COMMAND BASED PROGRAMMING

Last Updated: 08-13-2019

Table of Contents

Command based programming

Command based programming

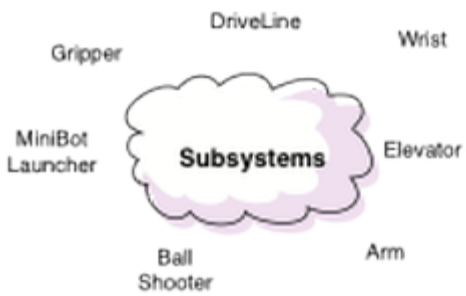
What is Command based programming?

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



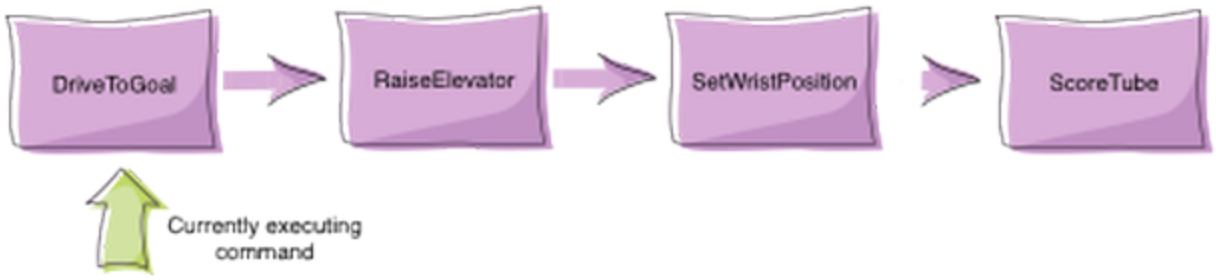
Programs based on the WPILib library are organized around two fundamental concepts: **Subsystems** and **Commands**.

Command based programming

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

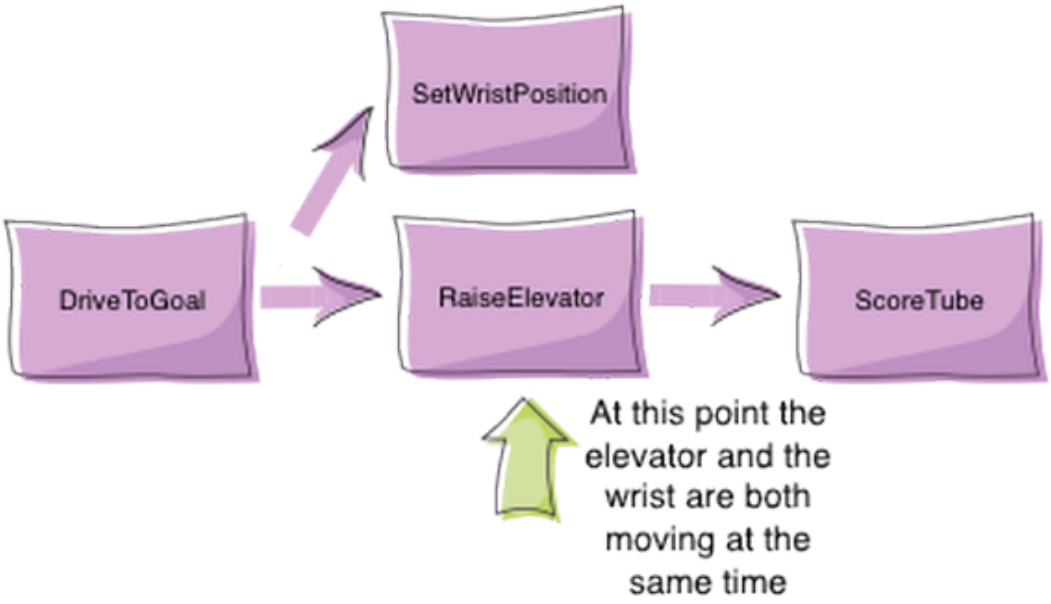
Commands - define the operation of the robot incorporating the capabilities defined in the subsystems. Commands are subclasses of Command or CommandGroup. Commands run when scheduled or in response to buttons being pressed or virtual buttons from the SmartDashboard.

How commands work



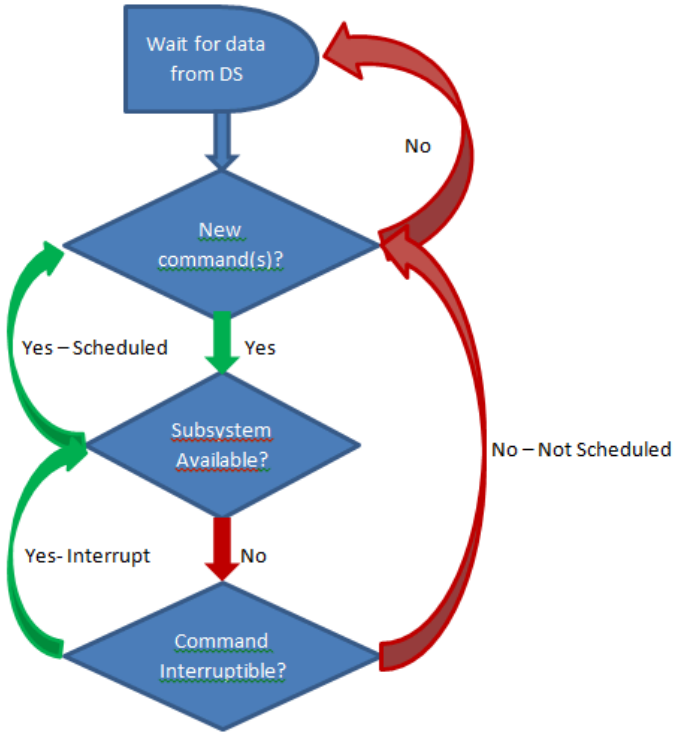
Commands let you break up the tasks of operating the robot into small chunks. Each command has an execute() method that does some work and an isFinished() method that tells if it is done. This happens on every update from the driver station or about every 20ms. Commands can be grouped together and executed sequentially, starting the next one in the group as the previous one finishes.

Concurrency



Sometimes it is desirable to have several operations happening concurrently. In the previous example you might want to set the wrist position while the elevator is moving up. In this case a command group can start a parallel command (or command group) running.

