME
    private int pullHomeCount (int color, Player[] p) {
        int n = 0;
        if(color == RED) n = p[0].countHome();
        else if(color == BLUE) n = p[1].countHome();
        else if(color == YELLOW) n = p[2].countHome();
        else if(color == GREEN) n = p[3].countHome();
        return n;
    }
}
```

“BoardParams.java”

```
package board;

import piece.Pawn;
import player.Player;

public interface BoardParams {
	abstract int readParams (Coord o);
	abstract int readColor (Coord o);
	abstract int readNumber (Coord o);
	abstract int readActive (Coord o);
	abstract int readHome (Coord o);
	abstract void update (Pawn pawn);
	abstract Pawn convertIntToPawn (Player[] p, int params);
	abstract int convertPawnToInt (Pawn pawn);
}
```

“Coord.java”

```
package board;

public class Coord {
	private int x, y;

	public Coord () {
		this.x = -1;
		this.y = -1;
	}

	public Coord (Coord coord) {
		this.x = coord.getX();
		this.y = coord.getY();
	}

	public Coord (int X, int Y) {
		this.x = X;
		this.y = Y;
	}

	public int getX () {return this.x;}
	public void setX (int n) {this.x = n;}
	public int getY () {return this.y;}
	public void setY (int n) {this.y = n;}
	public Coord getCoord () {return this;}
	public void setCoord (int X, int Y) {this.x = X; this.y = Y;}
	public void setCoord (Coord o) {this.x = o.x; this.y = o.y;}
	public boolean equals (Coord o) {
		if(o == null) return false;
		return this.x == o.x && this.y == o.y;
	}
}
```

“Card.java”

```
package deck;

public class Card {
	private int number;

	public Card (int num) {this.number = num;}
	public Card getCard () {return this;}
	public int getNumber () {return number;}
}
```

“CardFactory.java”

```
package deck;

public interface CardFactory {
	DownDeck createDeck ();
}
```

“CardFactoryDown.java”

```
package deck;

public class CardFactoryDown implements CardFactory {
	public DownDeck createDeck () {
		DownDeck dd = DownDeck.getInstance();
		for(int i = 0 ; i < 4 ; i++) BuildDeckSet(dd);
		return dd;
	}

	private void BuildDeckSet (DownDeck dd) {
		// ajout des cartes requises (un set sur quatre)
		dd.add(new Card(1));
		dd.add(new Card(2));
		dd.add(new Card(3));
		dd.add(new Card(4));
		dd.add(new Card(5));
		dd.add(new Card(7));
		dd.add(new Card(8));
		dd.add(new Card(10));
		dd.add(new Card(11));
		dd.add(new Card(12));
		dd.add(new Card(13)); // carte "Sorry!"
	}
}
```

“DownDeck.java”

```
package deck;

import java.util.*;

// implémenté en tant que singleton
public class DownDeck {
	private static final DownDeck INSTANCE = new DownDeck();
	private static LinkedList<Card> dDeck;
	private DownDeck () {dDeck = new LinkedList<Card>();}
	public static DownDeck getInstance () {return INSTANCE;}
	public boolean add (Card n) {return dDeck.add(n);}
	public boolean isEmpty () {return dDeck.isEmpty();}
	public Card remove () {return dDeck.removeFirst();}
	public Iterator<Card> getIterator () {return dDeck.iterator();}
	public int peekCard () {return dDeck.peekFirst().getNumber();} // inutilisée
	public LinkedList<Card> getList () {return dDeck;}
	public boolean transfer (LinkedList<Card> c) {dDeck.clear(); return dDeck.addAll(c);}

	public Card getCard () throws NoSuchElementException {
		return dDeck.getFirst();
	}

	public void mixDeck () {
		Object[] arr = dDeck.toArray();
		Random rand = new Random();
		for (int i = 0; i < arr.length - 1; i++) {
			// sélectionne l'index au hasard
			int index = rand.nextInt(i + 1);
			// permutation
			Object g = arr[index];
			arr[index] = arr[i];
			arr[i] = g;
		}
		// recopie Cards dans la liste
		ListIterator<Card> it = dDeck.listIterator();
		for(int i = 0 ; i < arr.length ; i++) {
			it.next();
			it.set((Card)arr[i]);
		}
	}
}
```

“UpDeck.java”

```
package deck;

import java.util.*;

// implémenté en tant que singleton
public class UpDeck {
	private static final UpDeck INSTANCE = new UpDeck();
	private static LinkedList<Card> uDeck;
	public static UpDeck getInstance () {return INSTANCE;}
	private UpDeck () {uDeck = new LinkedList<Card>();}
	public boolean add (Card n) {return uDeck.add(n);}
	public Card remove () {return uDeck.removeFirst();}
	public Iterator<Card> getIterator () {return uDeck.iterator();}
	public void clearDeck () {uDeck.clear();}
	public LinkedList<Card> getList () {return uDeck;}
}
```

“PickContext”

```
package game.cardalgo;

import board.*;
import player.Player;

public class PickContext {
	private PickStrategy strategy;
	private Board b;
	private Player[] p;
	private int n;
	private int turn;
	private boolean forward;

	public PickContext (PickStrategy strategy_, Board b_, Player[] p_, int n_, int turn_, boolean fw) {
		strategy = strategy_;
		b = b_;
		p = p_;
		n = n_;
		turn = turn_;
		forward = fw;
	}

