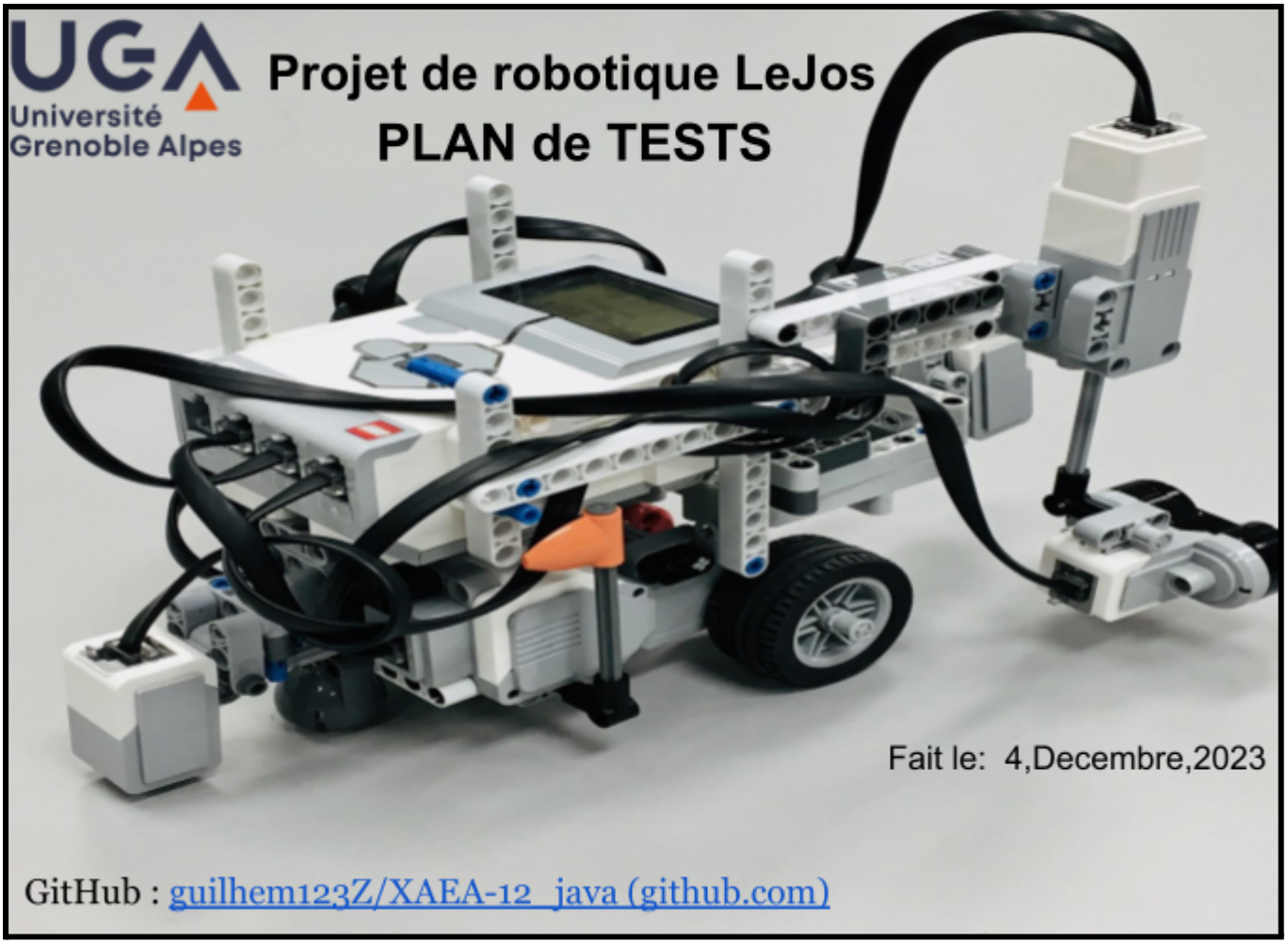

Auteurs : **FARDOUS** Nacer Eddine
LABIB Abderrahmane
LONG-MERLE Estelle
SIMEAO Guilhem
SLIKA Karam

Encadrant & Maître d'ouvrage : **PELLIER** Damien

Contexte

1. Introduction

Ce plan de tests vise à guider le développement de notre projet lejos ev3 axé sur la programmation d'un robot capable de ramasser des palets sur un plateau en un minimum de temps. Il s'agit d'une étape cruciale pour garantir que le logiciel et le matériel intégrés à notre robot répondent aux exigences spécifiées pour une sélection efficace des palets.Ce document détaille les différentes phases du plan de test dans le projet, assurant une validation rigoureuse du logiciel pour répondre à tout ce qui est recommandé pour la réussite du projet.