How It Works - Scheduling Commands



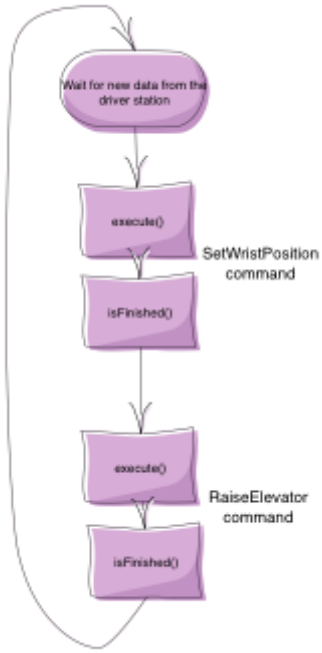
There are three main ways commands are scheduled:

1. Manually, by calling the start() method on the command (used for autonomous)
2. Automatically by the scheduler based on button/trigger actions specified in the code (typically defined in the OI class but checked by the Scheduler).
3. Automatically when a previous command completes (default commands and command groups).

Each time the driver station gets new data, the periodic method of your robot program is called. It runs a Scheduler that checks the trigger conditions to see if any commands need to be scheduled or canceled.

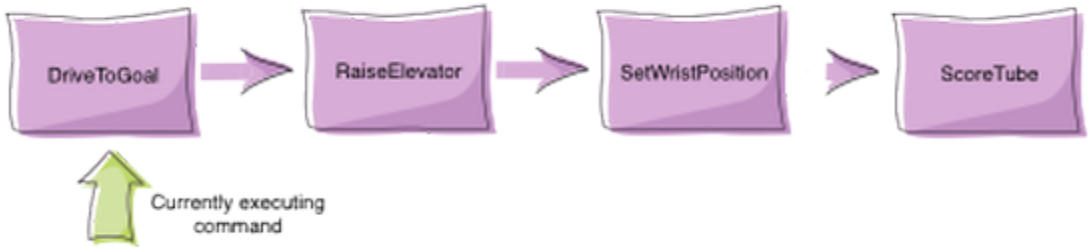
When a command is scheduled, the Scheduler checks to make sure that no other commands are using the same subsystems that the new command requires. If one or more of the subsystems is currently in use, and the current command is interruptible, it will be interrupted and the new command will be scheduled. If the current command is not interruptible, the new command will fail to be scheduled.

How It Works - Running Commands



After checking for new commands, the scheduler proceeds through the list of active commands and calls the execute() and isFinished() methods on each command. Notice that the apparent concurrent execution is done without the use of threads or tasks which would add complexity to the program. Each command simply has some code to execute (execute method) to move it further along towards its goal and a method (isFinished) that determines if the command has reached the goal. The execute and isFinished methods are just called repeatedly.

Command groups



More complex commands can be built up from simpler commands. For example, shooting a disc may be a long sequence of commands that are executed one after another. Maybe some of these commands in the sequence can be executed concurrently. Command groups are commands, but instead of having an isFinished and execute method, they have a list of other commands to execute. This allows more complex operations to be built up out of simpler operations, a basic principle in programming. Each of the individual smaller commands can be easily tested, the the group can be tested. More information on command groups can be found in the Creating groups of commands article.

Creating a command based robot project in C++

Installing the C++ Workbench plugin

This article is about command based programming in C++. If using Java, click here.

There is a plugin for Workbench that will make the creation and editing of command-based robot programs much simpler. The plugin had built-in templates for various types of commands, subsystems, and an overall robot program template. To use the plugin it must first be installed from the internet.

Show Advanced Device Development Perspective

Show Advanced Device Development Perspective

Select **Window** -> **Open Perspective** then click **Advanced Device Development** to open the perspective.

Install new software

Install new software

Select "Install new software..." from the Help menu in Workbench.

Select the plugin location

Select the plugin location

Select "http://first.wpi.edu/FRC/c/eclipse/update/" in the "Work with" text field and check the box to the left of "FRC Cpp Development Tools" in the list of plugins. Click "Next>" and continue through the dialog boxes accepting the terms of the license and the certificate if prompted. When finished, allow Wind River Workbench to restart when prompted to do so.

Creating a robot project - C++

Create a command-based robot project by using one of the template projects that are provided with the Wind River Workbench plugins.

Create the project

Create the project

Right-click in the project explorer window in some empty space. Select "New" then "Example...".

Selecting the project type

Selecting the project type

Select "VxWorks Downloadable Kernel Module Sample Project" from the "New Example" dialog box and click "Next>".

Select the project example

Select the project example

Select the "FRC Command Based Robot Template" project example and click "Finish".

Observe sample project in the Project Explorer window

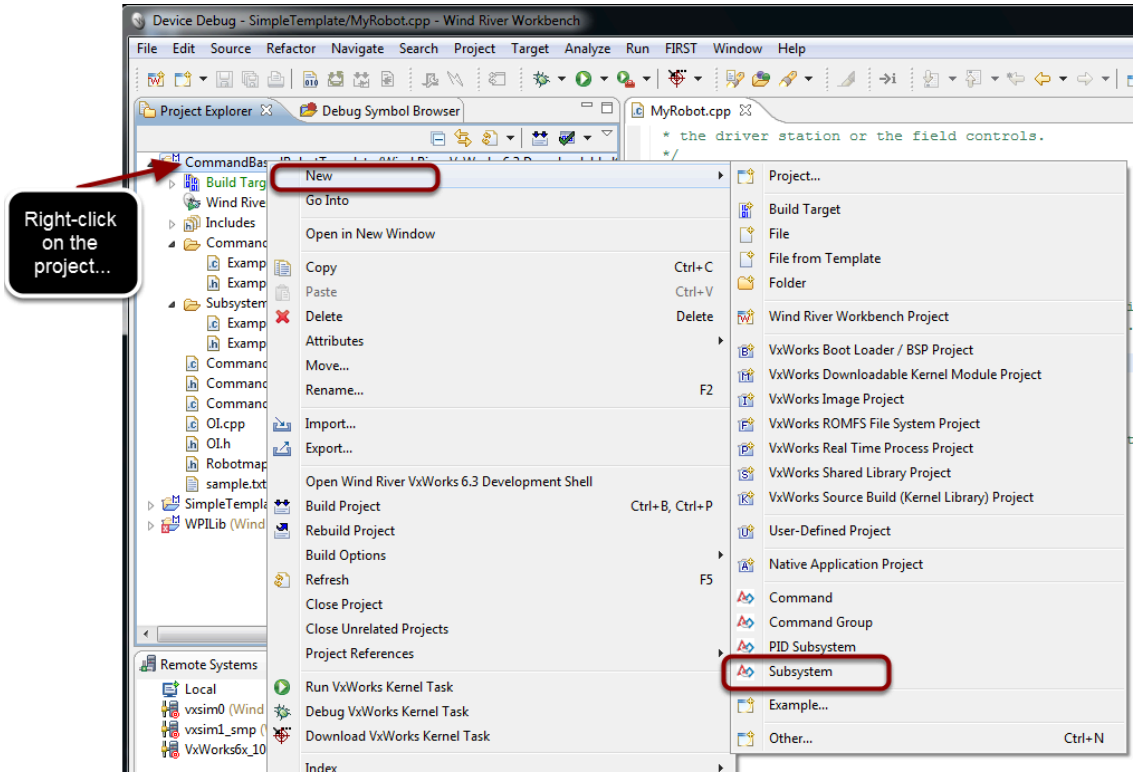
Observe sample project in the Project Explorer window

