Compétition de robotique *FIRST*® 2024

Guide de programmation Java pour KitBot

1 Table des matières

2 Présentation du document

Ce document vous guidera à travers la façon de préparer votre KitBot 2024 et de le faire fonctionner en utilisant l'exemple de code Java fourni. Pour éviter la duplication de contenu, ce document fait fréquemment référence à la documentation WPILib [NdT La documentation WPILib est disponible en français] pour des étapes spécifiques du développement. En plus de vous rendre fonctionnel avec le code fourni, ce document passera en revue la structure de ce code afin que vous puissiez comprendre comment il fonctionne. Enfin, nous passerons en revue certains des changements les plus probables que vous voudrez peut-être apporter au code et fournirons des exemples concrets de la façon d'apporter ces modifications.

Pour commencer avec l'exemple de code, ou pour y apporter certaines modifications, une compréhension minimale de Java est nécessaire. Le code et les exemples de modification fournis vous permettront d'avancer avec suffisamment de structure. Pour comprendre la procédure pas à pas, ou pour apporter des modifications non décrites dans ce document, une compréhension plus complète de Java est probablement requise. Le module Intro to Programming (voa) sur Thinkscape est un excellent moyen d'en apprendre davantage sur Java en utilisant WPILib et les plates-formes robotisées Romi ou XRP. Pour d'autres options, consultez les liens sur la page Web d'Introduction De Zéro à Robot.

Ce document, et l'exemple de code fourni, suppose l'utilisation des contrôleurs SPARK MAX fournis dans le kit de lancement des recrues.

3 Prise en main de votre code pour KitBot

3.1 Câblage de votre robot

Utilisez le document Introduction au câblage d'un robot FRC sur WPILib pour vous aider à câbler votre robot. Quelques notes spécifiques au KitBot 2024 :

- Le KitBot 2024 n'utilise pas de pneumatique. Vous pouvez ignorer les instructions concernant le concentrateur pneumatique ou le module de commande pneumatique, sauf si vous ajoutez de la pneumatique à votre design.
- Afin d'utiliser les mêmes ID pour le fonctionnement avec PWM et CAN, le code KitBot 2024 n'utilise pas le port PWM 0. Câblez les ports PWM en fonction des ID dans Constants.java (Gauche = 1,2; Droite = 3,4) ou modifiez les constantes pour refléter votre câblage.
- Le KitBot 2024 contient deux moteurs supplémentaires non inclus dans le document de câblage de base. Câblez-les de la même manière que les moteurs du système motopropulseur. Si vous utilisez les PWM, connectez le moteur de l'admission (le plus près du centre du robot) au port PWM 5 et le moteur du lanceur (le moteur plus proche de l'extérieur du robot) au port PWM 6.

3.2 Configuration du matériel et de l'environnement de développement

Avant de pouvoir charger du code et tester votre robot, vous devrez configurer votre matériel (roboRIO, radio, etc.) et configurer votre environnement de développement. Suivez les étapes 2 à 4 du guide d'Installation des composants logiciels de WPILib pour tout configurer et vous assurer que vous pourrez déployer un projet de robot de base.

Si vous utilisez les PWM, assurez-vous que les 6 MAX SPARK sont en mode « brushed » (voa). Lorsqu'alimenté, la DEL doit clignoter en jaune ou en bleu, pas en magenta ou en cyan. Pour changer le mode, vous pouvez soit maintenir le bouton Mode enfoncé pendant 3 secondes, soit utiliser la connexion USB et le client matériel REV. Vous pouvez également vérifier les modes neutres, frein ou libre (voa). Il est recommandé de régler les moteurs de l'admission et du lanceur en mode libre (jaune clignotant). Pour modifier le mode, appuyez brièvement sur le bouton Mode (moins de 3 secondes) ou utilisez la connexion USB et REV Hardware Client. Il n'y a pas de recommandation spécifique pour les moteurs de la propulsion, mais vous voudrez probablement que les 4 moteurs de propulsion soient sur le même mode, vous voudrez peut-être essayer de piloter avec chaque réglage pour décider ce que vous préférez.

3.3 Ouverture de l'exemple KitBot 2024

L'exemple de code pour KitBot 2024 est fourni dans des fichiers zip individuels pour chaque langage sur la page Web du KitBot (voa). Pour ouvrir le code Java :

1. Téléchargez et décompressez l'exemple de code Java. Assurez-vous de décompresser ou de copier vers un emplacement permanent, pas dans un dossier temporaire.
2. Ouvrez **WPILib VS Code** à l'aide du menu Démarrer ou des raccourcis du bureau

3. En haut à gauche, cliquez sur **File->Open Folder** et accédez au dossier « Java » à l'intérieur de l'exemple de code décompressé, puis cliquez sur **Select Folder**.

3.4 Passage au contrôle via CAN

Si vous avez câblé vos contrôleurs de moteur SPARK MAX par filage CAN, vous devrez effectuer d'autres modifications de configuration et de code avant de continuer. Si vous utilisez les PWM, passez à la section 3.5 pour déployer et tester le code.

3.4.1 Configuration des SPARK MAX

Avant d'utiliser les SPARK MAX avec contrôle CAN, chacun d'eux doit se voir attribuer un ID unique. Étant donné que vos SPARK commencent tous avec le même ID, vous pouvez débrancher le bus CAN de chaque appareil pendant que vous mettez à jour et attribuez un ID.

1. Installez le client matériel REV Hardware Client (voa)
2. Avec le robot éteint, connectez un câble USB entre l'ordinateur et le port USB du SPARK MAX. Laissez le robot éteint garantit que seul le SPARK MAX est alimenté et évite de changer les ID sur des appareils non désirés.
3. Installez la mise à jour du firmware on the SPARK MAX (voa)
4. Sélectionnez le ID CAN et le type de moteur (CAN ID and Motor Type -voa). Vous pouvez passer la limite de courant et sauvegarder les changements.
    a. Les ID CAN de chaque appareil se trouvent dans Constants.java. Vous pouvez soit définir les appareils pour qu'ils correspondent à ces ID, soit définir les ID comme vous le souhaitez (certaines équipes définissent ID CAN = le numéro de canal auquel l'appareil est connecté sur le PD), puis mettez à jour ces constantes.
    b. Remarque : Si vous souhaitez « Faire tourner le moteur » comme décrit sur cette page, assurez-vous que le robot est dans un état sûr pour le faire (roues ne touchant pas le sol ou la table).
5. Répétez l'opération pour les 6 appareils du robot.
6. Bien que cela ne soit pas nécessaire, si vous utilisez le PDH de REV, vous voudrez peut-être vérifier maintenant qu'il a également le dernier firmware. Ne changez pas l'ID du PDH hors de la valeur par défaut, chaque type d'appareil a un espace distinct d'identification et votre PDH n'entrera pas en conflit avec votre SPARK MAX même s'il est placé au même ID.

Maintenant que tous vos appareils sont configurés, vous pouvez effectuer une vérification préliminaire que votre bus CAN est câblé correctement à l'aide du client matériel REV. Lorsque vous êtes branché avec un câble USB sur n'importe quel appareil REV de votre bus CAN, mettez le robot sous tension et vous devriez voir tous les autres appareils répertoriés dans le volet gauche du client matériel REV, sous l'en-tête CAN Bus. Si vous ne voyez pas tous les appareils, vous avez probablement un ou plusieurs problèmes avec le câblage de votre bus CAN :

1. Vérifiez que votre bus CAN commence par le roboRIO et se termine par une résistance de 120 ohms, ou le terminateur intégré d'un concentrateur de distribution d'énergie ou d'un panneau de distribution électrique (avec la terminaison réglée sur ON à l'aide du cavalier ou du commutateur approprié.