1.1 Objectifs et méthodes

Pendant une période de 12 semaines, notre robot doit être capable de rechercher, détecter, prendre et mettre dans l'en but adverse le plus grand nombre de palets possible, car il sera en compétition avec un deuxième robot d'un autre groupe.
Dans ce but, nous avons créé les classes Java suivantes:Capteur, Moteur, Fonctionnalités, Homologation, Tournoi.

1.2 Documents de reference

La documentation de référence est le site lejos ("Overview (leJOS EV3 API documentation) (sourceforge.io)"), qui contient les paquetages de classes, et les méthodes que nous avons utilisées pour développer le projet sont les documents utilisés dans l'espoir que ce projet aboutisse. La deuxième source est le cahier de charge et le plan de développement, qui organise, divise et gère les différentes étapes et tâches de , il sert également de feuille de route pour avancer étape par étape dans le processus de développement.

2. Guide de lecture

2.1. Maîtrise d'œuvre

FARDOUS Nacer Eddine
LABIB Abderrahmane
LONG-MERLE Estelle
SIMEAO Guilhem
SLIKA Karam

2.2. Maîtrise d'ouvrage

PELLIER Damien

3. Concepts de base

L'objectif du projet est de concevoir et de programmer un robot capable d'exécuter efficacement la tâche consistant à rassembler le plus grand nombre possible de palets dans l'en-but adverse en 3 minutes. La clé réside dans la création d'un programme intégré qui permet au robot de naviguer efficacement, de localiser et de rassembler des palets tout en respectant les contraintes imposées par la forme prédéterminée du robot, la contrainte de délai ainsi que la contrainte de communication entre les membres du groupe. Le projet réalisé par l'équipe sera évalué lors d'une compétition rigoureuse après une période de préparation de 12

semaines. Notre objectif ultime est de remporter la compétition en démontrant la fiabilité, la précision et l'efficacité de notre robot. L'évaluation consistera en des une compétition finale ainsi que la soumission de documents décrivant notre méthodologie, notre choix de conception et nos résultats.

4. Tests Fonctionnelles

4.1. Pour chaque scénario (movement, rotation, recherchePalet etc…..) :

4.1.1. Identification

4.1.2. Description

Le robot doit avancer tout droit pour récupérer un palet le plus rapidement possible puis l'amener dans l'en-but adverse. Ensuite il doit successivement ramener le maximum de palets restants sur le terrain pour marquer plus de points.

4.1.4. Dépendances

4.1.5. Procédure de test Tests unitaires

5. Test d'intégration

5.1. Pour chaque test d'intégration :

5.1.1. Identification

➔ **perpendiculaire_apres_essai()**

paramètre → pas de paramètre, pas de retour(methode void)

➔ **detecterPalet(float[] tab)**

paramètre → tableau de float tab, retourne une liste de float

➔ **avancerVersPalet(int angle_de_balayage, int delay)**

paramètre → entier angle de balayage et en entier le temps de delay , pas de retour(methode void)

➔ **versPalet(float seuilDetection)**

paramètre → float seuilDétection, pas de retour(methode void)

➔ **attraperPalet()**

paramètre → pas de paramètre, pas de retour(methode void)

5.1.2. Description

➔ **perpendiculaire_apres_essai()**

Retour : Pas de retour, méthode void.
Plage de paramètres : Aucun paramètre requis.
Comportement attendu : Exécuter des actions après l'essai, orientant peut-être le robot pour le rendre perpendiculaire à une référence spécifique.

➔ **detecterPalet(float[] tab)**

Retour : Retourne une liste de float.
Plage de paramètres : Un tableau de float (tab).
Comportement attendu : Analyse les données du tableau tab pour détecter un palet et retourne une liste de données associées au palet détecté.

➔ **avancerVersPalet(int angle_de_balayage, int delay)**

Retour : Pas de retour, méthode void.
Plage de paramètres : Un entier pour l'angle de balayage et un entier pour le temps de délai.
Comportement attendu : Fait avancer le robot vers le palet en utilisant un balayage spécifique et un délai défini.

