ROBOTBUILDER

Last Updated: 06-11-2016

Table of Contents

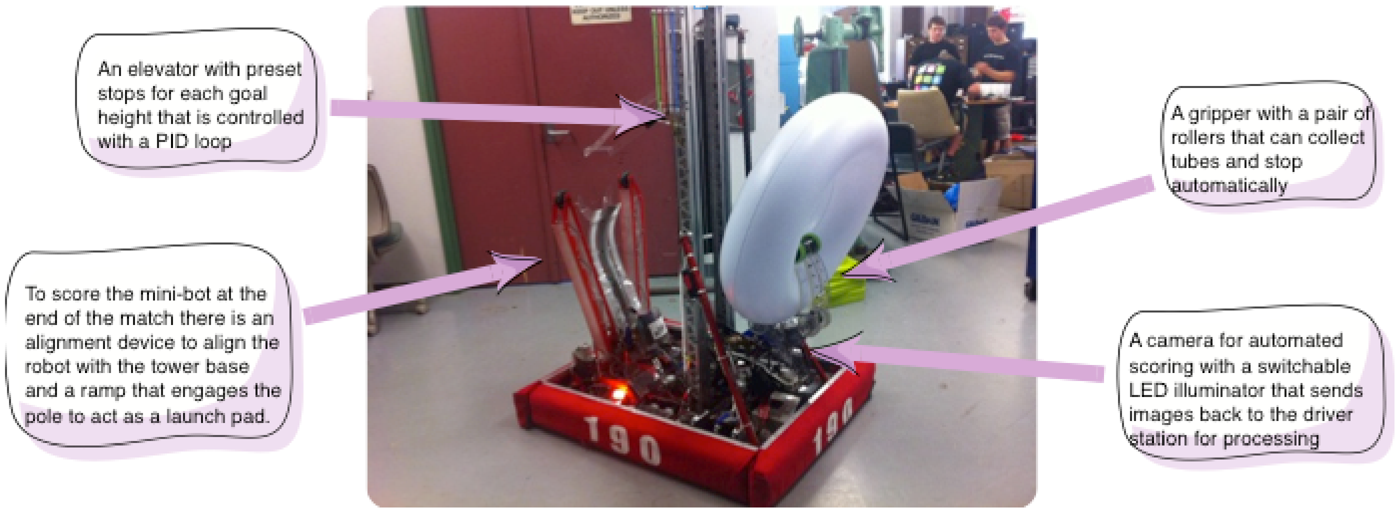
The basic steps to create a robot program

Overview of RobotBuilder

Creating a program with RobotBuilder is a very straight forward procedure by following a few steps that are the same for any robot. This lesson describes the steps that you can follow. You can find more details about each of these steps in subsequent sections of the document.

In addition to the text documentation provided here, a series of videos about RobotBuilder and many other FRC Robotics Engineering topics is also available.

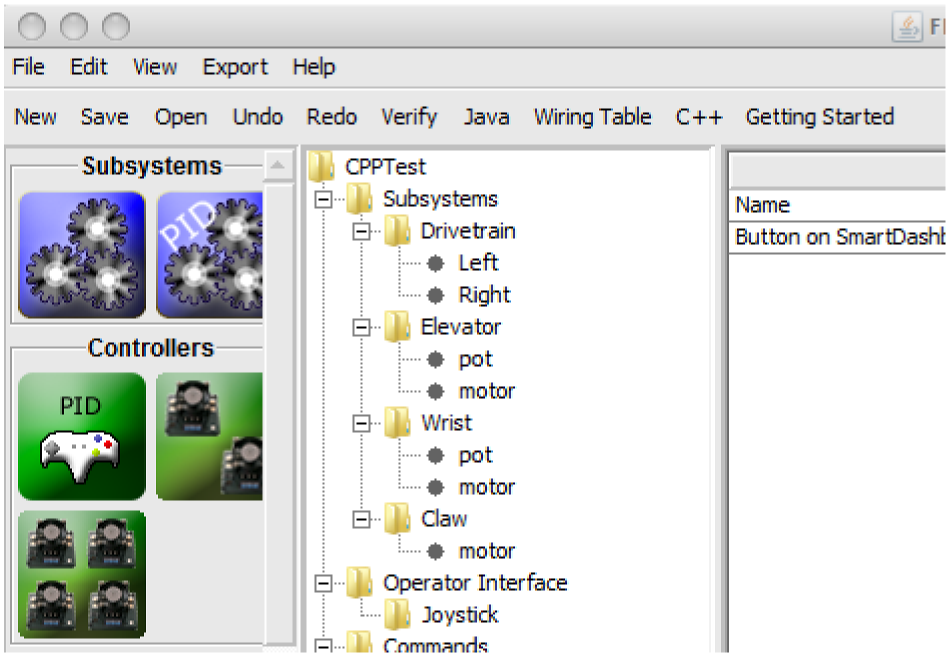
Divide the robot into subsystems


An elevator with preset stops for each goal height that is controlled with a PID loop
To score the mini-bot at the end of the match there is an alignment device to align the robot with the tower base and a ramp that engages the pole to act as a launch pad.
A gripper with a pair of rollers that can collect tubes and stop automatically
A camera for automated scoring with a switchable LED illuminator that sends images back to the driver station for processing
190

Your robot is naturally made up of a number of smaller systems like the drive trains, arms, shooters, collectors, manipulators, wrist joints, etc. You should look at the design of your robot and break it up into smaller, separately operated subsystems. In this particular example there is an elevator, a minibot alignment device, a gripper, and a camera system. In addition one might include the drive base. Each of these parts of the robot are separately controlled and make good candidates for subsystems.

For more information see: Creating a Subsystem

Add each subsystem to the RobotBuilder project



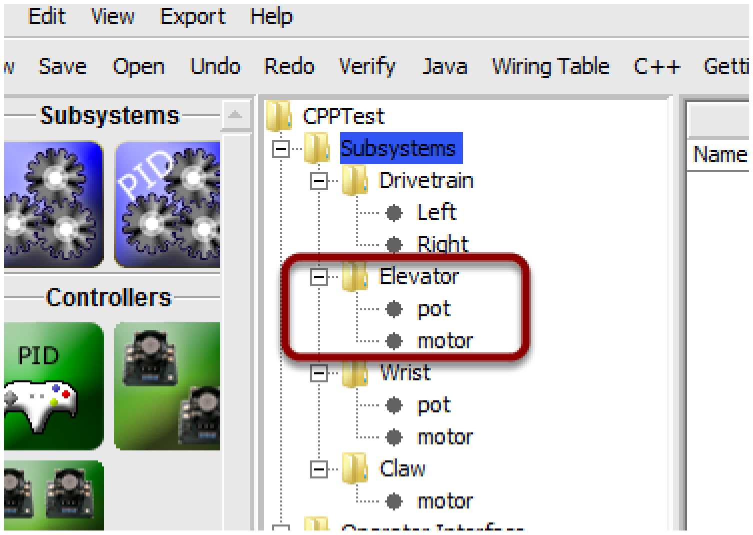
Each subsystem will be added to the "Subsystems" folder in the RobotBuilder and given a meaningful name. For each of the subsystems there are several attributes that get filled in to specify more information about the subsystems. In addition there are two types of subsystems that you might want to create:

1. PIDSubsystems - often it is desirable to control a subsystems operation with a PID controller. This is code in your program that makes the subsystem element, for example arm angle, more quickly to a desired position then stop when reaching it. PIDSubsystems have the PID Controller code built-in and are often more convenient then adding it yourself. PIDSubsystems have a sensor that determines when the device has reached the target position and an actuator (speed controller) that is driven to the setpoint.
2. Regular subsystem - these subsystems don't have an integrated PID controller and are used for subsystems without PID control for feedback or for subsystems requiring more complex control than can be handled with the default embedded PID controller.

As you look through more of this documentation the differences between the subsystem types will become more apparent.

For more information see: Creating a subsystem, Writing Java code for a subsystem and Writing C++ code for a subsystem

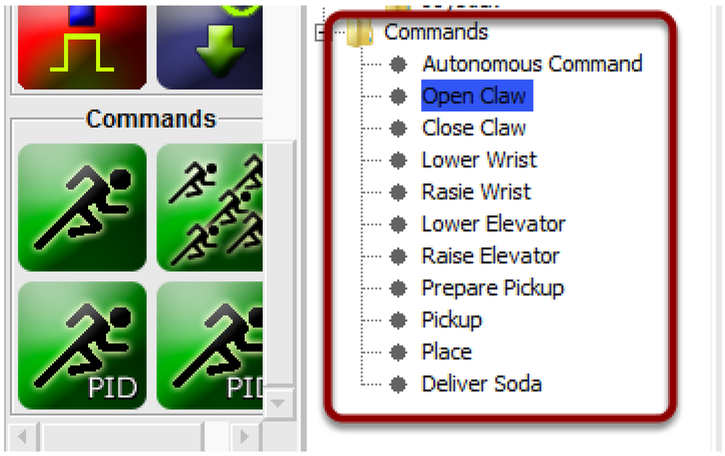
Add components to each of the subsystems



Each subsystem consists of a number of actuators, sensors and controllers that it uses to perform its operations. These sensors and actuators are added to the subsystem with which they are associated. Each of the sensors and actuators comes from the RobotBuilder palette and is dragged to the appropriate subsystem. For each, there are usually other properties that must be set such as port numbers and other parameters specific to the component.

In this example there is an Elevator subsystem that uses a motor and a potentiometer (motor and pot) that have been dragged to the Elevator subsystem.

Add commands to describe the goals for each subsystem

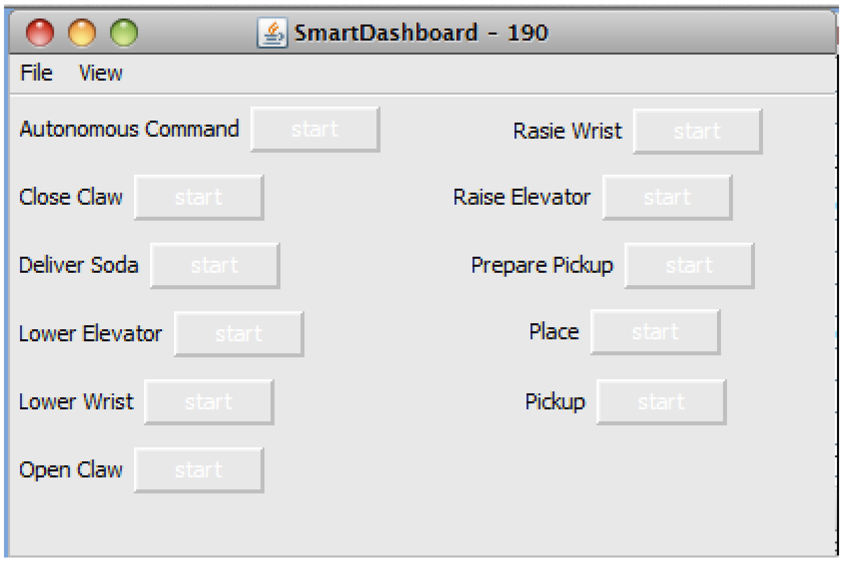


Commands are distinct goals that the robot will perform. These commands are added by dragging the command under the "Commands" folder. When creating a command, there are 3 primary choices (shown on the palette on the left of the picture):

- Normal commands - these are the most flexible command, you have to write all of the code to perform the desired actions necessary to accomplish the goal.
- Command groups - these commands are a combination of other commands running both in a sequential order and in parallel. Use these to build up more complicated actions after you have a number of basic commands implemented.
- Setpoint commands - setpoint commands move a PID Subsystem to a fixed setpoint, or the desired location.

For more information see: Creating a command, Writing the code for a command in Java and Writing the code for a command in C++

Test each command individually by starting it from the SmartDashboard

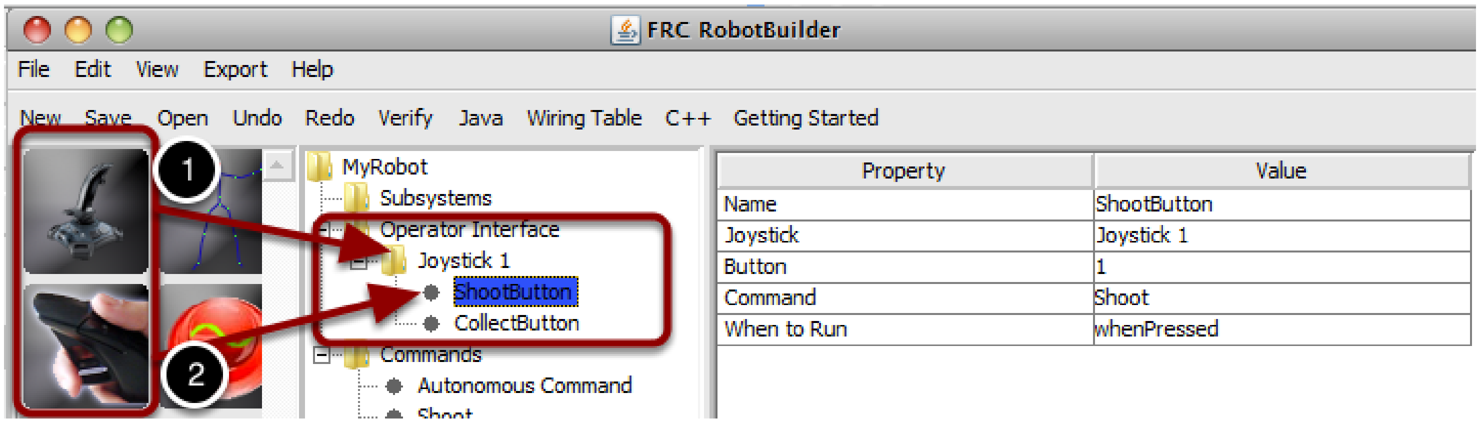


Each command can be run from the SmartDashboard. This is useful for testing commands before you add them to the operator interface or to a command group. As long as you leave the "Button on SmartDashboard" property checked, a button will be created on the SmartDashboard. When you press the start button, the command will run and you can check that it performs the desired action.

By creating buttons, each command can be tested individually. If all the commands work individually, you can be pretty sure that the robot will work as a whole.

For more information see: Adding a button to SmartDashboard to run a command

Add Operator Interface components

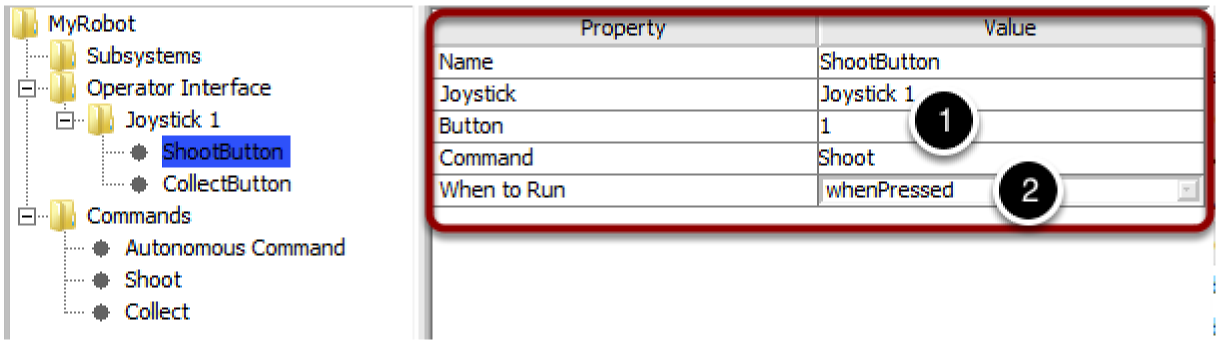


Property	Value
Name	ShootButton
Joystick	Joystick 1
Button	1
Command	Shoot
When to Run	whenPressed

The operator interface consists of joysticks, gamepads and other HID input devices. You can add operator interface components (joysticks, joystick buttons) to your program in RobotBuilder. It will automatically generate code that will initialize all of the components and allow them to be connected to commands.

The operator interface components are dragged from the palette to the "Operator Interface" folder in the RobotBuilder program. First (1) add Joysticks to the program then put buttons under the associated joysticks (2) and give them meaningful names, like ShootButton.

Connect the commands to the Operator Interface



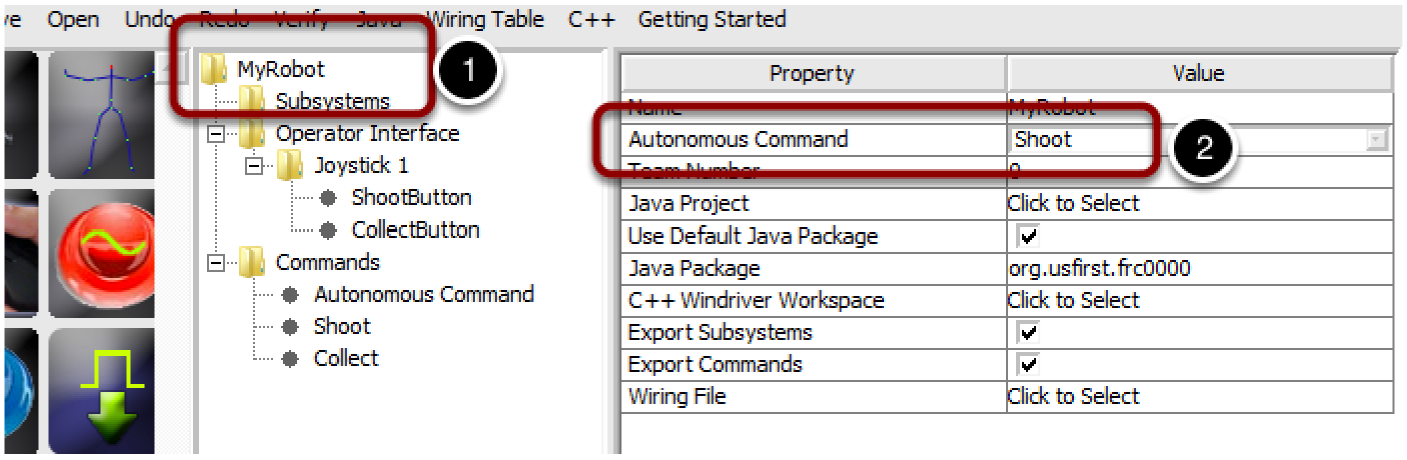
Property	Value
Name	ShootButton
Joystick	Joystick 1
Button	1
Command	Shoot
When to Run	whenPressed

Commands can be associated with buttons so that when a button is pressed the command is scheduled. This should, for the most part, handle most of the tele-operated part of your robot program.

