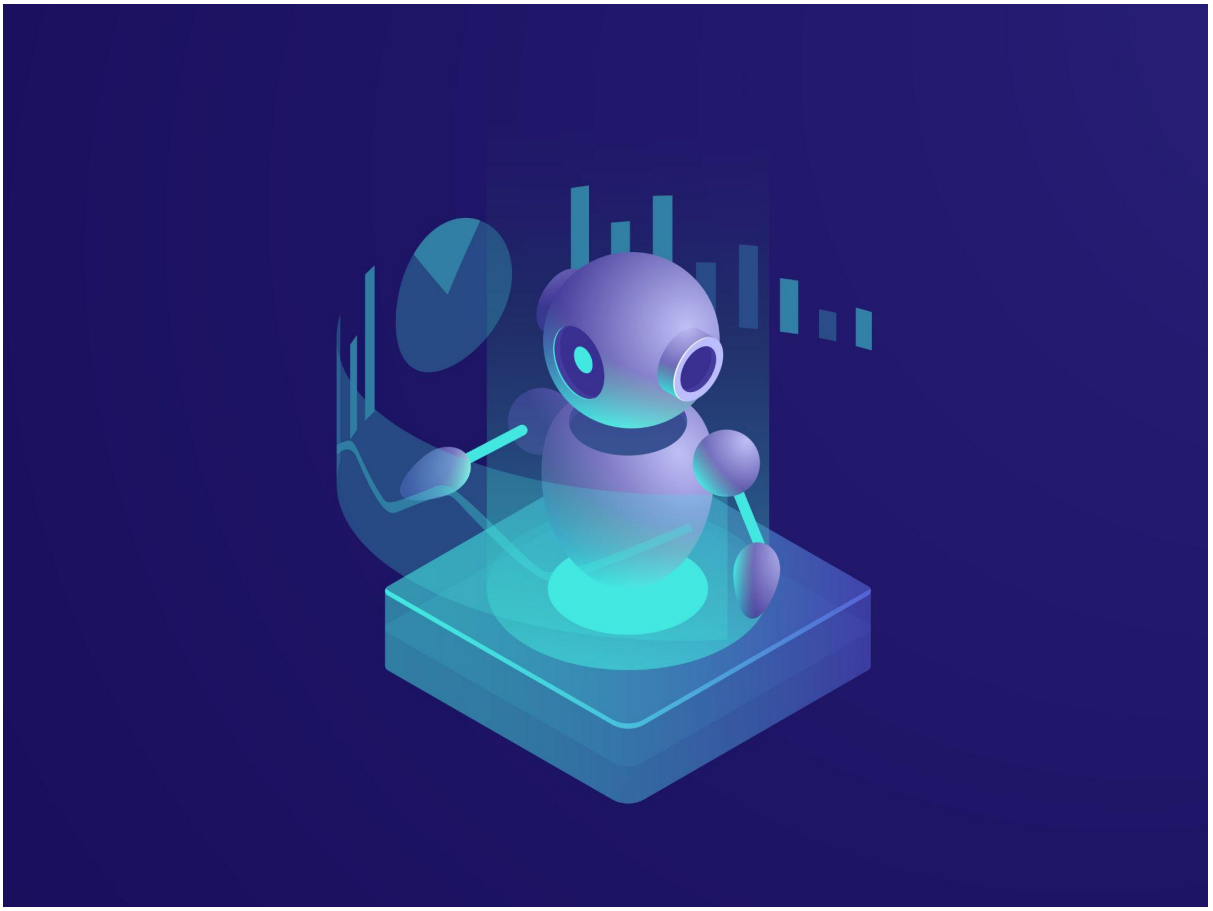
Université Grenoble Alpes

Année 2022-2023

PLAN DE TESTS



Chef de projet

Damien Pellier

Membres du groupe

Robin Chaussemy 12014629

Laurine De La Chapelle 12014762

Houda Lkima 11906442

Aurore Philippe 11900982

Mélisse Tilquin 12011231

SOMMAIRE

1. Présentation des tests

Dans un premier temps, nous procédons aux tests unitaires de calibration via la classe TestCalibration (7. Annexe, p. 13-16) pour vérifier que les classes Actionneur et Capteur contiennent un pourcentage d'erreurs acceptable. Ensuite, nous ferons des tests unitaires sur les méthodes de la classe Cedric_Strategie pour vérifier qu'elles répondent bien à nos attentes avec un pourcentage d'erreurs acceptable.