	public void setStrategy (PickStrategy strategy_) {
		strategy = strategy_;
	}

	public void operation () {
		strategy.algorithm(b, p, n, turn, forward);
	}
}
```

“PickStrategy.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

import java.util.InputMismatchException;
import java.util.Scanner;

import game.Game;
import game.GameConstants;

public interface PickStrategy extends GameConstants {
	Scanner scan = new Scanner(System.in);

	void algorithm (Board b, Player[] p, int n, int turn, boolean forw);

	// permute position de deux pions
	default void permutePawns (Board b, Pawn A, Pawn B) {
		Coord temp = new Coord(DEFAULT);
		// calcule les case depuis case départ
		int countA = Utils.countToStartFrom(B.getPos(), A.getColor());
		int countB = Utils.countToStartFrom(A.getPos(), B.getColor());
		// permute les positions
		temp.setCoord(A.getPos());
		A.setPos(B.getPos());
		B.setPos(temp);
		// ajuste les coompteurs count respectifs
		A.setCount(countA);
		B.setCount(countB);
		// update le board
		b.update(A);
		b.update(B);
	}

	// split mouvement entre deux pions si le 7 est pigé
	default void splitCount (Player[] p, Board b, Pawn A, Pawn B, int n, int turn) {
		moveProcess(p, b, A, 4, turn, true);
		moveProcess(p, b, B, 3, turn, true);
	}
```

```
// menu demander pour A et B, split ou permute
default void menuAskForAB (Board b, Player[] p, int turn, boolean willSplit) {
	PawnFactory factory = new PawnFactory();
	Pawn A = factory.createPawn(0, 0, DEFAULT);
	Pawn B = factory.createPawn(0, 0, DEFAULT);
	Pawn[] psetF, psetO;
	int loopCount;

	psetF = pawnsActiveFriendly(p, turn);
	// si exactement 2 pions actifs, demander seulement celui qui avance de 4 cases
	if(psetF.length == 2 && willSplit) {
		if(p[turn-1].getIsHuman()) {
			System.out.println("\n\t\t\t\t" + M +"Sélection pion " + R + "A" + M + " (qui avance 4 cases) : " + N);
			loopCount = 0;
			do {
				A = askNumberSameColor(p, turn);
				loopCount++;
			}
			while(!isPawnInList(psetF, A) && loopCount < 4);
			// détermine pion B
			for(int i = 0 ; i < psetF.length ; i++) {
				if(A != psetF[i] && !psetF[i].getPos().equals(DEFAULT)) {
					B = psetF[i];
					break;
				}
			}
		}
		else {
			if(Utils.intToBoolean(Utils.randomChoice(1))) {
				A = psetF[Utils.randomChoice(psetF.length-1)];
				for(int i = 0 ; i < psetF.length ; i++) {
					if(A != psetF[i] && !psetF[i].getPos().equals(DEFAULT)) {
						B = psetF[i];
						break;
					}
				}
			}
		}
	}
	else if(psetF.length > 2 && !willSplit) {
		psetO = pawnsActiveOpponent(p, B.getColor());
		if(p[turn-1].getIsHuman()) {
			System.out.println("\n\t\t\t\t" + M +"Sélection pion " + R + "A" + M + " (votre pion) : " + N);
			loopCount = 0;
```

```
            do {
                A = askColorNumber(p);
                loopCount++;
            }
            while(!isPawnInList(psetF, A) && loopCount < 4);
            System.out.println("\n\t\t\t\t" + M +"Sélectionner pion " + R + "B" + M + " (adversaire) : " + N);
            // si désire permuter
            loopCount = 0;
            do {
                B = askColorNumber(p);
                loopCount++;
            }
            while(!isPawnInList(psetO, B) && !b.isSafety(B.getPos(), B.getColor()) && loopCount < 4);
        }
        else {
            if(Utils.intToBoolean(Utils.randomChoice(1))) {
                A = psetF[Utils.randomChoice(psetF.length-1)];
                B = psetO[Utils.randomChoice(psetO.length-1)];
            }
        }
    }
    else if(psetF.length > 2 && willSplit) {
        if(p[turn-1].getIsHuman()) {
            System.out.println("\n\t\t\t\t" + M +"Sélection pion " + R + "A" + M + " (4 cases) : " + N);
            loopCount = 0;
            do {
                A = askNumberSameColor(p, turn);
                loopCount++;
            }
            while(!isPawnInList(psetF, A) && loopCount < 4);

            System.out.println("\n\t\t\t\t" + M +"Sélectionner pion " + R + "B" + M + " (3 cases) : " + N);
            // si désire spliter
            loopCount = 0;
            do {
                B = askNumberSameColor(p, turn);
                loopCount++;
            }
            while(!isPawnInList(psetF, B) && loopCount < 4);
        }
        else {
            if(Utils.intToBoolean(Utils.randomChoice(1))) {
                A = psetF[Utils.randomChoice(psetF.length-1)];
                do {
                    B = psetF[Utils.randomChoice(psetF.length-1)];
                }
                while(A.equals(B));
```

```
                }
            }
        }

        if(willSplit) { // peut avoir petit bug ici (pas noobproof)
            if(A.getColor() == B.getColor()) splitCount(p, b, A, B, 7, turn);
            else menuAskForAB(b, p, turn, willSplit);
        }
        else {
            permutePawns(b, A, B);
        }
    }

    // retourne un pion *** USER INPUT pour couleur ET numéro
    default Pawn askColorNumber (Player[] p) {
        PawnFactory factory = new PawnFactory();
        Pawn A = factory.createPawn(0, 0, DEFAULT);
        String choice = "";
        int loopCount;

        System.out.print("\n\t\t\t\t" + M +"Sélectionner un pion " + E + " (Ex. : 2 rouge = R2 ou r2)" + M + " : " + Y);
        loopCount = 0;
        do {
            choice = scan.nextLine().toLowerCase();
            loopCount++;
        }
        while(!Utils.isStringValid(choice) && loopCount < 4);

        if(choice.charAt(0) == 'r') {
            switch(Integer.parseInt(Character.toString(choice.charAt(1)))) {
                case 1:   return p[0].getSet()[0];
                case 2:   return p[0].getSet()[1];
                case 3:   return p[0].getSet()[2];
                case 4:   return p[0].getSet()[3];
            }
        }
        else if(choice.charAt(0) == 'b') {
            switch(Integer.parseInt(Character.toString(choice.charAt(1)))) {
                case 1:   return p[1].getSet()[0];
                case 2:   return p[1].getSet()[1];
                case 3:   return p[1].getSet()[2];
                case 4:   return p[1].getSet()[3];
            }
        }
        else if(choice.charAt(0) == 'y') {
            switch(Integer.parseInt(Character.toString(choice.charAt(1)))) {
                case 1:   return p[2].getSet()[0];
```

```
                case 2:   return p[2].getSet()[1];
                case 3:   return p[2].getSet()[2];
                case 4:   return p[2].getSet()[3];
            }
        }
        else if(choice.charAt(0) == 'g') {
            switch(Integer.parseInt(Character.toString(choice.charAt(1)))) {
                case 1:   return p[3].getSet()[0];
                case 2:   return p[3].getSet()[1];
                case 3:   return p[3].getSet()[2];
                case 4:   return p[3].getSet()[3];
            }
        }
        return A;
    }

    // retourne un pion de même couleur *** USER INPUT pour #
    default Pawn askNumberSameColor (Player[] p, int color) {
        PawnFactory factory = new PawnFactory();
        Pawn A = factory.createPawn(0, 0, DEFAULT);
        int choice = 0;
        int loopCount;

        System.out.print("\n\t\t\t\t\t" + M +"Sélectionner un pion (votre pion) : " + Y);
        loopCount = 0;
        do {
            try {
                choice = scan.nextInt();
                loopCount++;
            }
            catch(InputMismatchException e) {}
            scan.nextLine();
        }
        while(choice > 4 || choice < 1 && loopCount < 4);
        switch(color) {
            case 1:   switch(choice) {
                        case 1:   return p[0].getSet()[0];
                        case 2:   return p[0].getSet()[1];
                        case 3:   return p[0].getSet()[2];
                        case 4:   return p[0].getSet()[3];
                        default:  return A;
                    }
            case 2:   switch(choice) {
                        case 1:   return p[1].getSet()[0];
                        case 2:   return p[1].getSet()[1];
                        case 3:   return p[1].getSet()[2];
                        case 4:   return p[1].getSet()[3];
```

```
                    default:  return A;
                }
            case 3:  switch(choice) {
                    case 1:  return p[2].getSet()[0];
                    case 2:  return p[2].getSet()[1];
                    case 3:  return p[2].getSet()[2];
                    case 4:  return p[2].getSet()[3];
                    default:  return A;
                }
            case 4:  switch(choice) {
                    case 1:  return p[3].getSet()[0];
                    case 2:  return p[3].getSet()[1];
                    case 3:  return p[3].getSet()[2];
                    case 4:  return p[3].getSet()[3];
                    default:  return A;
                }
        }
        return A;
}

// retourne boolean = true pour avancer et false pour reculer
default boolean askFWorBW () {
        int choice = 0;
        System.out.print("\n\t\t\t\t\t" + M +"Avancer de 10 cases (" + C + "1" + M + ") ou reculer de 1 case (" + C + "0" +
        M + ") : " + Y);
        do {
            try {
                choice = scan.nextInt();
            }
            catch(InputMismatchException e) {}
            scan.nextLine();
        }
        while(!Utils.isNumberValid1(choice));
        if(choice == 1) return true;
        return false;
}
```

```
// retourne boolean = true pour avancer et false pour sortir un pion de START (si disponible)
default boolean askOutOrMove (Board b, Player[] p, int color) {
	int choice = 0;
	System.out.print("\n\t\t\t\t\t" + M +"Avancer pion (" + C + "1" + M + ") ou sortir pion (" + C + "0" + M + ") : " + Y);
	do {
		try {
			choice = scan.nextInt();
		}
		catch(InputMismatchException e) {}
		scan.nextLine();
	}
	while(!Utils.isNumberValid1(choice));
	if(choice == 1) return true;
	return false;
}

// retourne boolean = true pour avancer et false pour spliter
default boolean askGoOrSplit () {
	int choice = 0;
	System.out.print("\n\t\t\t\t\t" + M +"Avancer pion (" + C + "1" + M + ") ou séparer entre deux pions (" + C + "0" +
	M + ") : " + Y);
	do {
		try {
			choice = scan.nextInt();
		}
		catch(InputMismatchException e) {}
		scan.nextLine();
	}
	while(!Utils.isNumberValid1(choice));
	if(choice == 1) return true;
	return false;
}

// retourne boolean = true pour avancer et false pour permuter
default boolean askGoOrTrade () {
	int choice = 0;
	System.out.print("\n\t\t\t\t\t" + M + "Avancer pion (" + C + "1" + M + ") ou échanger avec pion adversaire (" + C +
	"0" + M + ") : " + Y);
	do {
		try {
			choice = scan.nextInt();
		}
		catch(InputMismatchException e) {}
		scan.nextLine();
	}
	while(!Utils.isNumberValid1(choice));
	if(choice == 1) return true;
```

```
        return false;
    }

    default Pawn askForPartnerPawn (Player[] p, int color, Pawn[] setP) {
        PawnFactory factory = new PawnFactory();
        Pawn A = factory.createPawn(0, 0, DEFAULT);
        int choice = 0;
        int loopCount;

        System.out.print("\n\t\t\t\t" + M +"Sélectionner un pion (partenaire) : " + Y);
        loopCount = 0;
        do {
            try {
                choice = scan.nextInt();
                loopCount++;
            }
            catch(InputMismatchException e) {}
            scan.nextLine();
        }
        while(choice > 4 || choice < 1 && loopCount < 4);
        switch(color) {
            case 1:   return p[p[0].getPartner()].getSet()[choice-1];
            case 2:   return p[p[1].getPartner()].getSet()[choice-1];
            case 3:   return p[p[2].getPartner()].getSet()[choice-1];
            case 4:   return p[p[3].getPartner()].getSet()[choice-1];
        }
        return A;
    }

    // retourne boolean = true pour conserver joker ou false sinon
    default boolean askKeepOrElse () {
        int choice = 0;
        System.out.print("\n\t\t\t" + M + "Conserver JOKER (" + C + "1"+ M + ") ou utiliser immédiatement (" + C + "0" +
        M + ") : " + Y);
        do {
            try {
                choice = scan.nextInt();
            }
            catch(InputMismatchException e) {}
            scan.nextLine();
        }
        while(!Utils.isNumberValid1(choice));
        if(choice == 1) return true;
        return false;
    }
```

```
// retourne 1 pour sortir friendly de START ou 0 pour renvoyer pion adversaire
default int askOutOrKick (Board b, Player[] p, int color) {
	int choice = 0;
	System.out.print("\n\t\t\t\t\t" + M + "Sortir pion (" + C + "1" + M + ") ou renvoyer un adversaire (" + C + "0" + M +
	") : " + Y);
	do {
		try {
			choice = scan.nextInt();
		}
		catch(InputMismatchException e) {}
		scan.nextLine();
	}
	while(!Utils.isNumberValid1(choice));
	return choice;
}

// retourne pions actifs "friendly"
default Pawn[] pawnsActiveFriendly (Player[] p, int color) {
	PawnFactory factory = new PawnFactory();
	Pawn temp = factory.createPawn(0, 0, DEFAULT);
	Pawn[] pawns;
	int n = 0;

	switch(color) {
		case 1:	for(int i = 0 ; i < 4 ; i++) {
				if(p[0].getSet()[i].getIsActive()) n++;
			}
			if(n == 0) {
				pawns = new Pawn[1];
				pawns[0] = temp;
				break;
			}
			pawns = new Pawn[n];

			n = 0;
			for(int i = 0 ; i < 4 ; i++) {
				if(p[0].getSet()[i].getIsActive()) {
					pawns[n] = p[0].getSet()[i];
					n++;
				}
			}
			break;
```

```
case 2:  for(int i = 0 ; i < 4 ; i++) {
             if(p[1].getSet()[i].getIsActive()) n++;
         }
         if(n == 0) {
             pawns = new Pawn[1];
             pawns[0] = temp;
             break;
         }
         pawns = new Pawn[n];
         n = 0;
         for(int i = 0 ; i < 4 ; i++) {
             if(p[1].getSet()[i].getIsActive()) {
                 pawns[n] = p[1].getSet()[i];
                 n++;
             }
         }
         break;
case 3:  for(int i = 0 ; i < 4 ; i++) {
             if(p[2].getSet()[i].getIsActive()) n++;
         }
         if(n == 0) {
             pawns = new Pawn[1];
             pawns[0] = temp;
             break;
         }
         pawns = new Pawn[n];
         n = 0;
         for(int i = 0 ; i < 4 ; i++) {
             if(p[2].getSet()[i].getIsActive()) {
                 pawns[n] = p[2].getSet()[i];
                 n++;
             }
         }
         break;
case 4:  for(int i = 0 ; i < 4 ; i++) {
             if(p[3].getSet()[i].getIsActive()) n++;
         }
         if(n == 0) {
             pawns = new Pawn[1];
             pawns[0] = temp;
             break;
         }
```

```
                    pawns = new Pawn[n];
                    n = 0;
                    for(int i = 0 ; i < 4 ; i++) {
                        if(p[3].getSet()[i].getIsActive()) {
                            pawns[n] = p[3].getSet()[i];
                            n++;
                        }
                    }
                    break;
            default:  pawns = new Pawn[1];
                    pawns[0] = temp;
        }
        return pawns;
}

// retourne le nombre de pions actifs de même couleur
default int countActiveFriendly (Player[] p, int color) {return pawnsActiveFriendly(p, color).length;}

// retourne pions actifs des adversaires
default Pawn[] pawnsActiveOpponent (Player[] p, int color) {
        PawnFactory factory = new PawnFactory();
        Pawn temp = factory.createPawn(0, 0, DEFAULT);
        Pawn[] pawns;
        int n = 0;
        switch(color) {
            case 1:  for(int i = 0 ; i < 4 ; i++) {
                        if(p[1].getSet()[i].getIsActive()) n++;
                    }
                    for(int i = 0 ; i < 4 ; i++) {
                        if(p[2].getSet()[i].getIsActive()) n++;
                    }
                    for(int i = 0 ; i < 4 ; i++) {
                        if(p[3].getSet()[i].getIsActive()) n++;
                    }
                    pawns = new Pawn[n];
                    if(n == 0) {
                        pawns = new Pawn[1];
                        pawns[0] = temp;
                        break;
                    }
                    n = 0;
                    for(int i = 0 ; i < 4 ; i++) {
                        if(p[1].getSet()[i].getIsActive()) {
                            pawns[n] = p[1].getSet()[i];
                            n++;
                        }
                    }
```

```
            for(int i = 0 ; i < 4 ; i++) {
                if(p[2].getSet()[i].getIsActive()) {
                    pawns[n] = p[2].getSet()[i];
                    n++;
                }
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[3].getSet()[i].getIsActive()) {
                    pawns[n] = p[3].getSet()[i];
                    n++;
                }
            }
            break;
        case 2:  for(int i = 0 ; i < 4 ; i++) {
                if(p[0].getSet()[i].getIsActive()) n++;
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[2].getSet()[i].getIsActive()) n++;
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[3].getSet()[i].getIsActive()) n++;
            }
            pawns = new Pawn[n];
            if(n == 0) {
                pawns = new Pawn[1];
                pawns[0] = temp;
                break;
            }
            n = 0;
            for(int i = 0 ; i < 4 ; i++) {
                if(p[0].getSet()[i].getIsActive()) {
                    pawns[n] = p[0].getSet()[i];
                    n++;
                }
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[2].getSet()[i].getIsActive()) {
                    pawns[n] = p[2].getSet()[i];
                    n++;
                }
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[3].getSet()[i].getIsActive()) {
                    pawns[n] = p[3].getSet()[i];
                    n++;
                }
            }
```

```
            break;
case 3:     for(int i = 0 ; i < 4 ; i++) {
                if(p[0].getSet()[i].getIsActive()) n++;
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[1].getSet()[i].getIsActive()) n++;
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[3].getSet()[i].getIsActive()) n++;
            }
            pawns = new Pawn[n];
            if(n == 0) {
                pawns = new Pawn[1];
                pawns[0] = temp;
                break;
            }
            n = 0;
            for(int i = 0 ; i < 4 ; i++) {
                if(p[0].getSet()[i].getIsActive()) {
                    pawns[n] = p[0].getSet()[i];
                    n++;
                }
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[1].getSet()[i].getIsActive()) {
                    pawns[n] = p[1].getSet()[i];
                    n++;
                }
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[3].getSet()[i].getIsActive()) {
                    pawns[n] = p[3].getSet()[i];
                    n++;
                }
            }
            break;
case 4:     for(int i = 0 ; i < 4 ; i++) {
                if(p[0].getSet()[i].getIsActive()) n++;
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[1].getSet()[i].getIsActive()) n++;
            }
            for(int i = 0 ; i < 4 ; i++) {
                if(p[2].getSet()[i].getIsActive()) n++;
            }
            pawns = new Pawn[n];
```

```
                    if(n == 0) {
                        pawns = new Pawn[1];
                        pawns[0] = temp;
                        break;
                    }
                    n = 0;
                    for(int i = 0 ; i < 4 ; i++) {
                        if(p[0].getSet()[i].getIsActive()) {
                            pawns[n] = p[0].getSet()[i];
                            n++;
                        }
                    }
                    for(int i = 0 ; i < 4 ; i++) {
                        if(p[1].getSet()[i].getIsActive()) {
                            pawns[n] = p[1].getSet()[i];
                            n++;
                        }
                    }
                    for(int i = 0 ; i < 4 ; i++) {
                        if(p[2].getSet()[i].getIsActive()) {
                            pawns[n] = p[2].getSet()[i];
                            n++;
                        }
                    }
                    break;
            default:  pawns = new Pawn[1];
                    pawns[0] = temp;
        }
        return pawns;
}

// retourne le nombre de pions actifs des adversaires
default int countActiveOpponent (Player[] p, int color) {return pawnsActiveOpponent(p, color).length;}

// retourne pions actifs du partner
default Pawn[] pawnsActivePartner (Player[] p, int color) {
        PawnFactory factory = new PawnFactory();
        Pawn temp = factory.createPawn(0, 0, DEFAULT);
        Pawn[] pawns;

        switch(color) {
            case 1:  switch(p[0].getPartner()) {
                        case 2:  pawns = pawnsActiveFriendly(p, 2);
                                 break;
                        case 3:  pawns = pawnsActiveFriendly(p, 3);
                                 break;
                        case 4:  pawns = pawnsActiveFriendly(p, 4);
```

```
                        break;
                    default:  pawns = pawnsActiveFriendly(p, 0);
                        break;
                }
            case 2:  switch(p[1].getPartner()) {
                    case 1:  pawns = pawnsActiveFriendly(p, 1);
                        break;
                    case 3:  pawns = pawnsActiveFriendly(p, 3);
                        break;
                    case 4:  pawns = pawnsActiveFriendly(p, 4);
                        break;
                    default:  pawns = pawnsActiveFriendly(p, 0);
                        break;
                }
            case 3:  switch(p[2].getPartner()) {
                    case 1:  pawns = pawnsActiveFriendly(p, 1);
                        break;
                    case 2:  pawns = pawnsActiveFriendly(p, 2);
                        break;
                    case 4:  pawns = pawnsActiveFriendly(p, 4);
                        break;
                    default:  pawns = pawnsActiveFriendly(p, 0);
                        break;
                }
            case 4:  switch(p[3].getPartner()) {
                    case 1:  pawns = pawnsActiveFriendly(p, 1);
                        break;
                    case 2:  pawns = pawnsActiveFriendly(p, 2);
                        break;
                    case 3:  pawns = pawnsActiveFriendly(p, 3);
                        break;
                    default:  pawns = pawnsActiveFriendly(p, 0);
                        break;
                }
            default:  pawns = new Pawn[1];
                    pawns[0] = temp;
        }
        return pawns;
    }

    // retourne le nombre de pions actifs du partner
    default int countActivePartner (Player[] p, int color) {return pawnsActivePartner(p, color).length;}
```

```
// retourne pions dans START
default Pawn[] pawnsOnStart (Player[] p, int color) {
	PawnFactory factory = new PawnFactory();
	Pawn temp = factory.createPawn(0, 0, DEFAULT);
	Pawn[] pawns;
	int n = 0;

	switch(color) {
		case 1:	for(int i = 0 ; i < 4 ; i++) {
				if(!p[0].getSet()[i].getIsActive()) n++;
			}
			if(n == 0) {
				pawns = new Pawn[1];
				pawns[0] = temp;
				break;
			}
			pawns = new Pawn[n];
			n = 0;
			for(int i = 0 ; i < 4 ; i++) {
				if(!p[0].getSet()[i].getIsActive()) {
					pawns[n] = p[0].getSet()[i];
					n++;
				}
			}
			break;
		case 2:	for(int i = 0 ; i < 4 ; i++) {
				if(!p[1].getSet()[i].getIsActive()) n++;
			}
			if(n == 0) {
				pawns = new Pawn[1];
				pawns[0] = temp;
				break;
			}
			pawns = new Pawn[n];
			n = 0;
			for(int i = 0 ; i < 4 ; i++) {
				if(!p[1].getSet()[i].getIsActive()) {
					pawns[n] = p[1].getSet()[i];
					n++;
				}
			}
			break;
```

```
            case 3:  for(int i = 0 ; i < 4 ; i++) {
                         if(!p[2].getSet()[i].getIsActive()) n++;
                     }
                     if(n == 0) {
                         pawns = new Pawn[1];
                         pawns[0] = temp;
                         break;
                     }
                     pawns = new Pawn[n];
                     n = 0;
                     for(int i = 0 ; i < 4 ; i++) {
                         if(!p[2].getSet()[i].getIsActive()) {
                             pawns[n] = p[2].getSet()[i];
                             n++;
                         }
                     }
                     break;
            case 4:  for(int i = 0 ; i < 4 ; i++) {
                         if(!p[3].getSet()[i].getIsActive()) n++;
                     }
                     if(n == 0) {
                         pawns = new Pawn[1];
                         pawns[0] = temp;
                         break;
                     }
                     pawns = new Pawn[n];
                     n = 0;
                     for(int i = 0 ; i < 4 ; i++) {
                         if(!p[3].getSet()[i].getIsActive()) {
                             pawns[n] = p[3].getSet()[i];
                             n++;
                         }
                     }
                     break;
            default: pawns = new Pawn[1];
                     pawns[0] = temp;
        }
        return pawns;
    }
```

```
// déplace le pion (coeur)
default boolean moveProcess (Player[] p, Board b, Pawn o, int n, int turn, boolean forw) {
	PawnFactory factory = new PawnFactory();
	Pawn temp1 = factory.createPawn(0, 0, DEFAULT);
	Pawn temp2 = factory.createPawn(0, 0, DEFAULT);
	Pawn[] psetF = pawnsActiveFriendly(p, turn);
	Pawn[] psetP = pawnsActivePartner(p, turn);
	Pawn[] psetO = pawnsActiveOpponent(p, turn);
	Coord compute = new Coord(DEFAULT);
	Coord old = new Coord(DEFAULT);
	boolean ok = false;

	// calcul de la destination
	if(forw) {
		compute.setCoord(Utils.computeCoordFW(o.getPos(), n, o.getColor()));
	}
	else {
		compute.setCoord(Utils.computeCoordBW(o.getPos(), n, turn));
	}

	// si destination est possible et pas occupée par friendly et n'est pas case fantôme
	if(!compute.equals(o.getPos()) && !b.isFriendly(o, compute) && !isNoGo(compute, turn)) {
		// vérifie si case est occupée
		if(!b.isFree(compute)) {
			// retourne pion dans START
			b.goSTART(b.getPawnFromCoord(p, compute));
		}
		// vérifie si destination est un début de slide
		if(b.isOnSlide(compute) && !b.isFriendlySlide(o, compute)) {
			// fait le "ménage" de la slide
			b.goSTART(p, compute);
			// ajoute le déplacement sur la slide
			compute.setCoord(Utils.computePawnSliding(o, compute));
		}
		// update position du pion si compute = HOME
		if(o.getColor() == 1 && compute.equals(RHOME)) ok = b.sendHOME(p[0], o);
		else if(o.getColor() == 2 && compute.equals(BHOME)) ok = b.sendHOME(p[1], o);
		else if(o.getColor() == 3 && compute.equals(YHOME)) ok = b.sendHOME(p[2], o);
		else if(o.getColor() == 4 && compute.equals(GHOME)) ok = b.sendHOME(p[3], o);
		// valide si HOME contient 4 pions, GAME OVER
		else {
			// déplace le pion
			old.setCoord(o.getPos());
			o.setPos(compute);
			if(forw) o.setCount(o.getCount() + n);
			else o.setCount(o.getCount() - 1); // carte 10
			b.update(o);
```

```
            b.update(old, 0);
        }
        if(ok) {
            // GAME OVER
            GameOver(p);
        }
        return true;
    }
    // pour mode coop
    else if(n == 11 && p[turn-1].getPartner() != 0 && psetO[0].getPos().equals(DEFAULT)) {
        // si il y a des pions partner actifs
        if(psetP.length > 0 && !psetP[0].getPos().equals(DEFAULT)) {
            // si un seul partner
            if(psetP.length == 1 && !psetP[0].getPos().equals(DEFAULT)) {
                // un seul friendly actif
                if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                    permutePawns(b, psetF[0], psetP[0]);
                }
                // si plusieurs friendly actifs
                if(psetF.length > 1) {
                    do {
                        // demander lequel échanger (friendly)
                        temp1 = askNumberSameColor(p, turn);
                    }
                    while(!isPawnInList(psetF, temp1));
                    if(isPawnInList(psetF, temp1)) {
                        permutePawns(b, temp1, psetP[0]);
                    }
                }
            }
            else {
                // si plusieurs partners
                if(psetP.length > 1) {
                    // si un seul friendly
                    if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                        do {
                            // demander lequel échanger (partner)
                            temp1 = askNumberSameColor(p, turn);
                        }
                        while(!isPawnInList(psetP, temp1));
                        if(isPawnInList(psetP, temp1)) {
                            permutePawns(b, temp1, psetF[0]);
                        }
                    }
                    // si plusieurs friendly
                    if(psetF.length > 1) {
                        do {
```

```
                                    // demander lequel échanger (friendly)
                                    temp1 = askNumberSameColor(p, turn);
                                }
                                while(!isPawnInList(psetF, temp1));
                                do {
                                    // demander lequel échanger (partner)
                                    temp2 = askNumberSameColor(p, turn);
                                }
                                while(!isPawnInList(psetP, temp2));
                                // valider pions
                                if(isPawnInList(psetF, temp1) && isPawnInList(psetP, temp2)) {
                                    permutePawns(b, temp1, temp2);
                                }
                            }
                        }
                    }
                    return true;
                }
            }
            return false;
        }

        // gestion de GAME OVER
        default void GameOver (Player[] p) {
            for(int i = 0 ; i < p.length ; i++) {
                if(p[i].countHome() == 4 && p[i].getPartner() == 0) {
                    WinnerIs(p[i], false);
                    break;
                }
                else if(p[i].countHome() == 4 && p[i].getPartner() != 0) {
                    WinnerIs(p[i], true);
                    break;
                }
            }
            System.out.println();
            // si nouvelle partie
            if(menuEndGame()) {
                Game newgame = new Game(false);
                newgame.playTurn();
            }
            // quitte le programme
            else System.exit(0);
        }
```

```
// affiche le gagnant
default void WinnerIs (Player winner, boolean hasPartner) {
    switch(winner.getColor()) {
        case 1:  System.out.println("\n\n\t\t\t\t\033[1;37m Le grand gagnant est \033[1;36m... \033[1;31mROUGE \
                 033[1;37m!\033[0m\n");
                 if(hasPartner) {
                     switch(winner.getPartner()) {
                         case 2:  System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                                  033[1;36m... \033[1;34mBLEU \033[1;37m!\033[0m\n");
                                  break;
                         case 3:  System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                                  033[1;36m... \033[1;33mJAUNE \033[1;37m!\033[0m\n");
                                  break;
                         case 4:  System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                                  033[1;36m... \033[1;32mVERT \033[1;37m!\033[0m\n");
                                  break;
                     }
                 }
                 scan.nextLine();
                 break;
        case 2:  System.out.println("\n\n\t\t\t\t\033[1;37m Le grand gagnant est \033[1;36m... \033[1;34mBLEU \
                 033[1;37m!\033[0m\n");
                 if(hasPartner) {
                     switch(winner.getPartner()) {
                         case 1:  System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                                  033[1;36m... \033[1;31mROUGE \033[1;37m!\033[0m\n");
                                  break;
                         case 3:  System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                                  033[1;36m... \033[1;33mJAUNE \033[1;37m!\033[0m\n");
                                  break;
                         case 4:  System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                                  033[1;36m... \033[1;32mVERT \033[1;37m!\033[0m\n");
                                  break;
                     }
                 }
                 scan.nextLine();
                 break;
        case 3:  System.out.println("\n\n\t\t\t\t\033[1;37m Le grand gagnant est \033[1;36m... \033[1;33mJAUNE \
                 033[1;37m!\033[0m\n");
                 if(hasPartner) {
                     switch(winner.getPartner()) {
                         case 1:  System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                                  033[1;36m... \033[1;31mROUGE \033[1;37m!\033[0m\n");
                                  break;
                         case 2:  System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                                  033[1;36m... \033[1;34mBLEU \033[1;37m!\033[0m\n");
                                  break;
```

```
                    case 4:   System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                              033[1;36m... \033[1;32mVERT \033[1;37m!\033[0m\n");
                              break;
                }
            }
            scan.nextLine();
            break;
        case 4:   System.out.println("\n\n\t\t\t\t\033[1;37m Le grand gagnant est \033[1;36m... \033[1;32mVERT \
            033[1;37m!\033[0m\n");
            if(hasPartner) {
                switch(winner.getPartner()) {
                    case 1:   System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                              033[1;36m... \033[1;31mROUGE \033[1;37m!\033[0m\n");
                              break;
                    case 2:   System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                              033[1;36m... \033[1;34mBLEU \033[1;37m!\033[0m\n");
                              break;
                    case 3:   System.out.println("\n\n\t\t\t\t\033[1;37m Le meilleur partenaire est \
                              033[1;36m... \033[1;33mJAUNE \033[1;37m!\033[0m\n");
                              break;
                }
            }
            scan.nextLine();
            break;
    }
}

// menu fin de partie
default boolean menuEndGame () {
    int choice = 0;
    System.out.println(R+ "\n\n\t\t\t\t\t**" + M + " Fin de la partie " + R + "**");
    System.out.println(G + "\t\t\t\t\t1." + R + "Nouvelle partie" + N);
    System.out.println(G + "\t\t\t\t\t2." + R + "Quitter" + N);
    System.out.print(M + "\t\t\t\t\t\tChoix : " + Y);

    do {
        try {
            choice = scan.nextInt();
        }
        catch(InputMismatchException e) {}
        scan.nextLine();
    }
    while(!Utils.isNumberValid2(choice));
    if(choice == 2) return false;
    return true;
}
```

```
    // valide présence d'un pion dans une liste de pions
    default boolean isPawnInList (Pawn[] list, Pawn o) {
        for(int i = 0 ; i < list.length ; i++) {
            if(o.equals(list[i])) return true;
        }
        return false;
    }

    // détermine si coordonnée est la case impossible pour une couleur
    default boolean isNoGo (Coord coord, int color) {
        switch(color) {
            case 1:   return coord.equals(RNOGO);
            case 2:   return coord.equals(BNOGO);
            case 3:   return coord.equals(YNOGO);
            case 4:   return coord.equals(GNOGO);
        }
        return false;
    }
}
```

