**Architecture logiciel**

Rapport du projet Turtle

L’objectif du projet est d’apprendre écrire un code sans bug (sans faille) grâce au test unitaire et aussi disposer d’un code maintenable, modulable et facile à comprendre d’apres notre compréhension du concept.

Partie I :

Problème I : mise en place de l’environnement de développement

Installation de jdk1-8 et git 2.20 et Eclipse Version : Oxygen.3a Release (4.7.3a) effectué avec succès.

**Problème II : Implémentation de méthode mayUseCodeInAssignment dans la classe RulesOf6005**

a) Suivant les conditions indiquées dans les commentaires nous parvenons à atteindre notre objectif avec les lignes suivantes :

**public** **static** **boolean** mayUseCodeInAssignment(**boolean** writtenByYourself,

**boolean** availableToOthers, **boolean** writtenAsCourseWork,

**boolean** citingYourSource, **boolean** implementationRequired) {

// **TODO**: Fill in this method, then remove the exception

**if** (writtenByYourself) {

**return** **true**;

}**else** **if** (implementationRequired) {

**return** **true**;

}**else** **if** (writtenAsCourseWork && availableToOthers && citingYourSource ) {

**return** **true**;

}

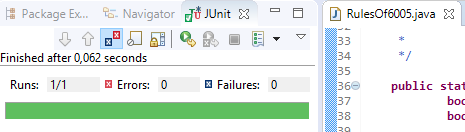
**else** {

**return** **false**;

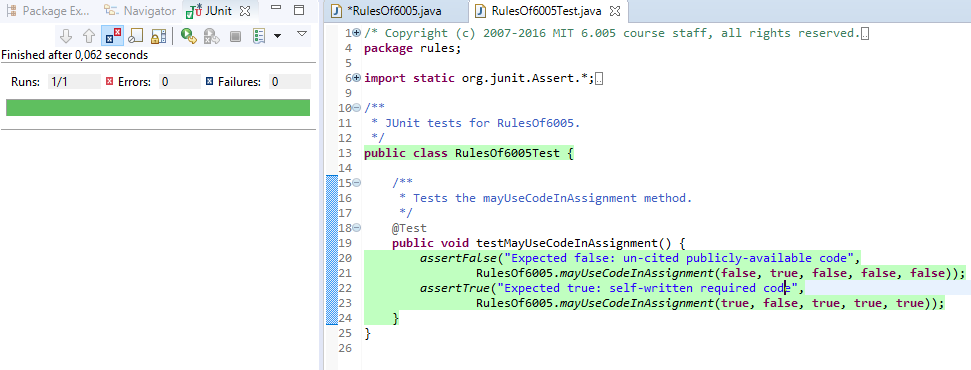
}

}

b) Nous obtenons le résultat suivant :

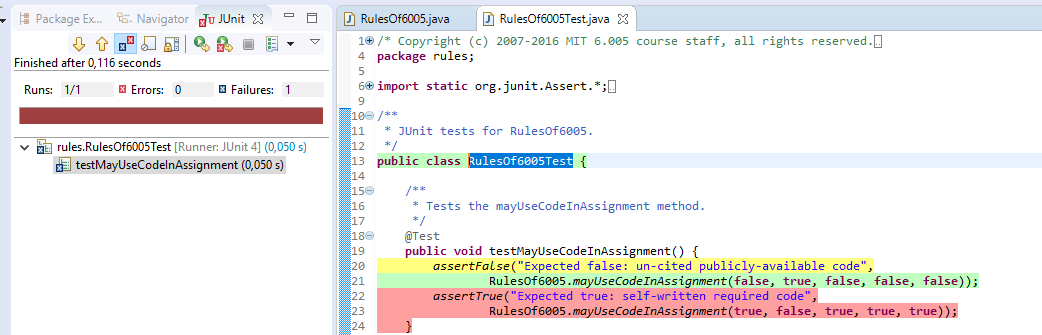


c) En effectuant un test sur la classe RulesOf6005Test , nous obtenons effectivement un test réussi :



Preuve que l’implémentation est bonne (ou du moins nous donne une certaine garantie).

d) En passant une valeur « true » à toutes les combinaisons, nous obtenons un bar rouge signifiant que le test à échouer.



e) Nous constatons effectivement que les tests « assertFalse/True » permet de tester effectivement de la valeur attendue, si assertFalse est utilisé alors la valeur attendue est False dans le cas contraire le test échoue.