This is simply done by (1) adding the command to the JoystickButton object in the RobotBuilder program, then (2) setting the condition in which the command is scheduled.

For more information see: Connecting the operator interface to a command

Develop one or more Autonomous commands



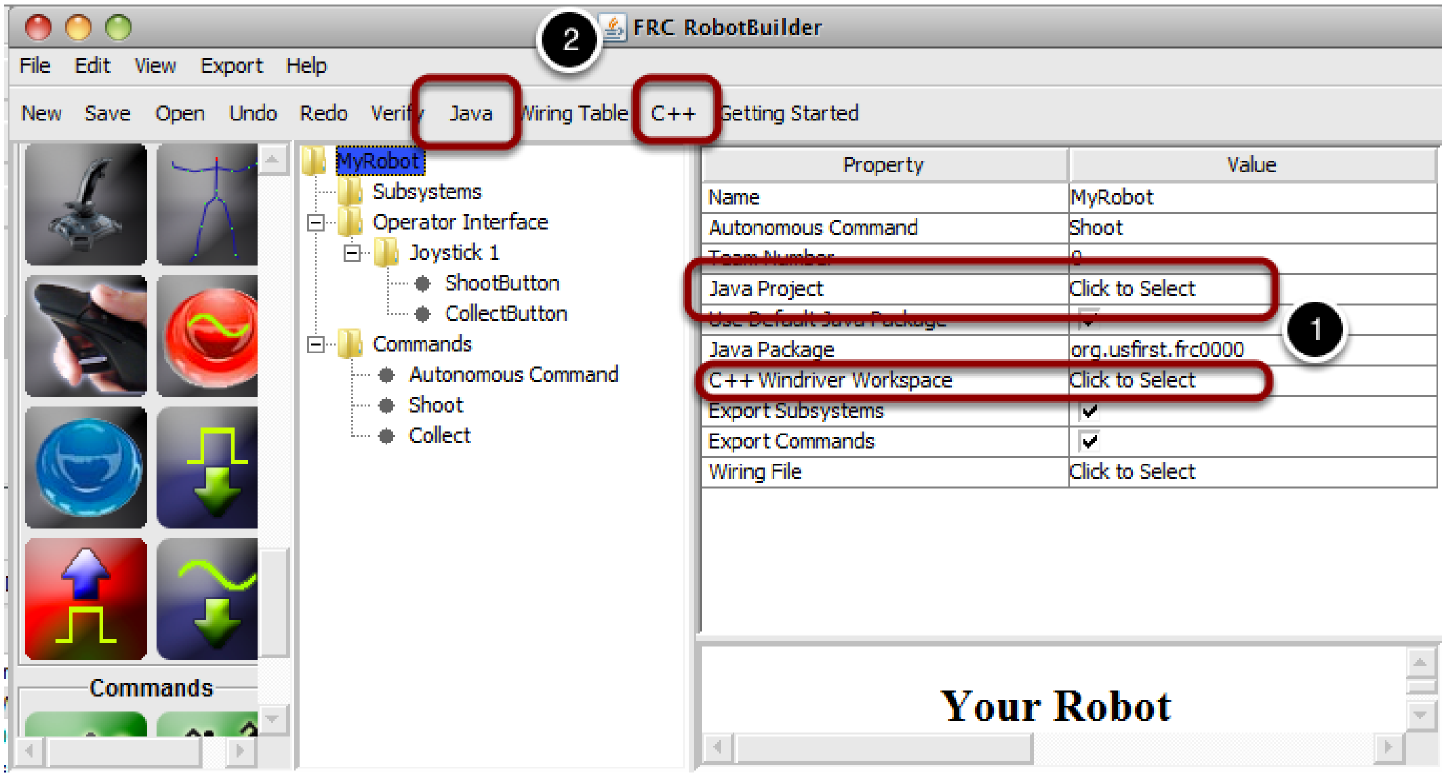
Property	Value
Name	MyRobot
Autonomous Command	Shoot
Team Number	0
Java Project	Click to Select
Use Default Java Package	✓
Java Package	org.usfirst.frc0000
C++ Windriver Workspace	Click to Select
Export Subsystems	✓
Export Commands	✓
Wiring File	Click to Select

Commands make it simple to develop autonomous programs. You simply specify which command should run when the robot enters the autonomous period and it will automatically be scheduled. If you have tested commands as discussed above, this should simply be a matter of choosing which command should run.

Select the robot at the root of the RobotBuilder project, then edit the Autonomous Command property to choose the command to run. It's that simple!

For more information see: Setting the default autonomous command

Generating code for the program



Property	Value
Name	MyRobot
Autonomous Command	Shoot
Team Number	0
Java Project	Click to Select
Use Default Java Package	✓
Java Package	org.usfirst.frc0000
C++ Windriver Workspace	Click to Select
Export Subsystems	✓
Export Commands	✓
Wiring File	Click to Select

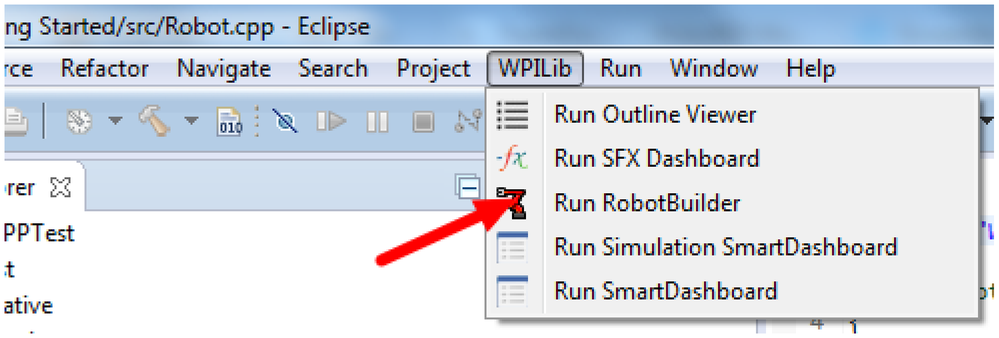
At any point in the process outlined above you can have RobotBuilder generate a C++ or Java program that will represent the project you have created. This is done by specifying the location of the project in the project properties (1), then clicking the appropriate toolbar button to generate the code.

For more information see: NEEDS UPDATE

Starting RobotBuilder

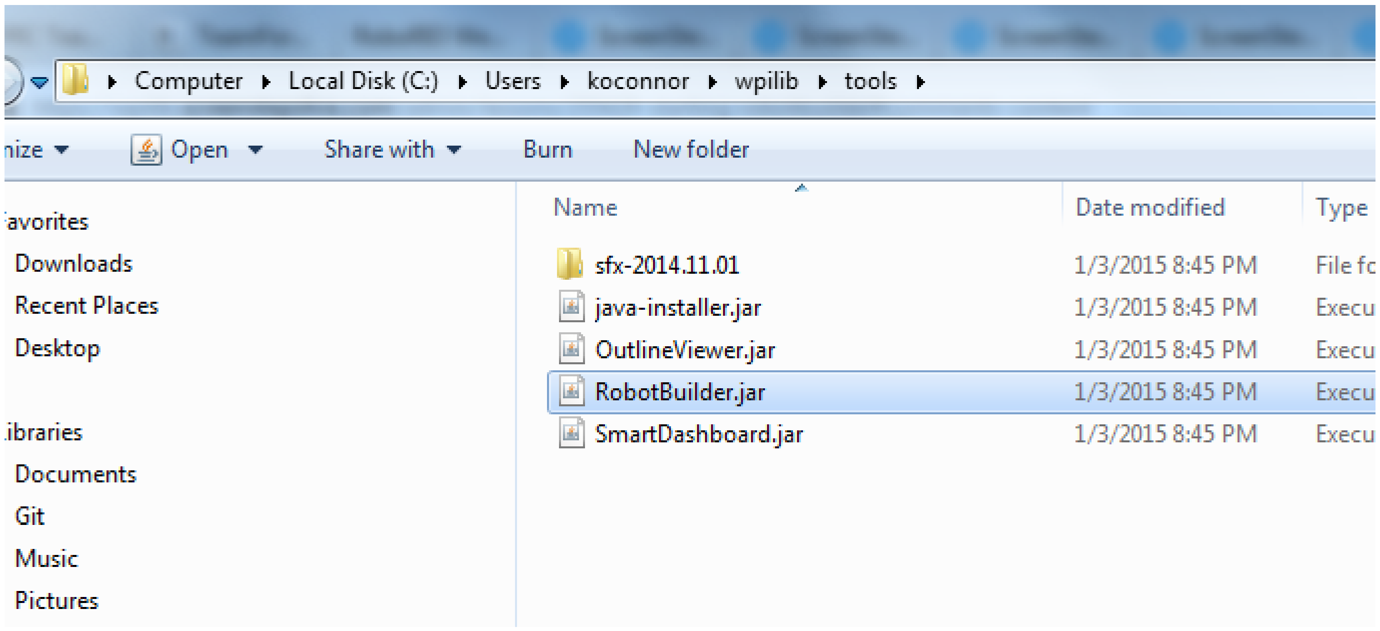
RobotBuilder is a Java program and as such should be able to run on any platform that is supported by Java. We have been running RobotBuilder on Mac OS X, Windows 7, and various versions of Linux successfully.

Option 1 - Starting RobotBuilder from Eclipse


ng Started/src/Robot.cpp - Eclipse
rce Refactor Navigate Search Project WPILib Run Window Help
Run Outline Viewer
Run SFX Dashboard
Run RobotBuilder
Run Simulation SmartDashboard
Run SmartDashboard
PPTest

RobotBuilder can be launched directly from Eclipse by clicking on the WPILib menu and selecting Run RobotBuilder.

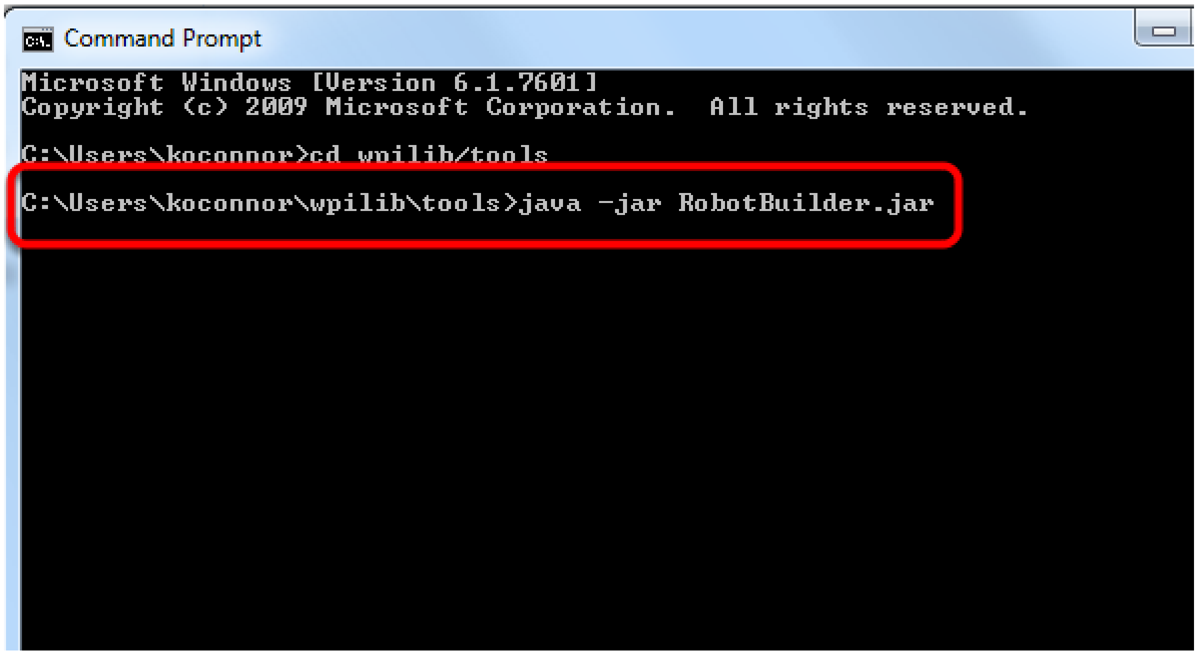
Option 2 - Locating the RobotBuilder .jar file


Computer ▸ Local Disk (C:) ▸ Users ▸ koconnor ▸ wpilib ▸ tools ▸
nize Open Share with Burn New folder
avorites
Downloads
Recent Places
Desktop
ibraries
Documents
Git
Music
Pictures

Name	Date modified	Type
sfx-2014.11.01	1/3/2015 8:45 PM	File fc
java-installer.jar	1/3/2015 8:45 PM	Execu
OutlineViewer.jar	1/3/2015 8:45 PM	Execu
RobotBuilder.jar	1/3/2015 8:45 PM	Execu
SmartDashboard.jar	1/3/2015 8:45 PM	Execu

RobotBuilder is shipped as a .jar file (Java archive) included with the Eclipse plugins. In most cases you can simply double-click on the file in a graphical file browser for your operating system and it will start. For both languages, RobotBuilder is located in the wpilib/tools directory. This directory is in your user home directory, usually something like /Users/<username>/wpilib.

