Commands & Subsystems

Overview

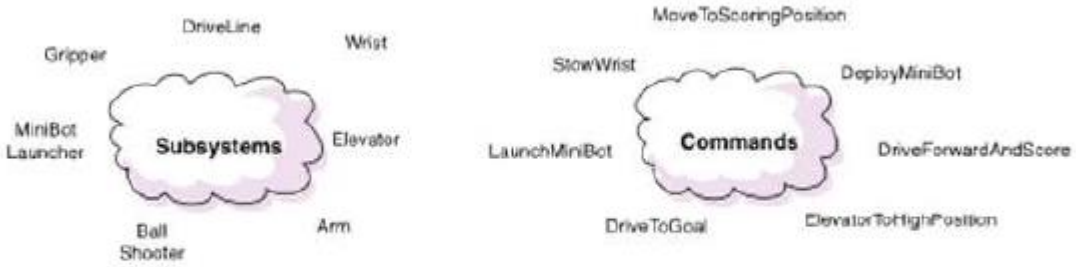
What is Command-Based Programming?

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. Some of the characteristics of robot programs that might be different from other desktop programs are:

- Activities happen over time, for example a sequence of steps to shoot a Frisbee or raise an elevator and place a tube on a goal.
- These activities occur concurrently, that is it might be desirable for an elevator, wrist and gripper to all be moving into a pickup position at the same time to increase robot performance.
- It is desirable to test the robot mechanisms and activities each individually to help debug your robot.
- Often the program needs to be augmented with additional autonomous programs at the last minute, perhaps at competitions, so easily extendable code is important.

Command based programming supports all these goals easily to make the robot program much simpler than using some less structured technique.

Commands and subsystems



Programs based on the WPILib library are organized around two fundamental concepts: **Subsystems** and **Commands**.

Subsystems - define the capabilities of each part of the robot and are subclasses of Subsystem.

Commands – define the operation of the robot incorporating the capabilities defined in the subsystems. Commands are subclasses of Command or CommandGroup. Commands run when scheduled or in response to buttons being pressed or virtual buttons from the SmartDashboard.

FRC Programming (1-23-2018), Pages 209-210

Simple Subsystems

Subsystems are the parts of your robot that are independently controlled like collectors, shooters, drive bases, elevators, arms, wrists, grippers, etc. Each subsystem is coded as an instance of the Subsystem class. Subsystems should have methods that define the operation of the actuators and sensors but not more complex behavior that happens over time.

This is an example of a fairly straightforward subsystem that operates a claw on a robot. The claw mechanism has a single motor to open or close the claw and no sensors (not necessarily a good idea in practice, but works for the example). The idea is that the open and close operations are simply timed. There are three methods, open(), close(), and stop() that operate the claw motor. Notice that there is not specific code that actually checks if the claw is opened or closed. The open method gets the claw moving in the open direction and the close method gets the claw moving in the close direction. Use a command to control the timing of this operation to make sure that the claw opens and closes for a specific period of time.

Creating a subsystem

```
import edu.wpi.first.wpilibj.*;
import edu.wpi.first.wpilibj.command.Subsystem;
import org.usfirst.frc.team1.robot.RobotMap;

public class Claw extends Subsystem {

        Victor motor = RobotMap.clawMotor;

    public void initDefaultCommand() {
    }

    public void open() {
            motor.set(1);
    }

    public void close() {
            motor.set(-1);
    }

    public void stop() {
            motor.set(0);
    }
}
```

Note: At Mater Dei, our subsystems extend MDSubsystem, not Subsystem

FRC Programming (1-23-2018), Pages 219-220

Operating the Claw with a Command

```
package org.usfirst.frc.team1.robot.commands;

import edu.wpi.first.wpilibj.command.Command;
import org.usfirst.frc.team1.robot.Robot;
/**
 *
 */
public class OpenClaw extends Command {

    public OpenClaw() {
        requires(Robot.claw);
        setTimeout(.9);
    }

    protected void initialize() {
            Robot.claw.open()
    }

    protected void execute() {
    }
```

Note: At Mater Dei, our commands extend MDCommand, not Command

```
    protected boolean isFinished() {
        return isTimedOut();
    }

    protected void end() {
            Robot.claw.stop();
    }

    protected void interrupted() {
            end();
    }
}
```

Commands provide the timing of the subsystems operations. Each command would do a different operation with the subsystem, the Claw in this case. The commands provides the timing for opening or closing. Here is an example of a simple Command that controls the opening of the claw. Notice that a timeout is set for this command (0.9 seconds) to time the opening of the claw and a check for the time in the isFinished() method. You can find more details in the article about using commands.