**Problème III : Commit effectuer en local apres avoir suivi les instructions**

**Deuxième partie**

**Problème IV : Implémentation la méthode drawsquare**

L’implémentation a parfaitement marché avec ce bout de code :

**public** **static** **void** drawSquare(Turtle turtle, **int** sideLength) {

**for** (**int** i = 1; i <= 4; i++) {

turtle.forward(sideLength);

turtle.turn(90.0);

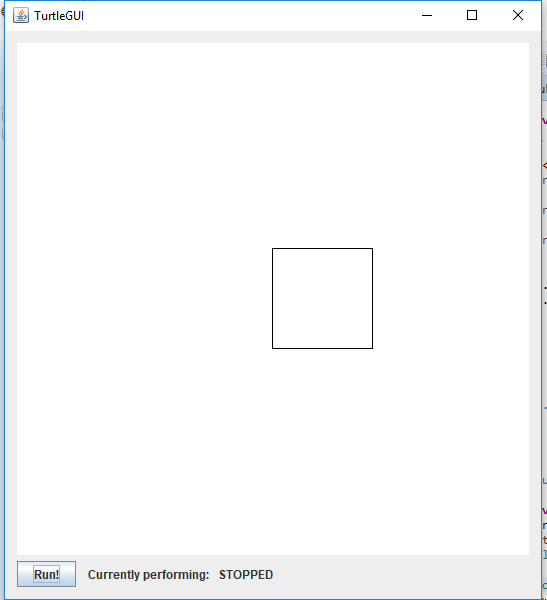
turtle.forward(sideLength);

}

}

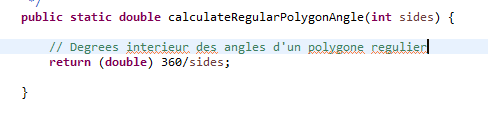
L’incrémentation de « i » allant jusqu’à 4 parce que un carré a quatre coté et « turn(90.0) » parce que un carré c’est un angle rectangle donc 90 degré.

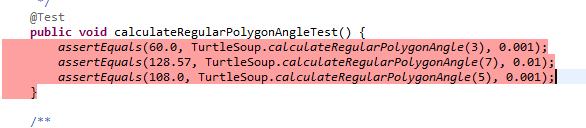
Voila une image correspond à l’effet de ce code :



**Problème V à X :**

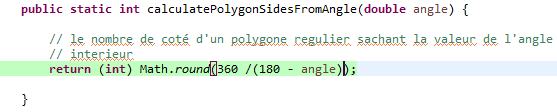
L’implémentation de la formule des angles d’un polygone régulier à fonctionner (nous effectuons un tracé de polygone régulier normalement). Par contre Le test JUnit ne marche pas :



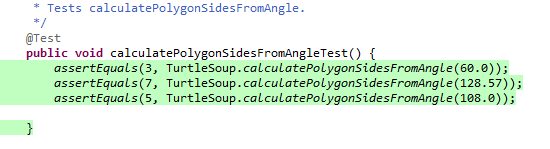


Une autre formule marche pour ce test mais ne nous permet de tracé un polygone régulier normalement.

L’implémentation de la méthode calculatePolygonSidesFromAngle donne également satisfaction (utilisation de la librairie Math pour arrondir ainsi donc obtenir un entier) à partir de la méthode test :



Code



Test

**DrawPolygonRegular :**

En nous inspirant de la methode drawsquare : nous rendons ainsi flexible la creation d’un polygone :

**public** **static** **void** drawRegularPolygon(Turtle turtle, **int** sides, **int** sideLength) {

**for** (**int** i = 1; i <= sides; i++) {

turtle.forward(sideLength);

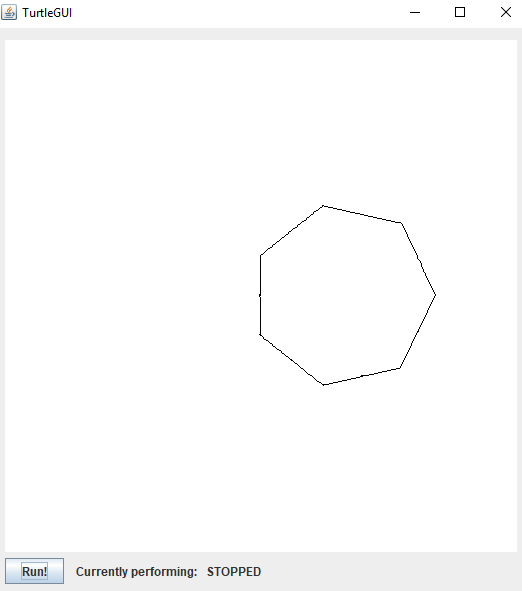
turtle.turn(TurtleSoup.*calculateRegularPolygonAngle*(sides));

turtle.forward(sideLength);

}

}

Ainsi dans la méthode « main » nous pouvons l’exécuter avec par exemple un polygone régulier de 7 cotés :



**Implémentation de la methode calculHeadingToPoint :**

Avec quelque recherche effectué sur la méthode math.ant2 java nous obtenons avec succès un test JUnit vert :