Option 3- Starting RobotBuilder from the command line

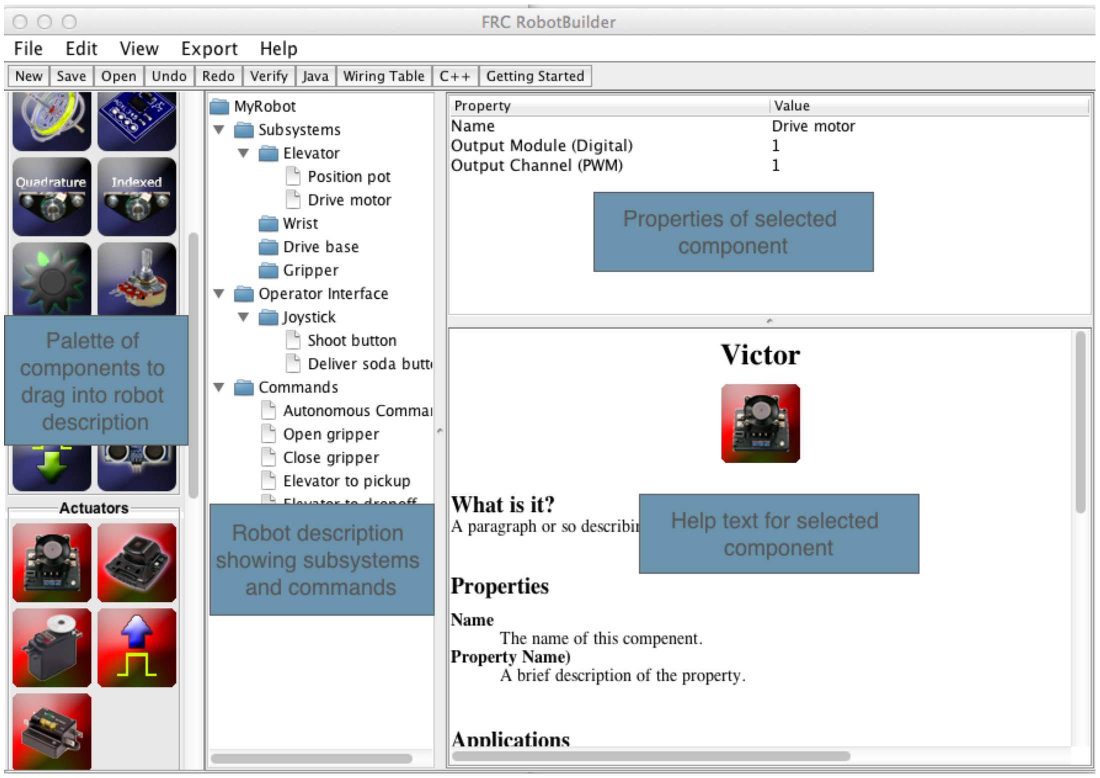


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\koconnor>cd wpilib/tools

C:\Users\koconnor\wpilib\tools>java -jar RobotBuilder.jar
```

In some cases Java and your file browser might not be properly configured to run the .jar file by double-clicking. Simply type "java -jar RobotBuilder.jar" from the directory that contains RobotBuilder.

The RobotBuilder user interface

RobotBuilder User Interface



Property	Value
Name	Drive motor
Output Module (Digital)	1
Output Channel (PWM)	1

Victor

What is it?
A paragraph or so describi

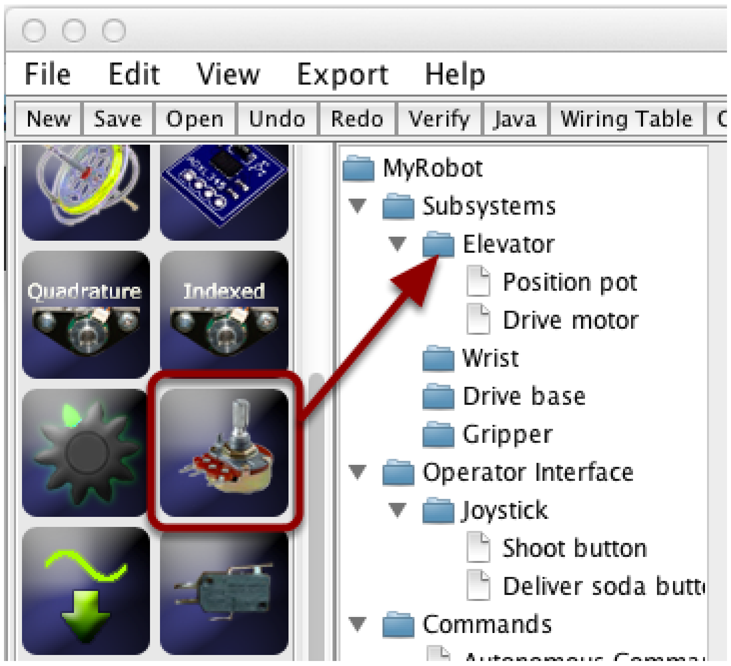
Properties

Name
The name of this compenent.
Property Name)
A brief description of the property.

Applications

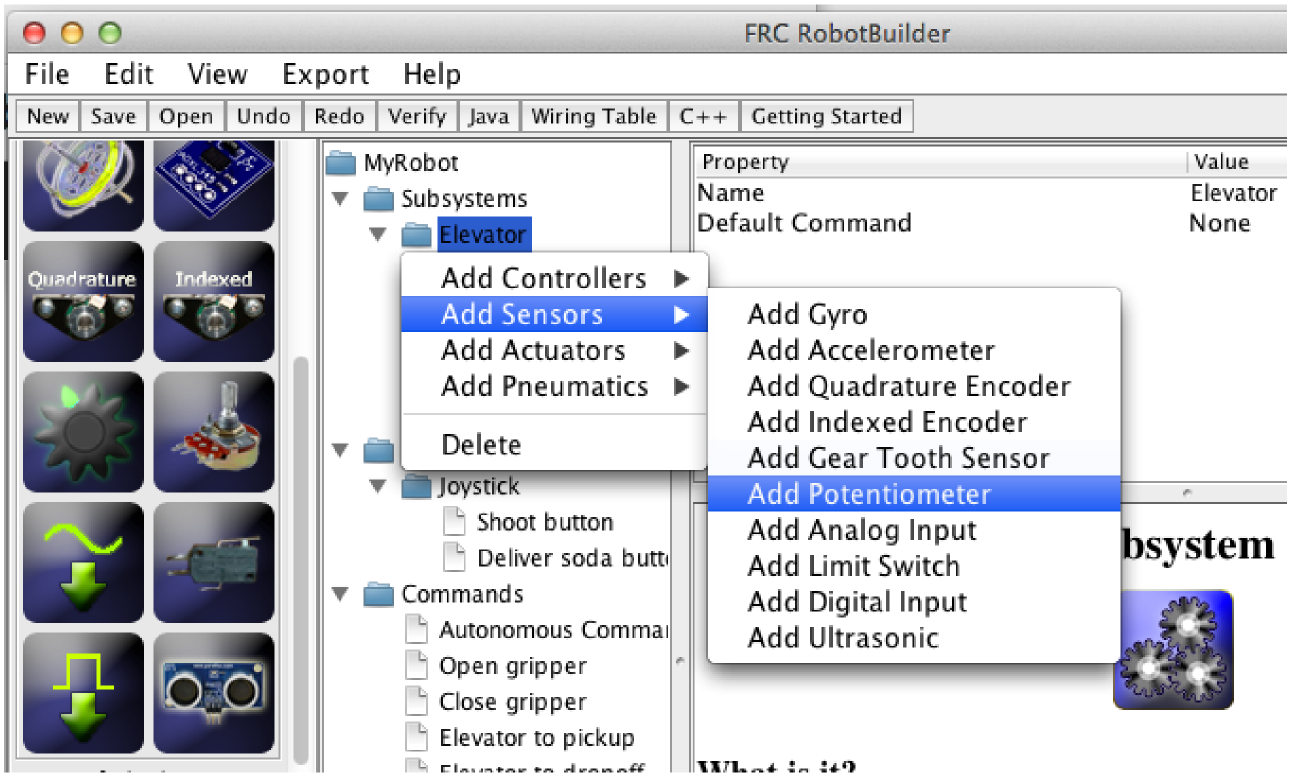
RobotBuilder has a user interface designed for rapid development of robot programs. Almost all operations are performed by drag and drop or selecting options from drop-down lists.

Dragging items from the palette to the robot description



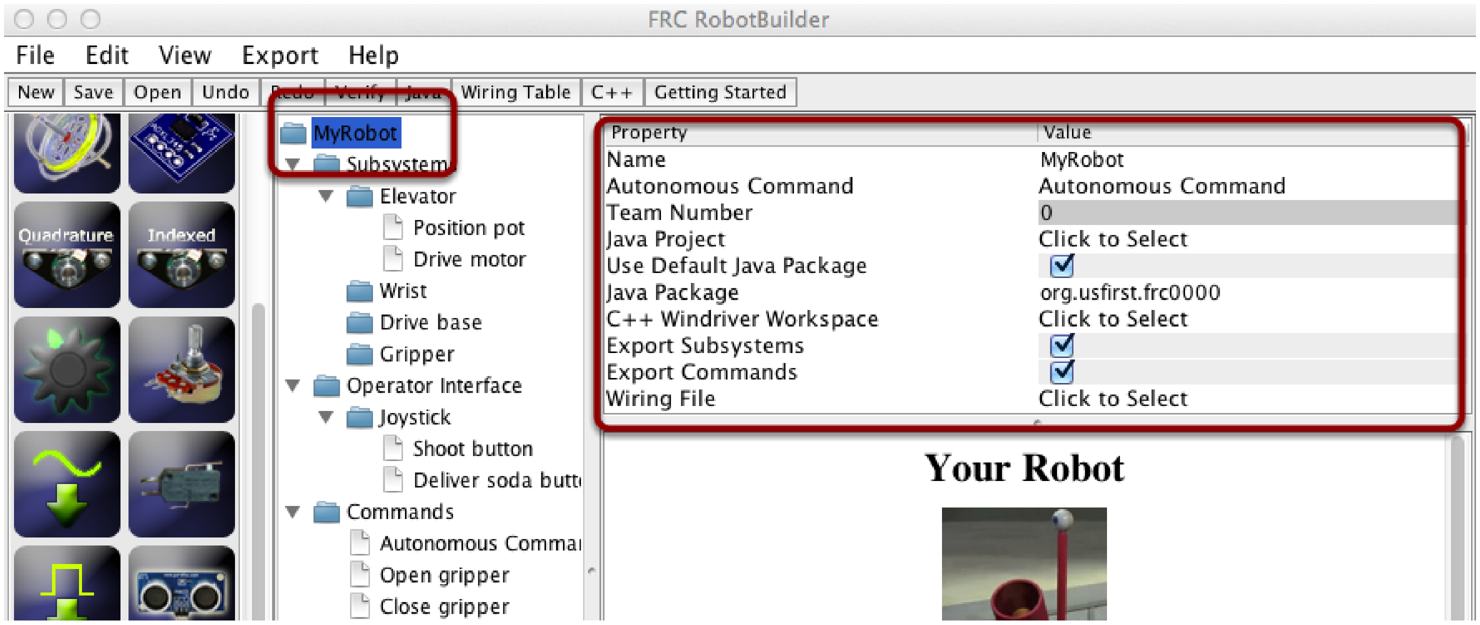
You can drag items from the palette to the robot description by starting the drag on the palette item and ending on the container where you would like the item to be located. In this example, dropping a potentiometer to the Elevator subsystem.

Adding components using the right-click context menu



A shortcut method of adding items to the robot description is to right-click on the container object (Elevator) and select the item that should be added (Potentiometer). This is identical to using drag and drop but might be easier for some people.

Editing properties of robot desciption items

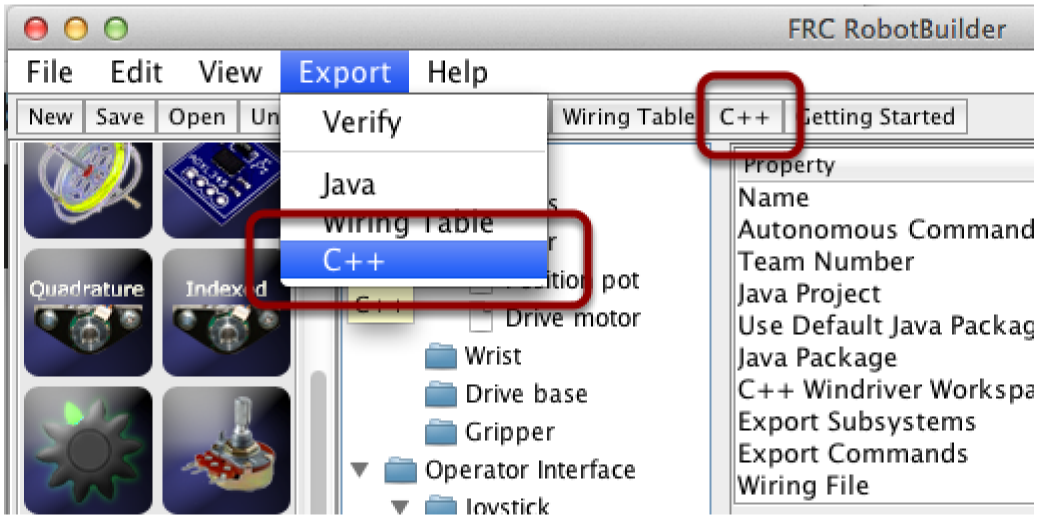

FRC RobotBuilder
File Edit View Export Help
New Save Open Undo Redo Verify Java Wiring Table C++ Getting Started
Quadrature
Indexed
Your Robot

MyRobot
- Subsystems
  - Elevator
    - Position pot
    - Drive motor
  - Wrist
  - Drive base
  - Gripper
- Operator Interface
  - Joystick
    - Shoot button
    - Deliver soda butt
- Commands
  - Autonomous Commai
  - Open gripper
  - Close gripper

Property	Value
Name	MyRobot
Autonomous Command	Autonomous Command
Team Number	0
Java Project	Click to Select
Use Default Java Package	✓
Java Package	org.usfirst.frc0000
C++ Windriver Workspace	Click to Select
Export Subsystems	✓
Export Commands	✓
Wiring File	Click to Select

The properties for a selected item will appear in the properties viewer. The properties can be edited by selecting the value in the right hand column.

Using the menu system

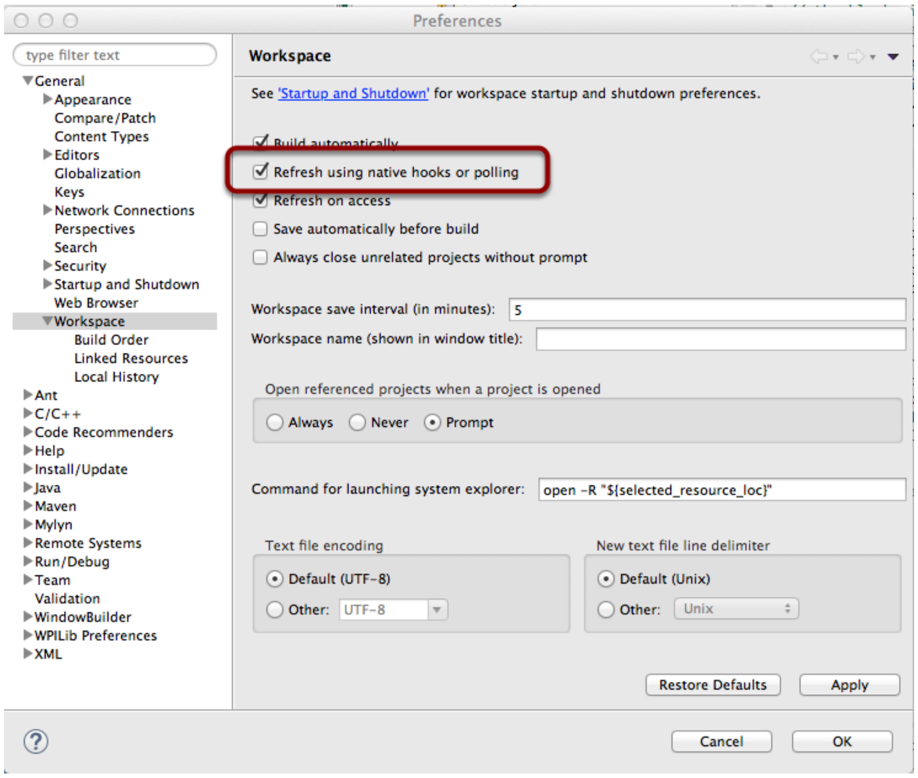


Operations for RobotBuilder can either be selected through the menu system or the equivalent item (if it is available) from the toolbar.

Setting up the robot project

The RobotBuilder program has some default properties that need to be set up so the generated program and other generated files work properly. This setup information is stored in the properties for robot description (the first line).

Using RobotBuilder with Eclipse



Workspace

See 'Startup and Shutdown' for workspace startup and shutdown preferences.

- [✓] Build automatically
- [✓] Refresh using native hooks or polling
- [✓] Refresh on access
- [ ] Save automatically before build
- [ ] Always close unrelated projects without prompt

Workspace save interval (in minutes): 5

Workspace name (shown in window title):

Open referenced projects when a project is opened: Always Never [✓ Prompt]

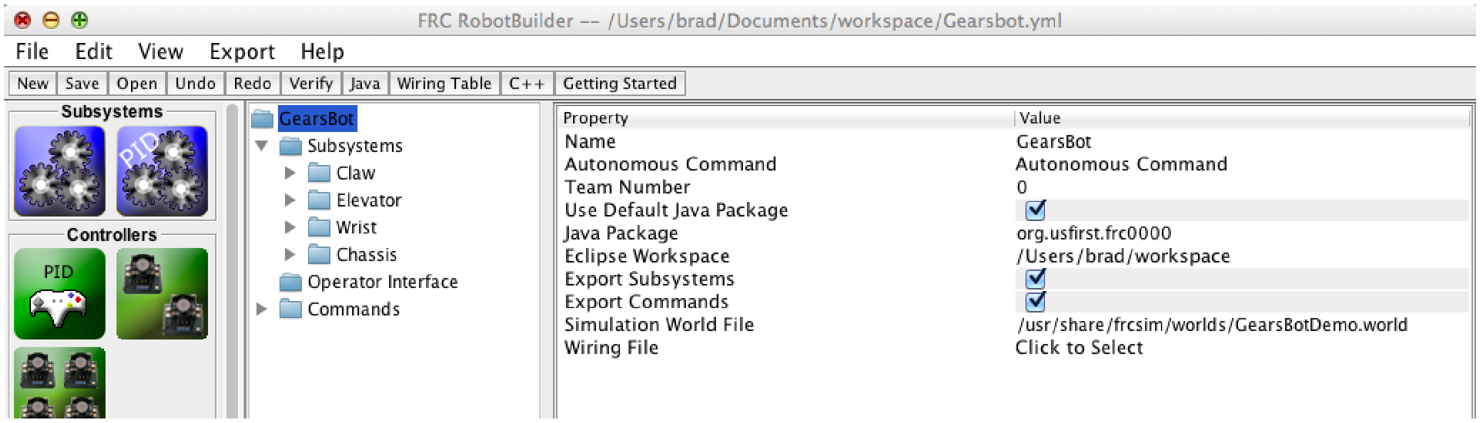
Command for launching system explorer: open -R "${selected_resource_loc}"

Text file encoding: [✓ Default (UTF-8)] Other: UTF-8

New text file line delimiter: [✓ Default (Unix)] Other: Unix

When using RobotBuilder files are saved inside the RobotBuilder interface and also from the Eclipse interface. It is important to keep the files in sync. To get eclipse to automatically notice that RobotBuilder has saved new versions of the project files and automatically load them check the "Refresh using native hooks or polling" option in the Workspace preferences under General as shown here. With this option checked, the source files open in the Eclipse editor will automatically be refreshed when RobotBuilder generates new files.

Robot project properties



Property	Value
Name	GearsBot
Autonomous Command	Autonomous Command
Team Number	0
Use Default Java Package	✓
Java Package	org.usfirst.frc0000
Eclipse Workspace	/Users/brad/workspace
Export Subsystems	✓
Export Commands	✓
Simulation World File	/usr/share/frcsim/worlds/GearsBotDemo.world
Wiring File	Click to Select

The properties that describe the robot are:

Name - The name of the robot project that is created

Autnomous Command - the command that will run by default when the program is placed in autonomous mode

Team Number - the team number is used for creating the package names

Java Project - The folder that the java project is generated into when Export to Java is selected

Use Default Java Package - If checked RobotBuilder will use the default package (org.usfirst.frc####). Otherwise you can specify a custom package name to be used.

Java Package - The name of the generated Java package used when generating the project code

Eclipse Workspace - The location of the Eclipse workspace that your project should be saved to

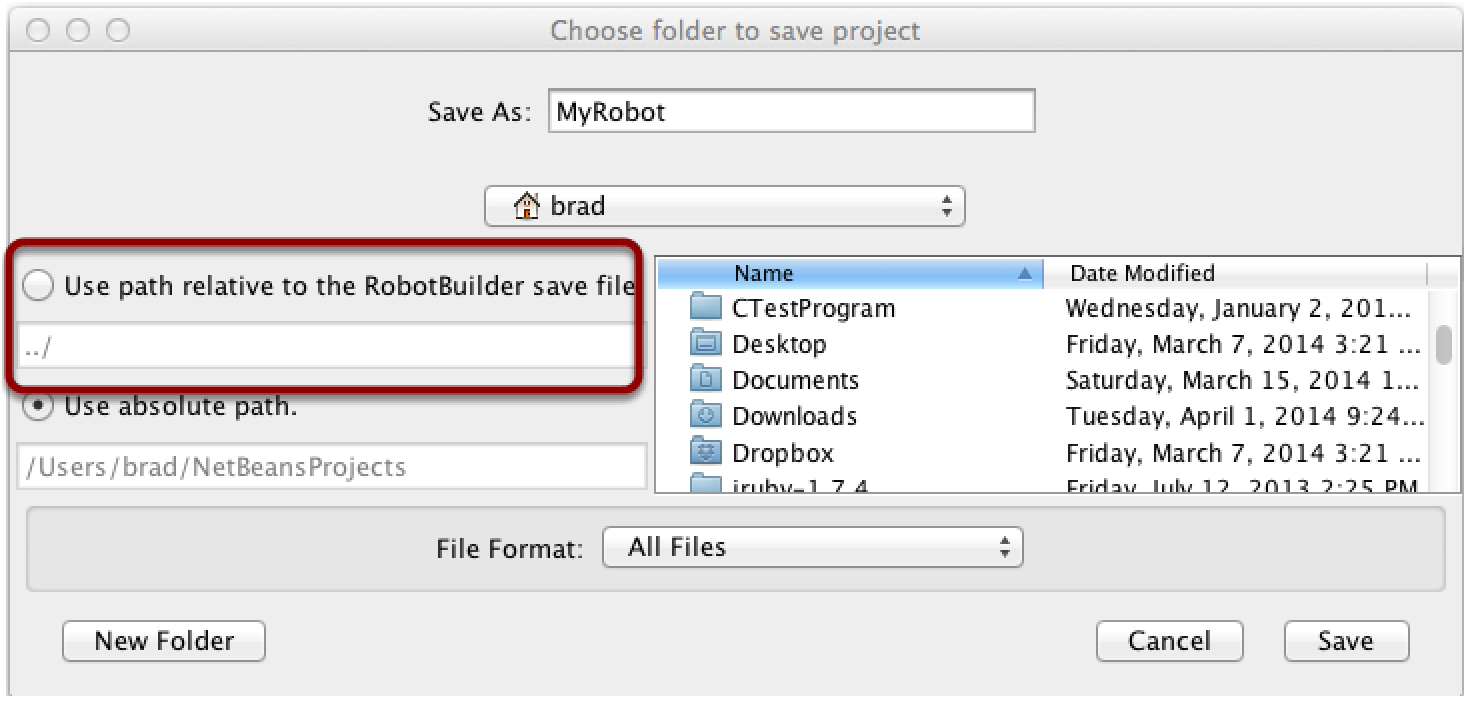
Export Subsystems - Checked if RobotBuilder should export the Subsystem classes from your project

Export Commands - Checked if RobotBuilder should export the Command classes from your project

Simulation World File - The World File that is used for simulation of your robot project

Wiring File - the location of the html file that contains the wiring diagram for your robot

Using source control with the RobotBuilder project


Choose folder to save project

Save As: MyRobot

brad

( ) Use path relative to the RobotBuilder save file

../

(•) Use absolute path.

/Users/brad/NetBeansProjects

Name	Date Modified
CTestProgram	Wednesday, January 2, 201...
Desktop	Friday, March 7, 2014 3:21 ...
Documents	Saturday, March 15, 2014 1...
Downloads	Tuesday, April 1, 2014 9:24...
Dropbox	Friday, March 7, 2014 3:21 ...
jruby-1.7.4	Friday, July 12, 2013 2:25 PM

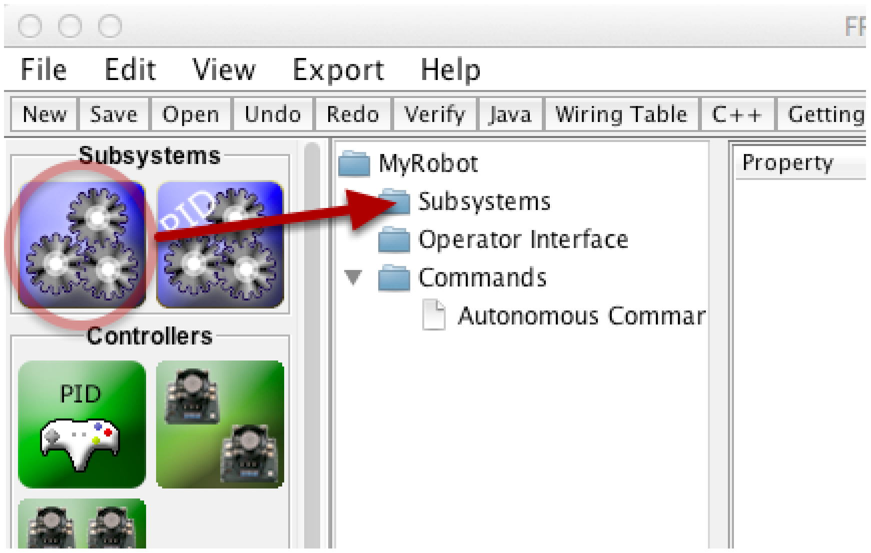
File Format: All Files

When using source control the project will typically be used on a number of computers and the path to the project directory might be different from one users computer to another. If the RobotBuilder project file is stored using an absolute path, it will typically contain the user name and won't be usable across multiple computers. To make this work, select "relative path" and specify the path as an directory offset from the project files. In the above example, the project file is stored in the folder just above the project files in the file hierarchy. In this case, the user name is not part of the path and it will be portable across all of your computers.

Creating a subsystem

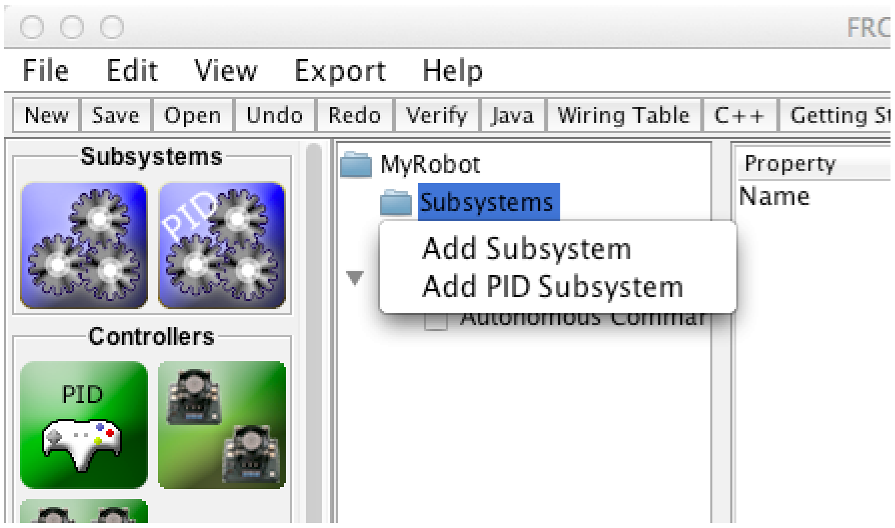
Subsystems are classes that encapsulate (or contain) all the data and code that make a subsystem on your robot operate. The first step in creating a robot program with the RobotBuilder is to identify and create all the subsystems on the robot. Examples of subsystems are grippers, ball collectors, the drive base, elevators, arms, etc. Each subsystem contains all the sensors and actuators that are used to make it work. For example, an elevator might have a Jaguar speed controller and a potentiometer to provide feedback of the robot position.

Creating a subsystem by dragging from the palette



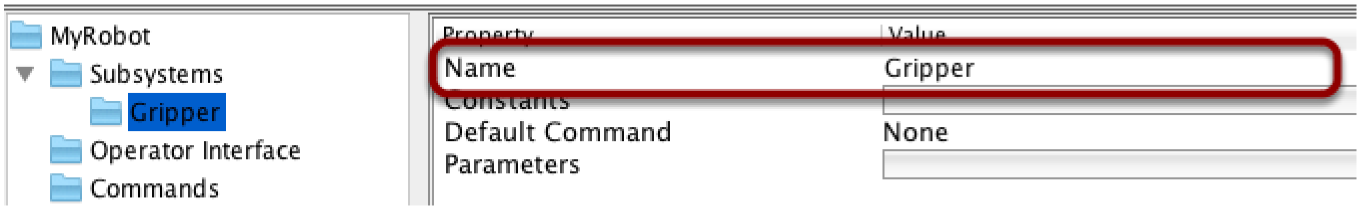
Drag the subsystem icon from the palette to the Subsystems folder in the robot description to create a subsystem class.

Creating a subsystem by using the context menu on the Subsystem folder



Right-click on the Subsystem folder in the robot description to add a subsystem to that folder.

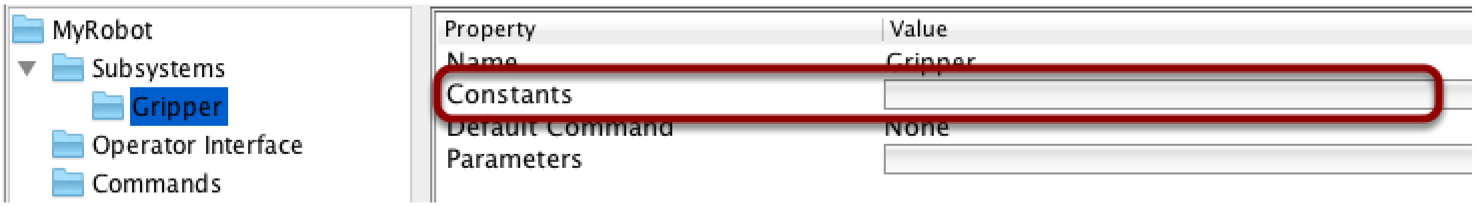
Name the subsystem



Property	Value
Name	Gripper
Constants	
Default Command	None
Parameters	

After creating the subsystem by either dragging or using the context menu as described above, simply type the name you would like to give the subsystem. The name can be multiple words separated by spaces, RobotBuilder will concatenate the words to make a proper Java or C++ class name for you.

Adding constants (2016 only)

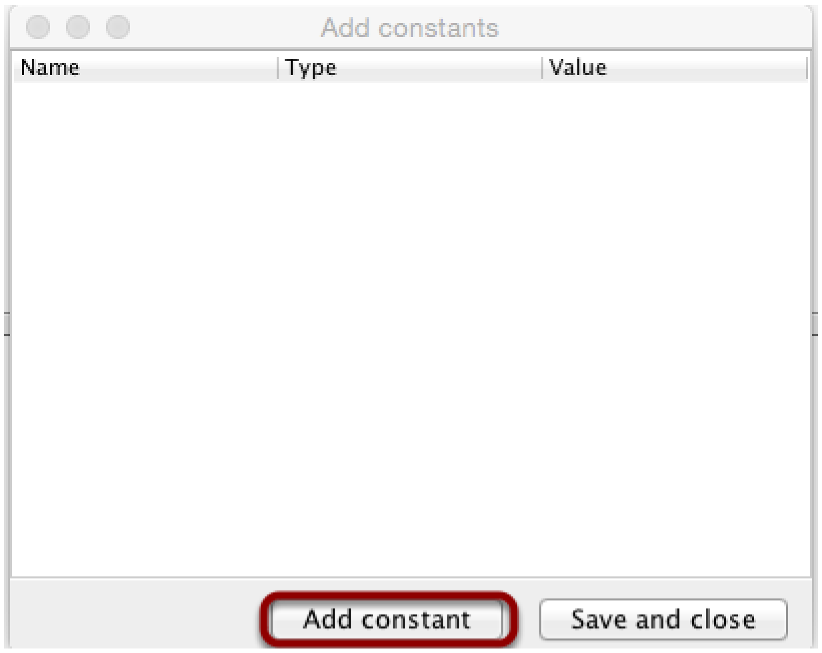


Property	Value
Name	Gripper
Constants	
Default Command	None
Parameters	

Constants are very useful to reduce the amount of magic numbers in your code. In subsystems, they can be used to keep track of certain values, such as sensor values for specific heights of an elevator, or the speed at which to drive the robot.

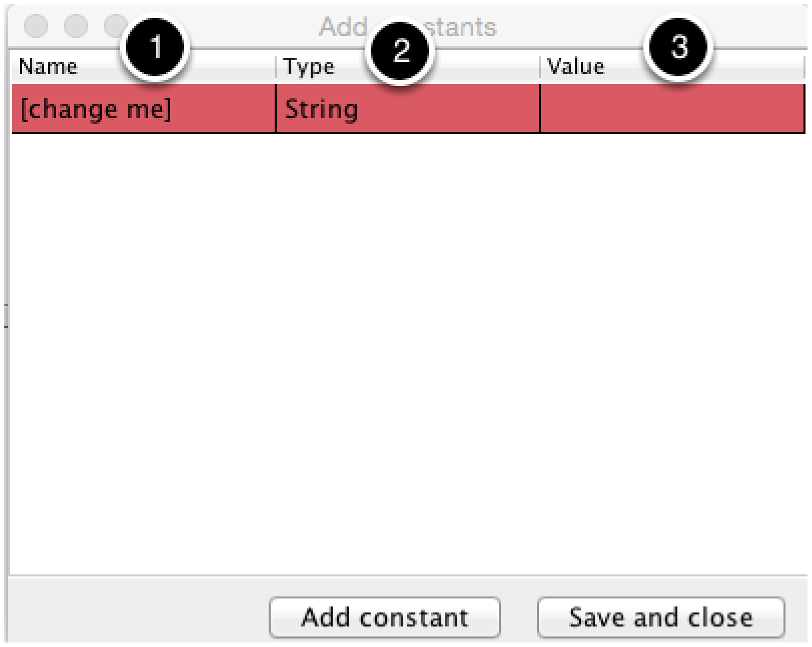
By default, there will be no constants in a subsystem. Press the button next to "Constants" to open a dialog to create some.

Creating constants (2016 only)



The constants table will be empty at first. Press "Add constant" to add one.

Add constants (2016 only)

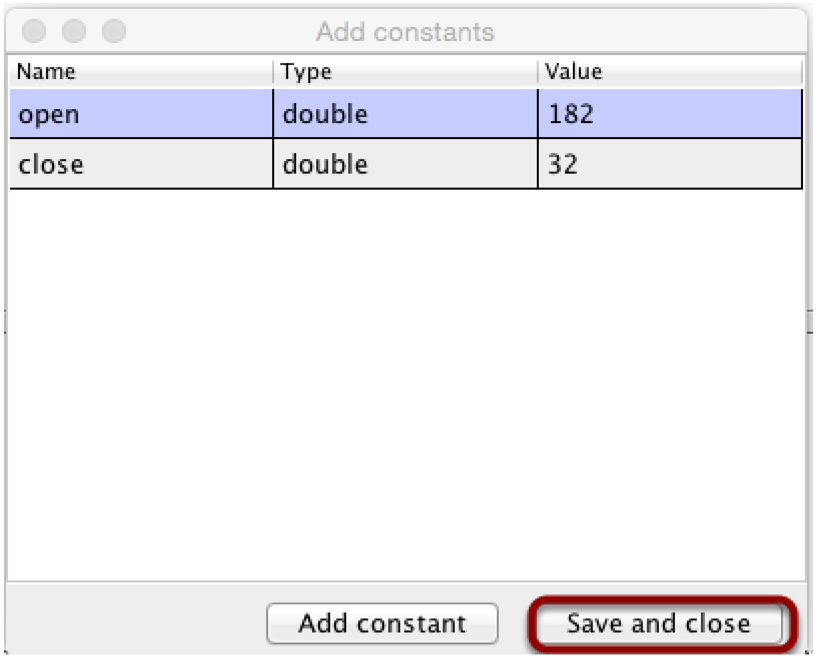


Name	Type	Value
[change me]	String	

1. The name of the constant. Change this to something descriptive. In this example of a gripper, some good constants might be "open" and "close".
2. The type of the constant. This will most likely be a double, but you can choose from one of: String, double, int, long int, boolean, or byte.
3. The value of the constant.

If a constant is valid, it will be exported to the subsystem class as a public static final/public const static variable depending on the language. If it isn't, that constant will be highlighted red to show there's something wrong with it. Common problems are an invalid Java/C++ variable name (such as having spaces or non-letter characters in the name) or having non-numeral characters in the value field of a numeric constant.

