Organiser un programme à l'aide de commandes et de sous-systèmes

Étienne Beaulac

Colloque de la communauté *FIRST* au Québec
Institut secondaire Keranna, Trois-Rivières, le 26 octobre 2019

Vue d'ensemble

Vue d'ensemble

Vue d'ensemble

La programmation orientée commande
Les problèmes du fichier unique

- En Java, un programme de base est contenu dans un seul fichier : `Robot.java`.
- Les composantes du robot, les manettes de jeu et toutes les actions sont au même endroit.

La programmation orientée commande
Les problèmes du fichier unique

```
public class Robot extends TimedRobot {

    private VictorSP moteurGauche = new VictorSP(0);
    private VictorSP moteurDroit = new VictorSP(1);
    private DifferentialDrive drive = new DifferentialDrive(moteurGauche,
    ↪  moteurDroit);
    private Joystick stick = new Joystick(0);

    @Override
    public void teleopPeriodic() {
        drive.arcadeDrive(-stick.getY(), stick.getX());
    }

}
```

La programmation orientée commande
Les problèmes du fichier unique

Lorsque la complexité du robot augmente, cela peut causer différents problèmes :

- des actions utilisant les mêmes actuateurs peuvent entrer en conflit ;
- il est difficile de maintenir un tel programme ;
- travailler simultanément sur le même fichier est ardu.

La programmation orientée commande
Les avantages de la POC

La solution : la programmation orientée commande, ou POC (*Command-based programming*).

> *« WPILib supports a method of writing programs called "Command-based programming". Command based programming is a design pattern to help you organize your robot programs. (...) Command based programming aims to make the robot program much simpler than using some less structured technique*[1]*. »*

1. wpilib.screenstepslive.com/s/currentCS/m/java/l/599732-what-is-command-based-programming

La programmation orientée commande
Les avantages de la POC

La POC consiste à séparer dans des fichiers différents chaque sous-système et commande du robot.
Cela comporte plusieurs avantages :

- il est beaucoup plus facile de se repérer dans le programme ;
- chaque personne peut travailler dans des fichiers différents ;
- les fichiers de sous-systèmes généraux (ex. base pilotable) peuvent être réutilisés d'année en année ;
- les comportements plus complexes sont plus faciles à implémenter.

La programmation orientée commande
Changements pour 2020

- Plusieurs modifications et ajouts importants au *Command framework* sont prévus pour la saison 2020.
- Ces changements sont présentement en période de test bêta.
- Il peut donc y avoir des différences entre le contenu de la présentation et la version finale de WPILib, disponible en janvier 2020.

Bêta 2020

Les exemples qui suivent sont basés sur la documentation de la version de 2020 disponible en octobre 2019.

La programmation orientée commande
Changements pour 2020

Entre autres, ces changements comprennent :

- `Subsystem` et `Command` deviennent des interfaces au lieu d'être des classes abstraites. Les classes abstraites `SubsystemBase` et `CommandBase` ont été ajoutées pour les remplacer.
- La classe `Scheduler` a été renommée `CommandScheduler`.
- La classe `CommandGroup` (ayant les méthodes `addSequential(Command)` et `addParallel(Command)`) n'existe plus. Des alternatives plus spécifiques ont été ajoutées : `SequentialCommandGroup`, `ParallelCommandGroup`, `ParallelRaceGroup` et `ParallelDeadlineGroup`.

Liste complète des changements

Pour débuter

Installer Java, WPILib et Visual Studio Code

Les instructions pour Windows, macOS et Linux sont disponibles ici : WPILib Installation Guide.