Notice that the CommandBasedRobotTemplate project has been added to the other projects that might have been in the Project Explorer window. There is a folder for Commands and another folder for Subsystems.

Adding Commands and Subsystems to the project - C++

Commands and Subsystems each are created as classes in C++. The plugin has built-in templates for both Commands and Subsystems to make it easier for you to add them to your program.

Adding subsystems to the project



To add a subsystem, right-click on the project name and select "New" then "Subsystem" in the drop down menu.

Naming the subsystem

Naming the subsystem

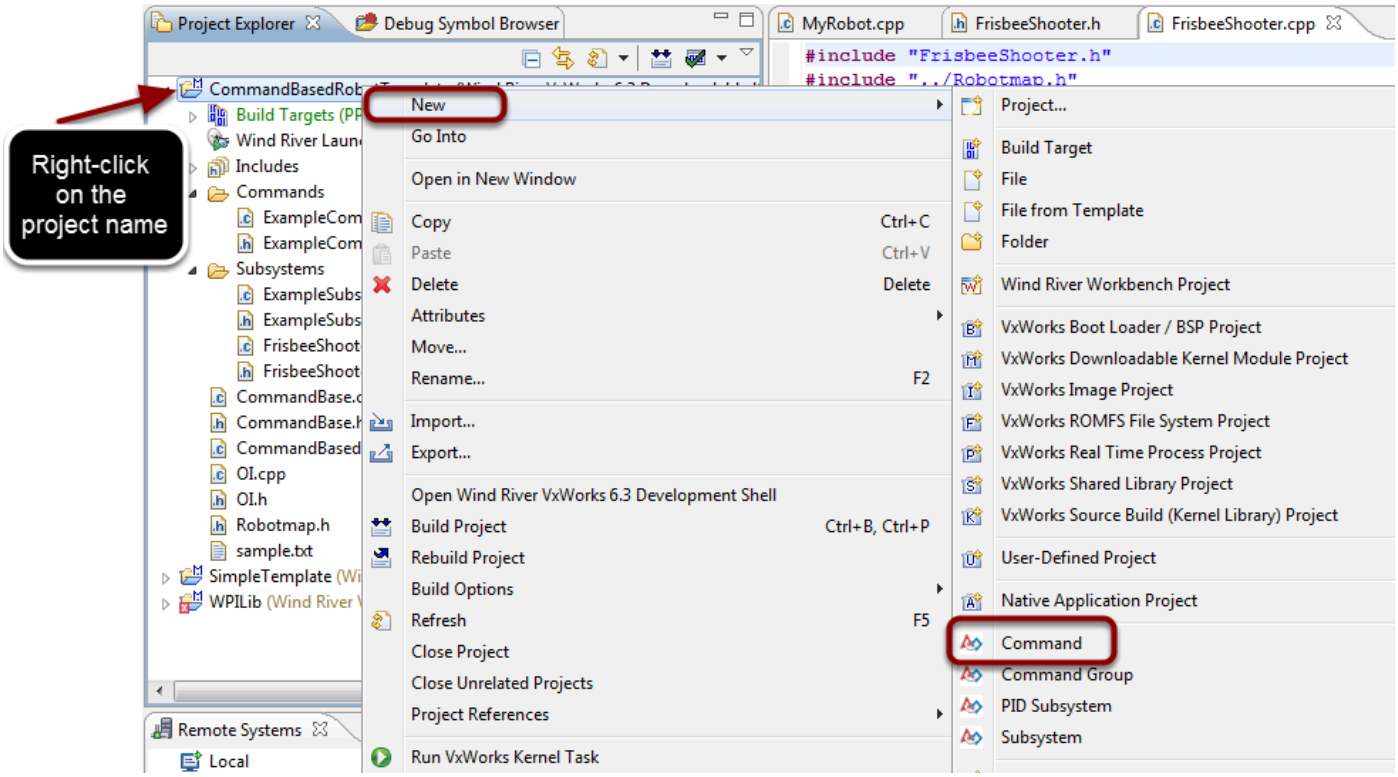
Fill in a name for the subsystem. This will become the resultant class name for the subsystem so the name has to be a valid C++ class name.

Subsystem created in project

Subsystem created in project

You can see the new subsystem created in the Subsystems folder in the project. To learn more about creating subsystems, see the Simple Subsystems article.

Adding a command to the project



A command can be created for the project using steps similar to creating a subsystem. First right-click on the project name in the Project Explorer and select "New Command".

Set the command name

Set the command name

Enter the Command name into the "Desired Command Name" field in the dialog box. This will be the class name for the Command so it must be a valid C++ name.