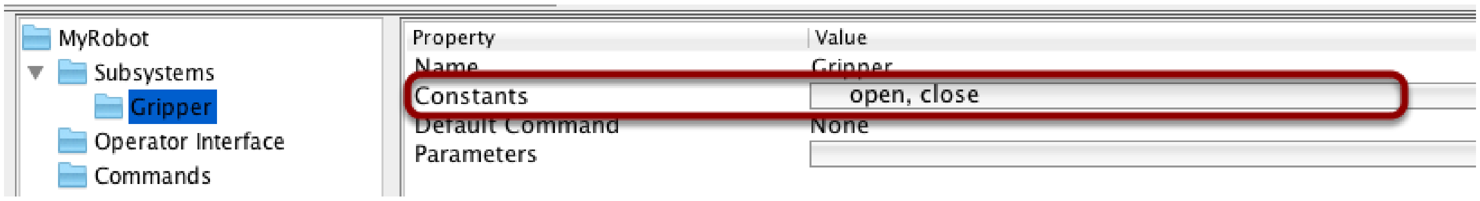
Saving constants (2016 only)



Name	Type	Value
open	double	182
close	double	32

After adding constants and setting their values, just press "Save and close" to save the constants and close the dialog. If you don't want to save, press the exit button on the top of the window.

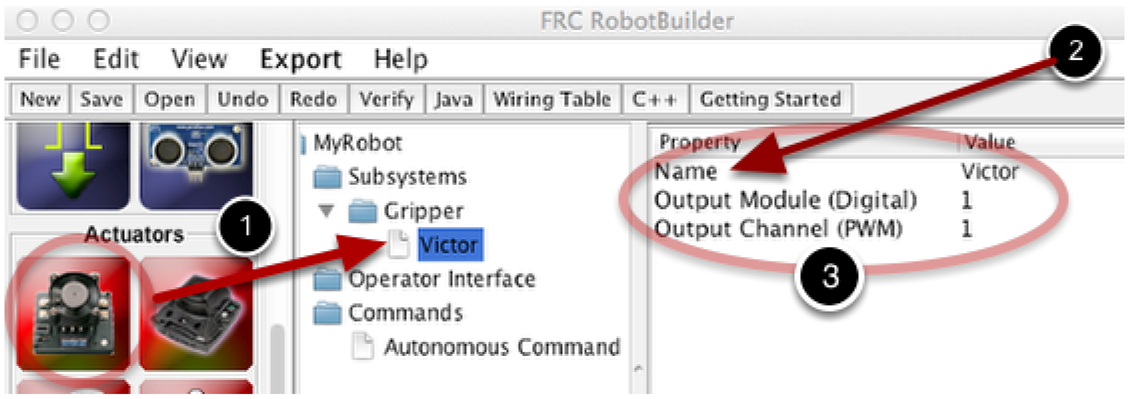
After saving (2016 only)



Property	Value
Name	Gripper
Constants	open, close
Default Command	None
Parameters	

After saving constants, the names will appear in the "Constants" button in the subsystem properties.

Drag actuators and sensors into the subsystem



Property	Value
Name	Victor
Output Module (Digital)	1
Output Channel (PWM)	1

There are two steps to adding components to a subsystem:

1. Drag actuators or sensors from the palette into the subsystem as required.
2. Give the actuator or sensor a meaningful name
3. Edit the properties such as module numbers and channel numbers for each item in the subsystem.

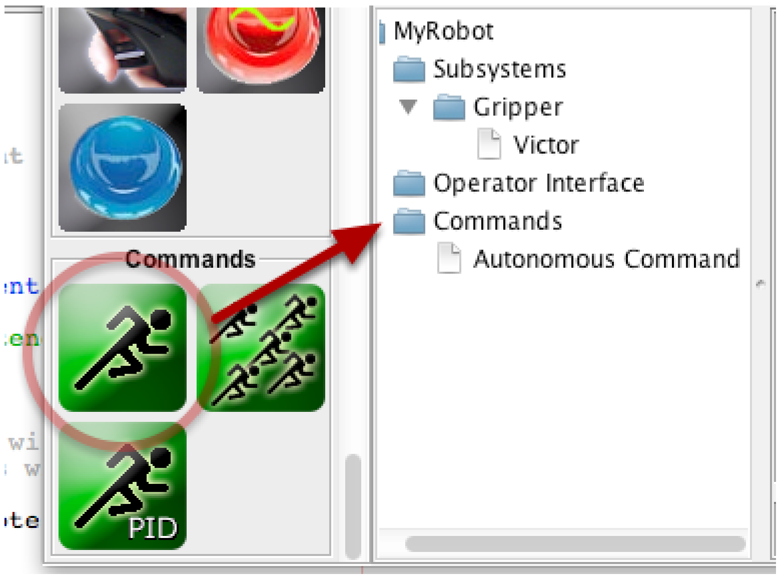
RobotBuilder will automatically use incrementing channel numbers for each module on the robot. If you haven't yet wired the robot you can just let RobotBuilder assign unique channel numbers for each sensor or actuator and wire the robot according to the generating wiring table.

This just creates the subsystem in RobotBuilder, and will subsequently generate skeleton code for the subsystem. To make it actually operate your robot please refer to: Writing the code for a subsystem in Java or Writing the code for a subsystem in C++.

Creating a command

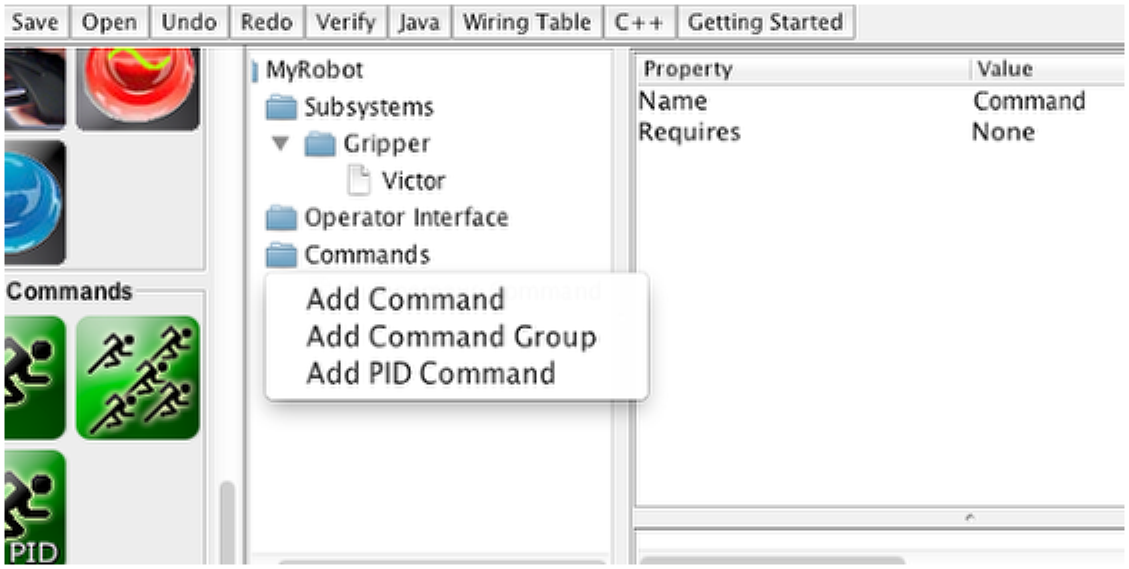
Commands are classes you create that provide behaviors or actions for your subsystems. The subsystem class should set the operation of the subsystem, like MoveElevator to start the elevator moving, or ElevatorToSetPoint to set the elevator's PID setpoint. The commands initiate the subsystem operation and keep track of when it is finished.

Drag a command to the robot description Commands folder



Simple commands can be dragged from the palette to the robot description. The command will be created under the Commands folder.

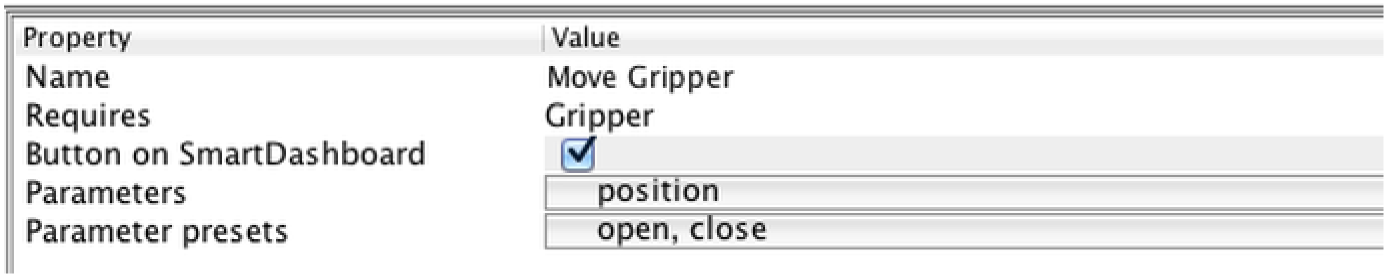
Creating commands using the context menu



Property	Value
Name	Command
Requires	None

You can also create commands using the right-click context menu on the Command folder in the robot description.

Configure the command



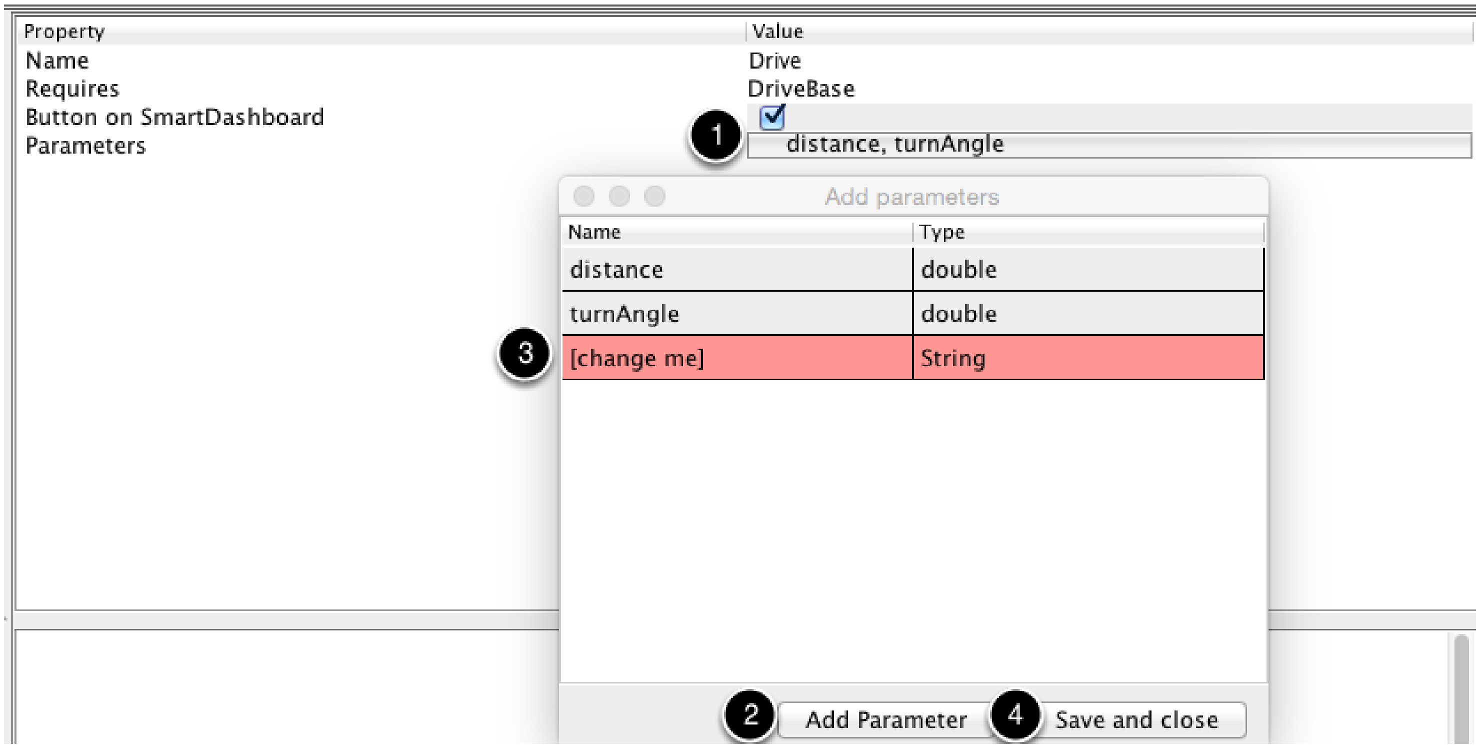
Property	Value
Name	Move Gripper
Requires	Gripper
Button on SmartDashboard	✓
Parameters	position
Parameter presets	open, close

1. Name the command with something meaningful that describes what the command will do. Commands should be named as if they were in code, although there can be spaces between words.
2. Set the subsystem that is used by this command. When this command is scheduled, it will automatically stop any command currently running that also requires this command. If a command to open the gripper is currently running (requiring the gripper subsystem) and the close gripper command is scheduled, it will immediately stop opening and start closing.
3. Tell RobotBuilder if it should create buttons on the SmartDashboard for the command. A button will be created for each parameter preset.

4. Set the parameters this command takes. A single command with parameters can do the same thing as two or more commands that do not take parameters. For example, "Drive Forward", "Drive Backward", and "Drive Distance" commands can be consolidated into a single command that takes values for direction and distance.
5. Set presets for parameters. These can be used elsewhere in RobotBuilder when using the command, such as binding it to a joystick button or setting the default command for a subsystem.

Setpoint commands come with a single parameter ('setpoint', of type double); parameters cannot be added, edited, or deleted for setpoint commands.

Adding and editing parameters



Property	Value
Name	Drive
Requires	DriveBase
Button on SmartDashboard	✓
Parameters	distance, turnAngle

Add parameters

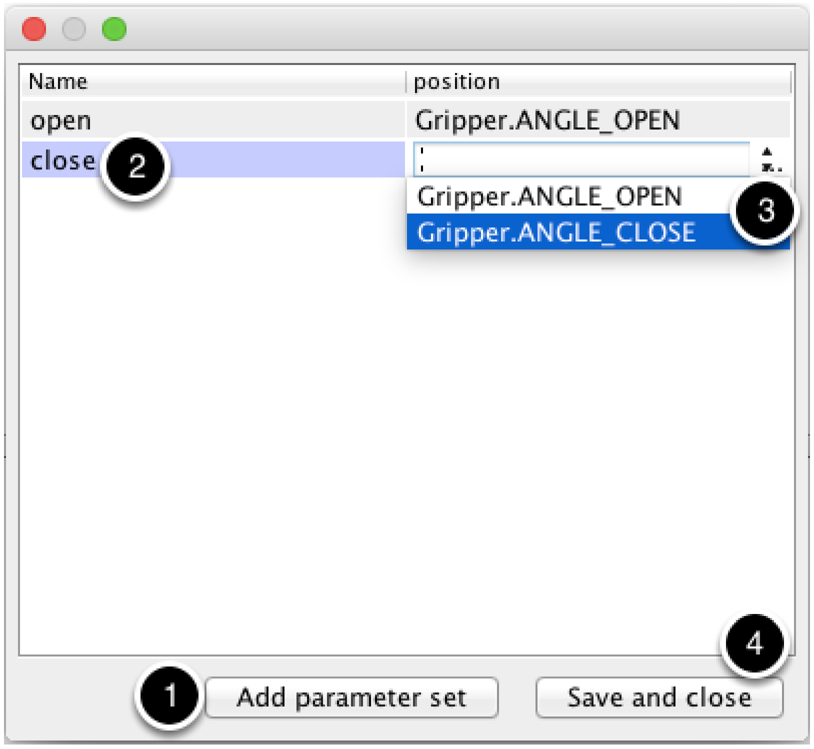
Name	Type
distance	double
turnAngle	double
[change me]	String

To add or edit parameters:

1. Click the button in the "Value" column of the property table
2. Press the "Add Parameter" button to add a parameter
3. A parameter that has just been added. The name defaults to "[change me]" and the type defaults to String. The default name is invalid, so you will have to change it before exporting. Double click the "Name" cell to start changing the name; double click the "Type" cell to select the type.
4. Save and close button will save all changes and close the window.

Rows can be reordered simply by dragging, and can be deleted by selecting them and pressing delete or backspace.

Adding and editing parameter presets

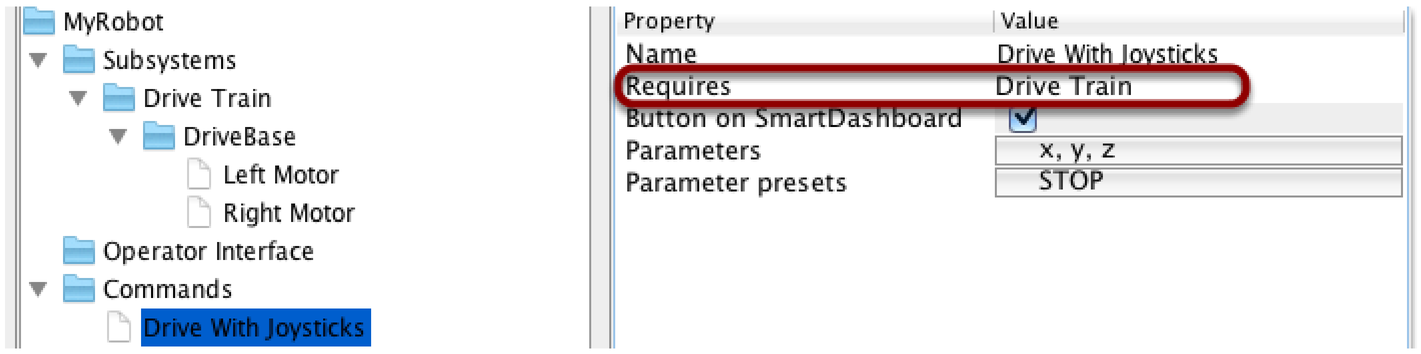


1. Click "Add parameter set" to add a new preset.
2. Change the name of the preset to something descriptive. The presets in this example are for opening and closing the gripper subsystem.
3. Change the value of the parameter(s) for the preset. You can either type a value in (e.g. "3.14") or select from constants defined in the subsystem that the command requires. Note that the type of the constant has to be the same type as the parameter -- you can't have an int-type constant be passed to a double-type parameter, for example
4. Click "Save and close" to save changes and exit the dialog; to exit without saving, press the exit button in the top bar of the window.

Setting the default command for a subsystem

Once you have some commands created, you can set one of them to be the default command for a subsystem. Default commands run automatically when nothing else is running that requires that subsystem. A good example is having a drive train subsystem with a default command that reads joysticks. That way, whenever the robot program isn't running other commands to operate the drive train under program control, it will operate with joysticks.

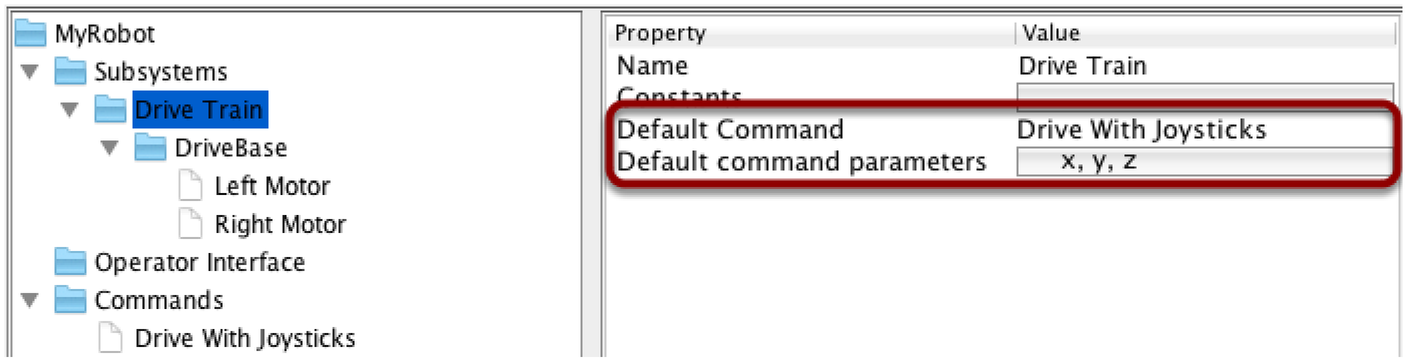
Create the command that should be the default for a subsystem



Property	Value
Name	Drive With Joysticks
Requires	Drive Train
Button on SmartDashboard	✓
Parameters	x, y, z
Parameter presets	STOP

Here a command is created called "Drive with joysticks" that would read the joystick values and set them in the Drive Train subsystem. This is what the Drive Train should be doing if it isn't being asked to do anything else.

Set the command as the default for the subsystem



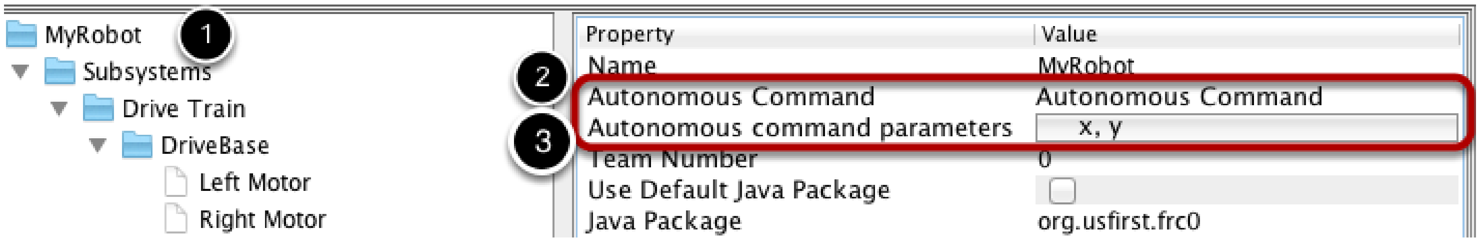
Property	Value
Name	Drive Train
Constants	
Default Command	Drive With Joysticks
Default command parameters	x, y, z

The "Drive with joysticks" command is set as the default command for the Drive Train subsystem. If it takes parameters, a row will appear in the table to set the parameters.

You can also set the default Autonomous command, that is the command that runs when the robot enters the Autonomous state.

Setting the default autonomous command

Since a command is simply one or more actions (behaviors) that the robot performs, it makes sense to describe the autonomous operation of a robot as a command. While it could be a single command, it is more likely going to be a command group (a group of commands that happen together).



Property	Value
Name	MyRobot
Autonomous Command	Autonomous Command
Autonomous command parameters	x, y
Team Number	0
Use Default Java Package	☐
Java Package	org.usfirst.frc0

To designate a command that runs when the robot starts during the autonomous period of a competition:

1. Select the robot in the robot program description
2. Fill in the Autonomous command field with the command that should run when the robot is placed in autonomous mode. This is a drop-down field and will give you the option to select any command that has been defined.
3. Set the parameters the command takes, if any

When the robot is put into autonomous mode, the defined Autonomous command will be scheduled.

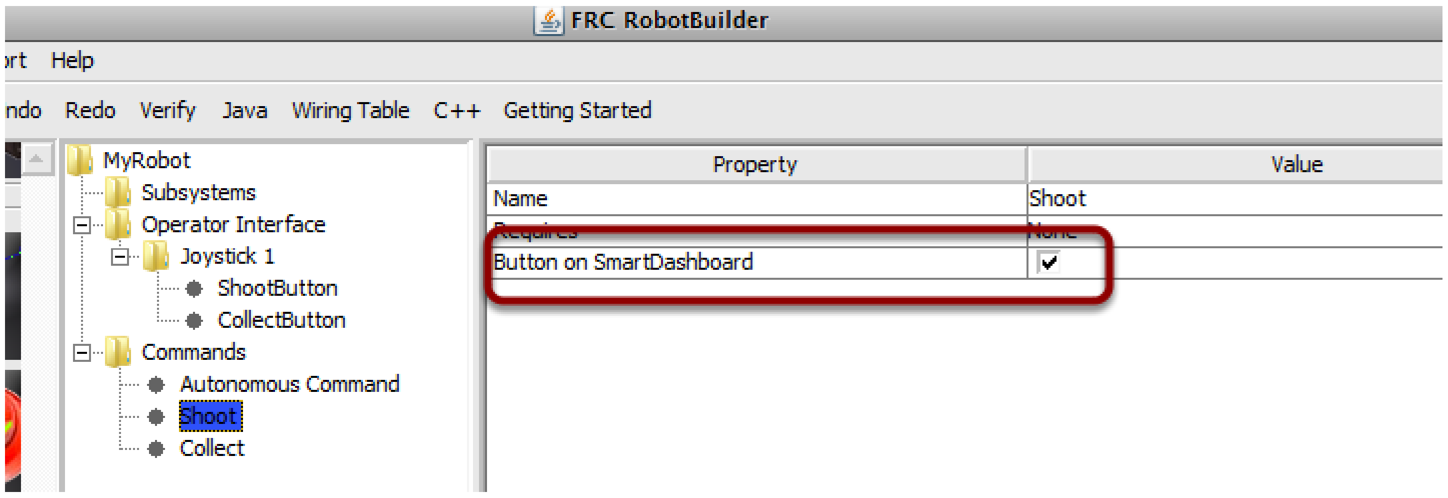
See: Creating a command and Setting the default command for a subsystem.

Adding a button to SmartDashboard to test a command

Commands are easily tested by adding a button to the SmartDashboard to trigger the command. In this way, no integration with the rest of the robot program is necessary and commands can easily be independently tested. This is the easiest way to verify commands since with a single line of code in your program, a button can be created on the SmartDashboard that will run the command. These buttons can then be left in place to verify subsystems and command operations in the future.

This has the added benefit of accommodating multiple programmers, each writing commands. As the code is checked into the main robot project, the commands can be individually tested.

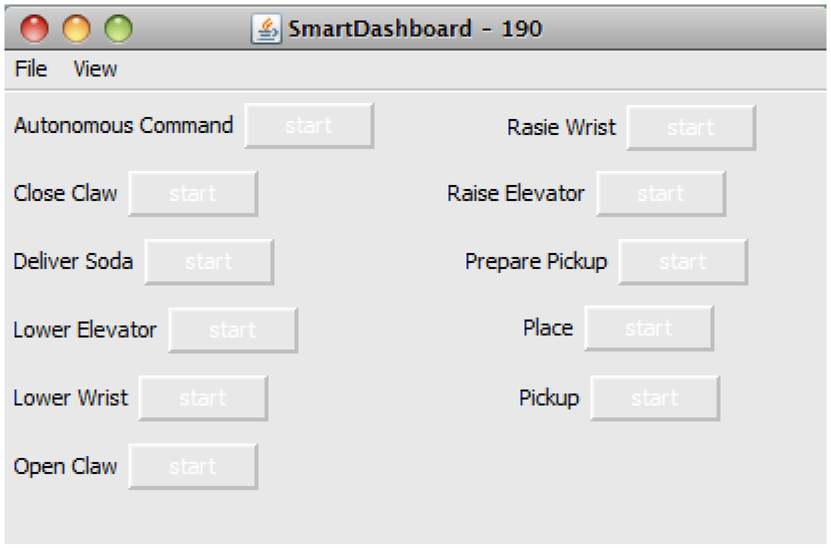
Creating the button on the SmartDashboard



Property	Value
Name	Shoot
Requires	None
Button on SmartDashboard	✓

The button is created on the SmartDashboard by putting an instance of the command from the robot program to the dashboard. This is such a common operation that it has been added to RobotBuilder as a checkbox. When writing your commands, be sure that the box is checked, and buttons will be automatically generated for you.

How to operate the buttons on the SmartDashboard



The buttons will be generated automatically and will appear on the dashboard screen. You can put the SmartDashboard into edit mode, and the buttons can then be rearranged along with other values that are being generated. In this example there are a number of commands, each with an associated button for testing. The button is labeled "Start" and pressing it will run the command. As soon as it is pressed, the label changes to "Cancel" and pressing it will interrupt the command causing the Interrupted() method to be called.

Adding commands manually