Pour débuter
Créer un projet

1. Dans FRC VS Code, cliquez sur le bouton WPILib.
2. Sélectionnez **WPILib : Create new project** .

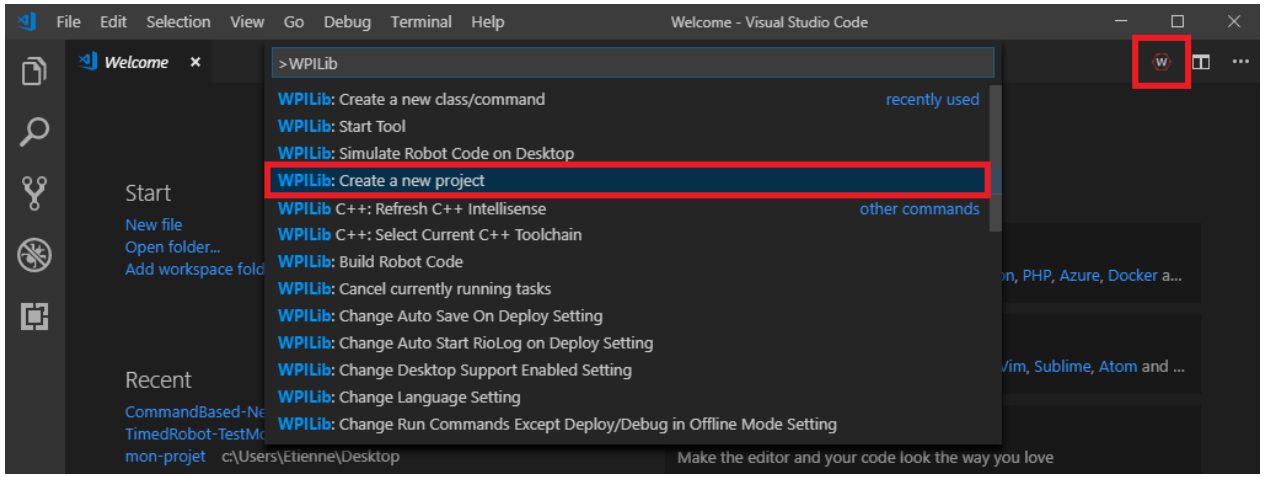

FIGURE 1 – Créer un projet

Pour débuter

Créer un projet

1. Comme options, sélectionnez `Template` , `Java` , puis `Command Robot` .
2. Sélectionnez le dossier de sauvegarde de votre projet.
3. Saisissez le nom de votre projet (sans espace).
4. Saisissez le numéro de l'équipe.
5. Cliquez sur `Generate project` .

Bêta 2020

Avant la mise à jour de 2020, le projet généré sera conforme aux standards 2019. En attendant, je vous invite à utiliser ce modèle de projet : Modèle de projet 2020.

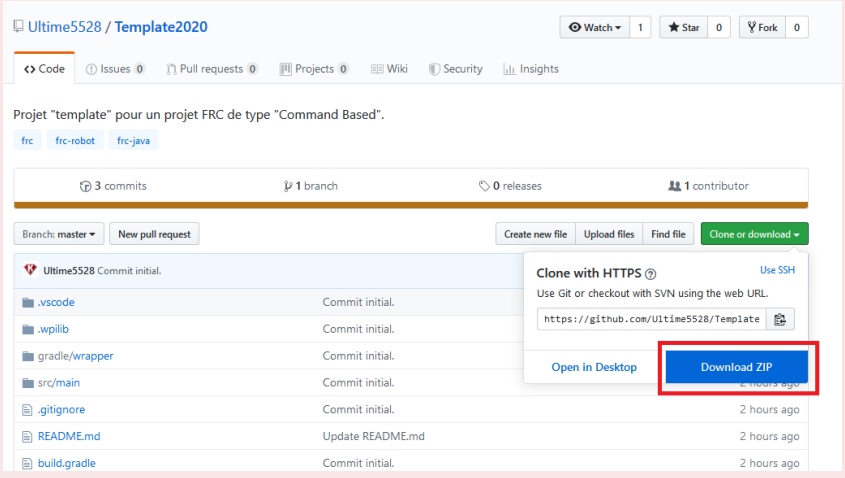

FIGURE 2 – Télécharger le modèle de projet 2020

Créer un sous-système

Qu'est-ce qu'un sous-système ?

Un **sous-système** est un regroupement logique de différentes composantes du robot. Par exemple :

- les deux moteurs qui font rouler le robot peuvent former le sous-système `BasePilotable` ;
- le moteur relié au shooter et l'encodeur qui mesure sa vitesse peuvent former le sous-système `Shooter` ;
- les moteurs qui servent à prendre un ballon et la *limit switch* qui détecte si le robot en possède un peuvent former le sous-système `Intake`.

Créer un sous-système
Qu'est-ce qu'un sous-système ?