Command created in the project

Command created in the project

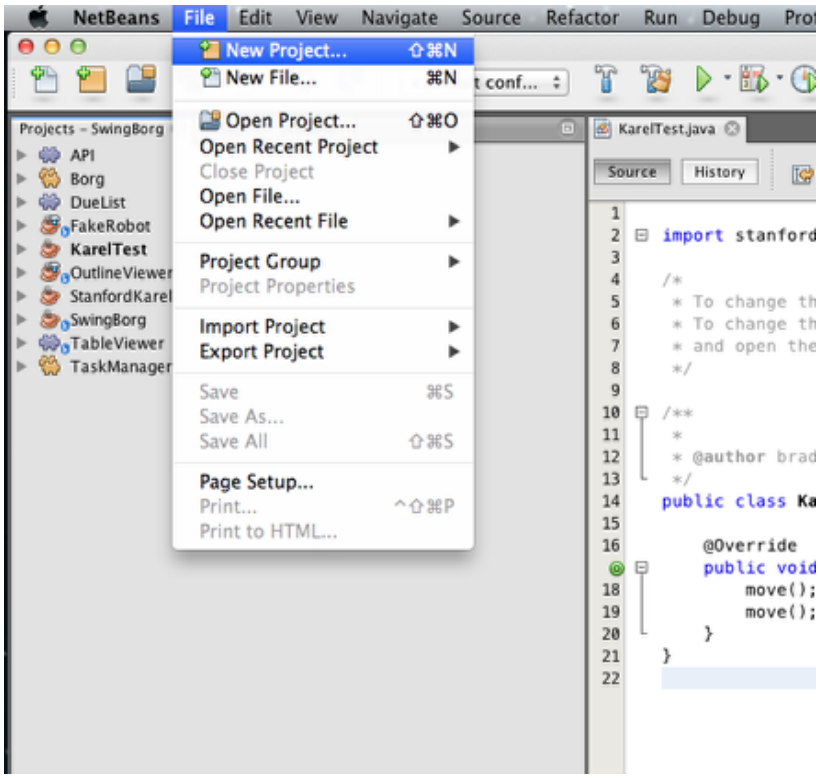
You can see that the Command has been created in the Commands folder in the project in the Project Explorer window. To learn more about creating commands, see the Creating Simple Commands article.

Creating a command based robot project in Java

Creating a robot project - Java

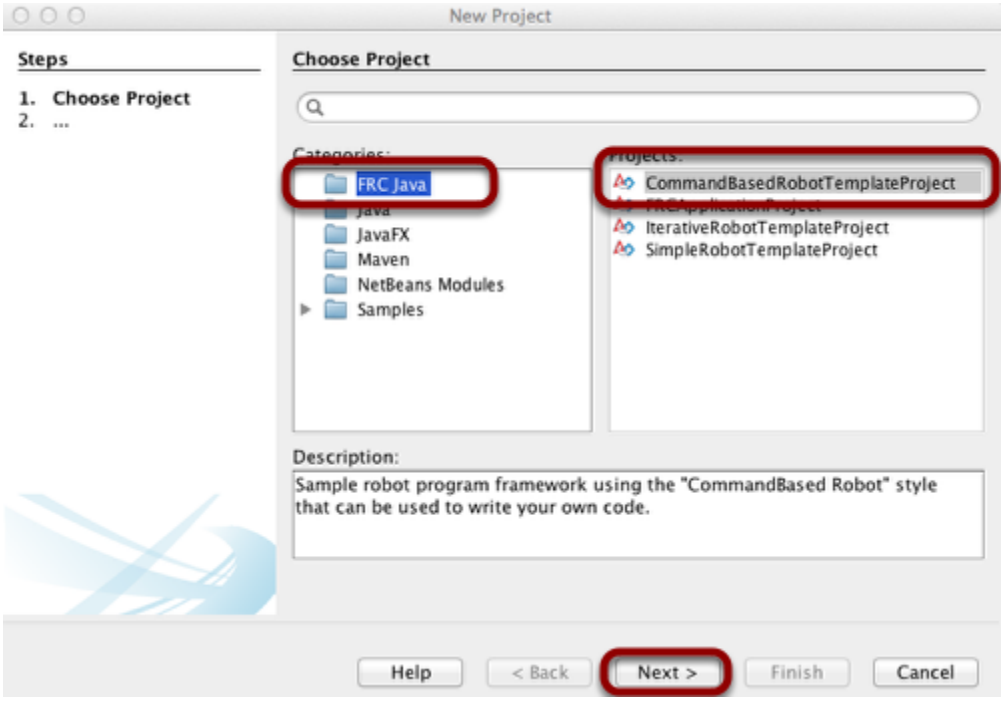
This article is about command based programming in Java, if you are programming in C++ click here to skip ahead to learning about subsytems.

Create a new project



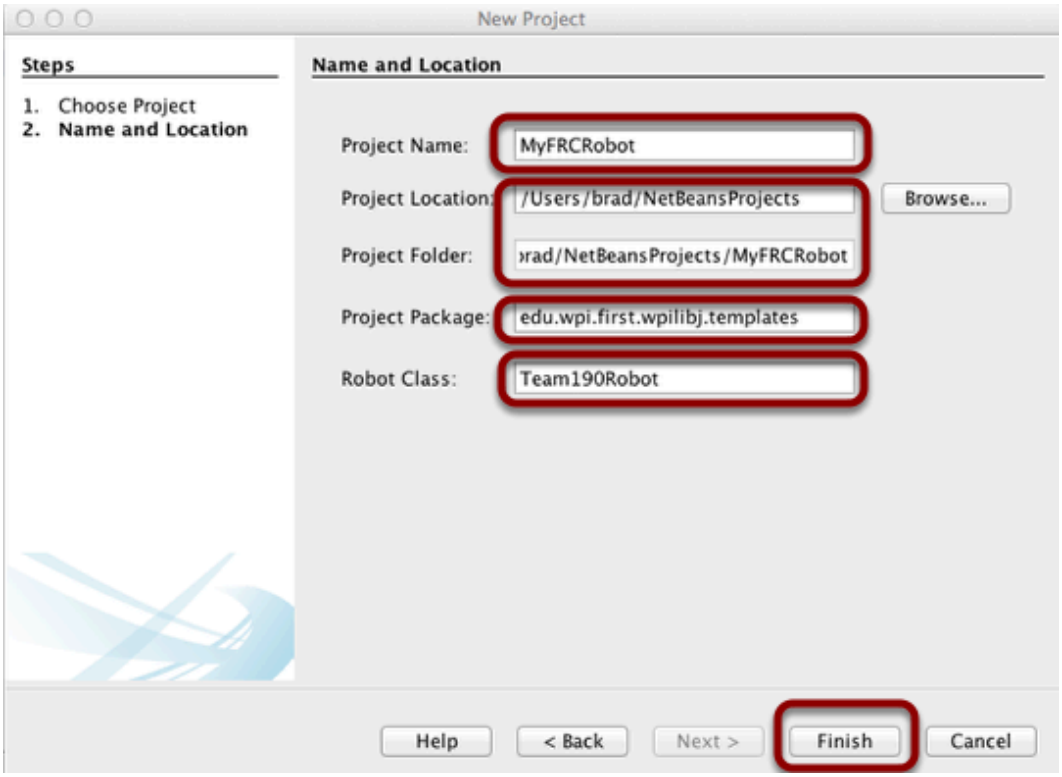
Create the project in NetBeans by selecting File then New Project...

