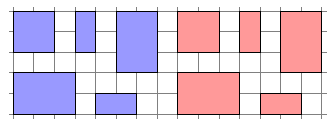
**Dossier de développement logiciel**

The Tiler Team

---

****

Jules Doumèche & Gwénolé Martin

2020 - Groupes 110 et 103

Sommaire

[I - Présentation du projet 2](#_Toc34557927)

[II – Diagramme UML 2](#_Toc34557928)

[III – Code des tests unitaires 2](#_Toc34557929)

[IV – Code du projet 2](#_Toc34557930)

[V – Bilan de projet 2](#_Toc34557931)

*Contexte du projet :*

Ce projet a été réalisé en période C (Semestre 2). Nous avons été encadrés par M. Poitreneau et Mme Caraty.

# I - Présentation du projet

Le but de ce projet était de coder une adaptation du jeu « The Tiler Team », un jeu de pavage. C’est un jeu collaboratif où une équipe de carreleurs doit coopérer pour paver au mieux un mur.

Ce jeu utilise des cartes, des pièces de pavage (des carreaux) et un « mur » pour les poser.

Le but du jeu est simple, il faut obtenir un maximum de points à la fin de la partie (quand il n’y a plus de cartes, de carreaux, ou que le joueur décide d’arrêter la partie). Pour obtenir des points, il faut poser des carreaux en suivant les indications données par les cartes et en respectant des règles de placement :

* Les cartes permettent de savoir quels carreaux peuvent être utilisées pour le tour de jeu (carreaux de couleurs rouge ou bleue, ou de taille spécifique).
* Le carreau ne doit pas dépasser la zone à carreler.
* Le carreau doit toucher un carreau déjà posé.
* Le carreau doit reposer sur d’autres carreaux (ou sur le « sol » pour les premiers tours de jeu »).
* Le carreau n’a pas de côté adjacent de même longueur avec un autre carreau.

Des pénalités sont appliquées : 1 point par carte passée, et 1 point par carreau non posé sur le mur.

Pour le programme, nous devions mettre en place des « tours de jeu », sans différencier les joueurs. Le joueur peut, à chaque tour, effectuer 3 actions différentes :

* Tenter de poser un carreau en fonction des carreaux proposés (définis par la carte tirée automatiquement).
* Taper « next » pour tirer une autre carte s’il ne peut ou ne sait pas comment poser la pièce.
* Taper « stop » pour arrêter la partie (pour éviter les pénalités dues aux cartes passées).

A la fin du jeu, nous calculons les points obtenus et affichons le score, nous avons pris le parti d’afficher 0 quand le score du joueur est négatif ou égal à 0.

# II – Diagramme UML

Une image contenant capture d’écran

Description générée automatiquement

# III – Code des tests unitaires

Tous nos tests unitaires fonctionnent, on utilise des assertTrue, Equals ou False en fonction de ce que l’on teste.

**JeuDeCartesTest.java**

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

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

**class** JeuDeCartesTest {

@Test

**void** testInit() {

JeuDeCartes jdc = **new** JeuDeCartes();

*assertEquals*(33, jdc.getCartesRestantes());

}

@Test

**void** testTirer() {

JeuDeCartes jdc = **new** JeuDeCartes();

**for**(**int** i = 33; i > 0; --i) {

*assertEquals*(i, jdc.getCartesRestantes());

jdc.tirer();

}

*assertEquals*(**null**, jdc.tirer());

}

}

**---**

**CarreauTest.java**

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

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

**class** CarreauTest {

@Test

**void** testConstructeur() {

**for**(**char** i = 'a'; i < 'i'; ++i) {

Carreau c = **new** Carreau(i);

*assertTrue*(c.getLettre() == i);

*assertTrue*(c.getHauteur() > 0 && c.getLargeur() > 0);

Carreau c2 = **new** Carreau(Character.*toUpperCase*(i));

*assertTrue*(Character.*isUpperCase*(c2.getLettre()));

}

}

@Test

**void** testNeutre() {

Carreau ver = **new** Carreau(**true**);

*assertTrue*(ver.getLettre() == 'x' && ver.getLargeur() == 1 && ver.getHauteur() == 3);

Carreau hor = **new** Carreau(**false**);

*assertTrue*(hor.getLettre() == 'x' && hor.getLargeur() == 3 && hor.getHauteur() == 1);

}

@Test

**void** testEst() {

Carreau b = **new** Carreau('a');

Carreau r = **new** Carreau('A');

Carreau t1 = **new** Carreau('a');

Carreau t2 = **new** Carreau('b');

Carreau t3 = **new** Carreau('H');

**for**(**int** i = 0; i < 5; ++i) {

**switch**(i) {

**case** 0:

*assertTrue*(b.est(Type.***BLEU***));

*assertFalse*(r.est(Type.***BLEU***));

**break**;

**case** 1:

*assertTrue*(r.est(Type.***ROUGE***));