➔ **versPalet(float seuilDetection)**

Retour : Pas de retour, méthode void.
Plage de paramètres : Un float pour le seuil de détection.
Comportement attendu : Oriente le robot ou effectue des actions spécifiques une fois qu'un palet a été détecté, en utilisant le seuil de détection spécifié.

➔ **attraperPalet()**

Retour : Pas de retour, méthode void.
Plage de paramètres : Aucun paramètres requis.
Comportement attendu : Se diriger vers le palet en ouvrant les pinces puis les fermer pour attraper le palet.

5.1.4. Dépendances

➢ **Composants physiques du robot**

Garantir le bon fonctionnement de tous les moteurs, capteurs, actionneurs (pinces) et autres composants physiques du robot est une condition préalable à toutes les méthodes mises en œuvre.

➢ **Capteurs de Détection**

vérifiez spécialement le fonctionnement des capteurs du robot pour vous assurer que certaines fonctions comme echantillon(), getTouche(), ecartImp(), estDansLaPortee() et auPalet() fonctionnent correctement car ces fonctions en dépendent a cette détecteurs utilisés pour localiser des objets
comme des palettes et détecter les obstacles et les distances.

➢ **Power Supply et résilience**

La stabilité et l'adéquation de l'alimentation électrique ont un impact direct sur les performances du robot. Assurez-vous que le robot est complètement chargé et qu'il est donc capable de rester dans les meilleures performances tout au long de la phase de test et de la compétition finale.

5.1.5. Procédure de test Annexes

La procédure de test est une étape essentielle pour garantir que les sous-méthodes répondent aux exigences spécifiées. Les méthodes principales sont donc conçues pour garantir une vérification rigoureuse de chaque fonctionnalité. La procédure est divisée en plusieurs étapes :

➢ **Initialisation du robot**

Vérification de l'état initial du robot, cette étape est essentiellement liée à la section de dépendance écrite juste avant telle que les composants physiques du robot ,capteurs de Détection et le power Supply et résilience.

➢ **Exécution de tests unitaires**

chaque méthode est testée individuellement en utilisant différentes entrées, couvrant différents scénarios :

- Vérification des valeurs limites (positives, négatives, zéro).
- Testez avec des valeurs aléatoires et extrêmes.

- Intégration de plusieurs méthodes pour évaluer le comportement dans des situations réelles. (méthode Sync).

➢ **Analyser et évaluer les retours et le comportement des méthodes**

s'assurer que le comportement de la méthode répond aux attentes énoncées dans la description.

6. Documentation des résultats

Les résultats de chaque test unitaire sont documentés, en notant les entrées utilisées, les retours obtenus et les comportements observés. Toute anomalie ou erreur est documentée pour des contrôles ultérieurs.

7. Test Unitaires

Dans cette section, nous discutons et fournissons une ventilation complète de toutes les sous-méthodes du robot que nous avons développées tout au long de la phase de développement, allant des ajustements de vitesse pour les mouvements linéaires et rotatifs aux lectures de capteurs, au contrôle des mouvements et à l'exécution des tâches, chaque méthode étant décrite avec ses paramètres, le type de retour et une description de ses comportements.

6.1. Pour chaque test unitaires:

6.1.1. Identification

➔ **changerVitLin(double s):**

paramètre → double s, pas de retour(methode void)

➔ **changerVitRot(double s):**

paramètre → double s, pas de retour(methode void)

➔ **tournerSync(double angle)**

paramètre → double angle, pas de retour(methode void)

➔ **avancer(double distance)**

paramètre → double distance, pas de retour(methode void)

➔ **avancerSync(double distance)**

paramètre → double distance, pas de retour(methode void)

➔ **tourner(double angle)**

paramètre → double angle, pas de retour(methode void)

➔ **stop()**

paramètre → pas de paramètre, pas de retour(methode void)

➔ **echantillon()**

paramètre → pas de paramètre, retourne une table de distances en cm

➔ **isMoving()**

paramètre → pas de paramètre, retourne une boolean

➔ **getTouche()**

paramètre → pas de paramètre, retourne toucher

➔ **ouvrirPinces(int angle)**

paramètre → entier angle, pas de retour(methode void)

➔ **fermerPinces(int angle)**

paramètre → entier angle, pas de retour(methode void)

➔ **msDelay(int ms)**

paramètre → entier nombre de ms d'attente

➔ **ecartImp(float[] tab, int i)**

paramètre → tableau de float et en entier i, retourne un boolean

➔ **estDansLaPortee(float[] tab, int i)**

paramètre → tableau de float et en entier i, retourne un boolean

➔ **auPalet(List<Float> positions_potentielles, int angle_de_balayage)**

paramètre → liste de position potentielles et angle de balayage, retourne entier angle pour chercher le palet.