Choose project type



Select the project type to create. In this case since we are creating a command based robot project, select FRC Java for the category, then CommandBasedRobotTemplateProject from the list of project types and select Next. All the base files will be automatically created for you.

Name the project and set parameters for create



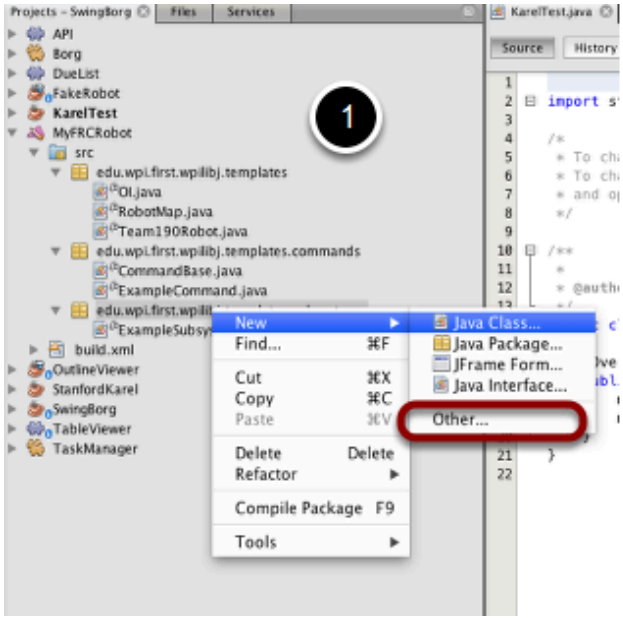
Here you can specify a project name and location where it will be stored. In addition you can supply a name for the base robot class. The Package should be something that uniquely identifies your team or organization. The package declaration qualifies all the code in the project so that code could be shared between organizations without conflicts.

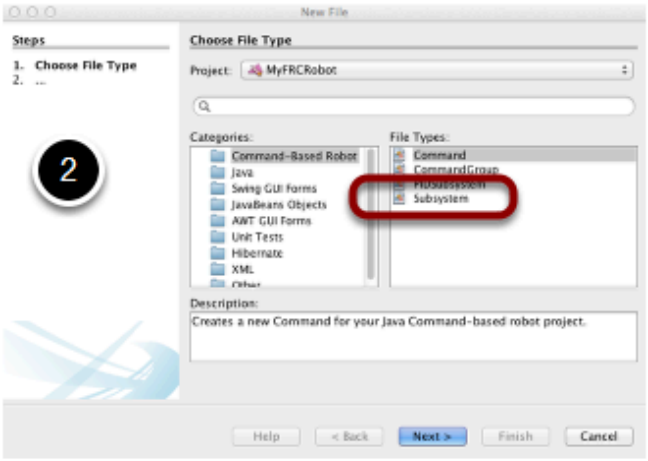
After you hit Finish, the project will be created.

Adding Commands and Subsystems - Java

A newly created command based robot project will have a set of default files and packages that were provided by the template. Using these files will make it easy to extend the default program into a custom robot program for your application.

Adding subsystems

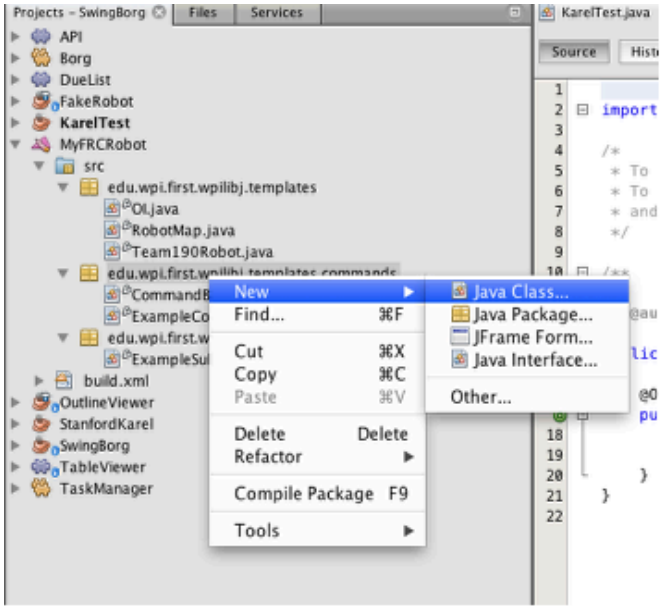


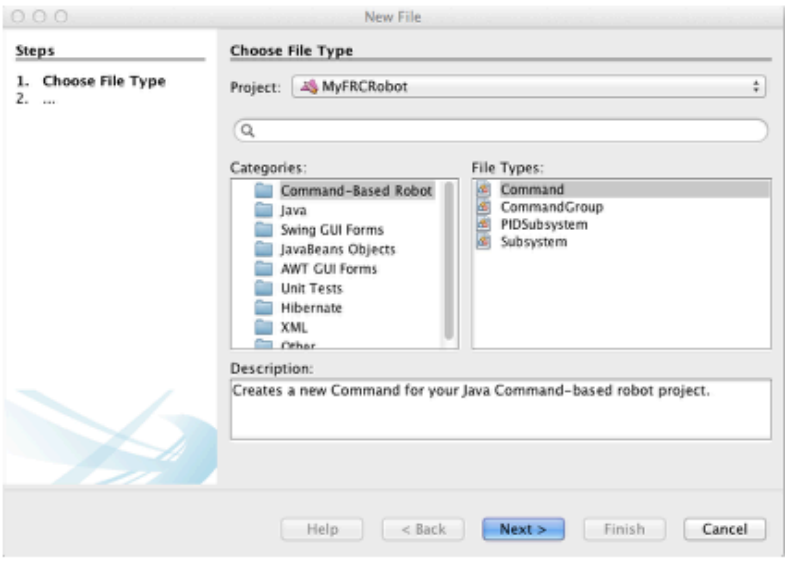


Add a new Subsystem class instance. Right-click on the subsystem package, and select New Subsystem. If it is not there, then select New Other (as shown above), then select subsystem. In the future, subsystem will be a choice on the New menu. Name the subsystem in the next dialog that pops up and click Finish.

To learn more about defining subsystems, see the Simple Susbsystems article.