2. Vérifiez que vos connexions de bus CAN correspondent toutes à jaune-jaune et vert-vert.
3. Vérifiez que toutes les connexions de fil CAN sont sécurisées les unes aux autres et que les connecteurs sont installés en toute sécurité dans chaque SPARK Max
4. Si vous rencontrez toujours des problèmes, déplacer la connexion USB vers différents appareils. Vérifier ce que chaque appareil peut « voir » dans le bus peut aider à localiser l'emplacement d'un problème.

3.4.2 Installation de REVLib

La bibliothèque logicielle du SPARK MAX en mode CAN est fournie par le fournisseur (REV Robotics). La configuration de la bibliothèque de tierce partie est déjà incluse dans le projet, mais vous devrez installer la bibliothèque elle-même. Il y a deux façons de le faire :

1. **Recommandé** Installer la bibliothèque hors ligne - Cela garantira que la bibliothèque demeure sur votre ordinateur même si vous ne créez pas de nouveau code pendant un certain temps (les installations en ligne peuvent être nettoyées automatiquement par Gradle).
    a. Téléchargez la dernière version de REVLib à l'aide du lien de la documentation REV (voa).
    b. Décompressez dans le répertoire *C :\Users\Public\wpilib\2024* sous Windows et le répertoire *~/wpilib/2024* sur les systèmes de type Unix.
2. Installer en ligne - Pendant que l'ordinateur est connecté à Internet, cliquez sur l'icône WPILib en haut à droite de la fenêtre VSCode pour afficher l'invite d'extension WPILib, puis commencez à taper « Build Robot Code » et sélectionnez cette option lorsqu'elle apparaît. Cela récupérera automatiquement la bibliothèque en ligne.

3.4.3 Mise à jour du code

Le code est principalement en place dans le projet pour passer du contrôle PWM au contrôle CAN, il vous suffira de faire quelques modifications pour basculer.

1. Dans RobotContainer:

```
//import frc.robot.subsystems.PWMDrivetrain;
//import frc.robot.subsystems.PWMLauncher;
import frc.robot.subsystems.CANDrivetrain;
import frc.robot.subsystems.CANLauncher;
```

```
public class RobotContainer {
  // The robot's subsystems are defined here.
  //private final PWMDrivetrain m_drivetrain = new PWMDrivetrain();
  private final CANDrivetrain m_drivetrain = new CANDrivetrain();
//private final PWMLauncher m_launcher = new PWMLauncher();
  private final CANLauncher m_launcher = new CANLauncher();
```

    a. Décommentez les instructions d'importation pour CANDrivetrain et CANLauncher. Vous pouvez porter en commentaire ou supprimer les instructions pour les sous-systèmes PWM si vous le souhaitez.

b. Portez en commentaire les lignes de déclaration des variables membres pour les sous-systèmes PWM et décommentez celles des sous-systèmes CAN.

2. Dans LaunchNote et PrepareLaunch:

```
import edu.wpi.first.wpilibj2.command.CommandBase;
//import frc.robot.subsystems.PWMLauncher;
import frc.robot.subsystems.CANLauncher;
import static frc.robot.Constants.LauncherConstants.*;

public class PrepareLaunch extends CommandBase {
  //PWMLauncher m_launcher;
  CANLauncher m_launcher;

  /** Creates a new PrepareLaunch. */
  public PrepareLaunch(CANLauncher launcher) {
```

   a. Décommentez l'importation pour CANLauncher et portez en commentaire ou supprimer celui pour PWMLauncher
   b. Décommentez la déclaration de variable membre pour le CANLauncher et portez en commentaire ou supprimez celle de PWMLauncher.
   c. Remplacez le type de paramètre du constructeur par CANLauncher.

3. Dans Autos:

```
//import frc.robot.subsystems.PWMDrivetrain;
import frc.robot.subsystems.CANDrivetrain;
import edu.wpi.first.wpilibj2.command.CommandBase;
import edu.wpi.first.wpilibj2.command.RunCommand;

public final class Autos {
  /** Example static factory for an autonomous command. */
  public static CommandBase exampleAuto(CANDrivetrain drivetrain) {
```

   a. Décommentez l'importation du CANDrivetrain et commentez ou supprimez l'importation du PWMDrivetrain
   b. Changez le type de paramètre de méthode de PWMDrivetrain à CANDrivetrain

3.5 Déploiement et test de l'exemple pour KitBot

Pour déployer l'exemple sur votre robot, vous devrez définir le numéro d'équipe dans le projet. Cliquez sur **l'icône WPILib** dans le coin supérieur droit de la fenêtre VS Code (W à l'intérieur d'un engrenage) pour ouvrir l'invite WPILib et commencez à taper « Set Team Number » et sélectionnez cette option lorsqu'elle apparaît. Entrez votre numéro d'équipe (pas de préfixe 0 - par exemple 123 ou 9996) et appuyez sur Enter.

Vous êtes maintenant prêt à déployer l'exemple pour KitBot tout comme vous avez déployé le projet test à l'étape 4 du guide Zéro à Robot.

Avertissement: Assurez-vous d'avoir de l'espace dans toutes les directions lorsque vous faites fonctionner un robot. Même avec un code connu, le robot peut se déplacer à une vitesse ou dans des directions inattendues. Soyez prêt à désactiver le robot (Enter) ou à faire un arrêt d'urgence E-stop (barre d'espace) si nécessaire. Le code pour KitBot 2024 contient une routine autonome très simple qui déplacera le robot vers l'arrière à une vitesse de 1/2 pendant 1 seconde lorsque le robot est activé en mode autonome.

3.6 Configuration des manettes de jeu

Le code est configuré pour utiliser la classe de manette Xbox. Les manettes de jeu Logitech F310 fournies dans le kit de pièces apparaîtront comme des manettes Xbox dans le logiciel WPILib si elles sont configurées dans le mode correct. Pour configurer les contrôleurs, vérifiez que le commutateur à l'arrière du contrôleur est défini sur le paramètre « X ». Ensuite, lorsque vous utilisez le contrôleur, assurez-vous que le voyant à côté du bouton Mode est éteint, s'il est allumé, appuyez sur le bouton Mode pour le basculer. Lorsque le bouton Mode est allumé, le contrôleur permute la fonction du joystick analogique gauche et du D-pad.