```
SmartDashboard::PutData("Autonomous Command", new AutonomousCommand());
SmartDashboard::PutData("Open Claw", new OpenClaw());
SmartDashboard::PutData("Close Claw", new CloseClaw());
SmartDashboard::PutData("Lower Wrist", new LowerWrist());
SmartDashboard::PutData("Rasie Wrist", new RasieWrist());
SmartDashboard::PutData("Lower Elevator", new LowerElevator());
SmartDashboard::PutData("Raise Elevator", new RaiseElevator());
SmartDashboard::PutData("Prepare Pickup", new PreparePickup());
SmartDashboard::PutData("Pickup", new Pickup());
SmartDashboard::PutData("Place", new Place());
SmartDashboard::PutData("Deliver Soda", new DeliverSoda());
```

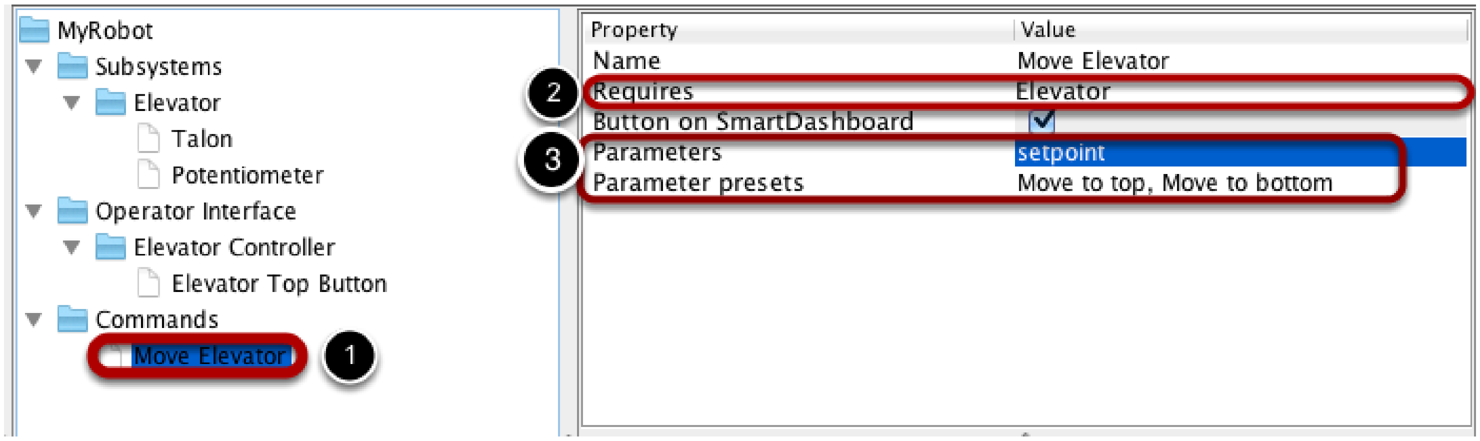
Commands can be added to the SmartDashboard manually by writing the code yourself. This is done by passing instances of the command to the PutData method along with the name that should be associated with the button on the SmartDashboard. These instances are scheduled whenever the button is pressed. The result is exactly the same as RobotBuilder generated code, although clicking the checkbox in RobotBuilder is much easier than writing all the code by hand.

Connecting the operator interface to a command

Commands handle the behaviors for your robot. The command starts a subsystem to some operating mode like raising and elevator and continues running until it reaches some setpoint or timeout. The command then handles waiting for the subsystem to finish. That way commands can run in sequence to develop more complex behaviors.

RobotBuilder will also generate code to schedule a command to run whenever a button on your operator interface is pressed. You can also write code to run a command when a particular trigger condition has happened.

Set up a command to be run by the button press

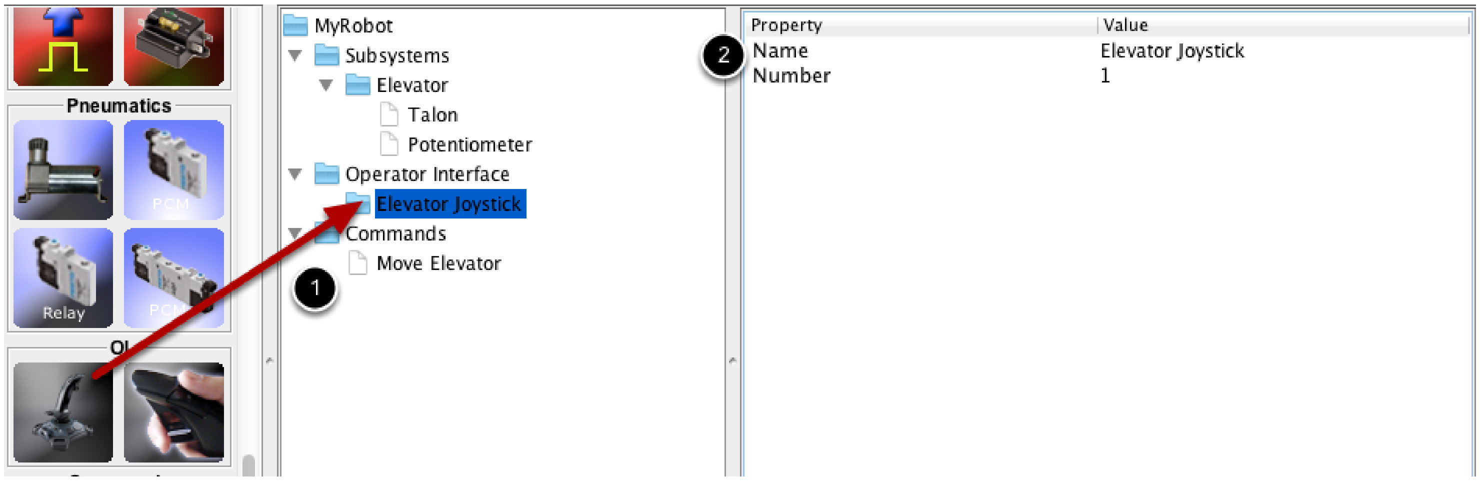


Property	Value
Name	Move Elevator
Requires	Elevator
Button on SmartDashboard	✓
Parameters	setpoint
Parameter presets	Move to top, Move to bottom

In this example we want to schedule the "Move Elevator" command to run whenever joystick button 1 is pressed.

1. The command to run is called "Move Elevator" and its function is to move the elevator on the robot to the top position
2. Notice that the command requires the Elevator subsystem. This will ensure that this command starts running even if there was another operation happening at the same time that used the elevator. In this case the previous command would be interrupted.
3. Parameters make it possible for one command to do multiple things; presets let you define values you pass to the command and reuse them

Adding the Joystick to the robot program



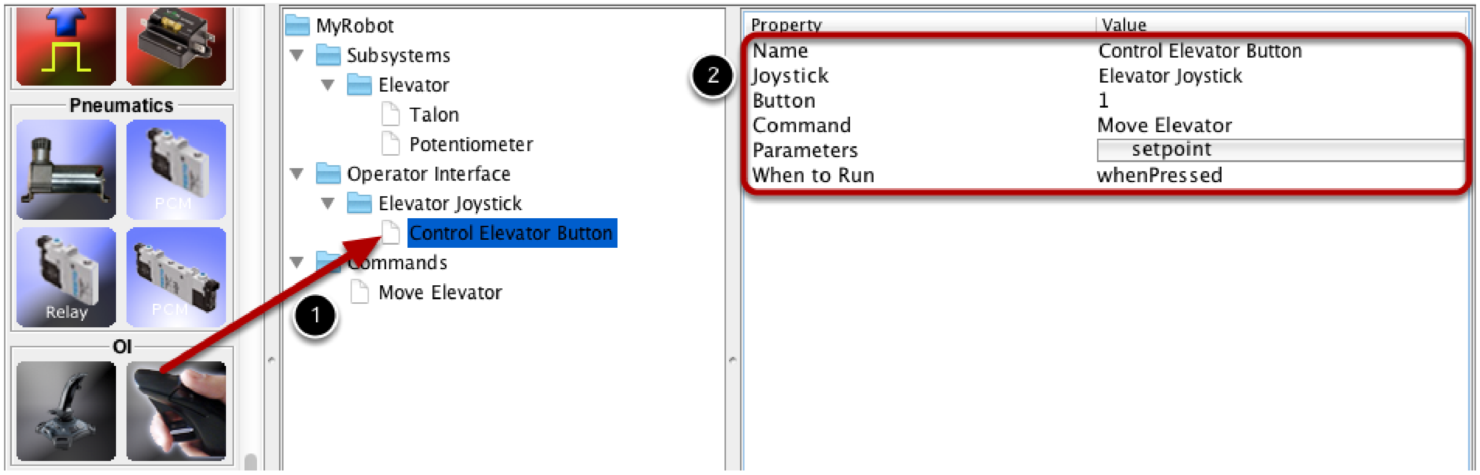
MyRobot
Subsystems
Elevator
Talon
Potentiometer
Operator Interface
Elevator Joystick
Commands
Move Elevator

Property	Value
Name	Elevator Joystick
Number	1

Add the joystick to the robot program

1. Drag the joystick to the Operator Interface folder in the robot program
2. Name the joystick so that it reflects the use of the joystick and set the USB port number

Add a button and link it to the "Move Elevator" command



MyRobot
Subsystems
Elevator
Talon
Potentiometer
Operator Interface
Elevator Joystick
Control Elevator Button
Commands
Move Elevator

Property	Value
Name	Control Elevator Button
Joystick	Elevator Joystick
Button	1
Command	Move Elevator
Parameters	setpoint
When to Run	whenPressed

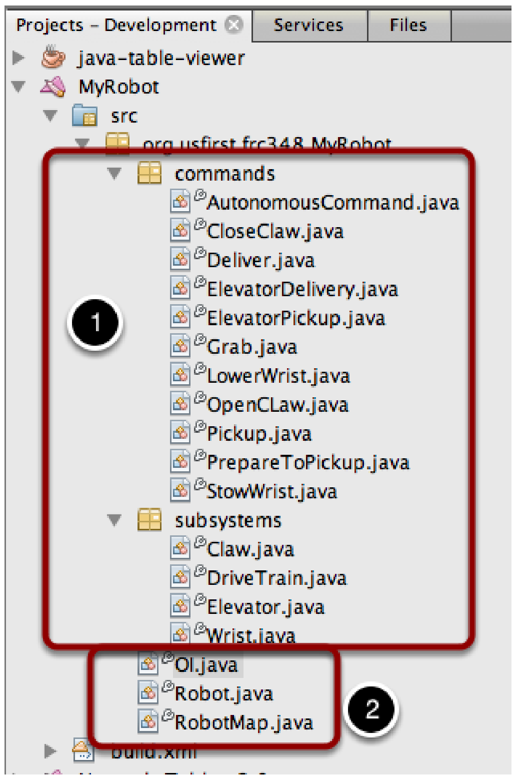
Add the button that should be pressed to the program

1. Drag the joystick button to the Joystick (Driver gamepad) so that it's under the joystick
2. Set the properties for the button: the button number, the command to run when the button is pressed, parameters the command takes, and the "When to run" property to "whenPressed" to indicate that the command should run whenever the joystick button is pressed.

Note: **Joystick buttons must be dragged to (under) a Joystick**. You must have a joystick in the Operator Interface folder before adding buttons.

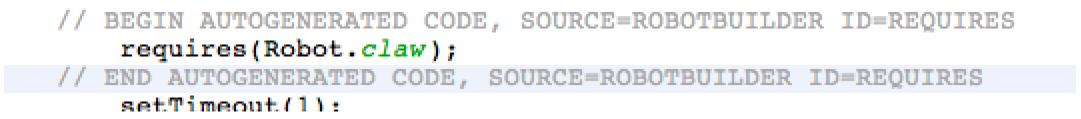
RobotBuilder created code

The layout of a RobotBuilder generated project



A RobotBuilder generated project consists of a package (in Java) or a folder (in C++) for Commands and another for Subsystems (1). Each command or subsystem object is stored under those containers (2). At the top level of the project you'll find the robot main program (Robot.java), the Operator Interface file (OI.java) and the RobotMap that contains the code to create all the subsystem components that were added to the robot description.

Autogenerated code

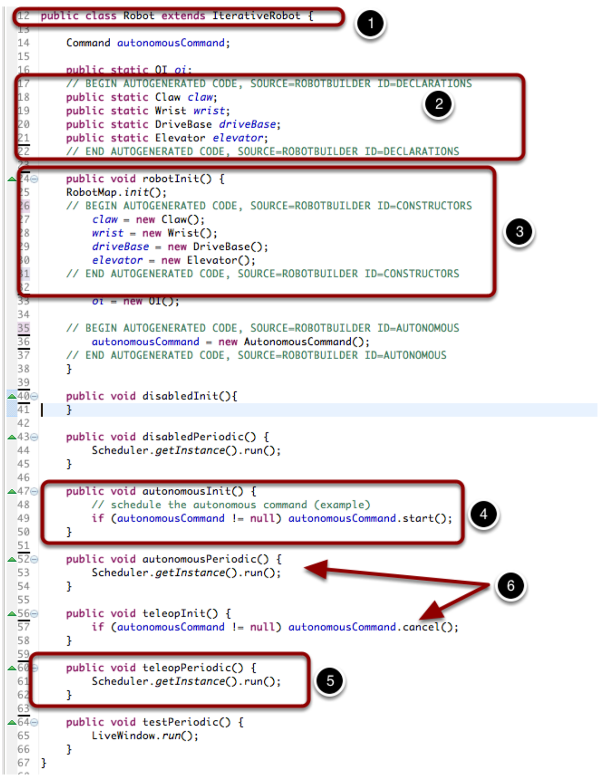