- Chaque composante du robot doit être contenue par un et un seul sous-système.
- C'est le rôle du programmeur d'identifier les sous-systèmes nécessaires.

Attention ! Un sous-système ne peut exécuter qu'une seule commande à la fois.

Créer un sous-système
Ajouter un fichier

1. Faites un clic droit sur le dossier `subsystems` .
2. Sélectionnez `Create a new class/command` puis `Subsystem` .
3. Saisissez le nom de votre sous-système (débutant par une majuscule, sans espace).

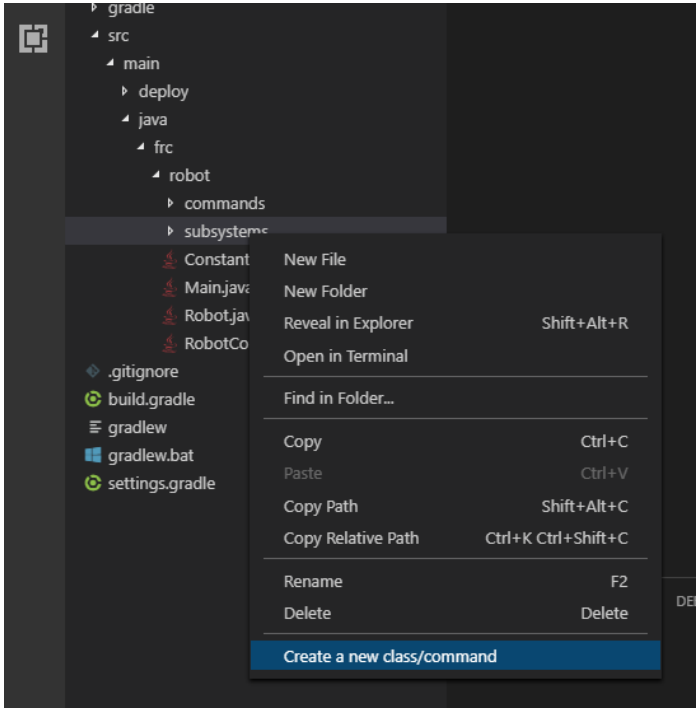

FIGURE 3 – Ajout d'un sous-système.

Créer un sous-système
Ajouter un fichier

Bêta 2020

Le sous-système généré n'est pas compatible avec les nouveautés de 2020. En attendant la mise à jour officielle, je vous suggère de dupliquer le sous-système fourni dans le Modèle de projet 2020.

Créer un sous-système
Ajouter des composantes

1. Les actuateurs et capteurs de notre sous-système doivent être déclarés au début de la classe, comme attributs privés (`private`).
2. On les instancie (`new`) dans le constructeur.
3. On ajoute chaque composante au sous-système avec `addChild()`, afin qu'ils soient tous affichés dans *Shuffleboard*.

Créer un sous-système
Ajouter des composantes

```
public class BasePilotable extends SubsystemBase {
    private VictorSP moteurGauche, moteurDroit;
    private DifferentialDrive drive;

    private Encoder encoderGauche, encoderDroit;
    private ADXRS450_Gyro gyro;

    public BasePilotable() {
        moteurGauche = new VictorSP(0);
        addChild("Moteur gauche", moteurGauche);

        moteurDroit = new VictorSP(1);
        addChild("Moteur droit", moteurDroit);

        drive = new DifferentialDrive(moteurGauche, moteurDroit);
        addChild("Drive", drive);

        encoderGauche = new Encoder(0, 1);
        addChild("Encoder gauche", encoderGauche);

        encoderDroit = new Encoder(2, 3);
        addChild("Encoder droit", encoderDroit);

        gyro = new ADXRS450_Gyro();
        addChild("Gyro", gyro);
    }
}
```

Créer un sous-système
Ajouter des méthodes

- Puisque les composantes internes du sous-système sont privées, il faut lui ajouter des méthodes publiques.
- Généralement, les méthodes sont séparées en deux catégories :
  - action : la méthode actionne une ou des composantes du sous-système. Elle peut recevoir un ou des paramètres (ex. la vitesse).
  - accès aux données : la méthode retourne une donnée sur l'état du sous-système (ex. l'angle du gyro). Elle ne reçoit habituellement aucun paramètre.