“PickStrategy1.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.Pawn;

public class PickStrategy1 implements PickStrategy {
	public void algorithm (Board b, Player[] p, int n, int turn, boolean forw) {
		Pawn[] psetF, psetS;
		Pawn temp;
		boolean choice = false;
		boolean process = false;
		int loopCount;

		// recupère sets de pions
		psetF = pawnsActiveFriendly(p, turn);
		psetS = pawnsOnStart(p, turn);
		// si aucun pion actif et pions dans START
		if(p[turn-1].countStart() > 0 && psetF[0].getPos().equals(DEFAULT) && !psetS[0].getPos().equals(DEFAULT)) {
			if(!b.isFree(b.getStartingPos(turn)) && !b.isFriendly(psetS[0], b.getStartingPos(turn))) {
				// renvoie adversaire
				b.goSTART(b.getPawnFromCoord(p, b.getStartingPos(turn)));
			}
			// sortir pion
			b.getOnStart(p, turn);
		}
		else {
			// si pions actifs et pions dans START
			if(!psetF[0].getPos().equals(DEFAULT) && !psetS[0].getPos().equals(DEFAULT)) {
				// si case départ libre (ou occupée par adversaire)
				if(b.isFree(b.getStartingPos(turn)) || (!b.isFree(b.getStartingPos(turn)) && !b.isFriendly(psetS[0],
					b.getStartingPos(turn)))) {
					// si friendly sur case départ, avancer (choix imposé)
					if(b.isFriendly(psetS[0], b.getStartingPos(turn))) {
						choice = true;
					}
					// demander si avancer/sortir
					else {
						if(p[turn-1].getIsHuman()) choice = askOutOrMove(b, p, turn);
						else choice = Utils.intToBoolean(Utils.randomChoice(1));
					}
					// sortir pion
					if(!choice) {
						if(!b.isFree(b.getStartingPos(turn)) && !b.isFriendly(psetS[0], b.getStartingPos(turn))) {
							// renvoie adversaire
```

```
                b.goSTART(b.getPawnFromCoord(p, b.getStartingPos(turn)));
            }
            b.getOnStart(p, turn);
        }
        // avancer pion
        else {
            // si un seul
            if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                moveProcess(p, b, psetF[0], n, turn, forw);
            }
            // si plusieurs
            else if(psetF.length > 1) {
                if(p[turn-1].getIsHuman()) {
                    loopCount = 0;
                    do {
                        temp = askNumberSameColor(p, turn);
                        loopCount++;
                        // valide que pion est dans la liste
                        if(isPawnInList(psetF, temp)) {
                            process = moveProcess(p, b, temp, n, turn, forw);
                        }
                    }
                    while(!process && !temp.getIsHome() && loopCount < 4);
                }
                else {
                    temp = psetF[Utils.randomChoice(psetF.length-1)];
                    if(isPawnInList(psetF, temp)) {
                        process = moveProcess(p, b, temp, n, turn, forw);
                    }
                }
            }
        }
    }
    else{
        // si un seul actif, avancer
        if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
            moveProcess(p, b, psetF[0], n, turn, forw);
        }
        else {
            // si plusieurs, demander lequel, avancer
            if(psetF.length > 1) {
                if(p[turn-1].getIsHuman()) {
                    loopCount = 0;
                    do {
                        temp = askNumberSameColor(p, turn);
                        loopCount++;
                        if(isPawnInList(psetF, temp)) {
```

```
                                    process = moveProcess(p, b, temp, n, turn, forw);
                                }
                            }
                            while(!process && !temp.getIsHome() && loopCount < 4);
                        }
                        else {
                            temp = psetF[Utils.randomChoice(psetF.length-1)];
                            if(isPawnInList(psetF, temp)) {
                                process = moveProcess(p, b, temp, n, turn, forw);
                            }
                        }
                    }
                }
            }
            // si pions actifs mais aucun dans START
            else if(!psetF[0].getPos().equals(DEFAULT) && psetS[0].getPos().equals(DEFAULT)) {
                // si un seul, avancer
                if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                    moveProcess(p, b, psetF[0], n, turn, forw);
                }
                // si plusieurs, demander lequel, avancer
                else if(psetF.length > 1) {
                    if(p[turn-1].getIsHuman()) {
                        loopCount = 0;
                        do {
                            temp = askNumberSameColor(p, turn);
                            loopCount++;
                            if(isPawnInList(psetF, temp)) {
                                process = moveProcess(p, b, temp, n, turn, forw);
                            }
                        }
                        while(!process && !temp.getIsHome() && loopCount < 4);
                    }
                    else {
                        temp = psetF[Utils.randomChoice(psetF.length-1)];
                        if(isPawnInList(psetF, temp))
                            process = moveProcess(p, b, temp, n, turn, forw);
                    }
                }
            }
        }
    }
}
```

“PickStrategy2.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy2 implements PickStrategy {
	public void algorithm (Board b, Player[] p, int n, int turn, boolean forw) {
		Pawn[] psetF, psetS;
		Pawn temp;
		boolean choice = false;
		boolean process = false;
		int loopCount;

		// recupère sets de pions
		psetF = pawnsActiveFriendly(p, turn);
		psetS = pawnsOnStart(p, turn);
		// si aucun pion actif et pions dans START
		if(p[turn-1].countStart() > 0 && psetF[0].getPos().equals(DEFAULT) && !psetS[0].getPos().equals(DEFAULT)) {
			if(!b.isFree(b.getStartingPos(turn)) && !b.isFriendly(psetS[0], b.getStartingPos(turn))) {
				// renvoie adversaire
				b.goSTART(b.getPawnFromCoord(p, b.getStartingPos(turn)));
			}
			// sortir pion
			b.getOnStart(p, turn);
		}
		else {
			// si pions actifs et pions dans START
			if(!psetF[0].getPos().equals(DEFAULT) && !psetS[0].getPos().equals(DEFAULT)) {
				// si case départ libre
				if(b.isFree(b.getStartingPos(turn)) || (!b.isFree(b.getStartingPos(turn)) && !b.isFriendly(psetS[0],
					b.getStartingPos(turn)))) {
					// si friendly sur case départ avancer (choix imposé)
					if(b.isFriendly(psetS[0], b.getStartingPos(turn))) {
						choice = true;
					}
					// demander si avancer/sortir
					else {
						if(p[turn-1].getIsHuman()) choice = askOutOrMove(b, p, turn);
						else choice = Utils.intToBoolean(Utils.randomChoice(1));
					}
```

```
        // sortir pion
        if(!choice) {
            if(!b.isFree(b.getStartingPos(turn)) && !b.isFriendly(psetS[0], b.getStartingPos(turn))) {
                // renvoie adversaire
                b.goSTART(b.getPawnFromCoord(p, b.getStartingPos(turn)));
            }
            b.getOnStart(p, turn);
        }
        else {
            // avancer pion (un seul actif)
            if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                moveProcess(p, b, psetF[0], n, turn, forw);
            }
            // si plusieurs
            else if(psetF.length > 1) {
                if(p[turn-1].getIsHuman()) {
                    loopCount = 0;
                    do{
                        temp = askNumberSameColor(p, turn);
                        loopCount++;
                        // valide que pion est dans la liste
                        if(isPawnInList(psetF, temp)) {
                            process = moveProcess(p, b, temp, n, turn, forw);
                        }
                    }
                    while(!process && !temp.getIsHome() && loopCount < 4);
                }
                else {
                    temp = psetF[Utils.randomChoice(psetF.length-1)];
                    if(isPawnInList(psetF, temp)) {
                        process = moveProcess(p, b, temp, n, turn, forw);
                    }
                }
            }
        }
    }
    else{
        // si un seul actif, avancer
        if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
            moveProcess(p, b, psetF[0], n, turn, forw);
        }
        else {
            // si plusieurs, demander lequel, avancer
            if(psetF.length > 1) {
                if(p[turn-1].getIsHuman()) {
                    loopCount = 0;
                    do{
```

```
                                    temp = askNumberSameColor(p, turn);
                                    loopCount++;
                                    if(isPawnInList(psetF, temp)) {
                                        process = moveProcess(p, b, temp, n, turn, forw);
                                    }
                                }
                                while(!process && !temp.getIsHome() && loopCount < 4);
                            }
                            else {
                                temp = psetF[Utils.randomChoice(psetF.length-1)];
                                if(isPawnInList(psetF, temp)) {
                                    process = moveProcess(p, b, temp, n, turn, forw);
                                }
                            }
                        }
                    }
                }
                // si pions actifs mais aucun dans START
                else if(!psetF[0].getPos().equals(DEFAULT) && psetS[0].getPos().equals(DEFAULT)) {
                    // si un seul, avancer
                    if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                        moveProcess(p, b, psetF[0], n, turn, forw);
                    }
                    // si plusieurs, demander lequel, avancer
                    else if(psetF.length > 1) {
                        if(p[turn-1].getIsHuman()) {
                            loopCount = 0;
                            do{
                                temp = askNumberSameColor(p, turn);
                                loopCount++;
                                if(isPawnInList(psetF, temp)) {
                                    process = moveProcess(p, b, temp, n, turn, forw);
                                }
                            }
                            while(!process && !temp.getIsHome() && loopCount < 4);
                        }
                        else {
                            temp = psetF[Utils.randomChoice(psetF.length-1)];
                            if(isPawnInList(psetF, temp))
                                process = moveProcess(p, b, temp, n, turn, forw);
                        }
                    }
                }
            }
        }
    }
}
```

“PickStrategy3.java”

```
package game.cardalgo;


import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy3 implements PickStrategy {
	public void algorithm (Board b, Player[] p, int n, int turn, boolean forw) {
		PawnFactory factory = new PawnFactory();
		Pawn[] pset;
		Pawn temp = factory.createPawn(0, 0, DEFAULT);
		boolean process = false;
		int loopCount;

		// récupère set de pions actifs
		pset = pawnsActiveFriendly(p, turn);
		// si pions actifs sur le board
		if(!pset[0].getPos().equals(DEFAULT)) {
			// vérifie si un seul ou plusieurs
			if(pset.length > 0 && !pset[0].getPos().equals(DEFAULT)) {
				// un seul
				if(pset.length == 1 && !pset[0].getPos().equals(DEFAULT)) {
					moveProcess(p, b, pset[0], n, turn, true);
				}
				// plusieurs, demander quel pion avancer
				else {
					if(p[turn-1].getIsHuman()) {
						loopCount = 0;
						do {
							temp = askNumberSameColor(p, turn);
							loopCount++;
							// si pion figure dans la liste de pions, avancer
							if(isPawnInList(pset, temp)) {
								if(isPawnInList(pset, temp)) {
									process = moveProcess(p, b, temp, n, turn, true);
								}
							}
						}
						while(!process && !temp.getIsHome() && loopCount < 4);
					}
					else {
						temp = pset[Utils.randomChoice(pset.length-1)];
						if(isPawnInList(pset, temp)) {
```

```
                        if(isPawnInList(pset, temp)) {
                            process = moveProcess(p, b, temp, n, turn, true);
                        }
                    }
                }
            }
        }
    }
}
```

“PickStrategy4.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy4 implements PickStrategy {
	public void algorithm (Board b, Player[] p, int n, int turn, boolean forw) {
		PawnFactory factory = new PawnFactory();
		Pawn[] pset;
		Pawn temp = factory.createPawn(0, 0, DEFAULT);
		boolean process = false;
		int loopCount;

		// récupère set de pions actifs
		pset = pawnsActiveFriendly(p, turn);
		// si pions actifs sur le board
		if(!pset[0].getPos().equals(DEFAULT)) {
			// vérifie si un seul ou plusieurs
			if(pset.length > 0 && !pset[0].getPos().equals(DEFAULT)) {
				// un seul
				if(pset.length == 1 && !pset[0].getPos().equals(DEFAULT)) {
					moveProcess(p, b, pset[0], n, turn, true);
				}
				// plusieurs, demander quel pion avancer
				else {
					if(p[turn-1].getIsHuman()) {
						loopCount = 0;
						do {
							temp = askNumberSameColor(p, turn);
							loopCount++;
							// si pion figure dans la liste de pions, avancer
							if(isPawnInList(pset, temp)) {
								process = moveProcess(p, b, temp, n, turn, true);
							}
						}
						while(!process && !temp.getIsHome() && loopCount < 4);
					}
					else {
						temp = pset[Utils.randomChoice(pset.length-1)];
						if(isPawnInList(pset, temp)) {
							process = moveProcess(p, b, temp, n, turn, true);
						}
					}
				}
```

```
            }
        }
    }
}
```

“PickStrategy5.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy5 implements PickStrategy {
	public void algorithm (Board b, Player[] p, int n, int turn, boolean fw) {
		PawnFactory factory = new PawnFactory();
		Pawn[] pset;
		Pawn temp = factory.createPawn(0, 0, DEFAULT);
		boolean process = false;
		int loopCount;

		// récupère set de pions actifs
		pset = pawnsActiveFriendly(p, turn);
		// si pions actifs sur le board
		if(!pset[0].getPos().equals(DEFAULT)) {
			// vérifie si un seul ou plusieurs
			if(pset.length > 0 && !pset[0].getPos().equals(DEFAULT)) {
				// un seul
				if(pset.length == 1 && !pset[0].getPos().equals(DEFAULT)) {
					moveProcess(p, b, pset[0], n, turn, true);
				}
				// plusieurs, demander quel pion avancer
				else {
					if(p[turn-1].getIsHuman()) {
						loopCount = 0;
						do {
							temp = askNumberSameColor(p, turn);
							loopCount++;
							// si pion figure dans la liste de pions, avancer
							if(isPawnInList(pset, temp)) {
								process = moveProcess(p, b, temp, n, turn, true);
							}
						}
						while(!process && !temp.getIsHome() && loopCount < 4);
					}
					else {
						temp = pset[Utils.randomChoice(pset.length-1)];
						if(isPawnInList(pset, temp)) {
							process = moveProcess(p, b, temp, n, turn, true);
						}
					}
```

```
                }
            }
        }
    }
}
```

“PickStrategy7.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy7 implements PickStrategy {
    public void algorithm (Board b, Player[] p, int n, int turn, boolean fw) {
        PawnFactory factory = new PawnFactory();
        Pawn[] psetF;
        Pawn temp = factory.createPawn(0, 0, DEFAULT);
        boolean choice = false;
        boolean process = false;
        int loopCount;

        // récupère set de pions actifs
        psetF = pawnsActiveFriendly(p, turn);
        // si pions actifs sur le board
        if(!psetF[0].getPos().equals(DEFAULT)) {
            // vérifie si un seul ou plusieurs
            if(psetF.length > 0 && !psetF[0].getPos().equals(DEFAULT)) {
                // un seul
                if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                    moveProcess(p, b, psetF[0], n, turn, true);
                }
                // plusieurs, demander avancer/spliter
                else {
                    if(p[turn-1].getIsHuman()) choice = askGoOrSplit();
                    else choice = Utils.intToBoolean(Utils.randomChoice(1));
                    // si avancer
                    if(choice) {
                        if(p[turn-1].getIsHuman()) {
                            loopCount = 0;
                            do {
                                // demander quel pion avancer
                                temp = askNumberSameColor(p, turn);
                                loopCount++;
```

```
                                    // si pion figure dans la liste de pions, avancer
                                    if(isPawnInList(psetF, temp)) {
                                        process = moveProcess(p, b, temp, n, turn, true);
                                    }
                                }
                                while(!process && !temp.getIsHome() && loopCount < 4);
                            }
                            else {
                                if(Utils.intToBoolean(Utils.randomChoice(1))) {
                                    temp = psetF[Utils.randomChoice(psetF.length-1)];
                                    if(isPawnInList(psetF, temp)) {
                                        process = moveProcess(p, b, temp, n, turn, true);
                                    }
                                }
                            }
                        }
                        // si spliter
                        else {
                            menuAskForAB(b, p, turn, true);
                        }
                    }
                }
            }
        }
    }
```

“PickStrategy8.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy8 implements PickStrategy {
	public void algorithm (Board b, Player[] p, int n, int turn, boolean fw) {
		PawnFactory factory = new PawnFactory();
		Pawn[] pset;
		Pawn temp = factory.createPawn(0, 0, DEFAULT);
		boolean process = false;
		int loopCount;

		// récupère set de pions actifs
		pset = pawnsActiveFriendly(p, turn);
		// si pions actifs sur le board
		if(!pset[0].getPos().equals(DEFAULT)) {
			// vérifie si un seul ou plusieurs
			if(pset.length > 0) {
				// un seul
				if(pset.length == 1 && !pset[0].getPos().equals(DEFAULT)) {
					moveProcess(p, b, pset[0], n, turn, true);
				}
				// plusieurs, demander quel pion avancer
				else {
					if(p[turn-1].getIsHuman()) {
						loopCount = 0;
						do {
							temp = askNumberSameColor(p, turn);
							loopCount++;
							// si pion figure dans la liste de pions, avancer
							if(isPawnInList(pset, temp)) {
								process = moveProcess(p, b, temp, n, turn, true);
							}
						}
						while(!process && !temp.getIsHome() && loopCount < 4);
					}
					else {
						temp = pset[Utils.randomChoice(pset.length-1)];
						if(isPawnInList(pset, temp)) {
							process = moveProcess(p, b, temp, n, turn, true);
						}
					}
```

```
                }
            }
        }
    }
}
```