Adding commands





The procedure to add commands is similar to adding subsystems (above), except select New Command from the menu (or New Other...), then Command.

To learn more about creating commands, see the Creating Simple Commands article.

Defining robot subsystems

Simple subsystems

Subsystems are the parts of your robot that are independently controller 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.

Creating a subsystem

Creating a subsystem

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.

Operating the claw with a command

Operating the claw with a command

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 th e opening of the claw and a check for the time in the isFinished() method. You can find more details in the section on using Commands.

PIDSubsystems for built-in PID control

If a mechanism uses a sensor for feedback then most often a PID controller will be used to control the motor speed or position. Examples of subsystems that might use PID control are: elevators with potentiometers to track the height, shooters with encoders to measure the speed, wrists with potentiometers to measure the joint angle, etc.

There is a PIDController class built into WPILib, but to simplify its use for command based programs there is a PIDSubsystem. A PIDSubsystem is a normal subsystem with the PIDController built in and exposes the required methods for operation.

A PIDSubsystem to control the angle of a wrist joint

A PIDSubsystem to control the angle of a wrist joint

In this example you can see the basic elements of a PIDSubsystem for the wrist joint:

1. The Wrist subsystem extends PIDSubsystem.
2. The constructor passes a name for the subsystem and the P, I, and D constants that are used when computing the motor output values.
3. The returnPIDInput() method is where you return the sensor value that is providing the feedback for this subsystem. In this case it's a potentiometer connected to an AnalogChannel. This method is called about every 20ms and is used for the PID output calculation.
4. The usePIDOutput method is where the computed output value from the PIDController is applied to your motor. This method is called about every 20 ms to update the motor speed based on the PID parameters from the constructor and the sensor value from the returnPIDInput() method.

Adding robot behaviors - commands

Creating Simple Commands

This article describes the basic format of a Command and walks through an example of creating a command to drive your robot with Joysticks.

Basic Command Format

```
public class MyCommandName extends CommandBase {

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

   public void initialize() {
   }

   public void execute() {
   }

   public boolean isFinished() {
      return true-if-command-is-finished;
   }
}
```

To implement a command, a number of methods are overridden from the WPILib Command class. Most of the methods are boiler plate and can often be ignored, but are there for maximum flexibility when you need it. There a number of parts to this basic command class:

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.
2. 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.
3. 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.

4. isFinished() - This method returns true if the command is finished. This would be the case if the command has reached its target position, run for the set time, etc. There are other methods that might be useful to override and these will be discussed in later sections

Simple Command Example

```
public class DriveWithJoysticks extends CommandBase {

    public DriveWithJoysticks() {
        requires(drivetrain); 1
    }

    protected void initialize() {
    }

    protected void execute() {
        drivetrain.tankDrive(oi.getLeftSpeed(), oi.getRightSpeed());
    } 2

    protected boolean isFinished() {
        return false; 3
    }

    protected void end() {
    }

    protected void interrupted() {
    }
}
```

1. This example illustrates a simple command that will drive the robot using tank drive with values provided by the joysticks. The elements we've used in this command:
2. requires(drivetrain) - "drivetrain" is an instance of our Drivetrain subsystem. The instance is instantiated as static in Command Base so it can be referenced here. We need to require the drivetrain system as this command uses it when it executes.
3. 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).

4. 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. For more on default commands see Default Commands.

Creating groups of commands

Once you have created commands to operate the mechanisms in your robot, they can be grouped together to get more complex operations. These groupings of commands are called CommandGroups and are easily defined as shown in this article.

Creating a command to do a complex operation

Creating a command to do a complex operation

This is an example of a command group that places a soda can on a table. To accomplish this, (1) the robot elevator must move to the "TABLE_HEIGHT", then (2) set the wrist angle, then (3) open the claw. All of these tasks must run sequentially to make sure that the soda can isn't dropped. The addSequential() method takes a command (or a command group) as a parameter and will execute them one after another when this command is scheduled.

Running commands in parallel

Running commands in parallel

To make the program more efficient, often it is desirable to run multiple commands at the same time. In this example, the robot is getting ready to grab a soda can. Since the robot isn't holding anything, all the joints can move at the same time without worrying about dropping anything. Here all the commands are run in parallel so all the motors are running at the same time and each completes whenever the isFinished() method is called. The commands may complete out of order. The steps are: (1) move the wrist to the pickup setpoint, then (2) move the elevator to the floor pickup position, and (3) open the claw.

Mixing parallel and sequential commands

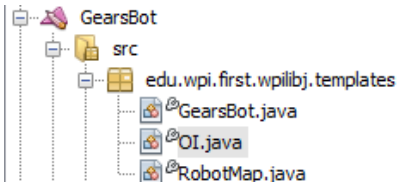
Mixing parallel and sequential commands

Often there are some parts of a command group that must complete before other parts run. In this example, a soda can is grabbed, then the elevator and wrist can move to their stowed positions. In this case, the wrist and elevator have to wait until the can is grabbed, then they can operate independently. The first command (1) CloseClaw grabs the soda and nothing else runs until it is finished since it is sequential, then the (2) elevator and (3) wrist move at the same time.

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.

The OI Class




```
21  public class OI {
22      // Create the joystick and of the 8 buttons on it
23      Joystick leftJoy = new Joystick(1);
24      Button button1 = new JoystickButton(leftJoy, 1),
25             button2 = new JoystickButton(leftJoy, 2),
```

The command based template contains a class called OI, located in OI.java, where Operator Interface behaviors are typically defined. If you are using RobotBuilder this file can be found in the org.usfirst.frc####.NAME package

Create the Joystick object and JoystickButton objects

Create the Joystick object and JoystickButton objects

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

Associate the buttons with commands

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.

Other options

In addition to the "whenPressed()" condition showcased above, there are a few other conditions you can use to link buttons to commands:

- Commands can run when a button is released by using whenReleased() instead of whenPressed().
- Commands can run continuously while the button is depressed by calling whileHeld().
- Commands can be toggled when a button is pressed using toggleWhenPressed().
- A command can be canceled when a button is pressed using cancelWhenPressed().

Additionally commands can be triggered by arbitrary conditions of your choosing by using the Trigger class instead of Button. Triggers (and Buttons) are usually polled every 20ms or whenever the scheduler is called.

Running commands during the autonomous period

Once commands are defined they can run in either the teleop or autonomous part of the program. In fact, the power of the command based programming approach is that you can reuse the same commands in either place. If the robot has a command that can shoot Frisbees during autonomous with camera aiming and accurate shooting, there is no reason not to use it to help the drivers during the teleop period of the game.