**public** **static** **double** calculateHeadingToPoint(**double** currentHeading, **int** currentX, **int** currentY, **int** targetX,

**int** targetY) {

**double** radian;

**double** degre;

radian = Math.*atan2*(targetX - currentX, targetY - currentY);

degre = (radian \* 180 / Math.***PI***) - currentHeading;

**if** (degre >= 0 && degre < 360) {

**return** degre;

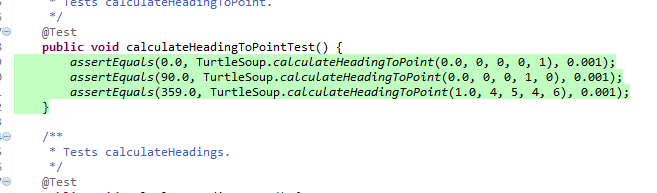
} **else** {

**return** degre = degre +360;

}

}

Avec un test JUnit :



**Implémentation de la Méthode calculHeading**

A partir de l’exemple fournie dans le projet et de quelque recherche, nous obtenons ce code :

**public** **static** List<Double> calculateHeadings(List<Integer> xCoords, List<Integer> yCoords) {

List<Double> degreList = **new** ArrayList<Double>();

**int** size = xCoords.size();

**if** (xCoords.size() == yCoords.size() && size >= 2) {

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

**if** (i == 0) {

degreList.add(*calculateHeadingToPoint*(0, xCoords.get(i), yCoords.get(i), xCoords.get(i + 1),

yCoords.get(i + 1)));

} **else** {

degreList.add(*calculateHeadingToPoint*(degreList.get(i - 1), xCoords.get(i), yCoords.get(i),

xCoords.get(i + 1), yCoords.get(i + 1)));

}

}

**return** degreList;

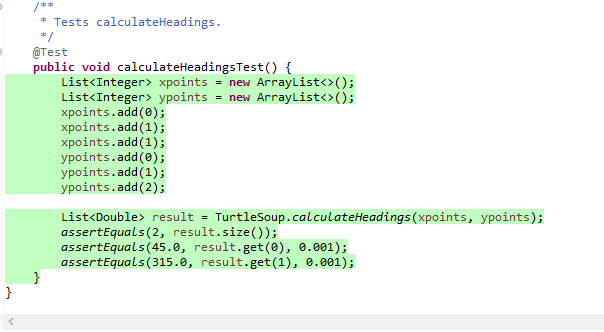
} **else** {

**throw** **new** RuntimeException("Liste Invalide");

}

}

Qui nous donne satisfaction avec un succès du test :



**Problème XI : Art personnel**

Nous avons effectuer quelque pour déterminer l’angle d’une étoile et l’ayant ainsi obtenue nous avions dessiner une multitude d’étoiles ( 6 exactement )

**public** **static** **void** drawPersonalArt(Turtle turtle) {

**for** (**int** i = 1; i <= 30; i++) {

**if** (i <= 10) {

turtle.color(PenColor.***RED***);

} **else** **if** (10 <= i && 20 <= i) {

turtle.color(PenColor.***BLUE***);

} **else** {

turtle.color(PenColor.***GREEN***);

}

turtle.forward(i \* 10);

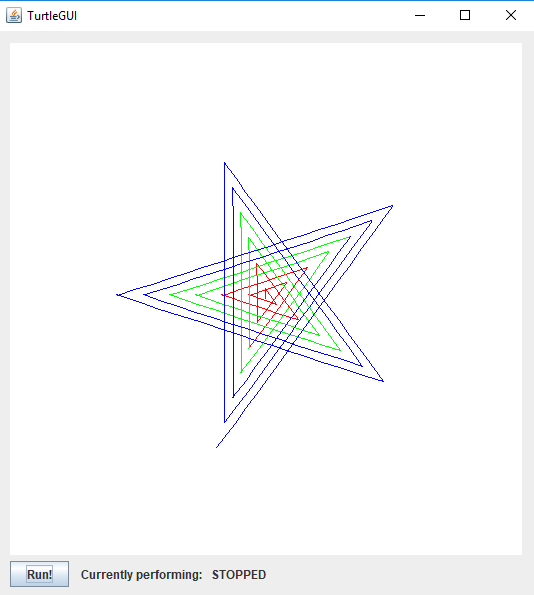
turtle.turn(144);

}

}

Nous agrandissons notre étoile à chaque boucle (turtle.forwar(i\*30)) et changeons de couleur à tous les 10 tours.

Ce qui donne cet effet graphiquement.



Nous avons au cours de ce projet appris tester notre code avec JUnit (assertEquals et assertFalse/True) ainsi nous pouvions ainsi créer notre structure de test avec c’est deux méthodes pour élargir l’angle test de nos objets ou méthodes.