Créer un sous-système
Ajouter des méthodes

```java
public class BasePilotable extends SubsystemBase {

    // Composantes...

    public BasePilotable() {
        // Constructeur...
    }

    public void drive(double forward, double turn) {
        drive.arcadeDrive(forward, turn);
    }

    public void stop() {
        drive.arcadeDrive(0, 0);
    }

    public double getAngle() {
        return gyro.getAngle();
    }

    public double getDistanceRoueGauche() {
        return encoderGauche.getDistance();
    }

    // ...

}
```

Créer une commande simple

Qu'est-ce qu'une commande ?

Une **commande** est une action que peut effectuer le robot à l'aide d'un ou plusieurs sous-systèmes. Par exemple :

- `AvancerLigneDroite` utilise `BasePilotable` ;
- `PrendreBallon` utilise `Shooter` ;
- `AvancerEtLancerBallon` utilise `BasePilotable` et `Shooter`.

Un sous-système ne peut être utilisé que par une commande à la fois. Par exemple, `PrendreBallon` et `LancerBallon` ne peuvent pas s'exécuter en même temps, car elles utilisent toutes les deux le `Shooter`.
On doit garder en tête cette contrainte en tout temps.

Créer une commande simple

Ajouter un fichier

1. Faites un clic droit sur le dossier `commands` .
2. Sélectionnez `Create a new class/command` puis `Command` .
3. Saisissez le nom de votre commande (débutant par une majuscule, sans espace).

Bêta 2020

La commande générée n'est pas compatible avec les nouveautés de 2020. En attendant la mise à jour officielle, je vous suggère de dupliquer la commande fournie dans le Modèle de projet 2020.

Créer une commande simple

Compléter les méthodes

Une commande contient un constructeur et 4 méthodes que l'on doit compléter.

```
public class ExampleCommand extends CommandBase {

    public ExampleCommand() { ... }

    @Override
    public void initialize() { ... }

    @Override
    public void execute() { ... }

    @Override
    public boolean isFinished() { ... }

    @Override
    public void end(boolean interrupted) { ... }

}
```