3.7 Que fait le code ?

Le code fourni implémente les commandes de robot suivantes dans le mode téléopéré :

- La manette de contrôle est une manette Xbox dans l'emplacement 0 de DriverStation
    - Contrôle la propulsion du robot à l'aide de Split-stick Arcade Drive
        - L'axe Y (vertical) du joystick gauche contrôle le mouvement avant-arrière de la plateforme pilotable
        - L'axe X (horizontal) du joystick droit contrôle la rotation de la plateforme pilotable
- La manette d'opérateur est une manette Xbox dans l'emplacement 1 de la station de pilotage
    - Gâchette avant gauche - Active les deux roues du lanceur de pièce de jeu vers l'intérieur à des vitesses différentes pendant que le bouton est maintenu. Cela permet au robot d'admettre une pièce de jeu
    - Bouton A – Exécute une courte séquence pour lancer une pièce de jeu pendant que le bouton est maintenu
        - Démarre la roue avant pour atteindre sa pleine vitesse
        - Attend 1 seconde
        - Active la roue arrière pour transmettre une pièce de jeu vers la roue avant qui tourne

4 Structure globale du code

Le code fourni utilise la structure de programmation orientée commandes (Command-Based) fournie par WPILib. Cette structure divise les actionneurs du robot en « sous-systèmes » qui sont contrôlés par des « commandes » ou des collections de commandes (aussi appelées « groupes de commandes »). La structure orientée commande peut être un peu exagérée pour un robot de cette complexité, mais elle s'adapte très bien pour les équipes qui cherchent à ajouter des fonctionnalités supplémentaires à leur KitBot, et elle fournit de nombreux outils utiles pour gérer les séquences et les actions chronométrées comme on le verra en regardant le code du lanceur de pièce de jeu. De plus, cette structure de code a été utilisée par plus de 60% des équipes en 2023, ce qui augmente la probabilité que les équipes autour de vous puissent fournir de l'aide avant ou pendant l'événement.

Pour en savoir plus sur la structure orientée commandes, consultez le chapitre sur la Programmation orientée commandes sur WPILib.

4.1 Façons de créer des commandes

Il existe plusieurs façons de définir des commandes dans la structure orientée commandes. Ce projet utilise bon nombre de ces différents types afin d'illustrer ce à quoi ils ressembleraient dans un projet de robot complet. Si un type vous semble étrange ou n'a pas beaucoup de sens, ne vous inquiétez pas, vous devriez être en mesure d'utiliser la documentation sur ce que fait la commande combinée avec des exemples d'autres façons de créer des commandes pour la recréer sous la forme que vous préférez. Les méthodes courantes de création de commandes utilisées dans ce projet sont les suivantes :

- Définie comme leur propre classe dans leur propre fichier (par exemple PrepareLaunch.java et LaunchNote.Java)
- Via une méthode « Fabrique de commande » dans le sous-système (par exemple getIntakeCommand() dans *Launcher.java)
- Inline où la commande est liée à un bouton ou définie par défaut (par exemple, la ligne 53 de RobotContainer.java)

Ces mêmes méthodes s'appliquent également aux groupes de commandes. Ce projet utilise :

- Groupe de commandes inline où la liaison se produit. Cela se fait via des « décorateurs », des méthodes auxquelles vous pouvez faire appel sur une commande pour la transformer en un groupe de commandes avec des propriétés spécifiques, comme « withTimeout » (par exemple, la ligne 62 de RobotContainer.java)
- Fabrique de group de commandes dans son propre fichier (Autos.java)

Ce projet ne crée aucun groupe de commandes dans son propre fichier, un exemple peut être vu au bas de la page Groupes de commandes liée ci-dessus ainsi que sur la page Web Organisation des projets orientés commandes sur WPILib.

La plupart des équipes n'utiliseront pas tous ces styles dans leur code, optant plutôt pour sélectionner un ou deux types qui leur conviennent le mieux.

5 Présentation du code

5.1 Sous-systèmes

Comme décrit dans Qu'est-ce que la programmation « orientée commandes » ? les « sous-systèmes représentent des collections d'équipements contrôlées indépendamment (tels que des contrôleurs de moteur, des capteurs, des actionneurs pneumatiques, etc.) qui fonctionnent ensemble.

Pour le KitBot 2024, nous avons regroupé les 6 moteurs en 2 groupes, la propulsion et le lanceur. Pour ce robot, ces choix étaient assez faciles, les 4 moteurs de la propulsion doivent toujours travailler ensemble pour déplacer le robot sur le terrain et les 2 moteurs du lanceur doivent toujours travailler ensemble pour manipuler les pièces de jeu. Parfois, les limites entre les sous-systèmes peuvent ne pas être si claires, si vous avez un bras avec une articulation de l'épaule et du poignet et un ensemble de roues motorisées à l'extrémité, est-ce un sous-système ou plusieurs ? La règle générale à suivre est de réfléchir aux actions, ou commandes, que vous pourriez avoir pour contrôler les sous-systèmes. Pensez-vous que vous voudrez que deux pièces soient contrôlées indépendamment l'une de l'autre (par exemple, actionner l'admission tout en déplaçant le bras ou le poignet ?). Si vous n'êtes pas sûr, allez vers des sous-systèmes plus petits ; vous pouvez toujours créer des commandes qui nécessitent plusieurs sous-systèmes, mais si vous finissez par vouloir des commandes distinctes pour contrôler un seul sous-système en même temps, vous devrez refactoriser le sous-système pour le diviser.

5.1.1 PWMDrivetrain

Cette classe est le sous-système de la propulsion si vous avez câblé votre contrôleur de moteur par filage PWM. Si vous avez câblé votre contrôleur de moteur à l'aide de CAN, consultez la section 3.4 pour obtenir des renseignements sur la façon de décommenter l'utilisation de cette classe et remplacez-la par la classe CANDrivetrain.

5.1.1.1 Packs et importations

```
package frc.robot.subsystems;

import static frc.robot.Constants.DrivetrainConstants.*;

import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj.motorcontrol.MotorControllerGroup;
import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
```

Cette section indique de quel pack notre sous-système fait partie (les packs sont un moyen d'organiser les classes Java) et quelles autres classes nous devons référencer dans ce code (importations). Une pratique courante consiste à ajouter des importations au fur et à mesure ; lorsque vous référencez une classe que vous n'avez pas encore importée. Vous pouvez utiliser l'ampoule que VSCode affichera pour que vous puissiez ajouter une importation pour cette classe. La ligne du milieu est un type spécial d'instruction d'importation, appelé importation statique, cela nous permet de référencer les constantes

déclarées dans cette classe sans aucun modificateur de classe (par exemple kLeftFrontID au lieu de DrivetrainConstants.kLeftFrontID) permettant au code d'être un peu plus compact.

5.1.1.2 Déclaration de classe, variables de membre et constructeur

```
public class PWMDrivetrain extends SubsystemBase {
  /*Class member variables. These variables represent things the class needs to keep track of and use between
  different method calls. */
  DifferentialDrive m_drivetrain;

  /*Constructor. This method is called when an instance of the class is created. This should generally be used to set up
   * member variables and perform any configuration or set up necessary on hardware.
   */
  public PWMDrivetrain() {
    /*Create MotorControllerGroups for each side of the drivetrain. These are declared here, and not at the class level
     * as we will not need to reference them directly anymore after we put them into a DifferentialDrive.
     */
    MotorControllerGroup leftMotors =
        new MotorControllerGroup(new PWMSparkMax(kLeftFrontID), new PWMSparkMax(kLeftRearID));
    MotorControllerGroup rightMotors =
        new MotorControllerGroup(new PWMSparkMax(kRightFrontID), new PWMSparkMax(kRightRearID));

    // Invert left side motors so both sides drive forward with positive output values
    leftMotors.setInverted(isInverted:true);
    rightMotors.setInverted(isInverted:false);

    // Put our controller groups into a DifferentialDrive object. This object represents all 4 motor
    // controllers in the drivetrain
    m_drivetrain = new DifferentialDrive(leftMotors, rightMotors);
  }
```

La première ligne de cette image est la déclaration de classe. Cela déclare le nom de notre classe et indique qu'il s'agit d'une extension de la classe SubsystemBase. Tous les sous-systèmes doivent étendre cette classe qui fournit certaines fonctions utilitaires concernant la définition du nom du sous-système, son enregistrement auprès du Planificateur (scheduler) et l'envoi d'informations à ce sujet au tableau de bord.