*assertFalse*(b.est(Type.***ROUGE***));

**break**;

**case** 2:

*assertTrue*(t1.est(Type.***T1***));

*assertFalse*(t3.est(Type.***T1***));

**break**;

**case** 3:

*assertTrue*(t2.est(Type.***T2***));

*assertFalse*(t1.est(Type.***T2***));

**break**;

**case** 4:

*assertTrue*(t3.est(Type.***T3***));

*assertFalse*(t1.est(Type.***T3***));

*assertFalse*(t2.est(Type.***T3***));

**break**;

}

}

}

}

**---**

**JeuDeCarreauxTest.java**

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

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

**class** JeuDeCarreauxTest {

@Test

**void** testConstructeur() {

JeuDeCarreaux jdc = **new** JeuDeCarreaux();

*assertEquals*(18, jdc.carreauxRestants());

}

@Test

**void** testCréerListe() {

JeuDeCarreaux jdc = **new** JeuDeCarreaux();

JeuDeCarreaux b = jdc.créerListe(Type.***BLEU***);

JeuDeCarreaux r = jdc.créerListe(Type.***ROUGE***);

JeuDeCarreaux t1 = jdc.créerListe(Type.***T1***);

JeuDeCarreaux t2 = jdc.créerListe(Type.***T2***);

JeuDeCarreaux t3 = jdc.créerListe(Type.***T3***);

*assertEquals*(9, b.carreauxRestants());

*assertEquals*(9, r.carreauxRestants());

*assertEquals*(10, t1.carreauxRestants());

*assertEquals*(10, t2.carreauxRestants());

*assertEquals*(10, t3.carreauxRestants());

}

@Test

**void** testContient() {

JeuDeCarreaux jdc = **new** JeuDeCarreaux();

*assertTrue*(jdc.contient('a') && jdc.contient('i')

&& jdc.contient('A') && jdc.contient('I'));

JeuDeCarreaux b = jdc.créerListe(Type.***BLEU***);

JeuDeCarreaux r = jdc.créerListe(Type.***ROUGE***);

*assertTrue*(b.contient('a') && b.contient('i'));

*assertFalse*(b.contient('A') && b.contient('I'));

*assertTrue*(r.contient('A') && r.contient('I'));

*assertFalse*(r.contient('a') && r.contient('i'));

}

@Test

**void** testEnlever() {

JeuDeCarreaux jdc = **new** JeuDeCarreaux();

*assertTrue*(jdc.contient('a'));

jdc.enlever('a');

*assertFalse*(jdc.contient('a'));

}

}

**---**

**MurTest.java**

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

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

**class** MurTest {

**private** **final** **static** **int** ***LARGEUR*** = 5;

@Test

**void** testPoserNeutre() {

Mur m = **new** Mur();

m.poserNeutre();

**int** i = 0;

**boolean** trouvé = **false**;

**while**(i < ***LARGEUR***) {

**if**(m.getMur().get(0)[i] == 'x') {

trouvé = **true**;

**break**;

}

}

*assertTrue*(trouvé);

}

@Test

**void** testPoser() {

Mur m = **new** Mur();

m.poser('c', 1, 1);

*assertTrue*(m.getMur().get(0)[0] == 'c' && m.getMur().get(0)[1] == 'c');

m.poser('c', 2, 1);

*assertTrue*(m.getMur().get(1)[0] == 'c' && m.getMur().get(1)[1] == 'c');

}

@Test

**void** testNiveauComplets() {

Mur m = **new** Mur();

**char**[] complet = **new** **char**[] {'a', 'a', 'a', 'a', 'a' };

**for**(**int** i = 0; i < 10; ++i) {

m.getMur().add(complet);

}

System.***out***.println(m.niveauComplets());

*assertEquals*(10, m.niveauComplets());

}

@Test

**void** testVerifier() {

Mur m = **new** Mur();

*assertFalse*(m.verifier(1, 1, 'a')); // carreau isolé

m.poser('a', 1, 3);

*assertFalse*(m.verifier(1, 2, 'a')); // clone gauche

*assertFalse*(m.verifier(1, 4, 'a')); // clone droite

*assertFalse*(m.verifier(2, 3, 'a')); // clone base

*assertFalse*(m.verifier(2, 3, 'h')); // le carreau repose sur une case vide

*assertFalse*(m.verifier(1, 2, 'h')); // carreau superposé

*assertFalse*(m.verifier(1, 4, 'h')); // dépassement zone

}

}

# IV – Code du projet

**Type.java**

/\*\*

\* Type.java Définit les types utilisés pour gérer les cartes (et donc les carreaux)