“PickStrategy10.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy10 implements PickStrategy {
    public void algorithm (Board b, Player[] p, int n,int turn, boolean fw) {
        PawnFactory factory = new PawnFactory();
        Pawn[] psetF;
        Pawn temp = factory.createPawn(0, 0, DEFAULT);
        boolean choice = false;
        boolean process = false;
        int loopCount;

        // récupère set de pions actifs
        psetF = pawnsActiveFriendly(p, turn);
        // si pions actifs sur le board
        if(!psetF[0].getPos().equals(DEFAULT)) {
            // demande pour avancer/reculer
            if(psetF.length > 0) {
                // si un seul
                if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                    // demande avancer ou reculer
                    if(!b.isOnOwnStart(psetF, turn)) {
                        if(p[turn-1].getIsHuman()) choice = askFWorBW();
                        else choice = Utils.intToBoolean(Utils.randomChoice(1));
                    }
                    else choice = true;
                    // si avancer
                    if(choice) {
                        moveProcess(p, b, psetF[0], n, turn, true);
                    }
                    // si reculer (vérifie si pion sur SA case départ)
                    else {
                        if(!b.isOnOwnStart2(psetF[0], psetF[0].getColor())) {
                            moveProcess(p, b, psetF[0], 1, turn, false);
```

```
            }
            // sinon on avance (choix imposé)
            else moveProcess(p, b, psetF[0], n, turn, true);
        }
    }
    // si plusieurs
    else {
        // demande avancer ou reculer
        if(!b.isOnOwnStart2(temp, temp.getColor())) {
            if(p[turn-1].getIsHuman()) choice = askFWorBW();
            else choice = Utils.intToBoolean(Utils.randomChoice(1));
        }
        else choice = true;
        // si avancer
        if(choice) {
            if(p[turn-1].getIsHuman()) {
                loopCount = 0;
                do {
                    // demander quel pion déplacer
                    temp = askNumberSameColor(p, turn);
                    loopCount++;
                    // valide le pion choisi
                    if(isPawnInList(psetF, temp)) {
                        process = moveProcess(p, b, temp, n, turn, true);
                    }
                }
                while(!process || (!isPawnInList(psetF, temp) && temp.getIsHome()) && loopCount < 4);
            }
            else {
                temp = psetF[Utils.randomChoice(psetF.length-1)];
                if(isPawnInList(psetF, temp)) {
                    process = moveProcess(p, b, temp, n, turn, true);
                }
            }
        }
        // si reculer
        else {
            if(p[turn-1].getIsHuman()) {
                loopCount = 0;
                do {
                    // demander quel pion déplacer
                    temp = askNumberSameColor(p, turn);
                    loopCount++;
                    // valide le pion choisi
                    if(isPawnInList(psetF, temp)) {
                        process = moveProcess(p, b, temp, 1, turn, false);
                    }
```

```
                        }
                        while(!process || (!isPawnInList(psetF, temp) && temp.getIsHome()) && loopCount < 4);
                    }
                    else {
                        temp = psetF[Utils.randomChoice(psetF.length-1)];
                        if(isPawnInList(psetF, temp)) {
                            process = moveProcess(p, b, temp, 1, turn, false);
                        }
                    }
                }
            }
        }
    }
}
```

“PickStrategy11.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy11 implements PickStrategy {
	public void algorithm (Board b, Player[] p, int n, int turn, boolean fw) {
		PawnFactory factory = new PawnFactory();
		Pawn[] psetF, psetO, psetP;
		Pawn temp1 = factory.createPawn(0, 0, DEFAULT);
		Pawn temp2 = factory.createPawn(0, 0, DEFAULT);
		boolean choice = false;
		boolean process = false;
		int loopCount;

		// récupère set de pions actifs
		psetF = pawnsActiveFriendly(p, turn);
		psetO = pawnsActiveOpponent(p, turn);
		psetP = pawnsActivePartner(p, turn);
		// si pions actifs sur le board
		if(!psetF[0].getPos().equals(DEFAULT) || !psetO[0].getPos().equals(DEFAULT)) {
			// au moins un pion friendly actif
			if(!psetF[0].getPos().equals(DEFAULT)) {
				// au moins un pion adversaire actif
				if(!psetO[0].getPos().equals(DEFAULT)) {
					// si sur case NO GO, avancer (choix imposé)
					if(psetO.length == 1 && !psetO[0].getPos().equals(DEFAULT) && !isNoGo(psetO[0].getPos(), turn)) {
						choice = true;
					}
					else{
						// demande pour avancer/permuter avec adversaire
						if(p[turn-1].getIsHuman()) choice = askGoOrTrade();
						else choice = Utils.intToBoolean(Utils.randomChoice(1));
					}
					// si on permute
					if(!choice) {
						// un seul adversaire actif
						if(psetO.length == 1 && !psetO[0].getPos().equals(DEFAULT) && !b.isSafety(psetO[0].getPos(),
						psetO[0].getColor())) {
							// un seul friendly actif
							if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT) && !
							b.isSafety(psetF[0].getPos(), psetF[0].getColor())) {
								permutePawns(b, psetF[0], psetO[0]);
							}
```

```
        // si plusieurs friendly actifs
        else if(psetF.length > 1) {
            if(p[turn-1].getIsHuman()) {
                loopCount = 0;
                do {
                    // demander lequel échanger (friendly)
                    temp1 = askNumberSameColor(p, turn);
                    loopCount++;
                }
                while(!isPawnInList(psetF, temp1) || b.isSafety(temp1.getPos(), temp1.getColor())
                && loopCount < 4);
            }
            else temp1 = psetF[Utils.randomChoice(psetF.length-1)];
            if(isPawnInList(psetF, temp1) && !b.isSafety(temp1.getPos(), temp1.getColor())) {
                permutePawns(b, temp1, psetO[0]);
            }
        }
    }
    // si plusieurs adversaires actifs
    else if(psetO.length > 1) {
        // si un seul friendly actif et pas dans SAFETY
        if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT) && !
        b.isSafety(psetF[0].getPos(), turn)) {
            if(p[turn-1].getIsHuman()) {
                loopCount = 0;
                do {
                    // demander lequel échanger (adversaire)
                    temp2 = askColorNumber(p);
                    loopCount++;
                }
                while(!isPawnInList(psetO, temp2) || b.isSafety(temp2.getPos(), temp2.getColor())
                && loopCount < 4);
            }
            else temp2 = psetF[Utils.randomChoice(psetF.length-1)];
            if(isPawnInList(psetO, temp2) && !b.isSafety(temp2.getPos(), temp2.getColor()) &&
            !b.isSafety(psetF[0].getPos(), psetF[0].getColor())) {
                permutePawns(b, psetF[0], temp2);
            }
        }
        // si plusieurs friendly actifs
        else if(psetF.length > 1) {
            if(p[turn-1].getIsHuman()) {
                loopCount = 0;
                do {
                    // demander lequel échanger (friendly)
                    temp1 = askNumberSameColor(p, turn);
                    loopCount++;
```

```
                    }
                    while(!isPawnInList(psetF, temp1) || b.isSafety(temp1.getPos(), temp1.getColor())
                    && loopCount < 4);
                    loopCount = 0;
                    do {
                        // demander lequel échanger (adversaire)
                        temp2 = askColorNumber(p);
                        loopCount++;
                    }
                    while(!isPawnInList(psetO, temp2) || b.isSafety(temp2.getPos(), temp2.getColor())
                    && loopCount < 4);
                }
                else {
                    temp1 = psetF[Utils.randomChoice(psetF.length-1)];
                    temp2 = psetO[Utils.randomChoice(psetO.length-1)];
                }
                if(isPawnInList(psetF, temp1) && !b.isSafety(temp1.getPos(), temp1.getColor()) &&
                isPawnInList(psetO, temp2) && !b.isSafety(temp2.getPos(), temp2.getColor())) {
                    permutePawns(b, temp1, temp2);
                }
            }
        }
    }
    // si on avance
    else {
        // vérifie si un seul ou plusieurs
        if(!psetF[0].getPos().equals(DEFAULT)) {
            // un seul
            if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                moveProcess(p, b, psetF[0], n, turn, true);
            }
            // plusieurs, demander quel pion avancer
            else {
                if(p[turn-1].getIsHuman()) {
                    loopCount = 0;
                    do {
                        temp1 = askNumberSameColor(p, turn);
                        loopCount++;
                        // si pion figure dans la liste de pions, avancer
                        if(isPawnInList(psetF, temp1)) {
                            process = moveProcess(p, b, temp1, n, turn, true);
                        }
                    }
                    while(!process && !temp1.getIsHome() && loopCount < 4);
                }
                else {
                    temp1 = psetF[Utils.randomChoice(psetF.length-1)];
```

```
                        if(isPawnInList(psetF, temp1)) {
                            process = moveProcess(p, b, temp1, n, turn, true);
                        }
                    }
                }
            }
        }
    }
    // si aucun adversaire actif (aussi seul point d'entrée pour action "coop")
    else {
        // si pion actifs, avancer (choix imposé)
        // au moins un pion friendly actif
        if(!psetF[0].getPos().equals(DEFAULT)) {
            // un seul friendly
            if(psetF.length == 1 && !psetF[0].getPos().equals(DEFAULT)) {
                // si mode coop
                if(p[turn-1].getPartner() != 0) {
                    // si partners
                    if(!psetP[0].getPos().equals(DEFAULT)) {
                        // si un seul pion partner
                        if(psetP.length == 1 && !psetP[0].getPos().equals(DEFAULT)){
                            // propose de permuter
                            if(p[turn-1].getIsHuman()) choice = askGoOrTrade();
                            else choice = Utils.intToBoolean(Utils.randomChoice(1));
                            // avance
                            if(choice) {
                                moveProcess(p, b, psetF[0], n, turn, true);
                            }
                            // permute
                            else {
                                permutePawns(b, psetF[0], psetP[0]);
                            }
                        }
                        // si plusieurs pions partner
                        else {
                            if(p[turn-1].getIsHuman()) choice = askGoOrTrade();
                            else choice = Utils.intToBoolean(Utils.randomChoice(1));
                            // avance
                            if(choice) {
                                moveProcess(p, b, psetF[0], n, turn, true);
                            }
                            // permute
                            else {
                                if(p[turn-1].getIsHuman()) {
                                    loopCount = 0;
                                    do{
                                        temp1 = askForPartnerPawn(p, turn, psetP);
```

```
                                    loopCount++;
                                }
                                while(!isPawnInList(psetP, temp1) && loopCount < 4);
                            }
                            else {
                                temp1 = psetP[Utils.randomChoice(psetP.length-1)];
                            }
                            if(isPawnInList(psetP, temp1)) {
                                permutePawns(b, psetF[0], temp1);
                            }
                        }
                    }
                }
            }
            // si mode normal on doit avancer
            else {
                moveProcess(p, b, psetF[0], n, turn, true);
            }
        }
        // plusieurs friendly
        else if(psetF.length > 1) {
            // propose avancer ou permuter
            if(!psetP[0].getPos().equals(DEFAULT)) {
                if(p[turn-1].getIsHuman()) choice = askGoOrTrade();
                else choice = Utils.intToBoolean(Utils.randomChoice(1));
            }
            // si aucun pion partner on avance (choix imposé)
            else choice = true;
            // si avance
            if(choice) {
                if(p[turn-1].getIsHuman()) {
                    loopCount = 0;
                    do {
                        temp1 = askNumberSameColor(p, turn);
                        loopCount++;
                    }
                    while(!temp1.getIsHome() && !isPawnInList(psetF, temp1) && loopCount < 4);
                    // si pion figure dans la liste de pions, avancer
                    loopCount = 0;
                    do {
                        if(isPawnInList(psetF, temp1)) {
                            process = moveProcess(p, b, temp1, n, turn, true);
                        }
                    loopCount++;
                    }
                    while(!process && loopCount < 4);
                }
```

```
        else {
            temp1 = psetF[Utils.randomChoice(psetF.length-1)];
            if(isPawnInList(psetF, temp1)) {
                process = moveProcess(p, b, temp1, n, turn, true);
            }
        }
    }
    // si permute
    else {
        // si un seul pion partner
        if(psetP.length == 1 && !psetP[0].getPos().equals(DEFAULT)) {
            if(p[turn-1].getIsHuman()) {
                loopCount = 0;
                do {
                    // demande quel pion friendly
                    temp1 = askNumberSameColor(p, turn);
                    loopCount++;
                }
                while(!temp1.getIsHome() && !isPawnInList(psetF, temp1) && loopCount <
                4);
            }
            else {
                temp1 = psetF[Utils.randomChoice(psetF.length-1)];
            }
            if(isPawnInList(psetF, temp1)) {
                permutePawns(b, temp1, psetP[0]);
            }
        }
        // si plusieurs pions partner
        else if(psetP.length > 1) {
            if(p[turn-1].getIsHuman()) {
                loopCount = 0;
                do {
                    // demander quel pion friendly
                    temp1 = askNumberSameColor(p, turn);
                    loopCount++;
                }
                while(!temp1.getIsHome() && !isPawnInList(psetF, temp1) && loopCount <
                4);
                loopCount = 0;
                do {
                    // demander quel pion partner
                    temp2 = askForPartnerPawn(p, turn, psetP);
                    loopCount++;
                }
                while(!temp2.getIsHome() && !isPawnInList(psetP, temp2) && loopCount <
                4);
```

```
                                        }
                                        else {
                                            temp1 = psetF[Utils.randomChoice(psetF.length-1)];
                                            temp2 = psetP[Utils.randomChoice(psetP.length-1)];
                                        }
                                        if(isPawnInList(psetF, temp1) && isPawnInList(psetP, temp2)) {
                                            permutePawns(b, temp1, temp2);
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

"PickStrategy12.java"

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy12 implements PickStrategy {
    public void algorithm (Board b, Player[] p, int n, int turn, boolean fw) {
        PawnFactory factory = new PawnFactory();
        Pawn[] pset;
        Pawn temp = factory.createPawn(0, 0, DEFAULT);
        boolean process = false;
        int loopCount;

        // récupère set de pions actifs
        pset = pawnsActiveFriendly(p, turn);
        // si pions actifs sur le board
        if(!pset[0].getPos().equals(DEFAULT)) {
            // vérifie si un seul ou plusieurs
            if(pset.length > 0) {
                // un seul
                if(pset.length == 1) {
                    moveProcess(p, b, pset[0], n, turn, true);
                }
```

```
                    // plusieurs, demander quel pion avancer
                    else {
                        if(p[turn-1].getIsHuman()) {
                            loopCount = 0;
                            do{
                                temp = askNumberSameColor(p, turn);
                                loopCount++;
                                // si pion figure dans la liste de pions, avancer
                                if(isPawnInList(pset, temp)) {
                                    if(isPawnInList(pset, temp)) {
                                        process = moveProcess(p, b, temp, n, turn, true);
                                    }
                                }
                            }
                            while(!process && !temp.getIsHome() && loopCount < 4);
                        }
                        else {
                            temp = pset[Utils.randomChoice(pset.length-1)];
                            if(isPawnInList(pset, temp)) {
                                if(isPawnInList(pset, temp)) {
                                    process = moveProcess(p, b, temp, n, turn, true);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

“PickStrategy13.java”

```
package game.cardalgo;

import board.*;
import player.Player;
import util.Utils;
import piece.*;

public class PickStrategy13 implements PickStrategy {
	public void algorithm (Board b, Player[] p, int n, int turn, boolean forw) {
		PawnFactory factory = new PawnFactory();
		Pawn temp = factory.createPawn(0, 0, DEFAULT);
		Pawn[] psetO;
		Pawn[] psetS;
		boolean choice = false;
		int choiceInt = 2;
		Coord dest = new Coord(DEFAULT);
		int loopCount;

		switch(turn) {
			case 1:	dest.setCoord(RSTART);
				break;
			case 2:	dest.setCoord(BSTART);
				break;
			case 3:	dest.setCoord(YSTART);
				break;
			case 4:	dest.setCoord(GSTART);
				break;
		}

		// récupère les pions actifs
		psetO = pawnsActiveOpponent(p, turn);
		psetS = pawnsOnStart(p, turn);

		// en mode auto, si aucun adversaire et case départ non libre, garder
		// si départ libre, sortir
		// si les deux, random

		// demande garder/utiliser
		if(p[turn-1].getIsHuman()) choice = askKeepOrElse();
		else choice = Utils.intToBoolean(Utils.randomChoice(1));
		// si garder joker
		if(choice) {
			p[turn-1].setJoker(b.getPickedCard());
		}
```

```
// si utiliser
else {
	// si aucun adversaire actif
	if(psetO[0].getPos().equals(DEFAULT)) {
		// si des pions dans START
		if(!psetS[0].getPos().equals(DEFAULT)) {
			// si case départ libre
			if(b.isFree(b.getStartingPos(turn))) {
				// sortir
				if(p[turn-1].getIsHuman()) b.getOnStart(p, turn);
				else {
					if(Utils.intToBoolean(Utils.randomChoice(1))) {
						b.getOnStart(p, turn);
					}
					// sinon rien, pour maximiser l'expérience "random"
				}
			}
			// si occupée mais par adversaire
			else if(b.isStartFree(turn)) {
				if(p[turn-1].getIsHuman())
					// renvoyer adversaire
					if(isPawnInList(psetO, b.getPawnFromCoord(p, b.getStartingPos(turn)))) {
						b.goSTART(b.getPawnFromCoord(p, b.getStartingPos(turn)));
						b.getOnStart(p, turn);
					}
				}
				else {
					if(Utils.intToBoolean(Utils.randomChoice(1))) {
						if(isPawnInList(psetO, b.getPawnFromCoord(p, b.getStartingPos(turn)))) {
							b.goSTART(b.getPawnFromCoord(p, b.getStartingPos(turn)));
							b.getOnStart(p, turn);
						}
					}
				}
			}
		}
		// si adversaires actifs
		else if(!psetO[0].getPos().equals(DEFAULT)) {
			// si aucun pion dans START
			if(psetS[0].getPos().equals(DEFAULT)) {
				// si un seul adversaire
				if(psetO.length == 1 && !psetO[0].getPos().equals(DEFAULT)) {
					if(p[turn-1].getIsHuman()) {
						// renvoyer ce pion
						b.goSTART(psetO[0]);
					}
					else {
```