La section suivante présente les variables membres de la classe. Ce sont des objets que nous devons conserver entre les appels aux méthodes de classe. Cela inclut généralement le matériel associé au sous-système et parfois certaines variables d'état représentant l'état du système. Pour notre plateforme pilotable simple, l'objet DifferentialDrive est tout ce dont nous avons besoin de stocker.

La dernière section est le constructeur. Ici, nous initialisons toutes les variables contenues dans le sous-système.

5.1.1.3 Méthodes

```
/*Method to control the drivetrain using arcade drive. Arcade drive takes a speed in the X (forward/back) direction
* and a rotation about the Z (turning the robot about it's center) and uses these to control the drivetrain motors */
public void arcadeDrive(double speed, double rotation) {
  m_drivetrain.arcadeDrive(speed, rotation);
}

@Override
public void periodic() {
  /*This method will be called once per scheduler run. It can be used for running tasks we know we want to update each
  * loop such as processing sensor data. Our drivetrain is simple so we don't have anything to put here */
}
```

Le reste de la classe de sous-système est des méthodes. Ici, nous définissons toutes les méthodes que les commandes peuvent avoir besoin d'appeler pour obtenir l'état ou effectuer des actions sur le sous-système. Pour notre simple base pilotable, la seule méthode dont nous avons besoin est la méthode arcadeDrive qui transmet simplement les paramètres à la même méthode sur l'objet DifferentialDrive. La dernière méthode de cette classe, la méthode « periodic », est une méthode spéciale qui est appelée à chaque cycle par le Planificateur (scheduler), quelle que soit la commande en cours d'exécution. Vous pouvez effectuer ici des tâches que vous savez que vous souhaitez effectuer périodiquement, telles que la mise à jour des données des capteurs. Notre propulsion n'a pas de tâches comme celle-ci, vous pouvez choisir de supprimer cette méthode si vous le souhaitez.

5.1.2 CANDrivetrain

Cette classe est le sous-système de la base pilotable si vous avez câblé votre contrôleur de moteur par filage CAN. Si vous avez câblé votre contrôleur de moteur à l'aide de CAN, consultez la section 3.4 pour plus d'informations sur la mise en commentaire de l'utilisation de PWMDrivetrain et son remplacement par la classe CANDrivetrain.

5.1.2.1 Packs et importations

```
package frc.robot.subsystems;

import static frc.robot.Constants.DrivetrainConstants.*;

import com.revrobotics.CANSparkMax;
import com.revrobotics.CANSparkMaxLowLevel.MotorType;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
```

Cette section indique de quel pack notre sous-système fait partie (les packs sont un moyen d'organiser les classes Java) et quelles autres classes nous devons référencer dans ce code (importations). Une pratique courante consiste à ajouter les importations au fur et à mesure ; lorsque vous vous trouvez à référencer une classe que vous n'avez pas encore importée, vous pouvez utiliser l'ampoule que VSCode affichera pour que vous puissiez ajouter une importation pour cette classe. La ligne du milieu est un type spécial d'instruction d'importation, appelé importation statique, cela nous permet de référencer les constantes déclarées dans cette classe sans aucun modificateur de classe (par exemple

kLeftFrontID au lieu de DrivetrainConstants.kLeftFrontID) permettant au code d'être un peu plus compact.

Les deux premières lignes de la dernière section déclarent les importations REVRobotics. Alors que les appareils de base sont pris en charge directement par WPILib, les appareils plus complexes sont pris en charge par les logiciels fournis par leur fournisseur. La bibliothèque du fournisseur REV a déjà été ajoutée à cet exemple de projet pour vous, mais vous pouvez en savoir plus sur la gestion des bibliothèques de fournisseurs sur la page Web des Librairies tierces sur WPILib si vous devez en ajouter d'autres ou créer de nouveaux projets qui utilisent CANSparkMax.

5.1.2.2 Déclaration de classe, variables de membre

```
public class CANDrivetrain extends SubsystemBase {
/*Class member variables. These variables represent things the class needs to keep track of and use between
different method calls. */
DifferentialDrive m_drivetrain;
```

La première ligne de cette image est la déclaration de classe. Cela déclare le nom de notre classe et indique qu'il s'agit d'une extension de la classe SubsystemBase. Tous les sous-systèmes doivent étendre cette classe qui fournit certaines fonctions utilitaires concernant la définition du nom du sous-système, son enregistrement auprès du Planificateur (scheduler) et l'envoi d'informations au tableau de bord.

La section suivante présente les variables membres de la classe. Ce sont des objets que nous devons conserver entre les appels aux méthodes de classe. Cela inclut généralement le matériel associé au sous-système et parfois certaines variables d'état représentant l'état du système. Pour notre base pilotable, l'objet DifferentialDrive est tout ce dont nous avons besoin de stocker

5.1.2.3 Constructeur

```
public CANDrivetrain() {
  CANSparkMax leftFront = new CANSparkMax(kLeftFrontID, MotorType.kBrushed);
  CANSparkMax leftRear = new CANSparkMax(kLeftRearID, MotorType.kBrushed);
  CANSparkMax rightFront = new CANSparkMax(kRightFrontID, MotorType.kBrushed);
  CANSparkMax rightRear = new CANSparkMax(kRightRearID, MotorType.kBrushed);

  /*Sets current limits for the drivetrain motors. This helps reduce the likelihood of wheel spin, reduces motor heating
   *at stall (Drivetrain pushing against something) and helps maintain battery voltage under heavy demand */
  leftFront.setSmartCurrentLimit(kCurrentLimit);
  leftRear.setSmartCurrentLimit(kCurrentLimit);
  rightFront.setSmartCurrentLimit(kCurrentLimit);
  rightRear.setSmartCurrentLimit(kCurrentLimit);

  // Set the rear motors to follow the front motors.
  leftRear.follow(leftFront);
  rightRear.follow(rightFront);

  // Invert the left side so both side drive forward with positive motor outputs
  leftFront.setInverted(isInverted:true);
  rightFront.setInverted(isInverted:false);

  // Put the front motors into the differential drive object. This will control all 4 motors with
  // the rears set to follow the fronts
  m_drivetrain = new DifferentialDrive(leftFront, rightFront);
}
```

Cette section est le constructeur, où nous initialisons et configurons le matériel pour le sous-système. La première section déclare les contrôleurs de moteur et indique qu'ils sont connectés à des moteurs à brosse (les moteurs CIM sont des moteurs à brosse, vous changeriez cela en « brushless » si vous utilisez un moteur NEO ou NEO 500 qui se connecte aux 3 fils du Spark Max). Ceux-ci sont utilisés comme variables locales car nous n'avons pas besoin de les référencer directement après les avoir utilisés dans le constructeur. Si vous aviez des capteurs rattachés à eux ou si vous vouliez obtenir d'autres informations comme le courant ou la température, vous pourriez choisir de les déplacer pour être des variables membres de classe

La section suivante définit les limites de courant sur chacun des moteurs. Après cela, les moteurs arrière sont réglés pour suivre les moteurs avant, de leur côté respectif. Ceci indique à ces contrôleurs d'écouter le trafic vers ou depuis les contrôleurs avant et d'utiliser ces informations pour apparier la direction et la puissance de sortie. Cela crée un peu moins de trafic CAN que de mettre les contrôleurs de chaque côté dans un groupe comme cela se fait dans le PWMDrivetrain (bien que cela fonctionne très bien aussi !).