```
// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
    requires(Robot.claw);
// END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
    setTimeout(1);
```

When the robot description is modified and code is re-exported RobotBuilder is designed to not modify any changes you made to the file, thus preserving your code. This makes RobotBuilder a full-lifecycle tool. To know what code is OK to be modified by RobotBuilder, it generates sections that will potentially have to be rewritten delimited with some special comments. These comments are shown in the example above. Don't add any code within these comment blocks, it will be rewritten next time the project is exported from RobotBuilder.

If code inside one of these blocks must be modified, the comments can be removed, but this will prevent further updates from happening later. In the above example, if the //BEGIN and //END comments were removed, then later another required subsystem was added in RobotBuilder, it would not be generated on that next export.

Main robot program



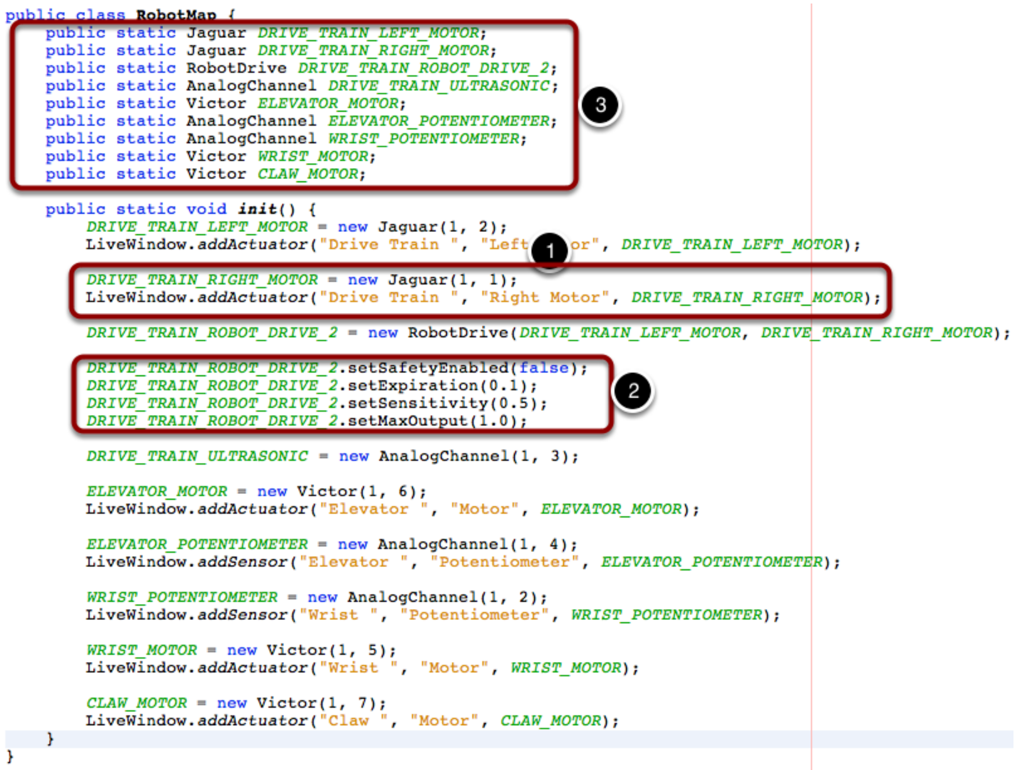
```
public class Robot extends IterativeRobot {