Enfin, nous procéderons aux tests d'intégration sur les différentes étapes des programmes qui seront implémentés au robot le jour de la compétition. Pour ce faire, nous utiliserons une stratégie ascendante en testant en premier lieu le programme pour l'homologation pour en relever les erreurs potentielles et vérifier que le robot fasse ce qui est attendu avec un pourcentage d'erreurs acceptable, puis nous ferons de même avec le programme utilisé lors des matchs de la compétition pour être assuré que tout fonctionne correctement selon nos attentes avec un pourcentage d'erreurs acceptable.

Ces différents tests vérifient, entre autres, que les fonctions du robot correspondent à celles décrites dans le cahier des charges. C'est-à-dire que nous vérifions que le robot est capable de se déplacer, de se repérer dans son environnement, de percevoir son environnement et d'interagir avec lui, de prendre un palet et de le déposer dans le camp adverse.

2. Guide de lecture

Toute personne ayant des connaissances en robotique, informatique, langage java, leJos et intelligence artificielle peut supposément lire ce document sans se préoccuper du glossaire et des index.

Toute personne ayant aucune connaissance en robotique, informatique, langage java, leJos et intelligence artificielle, dans au moins l'un d'eux, et toute personnes ayant des difficultés dans au moins l'un d'eux doit supposément s'aider des références pour comprendre ce document.

3. Tests unitaires

a. Tests de calibration

Pour actionneurs

Test méthode **rotateSC()** de la classe Actionneur avec la méthode TestRotation() de la classe TestCalibration :

Données d’entrées : un nombre entier contenant la vitesse (v) à laquelle le robot effectue les rotations.

Résultats attendus : le robot doit effectuer dix rotations d’angles aléatoires de manière synchrone selon la vitesse entrée par le programmeur.

Critère d’évaluation : si le robot effectue moins de dix rotations, s’il effectue plus de dix rotations ou s’il n’effectue aucune rotation, la méthode rotateSC() est erronée.

Pour capteurs

Test pour la méthode **getDistance()** de la classe Capteur avec la méthode TestFourDistance() de la classe TestCalibration :

Données d’entrées : une instance de la classe Capteur (c) et un nombre réel contenant le pourcentage d'erreurs autorisé divisé par 100 (error).

Résultats attendus : Le robot est placé manuellement et de manière successive à quatre distances prédéfinies d’un obstacle : à 15, 30, 50 et 80 cm d’un obstacle. Après chaque placement et après que l’un des membres de l’équipe projet est appuyé sur le capteur de toucher du robot, le robot retourne la distance réelle qu’il détecte entre lui et l’obstacle.

Critère d’évaluation : si au moins l’une des distances retournée n’est pas égale à la distance prédéfinie correspondante à plus ou moins le pourcentage d'erreurs autorisé divisé par 100 entré par le programmeur, alors la méthode getDistance() est sûrement erronée (il peut aussi s’agir d’une erreur de placement). Toutefois, si les quatre distances prédéfinies sont différentes des distances retournées correspondantes à plus ou moins le pourcentage d'erreurs autorisé divisé par 100 entré par le programmeur, alors la méthode getDistance() est erronée.

Test pour la méthode **getDistance()** de la classe Capteur avec la méthode TestDistance() de la classe TestCalibration :

Données d’entrées : une instance de la classe Capteur (c), un nombre réel contenant la distance entre le robot et un obstacle (distance) et un nombre réel contenant le pourcentage d'erreurs autorisé divisé par 100 (error).

Résultats attendus : Après que l’un des membres de l’équipe projet est appuyé sur le capteur de toucher du robot, le robot retourne la distance réelle qu’il détecte entre lui et l’obstacle.

Critère d'évaluation : si la distance retournée n'est pas égale à la distance prédéfinie, par le programmeur, correspondante à plus ou moins le pourcentage d'erreurs autorisé divisé par 100, entré par le programmeur, alors la méthode getDistance() est sûrement erronée (il peut aussi s'agir d'une erreur dans la mesure de la distance entre le robot et l'obstacle effectuée par l'un des membres de l'équipe projet).

Test pour la méthode **getColor()** de la classe Capteur avec la méthode TestColor() de la classe TestCalibration :

Données d'entrées : une instance de la classe Capteur (c) et un nombre réel contenant le pourcentage d'erreurs autorisé divisé par 100 (error).