FRC Programming (1-23-2018), Pages 220-221

Basic Command Format

```
public class MyCommandName extends Command {

        /*
         * 1.       Constructor - Might have parameters for this command such as target
positions of devices. Should also set the name of the command for debugging purposes.
         *  This will be used if the status is viewed in the dashboard. And the command
should require (reserve) any devices is might use.
         */

    public MyCommandName() {
            super("MyCommandName");
        requires(elevator);
    }
```

```
    //          initialize() - This method sets up the command and is called immediately
before the command is executed for the first time and every subsequent time it is started .
    //  Any initialization code should be here.
    protected void initialize() {
    }

    /*
     *          execute() - This method is called periodically (about every 20ms) and does
the work of the command. Sometimes, if there is a position a
     *  subsystem is moving to, the command might set the target position for the
subsystem in initialize() and have an empty execute() method.
     */
    protected void execute() {
    }

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return false;
    }
}
```

FRC Programming (1-23-2018), Pages 225-226

Simple Command Example

This example illustrates a simple command that will drive the robot using tank drive with values provided by the joysticks.

```
public class DriveWithJoysticks extends Command {

    public DriveWithJoysticks() {
            requires(drivetrain);// drivetrain is an instance of our Drivetrain subsystem
    }

    protected void initialize() {
    }

    /*
     * execute() - In our execute method we call a tankDrive method we have created in our
subsystem. This method takes two speeds as a parameter which we get from methods in the OI
class.
     * These methods abstract the joystick objects so that if we want to change how we get
the speed later we can do so without modifying our commands
     * (for example, if we want the joysticks to be less sensitive, we can multiply them
by .5 in the getLeftSpeed method and leave our command the same).
     */
    protected void execute() {
            drivetrain.tankDrive(oi.getLeftSpeed(), oi.getRightSpeed());
    }
```

```
    /*
     * isFinished - Our isFinished method always returns false meaning this command never
completes on it's own. The reason we do this is that this command will be set as the
default command for the subsystem. This means that whenever the subsystem is not running
another command, it will run this command. If any other command is scheduled it will
interrupt this command, then return to this command when the other command completes.
     */
    protected boolean isFinished() {
        return false;
    }

    protected void end() {
    }

    protected void interrupted() {
    }
}
```

FRC Programming (1-23-2018), Pages 228-229

Running Commands on Joystick Input

You can cause commands to run when joystick buttons are pressed, released, or continuously while the button is held down. This is extremely easy to do only requiring a few lines of code.

Create the Joystick object and JoystickButton objects

Java

```
public class OI {
    // Create the joystick and the 8 buttons on it
        Joystick leftJoy = new Joystick(1);
        Button button1 = new JoystickButton(leftJoy, 1),
                         button2 = new JoystickButton(leftJoy, 2),
                         button3 = new JoystickButton(leftJoy, 3),
                         button4 = new JoystickButton(leftJoy, 4),
                         button5 = new JoystickButton(leftJoy, 5),
                         button6 = new JoystickButton(leftJoy, 6),
                         button7 = new JoystickButton(leftJoy, 7),
                         button8 = new JoystickButton(leftJoy, 8);

}
```

In this example there is a Joystick object connected as Joystick 1. Then 8 buttons are defined on that joystick to control various aspects of the robot. This is especially useful for testing although generating buttons on SmartDashboard is another alternative for testing commands.

Associate the buttons with commands

```
public OI() {
        button1.whenPressed(new PrepareToGrab());
        button2.whenPressed(new Grab());
        button3.whenPressed(new DriveToDistance(0.11));
        button4.whenPressed(new PlaceSoda());
        button6.whenPressed(new DriveToDistance(0.2));
        button8.whenPressed(new Stow());

        button7.whenPressed(new SodaDelivery());
}
```

In this example most of the joystick buttons from the previous code fragment are associated with commands. When the associated button is pressed the command is run. This is an excellent way to create a teleop program that has buttons to do particular actions.

Default Command

In some cases you may have a subsystem which you want to always be running a command no matter what. So what do you do when the command you are currently running ends? That's where default commands come in.

What is the default command?

Each subsystem may, but is not required to, have a default command which is scheduled whenever the subsystem is idle (the command currently requiring the system completes). The most common example of a default command is a command for the drivetrain that implements the normal joystick control. This command may be interrupted by other commands for specific maneuvers ("precision mode", automatic alignment/targeting, etc.) but after any command requiring the drivetrain completes the joystick command would be scheduled again.

Setting the default command

```
public class ExampleSubsystem extends Subsystem {

    // Put methods for controlling this subsystem
    // here. Call these from Commands.

    public void initDefaultCommand() {
        // Set the default command for a subsystem here.
        setDefaultCommand(new MyDefaultCommand());
    }
}
```

All subsystems should contain a method called initDefaultCommand() which is where you will set the default command if desired. If you do not wish to have a default command, simply leave this method blank. If you do wish to set a default command, call setDefaultCommand from within this method, passing in the command to be set as the default.