```
                    if(Utils.intToBoolean(Utils.randomChoice(1))) {
                        b.goSTART(psetO[0]);
                    }
                }
            }
            // si plusieurs adversaires
            else if(psetO.length > 1) {
                if(p[turn-1].getIsHuman()) {
                    loopCount = 0;
                    do {
                        temp = askColorNumber(p);
                        loopCount++;
                        if(isPawnInList(psetO, temp) && !b.isSafety(psetO[0].getPos(), temp.getColor())) {
                            b.goSTART(temp);
                        }
                    }
                    while(!isPawnInList(psetO, temp) && temp.getIsHome() && loopCount < 4);
                }
                else {
                    if(Utils.intToBoolean(Utils.randomChoice(1))) {
                        temp = psetO[Utils.randomChoice(psetO.length-1)];
                        if(isPawnInList(psetO, temp) && !b.isSafety(psetO[0].getPos(), temp.getColor())) {
                            b.goSTART(temp);
                        }
                    }
                }
            }
        }
    }
    // si adversaires actifs et pions dans START
    if(!psetO[0].getPos().equals(DEFAULT) && !psetS[0].getPos().equals(DEFAULT)) {
        if(p[turn-1].getIsHuman()) {
            // demander si sortir ou renvoyer
            choiceInt = askOutOrKick(b, p, turn);
        }
        else {
            choiceInt = Utils.randomChoice(1);
        }
    }
    // si sortir
    if(choiceInt == 1) {
        // si case départ libre
        if(b.isFree(b.getStartingPos(turn))) {
            if(p[turn-1].getIsHuman()) {
                // sortir
                b.getOnStart(p, turn);
            }
```

```
                else {
                    if(Utils.intToBoolean(Utils.randomChoice(1))) {
                        b.getOnStart(p, turn);
                    }
                }
            }
            // si occupée mais par adversaire
            else if(!b.isFree(b.getStartingPos(turn)) && !b.isFriendly(psetS[0], b.getStartingPos(turn))) {
                if(p[turn-1].getIsHuman()) {
                    // renvoyer adversaire
                    if(isPawnInList(psetO, b.getPawnFromCoord(p, b.getStartingPos(turn)))) {
                        b.goSTART(b.getPawnFromCoord(p, b.getStartingPos(turn)));
                        b.getOnStart(p, turn);
                    }
                }
                else {
                    if(Utils.intToBoolean(Utils.randomChoice(1))) {
                        if(isPawnInList(psetO, b.getPawnFromCoord(p, b.getStartingPos(turn)))) {
                            b.goSTART(b.getPawnFromCoord(p, b.getStartingPos(turn)));
                            b.getOnStart(p, turn);
                        }
                    }
                }
            }
        }
        // si renvoyer
        else if(choiceInt == 0) {
            // si un seul adversaire
            if(psetO.length == 1 && !psetO[0].getPos().equals(DEFAULT) && !b.isSafety(psetO[0].getPos(),
            psetO[0].getColor())) {
                if(p[turn-1].getIsHuman()) {
                    // renvoyer ce pion
                    b.goSTART(psetO[0]);
                }
                else {
                    if(Utils.intToBoolean(Utils.randomChoice(1))) {
                        b.goSTART(psetO[0]);
                    }
                }
            }
            // si plusieurs adversaires
            else if(psetO.length > 1) {
                if(p[turn-1].getIsHuman()) {
                    loopCount = 0;
                    do {
                        temp = askColorNumber(p);
                        loopCount++;
```

```
                    }
                    while(!isPawnInList(psetO, temp) && loopCount < 4);
                    if(isPawnInList(psetO, temp)) {
                        b.goSTART(temp);
                    }
                }
                else {
                    if(Utils.intToBoolean(Utils.randomChoice(1))) {
                        temp = psetO[Utils.randomChoice(psetO.length-1)];
                        if(isPawnInList(psetO, temp)) {
                            b.goSTART(temp);
                        }
                    }
                }
            }
        }
    }
}
```

"PickStrategyDEFAULT.java"

```
package game.cardalgo;

import board.Board;
import player.Player;

public class PickStrategyDEFAULT implements PickStrategy {
    public void algorithm (Board b, Player[] p, int n, int turn, boolean forw) {}
}
```

“Game.java”

```
package game;

import board.Board;
import game.cardalgo.*;
import player.Player;
import util.Utils;
import java.util.Scanner;

public class Game implements PickStrategy {
	private GameSetup setup;
	private Player[] p;
	private Board b;
	private Scanner scan;
	private PickContext context;
	private PickStrategy strategy;

	public Game (boolean splash) {
		Utils.Clear();
		setup = new GameSetup();
		setup.setup(splash);
		Utils.Clear();
		p = setup.getPlayers();
		b = setup.getBoard();
		b.setTurn(setup.getTurn());
		scan = new Scanner(System.in);
	}

	// fonction principale (tour à jouer)
	public void playTurn () {
		while(true) {
			Utils.Clear();
			displayBoard();
			// vérifier si joker et actions
			jokerManager(p,b);
			//
			pleasePick();
			pickCard();
			invokeOp();
			Utils.Clear();
			displayBoard();
			if(b.getPickedCard().getNumber() != 2) {
				b.setTurn();
			}
			pleaseContinue();
		}
	}
```

```
// gestion du joker si présent
private void jokerManager (Player[] p, Board b) {
	// vérifier si joker et actions
	if(!Utils.getPlayer(p, b.getTurn()).isJokerNULL()) {
		if(Utils.getPlayer(p, b.getTurn()).countJoker() > 0) {
			// si "human"
			if(p[b.getTurn()-1].getIsHuman()) {
				// demander si utiliser avant de piger
				if(askUseJoker() == 1) {
					context = new PickContext(new PickStrategy13(), b, p, 13, b.getTurn(), true);
					b.setPickContext(context);
					b.movePawn();
					drop();
				}
			}
			// si "auto"
			else {
				context = new PickContext(new PickStrategy13(), b, p, 13, b.getTurn(), true);
				b.setPickContext(context);
				b.movePawn();
				drop();
			}
		}
	}
}

// remet le joker dans dropDeck
private void drop () {b.dropJoker(p[b.getTurn()-1].getJoker());}

// attente de saisie clavier pour piger
private void pleasePick () {
	if(p[b.getTurn()-1].getIsHuman()) {
		switch(b.getTurn()) {
			case 1:	System.out.print("\n\t\t\t\t\t" + R + " Piger une carte ..." + N);
				break;
			case 2:	System.out.print("\n\t\t\t\t\t" + B + " Piger une carte ..." + N);
				break;
			case 3:	System.out.print("\n\t\t\t\t\t" + Y + " Piger une carte ..." + N);
				break;
			case 4:	System.out.print("\n\t\t\t\t\t" + G + " Piger une carte ..." + N);
				break;
			default:	break;
		}
		scan.nextLine();
	}
}
```

```
// attente de saisie clavier pour continuer
private void pleaseContinue () {
	if(p[b.getTurn()-1].getIsHuman()) {
		System.out.print("\n\t\t\t\t" + M + " Appuyer sur ENTRÉE pour jouer" + C + " ..." + N);
		scan.nextLine();
	}
}

// attente de saisie clavier pour joker (utiliser ou non)
private int askUseJoker () {
	int choice = 0;

	if(p[b.getTurn()-1].getIsHuman()) {
		System.out.println("\n\t\t\t\t\t" + M + "Utiliser JOKER ?");
		System.out.println("\n\t\t\t\t\t" + G + "1." + R + "Oui" + N);
		System.out.println("\n\t\t\t\t\t" + G + "2." + R + "Non" + N);
		do {
			System.out.print("\n\t\t\t\t\t\t" + B + "Choix : " + Y);
			choice = scan.nextInt();
			scan.nextLine();
		}
		while(!Utils.isNumberValid2(choice));
		return choice;
	}
	else {
		do {
			choice = Utils.randomChoice(2);
		}
		while(choice < 1 || choice > 2);
		return choice;
	}
}

// pige une carte et affiche sur le board
private void pickCard () {
	b.pickCard();
	b.dropCard();
	Utils.Clear();
	displayBoard();
}

// déplacement du pion (invoque PickContext.operation)
private void invokeOp () { // utiliser board!!!!!!!!!!!!!!!!!!!!
	context = new PickContext(setStrategy(), b, p, b.getPickedCard().getNumber(), b.getTurn(), true);
	b.setPickContext(context);
	b.movePawn();
}
```

```
    // configuration de la stratégie
    private PickStrategy setStrategy () {
        switch(b.getPickedCard().getNumber()) {
            case 1:   strategy = new PickStrategy1();
                      break;
            case 2:   strategy = new PickStrategy2();
                      break;
            case 3: strategy = new PickStrategy3();
                      break;
            case 4:   strategy = new PickStrategy4();
                      break;
            case 5:   strategy = new PickStrategy5();
                      break;
            case 7:   strategy = new PickStrategy7();
                      break;
            case 8:   strategy = new PickStrategy8();
                      break;
            case 10: strategy = new PickStrategy10();
                      break;
            case 11: strategy = new PickStrategy11();
                      break;
            case 12: strategy = new PickStrategy12();
                      break;
            case 13: strategy = new PickStrategy13();
                      break;
            default:  strategy = new PickStrategyDEFAULT();
        }
        return strategy;
    }

    // affiche le(s) board(s)
    private void displayBoard () {
        b.displayBoard(p, b.getTurn());
        // pour les tests
        //b.displaySituation();
    }

    // implementation vide (pour interface PickStrategy)
    public void algorithm (Board b, Player[] p, int n, int turn, boolean forw) {}
}
```

“GameConstants.java”

```
package game;

import board.Coord;

public interface GameConstants {
	public static final Coord RBASE1 = new Coord(12, 10);
	public static final Coord RBASE2 = new Coord(12, 11);
	public static final Coord RBASE3 = new Coord(13, 10);
	public static final Coord RBASE4 = new Coord(13, 11);

	public static final Coord BBASE1 = new Coord(10, 3);
	public static final Coord BBASE2 = new Coord(11, 3);
	public static final Coord BBASE3 = new Coord(10, 2);
	public static final Coord BBASE4 = new Coord(11, 2);

	public static final Coord YBASE1 = new Coord(3, 5);
	public static final Coord YBASE2 = new Coord(3, 4);
	public static final Coord YBASE3 = new Coord(2, 5);
	public static final Coord YBASE4 = new Coord(2, 4);

	public static final Coord GBASE1 = new Coord(5, 12);
	public static final Coord GBASE2 = new Coord(4, 12);
	public static final Coord GBASE3 = new Coord(5, 13);
	public static final Coord GBASE4 = new Coord(4, 13);

	public static final Coord RSHORT_START = new Coord(15, 14);
	public static final Coord RSHORT_END = new Coord(15, 11);
	public static final Coord RLONG_START = new Coord(15, 6);
	public static final Coord RLONG_END = new Coord(15, 2);

	public static final Coord BSHORT_START = new Coord(14, 0);
	public static final Coord BSHORT_END = new Coord(11, 0);
	public static final Coord BLONG_START = new Coord(6, 0);
	public static final Coord BLONG_END = new Coord(2, 0);

	public static final Coord YSHORT_START = new Coord(0, 1);
	public static final Coord YSHORT_END = new Coord(0, 4);
	public static final Coord YLONG_START = new Coord(0, 9);
	public static final Coord YLONG_END = new Coord(0, 13);

	public static final Coord GSHORT_START = new Coord(1, 15);
	public static final Coord GSHORT_END = new Coord(4, 15);
	public static final Coord GLONG_START = new Coord(9, 15);
	public static final Coord GLONG_END = new Coord(13, 15);
```

```
public static final Coord RSTART = RSHORT_END;
public static final Coord BSTART = BSHORT_END;
public static final Coord YSTART = YSHORT_END;
public static final Coord GSTART = GSHORT_END;

public static final Coord RSAFETY1 = new Coord(14, 13);
public static final Coord RSAFETY2 = new Coord(13, 13);
public static final Coord RSAFETY3 = new Coord(12, 13);
public static final Coord RSAFETY4 = new Coord(11, 13);
public static final Coord RSAFETY5 = new Coord(10, 13);

public static final Coord BSAFETY1 = new Coord(13, 1);
public static final Coord BSAFETY2 = new Coord(13, 2);
public static final Coord BSAFETY3 = new Coord(13, 3);
public static final Coord BSAFETY4 = new Coord(13, 4);
public static final Coord BSAFETY5 = new Coord(13, 5);

public static final Coord YSAFETY1 = new Coord(1, 2);
public static final Coord YSAFETY2 = new Coord(2, 2);
public static final Coord YSAFETY3 = new Coord(3, 2);
public static final Coord YSAFETY4 = new Coord(4, 2);
public static final Coord YSAFETY5 = new Coord(5, 2);

public static final Coord GSAFETY1 = new Coord(2, 14);
public static final Coord GSAFETY2 = new Coord(2, 13);
public static final Coord GSAFETY3 = new Coord(2, 12);
public static final Coord GSAFETY4 = new Coord(2, 11);
public static final Coord GSAFETY5 = new Coord(2, 10);

// case précédente aux safeties
public static final Coord R_TURNLIMIT = new Coord(15, 13);
public static final Coord B_TURNLIMIT = new Coord(13, 0);
public static final Coord Y_TURNLIMIT = new Coord(0, 2);
public static final Coord G_TURNLIMIT = new Coord(2, 15);

public static final Coord RHOME = new Coord(9, 13);
public static final Coord BHOME = new Coord(13, 6);
public static final Coord YHOME = new Coord(6, 2);
public static final Coord GHOME = new Coord(2, 9);

public static final Coord RNOGO = new Coord(15, 12);
public static final Coord BNOGO = new Coord(12, 0);
public static final Coord YNOGO = new Coord(0, 3);
public static final Coord GNOGO = new Coord(3, 15);

public static final Coord DEFAULT = new Coord(9, 9);
public static final Coord INFINITE = new Coord(7, 7);
```

```
    public static final int RED = 1;
    public static final int BLUE = 2;
    public static final int YELLOW = 3;
    public static final int GREEN = 4;
    public static final int EMPTY = 0;

    public static final String R = "\033[1;31m"; // fg rouge
    public static final String B = "\033[1;34m"; // fg bleu
    public static final String Y = "\033[1;33m"; // fg jaune
    public static final String G = "\033[1;32m"; // fg vert
    public static final String E = "\033[1;95m"; // fg bright magenta
    public static final String M = "\033[1;35m"; // fg magenta
    public static final String C = "\033[1;36m"; // fg cyan
    public static final String N = "\033[0m"; // normal (reset)
    public static final String W = "\033[1;97m"; // fg blanc
    public static final String BK = "\033[1;30m"; // fg noir

    public static final String RR = "\033[1;101m"; // bg rouge
    public static final String BB = "\033[1;104m"; // bg bleu
    public static final String YY = "\033[1;103m"; // bg jaune
    public static final String GG = "\033[1;102m"; // bg vert
}
```

“GameSetup.java”

```
package game;

import board.*;
import player.*;
import player.creator.*;
import util.Utils;
import java.util.InputMismatchException;

import java.util.Scanner;

public class GameSetup implements GameConstants {
	private Board b;
	private Player[] p;
	private Player p1, p2, p3, p4;
	private int turn;
	private Scanner scan;
	private int mode;

	public GameSetup () {
		Init();
	}

	private void Init () {
		mode = -1;
		scan = new Scanner(System.in);
		// création du board
		b = new Board();
		// création des joueurs (et des pions)
		AbstractCreator creator = new CreatorRED();
		p1 = creator.createPlayer();
		creator = new CreatorBLUE();
		p2 = creator.createPlayer();
		creator = new CreatorYELLOW();
		p3 = creator.createPlayer();
		creator = new CreatorGREEN();
		p4 = creator.createPlayer();
		Player[] p_ = {p1, p2, p3, p4};
		p = p_;
		// update de situation pour initialiser les positions des pions
		for(int i = 0 ; i < 4 ; i++) p[i].updateSituation(b);
	}
```

```
// fonction principale (menus)
public void setup (boolean splash) {
	if(splash) showSplash();
	mode = menuMode(); // 1: normal et 2: coop
	menuPlayers();
	if(mode == 2) menuCoopPairing();
	menuFirstToPlay();
}

// getters - setters
public int getTurn () {return turn;}
public int getMode () {return mode;}
public Player[] getPlayers () {return p;}
public Board getBoard () {return b;}

// menu MODE
private int menuMode () {
	int choice = 0;

	System.out.println(R + "** " + E + "Sélectionner le mode" + R + " **" + N);
	System.out.println(G + "1. " + E + "Mode " + C + "NORMAL" + N);
	System.out.println(G + "2. " + E + "Mode " + C + "COOP" + N);
	do {
		System.out.print(B + "\tChoix : " + Y);
		try {
			choice = scan.nextInt();
		}
		catch(InputMismatchException e) {}
		scan.nextLine();
	}
	while(!Utils.isNumberValid2(choice));
	return choice;
}

// menu joueurs
private void menuPlayers () {
	String choice;

	System.out.println(R + "\n** " + E + "Sélectionner les joueurs ([" + C + "H" + E + "]umain ou [" + C + "A" + E +
	"]uto)" + R + " **" + N);
	System.out.print(R + "ROUGE " + E + "[" + C + "H" + E + "] / [" + C + "A" + E + "] : " + Y);
	do {choice = scan.nextLine().toLowerCase();}
	while(!typeCheck(choice));
	setType(p[0], choice);

	System.out.print(B + "BLEU " + E + "[" + C + "H" + E + "] / [" + C + "A" + E + "] : " + Y);
	do {choice = scan.nextLine().toLowerCase();}
```

```
        while(!typeCheck(choice));
        setType(p[1], choice);

        System.out.print(Y + "JAUNE " + E + "[" + C + "H" + E + "] / [" + C + "A" + E + "] : " + Y);
        do {choice = scan.nextLine().toLowerCase();}
        while(!typeCheck(choice));
        setType(p[2], choice);