6.1.2. Description

➔ **changerVitLin(double s):**

Retour: Change la vitesse de déplacement du robot pour qu'elle soit égale à s.
Plage de paramètres : Tout double valide (positif, négatif, zéro).
Comportement attendu : La vitesse de déplacement doit changer pour correspondre à la valeur de s.

➔ **changerVitRot(double s):**

Retour:Change la vitesse de rotation du robot pour qu'elle soit égale à s.
Plage de paramètres : Tous les entiers (positifs, négatifs, zéro).
Comportement attendu : Le robot doit tourner de l'angle spécifié sans interrompre d'autres fonctionnalités.

➔ **tournerSync(int a)**

Retour: Tourne de l'angle passé en paramètres en effectuant une autre tâche.
Plage de paramètres : Tout entier est accepté en tant que vitesse de rotation (s).
Comportement attendu : Le robot doit tourner à une vitesse correspondant à l'entier spécifié.

➔ **avancer(double s)**

Retour : Avance de la distance passée en paramètre.
Plage de paramètres : Tout nombre réel valide est accepté comme distance (s).
Comportement attendu : Le robot doit avancer d'une distance correspondant à la valeur spécifiée.

➔ **avancerSync(double distance)**

Retour : Avance de la distance passée en paramètre tout en permettant d'exécuter d'autres tâches.
Plage de paramètres : Tous les nombres réels sont acceptés comme distances (distance).
Comportement attendu : Le robot doit avancer selon la distance spécifiée sans bloquer l'exécution d'autres tâches.

➔ **tourner(int angle)**

Retour : Tourne de l'angle passé en paramètre.
Plage de paramètres : Tous les entiers sont acceptés comme angles (angle).
Comportement attendu : Le robot doit tourner selon l'angle spécifié.

➔ **stop()**

Retour : Arrête l'exécution de la fonction.
Plage de paramètres : Pas de paramètres requis.
Comportement attendu : La fonction doit arrêter l'exécution du robot.

➔ **echantillon()**

Retour : Retourne un tableau contenant les distances captées en centimètres.
Plage de paramètres : Pas de paramètres requis.

Comportement attendu : La fonction doit retourner un tableau valide de distances captées par les capteurs.

➔ **isMoving()**

Retour : Retourne si le robot bouge ou pas.
Plage de paramètres : Pas de paramètres requis.
Comportement attendu : La fonction doit renvoyer un booléen indiquant si le robot est en mouvement ou non.

➔ **getTouche()**

Retour : Retourne l'attribut du capteur.
Plage de paramètres : Pas de paramètres requis.
Comportement attendu : La fonction doit retourner l'état actuel de l'attribut du capteur.

➔ **ouvrirPinces(int a)**

Retour : Ouvre les pinces de l'angle passé en paramètres.
Plage de paramètres : Tous les entiers sont acceptés comme angles (a).
Comportement attendu : Les pinces doivent s'ouvrir selon l'angle spécifié.

➔ **fermerPinces(int a)**

Retour : Ferme les pinces de l'angle passé en paramètres.
Plage de paramètres : Tous les entiers sont acceptés comme angles (a).
Comportement attendu : Les pinces doivent se fermer selon l'angle spécifié.

➔ **msDelay(int s)**

Retour : Le robot s'arrête pendant s millisecondes.
Plage de paramètres : Tous les entiers sont acceptés comme délai en millisecondes (s).
Comportement attendu : Le robot doit s'arrêter pendant la durée spécifiée.

➔ **ecartImp(float[], int)**

Retour : Retourne s'il y a un écart important (supérieur au seuil i passé en paramètres) entre la distance de l'objet détecté et les distances autour de cet objet.
Plage de paramètres : Un tableau de valeurs flottantes et un entier comme seuil (i).
Comportement attendu : La fonction doit évaluer s'il existe un écart significatif entre la distance de l'objet détecté et les distances environnantes, selon le seuil spécifié.