Ensuite, le côté gauche est inversé. C'est pour qu'une commande positive entraîne le déplacement des roues de ce côté du robot vers l'avant.

La dernière ligne configure le DifferentialDrive avec les contrôleurs avant. Les contrôleurs arrière suivront de sorte que l'objet DifferentialDrive n'a pas besoin de les connaître.

5.1.2.4 Méthodes

```
/*Method to control the drivetrain using arcade drive. Arcade drive takes a speed in the X (forward/back) direction
* and a rotation about the Z (turning the robot about it's center) and uses these to control the drivetrain motors */
public void arcadeDrive(double speed, double rotation) {
  m_drivetrain.arcadeDrive(speed, rotation);
}

@Override
public void periodic() {
  /*This method will be called once per scheduler run. It can be used for running tasks we know we want to update each
  * loop such as processing sensor data. Our drivetrain is simple so we don't have anything to put here */
}
```

Le reste de la classe de sous-système est des méthodes. Ici, nous définissons toutes les méthodes que les commandes peuvent avoir besoin d'appeler pour obtenir l'état ou effectuer des actions sur le sous-système. Pour notre simple plateforme pilotable, la seule méthode dont nous avons besoin est la méthode arcadeDrive qui transmet simplement les paramètres à la même méthode sur l'objet DifferentialDrive. La dernière méthode de cette classe, la méthode « periodic », est une méthode spéciale qui est appelée à chaque cycle par le Planificateur (scheduler), quelle que soit la commande en cours d'exécution. Vous pouvez effectuer ici des tâches que vous souhaitez effectuer périodiquement, telles que la mise à jour des données des capteurs. Notre propulsion n'a pas de tâches comme celle-ci, vous pouvez choisir de supprimer cette méthode si vous le souhaitez.

5.1.3 PWMLauncher

Cette classe est le sous-système du lanceur si vous avez câblé votre contrôleur de moteur par filage PWM. Si vous avez câblé votre contrôleur de moteur à l'aide de filage CAN, consultez la section 3.4

pour plus d'informations sur la mise en commentaire de l'utilisation de cette classe et son remplacement par la classe CANLauncher.

5.1.3.1 Packs, importations, déclaration de classes, variables membres, et le constructeur

Les premières sections de ce sous-système sont très similaires au sous-système PWMDrivetrain. Voir la section 5.1.1 pour une description plus détaillée de chacune de ces parties du code.

5.1.3.2 Méthodes – Fabrique de commande

```
public CommandBase getIntakeCommand() {
    return this.startEnd(
        () -> {
          setFeedWheel(kIntakeFeederSpeed);
          setLaunchWheel(kIntakeLauncherSpeed);
        },
        () -> {
          stop();
        });
}
```

Cette méthode est ce qu'on appelle une "Fabrique de commande" car elle crée les instances d'une commande. Nous pouvons appeler cette méthode d'où on configure des boutons, d'où on crée des groupes de commande, etc. pour obtenir une instance de cette commande. La commande elle-même est créée à l'aide d'un assistant de commande inline (dans notre cas startEnd). Pour voir les différentes options disponibles, consultez JavaDoc for the Subsystem class (voa) et recherchez les méthodes qui retournent un objet CommandBase.

Dans notre cas, nous utilisons startEnd() qui appelle le premier paramètre lorsque la commande démarre et le deuxième paramètre lorsque la commande est interrompue (par exemple par une nouvelle commande étant planifiée ou un bouton étant relâché). Lorsque la commande démarre, nous voulons commencer à faire tourner les deux roues vers l'intérieur à une vitesse spécifique pour faire entrer une pièce de jeu et quand c'est terminé, nous voulons arrêter les roues.

La technique de programmation utilisée ici s'appelle une « expression lambda ». Pour en savoir plus sur les expressions lambda, consultez cette section des documents WPILib sur les Expressions Lambda en Java.

5.1.3.3 Méthodes – Contrôle des équipements

```
public void setLaunchWheel(double speed)
{
  m_launchWheel.set(speed);
}

public void setFeedWheel(double speed)
{
  m_feedWheel.set(speed);
}

public void stop()
{
  m_launchWheel.set(speed: 0);
  m_feedWheel.set(speed: 0);
}
```

Le reste de la classe de sous-système est des méthodes d'accès aux équipements. Ici, nous définissons toutes les méthodes que les commandes peuvent avoir besoin d'appeler pour obtenir l'état ou effectuer des actions sur le sous-système. Pour notre lanceur, cela inclut des méthodes pour régler la vitesse de chaque roue et une méthode pour arrêter les deux roues à la fois. La méthode stop() est un choix de conception, vous pouvez absolument ignorer cette méthode; il suffit d'appeler les méthodes de d'ajustement de vitesse pour chacune des deux roues partout où vous voudriez les arrêter.

5.1.4 CANLauncher

Cette classe est le sous-système du lanceur si vous avez câblé votre contrôleur de moteur par filage CAN. Si vous avez câblé votre contrôleur de moteur à l'aide de CAN, consultez la section 3.4 pour obtenir des renseignements sur la façon de commenter l'utilisation de la classe PWMLauncher et de la remplacer par la classe CANLauncher

5.1.4.1 Packs, importations, Déclaration de clases, Variables membres, et le constructeur

Les premières sections de ce sous-système sont très similaires au sous-système CANDrivetrain. Voir la section 5.1.1 pour une description plus détaillée de chacune des parties du code.

5.1.4.2 Méthodes – Fabrique de commande

```
public CommandBase getIntakeCommand() {
    return this.startEnd(
        () -> {
          setFeedWheel(kIntakeFeederSpeed);
          setLaunchWheel(kIntakeLauncherSpeed);
        },
        () -> {
          stop();
        });
}
```

Cette méthode est ce qu'on appelle une "Fabrique de commande" car elle crée des instances d'une commande. Nous pouvons appeler cette méthode de n'importe où où nous mettons en place des boutons, créons des groupes de commande, etc. pour obtenir une instance de cette commande. La commande elle-même est créée à l'aide d'un assistant de commande inline (dans notre cas startEnd). Pour voir les différentes options disponibles, consultez JavaDoc for the Subsystem class (voa) et recherchez les méthodes qui retournent un objet CommandBase.

Dans notre cas, nous utilisons startEnd() qui appelle le premier paramètre lorsque la commande démarre et le deuxième paramètre lorsque la commande est interrompue (par exemple par une nouvelle commande planifiée ou un bouton libéré). Lorsque la commande démarre, nous voulons commencer à faire tourner les deux roues vers l'intérieur à une vitesse spécifique pour faire entrer une pièce de jeu et quand elle se termine, nous voulons arrêter les roues.

La technique de programmation utilisée ici s'appelle une « expression lambda ». Pour en savoir plus sur les expressions lambda, consultez cette section sur les Expressions Lambda en Java.