\*

\* **@author** Jules Doumèche, Gwénolé Martin

\*/

**public** **enum** Type {

***ROUGE***,

***BLEU***,

***T1***,

***T2***,

***T3***;

/\*\*

\* Permet l'affichage des instructions des cartes selon leurs types

\*

\* **@return** l'instruction correspondant au type

\*/

@Override

**public** String toString() {

**switch**(**this**) {

**case** ***ROUGE***:

**return** "Rouge";

**case** ***BLEU***:

**return** "Bleu";

**case** ***T1***:

**return** "Taille 1";

**case** ***T2***:

**return** "Taille 2";

**case** ***T3***:

**return** "Taille 3";

**default**:

**return** "null";

}

}

}

**---**

**JeuDeCartes.java**

**import** java.util.Collections;

**import** java.util.Stack;

/\*\*

\* JeuDeCartes.java Définit une pile de 33 Cartes avec une instruction spécifiée par Type

\*

\* **@author** Jules Doumèche, Gwénolé Martin

\*/

**public** **class** JeuDeCartes {

**private** **static** **final** **int** ***NB\_CARTES*** = 33;

**private** Stack<Type> pile;

**private** **int** CartesRestantes;

/\*

\* Constructeur: JeuDeCartes

\*

\* @return JeuDeCartes de 33 cartes, 9 rouges, 9 bleues, 5 "taille 1", 5 "taille 2" et 5 "taille 3"

\*/

**public** JeuDeCartes() {

**this**.pile = **new** Stack<>();

**this**.CartesRestantes = ***NB\_CARTES***;

//Ajouts des cartes des différents types

**for**(**int** i = 0; i < 9; ++i) {

**this**.pile.add(Type.***BLEU***);

**this**.pile.add(Type.***ROUGE***);

}

**for**(**int** i = 0; i < 5; ++i) {

**this**.pile.add(Type.***T1***);

**this**.pile.add(Type.***T2***);

**this**.pile.add(Type.***T3***);

}

//Mélange du jeu de cartes

Collections.*shuffle*(pile);

}

/\*\*

\* Tire une carte et renvoie son type

\*

\* **@return** le type de la carte permettant de connaître l'instruction de la carte

\* **@see** Type.java

\*/

**public** Type tirer() {

**if**(CartesRestantes == 0) {

**return** **null**;

}

**this**.CartesRestantes--;

**return** pile.pop();

}

/\*

\* Vérifie si toutes les cartes ont été tirées

\*

\* @return true s'il ne reste plus aucune carte false sinon

\*/

**public** **boolean** estVide() {

**return** pile.isEmpty();

}

/\*

\* Retourne le nombre de cartes restantes dans la pile

\* (pour les tests unitaires)

\*

\* @return le nombre de cartes restantes

\*/

**public** **int** getCartesRestantes() {

**return** **this**.CartesRestantes;

}

}

**---**

**Carreau.java**

/\*\*

\* Carreau.java Définit un carreau, sa lettre et ses dimensions.

\*

\* **@author** Jules Doumèche, Gwénolé Martin

\*/

**public** **class** Carreau {

**private** **char** lettre;

**private** **int** hauteur;

**private** **int** largeur;

/\*

\* Constructeur: Carreau en fonction de sa lettre

\*

\* @param l : la lettre correspondante

\* @return Carreau

\*/

**public** Carreau(**char** l){

**this**.lettre = l;

l = Character.*toLowerCase*(l);