Créer une commande simple
Compléter les méthodes

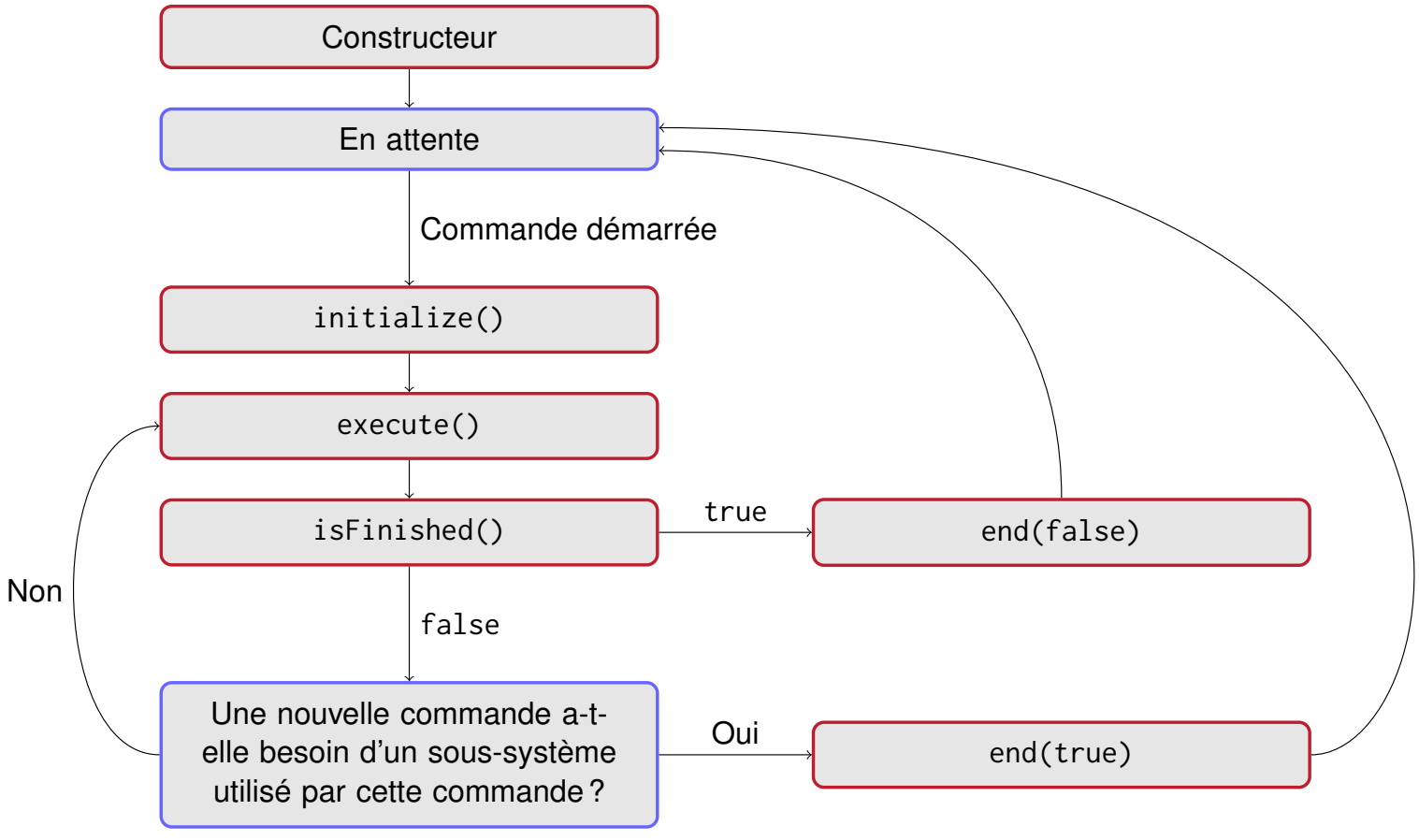

FIGURE 4 – Le cycle de vie d'une commande.

Créer une commande simple
Compléter les méthodes

1. Constructeur
   - Le constructeur est appelé lors de la création (`new`) de la commande.
   - Au minimum, il doit avoir, comme paramètres, le ou les sous-systèmes utilisés par la commande.
   - Ces sous-systèmes doivent être stockés comme attributs et ajoutés dans les *requirements* avec `addRequirements()`.
   - D'autres paramètres peuvent être ajoutés (ex. jusqu'à quelle distance avancer, le *joystick* pour piloter).

Créer une commande simple

Compléter les méthodes

```
public class Piloter extends CommandBase {

    private BasePilotable basePilotable;
    private Joystick stick;

    public Piloter(BasePilotable _basePilotable, Joystick _stick) {

        basePilotable = _basePilotable;
        stick = _stick;

        addRequirements(basePilotable);

    }

    // ...

}
```

Créer une commande simple
Compléter les méthodes

2 `initialize()`

- Cette méthode est appelée une fois, au début de l'exécution de la commande.
- Par exemple, on peut y réinitialiser les encodeurs ou le gyroscope.

```
@Override
public void initialize() {
    basePilotable.resetGyro();
    basePilotable.resetEncoders();
}
```

Créer une commande simple

Compléter les méthodes

3 `execute()`

- Il s'agit du coeur de la commande.
- Cette méthode est appelée en boucle, tant que la commande continue de s'exécuter.

```
@Override
public void execute() {
    basePilotable.drive(-1 * stick.getY(), stick.getX());
}
```

Créer une commande simple
Compléter les méthodes

4 `isFinished()`

- Cette méthode doit retourner **vrai** (`true`) ou **faux** (`false`).
- Le système appelle cette méthode après chaque `execute()` afin de vérifier si la commande doit s'arrêter.
- Si la méthode retourne `false` , la commande poursuit son exécution.
- Si elle retourne `true` , la commande s'arrêtera.

```java
@Override
public boolean isFinished() {
    // La commande Piloter ne se termine jamais par elle-même.
    return false;
}
```

Créer une commande simple
Compléter les méthodes

5 `end()`

- Cette méthode s'exécute une fois, à la fin de l'exécution de la commande.
- On peut, par exemple, arrêter les moteurs qui ont été utilisés.
- Cette méthode reçoit une variable booléenne :
  - Si `true`, la commande s'est arrêtée à la suite d'une interruption causée par une autre commande.
  - Si `false`, la commande s'est arrêtée d'elle-même (`isFinished()` a retourné `true`).

```
@Override
public void end(boolean interrupted) {
    basePilotable.stop();
}
```

Groupe de commandes

Qu'est-ce qu'un groupe de commandes ?