Résultats attendus : Pour chacune des six lignes de couleurs différentes qui délimitent le terrain, la valeur de la couleur détectée par le capteur de couleur du robot, qui est prélablement placé sur l'une des six lignes du terrain (d'abord sur la ligne blanche, puis sur les lignes bleue, jaune, rouge, verte et enfin noire), est retournée.

Critère d'évaluation : si au moins l'une des valeurs retournées n'est pas égale à la valeur de la couleur de la ligne correspondante, sur laquelle le robot capte la couleur, alors la méthode getColor() est sûrement erronée. Si les six valeurs retournées ne sont pas égales aux valeurs des couleurs des lignes correspondantes, alors la méthode getColor() est erronée.

Test pour la méthode **getTouche()** de la classe Capteur avec la méthode TestTouch() de la classe TestCalibration :

Données d'entrées : une instance de la classe Capteur (c).

Résultats attendus : La méthode retourne true si le capteur de touché est activé.

Critère d'évaluation : Si la méthode retourne la valeur booléenne false, alors la méthode getTouche() est erronée.

b. Tests de la stratégie

```
public static void changementEtat(String e) {

    Cedric_Strategie c = new Cedric_Strategie();
    System.out.println(c.getEtat());
    c.setEtat(e);
    System.out.println(c.getEtat());
```

}

Pour Cedric_Strategie

Test des méthodes **setEtat(String e)** et **getEtat()** de la classe Cedric_Strategie avec la méthode changementEtat(String e) :

Données d'entrées : L'état (e) prend successivement les valeurs : "DEPART", "RECUPERE", "VIDE" et "PLEIN".

Résultats attendus : Après chaque appel de la méthode changementEtat(String e), la console doit afficher l'état dans lequel se trouvait le robot avant l'exécution de la méthode puis, sur la ligne d'en dessous, l'état (e) entré par le compilateur.

Critère d'évaluation : Si l'exécution de la méthode provoque une erreur ou si la console n'affiche pas les résultats attendus après l'exécution de la méthode avec au moins 'une des quatre valeurs de l'état (e), alors au moins l'une des méthode testée est erronée.

Test de la méthode **goToPalet(int angle)** de la classe Cedric_Strategie :

Données d'entrées : un entier représentant la valeur de l'angle de rotation

Résultats attendus : Le robot doit se tourner de l'angle fourni en entrée et se diriger vers droit jusqu'à l'enclenchement du détecteur de toucher ce qui entraîne la fermeture des pinces puis une dernière rotation en direction du camp adverse.

Critère d'évaluation : La méthode est une réussite si l'enchaînement des actions s'effectue dans l'ordre avec une rotation de l'angle fourni en paramètre. Le robot doit partir sans palet et se retrouver avec un palet dans les pinces fermées.

Test de la méthode **recherche()** de la classe Cedric_Strategie :

Données d'entrées : un tableau de String avec des coordonnées, et un tableau de double avec les coordonnées du robot

Résultats attendus : la méthode retourne un entier int donnant l'angle vers le palet ayant la position la moins coûteuse où se rendre.

Critère d'évaluation : La méthode doit retourner l'angle que doit effectuer le robot pour se trouver face au palet le plus optimal selon la méthode angleOptimal de la classe Outils_Math avec comme paramètre : un tableau de String avec des coordonnées, et un tableau de double avec les coordonnées du robot, la valeur de la boussole, et le paramètre boolean fenetre de la classe Cedric_Strategie

Test de la méthode **goToCamp()** de la classe Cedric_Strategie :

Données d'entrées : aucune

Résultats attendus : Le robot doit se rendre dans le camp adverse et y déposer le palet qu'il possédait déjà entre ses pinces avant le lancement de la méthode. Le robot se tourne vers le camp adverse, avance jusqu'à capter la ligne blanche par le capteur couleur, s'arrête, ouvre les pinces, recule et referme les pinces.

Critère d'évaluation : La méthode est une réussite si l'enchaînement des actions s'effectue dans l'ordre. Le robot doit se retrouver dans le camp adverse dos au terrain avec les pinces vides et fermées.

Test de la méthode **champLibre()** de la classe Cedric_Strategie :

Données d'entrées : aucune

Résultats attendus : La méthode doit arrêter et tourner le robot vers la droite de 90 degrés dès qu'il se trouve à moins de 18 cm d'un obstacle (mur ou robot adverse) puis retourner le boolean false. Si aucun obstacle ne se présente elle retourne true sans rien faire.

Critère d'évaluation : Lorsque la méthode est lancée sans aucun obstacle devant le robot en mouvement, elle doit retourner true. Lorsque la méthode est lancée avec un obstacle devant le robot à moins de 18 cm, elle doit arrêter le robot en mouvement, le faire tourner et retourner false.