**switch**(l) {

**case** 'a':

**this**.largeur = 1;

**this**.hauteur = 1;

**break**;

**case** 'b':

**this**.largeur = 1;

**this**.hauteur = 2;

**break**;

**case** 'c':

**this**.largeur = 2;

**this**.hauteur = 1;

**break**;

**case** 'd':

**this**.largeur = 2;

**this**.hauteur = 2;

**break**;

**case** 'e':

**this**.largeur = 1;

**this**.hauteur = 3;

**break**;

**case** 'f':

**this**.largeur = 3;

**this**.hauteur = 1;

**break**;

**case** 'g':

**this**.largeur = 2;

**this**.hauteur = 3;

**break**;

**case** 'h':

**this**.largeur = 3;

**this**.hauteur = 2;

**break**;

**case** 'i':

**this**.largeur = 3;

**this**.hauteur = 3;

**break**;

}

}

/\*

\* Constructeur : Carreau neutre en fonction de son orientation

\*

\* @param estVert : orientation verticale ou horizontale

\* @return Carreau neutre

\*/

**public** Carreau(**boolean** estVert) {

**this**.lettre = 'x';

**if**(estVert) {

**this**.hauteur = 3;

**this**.largeur = 1;

}

**else** {

**this**.hauteur = 1;

**this**.largeur = 3;

}

}

/\*

\* Permet de savoir si le carreau est du type indiqué

\*

\* @param t : le type à tester

\* @return true si le carreau est du type indiqué, false sinon

\* @see Type.java

\*/

**public** **boolean** est(Type t) {

**switch**(t) {

**case** ***BLEU***:

**return** Character.*isLowerCase*(**this**.lettre);

**case** ***ROUGE***:

**return** Character.*isUpperCase*(**this**.lettre);

**case** ***T1***:

**return** (**this**.largeur == 1 || **this**.hauteur == 1);

**case** ***T2***:

**return** (**this**.largeur == 2 || **this**.hauteur == 2);

**case** ***T3***:

**return** (**this**.largeur == 3 || **this**.hauteur == 3);

}

**return** **false**;

}

/\*

\* Renvoie la lettre du carreau

\*

\* @return lettre du carreau

\*/

**public** **char** getLettre() {

**return** **this**.lettre;

}

/\*

\* Renvoie la hauteur du carreau

\*

\* @return hauteur(y) du carreau

\*/

**public** **int** getHauteur() {

**return** **this**.hauteur;

}

/\*

\* Renvoie la largeur du carreau

\*

\* @return largeur(x) du carreau

\*/

**public** **int** getLargeur() {

**return** **this**.largeur;

}

}

**---**

**JeuDeCarreaux.java**

**import** java.util.ArrayList;

/\*\*

\* JeuDeCarreaux.java Définit une liste de Carreau

\*

\* **@author** Jules Doumèche, Gwénolé Martin

\*/

**public** **class** JeuDeCarreaux {

**private** **static** **final** **int** ***TMAX*** = 3;

**private** ArrayList<Carreau> jdc;

/\*

\* Constructeur: JeuDeCarreaux

\*

\* @return le jeu de carreaux initialisé avec les 18 carreaux bleus et rouges

\*/

**public** JeuDeCarreaux() {

jdc = **new** ArrayList<>();

**for**(**char** i = 'a'; i < ( 'a' + ***TMAX***\****TMAX*** ); ++i) {

jdc.add(**new** Carreau(i));

jdc.add(**new** Carreau(Character.*toUpperCase*(i)));

}

}

/\*

\* Crée et initialise un jeu de carreaux contenant tous les carreaux restants du type indiqué par la carte

\*

\* @param type : le type du jeu de carreaux à créer

\* @return le jeu de carreaux créé

\*/

**public** JeuDeCarreaux créerListe(Type type) {

JeuDeCarreaux retour = **new** JeuDeCarreaux();

retour.jdc = **new** ArrayList<>();

**for**(**int** i = 0; i < **this**.jdc.size(); ++i) {

**if**(**this**.jdc.get(i).est(type)) {

retour.jdc.add(**this**.jdc.get(i));

}

}

**return** retour;

}

/\*

\* Vérifie si tous les carreaux ont été posés

\*

\* @return true s'il ne reste plus aucun carreau, false sinon

\*/

**public** **boolean** estVide() {

**return** **this**.jdc.isEmpty();

}

/\*

\* Vérifie si le jeu de carreau contient le carreau correspondant à la lettre indiquée

\*

\* @param lettre : la lettre qui correspond au carreau à vérifier

\* @return true si le jeu de carreaux contient le carreau de la lettre indiqué, false sinon

\*/

**public** **boolean** contient(**char** lettre) {

**for**(**int** i = 0; i < jdc.size(); ++i) {

**if**(lettre == jdc.get(i).getLettre()) {

**return** **true**;

}

}

**return** **false**;

}

/\*

\* Enlève du jeu de carreaux le carreau correspondant à la lettre indiquée

\*

\* @param lettre : la lettre qui correspond au carreau à enlever

\* @return true si la suppression a été effectuée, false sinon

\* @see contient(char lettre) pour vérifier la présence du carreau avant de l'enlever

\*/

**public** **boolean** enlever(**char** lettre) {

**if**(**this**.contient(lettre)) {

**for**(**int** i = 0; i < jdc.size(); ++i) {

**if**(lettre == jdc.get(i).getLettre()) {

jdc.remove(i);

**return** **true**;

}

}

}

**return** **false**;

}

/\*

\* Retourne le nombre de carreaux non posés

\*

\* @return nombre de carreaux non posés

\*/

**public** **int** carreauxRestants() {

**return** jdc.size();

}

/\*

\* Obtient l'affichage des carreaux dans l'ordre alphabétique et des carreaux bleus à rouges

\*

\* @return la chaîne avec les carreaux côte à côte sur une ligne

\*/

**public** String toString() {

StringBuilder sb = **new** StringBuilder();

//Récupère y max

**int** yMax = 0;

**for**(**int** i = 0; i < jdc.size() - 1; ++i) {

**if**(jdc.get(i).getHauteur() > yMax) {

yMax = jdc.get(i).getHauteur();

}

}

sb.append("\n");

//Affiche les carreaux dans l'ordre

**for**(**int** i = yMax; i > 0; --i) {

**for**(**int** j = 0; j < jdc.size(); ++j) {

**for**(**int** h = 0; h < jdc.get(j).getLargeur(); ++h) {

**if**(jdc.get(j).getHauteur() < i) {

sb.append(" ");

}

**else** {

sb.append(" " + jdc.get(j).getLettre());

}

}

sb.append(" ");