Creating a command to use for Autonomous

Creating a command to use for Autonomous

Our robot must do the following tasks during the autonomous period: pick up a soda can off the floor then drive a set distance from a table and deliver the can there. The process consists of:

1. Prepare to grab (move elevator, wrist, and gripper into position)
2. Grab the soda can
3. Drive to a distance from the table indicated by an ultrasonic rangefinder
4. Place the soda
5. Back off to a distance from the rangefinder
6. Re-stow the gripper

To do these tasks there are 6 command groups that are executed sequentially as shown in this example.

Setting that command to run as the autonomous behavior

Setting that command to run as the autonomous behavior

To get the SodaDelivery command to run as the Autonomous program, (1) simply instantiate it in the robotInit() method, (2) start it during the autonomousPeriodic() method, and (3) be sure the

scheduler is called repeatedly during the teleopPeriodic() method. RobotInit() is called only once when the robot starts so it is a good time to create the command instance. AutonomousPeriodic() is called once at the start of the autonomous period so we schedule the command there. AutonomousPeriodic() is called every 20ms so that is a good time to run the scheduler which makes a pass through all the currently scheduled commands.

Converting a Simple Autonomous program to a Command based autonomous program

This document describes how to rewrite a simple autonomous into a command based autonomous. Hopefully, going through this process will help those more familiar with the older simple autonomous method understand the command based method better. By re-writing it as a command based program, there are several benefits in terms of testing and reuse. For this example, all of the logic is abstracted out into functions primarily so that the focus of this example can be on the structure.

The initial autonomous code with loops

The initial autonomous code with loops

The code above aims a shooter, then it spins up a wheel and, finally, once the wheel is running at the desired speed, it shoots the frisbee. The code consists of three distinct actions: aim, spin up to speed and shoot the Frisbee. The first two actions follow a command pattern that consists of four parts:

1. Initialization: Seen in lines 2 & 10, prepares for the action to be per- formed.

2. Condition: Seen in lines 3 & 11, keeps the loop going while it is satisfied.

3. Execution: Seen in lines 4 & 12, repeatedly updates the code to try to make the condition false.

4. End: Seen in lines 7 & 15, performs any cleanup and final task before moving on to the next action.

The last action seen in line 18 only has an explicit initialize, though depending on how you read it, it can implicitly end under a number of conditions. The most obvious one two in this case are when it's done shooting or when autonomous has ended.

Rewriting it as Commands

Rewriting it as Commands

The same code can be rewritten as a CommandGroup that groups the three actions, where each action is written as it's own command. First, the command group will be written, then the commands will be written to accomplish the three actions. This code is pretty straightforward. It does the three actions sequentially, that is one after the other. Line 3 aims the robot, then line 4 spins the shooter

2up and, finally, line 5 actually shoots the frisbee. The addSequential() method sets it so that these commands run one after the other.

The Aim command

The Aim command

As you can see, the command reflects the four parts of the action we discussed earlier. It also has the interrupted() method which will be discussed below. The other significant difference is that the condition in the isFinished() is the opposite of what you would put as the condition of the while loop, it returns true when you want to stop running the execute method as opposed to false. Initializing, executing and ending are exactly the same, they just go within their respective method to indicate what they do.

SpinUpShooter command

SpinUpShooter command

The spin up shooter command is very similar to the Aim command, it's the same basic idea.

Shoot command

Shoot command

The shoot command is the same basic transformation yet again, however it is set to end immediately. In CommandBased programming, it is better to have it's isFinished method return true when the act of shooting is finished, but this is a more direct translation of the original code.

Benefits of the command based approach

Why bother re-writing the code as CommandBased? Writing the code in the CommandBased style offers a number of benefits:

- **Re-Usability** You can reuse the same command in teleop and multiple autonomous modes. They all reference the same code, so if you need to tweak it to tune it or fix it, you can do it in one place without having to make the same edits in multiple places.
- **Testability** You can test each part using tools such as the SmartDashboard to test parts of the autonomous. Once you put them together, you'll have more confidence that each piece works as desired.
- **Parallelization** If you wanted this code to aim and spin up the shooter at the same time, it's trivial with CommandBased programming. Just use AddParallel() instead of AddSequential() when adding the Aim command and now aiming and spinning up will happen simultaneously.
- **Interruptibility** Commands are interruptible, this provides the ability to exit a command early, a task that is much harder in the equivalent while loop based code.