    Command autonomousCommand;

    public static OI oi;
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    public static Claw claw;
    public static Wrist wrist;
    public static DriveBase driveBase;
    public static Elevator elevator;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    public void robotInit() {
    RobotMap.init();
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
        claw = new Claw();
        wrist = new Wrist();
        driveBase = new DriveBase();
        elevator = new Elevator();
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS

        oi = new OI();

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
        autonomousCommand = new AutonomousCommand();
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
    }

    public void disabledInit(){
    }

    public void disabledPeriodic() {
        Scheduler.getInstance().run();
    }

    public void autonomousInit() {
        // schedule the autonomous command (example)
        if (autonomousCommand != null) autonomousCommand.start();
    }

    public void autonomousPeriodic() {
        Scheduler.getInstance().run();
    }

    public void teleopInit() {
        if (autonomousCommand != null) autonomousCommand.cancel();
    }

    public void teleopPeriodic() {
        Scheduler.getInstance().run();
    }

    public void testPeriodic() {
        LiveWindow.run();
    }
}
```

This is the main program generated by RobotBuilder. There are a number of parts to this program:

1. This class extends IterativeRobot. IterativeRobot will call your autonomousPeriodic() and teleopPeriodic() methods every 20ms (each time the driver station exchanges Joystick and other data with the robot).
2. Each of the subsystems is declared here These are public static variables so that they can be referenced from throughout your robot program by writing Robot.<*subsystem-name*>.method(), for example Robot.elevator.setSetpoint(4).

3. The subsystems are instantiated in the robotInit() method that is called after the construtor runs for this class. It is important to be create the subsystems after the constructor to avoid recursive loops. Also instance of the OI() class (for your operator interface) and the autonomous command are created here.
4. In the autonomousInit() method which is called every 20ms, make one scheduling pass. That is call the isFinished() and execute() methods of every command that is currently scheduled.
5. In the teleopPeriodic method which is called every 20ms, make one scheduling pass.
6. If there is an autonomous command provided in RobotBuilder robot properties, it is scheduled at the start of autonomous in the autonomousInit() method and canceled at the end of the autonomous period in teleopInit().

RobotMap - generation of all the actuator and sensor objects



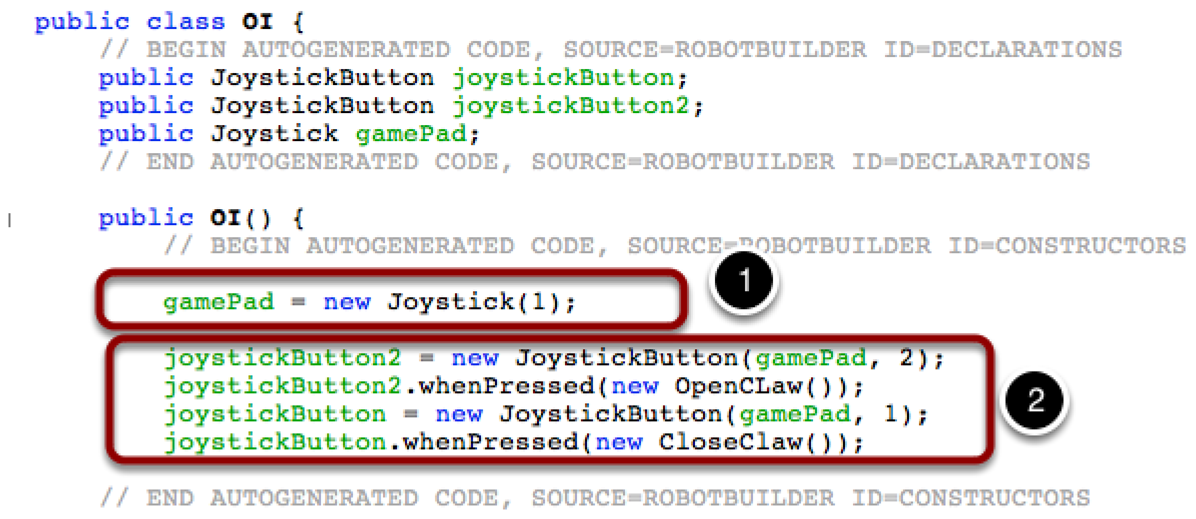
```java
public class RobotMap {
    public static Jaguar DRIVE_TRAIN_LEFT_MOTOR;
    public static Jaguar DRIVE_TRAIN_RIGHT_MOTOR;
    public static RobotDrive DRIVE_TRAIN_ROBOT_DRIVE_2;
    public static AnalogChannel DRIVE_TRAIN_ULTRASONIC;
    public static Victor ELEVATOR_MOTOR;
    public static AnalogChannel ELEVATOR_POTENTIOMETER;
    public static AnalogChannel WRIST_POTENTIOMETER;
    public static Victor WRIST_MOTOR;
    public static Victor CLAW_MOTOR;

    public static void init() {
        DRIVE_TRAIN_LEFT_MOTOR = new Jaguar(1, 2);
        LiveWindow.addActuator("Drive Train ", "Left   or", DRIVE_TRAIN_LEFT_MOTOR);

        DRIVE_TRAIN_RIGHT_MOTOR = new Jaguar(1, 1);
        LiveWindow.addActuator("Drive Train ", "Right Motor", DRIVE_TRAIN_RIGHT_MOTOR);

        DRIVE_TRAIN_ROBOT_DRIVE_2 = new RobotDrive(DRIVE_TRAIN_LEFT_MOTOR, DRIVE_TRAIN_RIGHT_MOTOR);

        DRIVE_TRAIN_ROBOT_DRIVE_2.setSafetyEnabled(false);
        DRIVE_TRAIN_ROBOT_DRIVE_2.setExpiration(0.1);
        DRIVE_TRAIN_ROBOT_DRIVE_2.setSensitivity(0.5);
        DRIVE_TRAIN_ROBOT_DRIVE_2.setMaxOutput(1.0);

        DRIVE_TRAIN_ULTRASONIC = new AnalogChannel(1, 3);

        ELEVATOR_MOTOR = new Victor(1, 6);
        LiveWindow.addActuator("Elevator ", "Motor", ELEVATOR_MOTOR);

        ELEVATOR_POTENTIOMETER = new AnalogChannel(1, 4);
        LiveWindow.addSensor("Elevator ", "Potentiometer", ELEVATOR_POTENTIOMETER);

        WRIST_POTENTIOMETER = new AnalogChannel(1, 2);
        LiveWindow.addSensor("Wrist ", "Potentiometer", WRIST_POTENTIOMETER);

        WRIST_MOTOR = new Victor(1, 5);
        LiveWindow.addActuator("Wrist ", "Motor", WRIST_MOTOR);

        CLAW_MOTOR = new Victor(1, 7);
        LiveWindow.addActuator("Claw ", "Motor", CLAW_MOTOR);
    }
}
```

The RobotMap is a mapping from the ports sensors and actuators are wired into to a variable name. This provides flexibility changing wiring, makes checking the wiring easier and significantly reduces the number of magic numbers floating around. All the definitions of sensors and motors from the robot description are generated here.

Notice that each sensor and actuator is added to the LiveWindow class (1) so that the can be automatically displayed when the SmartDashboard is set to LiveWindow mode. Also any properties for the particular sensor or actuator is set here to reflect the settings made in the robot description. (2)

Each of the references for the objects are declared and instantiated here (3, 1), but they are copied into every subsystem to make it easy and clean to write code that uses them.

OI class - the Operator Interface



```
public class OI {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    public JoystickButton joystickButton;
    public JoystickButton joystickButton2;
    public Joystick gamePad;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    public OI() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS

        gamePad = new Joystick(1);

        joystickButton2 = new JoystickButton(gamePad, 2);
        joystickButton2.whenPressed(new OpenCLaw());
        joystickButton = new JoystickButton(gamePad, 1);
        joystickButton.whenPressed(new CloseClaw());