5.1.4.3 Méthodes – Contrôle des équipements

```
public void setLaunchWheel(double speed)
{
  m_launchWheel.set(speed);
}

public void setFeedWheel(double speed)
{
  m_feedWheel.set(speed);
}

public void stop()
{
  m_launchWheel.set(speed: 0);
  m_feedWheel.set(speed: 0);
}
```

Le reste de la classe de sous-système est des méthodes d'accès au matériel. Ici, nous définissons toutes les méthodes que les commandes peuvent avoir besoin d'appeler pour obtenir l'état ou effectuer des actions sur un sous-système. Pour notre lanceur, cela inclut des méthodes pour régler la vitesse de chaque roue et une méthode pour arrêter les deux roues à la fois. La méthode stop() est un choix de design, vous pouvez absolument ignorer cette méthode et il suffit d'appeler les méthodes de vitesse paramétrée de chacune des deux roues partout où vous voudriez les arrêter.

5.2 Commandes

Cette section couvrira les 3 fichiers de commande dans le dossier des commandes. Des commandes inline supplémentaires sont définies dans le fichier RobotContainer, elles seront couvertes dans la section suivante.

5.2.1 Autos

Le fichier Autos est un exemple d'une "Fabrique de commande statique". Votre programme ne doit jamais créer un objet Autos (comme le montre le constructeur affichant simplement un message d'erreur). Au lieu de cela, vous appelez des méthodes de classe statiquement en utilisant la syntaxe de type Autos.exampleAuto(). Cette structure est un des moyens de définir des groupes complexes qui impliquent plusieurs sous-systèmes (bien que notre exemple ici ne soit pas complexe et ne nécessite qu'un seul sous-système).

```
public final class Autos {
  /** Example static factory for an autonomous command. */
  public static CommandBase exampleAuto(PWMDrivetrain drivetrain) {
    return new RunCommand(()-> drivetrain.arcadeDrive(-.5, rotation: 0)).withTimeout(seconds: 1)
    .andThen(new RunCommand(()-> drivetrain.arcadeDrive(speed: 0,rotation: 0)));
  }

  private Autos() {
    throw new UnsupportedOperationException("This is a utility class!");
  }
```

Notre exemple de fichier n'a qu'une seule routine autonome à obtenir. Vous pouvez facilement allonger ce modèle en ajoutant des méthodes supplémentaires pour définir plus de routines autonomes et vous pouvez choisir entre elles à l'aide d'un objet SendableChooser sur le tableau de bord.

Cette routine autonome simple demande au robot de reculer pendant 1 seconde à 50% de puissance en utilisant le décorateur Timeout pour définir un délai d'attente de 1 seconde sur la commande de conduite. Il utilise le décorateur andThen pour dire au robot d'arrêter de bouger une fois la première commande terminée. Les différents types de compositions de commandes intégrées via des décorateurs et des méthodes d'usine sont décrits dans la page Command Compositions.

5.2.2 LaunchNote

La commande LaunchNote est le premier de deux exemples dans ce projet d'une commande en tant que classe indépendante définie dans son propre fichier. Cette structure est généralement très utile pour les commandes complexes, mais peut également être utilisée pour des commandes simples, comme indiqué ici, en fonction des préférences. La commande LaunchNote fait tourner les deux roues du lanceur pour éjecter la pièce de jeu du robot.

5.2.2.1 Déclaration de clases, Membres, et le Constructeur

```
public class LaunchNote extends Command {
  PWMLauncher m_launcher;

  //CANLauncher m_launcher;

  /** Creates a new LaunchNote. */
  public LaunchNote(PWMLauncher launcher) {
    // save the launcher system internally
    m_launcher = launcher;

    // indicate that this command requires the launcher system
    addRequirements(m_launcher);
  }
```

La première ligne de cette section définit la classe comme une extension de commande. Cette classe de base définit de nombreuses méthodes d'assistance utilisées pour gérer l'obtention et la définition des exigences, l'interaction avec le Planificateur (scheduler) et la gestion des décorateurs qui modifient la commande ou la collectent dans un groupe.

Ensuite, nous définissons nos variables membres de classe, dans notre cas le sous-système sur lequel la commande fonctionne. Nous devons enregistrer ce sous-système en interne afin d'appeler des méthodes dessus quand la commande est en cours d'exécution.

Enfin, nous avons le constructeur. Ce constructeur prend le sous-système comme paramètre afin qu'il puisse être enregistré localement. La dernière ligne du constructeur indique que cette commande requiert ce sous-système. Cette déclaration d'exigences est la colle qui maintient l'architecture orientée commandes cohérente. Toute commande qui exécute des actions de sortie sur un sous-système doit « exiger » ce sous-système pour aider le Planificateur à maintenir seulement une commande simple contrôlant un sous-système à la fois

```
  // Called when the command is initially scheduled.
  @Override
  public void initialize() {
    m_launcher.setLaunchWheel(kLauncherSpeed);
    m_launcher.setFeedWheel(kLaunchFeederSpeed);
  }
```

Viennent ensuite les méthodes de « cycle » de la commande. Ce sont les méthodes qui sont appelées lorsque la commande est planifiée (initialize), pendant qu'elle est en cours d'exécution (execute, isFinished) et quand elle est terminée ou interrompue (end). Ces méthodes sont tronquées dans la classe de commande de base et sont généralement remplacées par des classes enfants (ceci est indiqué par l'annotation @Override au-dessus de la méthode). Pour la classe LaunchNote, nous définissons une vitesse pour les roues lorsque la commande est initialisée. Nous pouvons faire ainsi parce que la vitesse souhaitée ne change pas pendant l'exécution de la commande. Si vous vouliez

modifier la vitesse pendant l'exécution de la commande (en fonction d'une entrée de joystick par exemple), le réglage de cette vitesse dans execute serait plus approprié.

```
// Called every time the scheduler runs while the command is scheduled.
@Override
public void execute() {

}
```

Étant donné que notre commande LaunchNote utilise une seule valeur de vitesse ajustée pendant initialize(), nous n'avons rien à faire ici dans la méthode execute. Cette méthode est appelée chaque exécution du Planificateur (généralement toutes les 20ms) pendant que la commande est en cours d'exécution.

```
// Returns true when the command should end.
@Override
public boolean isFinished() {
  return false;
}
```

La méthode isFinished() est également appelée à chaque exécution du Planificateur, après execute(), pour vérifier si la commande a fini de s'exécuter. Dans notre cas, nous voulons que les roues continuent de fonctionner tant que l'opérateur maintient le bouton, nous retournons donc toujours False pour indiquer que la commande n'est pas terminée. Le Planificateur gérera l'annulation de la commande lorsque l'opérateur relâchera le bouton (voir la section suivante sur RobotContainer).

```
// Called once the command ends or is interrupted.
@Override
public void end(boolean interrupted) {
  m_launcher.stop();
}
```

La commande end() est appelée quand la commande est retirée du Planificateur. Vous devriez effectuer tout « nettoyage » nécessaire sur votre mécanisme ici. Souvent, mais pas toujours, vous voudrez arrêter le mécanisme dans cette méthode, comme c'est notre cas ici. Le Planificateur passe une valeur booléenne pour indiquer si la commande a été interrompue ou non (non interrompue = terminée par elle-même en retournant True depuis isFinished()). Dans notre cas, nous ne nous soucions pas de savoir si la commande a été interrompue ou non, nous voulons arrêter les roues lorsque la commande se termine.