}

sb.append("\n");

}

**return** sb.toString();

}

}

**---**

**Mur.java**

**import** java.security.SecureRandom;

**import** java.util.ArrayList;

**import** java.util.Arrays;

/\*\*

\* Mur.java Définit le mur sur lequel les carreaux sont posés

\*

\* **@author** Jules Doumèche, Gwénolé Martin

\*/

**public** **class** Mur {

**private** **static** **final** **int** ***LARGEUR*** = 5;

**private** **static** **final** **char** ***LIBRE*** = ' ';

**private** ArrayList<**char**[]> grille;

**private** **int** nx;

**private** **int** ny;

/\*

\* Constructeur: Mur et initialisation sans carreau neutre (pour les tests)

\*

\* @return le Mur initialisé

\* @see Mur(boolean pNeutre) pour pouvoir poser le carreau neutre à l'initialisation du Mur

\*/

**public** Mur() {

**this**.grille = **new** ArrayList<>();

}

/\*

\* Constructeur: Mur et initialisation sans carreau neutre(pour les tests)

\*

\* @param pNeutre : si vrai le carreau neutre est posé

\* @return le Mur initialisé

\* @see Mur() pour pouvoir initialiser le Mur sans paramètre d'entrée si aucun carreau neutre

\*/

**public** Mur(**boolean** pNeutre) {

**this**();

**if**(pNeutre) {

**this**.poserNeutre();

}

}

/\*

\* Pose le carreau neutre à l'une des 4 positions possibles aléatoirement

\*

\* @see Mur(boolean pNeutre) pour pouvoir poser le carreau neutre à l'initialisation du Mur

\*/

**public** **void** poserNeutre() {

SecureRandom r = **new** SecureRandom();

**boolean** x = r.nextBoolean();

**boolean** y = r.nextBoolean();

Carreau neutre = **new** Carreau(x);

**this**.nx = neutre.getLargeur();

**this**.ny = neutre.getHauteur();

**if**(y) {

poser(neutre, grille.size() + 1, ***LARGEUR*** + 1 - neutre.getLargeur());

}

**else** {

poser(neutre, grille.size() + 1, 1);

}

}

/\*\*

\* Pose le Carreau dans la grille aux coordonnées entrées (qui ont été vérifiées au préalable)

\*

\* **@param** c : le carreau à poser

\* **@param** x : coordonnée x où placer le carreau (le plus à gauche)

\* **@param** y : coordonnée y où placer le carreau (le plus en bas)

\* **@see** poser(char lettre, int y, int x) pour pouvoir poser le Carreau en ne spécifiant que sa lettre et les coordonnées

\* **@see** verifier(char lettre, int y, int x) pour vérifier la validité du placement du carreau aux coordonnées indiquées

\*/

**public** **void** poser(Carreau c, **int** y, **int** x) {

**while**(grille.size() - y < c.getHauteur()) {

**char**[] ligne = **new** **char**[***LARGEUR***];

Arrays.*fill*(ligne, ***LIBRE***);

grille.add(ligne);

}

**for**(**int** i = x; i < x + c.getLargeur(); ++i) {

**for**(**int** j = y; j < y + c.getHauteur(); ++j) {

grille.get(j - 1)[i - 1] = c.getLettre();

}

}

}

/\*\*

\* Pose le Carreau dans la grille aux coordonnées entrées (qui ont été vérifiées au préalable)

\*

\* **@param** lettre : la lettre correspondant au carreau à poser

\* **@param** x : coordonnée x où placer le carreau(le plus à gauche)

\* **@param** y : coordonnée y où placer le carreau(le plus en bas)

\* **@see** poser(Carreau c, int y, int x) pour pouvoir poser le Carreau en spécifiant le Carreau et les coordonnées

\* **@see** verifier(char lettre, int y, int x) pour vérifier la validité du placement du carreau aux coordonnées indiquées

\*/

**public** **void** poser(**char** lettre, **int** y, **int** x) {

Carreau c = **new** Carreau(lettre);

poser(c, y, x);

}

/\*

\* Vérifie si le carreau correspondant à la lettre indiquée peut-être posé aux coordonnées spécifiées

\*

\* @param lettre : la lettre correspondant au carreau à vérifier

\* @param x : coordonnée x du carreau(le plus à gauche)

\* @param y : coordonnée y du carreau(le plus en bas)

\* @return "Valide" si valide, sinon description de l'erreur

\* @see poser pour pouvoir poser le Carreau après vérification

\* @see fonctions de règles : rDépassement, rVide, rSuperposé, rCloné et rIsolé

\*/

**public** String verifier(**char** lettre, **int** y, **int** x) {

Carreau c = **new** Carreau(lettre);