Test de la méthode **premierPalet()** de la classe Cedric_Strategie :

Données d'entrées : aucune

Résultats attendus : Le robot doit marquer le palet qui se trouve en face de lui sans toucher aux autres palets sur le terrain au coup d'envoi du match.

Critère d'évaluation : La méthode doit permettre au robot placer au préalable sur la ligne blanche face à un palet placé au croisement de la première ligne horizontale et d'une des ligne verticale, de le récupérer et le marquer sans toucher aux autre palet placés sur les croisement des lignes de couleur.

Pour Outils_Math

Test de la méthode **plusProche()** de la classe Outils_Math() :

Données d'entrées : un tableau de String contenant les coordonnées des objet sur la table autre que le robot en lui même, un tableau de double avec les coordonnées du robot

Résultats attendus : La méthode doit retourner un int donnant l'indice de l'objet le plus proche du robot dans le tableau de coordonnées.

Critère d'évaluation : La méthode est une réussite si l'indice retourné est celui du palet le plus proche en prenant en compte l'angle de rotation à fournir ainsi que la distance à parcourir pour l'atteindre.

Test de la méthode **angleOptimal()** de la classe Outils_Math() :

Données d'entrées : un tableau de String contenant les coordonnées des objet sur la table autre que le robot en lui même, un tableau de double avec les coordonnées du robot, un int contenant la valeur de la boussole, un boolean représentant si le camp est du coté fenêtre ou non

Résultats attendus : La méthode doit retourner un int donnat l'angle en degrés le moins coûteux vers un objet du tableau.

Critère d'évaluation : La méthode est une réussite si l'angle retourné est celui vers le palet le plus proche en prenant en compte en tenant compte de la distance du robot à l'objet et la distance de l'objet au camp.

Test de la méthode **coordsSansCed()** de la classe Outils_Math() :

Données d'entrées : un tableau de String contenant les coordonnées capté par la caméra, et un int contenant l'indice du robot dans le tableau

Résultats attendus : La méthode doit retourner un tableau de String capté par la caméra sans le robot.

Critère d'évaluation : La méthode est une réussite si le tableau retourné contient les coordonnées de tous les objets sur le terrain uniquement et à l'exception du robot.

Test de la méthode **coordsPrecise()** de la classe Outils_Math() :

Données d'entrées : un tableau de String contenant les coordonnées capté par la caméra, et un int contenant l'indice du robot dans le tableau

Résultats attendus : La méthode doit retourner un tableau de double avec les coordonnées du robot.

Critère d'évaluation : La méthode est une réussite si le tableau retourné contient les coordonnées du robot uniquement.

3. Tests d’intégration

a. Premier test

```
public static void main(String[] args){

        try{
                Cedric_Strategie c1 = new Cedric_Strategie();
        } catch(Exception e) {
                System.out.println("Il est impossible de créer une instance de la classe
                Cedric_Strategie().");
        }

        try{
                Cedric_Strategie c2 = new Cedric_Strategie();
                c2.homologation1();
        } catch(Exception e){
                System.out.println("Il y a des erreurs dans la méthode goToCamp() de la
                classe Cedric_Strategie.");
        }

}
```

Description du test : Si aucune erreur n'est réceptionnée à l'exécution du code, le robot doit attendre que l'un des membres de l'équipe projet spécifie de quel côté est son camp et s'il se trouve sur la ligne blanche de son camp. Puis il doit avancer et s'arrêter à la ligne blanche du camp adverse.

But du test : Ce premier test sert à vérifier que le constructeur Cedric_Strategie() ainsi que la méthode homologation1() de la classe Cedric_Strategie() ne provoquent pas d'erreur à l'exécution et qu’après leur exécution, le robot fasse ce qui est attendu de lui avec un pourcentage d'erreurs acceptable.