5.2.3 PrepareLaunch

La commande PrepareLaunch est le deuxième de deux exemples dans ce projet d'une commande en tant que classe indépendante définie dans son propre fichier. La commande PrepareLaunch ne fait

tourner que la roue extérieure du lanceur pour lui permettre d'atteindre la vitesse requise avant le tir. Le code de commande PrepareLaunch est presque identique à LaunchNote, à deux exceptions près.

```
// Called when the command is initially scheduled.
@Override
public void initialize() {
  m_launcher.setLaunchWheel(kLauncherSpeed);
}
```

Dans la méthode initialize(), la vitesse de la roue de tir (roue extérieure) est définie, mais la roue d'admission n'est pas affectée, de sorte que la pièce de jeu n'est pas encore introduite dans la roue de tir.

```
// Called once the command ends or is interrupted.
@Override
public void end(boolean interrupted) {
}
```

Dans la méthode end() nous n'arrêtons pas les roues parce que nous avons besoin que la roue de tir continue à fonctionner dans le cadre de la séquence de tir. Cela signifie que nous devons gérer l'arrêt de la roue si la séquence est interrompue lors de l'exécution de cette commande; ce que nous couvrirons dans la section 5.5 RobotContainer ci-dessous.

5.3 Constantes

Cette classe contient des constantes nommées utilisées ailleurs dans le code. Des sous-classes sont utilisées pour organiser les constantes en groupes distincts, dans notre cas par sous-système. Le fichier contient des commentaires pour indiquer ce que les constantes représentent.

5.4 Main et Robot

Ces deux fichiers sont identiques au modèle par défaut orienté sur les commandes. Vous trouverez une description des éléments dans la classe Robot dans Organisation d'un projet orienté commandes. La classe Main n'est généralement pas modifiée pour la programmation des robots FRC, quel que soit le modèle.

5.5 RobotContainer

La classe RobotContainer est l'endroit où les instances des sous-systèmes et contrôleurs du robot sont déclarées et où les commandes par défaut et les mappages des boutons vers les commandes sont définis.

```
public class RobotContainer {
  // The robot's subsystems are defined here.
  private final PWMDrivetrain m_drivetrain = new PWMDrivetrain();
  //private final CANDrivetrain m_drivetrain = new CANDrivetrain();
  private final PWMLauncher m_launcher = new PWMLauncher();
  //private final CANLauncher m_launcher = new CANLauncher();

  /*The gamepad provided in the KOP shows up like an XBox controller if the mode switch is set to X mode using the
  * switch on the top.*/
  private final CommandXboxController m_driverController =
      new CommandXboxController(OperatorConstants.kDriverControllerPort);
  private final CommandXboxController m_operatorController =
      new CommandXboxController(OperatorConstants.kOperatorControllerPort);
```

La première section définit les variables membres de la classe. Pour RobotContainer, cela inclut généralement tous vos sous-systèmes et périphériques de contrôle. Ce code utilise CommandXboxController pour représenter les manettes de jeu car il contient un certain nombre de méthodes d'assistance qui facilitent grandement la relation des commandes aux boutons.

```
/** The container for the robot. Contains subsystems, OI devices, and commands. */
public RobotContainer() {
  // Configure the trigger bindings
  configureBindings();
}
```

Le constructeur contient un seul appel à configureBindings() que nous couvrirons ci-dessous. Cette méthode est utilisée pour configurer des liaisons de boutons et des commandes par défaut. Ce constructeur a rarement besoin d'être modifié.

```
private void configureBindings() {
  // Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(
      new RunCommand(
          () ->
              m_drivetrain.arcadeDrive(
                  -m_driverController.getLeftY(), -m_driverController.getRightX()),
          m_drivetrain));
```

La méthode configureBindings() est l'endroit où nous plaçons tout le code qui indique aux commandes quand s'exécuter. La première section ici est pour la plateforme pilotable. Nous voulons qu'une commande s'exécute sur notre plateforme pilotable pour nous permettre de piloter le robot avec des joysticks chaque fois que nous n'avons pas une autre commande utilisant la plateforme pilotable (comme la commande exampleAuto). Pour ce faire, nous utilisons la méthode setDefaultCommand() du sous-système. Cela définit la commande qui s'exécutera chaque fois que le Planificateur ne voit rien d'autre s'exécuter sur ce sous-système.

Pour configurer la commande, nous utilisons une définition de commande inline à l'aide de la classe RunCommand. La classe RunCommand est utilisée pour transformer un appel de méthode unique en commande. La méthode que nous transmettons à RunCommand est insérée dans la section execute() du cycle des commandes discuté dans la section LaunchNote, ce qui signifie qu'elle sera appelée à

plusieurs reprises pendant que la commande est planifiée. La méthode d'exécution est à nouveau capturée à l'aide d'une Expressions Lambda en Java.

Dans ce code, la méthode que nous voulons appeler est la méthode arcadeDrive du sous-système de la plateforme pilotable. Pour le mouvement avant / arrière, nous passons dans la valeur de l'axe Y (vertical) du joystick gauche du contrôleur, mais nous le renversons. C'est parce que les joysticks définissent généralement la poussée vers l'avant comme négatif et de tirer le joystick vers vous comme positif (le résultat de l'utilisation originale dans des simulateurs de vol). Nous voulons pousser le joystick pour faire avancer le robot, donc nous inversons la valeur. Similairement pour la valeur de virage où nous inversons l'axe X (horizontal) du joystick droit du contrôleur. Le joystick considère pousser vers la droite comme une valeur positive, mais les classes WPILib considèrent la rotation dans le sens des aiguilles d'une montre (ce à quoi on s'attendrait en poussant le joystick à droite) comme négative.

```
/*Create an inline sequence to run when the operator presses and holds the A (green) button. Run the PrepareLaunch
* command for .25 seconds and the run the LaunchNote command */
m_operatorController.a().whileTrue(
  new PrepareLaunch(m_launcher)
  .withTimeout(seconds: .25)
  .andThen(new LaunchNote(m_launcher))
  .handleInterrupt(() -> m_launcher.stop())
);
```

Ensuite, nous mettons en place un groupe de commandes inline pour exécuter une séquence de tir pendant que l'opérateur maintient le bouton A du contrôleur pressé. Nous utilisons d'abord la méthode d'assistance a() sur le contrôleur pour obtenir un déclencheur (Trigger), puis utilisons la méthode whileTrue() pour exécuter la commande pendant que le bouton est maintenu pressé, et en l'annulant lorsque le bouton est libéré. Vous pouvez voir d'autres options disponibles pour les méthodes Trigger sur la page Trigger Javadoc (voa) ou sur la page WPILib Liaison de commandes à des déclencheurs.

La première commande de notre séquence est PrepareLaunch(). Nous utilisons ensuite le décorateur withTimeout() pour que cette commande soit exécutée pendant une durée fixe avant de se terminer. Le décorateur andThen() est utilisé pour exécuter la commande LaunchNote après l'expiration du délai d'attente. Enfin, le décorateur handleInterrupt() est utilisé pour fournir une méthode à exécuter si la commande est interrompue ; ceci est nécessaire pour s'assurer que les roues s'arrêtent si l'opérateur lâche le bouton pendant que la commande PrepareLaunch est en cours d'exécution car la méthode end() de cette commande n'arrêtera pas les roues par elle-même.

6 Apporter des modifications

Cette section détaille quelques modifications possibles que vous voudrez peut-être apporter au code KitBot et fournit quelques références sur la façon de réaliser ces modifications.

