Day 26 CS570

Command Based Robot

Command Based Robot

From WPILIB

The command-based pattern is based around two core abstractions: commands, and subsystems.

Commands represent actions the robot can take. Commands run when scheduled, until they are interrupted or their end condition is met. Commands are very recursively composable: commands can be composed to accomplish more-complicated tasks. See Commands for more info.

Subsystems represent independently-controlled collections of robot hardware (such as motor controllers, sensors, pneumatic actuators, etc.) that operate together. Subsystems back the resource-management system of command-based: only one command can use a given subsystem at the same time. Subsystems allow users to "hide" the internal complexity of their actual hardware from the rest of their code - this both simplifies the rest of the robot code, and allows changes to the internal details of a subsystem's hardware without also changing the rest of the robot code.

Commands are run by the CommandScheduler, which polls triggers (such as buttons) for commands to schedule, preventing resource conflicts, and executing scheduled commands. The scheduler's run() method must be called; it is generally recommended to call it from the robotPeriodic() method of the Robot class, which is run at a default frequency of 50Hz (once every 20ms).

Multiple commands can run concurrently, as long as they do not require the same resources on the robot. Resource management is handled on a per-subsystem basis: commands specify which subsystems they interact with, and the scheduler will ensure that no more more than one command requiring a given subsystem is scheduled at a time. This ensures that, for example, users will not end up with two different pieces of code attempting to set the same motor controller to different output values.

Command Compositions

It is often desirable to build complex commands from simple pieces. This is achievable by creating a composition of commands. The command-based library provides several types of command compositions for teams to use, and users may write their own. As command compositions are commands themselves, they may be used in a recursive composition. That is to say - one can create a command compositions from multiple command compositions. This provides an extremely powerful way of building complex robot actions from simple components.

Building a Command ROMI

Starting code:

You can go here to get starting code for a Command ROMI

In this code you will find a copy of Robot that has a robot.py, a robotcontainer.py, and a drivetrain.py

https://github.com/mbardoeChoate/CommandROMI/tree/starting_config

Starting to make a command robot Let's start by making changes to `robot.py`.

Start by subclassing a different kind of robot.

```
import commands2

class MyRobot(commands2.TimedCommandRobot):
    """
    Our default robot class, pass it to wpilib.run

    Command v2 robots are encouraged to inherit from TimedCommandRobot, which
    has an implementation of robotPeriodic which runs the scheduler for you
    """
```

Now we will basically be doing the equivalent of routines everytime through the main loop of the robot. The CommandScheduler will need to know what commands and subsytems it is working with each step of the way.

To make this all work we are going to need to remove some more code in `robot.py`

```
# Starter Robot Code
import os

import wpilib
import commands2
#from wpilib import TimedRobot

from robotcontainer import RobotContainer


class Robot(commands2.TimedCommandRobot):

    def robotInit(self) -> None:
        self.container = RobotContainer()

    #def robotPeriodic(self) -> None:
    #    ...

    def teleopInit(self) -> None:
        ...

    def teleopPeriodic(self) -> None:
        #forward = self.container.controller.getRawAxis(0)
        #rotate = self.container.controller.getRawAxis(1)
        #self.container.drivetrain.arcadeDrive(rotate, forward)
        #print(f"Forward: {forward}, Rotate: {rotate}")
        pass

    def autonomousInit(self) -> None:
        #self.auto = self.container.get_autonomous()
        pass

    def autonomousPeriodic(self) -> None:
        #self.auto.run()
        pass

    def autonomousExit(self) -> None:
        #self.container.drivetrain.resetGyro()
        pass

    def disabledInit(self) -> None:
        pass


if __name__ == "__main__":
    os.environ["HALSIMWS_HOST"] = "10.0.0.2"
    os.environ["HALSIMWS_PORT"] = "3300"

    wpilib.run(Robot)
```

You can see I have made it mostly empty. We will bring some of those ideas back later. Let's focus now on teleoperated movement.

We are going to start by making a command that allows us to drive.

Start by making a folder called *commands*.

And in that folder make a file called `arcadedrive.py`.

```
# Copyright (c) FIRST and other WPILib contributors.
# Open Source Software; you can modify and/or share it under the terms of
# the WPILib BSD license file in the root directory of this project.

import typing
import commands2
from drivetrain import Drivetrain


class ArcadeDrive(commands2.CommandBase):
    def __init__(
        self,
        drive: Drivetrain,
        forward: typing.Callable[[], float],
        rotation: typing.Callable[[], float],
    ) -> None:
        """Creates a new ArcadeDrive. This command will drive your robot according to the speed supplier
        lambdas. This command does not terminate.

        :param drivetrain:  The drivetrain subsystem on which this command will run
        :param forward:     Callable supplier of forward/backward speed
        :param rotation:    Callable supplier of rotational speed
        """
        super().__init__()

        self.drive = drive
        self.forward = forward
        self.rotation = rotation

        self.addRequirements([self.drive])

    def initialize(self) -> None:
        pass
    def end(self, interrupted: bool) -> None:
        pass

    def isFinished(self) -> bool:
        return False
    def execute(self) -> None:
        self.drive.arcadeDrive(self.rotation(), self.forward())
```

With this class we will be able to drive the robot in teleop.