        System.out.print(G + "VERT " + E + "[" + C + "H" + E + "] / [" + C + "A" + E + "] : " + Y);
        do {choice = scan.nextLine().toLowerCase();}
        while(!typeCheck(choice));
        setType(p[3], choice);
    }

    // valide la saisie du type (H/A)
    private boolean typeCheck (String s) {return s.equals("h") || s.equals("a");}

    // update les joueurs (H/A)
    private void setType (Player player, String s) {
        if(s.equals("h")) player.setIsHuman(true);
        else player.setIsHuman(false);
    }

    // menu de jumelage Coop (retourne la couleur avec laquelle ROUGE est jumelée)
    private void menuCoopPairing () {
        int choice = 0;
        System.out.println(R + "\n** " + E + "Sélectionner partenaire de " + E + RR + "ROUGE" + N + R + " **" + N);
        System.out.println(G + "1. " + B + "BLEU" + N);
        System.out.println(G + "2. " + Y + "JAUNE" + N);
        System.out.println(G + "3. " + G + "VERT" + N);
        do {
            System.out.print(B + "\tChoix : " + Y);
            try {
                choice = scan.nextInt();
            }
            catch(InputMismatchException e) {}
            scan.nextLine();
        }
        while(!Utils.isNumberValid3(choice));
        computeTeams(choice);
    }
```

```
// calcul des équipes pour mode Coop
private void computeTeams (int redPartner) {
	p[0].setPartner(redPartner);
	p[redPartner].setPartner(1);
	switch(redPartner) {
		case 2:	p[2].setPartner(4);
			p[3].setPartner(3);
			break;
		case 3:	p[1].setPartner(4);
			p[3].setPartner(2);
			break;
		case 4:	p[1].setPartner(3);
			p[2].setPartner(2);
			break;
	}
}

// menu sélection du premier joueur (retourne couleur)
private void menuFirstToPlay () {
	int[] num = new int[4];

	System.out.println(R + "\n** " + E + "Sélectionner le premier joueur" + R + " **" + N);
	System.out.print(R + "ROUGE" + N);
	num[0] = Utils.getRandomInt();
	System.out.print(C + "\t" + num[0] + "\t");
	System.out.print(M + "Appuyer sur ENTRÉE ... ");
	scan.nextLine();
	System.out.print(B + "BLEU" + N);
	num[1] = Utils.getRandomInt();
	System.out.print(C + "\t" + num[1] + "\t");
	System.out.print(M + "Appuyer sur ENTRÉE ... ");
	scan.nextLine();
	System.out.print(Y + "JAUNE" + N);
	num[2] = Utils.getRandomInt();
	System.out.print(C + "\t" + num[2] + "\t");
	System.out.print(M + "Appuyer sur ENTRÉE ... ");
	scan.nextLine();
	System.out.print(G + "VERT" + N);
	num[3] = Utils.getRandomInt();
	System.out.print(C + "\t" + num[3] + "\t");
	System.out.print(M + "Appuyer sur ENTRÉE ... ");
	scan.nextLine();
	int j = 0; // retient l'index du max
	for(int i = 0 ; i < num.length-1 ; i++) {
		if(num[j] < num[i+1]) j = i+1;
	}
	turn = j+1;
```

```
            System.out.println();
            switch(turn) {
                case 1:   System.out.println(E + "\tLe joueur " + R + "ROUGE" + E + " débute !" + N);
                          System.out.print(M + "\tAppuyer sur ENTRÉE" + R + "..." + N);
                          break;
                case 2:   System.out.println(E + "\tLe joueur " + B + "BLEU" + E + " débute !" + N);
                          System.out.print(M + "\tAppuyer sur ENTRÉE" + B + "..." + N);
                          break;
                case 3:   System.out.println(E + "\tLe joueur " + Y + "JAUNE" + E + " débute !" + N);
                          System.out.print(M + "\tAppuyer sur ENTRÉE" + Y + "..." + N);
                          break;
                case 4:   System.out.println(E + "\tLe joueur " + G + "VERT" + E + " débute !" + N);
                          System.out.print(M + "\tAppuyer sur ENTRÉE" + G + "..." + N);
                          break;
                default:  break;
            }
            scan.nextLine();
        }

        // splashscreen
        private void showSplash () {
            System.out.println(R + "  ######  " + B + "  ########  " + Y + "  ########  " + G + "  ########  " + C + "
#### #### " + G + "  ######  ");
            System.out.println(R + "  ##############  " + B + "  ################  " + Y + "  ##############  "
+ G + "  ##############  " + C + "  ######  ######  " + G + "  ########");
            System.out.println(R + "  ##################  " + B + "  ####################  " + Y + "
##################  " + G + "  ##################  " + C + "  ######  ######  " + G + "  ########");
            System.out.println(R + "  ########  ######  " + B + "  ########  ########  " + Y + "  ########  ######
" + G + "  ########  ######  " + C + "  ######  ######  " + G + "  ########");
            System.out.println(R + "######  " + B + "  ######  ######  " + Y + "  ######  ######  " + G + "
######  ######  " + C + "  ######  ######  " + G + "  ########");
            System.out.println(R + "#####  " + B + "  ######  ######  " + Y + "  ######  ######  " + G + "
######  ######  " + C + "  ######  ######  " + G + "  ########");
            System.out.println(R + "#####  " + B + "  ######  ######  " + Y + "  ######  ######  " + G + "
######  ######  " + C + "  ######  ######  " + G + "  ########");
            System.out.println(R + "######  " + B + "  ######  ######  " + Y + "  ######  ######  " + G + "
######  ######  " + C + "  ######  ######  " + G + "  ########");
            System.out.println(R + "########  " + B + "  ######  ######  " + Y + "  ######  ######  " + G + "
######  ######  " + C + "  ######  #####  " + G + "  ########");
            System.out.println(R + "  ##########  " + B + "  ######  ######  " + Y + "  ########  ########  " + G +
" ########  ########  " + C + "  ######  #####  " + G + "  ########");
            System.out.println(R + "  ################  " + B + "  ######  ######  " + Y + "
####################  " + G + "  ####################  " + C + "  ############  " + G + "
########");
            System.out.println(R + "  ################  " + B + "  ######  ######  " + Y + "
##################  " + G + "  ##################  " + C + "  ########  " + G + "  ########");
```

```
            System.out.println(R + " ######## " + B + " ###### ###### " + Y + " ###### ###### " + G + "
###### ###### " + C + " ###### " + G + " ########");
            System.out.println(R + " ######## " + B + " ###### ###### " + Y + " ###### ###### " + G + "
###### ###### " + C + " ###### " + G + " ########");
            System.out.println(R + " ###### " + B + " ###### ###### " + Y + " ###### ###### " + G + "
###### ###### " + C + " ###### " + G + " ##### ");
            System.out.println(R + " ###### " + B + " ###### ###### " + Y + " ###### ###### " + G + "
###### ###### " + C + " ###### " + G + " ### ");
            System.out.println(R + " ###### " + B + " ###### ###### " + Y + " ###### ###### " + G + "
###### ###### " + C + " ###### ");
            System.out.println(R + "###### ######## " + B + " ######## ######## " + Y + " ###### ##### " +
G + " ###### ##### " + C + " ###### ");
            System.out.println(R + " ################### " + B + " ################### " + Y + "
###### #####" + G + " ###### #####" + C + " ###### " + R + " #### ");
            System.out.println(R + " ############### " + B + " ############### " + Y + " ###### #####" +
G + " ###### #####" + C + " ###### " + R + " ###### ");
            System.out.println(R + " ######## " + B + " ######## " + Y + " ###### #####" + G + " ######
#####" + C + " ###### " + R + " #### ");
            System.out.println();
            System.out.print(M + "\t\t\t\t\t\t\tAppuyer sur ENTRÉE ..." + N);
            scan.nextLine();
            Utils.Clear();
        }
}
```

“Pawn.java”

```
package piece;

import board.Coord;
import game.GameConstants;

public class Pawn implements GameConstants {
	private int number;
	private int count;
	private int color;
	private boolean isActive;
	private boolean isHome;
	private Coord pos;

	// getters et setters
	public int getNumber () {return number;}
	public int getCount () {return count;}
	public void setCount (int n) {count = n;}
	public int getColor () {return color;}
	public void setColor (int n) {color = n;}
	public boolean getIsActive () {return isActive;}
	public void setIsActive (boolean n) {isActive = n;}
	public boolean getIsHome () {return isHome;}
	public void setIsHome (boolean n) {isHome = n;}
	public Coord getPos () {return pos;}
	public void setPos (Coord coord) {pos.setCoord(coord);}
	public void setPos (int X, int Y) {pos.setX(X); pos.setY(Y);}

	// constructeur
	public Pawn (int num, int color_, Coord coord) {
		number = num;
		count = 0;
		color = color_;
		isActive = false;
		isHome = false;
		pos = coord;
	}

	// utilisé comme valeur par défaut pour les tableaux de Pawn[]
	public Pawn () {pos = DEFAULT;}

	public boolean equals (Pawn o) {
		if(o == null) return false;
		return o.isActive == isActive && o.isHome == isHome && o.pos == pos &&
		o.color == color && o.count == count && o.number == number;
	}
}
```

“PawnFactory.java”

```
package piece;

import board.Coord;

public class PawnFactory {
	public Pawn createPawn (int number, int color, Coord coord) {
		return new Pawn(number, color, coord);
	}
}
```

“AbstractCreator.java”

```
package player.creator;

import player.*;

public abstract class AbstractCreator {
	public abstract Player factoryMethod ();

	public Player createPlayer () {
		return factoryMethod();
	}
}
```

“CreatorBLUE.java”

```
package player.creator;

import player.*;

public class CreatorBLUE extends AbstractCreator {
	public Player factoryMethod () {
		return new PlayerBLUE();
	}
}
```

“CreatorGREEN.java”

```
package player.creator;

import player.*;

public class CreatorGREEN extends AbstractCreator {
	public Player factoryMethod () {
		return new PlayerGREEN();
	}
}
```

“CreatorRED.java”

```
package player.creator;

import player.*;

public class CreatorRED extends AbstractCreator {
	public Player factoryMethod () {
		return new PlayerRED();
	}
}
```

“CreatorYELLOW.java”

```
package player.creator;

import player.*;

public class CreatorYELLOW extends AbstractCreator {
	public Player factoryMethod () {
		return new PlayerYELLOW();
	}
}
```

“Player.java”

```
package player;

import piece.Pawn;
import board.Board;
import game.GameConstants;
import deck.Card;

public interface Player extends GameConstants {
	abstract int getColor ();
	abstract void setIsHuman (boolean b);
	abstract boolean getIsHuman ();
	abstract void setIsActive (boolean b);
	abstract boolean getIsActive ();
	abstract Pawn[] getSet ();
	abstract boolean equals (Player o);
	abstract int countHome ();
	abstract int countStart ();
	abstract void updateSituation (Board b);
	abstract void initPawnsOnStart ();
	abstract int getPartner ();
	abstract void setPartner (int partner_);
	abstract Card getJoker ();
	abstract void setJoker (Card joker);
	abstract int countJoker ();
	abstract boolean isJokerNULL ();

	// renvoie un set de pions actifs sur le board
	default Pawn[] getActiveSet () {
		int n = 0;
		for(int i = 0 ; i < 4 ; i++) {
			if(getSet()[i].getIsActive()) n++;
		}
		Pawn[] as = new Pawn[n];
		n = 0;
		for(int i = 0 ; i < 4 ; i++) {
			if(getSet()[i].getIsActive()) {
				as[n] = getSet()[i];
				n++;
			}
		}
		return as;
	}
}
```

“PlayerDEFAULT.java”

```
package player;

import board.*;
import piece.*;
import deck.Card;
import java.util.*;

public class PlayerDEFAULT implements Player{
	private int color = 2;
	private int partner = 0;
	private int sorry = 0;
	private boolean isActive = false;
	private boolean isHuman = false;
	private Pawn[] pawnSet = new Pawn[4];
	private LinkedList<Card> jokers;

	public PlayerDEFAULT () {}

	public void initPawnsOnStart () {}
	public boolean equals (Player o) {return false;}
	public int countHome () {return 0;}
	public void updateSituation (Board b) {}

	public Pawn getNextPawnToGetOut () {
		PawnFactory factory = new PawnFactory();
		Pawn next = factory.createPawn(0, 0, DEFAULT);
		return next;
	}

	public int getSorry () {return sorry;}
	public void incrSorry () {}
	public void decrSorry () {}
	public int getColor () {return color;}
	public void setIsHuman (boolean b) {}
	public boolean getIsHuman () {return isHuman;}
	public void setIsActive (boolean b) {}
	public int countStart () {return 0;}
	public boolean getIsActive () {return isActive;}
	public Pawn[] getSet () {return pawnSet;}
	public int getPartner () {return partner;}
	public void setPartner (int partner_) {}
	public void setJoker (Card joker) {}
	public Card getJoker () {return jokers.getFirst();}
	public int countJoker () {return jokers.size();}
```

```
    public boolean isJokerNULL () {
        if(jokers == null) return true;
        return false;
    }
}
```

"PlayerBLUE.java"

```
package player;

import board.*;
import piece.*;
import deck.Card;
import java.util.*;

public class PlayerBLUE implements Player {
    private int color = 2;
    private int partner = 0;
    private boolean isActive = false;
    private boolean isHuman = false;
    private Pawn[] pawnSet = new Pawn[4];
    private PawnFactory factory = new PawnFactory();
    private LinkedList<Card> jokers;

    public PlayerBLUE () {
        for(int i = 0 ; i < pawnSet.length ; i++) {
            pawnSet[i] = factory.createPawn(i+1, color, new Coord(DEFAULT));
        }
        initPawnsOnStart();
        jokers = new LinkedList<Card>();
    }

    public void initPawnsOnStart () {
        pawnSet[0].setPos(BBASE1);
        pawnSet[1].setPos(BBASE2);
        pawnSet[2].setPos(BBASE3);
        pawnSet[3].setPos(BBASE4);
    }

    public boolean equals (Player o) {
        return color == o.getColor() && isActive == o.getIsActive() && isHuman == o.getIsHuman() && partner ==
        o.getPartner();
    }
```

```
	public int countHome () {
		int n = 0;
		for(int i = 0 ; i < 4 ; i++) {
			if(pawnSet[i].getIsHome()) n++;
		}
		return n;
	}

	public int countStart () {
		int n = 0;
		for(int i = 0 ; i < 4 ; i++) {
			if(!pawnSet[i].getIsActive() && !pawnSet[i].getIsHome()) n++;
		}
		return n;
	}

	public void updateSituation (Board b) {
		for(int i = 0 ; i < 4 ; i++) b.update(pawnSet[i]);
	}

	public int getColor () {return color;}
	public void setIsHuman (boolean b) {isHuman = b;}
	public boolean getIsHuman () {return isHuman;}
	public void setIsActive (boolean b) {isActive = b;}
	public boolean getIsActive () {return isActive;}
	public Pawn[] getSet () {return pawnSet;}
	public int getPartner () {return partner;}
	public void setPartner (int partner_) {partner = partner_;}
	public void setJoker (Card joker) {jokers.add(joker);}
	public int countJoker () {return jokers.size();}
	public boolean isJokerNULL () {
		if(jokers == null || jokers.isEmpty()) return true;
		return false;
	}
	public Card getJoker () {
		if(!jokers.isEmpty()) {
			Card temp = jokers.getFirst();
			jokers.remove(0);
			return temp;
		}
		return new Card(0);
	}
}
```

“PlayerGREEN.java”

```
package player;

import board.*;
import piece.*;
import deck.Card;
import java.util.*;

public class PlayerGREEN implements Player {
	private int color = 4;
	private int partner = 0;
	private boolean isActive = false;
	private boolean isHuman = false;
	private Pawn[] pawnSet = new Pawn[4];
	private PawnFactory factory = new PawnFactory();
	private LinkedList<Card> jokers;

	public PlayerGREEN () {
		for(int i = 0 ; i < pawnSet.length ; i++) {
			pawnSet[i] = factory.createPawn(i+1, color, new Coord(DEFAULT));
		}
		initPawnsOnStart();
		jokers = new LinkedList<Card>();
	}

	public void initPawnsOnStart () {
		pawnSet[0].setPos(GBASE1);
		pawnSet[1].setPos(GBASE2);
		pawnSet[2].setPos(GBASE3);
		pawnSet[3].setPos(GBASE4);
	}

	public boolean equals (Player o) {
		return color == o.getColor() && isActive == o.getIsActive() && isHuman == o.getIsHuman() && partner ==
		o.getPartner();
	}

	public int countHome () {
		int n = 0;
		for(int i = 0 ; i < 4 ; i++) {
			if(pawnSet[i].getIsHome()) n++;
		}
		return n;
	}
```

```
    public int countStart () {
        int n = 0;
        for(int i = 0 ; i < 4 ; i++) {
            if(!pawnSet[i].getIsActive() && !pawnSet[i].getIsHome()) n++;
        }
        return n;
    }

    public void updateSituation (Board b) {
        for(int i = 0 ; i < 4 ; i++) b.update(pawnSet[i]);
    }
    public int getColor () {return color;}
    public void setIsHuman (boolean b) {isHuman = b;}
    public boolean getIsHuman () {return isHuman;}
    public void setIsActive (boolean b) {isActive = b;}
    public boolean getIsActive () {return isActive;}
    public Pawn[] getSet () {return pawnSet;}
    public int getPartner () {return partner;}
    public void setPartner (int partner_) {partner = partner_;}
    public void setJoker (Card joker) {jokers.add(joker);}
    public int countJoker () {return jokers.size();}
    public boolean isJokerNULL () {
        if(jokers == null || jokers.isEmpty()) return true;
        return false;
    }
    public Card getJoker () {
        if(!jokers.isEmpty()) {
            Card temp = jokers.getFirst();
            jokers.remove(0);
            return temp;
        }
        return new Card(0);
    }
}
```

“PlayerRED.java”