➔ **estDansLaPortee(float[] tab, int i)**

Retour : Retourne si la différence entre deux distances est supérieure au seuil donné.
Plage de paramètres : Un tableau de valeurs flottantes et un entier comme seuil (i).
Comportement attendu : La fonction doit vérifier si la différence entre deux distances dans le tableau est supérieure au seuil spécifié.

➔ **auPalet(List<Float> positions_potentielles, int angle_de_balayage)**

Retour : Retourne un entier correspondant à la plus petite valeur de la liste de positions potentielles (la distance la plus courte entre le robot et un palet).
Plage de paramètres : Une liste de positions potentielles et un entier comme angle de balayage.
Comportement attendu : La fonction doit renvoyer la plus petite distance dans la liste des positions potentielles, tenant compte de l'angle de balayage spécifié.

6.1.4. Dépendances

➢ **Composants physiques du robot**

Garantir le bon fonctionnement de tous les moteurs, capteurs, actionneurs (pinces) et autres composants physiques du robot est une condition préalable à toutes les méthodes mises en œuvre.

➢ **Capteurs de Détection**

vérifiez spécialement le fonctionnement des capteurs du robot pour vous assurer que certaines fonctions comme echantillon(), getTouche(), ecartImp(), estDansLaPortee() et auPalet() fonctionnent correctement car ces fonctions en dépendent a cette détecteurs utilisés pour localiser des objets
comme des palettes et détecter les obstacles et les distances.

➢ **Power Supply et résilience**

La stabilité et l'adéquation de l'alimentation électrique ont un impact direct sur les performances du robot. Assurez-vous que le robot est complètement chargé et qu'il est donc capable de rester dans les meilleures performances tout au long de la phase de test et de la compétition finale.

6.1.5. Procédure de test Annexes

La procédure de test est une étape essentielle pour garantir que les sous-méthodes répondent aux exigences spécifiées. Les méthodes principales sont donc conçues pour garantir une vérification rigoureuse de chaque fonctionnalité. La procédure est divisée en plusieurs étapes :

➢ **Initialisation du robot**

Vérification de l'état initial du robot, cette étape est essentiellement liée à la section de dépendance écrite juste avant telle que les composants physiques du robot ,capteurs de Détection et le power Supply et résilience.

➢ **Exécution de tests unitaires**

chaque méthode est testée individuellement en utilisant différentes entrées, couvrant différents scénarios :

- Vérification des valeurs limites (positives, négatives, zéro).
- Testez avec des valeurs aléatoires et extrêmes.
- Intégration de plusieurs méthodes pour évaluer le comportement dans des situations réelles. (méthode Sync).

➢ **Analyser et évaluer les retours et le comportement des méthodes**

s'assurer que le comportement de la méthode répond aux attentes énoncées dans la description.

➢ **Documentation des résultats**

Les résultats de chaque test unitaire sont documentés, en notant les entrées utilisées, les retours obtenus et les comportements observés. Toute anomalie ou erreur est documentée pour des contrôles ultérieurs.

8. Glossaire

Ce glossaire sert de guide de référence de vocabulaire pour les mots spécifiques utilisés dans ce document, présentant une compilation de termes spécialisés ainsi que leurs définitions respectives, aidant à la compréhension et à la clarté du contenu.

➢ **Maîtrise d’œuvre**

Désigne l’équipe responsable de l’exécution des tâches du projet.

➢ **Maîtrise d'ouvrage**

Représente le chef de projet chargé de définir les exigences du projet et de l'évaluer.

➢ **Lejos EV3**

Référence à l'environnement de programmation et à l'API utilisés pour le développement du projet.

➢ **Test Fonctionnelles**

Scénarios de tests définis pour les fonctionnalités du robot telles que le mouvement, la rotation et la détection de palettes.

- **Test d’intégration**

Évalue l’intégration de divers composants ou modules du robot.

- **Test Unitaires**

Tests individuels pour valider la fonctionnalité de méthodes ou de composants spécifiques du robot.

9. Reference

- Site documentation Lejos, “Overview (leJOS EV3 API documentation) (sourceforge.io)”
- Cahier des charges
- Plan de Développement

10. Index

Les mots-clés suivants sont répartis dans différentes parties du document et sont liés à divers aspects de la programmation, des tests et des performances du robot.