- Les commandes simples peuvent être regroupées ensemble pour former des séquences plus complexes.
- Elles sont idéales pour créer un mode autonome et automatiser certaines actions du pilote.
- Il en existe quatre types : `SequentialCommandGroup`, `ParallelCommandGroup`, `ParallelRaceGroup` et `ParallelDeadlineGroup`.

Groupe de commandes
Ajouter un fichier

1. Faites un clic droit sur le dossier `commands` .
2. Sélectionnez `Create a new class/command` puis `Command Group` .
3. Saisissez le nom de votre groupe de commandes (débutant par une majuscule, sans espace).

Bêta 2020

Le groupe de commandes généré n'est pas compatible avec les nouveautés de 2020. En attendant la mise à jour officielle, je vous suggère de dupliquer le groupe de commandes fourni dans le Modèle de projet 2020.

Groupe de commandes
SequentialCommandGroup

Les commandes s'exécutent séquentiellement, l'une après l'autre. On ajoute les commandes au groupe avec `super()` ou `addCommands()`.

```
public class ModeAutonome extends SequentialCommandGroup {

    public ModeAutonome(BasePilotable basePilotable) {
        super(
            new Avancer(2.5, basePilotable),
            new Tourner(-65.5, basePilotable),
            new WaitCommand(1.5),
            new Avancer(1.2, basePilotable)
        );
    }

}
```

Groupe de commandes
ParallelCommandGroup

Toutes les commandes s'exécutent en même temps. Le groupe se termine lorsque toutes les sous-commandes sont terminées.

```
public class AvancerMonterElevateur extends ParallelCommandGroup {

    public AvancerMonterElevateur(BasePilotable basePilotable, Elevateur
    ↪  elevateur) {
        super(
            new Avancer(1.5, basePilotable),
            new MonterElevateur(elevateur)
        );
    }

}
```

Groupe de commandes
ParallelRaceGroup

Toutes les commandes s'exécutent en même temps. Le groupe se termine dès qu'une des sous-commandes se termine.

```
public class PrendreCubeAutonome extends ParallelRaceGroup {

    public PrendreCubeAutonome(BasePilotable basePilotable, Intake intake) {
        super(
            new Avancer(1.5, basePilotable),
            new PrendreCube(intake),
            new WaitCommand(5.0) // Commande vide qui termine après 5 sec.
        );
    }

}
```

Groupe de commandes
ParallelDeadlineGroup

Toutes les commandes s'exécutent en même temps. Le groupe se termine dès qu'une commande spécifique, le *deadline*, se termine.

```
public class AvancerPrendreBallon extends ParallelDeadlineGroup {

    public AvancerPrendreBallon(BasePilotable basePilotable, Intake intake,
    ↪  Elevateur elevateur) {
        super(
            // La première commande est le deadline
            // Le timeout ajoute une condition de fin à la commande
            new Avancer(2.0, basePilotable).withTimeout(3.5),
            new PrendreBallon(intake),
            new MaintienElevateur(elevateur)
        );
    }

}
```

Groupe de commandes
Combiner les types de groupes

Puisqu'un groupe de commandes est lui-même une commande, on peut combiner les différentes types de groupes. À cet effet, ces raccourcis sont disponibles : `sequential()`, `parallel()`, `race()` et `deadline()`.

```
public class Pickup extends SequentialCommandGroup {

    public Pickup(Claw claw, Wrist wrist, Elevator elevator) {
        addCommands(
            new CloseClaw(claw),
            parallel(
                new SetWristSetpoint(-45, wrist),
                new SetElevatorSetpoint(0.25, elevator)));
    }

}
```

RobotContainer

Qu'est-ce que le RobotContainer ?

Cette classe crée le lien entre les sous-systèmes, les commandes et les manettes de jeu. En effet, elle permet entre autres de :

- déclarer les sous-systèmes et les manettes de jeu ;
- lier les commandes aux boutons de chaque manette ;
- assigner des commandes par défaut.