```
package player;

import board.*;
import piece.*;
import deck.Card;
import java.util.*;

public class PlayerRED implements Player {
	private int color = 1;
	private int partner = 0;
	private boolean isActive = false;
	private boolean isHuman = false;
	private Pawn[] pawnSet = new Pawn[4];
	private PawnFactory factory = new PawnFactory();
	private LinkedList<Card> jokers;

	public PlayerRED () {
		for(int i = 0 ; i < pawnSet.length ; i++) {
			pawnSet[i] = factory.createPawn(i+1, color, new Coord(DEFAULT));
		}
		initPawnsOnStart();
		jokers = new LinkedList<Card>();
	}

	public void initPawnsOnStart () {
		pawnSet[0].setPos(RBASE1);
		pawnSet[1].setPos(RBASE2);
		pawnSet[2].setPos(RBASE3);
		pawnSet[3].setPos(RBASE4);
	}

	public boolean equals (Player o) {
		if(o == null) return false;
		return color == o.getColor() && isActive == o.getIsActive() && isHuman == o.getIsHuman() && partner ==
		o.getPartner();
	}

	public int countHome () {
		int n = 0;
		for(int i = 0 ; i < 4 ; i++) {
			if(pawnSet[i].getIsHome()) n++;
		}
		return n;
	}
```

```
    public int countStart () {
        int n = 0;
        for(int i = 0 ; i < 4 ; i++) {
            if(!pawnSet[i].getIsActive() && !pawnSet[i].getIsHome()) n++;
        }
        return n;
    }

    public void updateSituation (Board b) {
        for(int i = 0 ; i < 4 ; i++) b.update(pawnSet[i]);
    }
    public int getColor () {return color;}
    public void setIsHuman (boolean b) {isHuman = b;}
    public boolean getIsHuman () {return isHuman;}
    public void setIsActive (boolean b) {isActive = b;}
    public boolean getIsActive () {return isActive;}
    public Pawn[] getSet () {return pawnSet;}
    public int getPartner () {return partner;}
    public void setPartner (int partner_) {partner = partner_;}
    public void setJoker (Card joker) {jokers.add(joker);}
    public int countJoker () {return jokers.size();}
    public boolean isJokerNULL () {
        if(jokers == null || jokers.isEmpty()) return true;
        return false;
    }
    public Card getJoker () {
        if(!jokers.isEmpty()) {
            Card temp = jokers.getFirst();
            jokers.remove(0);
            return temp;
        }
        return new Card(0);
    }
}
```

“PlayerYELLOW.java”

```
package player;

import board.*;
import piece.*;
import deck.Card;
import java.util.*;

public class PlayerYELLOW implements Player {
	private int color = 3;
	private int partner = 0;
	private boolean isActive = false;
	private boolean isHuman = false;
	private Pawn[] pawnSet = new Pawn[4];
	private PawnFactory factory = new PawnFactory();
	private LinkedList<Card> jokers;

	public PlayerYELLOW () {
		for(int i = 0 ; i < pawnSet.length ; i++) {
			pawnSet[i] = factory.createPawn(i+1, color, new Coord(DEFAULT));
		}
		initPawnsOnStart();
		jokers = new LinkedList<Card>();
	}

	public void initPawnsOnStart () {
	pawnSet[0].setPos(YBASE1);
	pawnSet[1].setPos(YBASE2);
	pawnSet[2].setPos(YBASE3);
	pawnSet[3].setPos(YBASE4);
	}

	public boolean equals (Player o) {
		return color == o.getColor() && isActive == o.getIsActive() && isHuman == o.getIsHuman() && partner ==
		o.getPartner();
	}

	public int countHome () {
		int n = 0;
		for(int i = 0 ; i < 4 ; i++) {
			if(pawnSet[i].getIsHome()) n++;
		}
		return n;
	}
```

```
    public int countStart () {
        int n = 0;
        for(int i = 0 ; i < 4 ; i++) {
            if(!pawnSet[i].getIsActive() && !pawnSet[i].getIsHome()) n++;
        }
        return n;
    }

    public void updateSituation (Board b) {
        for(int i = 0 ; i < 4 ; i++) b.update(pawnSet[i]);
    }
    public int getColor () {return color;}
    public void setIsHuman (boolean b) {isHuman = b;}
    public boolean getIsHuman () {return isHuman;}
    public void setIsActive (boolean b) {isActive = b;}
    public boolean getIsActive () {return isActive;}
    public Pawn[] getSet () {return pawnSet;}
    public int getPartner () {return partner;}
    public void setPartner (int partner_) {partner = partner_;}
    public void setJoker (Card joker) {jokers.add(joker);}
    public int countJoker () {return jokers.size();}
    public boolean isJokerNULL () {
        if(jokers == null || jokers.isEmpty()) return true;
        return false;
    }
    public Card getJoker () {
        if(!jokers.isEmpty()) {
            Card temp = jokers.getFirst();
            jokers.remove(0);
            return temp;
        }
        return new Card(0);
    }
}
```

“Utils.java”

```
package util;

import java.io.IOException;
import board.Coord;
import piece.Pawn;
import game.GameConstants;
import player.*;

public class Utils implements GameConstants {
	public static void Clear () {
		try {
			if(System.getProperty("os.name").contains("Windows")) {
				new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
			}
			else {
				System.out.print("\033\143");
			}
		}
		catch (IOException | InterruptedException ex) {}
	}

	// retourne #safety d'une coordonnée (sinon 0)
	public static int coordSafetyNumber (Coord coord) {
		if(coord.equals(RSAFETY1) || coord.equals(BSAFETY1) || coord.equals(YSAFETY1) || coord.equals(GSAFETY1))
			return 1;
		else if(coord.equals(RSAFETY2) || coord.equals(BSAFETY2) || coord.equals(YSAFETY2) || coord.equals(GSAFETY2))
			return 2;
		else if(coord.equals(RSAFETY3) || coord.equals(BSAFETY3) || coord.equals(YSAFETY3) || coord.equals(GSAFETY3))
			return 3;
		else if(coord.equals(RSAFETY4) || coord.equals(BSAFETY4) || coord.equals(YSAFETY4) || coord.equals(GSAFETY4))
			return 4;
		else if(coord.equals(RSAFETY5) || coord.equals(BSAFETY5) || coord.equals(YSAFETY5) || coord.equals(GSAFETY5))
			return 5;
		else return 0;
	}
```

```
// calcule les coordonnées vers l'avant
public static Coord computeCoordFW (Coord start, int deckNum, int color) {
	Coord end = new Coord(DEFAULT);
	int x = start.getX();
	int y = start.getY();
	int n = 0;

	// si coordonnée est dans un safety
	if(start.equals(RSAFETY1) || start.equals(RSAFETY2) || start.equals(RSAFETY3) ||
	start.equals(RSAFETY4) || start.equals(RSAFETY5) || start.equals(BSAFETY1) ||
	start.equals(BSAFETY2) || start.equals(BSAFETY3) || start.equals(BSAFETY4) ||
	start.equals(BSAFETY5) || start.equals(YSAFETY1) || start.equals(YSAFETY2) ||
	start.equals(YSAFETY3) || start.equals(YSAFETY4) || start.equals(YSAFETY5) ||
	start.equals(GSAFETY1) || start.equals(GSAFETY2) || start.equals(GSAFETY3) ||
	start.equals(GSAFETY4) || start.equals(GSAFETY5)) {
		switch(color) {
			case 1:	if(start.equals(RSAFETY1)) {
					switch(deckNum) {
						case 1:	end.setCoord(RSAFETY2);
							break;
						case 2:	end.setCoord(RSAFETY3);
							break;
						case 3:	end.setCoord(RSAFETY4);
							break;
						case 4:	end.setCoord(RSAFETY5);
							break;
						case 5:	end.setCoord(RHOME);
							break;
						default:	end.setCoord(start);
					}
				}
				else if(start.equals(RSAFETY2)) {
					switch(deckNum) {
						case 1:	end.setCoord(RSAFETY3);
							break;
						case 2:	end.setCoord(RSAFETY4);
							break;
						case 3:	end.setCoord(RSAFETY5);
							break;
						case 4:	end.setCoord(RHOME);
							break;
						default:	end.setCoord(start);
					}
				}
				else if(start.equals(RSAFETY3)) {
					switch(deckNum) {
						case 1:	end.setCoord(RSAFETY4);
```

```
                        break;
                case 2: end.setCoord(RSAFETY5);
                        break;
                case 3: end.setCoord(RHOME);
                        break;
                default: end.setCoord(start);
            }
        }
        else if(start.equals(RSAFETY4)) {
            switch(deckNum) {
                case 1: end.setCoord(RSAFETY5);
                        break;
                case 2: end.setCoord(RHOME);
                        break;
                default: end.setCoord(start);
            }
        }
        else if(start.equals(RSAFETY5)) {
            switch(deckNum) {
                case 1: end.setCoord(RHOME);
                        break;
                default: end.setCoord(start);
            }
        }
case 2: if(start.equals(BSAFETY1)) {
            switch(deckNum) {
                case 1: end.setCoord(BSAFETY2);
                        break;
                case 2: end.setCoord(BSAFETY3);
                        break;
                case 3: end.setCoord(BSAFETY4);
                        break;
                case 4: end.setCoord(BSAFETY5);
                        break;
                case 5: end.setCoord(BHOME);
                        break;
                default: end.setCoord(start);
            }
        }
        else if(start.equals(BSAFETY2)) {
            switch(deckNum) {
                case 1: end.setCoord(BSAFETY3);
                        break;
                case 2: end.setCoord(BSAFETY4);
                        break;
                case 3: end.setCoord(BSAFETY5);
                        break;
```

```
                    case 4:   end.setCoord(BHOME);
                              break;
                    default:  end.setCoord(start);
                }
            }
            else if(start.equals(BSAFETY3)) {
                switch(deckNum) {
                    case 1:   end.setCoord(BSAFETY4);
                              break;
                    case 2:   end.setCoord(BSAFETY5);
                              break;
                    case 3:   end.setCoord(BHOME);
                              break;
                    default:  end.setCoord(start);
                }
            }
            else if(start.equals(BSAFETY4)) {
                switch(deckNum) {
                    case 1:   end.setCoord(BSAFETY5);
                              break;
                    case 2:   end.setCoord(BHOME);
                              break;
                    default:  end.setCoord(start);
                }
            }
            else if(start.equals(BSAFETY5)) {
                switch(deckNum) {
                    case 1:   end.setCoord(BHOME);
                              break;
                    default:  end.setCoord(start);
                }
            }
        case 3:   if(start.equals(YSAFETY1)) {
                switch(deckNum) {
                    case 1:   end.setCoord(YSAFETY2);
                              break;
                    case 2:   end.setCoord(YSAFETY3);
                              break;
                    case 3:   end.setCoord(YSAFETY4);
                              break;
                    case 4:   end.setCoord(YSAFETY5);
                              break;
                    case 5:   end.setCoord(YHOME);
                              break;
                    default:  end.setCoord(start);
                }
            }
```

```
                else if(start.equals(YSAFETY2)) {
                    switch(deckNum) {
                        case 1:  end.setCoord(YSAFETY3);
                                 break;
                        case 2:  end.setCoord(YSAFETY4);
                                 break;
                        case 3:  end.setCoord(YSAFETY5);
                                 break;
                        case 4:  end.setCoord(YHOME);
                                 break;
                        default: end.setCoord(start);
                    }
                }
                else if(start.equals(YSAFETY3)) {
                    switch(deckNum) {
                        case 1:  end.setCoord(YSAFETY4);
                                 break;
                        case 2:  end.setCoord(YSAFETY5);
                                 break;
                        case 3:  end.setCoord(YHOME);
                                 break;
                        default: end.setCoord(start);
                    }
                }
                else if(start.equals(YSAFETY4)) {
                    switch(deckNum) {
                        case 1:  end.setCoord(YSAFETY5);
                                 break;
                        case 2:  end.setCoord(YHOME);
                                 break;
                        default: end.setCoord(start);
                    }
                }
                else if(start.equals(YSAFETY5)) {
                    switch(deckNum) {
                        case 1:  end.setCoord(YHOME);
                                 break;
                        default: end.setCoord(start);
                    }
                }
            case 4:  if(start.equals(GSAFETY1)) {
                    switch(deckNum) {
                        case 1:  end.setCoord(GSAFETY2);
                                 break;
                        case 2:  end.setCoord(GSAFETY3);
                                 break;
                        case 3:  end.setCoord(GSAFETY4);
```

```
                    break;
                case 4:  end.setCoord(GSAFETY5);
                    break;
                case 5:  end.setCoord(GHOME);
                    break;
                default: end.setCoord(start);
            }
        }
        else if(start.equals(GSAFETY2)) {
            switch(deckNum) {
                case 1:  end.setCoord(GSAFETY3);
                    break;
                case 2:  end.setCoord(GSAFETY4);
                    break;
                case 3:  end.setCoord(GSAFETY5);
                    break;
                case 4:  end.setCoord(GHOME);
                    break;
                default: end.setCoord(start);
            }
        }
        else if(start.equals(GSAFETY3)) {
            switch(deckNum) {
                case 1:  end.setCoord(GSAFETY4);
                    break;
                case 2:  end.setCoord(GSAFETY5);
                    break;
                case 3:  end.setCoord(GHOME);
                    break;
                default: end.setCoord(start);
            }
        }
        else if(start.equals(GSAFETY4)) {
            switch(deckNum) {
                case 1:  end.setCoord(GSAFETY5);
                    break;
                case 2:  end.setCoord(GHOME);
                    break;
                default: end.setCoord(start);
            }
        }
        else if(start.equals(GSAFETY5)) {
            switch(deckNum) {
                case 1:  end.setCoord(GHOME);
                    break;
                default: end.setCoord(start);
            }
```

```
                    }
            }
        }
        // segment A-B (x = 0)
        else if(x == 0) {
            if(y < 3 && color == YELLOW) {
                if(y == 2) {
                    switch(deckNum) {
                        case 1:   end.setCoord(YSAFETY1);
                                  break;
                        case 2:   end.setCoord(YSAFETY2);
                                  break;
                        case 3:   end.setCoord(YSAFETY3);
                                  break;
                        case 4:   end.setCoord(YSAFETY4);
                                  break;
                        case 5:   end.setCoord(YSAFETY5);
                                  break;
                        case 6:   end.setCoord(YHOME);
                                  break;
                        default:  end.setCoord(start);
                    }
                }
                else if(y == 1) {
                    switch(deckNum) {
                        case 1:   end.setCoord(0, 2);
                                  break;
                        case 2:   end.setCoord(YSAFETY1);
                                  break;
                        case 3:   end.setCoord(YSAFETY2);
                                  break;
                        case 4:   end.setCoord(YSAFETY3);
                                  break;
                        case 5:   end.setCoord(YSAFETY4);
                                  break;
                        case 6:   end.setCoord(YSAFETY5);
                                  break;
                        case 7:   end.setCoord(YHOME);
                                  break;
                        default:  end.setCoord(start);
                    }
                }
                else if(y == 0) {
                    switch(deckNum) {
                        case 1:   end.setCoord(0, 1);
                                  break;
                        case 2:   end.setCoord(0, 2);
```

```
                    break;
                case 3:  end.setCoord(YSAFETY1);
                    break;
                case 4:  end.setCoord(YSAFETY2);
                    break;
                case 5:  end.setCoord(YSAFETY3);
                    break;
                case 6:  end.setCoord(YSAFETY4);
                    break;
                case 7:  end.setCoord(YSAFETY5);
                    break;
                case 8:  end.setCoord(YHOME);
                    break;
                default: end.setCoord(start);
            }
        }
    }
    else {
        if(y + deckNum <= 15) end.setCoord(0, y + deckNum);
        else {
            n = deckNum - (15 - y);
            if(color == GREEN && n <= 2) {
                end.setCoord(n, 15);
            }
            else if(color == GREEN && n > 2) {
                switch(n-2) {
                    case 1:  end.setCoord(GSAFETY1);
                        break;
                    case 2:  end.setCoord(GSAFETY2);
                        break;
                    case 3:  end.setCoord(GSAFETY3);
                        break;
                    case 4:  end.setCoord(GSAFETY4);
                        break;
                    case 5:  end.setCoord(GSAFETY5);
                        break;
                    case 6:  end.setCoord(GHOME);
                        break;
                    default: end.setCoord(start);
                }
            }
            else end.setCoord(n, 15);
        }
    }
}
// segment B-C (y = 15)
else if(y == 15) {
```

```
if(x < 3 && color == GREEN) {
    if(x == 2) {
        switch(deckNum) {
            case 1:  end.setCoord(GSAFETY1);
                     break;
            case 2:  end.setCoord(GSAFETY2);
                     break;
            case 3:  end.setCoord(GSAFETY3);
                     break;
            case 4:  end.setCoord(GSAFETY4);
                     break;
            case 5:  end.setCoord(GSAFETY5);
                     break;
            case 6:  end.setCoord(GHOME);
                     break;
            default : end.setCoord(start);
        }
    }
    else if(x == 1) {
        switch(deckNum) {
            case 1:  end.setCoord(13, 0);
                     break;
            case 2:  end.setCoord(GSAFETY1);
                     break;
            case 3:  end.setCoord(GSAFETY2);
                     break;
            case 4:  end.setCoord(GSAFETY3);
                     break;
            case 5:  end.setCoord(GSAFETY4);
                     break;
            case 6:  end.setCoord(GSAFETY5);
                     break;
            case 7:  end.setCoord(GHOME);
                     break;
            default : end.setCoord(start);
        }
    }
    else if(x == 0) {
        switch(deckNum) {
            case 1:  end.setCoord(14, 0);
                     break;
            case 2:  end.setCoord(13, 0);
                     break;
            case 3:  end.setCoord(GSAFETY1);
                     break;
            case 4:  end.setCoord(GSAFETY2);
                     break;
```

```
                    case 5:   end.setCoord(GSAFETY3);
                              break;
                    case 6:   end.setCoord(GSAFETY4);
                              break;
                    case 7:   end.setCoord(GSAFETY5);
                              break;
                    case 8:   end.setCoord(GHOME);
                              break;
                    default : end.setCoord(start);
               }
          }
     }
     else {
          if(x + deckNum <= 15) end.setCoord(x+deckNum, 15);
          else {
               n = deckNum - (15 - x);
               if(color == RED && n <= 2) {
                    end.setCoord(15, 15-n);
               }
               else if(color == RED && n > 2) {
                    switch(n-2) {
                         case 1:   end.setCoord(RSAFETY1);
                                   break;
                         case 2:   end.setCoord(RSAFETY2);
                                   break;
                         case 3:   end.setCoord(RSAFETY3);
                                   break;
                         case 4:   end.setCoord(RSAFETY4);
                                   break;
                         case 5:   end.setCoord(RSAFETY5);
                                   break;
                         case 6:   end.setCoord(RHOME);
                                   break;
                         default:  end.setCoord(start);
                    }
               }
               else end.setCoord(15, 15-n);
          }
     }
}
// segment C-D (x = 15)
else if(x == 15) {
     if(y > 12 && color == RED) {
          if(15 - y == 2) {
               switch(deckNum) {
                    case 1:   end.setCoord(RSAFETY1);
                              break;
```