Principe de réalisation : Les blocs d'instructions try permettent de tester le code étape par étape. Les instructions catch permettent de réceptionner les erreurs du code et permettent au compilateur de savoir à quelle étape se situe l'erreur relevée.
Dans le premier bloc try, une instance de la classe Cedric_Strategie() est créée. Dans le second bloc try, une instance de la classe Cedric_Strategie() est créée et la méthode homologation1() de la classe Cedric_Strategie() est appelée sur cette instance.

b. Second test

```
public static void main(String[] args){

        try{
```

```
            Cedric_Strategie c1 = new Cedric_Strategie();
            c1.premierPalet();
        } catch(Exception e) {
            System.out.println("Il y a des erreurs dans la méthode goToCamp() ou
            dans la méthode premierPalet() de la classe Cedric_Strategie().");
        }

        try{
            Cedric_Strategie c = new Cedric_Strategie();
            c.strategie();
        } catch(Exception e) {
            System.out.println("Il y a des erreurs dans la stratégie ou dans les
            méthodes appelées dans la méthode strategie() de la classe
            Cedric_Strategie().");
        }

}
```

Description du test : Si aucune erreur n'est réceptionnée à l'exécution du code du premier bloc try catch, le robot doit arriver à ramasser la palet face à lui et à le déposer dans le camp adverse. Si aucune erreur n'est réceptionnée à l'exécution du code du second bloc try catch, le robot doit finir par ramasser tous les palets présents sur le terrain et par les ramener dans le camp adverse si aucun autre robot ne s'interpose.

But du test : Ce test vise à vérifier que l'exécution des méthodes premierPalet() et strategie() de la classe Cedric_Strategie() ne provoquent pas d'erreur à l'exécution. Il vérifie aussi qu'après l'exécution des méthodes premierPalet() et strategie() de la classe Cedric_Strategie(), le robot fasse ce qui est attendu de lui avec un pourcentage d'erreurs acceptable.

Principe de réalisation : Dans le premier bloc d'instructions try, une instance de la classe Cedric_Strategie() est créée et la méthode premierPalet() de la classe Cedric_Strategie() est appelée sur cette instance. Dans le second bloc d'instructions try, une instance de la classe Cedric_Strategie() est créée et la méthode stratégie() de la classe Cedric_Strategie() est appelée sur cette instance.

4. Glossaire

Test unitaire : test utilisé pour vérifier que les unités d'un logiciel ne produisent pas d'erreur.

Test d'intégration : test utilisé pour vérifier que le programme créé s'intègre dans son environnement d'exécution.

Test fonctionnel : test utilisé pour vérifier que le produit fonctionne conformément aux attentes.

5. Index

6. Références

Source utilisée pour la création des tests unitaires :

Les tests unitaires en Java. (2016, 29 février). Zeste de Savoir. https://zestedesavoir.com/tutoriels/274/les-tests-unitaires-en-java/

Source utilisée pour la création des tests d'intégration :

Arthaud, G. (2021, 9 février). *Découvrez les tests d'intégration et les tests fonctionnels*. OpenClassrooms. https://openclassrooms.com/fr/courses/6100311-testez-votre-code-java-pour-realiser-des-applications-de-qualite/6616481-decouvrez-les-tests-dintegration-et-les-tests-fonctionnels

Source utilisée pour la conception du code :

Sourceforge. (s. d.). *leJOS EV3 API documentation*. LeJOS Java for LEGO Mindstorms. https://lejos.sourceforge.io/ev3/docs/

Source de l’image de la page de garde :

Analyse De Programme, Robot Ia, Processus Automatisé De Génération De Rapports De Données à Intelligence Artificielle. (s. d.). freepik. https://fr.freepik.com/vecteurs-libre/analyse-programme-robot-ia-processus-automatise-generation-rapports-donnees-intelligence-artificielle_3629608.htm#page=3&query=Robot%20intelligence%20artificielle&position=1&from_view=search&track=sph