RobotContainer

Ajout des sous-systèmes et des manettes

```java
public class RobotContainer {

    // Déclaration de tous les sous-systèmes
    private final BasePilotable basePilotable = new BasePilotable();
    private final Shooter shooter = new Shooter();

    // Manettes de jeu
    private Joystick stick = new Joystick(0);
    private XboxController xboxController = new XboxController(1);

    // ...

}
```

RobotContainer
Lier les boutons à des commandes

Dans `configureButtonBindings()`, On utilise la classe `JoystickButton` pour créer des boutons. On appelle ensuite une des méthodes suivantes sur les boutons pour les lier à des commandes.

- `whenPressed()` : la commande est démarrée à chaque fois que le bouton est appuyé.
- `whenHeld()` : la commande est démarrée lorsque le bouton est appuyé, puis elle est interrompue lorsque le bouton est relâché.
- `whileHeld()` : la commande est démarrée continuellement tant que le bouton est appuyé.
- `toggleWhenPressed()` : la commande est démarrée lorsque le bouton est appuyé. Elle est interrompue si on appuie à nouveau sur ce bouton.

RobotContainer

Lier les boutons à des commandes

```
private void configureButtonBindings() {

    new JoystickButton(stick, 1).whenPressed(new LancerBallon(shooter));
    new JoystickButton(stick, 2).toggleWhenPressed(new
    ↪  PrendreBallon(shooter));

}
```

RobotContainer
Commandes par défaut

La commande par défaut d'un sous-système est démarrée automatiquement lorsque celui-ci est libre. Pour ce faire, on appelle la méthode `setDefaultCommand()` sur le sous-système dans le constructeur de `RobotContainer`.

```
public class RobotContainer {

    private final BasePilotable basePilotable = new BasePilotable();
    private final Elevateur elevateur = new Elevateur();

    public RobotContainer() {
        // Commandes par défaut
        basePilotable.setDefaultCommand(new Piloter());
        elevateur.setDefaultCommand(new MaintienElevateur());

        configureButtonBindings();
    }
}
```

Fichier Constants

Il est recommandé de conserver dans un fichier à part les constantes utilisées dans le programme.
Par exemple, il est très utile de garder au même endroit les ports des composantes.

Fichier Constants

```java
public class Constants {

    public static class Ports {
        // PWM
        public static final int BASE_PILOTABLE_MOTEUR_GAUCHE = 0;
        public static final int BASE_PILOTABLE_MOTEUR_DROIT = 1;
        public static final int SHOOTER_MOTEUR = 2;

        // DIO
        public static final int BASE_PILOTABLE_ENCODER_GAUCHE_A = 0;
        public static final int BASE_PILOTABLE_ENCODER_GAUCHE_B = 1;
        public static final int BASE_PILOTABLE_ENCODER_DROIT_A = 2;
        public static final int BASE_PILOTABLE_ENCODER_DROIT_B = 3;
    }

    public static class BasePilotable {
        public static final double P = 0.5;
        public static final double I = 0.005;
        public static final double D = 0.0;
    }
}
```

Fichier Constants

```
public class BasePilotable extends SubsystemBase {

    private VictorSP moteurGauche = new
    ↪  VictorSP(Constants.Ports.BASE_PILOTABLE_MOTEUR_GAUCHE);
    private VictorSP moteurDroit = new
    ↪  VictorSP(Constants.Ports.BASE_PILOTABLE_MOTEUR_DROIT);

    // ...

}
```

Conseils pratiques

- Ctrl+Espace : VS Code permet l'autocomplétion. Vous éviterez ainsi des fautes de frappe sournoises et des importations manquantes.
- Git : l'outil idéal pour travailler en équipe et conserver des sauvegardes de son programme. Je vous recommande GitHub avec le logiciel GitKraken.
- chiefdelphi.com : ce forum FIRST est rempli d'informations utiles. Il permet d'être au courant des dernières nouveautés, de poser des questions, de lire les problèmes et les réussites d'autres équipes, etc.

Exemples

- De nombreux exemples sont disponibles dans ce répertoire Github.
- Entre autres, les exemples frisbeebot, gearsbot et hatchbottraditional ont été mis à jour et sont particulièrement intéressants.
- Nouvelle documentation
- Ancienne documentation

Questions

La présentation est disponible sur github.com/etiennebeaulac/Atelier-CommandBased et ultime5528.com/fr/ressources.

Courriel : etienne.beaulac@uqtr.ca