**if**(rDépassement(c, x)) {

**return** "Le carreau dépasse la zone";

}

**if**(rVide(c, y, x)) {

**return** "Le carreau repose sur une case vide";

}

**if**(rSuperposé(c, y, x)) {

**return** "Le carreau est superposé à un autre";

}

**switch**(rCloné(c, y, x)) {

**case** 0:

**break**;

**case** 1:

**return** "La base du carreau clone la face supérieure du carreau inférieur";

**case** 2:

**return** "La face droite du carreau clone la face gauche du carreau à droite";

**case** 3:

**return** "La face gauche du carreau clone la face droite du carreau à gauche";

**default**:

**break**;

}

**if**(rIsolé(c, y ,x)) {

**return** "Le carreau ne touche aucun autre carreau";

}

**return** "valide";

}

/\*

\* Vérifie si le carreau correspondant à la lettre indiquée peut être posé aux coordonnées spécifiées

\* (Pour les tests unitaires)

\*

\* @param x : coordonnée x du carreau (le plus à gauche)

\* @param y : coordonnée y du carreau (le plus en bas)

\* @param lettre : la lettre correspondant au carreau à vérifier

\* @return true si valide, false sinon

\* @see poser pour pouvoir poser le Carreau après vérification

\* @see fonctions de règles : rDépassement, rVide, rSuperposé, rCloné et rIsolé

\*/

**public** **boolean** verifier(**int** y, **int** x, **char** lettre) {

Carreau c = **new** Carreau(lettre);

**return** !(rDépassement(c, x) || rVide(c, y, x)

|| rSuperposé(c, y, x) || rCloné(c, y, x) > 0 || rIsolé(c, y ,x));

}

/\*

\* Compte le nombre de lignes complètes

\*

\* @return le nombre de lignes complètes

\*/

**public** **int** niveauComplets() {

**int** i = grille.size() - 1;

**while**(i >= 0) {

**for**(**int** j = 0; j < ***LARGEUR***; ++j) {

**if**(grille.get(i)[j] == ***LIBRE***) {

--i;

**break**;

}

**else** **if**(j == ***LARGEUR*** - 1) {

**return** i + 1;

}

}

}

**return** 0;

}

/\*

\* Retourne le mur ( uniquement pour les tests unitaires et débug )

\*

\* @return la liste de liste correspondant au mur

\*/

**public** ArrayList<**char**[]> getMur() {

**return** **this**.grille;

}

/\*\*

\* Permet d'afficher le mur de bas en haut avec les numérotations

\*

\* **@return** la chaîne de caractère correspondant à l'affichage du mur

\*/

**public** String toString() {

StringBuilder sb = **new** StringBuilder();

sb.append("\n");

**for**(**int** i = grille.size(); i > 0; --i) {

**if**(i < 10)

sb.append(i + " ");

**else**

sb.append(i);

**for**(**int** j = 0; j < ***LARGEUR***; ++j) {

sb.append(" " + grille.get(i - 1)[j]);

}

sb.append("\n");

}

sb.append(" ");

**for**(**int** j = 1; j <= ***LARGEUR***; ++j) {

sb.append(" " + j);

}

sb.append("\n");

**return** sb.toString();

}

/\*

\* Vérifie si le carreau dépasse les bordures à l'abscisse x

\*

\* @param c : le carreau

\* @param x : l'abscisse

\* @return true si le carreau dépasse les bordures du mur, false sinon

\*/

**private** **boolean** rDépassement(Carreau c, **int** x) {

**return** (x + c.getLargeur() - 1 > ***LARGEUR*** || x < 1);

}

/\*

\* Vérifie si le carreau ne repose pas entièrement sur des carreaux déjà posé aux positions x, y

\*

\* @param c : le Carreau

\* @param y : l'ordonnée

\* @param x : l'abscisse

\* @return true si le carreau repose sur au moins une case vide, false sinon

\*/

**private** **boolean** rVide(Carreau c, **int** y, **int** x) {

**if**( y > 1) {

**for**(**int** i = x; i < x + c.getLargeur(); ++i) {

**if**( **this**.grille.get(y - 2)[i - 1] == ***LIBRE***) {

**return** **true**;

}

}

}

**return** **false**;

}

/\*

\* Vérifie si le carreau est superposé à un autre carreau aux positions x, y

\*

\* @param c : le carreau

\* @param y : l'ordonnée

\* @param x : l'abscisse

\* @return true si le carreau se superpose à au moins une case pleine, false sinon

\*/

**private** **boolean** rSuperposé(Carreau c, **int** y, **int** x) {

ajouterLignes(c, y);

**for**(**int** j = y; j < y + c.getHauteur(); ++j) {

**for**(**int** i = x; i < x + c.getLargeur(); ++i) {

**if**( **this**.grille.get(j - 1)[i - 1] != ***LIBRE***) {

retirerLignes(c, y);

**return** **true**;

}

}

}

**return** **false**;

}

/\*

\* Vérifie si le carreau ne clone aucune des faces adjacentes

\*

\* @param c : le carreau

\* @param y : l'ordonnée

\* @param x : l'abscisse

\* @return 1 si la base du carreau est clonée, 2 si la face droite du carreau est clonée,