```
            case 2:   end.setCoord(RSAFETY2);
                      break;
            case 3:   end.setCoord(RSAFETY3);
                      break;
            case 4:   end.setCoord(RSAFETY4);
                      break;
            case 5:   end.setCoord(RSAFETY5);
                      break;
            case 6:   end.setCoord(RHOME);
                      break;
            default : end.setCoord(start);
        }
    }
    else if(15 - y == 1) {
        switch(deckNum) {
            case 1:   end.setCoord(15, 13);
                      break;
            case 2:   end.setCoord(RSAFETY1);
                      break;
            case 3:   end.setCoord(RSAFETY2);
                      break;
            case 4:   end.setCoord(RSAFETY3);
                      break;
            case 5:   end.setCoord(RSAFETY4);
                      break;
            case 6:   end.setCoord(RSAFETY5);
                      break;
            case 7:   end.setCoord(RHOME);
                      break;
            default : end.setCoord(start);
        }
    }
    else if(15 - y == 0) {
        switch(deckNum) {
            case 1:   end.setCoord(15, 14);
                      break;
            case 2:   end.setCoord(15, 13);
                      break;
            case 3:   end.setCoord(RSAFETY1);
                      break;
            case 4:   end.setCoord(RSAFETY2);
                      break;
            case 5:   end.setCoord(RSAFETY3);
                      break;
            case 6:   end.setCoord(RSAFETY4);
                      break;
            case 7:   end.setCoord(RSAFETY5);
```

```
                        break;
                case 8:   end.setCoord(RHOME);
                        break;
                default : end.setCoord(start);
            }
        }
    }
    else {
        if(y - deckNum >= 0) end.setCoord(15, y-deckNum);
        else {
            n = deckNum - y;
            If(color == BLUE && n <= 2) {
                end.setCoord(15-n, 0);
            }
            else if(color == BLUE && n > 2) {
                switch(n-2) {
                    case 1:   end.setCoord(BSAFETY1);
                            break;
                    case 2:   end.setCoord(BSAFETY2);
                            break;
                    case 3:   end.setCoord(BSAFETY3);
                            break;
                    case 4:   end.setCoord(BSAFETY4);
                            break;
                    case 5:   end.setCoord(BSAFETY5);
                            break;
                    case 6:   end.setCoord(BHOME);
                            break;
                    default:  end.setCoord(start);
                }
            }
            else end.setCoord(15-n, 0);
        }
    }
}
// segment D-A (y = 0)
else if(y == 0) {
    if(x > 12 && color == BLUE) {
        if(15 - x == 2) {
            switch(deckNum) {
                case 1:   end.setCoord(BSAFETY1);
                        break;
                case 2:   end.setCoord(BSAFETY2);
                        break;
                case 3:   end.setCoord(BSAFETY3);
                        break;
                case 4:   end.setCoord(BSAFETY4);
```

```
                    break;
            case 5:  end.setCoord(BSAFETY5);
                    break;
            case 6:  end.setCoord(BHOME);
                    break;
            default : end.setCoord(start);
        }
    }
    else if(15 - x == 1) {
        switch(deckNum) {
            case 1:  end.setCoord(13, 0);
                    break;
            case 2:  end.setCoord(BSAFETY1);
                    break;
            case 3:  end.setCoord(BSAFETY2);
                    break;
            case 4:  end.setCoord(BSAFETY3);
                    break;
            case 5:  end.setCoord(BSAFETY4);
                    break;
            case 6:  end.setCoord(BSAFETY5);
                    break;
            case 7:  end.setCoord(BHOME);
                    break;
            default : end.setCoord(start);
        }
    }
    else if(15 - x == 0) {
        switch(deckNum) {
                case 1:  end.setCoord(14, 0);
                        break;
                case 2:  end.setCoord(13, 0);
                        break;
                case 3:  end.setCoord(BSAFETY1);
                        break;
                case 4:  end.setCoord(BSAFETY2);
                        break;
                case 5:  end.setCoord(BSAFETY3);
                        break;
                case 6:  end.setCoord(BSAFETY4);
                        break;
                case 7:  end.setCoord(BSAFETY5);
                        break;
                case 8:  end.setCoord(BHOME);
                        break;
                default : end.setCoord(start);
        }
```

```
                }
            }
            else {
                if(x - deckNum >= 0) end.setCoord(x-deckNum, 0);
                else {
                    n = deckNum - x;
                    if(color == YELLOW && n <= 2) {
                        end.setCoord(0, n);
                    }
                    else if(color == YELLOW && n > 2) {
                        switch(n-2) {
                            case 1:   end.setCoord(YSAFETY1);
                                      break;
                            case 2:   end.setCoord(YSAFETY2);
                                      break;
                            case 3:   end.setCoord(YSAFETY3);
                                      break;
                            case 4:   end.setCoord(YSAFETY4);
                                      break;
                            case 5:   end.setCoord(YSAFETY5);
                                      break;
                            case 6:   end.setCoord(YHOME);
                                      break;
                            default:  end.setCoord(start);
                        }
                    }
                    else end.setCoord(0, n);
                }
            }
        }
        if(end.equals(DEFAULT)) end.setCoord(start);
        return end;
    }

    // calcule les coordonnées à reculons
    public static Coord computeCoordBW (Coord start, int deckNum, int color) {
        Coord end = new Coord(DEFAULT);
        int x = start.getX();
        int y = start.getY();
        int n = 0;
        // si coordonnée est dans un safety
        if(start.equals(RSAFETY1) || start.equals(RSAFETY2) || start.equals(RSAFETY3) ||
        start.equals(RSAFETY4) || start.equals(RSAFETY5) || start.equals(BSAFETY1) ||
        start.equals(BSAFETY2) || start.equals(BSAFETY3) || start.equals(BSAFETY4) ||
        start.equals(BSAFETY5) || start.equals(YSAFETY1) || start.equals(YSAFETY2) ||
        start.equals(YSAFETY3) || start.equals(YSAFETY4) || start.equals(YSAFETY5) ||
        start.equals(GSAFETY1) || start.equals(GSAFETY2) || start.equals(GSAFETY3) ||
```

```
start.equals(GSAFETY4) || start.equals(GSAFETY5)) {
    switch(color) {
        case 1:  if(start.equals(RSAFETY1)) {
                     switch(deckNum) {
                         case 1:  end.setCoord(15, 13);
                                  break;
                         default: end.setCoord(start);
                     }
                 }
                 else if(start.equals(RSAFETY2)) {
                     switch(deckNum) {
                         case 1:  end.setCoord(RSAFETY1);
                                  break;
                         default: end.setCoord(start);
                     }
                 }
                 else if(start.equals(RSAFETY3)) {
                     switch(deckNum) {
                         case 1:  end.setCoord(RSAFETY2);
                                  break;
                         default: end.setCoord(start);
                     }
                 }
                 else if(start.equals(RSAFETY4)) {
                     switch(deckNum) {
                         case 1:  end.setCoord(RSAFETY3);
                                  break;
                         default: end.setCoord(start);
                     }
                 }
                 else if(start.equals(RSAFETY5)) {
                     switch(deckNum) {
                         case 1:  end.setCoord(RSAFETY4);
                                  break;
                         default: end.setCoord(start);
                     }
                 }
        case 2:  if(start.equals(BSAFETY1)) {
                     switch(deckNum) {
                         case 1:  end.setCoord(13, 0);
                                  break;
                         default: end.setCoord(start);
                     }
                 }
                 else if(start.equals(BSAFETY2)) {
                     switch(deckNum) {
                         case 1:  end.setCoord(BSAFETY1);
```

```
                    break;
                default:  end.setCoord(start);
            }
        }
        else if(start.equals(BSAFETY3)) {
            switch(deckNum) {
                case 1:  end.setCoord(BSAFETY2);
                    break;
                default:  end.setCoord(start);
            }
        }
        else if(start.equals(BSAFETY4)) {
            switch(deckNum) {
                case 1:  end.setCoord(BSAFETY3);
                    break;
                default:  end.setCoord(start);
            }
        }
        else if(start.equals(BSAFETY5)) {
            switch(deckNum) {
                case 1:  end.setCoord(BSAFETY4);
                    break;
                default:  end.setCoord(start);
            }
        }
    case 3:  if(start.equals(YSAFETY1)) {
            switch(deckNum) {
                case 1:  end.setCoord(0, 2);
                    break;
                default:  end.setCoord(start);
            }
        }
        else if(start.equals(YSAFETY2)) {
            switch(deckNum) {
                case 1:  end.setCoord(YSAFETY1);
                    break;
                default:  end.setCoord(start);
            }
        }
        else if(start.equals(YSAFETY3)) {
            switch(deckNum) {
                case 1:  end.setCoord(YSAFETY2);
                    break;
                default:  end.setCoord(start);
            }
        }
        else if(start.equals(YSAFETY4)) {
```

```
                switch(deckNum) {
                    case 1:  end.setCoord(YSAFETY3);
                             break;
                    default: end.setCoord(start);
                }
            }
            else if(start.equals(YSAFETY5)) {
                switch(deckNum) {
                    case 1:  end.setCoord(YSAFETY4);
                             break;
                    default: end.setCoord(start);
                }
            }
        case 4:  if(start.equals(GSAFETY1)) {
                switch(deckNum) {
                    case 1:  end.setCoord(2, 15);
                             break;
                    default: end.setCoord(start);
                }
            }
            else if(start.equals(GSAFETY2)) {
                switch(deckNum) {
                    case 1:  end.setCoord(GSAFETY1);
                             break;
                    default: end.setCoord(start);
                }
            }
            else if(start.equals(GSAFETY3)) {
                switch(deckNum) {
                    case 1:  end.setCoord(GSAFETY2);
                             break;
                    default: end.setCoord(start);
                }
            }
            else if(start.equals(GSAFETY4)) {
                switch(deckNum) {
                    case 1:  end.setCoord(GSAFETY3);
                             break;
                    default: end.setCoord(start);
                }
            }
            else if(start.equals(GSAFETY5)) {
                switch(deckNum) {
                    case 1:  end.setCoord(GSAFETY4);
                             break;
                    default: end.setCoord(start);
                }
```

```
                        }
                }
        }
        // segment A-B (x = 0)
        else if(x == 0) {
                if(y - deckNum >= 0) end.setCoord(0, y-deckNum);
                else {
                        n = deckNum - y;
                        end.setCoord(n, 0);
                }
        }
        // segment B-C (y = 15)
        else if(y == 15) {
                if(x - deckNum >= 0) end.setCoord(x-deckNum, 15);
                else {
                        n = deckNum - x;
                        end.setCoord(0, 15-n);
                }
        }
        // segment C-D (x = 15)
        else if(x == 15) {
                if(y + deckNum <= 15) end.setCoord(15, y+deckNum);
                else {
                        n = deckNum - (15 - y);
                        end.setCoord(15-n, 15);
                }
        }
        // segment D-A (y = 0)
        else if(y == 0) {
                if(x + deckNum <= 15) end.setCoord(x+deckNum, 0);
                else {
                        n = deckNum - (15 - x);
                        end.setCoord(15, n);
                }
        }
        return end;
}

// retourne la position d'un pion après glissade
public static Coord computePawnSliding (Pawn o, Coord coord) {
        Coord dest = new Coord(DEFAULT);
        switch(o.getColor()) {
                case 1:   if(coord.equals(BSHORT_START)) dest.setCoord(BSHORT_END);
                          else if(coord.equals(BLONG_START)) dest.setCoord(BLONG_END);
                          else if(coord.equals(YSHORT_START)) dest.setCoord(YSHORT_END);
                          else if(coord.equals(YLONG_START)) dest.setCoord(YLONG_END);
                          else if(coord.equals(GSHORT_START)) dest.setCoord(GSHORT_END);
```

```
            else if(coord.equals(GLONG_START)) dest.setCoord(GLONG_END);
            else dest.setCoord(coord);
            if(dest.equals(BSHORT_END) || dest.equals(YSHORT_END) || dest.equals(GSHORT_END)) {
                o.setCount(o.getCount() + 3);
            }
            else if(dest.equals(BLONG_END) || dest.equals(YLONG_END) || dest.equals(GLONG_END)) {
                o.setCount(o.getCount() + 4);
            }
            break;

        case 2:  if(coord.equals(YSHORT_START)) dest.setCoord(YSHORT_END);
            else if(coord.equals(YLONG_START)) dest.setCoord(YLONG_END);
            else if(coord.equals(GSHORT_START)) dest.setCoord(GSHORT_END);
            else if(coord.equals(GLONG_START)) dest.setCoord(GLONG_END);
            else if(coord.equals(RSHORT_START)) dest.setCoord(RSHORT_END);
            else if(coord.equals(RLONG_START)) dest.setCoord(RLONG_END);
            else dest.setCoord(coord);
            if(dest.equals(RSHORT_END) || dest.equals(YSHORT_END) || dest.equals(GSHORT_END)) {
                o.setCount(o.getCount() + 3);
            }
            else if(dest.equals(RLONG_END) || dest.equals(YLONG_END) || dest.equals(GLONG_END)) {
                o.setCount(o.getCount() + 4);
            }
            break;
        case 3:  if(coord.equals(GSHORT_START)) dest.setCoord(GSHORT_END);
            else if(coord.equals(GLONG_START)) dest.setCoord(GLONG_END);
            else if(coord.equals(RSHORT_START)) dest.setCoord(RSHORT_END);
            else if(coord.equals(RLONG_START)) dest.setCoord(RLONG_END);
            else if(coord.equals(BSHORT_START)) dest.setCoord(BSHORT_END);
            else if(coord.equals(BLONG_START)) dest.setCoord(BLONG_END);
            else dest.setCoord(coord);
            if(dest.equals(BSHORT_END) || dest.equals(RSHORT_END) || dest.equals(GSHORT_END)) {
                o.setCount(o.getCount() + 3);
            }
            else if(dest.equals(BLONG_END) || dest.equals(RLONG_END) || dest.equals(GLONG_END)) {
                o.setCount(o.getCount() + 4);
            }
            break;
        case 4:  if(coord.equals(RSHORT_START)) dest.setCoord(RSHORT_END);
            else if(coord.equals(RLONG_START)) dest.setCoord(RLONG_END);
            else if(coord.equals(BSHORT_START)) dest.setCoord(BSHORT_END);
            else if(coord.equals(BLONG_START)) dest.setCoord(BLONG_END);
            else if(coord.equals(YSHORT_START)) dest.setCoord(YSHORT_END);
            else if(coord.equals(YLONG_START)) dest.setCoord(YLONG_END);
            else dest.setCoord(coord);
            if(dest.equals(BSHORT_END) || dest.equals(YSHORT_END) || dest.equals(RSHORT_END)) {
                o.setCount(o.getCount() + 3);
```

```
                    }
                    else if(dest.equals(BLONG_END) || dest.equals(YLONG_END) || dest.equals(RLONG_END)) {
                        o.setCount(o.getCount() + 4);
                    }
                    break;
            default:  dest.setCoord(coord);
                    break;
        }
        return dest;
    }
    // retourne un joueur par numéro
    public static Player getPlayer (Player[] p, int color) {
        Player player;
        switch(color) {
            case 1:   player = p[0];
                    break;
            case 2:   player = p[1];
                    break;
            case 3:   player = p[2];
                    break;
            case 4:   player = p[3];
                    break;
            default:  player = new PlayerDEFAULT();
        }
        return player;
    }

    // calcul count de case départ à coord
    public static int countToStartFrom (Coord coord, int color) {
        int x = coord.getX();
        int y = coord.getY();
        int n = 0;

        switch(color) {
            case 1:   if(x == 15) {
                        if(y > 12) n = 56 + (15 - y);
                        else n = 11 - y;
                    }
                    else if (y == 0) n = 11 + (15 - x);
                    else if(x == 0) n = 26 + y;
                    else if(y == 15) n = 41 + x;
                    break;
            case 2:   if(y == 0) {
                        if(x > 12) n = 56 + (15 - x);
                        else n = 11 - x;
                    }
                    else if (x == 0) n = 11 + y;
```

```
                    else if(y == 15) n = 26 + y;
                    else if(x == 15) n = 41 + (15 - y);
                    break;
            case 3:  if(x == 0) {
                        if(y < 3) n = 56 + y;
                        else n = y - 4;
                    }
                    else if (y == 15) n = 11 + x;
                    else if(x == 15) n = 26 + (15 - y);
                    else if(y == 0) n = 41 + (15 - x);
                    break;
            case 4:  if(y == 15) {
                        if(x < 3) n = 56 + x;
                        else n = x - 4;
                    }
                    else if (x == 15) n = 11 + (15 - y);
                    else if(y == 0) n = 26 + (15 - x);
                    else if(x == 0) n = 41 + y;
                    break;
            default: break;
        }
        return n;
}

// retourne un nombre random entre 1 et 16
public static int getRandomInt () {
        return ((int)(Math.random() * (double)16)) + 1;
}

// retourne un nombre random selon max voulu (à partir de 0)
public static int randomChoice (int maxValue) {
        return ((int)(Math.random() * (double)maxValue));
}

// convertit int en boolean
public static boolean intToBoolean (int n) {
        if(n == 1) return true;
            return false;
        }

// valide si saisie clavier est un int ET valide (0 ou 1)
public static boolean isNumberValid1 (int n) {
        if(n == 0 || n == 1) return true;
        return false;
}

// valide si saisie clavier est un int ET valide (1 ou 2)
```

```
    public static boolean isNumberValid2 (int n) {
        if(n == 1 || n == 2) return true;
        return false;
    }

    // valide si saisie clavier est un int ET valide (1, 2 ou 3)
    public static boolean isNumberValid3 (int n) {
        if(n == 1 || n == 2 || n == 3) return true;
        return false;
    }
    // valide si saisie clavier est un String non-vide et contient [r,b,y,g] et [1,2,3,4]
    public static boolean isStringValid (String s) {
        if(s.length() != 2 || s.length() > 2 ) return false;
        else {
            if(s.charAt(0) == 'r' || s.charAt(0) == 'b' || s.charAt(0) == 'y' || s.charAt(0) == 'g') {
                if(s.charAt(1) == '1' || s.charAt(1) == '2' || s.charAt(1) == '3' || s.charAt(1) == '4') {
                    return true;
                }
            }
        }
        return false;
    }
}
```

“Sorry.java”

```
import game.Game;

public class Sorry {
    private static Game sorry;

    public static void main (String[] args) {
        sorry = new Game(true);
        sorry.playTurn();
    }
}
```