Make the following changes to the robotcontainer.py to get that done. Add the following imports:

```
import commands2
import commands2.button
from commands.arcadedrive import ArcadeDrive
```

Then add the following method

```
    def getArcadeDriveCommand(self) -> ArcadeDrive:
        """Use this to pass the teleop command to the main robot class.

        :returns: the command to run in teleop
        """
```

```
        return ArcadeDrive(
            self.drivetrain,
            lambda: self.controller.getRawAxis(0),
            lambda: self.controller.getRawAxis(1),
        )
```

lambda methods One structure that you see a lot in commands are **lambda** functions. Lambda functions are a way or creating simple functions very quickly without the need to using `def`. One can see from the testing examples that we did that this could be very useful. When we were creating mock version of various methods. The keyword `lambda` indicates that we are making function, most often to feed into somethings that expects a function.

Common lambda syntax:

```
lambda argument(s): expression
```

```
#Normal python function
def a_name(x):
    return x+x
#Lambda function
lambda x: x+x
```

Now we have a command that will drive the robot we just have to set it up so that it gets called by the scheduler.

Before we do that we need to let the drivetrain know that it is a Subsystem in a command framework. Go into the `drivetrain.py` and subclass the drivetrain as commands.subsystem.

```
import commands2
class Drivetrain(commands2.SubsystemBase):


    def __init__(self):
        super().__init__()
```

Now in robotcontainer.py we can make ArcadeDrive the default command for that subsytem.

Creating an Autononomous Command

Let's make an autonomous command. Something that we can get to run in autonomous. Hopefully, you will see how similar it is to the way that we did it before. We won't do anything too difficult here (No PIDController, for now).

Start by making a new file in the commands folder, `drivedistance.py`. And outline the code as below:

```
import commands2

from drivetrain import Drivetrain


class DriveDistance(commands2.CommandBase):
    def __init__(self, speed: float, meters: float, drive: Drivetrain) -> None:
        """Creates a new DriveDistance. This command will drive your your robot for a desired distance at
        a desired speed.

        :param speed:  The speed at which the robot will drive
        :param meters: The number of meters the robot will drive
        :param drive:  The drivetrain subsystem on which this command will run
        """
        super().__init__()

    def initialize(self) -> None:
        """Called when the command is initially scheduled."""
```

```
        pass

    def execute(self) -> None:
        """Called every time the scheduler runs while the command is scheduled."""
        pass

    def end(self, interrupted: bool) -> None:
        """Called once the command ends or is interrupted."""
        pass

    def isFinished(self) -> bool:
        """Returns true when the command should end."""
        # Compare distance travelled from start to desired distance
        pass
```

This shows a typical structure for a command. Let's work through each piece separately. The `__init__` is called when we create the command. Probably before the robot is really doing anything. Shortly after we turn it on. It isn't called when we start the command. That is the `initialize` method. So this simply to set some class properties.

```
def __init__(self, speed: float, meters: float, drive: Drivetrain) -> None:
        """Creates a new DriveDistance. This command will drive your your robot for a desired distance at
        a desired speed.

        :param speed:  The speed at which the robot will drive
        :param meters: The number of meters the robot will drive
        :param drive:  The drivetrain subsystem on which this command will run
        """
        super().__init__()

        self.distance = meters
        self.speed = speed
        self.drive = drive
        self.addRequirements(drive)
```

Note the `addRequirements` command. Commands need to know what parts of the robot are being commanded at one time. Only one command for each subsystem.

Next the initialize method:

```
def initialize(self) -> None:
        """Called when the command is initially scheduled."""
        self.drive.arcadeDrive(0, 0)
        self.drive.resetEncoders()
```

This gives us the opportunity to reset anything that needs it before we start.

Then the execute command.

```
    def execute(self) -> None:
        """Called every time the scheduler runs while the command is scheduled."""
        self.drive.arcadeDrive(0, self.speed)
```

Then the ended part of the command:

```
    def end(self, interrupted: bool) -> None:
        """Called once the command ends or is interrupted."""
        self.drive.arcadeDrive(0, 0)
```

And finally the isFinished. To determine when the command is over.

```
    def isFinished(self) -> bool:
        """Returns true when the command should end."""
```

```
        # Compare distance travelled from start to desired distance
        return abs(self.drive.averageDistanceMeter()) >= self.distance
```

Now we have that command. We need to integrate it into the Smartdashboard.

Go to the robotcontainer.py file.

In the init method add a chooser.

```
        # Create SmartDashboard chooser for autonomous routines
        self.chooser = wpilib.SendableChooser()
```

In the `_configure` method add a default option to the chooser:

```
    def _configure(self):
        self.chooser.setDefaultOption("Drive Distance", DriveDistance(.5, .3, self.drivetrain))
#         self.chooser.addOption("Go straight 2m", DriveStraight(self.drivetrain, 2))
        wpilib.SmartDashboard.putData(self.chooser)
        self.drivetrain.setDefaultCommand(self.getArcadeDriveCommand())

    def get_autonomous(self):
        return self.chooser.getSelected()
```

Now in `robot.py` we will need to ask the robot to get the chosen autonomous.

```
   def autonomousInit(self) -> None:
        #self.auto = self.container.get_autonomous()
        self.autonomousCommand = self.container.get_autonomous()

        if self.autonomousCommand:
            self.autonomousCommand.schedule()
```

and

```
   def teleopInit(self) -> None:
        if self.autonomousCommand:
            self.autonomousCommand.cancel()
```

Homework

Rewrite drivedistance.py to incorporate a PID controller.