\* 3 si la face gauche du carreau est clonée, 0 si aucune face du carreau n'est clonée

\*/

**private** **int** rCloné(Carreau c, **int** y, **int** x) {

**if**( y > 1) {

**char** cTest = **this**.grille.get(y - 2)[x - 1];

**if**(cTest == **this**.grille.get(y - 2)[x + c.getLargeur() - 2]) {

**if**(cTest == 'x' && nx == c.getLargeur()) {

retirerLignes(c, y);

**return** 1;

}

**else** {

Carreau carInf = **new** Carreau(**this**.grille.get(y - 2)[x + c.getLargeur() - 2]);

**if**(carInf.getLargeur() == c.getLargeur()) {

retirerLignes(c, y);

**return** 1;

}

}

}

}

**if**( x - 1 + c.getLargeur() != ***LARGEUR*** ) {

**char** cTest = **this**.grille.get(y - 1)[x - 1 + c.getLargeur()];

**if**(cTest == **this**.grille.get(y + c.getHauteur() - 2)[x - 1 + c.getLargeur()]) {

**if**(cTest == 'x' && ny == c.getHauteur()) {

retirerLignes(c, y);

**return** 2;

}

**else** {

Carreau carInf = **new** Carreau(**this**.grille.get(y + c.getHauteur() - 2)[x - 1 + c.getLargeur()]);

**if**( carInf.getHauteur() == c.getHauteur()) {

retirerLignes(c, y);

**return** 2;

}

}

}

}

**if** ( x != 1 ) {

**char** cTest = **this**.grille.get(y - 1)[x - 2];

**if**(cTest == **this**.grille.get(y + c.getHauteur() - 2)[x - 2]) {

**if**(cTest == 'x' && ny == c.getHauteur()) {

retirerLignes(c, y);

**return** 3;

}

**else** {

Carreau carInf = **new** Carreau(**this**.grille.get(y + c.getHauteur() - 2)[x - 2]);

**if**( carInf.getHauteur() == c.getHauteur()) {

retirerLignes(c, y);

**return** 3;

}

}

}

}

**return** 0;

}

/\*

\* Permet d'ajouter les lignes supplémentaires nécessaires aux tests

\* si le carreau dépasse la ligne la plus haute

\*

\* @param c : le carreau à vérifier

\* @param y : la ligne y où ajouter les lignes nécessaires

\* @see retirerLignes pour retirer les lignes si le carreau n'est pas valide

\*/

**private** **void** ajouterLignes(Carreau c, **int** y) {

**while**(grille.size() - y < c.getHauteur()) {

**char**[] ligne = **new** **char**[***LARGEUR***];

Arrays.*fill*(ligne, ***LIBRE***);

grille.add(ligne);

}

}

/\*

\* Permet de supprimer les lignes supplémentaires nécessaires aux tests après ajouterLignes()

\*

\* @param c : le carreau à vérifier

\* @param y : la ligne y pour supprimer les lignes ajoutées précedemment

\* @see ajouterLignes pour ajouter les lignes

\*/

**private** **void** retirerLignes(Carreau c, **int** y) {

**while**(grille.size() - y < c.getHauteur()) {

grille.remove(grille.size());

}

}

/\*

\* Vérifie si le carreau ne touche aucun carreau adjacent aux positions x et y

\*

\* @param c : le carreau

\* @param y : l'ordonnée

\* @param x : l'abscisse

\* @return true si le carreau est isolé et ne touche aucun carreau, false sinon

\*/

**private** **boolean** rIsolé(Carreau c, **int** y, **int** x) {

**return** (y == 1

&&

(x == 1 || **this**.grille.get(y - 1)[x - 2] == ***LIBRE***)

&&

( x == ***LARGEUR*** || **this**.grille.get(y - 1)[x + c.getLargeur() - 1] == ***LIBRE***));

}

}

**---**

**Jeux.java**

**import java.security.NoSuchAlgorithmException;**

**import java.util.Scanner;**

**/\*\***

**\* Jeux.java Permet de jouer au jeu "The Tiler Team"**

**\***

**\* @author Jules Doumèche, Gwénolé Martin**

**\*/**

**public** **class** Jeux {

/\*

\* Vérifie si le jeu est fini

\*

\* @return true si le jeu est fini, false sinon

\*/

**private** **static** **boolean** estFini(**boolean** status, JeuDeCartes pile, JeuDeCarreaux jdc) {

**return** jdc.estVide() || pile.estVide() || !status;

}

/\*\*

\* Fonction principale (main) permettant l'exécution d'une partie du jeu

\*

\* **@throws** NoSuchAlgorithmException

\*/

**public** **static** **void** main(String[] args) **throws** NoSuchAlgorithmException {

JeuDeCartes pile = **new** JeuDeCartes();

JeuDeCarreaux jdc = **new** JeuDeCarreaux();