Default Commands

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());
    }
```

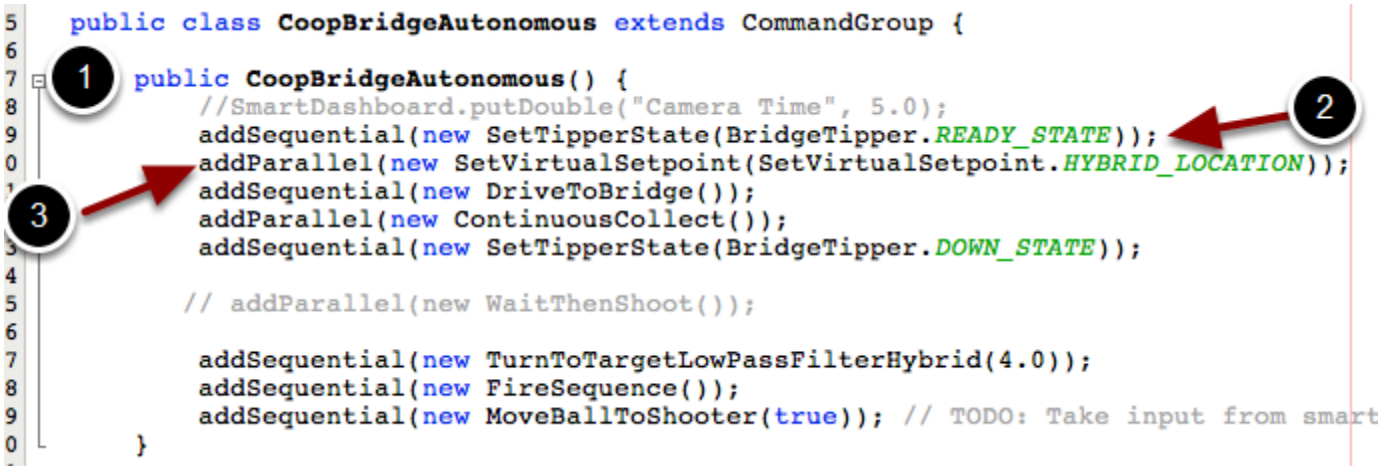
All subsystems should contain a method called initDefaultCommand() which is where you will set the default command if desired. If you do not wish to have a default command, simply leave this method blank. If you do wish to set a default command, call setDefaultCommand from within this method, passing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

Connecting behaviors to the operator interface

Synchronizing two commands

Commands can be nested inside of command groups to create more complex commands. The simpler commands can be added to the command groups to either run sequentially (each command finishing before the next starts) or in parallel (the command is scheduled, and the next command is immediately scheduled also). Occasionally there are times where you want to make sure that two parallel command complete before moving onto the next command. This article describes how to do that.

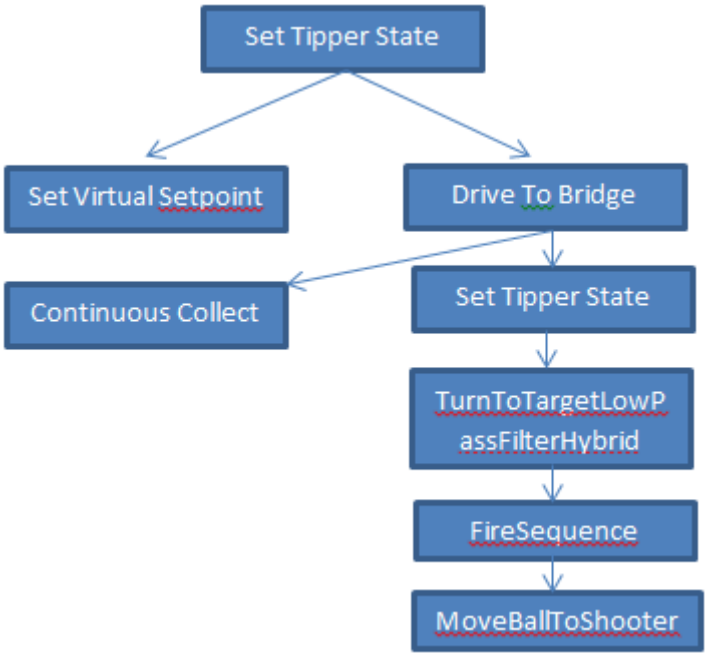
Creating a command group with sequential and parallel commands




```
public class CoopBridgeAutonomous extends CommandGroup {

    public CoopBridgeAutonomous() {
        //SmartDashboard.putDouble("Camera Time", 5.0);
        addSequential(new SetTipperState(BridgeTipper.READY_STATE));
        addParallel(new SetVirtualSetpoint(SetVirtualSetpoint.HYBRID_LOCATION));
        addSequential(new DriveToBridge());
        addParallel(new ContinuousCollect());
        addSequential(new SetTipperState(BridgeTipper.DOWN_STATE));

       // addParallel(new WaitThenShoot());

        addSequential(new TurnToTargetLowPassFilterHybrid(4.0));
        addSequential(new FireSequence());
        addSequential(new MoveBallToShooter(true)); // TODO: Take input from smart
    }
```

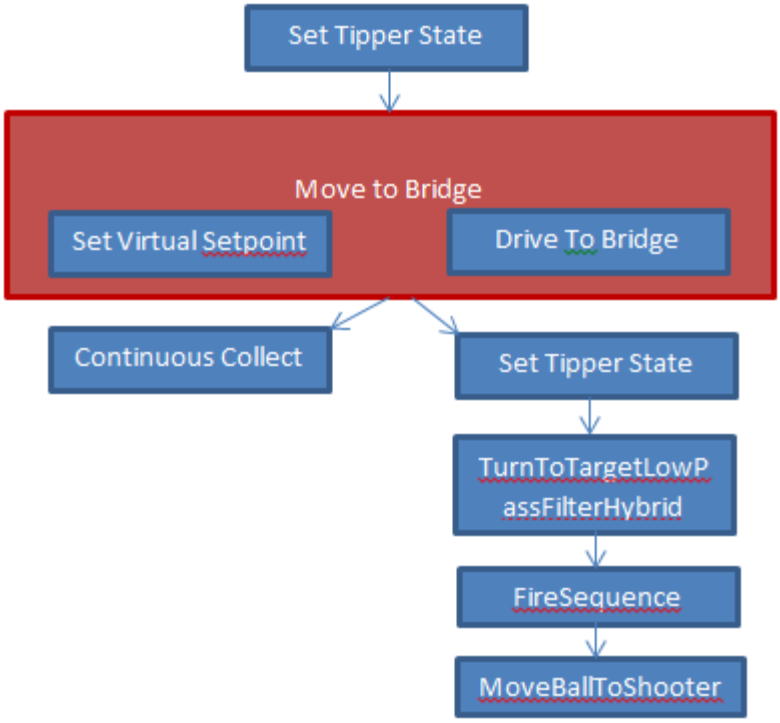
In this example some commands are added in parallel and others are added sequentially to the CommandGroup CoopBridgeAutonomous (1). The first command "SetTipperState" is added and completes before the SetVirtualSetpoint command starts (2). Before SetVirtualSetpoint command completes, the DriveToBridge command is immediately scheduled because of the SetVirtualSetpoint is added in parallel (3). This example might give you an idea of how commands are scheduled.

Example Flowchart



Here is the code shown above represented as a flowchart. Note that there is no dependency coming from the commands scheduled using "Add Parallel" either or both of these commands could still be running when the MoveBallToShooter command is reached. Any command in the main sequence (the sequence on the right here) that requires a subsystem in use by a parallel command will cause the parallel command to be canceled. For example, if the FireSequence required a subsystem in use by SetVirtualSetpoint, the SetVirtualSetpoint command will be canceled when FireSequence is scheduled.

Getting a command to wait for another command to complete



If there are two commands that need to complete before the following commands are scheduled, they can be put into a command group by themselves, adding both in parallel. Then that command group can be scheduled sequentially from an enclosing command group. When a command group is scheduled sequentially, the commands inside it will all finish before the next outer command is scheduled. In this way you can be sure that an action consisting of multiple parallel commands has completed before going on to the next command.

In this example you can see that the addition of a command group "Move to Bridge" containing the Set Virtual Setpoint and Drive to Bridge commands forces both to complete before the next commands are scheduled.