7. Annexe

```
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.Random;
import cedric.Actionneur;
import cedric.Capteur;

import lejos.hardware.Button;

public class TestCalibration {
	public static float[] White = new float[] {0.3029f ,0.3039f,0.2588f};
	private static float[] Blue = new float[] {0.0235f,0.0441f,0.0696f};
	private static float[] Yellow = new float[] {0.2392f,0.2167f,0.0510f};
	private static float[] Red = new float[] {0.1441f,0.0441f,0.0216f};
	private static float[] Green = new float[] {0.0568f,0.1275f,0.04118f};
	private static float[] Black = new float[] {0.0353f,0.0480f,0.0206f};
	public final static float COLORERROR=0.07f;

	public static boolean errorrange(float reference, float mesure, float e) {
		if (e<0 || e>1) {
			throw new IllegalArgumentException("Donner une valeur entre 0 et
1");
		}
		return (reference-(reference*e) <= mesure && reference+(reference*e)
>= mesure);
	}

	public static int genererInt(int borneInf, int borneSup){
		  Random random = new Random();
		  int nb;
		  nb = borneInf+random.nextInt(borneSup-borneInf);
		  return nb;
		}

	public static void TestRotation(Actionneur a, int v) {
		for (int j=0; j<10;j++) {
			a.rotateSC(genererInt(-360,360), v, false);
		}
		int Compass = a.getCompass();
		System.out.println(Compass);
		if (-180 < Compass & Compass < 180) {
			a.rotateSC(-Compass, v, false);
		}
		else {
			if (Compass <= -180) {
				a.rotateSC(-(360+Compass), v, false);
```

```
                }
                else {
                        a.rotateSC(360-Compass, v, false);
                }
        }
    }

    public static boolean TestFourDistance(Capteur c, float error) {
        System.out.println("15 cm");
        Button.ENTER.waitForPressAndRelease();
        if (!(errorrange(0.15f,c.getDistance(),error))) {
            return false;
        }
        System.out.println("30 cm");
        Button.ENTER.waitForPressAndRelease();
        if (!((errorrange(0.30f,c.getDistance(),error)))) {
            return false;
        }
        System.out.println("50 cm");
        Button.ENTER.waitForPressAndRelease();
        if (!((errorrange(0.50f,c.getDistance(),error)))) {
            return false;
        }
        System.out.println("80 cm");
        Button.ENTER.waitForPressAndRelease();
        if (!((errorrange(0.8f,c.getDistance(),error)))) {
            return false;
        }
        return true;

    }

    public static boolean TestDistance(Capteur c,float distance ,float error) {
        System.out.println(distance*10+" cm");
        Button.ENTER.waitForPressAndRelease();
        if (!(errorrange(distance,c.getDistance(),error))) {
            return false;
        }
        return true;

    }

    public static boolean TestColor(Capteur c, float error) {
        float[] Color = new float[3];
        System.out.println("Test Blanc");
        Button.ENTER.waitForPressAndRelease();
        Color = c.getColor();
        if (!errorrange(White[0],Color[0],COLORERROR) ||
!errorrange(White[1],Color[1],COLORERROR) ||
!errorrange(White[2],Color[2],COLORERROR)) {
```

```
                        return false;
                }
                // WARNING pas de modification faite pour les autres couleurs que blanc
depuis longtemps
                TestCalibration.White[0]=Color[0];
                TestCalibration.White[1]=Color[1];
                TestCalibration.White[2]=Color[2];
                System.out.println("Blanc mis à jour");

                System.out.println("Test Bleu");
                Button.ENTER.waitForPressAndRelease();
                Color = c.getColor();
                if (!errorrange(Blue[0],Color[0],error) ||
!errorrange(Blue[1],Color[1],error) || !errorrange(Blue[2],Color[2],error)) {
                        return false;
                }
                System.out.println("Test Jaune");
                Button.ENTER.waitForPressAndRelease();
                Color = c.getColor();
                if (!errorrange(Yellow[0],Color[0],error) ||
!errorrange(Yellow[1],Color[1],error) || !errorrange(Yellow[2],Color[2],error)) {
                        return false;
                }
                System.out.println("Test Rouge");
                Button.ENTER.waitForPressAndRelease();
                Color = c.getColor();
                if (!errorrange(Red[0],Color[0],error) || !errorrange(Red[1],Color[1],error)
|| !errorrange(Red[2],Color[2],error)) {
                        return false;
                }
                System.out.println("Test Vert");
                Button.ENTER.waitForPressAndRelease();
                Color = c.getColor();
                if (!errorrange(Green[0],Color[0],error) ||
!errorrange(Green[1],Color[1],error) || !errorrange(Green[2],Color[2],error)) {
                        return false;
                }
                System.out.println("Test Noir");
                Button.ENTER.waitForPressAndRelease();
                Color = c.getColor();
                if (!errorrange(Black[0],Color[0],error) ||
!errorrange(Black[1],Color[1],error) || !errorrange(Black[2],Color[2],error)) {
                        return false;
                }
                return true;
        }

        public float[] calibrationCouleur(Capteur c,float[]couleur) {
                TestColor(c);
                couleur[0]=White[0];
```

```
                couleur[1]=White[1];
                couleur[2]=White[2];
                System.out.println("Couleur mise à jour");
                return couleur;
        }

        public static boolean TestTouch(Capteur c) {
                System.out.println("Press");
                Button.ENTER.waitForPressAndRelease();
                return c.getTouche();
        }

        public static void main(String[] args) throws SocketException,
UnknownHostException {
        }

}
```