Scanner sc = **new** Scanner(System.***in***);

Mur mur = **new** Mur(**true**);

**int** cartesEc = 0;

**boolean** status = **true**;

**while**(!*estFini*(status, pile, jdc)) {

//Tirer la carte

Type carte = pile.tirer();

//Liste des carreaux pouvant être posés

JeuDeCarreaux listeCarreaux = jdc.créerListe(carte);

**if**(!listeCarreaux.estVide()) {

//Affichage des instructions d'entrées et initialisation

**boolean** saisieValide = **false**;

**while**(!saisieValide) {

//Affichage du mur et de l'instruction de la carte tirée

System.***out***.println(mur);

System.***out***.println(carte);

//Affichage de la liste de Carreaux

System.***out***.println(listeCarreaux);

System.***out***.println("Veuillez entrer la lettre correspondant au carreau choisi suivit du numéro de la ligne(y) et de la colonne(x).\n"

+ "Par exemple 'h 2 1' pour poser le carreau h à la 2ème ligne et à la première colonne.\n"

+ "Vous pouvez de plus écrire 'next' pour écarter la carte et passer au prochain tour ou 'stop' pour mettre fin à la partie.\n");

String input = sc.next();

**if**(input.equalsIgnoreCase("next")) {

++cartesEc;

**break**;

}

**else** **if**(input.equalsIgnoreCase("stop")) {

status = **false**;

**break**;

}

**else** {

// Initialisation des variables d'entrées

// Pour inverser le sens de saisie (qui est de base y puis x, vous pouvez inverser x et y ici (instructions l.76 et l.78))

**char** lettre = input.charAt(0);

**int** x = 0;

**int** y = 0;

**if**(sc.hasNextInt()) {

y = sc.nextInt();

**if**(sc.hasNextInt()) {

x = sc.nextInt();

**if**(listeCarreaux.contient(lettre)) {

String codeInput = mur.verifier(lettre, y, x);

**if**(codeInput.equals("valide")) {

mur.poser(lettre, y, x);

jdc.enlever(lettre);

saisieValide = **true**;

}

**else** {

System.***out***.println("Erreur! Impossible de poser le carreau " + lettre

+ " au positions " + y + " " + x +"\n Erreur: " + codeInput + ".\n");

}

}

**else** {

System.***out***.println("Erreur! Veuillez entrez un carreau affichée dans la liste précedente.\n");

}

}

}

}

}

}

**else** {

++cartesEc;

System.***out***.println("Aucun carreau restant ne correspond à la carte tirée");

//passage au tour suivant automatiquement

}

}

//Fin de la partie et calcul du score

**int** points = mur.niveauComplets() \* 5 - jdc.carreauxRestants() - cartesEc;

**if** (points < 0) {

points = 0;

}

System.***out***.println(points + " points (" + mur.niveauComplets() + " niveaux complets, "

+ jdc.carreauxRestants() + " carreaux non posés, " + cartesEc +" cartes écartées)");

sc.close();

}

}

# V – Bilan de projet

1 – Les difficultés rencontrées

La principale difficulté a été de créer une structure globale fonctionnelle (entre les classes), tout en structurant les classes pour qu’elles puissent fonctionner indépendamment les unes des autres.

Les conditions de validation pour vérifier si un carreau peut être posé ont aussi été difficiles à trouver (et il a fallu résoudre de nombreux bugs), mais finalement, nous avons réussi à les implémenter. La plus difficile de ces conditions était celle de non-adjacence.

2 – Ce qui est réussi

Globalement, malgré les difficultés rencontrées, nous avons une structure cohérente et fonctionnelle avec :

* une énumération Type (pour les types de cartes)
* une classe JeuDeCartes (pour le jeu de carte, cette classe utilise Type)
* une classe Mur (pour représenter le mur sur lequel on pose les carreaux)
* une classe Carreau (pour définir les carreaux)
* une classe JeuDeCarreaux (pour les listes de carreau, comme la liste de « départ » avec les 18 carreaux, cette classe utilise Carreaux
* une classe Jeu (qui permet de jouer, et qui affiche tous les messages nécessaires)

Le programme fonctionne donc comme demandé mais certaines choses pourraient être améliorée.

3 – Ce qui peut être amélioré

On pourrait rajouter une classe Appli, et enlever le main de la classe Jeu, de façon à ce que la classe Jeu ne serve qu’à faire le déroulement du jeu (c’est ce qu’elle fait déjà, mais on s’en sert aussi comme main, on pourrait donc changer le main de place de sorte à séparer vraiment ces deux aspects). On pourrait aussi peut-être réduire la classe Carreau, pour l’alléger (essayer de trouver un autre moyen de retourner les carreaux par exemple).

Sinon, il y a de nombreux ajouts qui peuvent être faits, comme permettre d’enregistrer les meilleurs scores, entrer différents joueurs, modifier la taille du mur (en largeur), etc …