6.1 Modification des boutons d'action

L'une des modifications les plus faciles à apporter au code du robot orienté commandes est de changer quels boutons ou quels comportements de bouton contrôlent une commande. Les commandes utilisées dans le KitBot 2024 ne se terminent pas (isFinished renvoie toujours false) elles ne doivent donc généralement être utilisées qu'avec le comportement whileHeld(), mais changer les boutons auxquels elles correspondent peut être fait très simplement.

Les mappages de boutons dans l'exemple de code sont effectués vers la fin de la méthode configureBindings() dans le fichier RobotContainer. Les liaisons utilisées pour ce projet sont effectuées à l'aide des méthodes d'assistance de la classe CommandXboxController. Ces méthodes d'assistance existent pour chaque bouton du contrôleur et renvoient un objet Trigger qui peut ensuite être utilisé pour spécifier un comportement pour la liaison.

Comme prévu, le code connecte le bouton **a** à la séquence de tir et la gâchette avant gauche **leftBumper** à l'admission d'une pièce de jeu. Pour les changer, il suffit de changer les méthodes d'assistance a() ou leftBumper(), pour la méthode d'un des autres boutons ! Vous pouvez voir toutes les options disponibles en parcourant le CommandXboxController Javadoc (voa) pour des méthodes qui ne prennent aucun paramètre et retournent un objet Trigger.

Par exemple, pour remplacer la commande d'admission de la gâchette avant gauche par le bouton x, remplacez simplement le leftBumper() par x().

```
//Before
m_operatorController.leftBumper().whileTrue(m_launcher.getIntakeCommand());
//After
m_operatorController.x().whileTrue(m_launcher.getIntakeCommand());
```

6.2 Modification du comportement des joysticks de pilotage

Un autre changement facile à faire est de modifier quels joysticks du contrôleur sont utilisés pour le pilotage du robot et comment. Le code fourni effectue ce mappage lors de la configuration de la commande par défaut de la plateforme pilotable en haut de configureBindings() dans RobotContainer.

L'exemple de code utilise l'axe Y du joystick gauche pour avancer et reculer et l'axe X du joystick droit pour tourner. Ceux-ci peuvent facilement être échangés ou déplacés sur un seul joystick ! Pour examiner les options disponibles, recherchez les méthodes qui renvoient un **double** dans CommandXboxController Javadoc (voa) Pour effectuer ce type de modification, localisez l'appel de méthode que vous souhaitez modifier, tel que getLeftY(), et remplacez-le par la nouvelle méthode souhaitée, telle que getRightY().

Exemple : changer la propulsion avant-arrière sur l'axe Y du joystick droit et laisser la rotation sur l'axe X du joystick droit.

```
private void configureBindings() {
  //Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(new RunCommand(() -> m_drivetrain.arcadeDrive(
    -m_driverController.getRightY(), -m_driverController.getRightX()),
    m_drivetrain
  ));
```

Vous pouvez également modifier les valeurs de joystick. Une modification courante consiste à porter au cube les valeurs. Cela préserve le signe de la valeur (positif reste postif, négatif reste négatif) et la valeur maximale ne réduit pas la vitesse maximale du robot tout en fournissant moins de sensibilité pour de faibles entrées, permettant potentiellement un contrôle plus précis à basse vitesse. Pour effectuer ce type de modification, vous pouvez appliquer la modification de la valeur du joystick là où elle est obtenue.

Exemple : modification uniquement de l'axe de rotation (x) à mettre au cube :

```
private void configureBindings() {
  //Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(new RunCommand(() -> m_drivetrain.arcadeDrive(
    -m_driverController.getLeftY(), -Math.pow(m_driverController.getRightX(), 3)),
    m_drivetrain
  ));
```

Une autre modification courante consiste à réduire par défaut les valeurs, mais à autoriser la valeur maximale si vous appuyez sur un bouton (mode turbo). Ce type de modification peut également être effectué au point de lecture, bien qu'à mesure que la complexité augmente, vous voudrez peut-être passer d'une définition de commande inline à un type différent où vous pourrez définir le comportement de commande plus clairement.

Exemple : Mise à l'échelle de la vitesse avant-arrière de 50 % à moins que la gâchette avant droite ne soit enfoncée.

```
private void configureBindings() {
  //Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(new RunCommand(() -> m_drivetrain.arcadeDrive(
    -m_driverController.getLeftY()*(m_driverController.getHID().getRightBumper()?1:.5), -Math.pow(m_driverController.getRightX(), 3)),
    m_drivetrain
  ));
```

Cet exemple utilise deux choses spécifiques qui méritent d'être expliquées. La première est qu'il utilise la méthode getHID() sur le contrôleur du pilote. Cela renvoie l'objet XboxController (voa) que CommandXboxController encapsule. Cela nous donne accès à des méthodes différentes (voa) de celles de CommandXboxController, dans ce cas getRightBumper() qui nous permet d'obtenir la valeur booléenne du bouton plutôt qu'un objet Trigger associé au bouton.

L'autre chose que cet exemple utilise est l'opérateur ?, appelé opérateur ternaire (voa). Cet opérateur nous permet d'écrire une simple déclaration « if » de manière très compacte, si la gâchette avant droite est enfoncée, nous multiplions par 1, sinon, par 0,5.

6.3 Modification du type de propulsion

Le dernier changement probable que nous couvrirons est le passage de la propulsion Arcade Drive à Tank Drive. Contrairement à la propulsion Arcade qui mappe un axe de joystick à la rotation et un axe de joystick au déplacement avant / arrière, la propulsion de style Tank mappe un axe de joystick (généralement l'axe Y) de chaque côté d'une plateforme pilotable différentielle. Pour effectuer ce changement, vous devrez aller au-delà de RobotContainer car les sous-systèmes de transmission fournis ne proposent pas une méthode de propulsion de style Tank. Dans le sous-système de plateforme pilotable approprié (PWMDrivetrain ou CANDrivetrain), faites une nouvelle méthode appelée tankDrive(). Cette méthode devrait ressembler beaucoup à la méthode arcadeDrive. Ensuite, modifiez le mappage des commandes par défaut dans RobotContainer pour utiliser cette nouvelle méthode avec l'axe de joystick approprié.

Exemple:

```
public void tankDrive(double leftSpeed, double rightSpeed)
{
  m_drivetrain.tankDrive(leftSpeed, rightSpeed);
}
```

```
private void configureBindings() {
  //Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(new RunCommand(() -> m_drivetrain.tankDrive(
    -m_driverController.getLeftY(), -m_driverController.getRightY()),
    m_drivetrain
  ));
```

6.4 Développer des routines autonomes

Le code fourni contient un mode autonome très basique qui roule vers l'arrière à 1/2 de puissance pendant 1 seconde. Des modes autonomes supplémentaires peuvent être développés, soit en ajoutant des méthodes supplémentaires dans le fichier Autos (voir l'exemple de projet Hatchbot Inlined (voa) pour un exemple avec un code autonome plus complexe), soit en créant des fichiers séparés pour chaque routine autonome (voir Hatchbot Traditional (voa) pour un exemple de cette approche).

Il est courant (mais certainement pas nécessaire !) d'avoir plusieurs routines autonomes que vous voudrez peut-être exécuter en fonction de différents emplacements de départ ou différentes stratégies. Si vous poursuivez dans cette voie, le moyen le plus courant de sélection pour chaque match est de choisir entre elles à l'aide d'un SendableChooser sur le tableau de bord.