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
```

The code for all the operator interface components is generated here (1). In addition the code to link the OI buttons to commands that should run is also generated here (2).

Writing C++ code for your robot

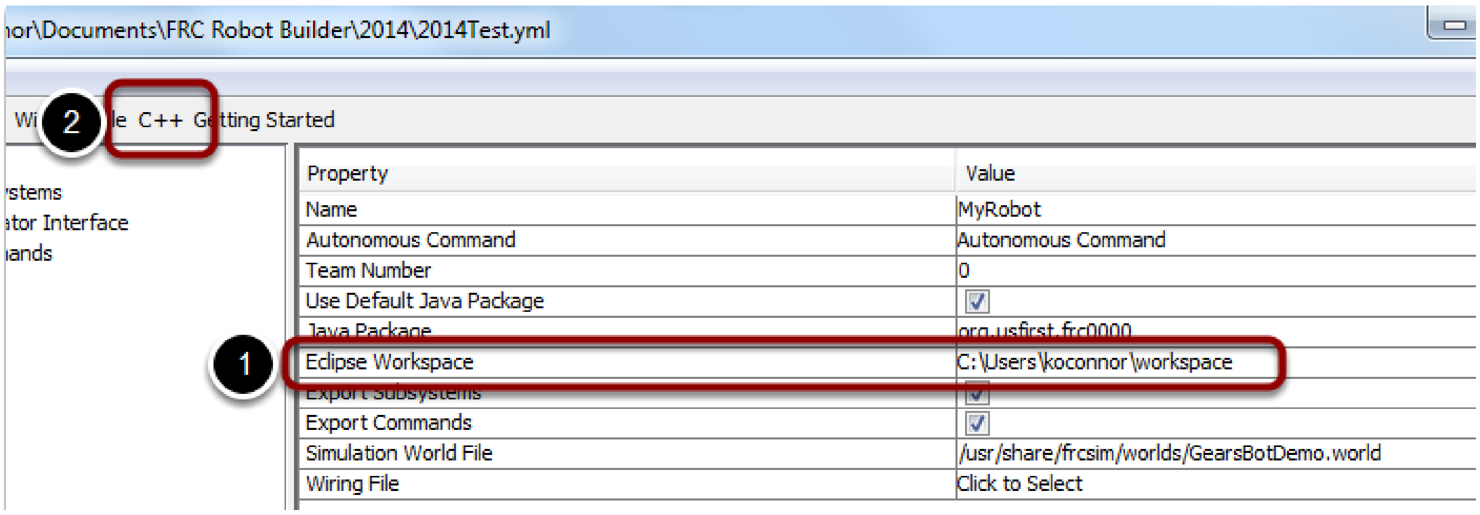
Generating C++ code for a project

After you've set up your robot framework in RobotBuilder, you'll need to export the code and load it into Eclipse. This article describes the process for doing so.

Prerequisite

A number of settings get configured when you create a project using the Eclipse plugins. If you have not yet created a project from within Eclipse, you should do so before proceeding (you can delete the project after). Instructions for doing this can be found in the Creating your Benchtop Test Program article.

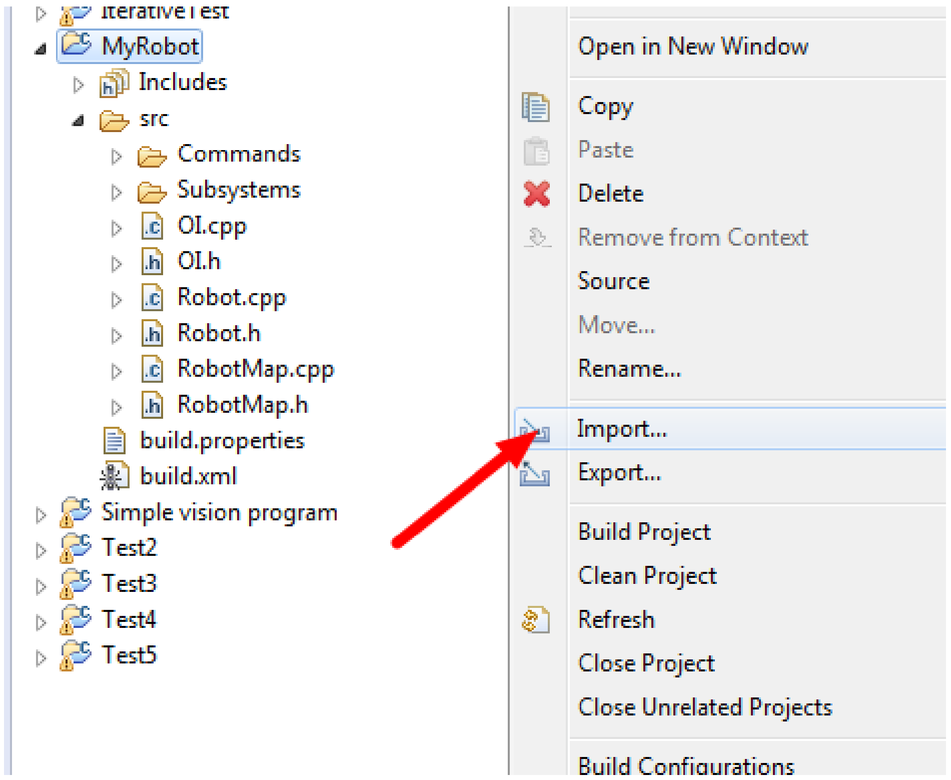
Generate the code for the project



Property	Value
Name	MyRobot
Autonomous Command	Autonomous Command
Team Number	0
Use Default Java Package	✓
Java Package	org.usfirst.frc0000
Eclipse Workspace	C:\Users\koconnor\workspace
Export Subsystems	✓
Export Commands	✓
Simulation World File	/usr/share/frcsim/worlds/GearsBotDemo.world
Wiring File	Click to Select

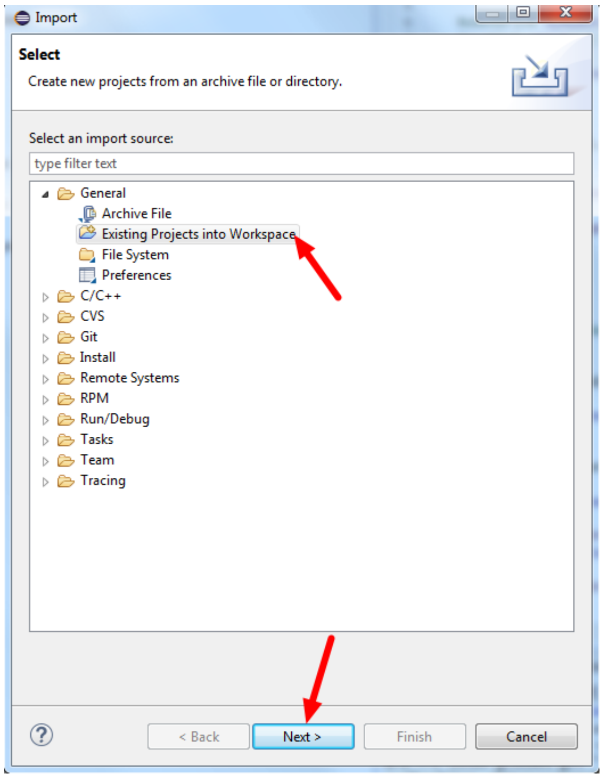
Verity that the Eclipse workspace location is set properly (1) and generate code for the C++ robot project (2).

Import the project into Eclipse



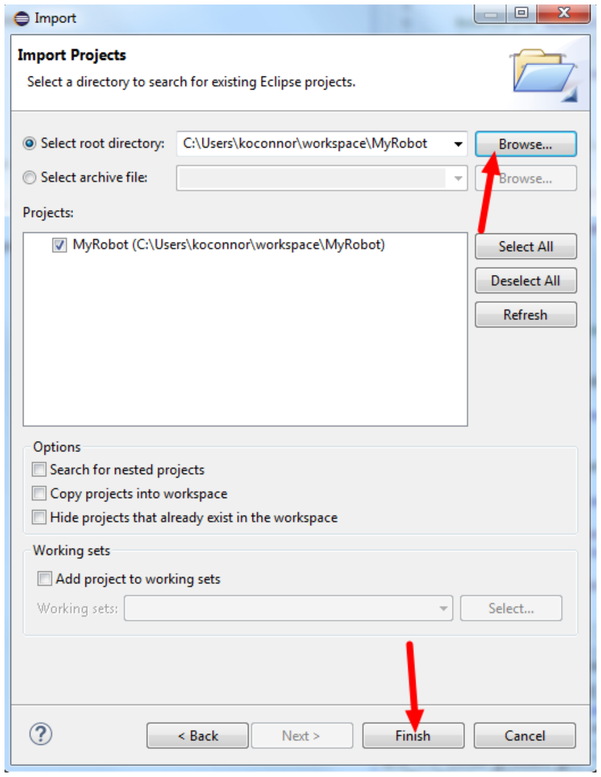
Right-click in the Project Explorer and import your project from the location set in RobotBuilder. Ideally the project has been saved in your workspace.

Project Type



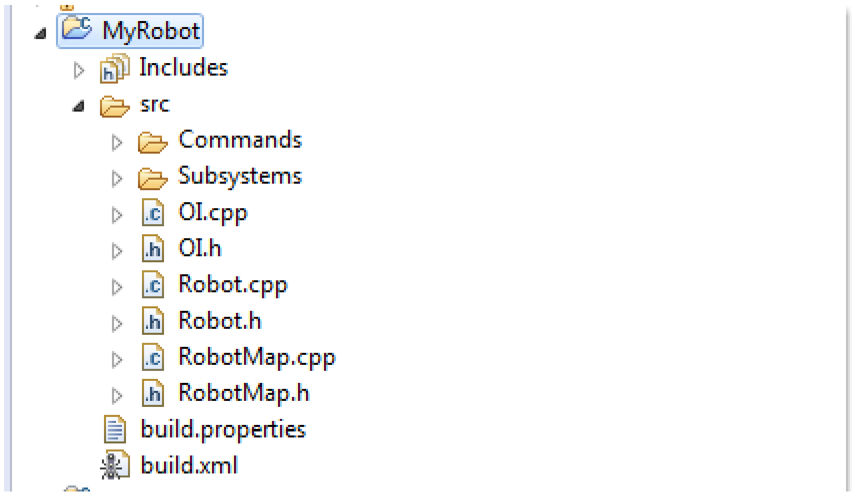
Select **Existing Projects into Workspace** then click Next.

Select directory


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory: C:\Users\koconnor\workspace\MyRobot
Browse...
Select archive file:
Browse...
Projects:
MyRobot (C:\Users\koconnor\workspace\MyRobot)
Select All
Deselect All
Refresh
Options
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel

Click **Browse..** locate the project directory created inside the workspace directory specified above, the click OK on the browse dialog and then Finish on the import window

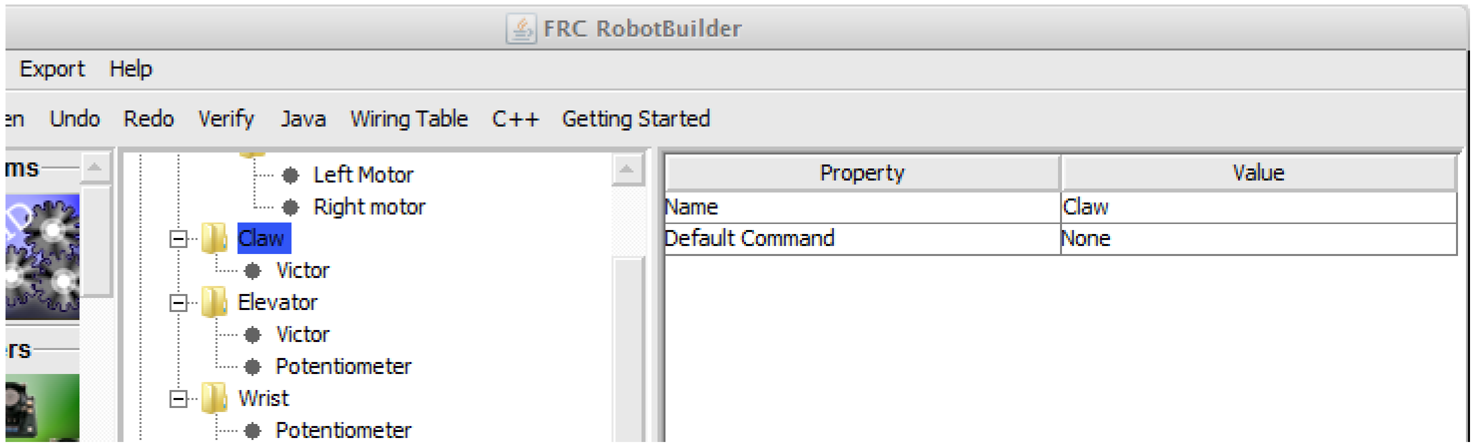
Viewing the imported project



You can view the project in the project explorer by double-clicking on the project name in the project explorer. Expand the src folder to view the source code. From there you can see all the project files. Your subsystems are in the Subsystems folder and the commands are in the Commands folder.

Writing the C++ code for a subsystem

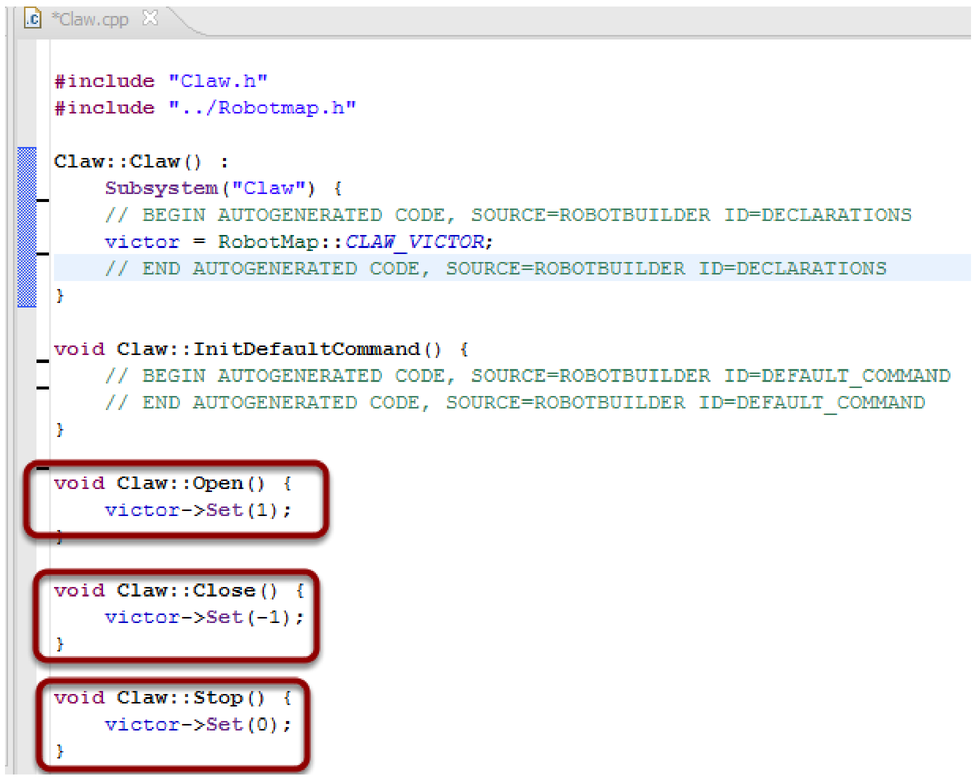
RobotBuilder representation of the Claw subsystem



Property	Value
Name	Claw
Default Command	None

The claw at the end of a robot arm is a subsystem operated by a single Victor speed controller. There are three things we want the claw to do, start opening, start closing, and stop moving. This is the responsibility of the subsystem. The timing for opening and closing will be handled by a command later in this tutorial.

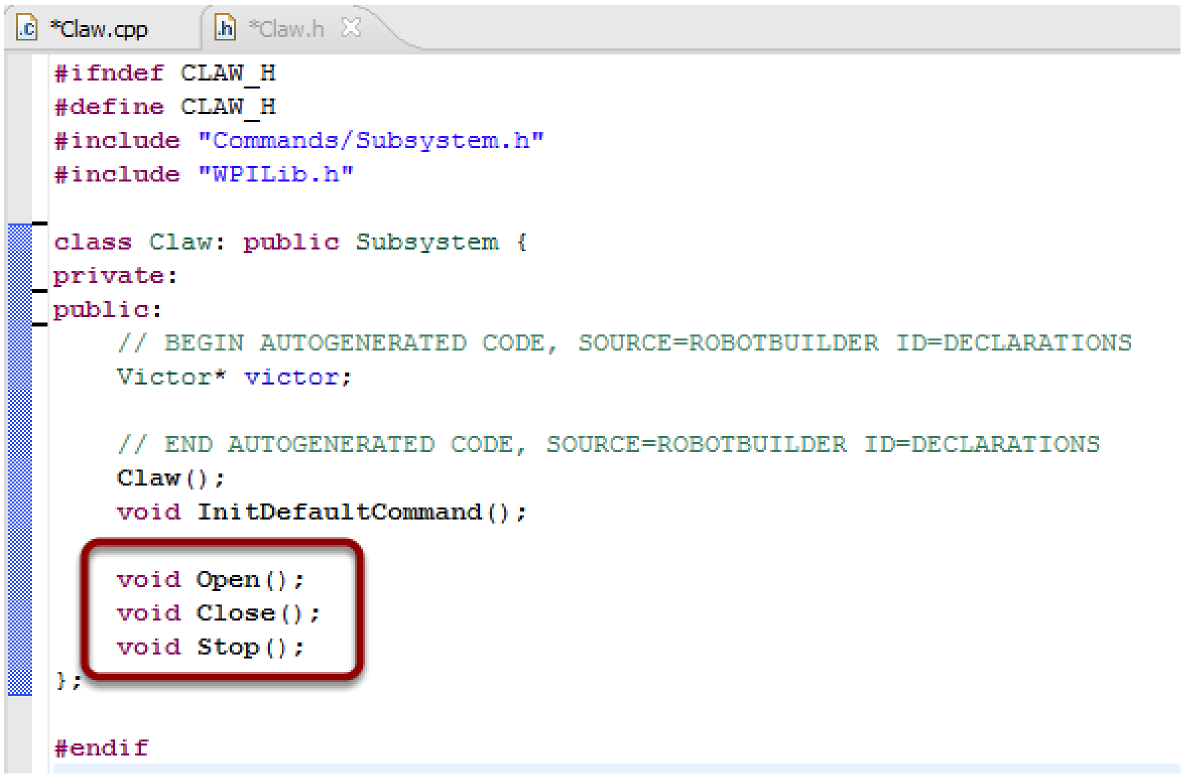
Adding capabilities to a simple subsystem


*Claw.cpp

```
#include "Claw.h"
#include "../Robotmap.h"

Claw::Claw() :
    Subsystem("Claw") {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    victor = RobotMap::CLAW_VICTOR;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
}

void Claw::InitDefaultCommand() {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND
}

void Claw::Open() {
    victor->Set(1);
}

void Claw::Close() {
    victor->Set(-1);
}

void Claw::Stop() {
    victor->Set(0);
}
```

The subsystem generated by RobotBuilder has the code to create the Claw class that corresponds to the subsystem and the code to copy the reference to the generated Victor object into the class to make it easily referenced. To add the capabilities to the subsystem to actually operate the motor add 3 methods, Open(), Close(), and Stop() to start the motor moving in the open or close direction or stop the motor.

Adding the method declarations to the generated header file - Claw.h



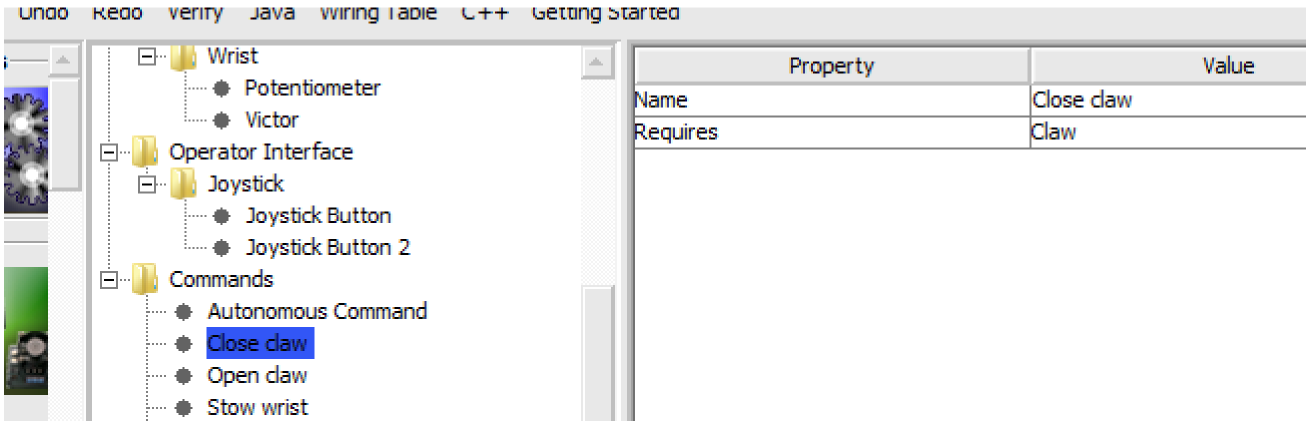
```
#ifndef CLAW_H
#define CLAW_H
#include "Commands/Subsystem.h"
#include "WPILib.h"

class Claw: public Subsystem {
private:
public:
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    Victor* victor;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    Claw();
    void InitDefaultCommand();

    void Open();
    void Close();
    void Stop();
};

#endif
```

In addition to adding the methods to the class implementation file, Claw.cpp, the declarations for the methods need to be added to the header file, Claw.h. Those declarations that must be added are shown here.

To add the behavior to the claw subsystem to operate it to handle opening and closing you need to define commands.

Writing the code for a command in C++

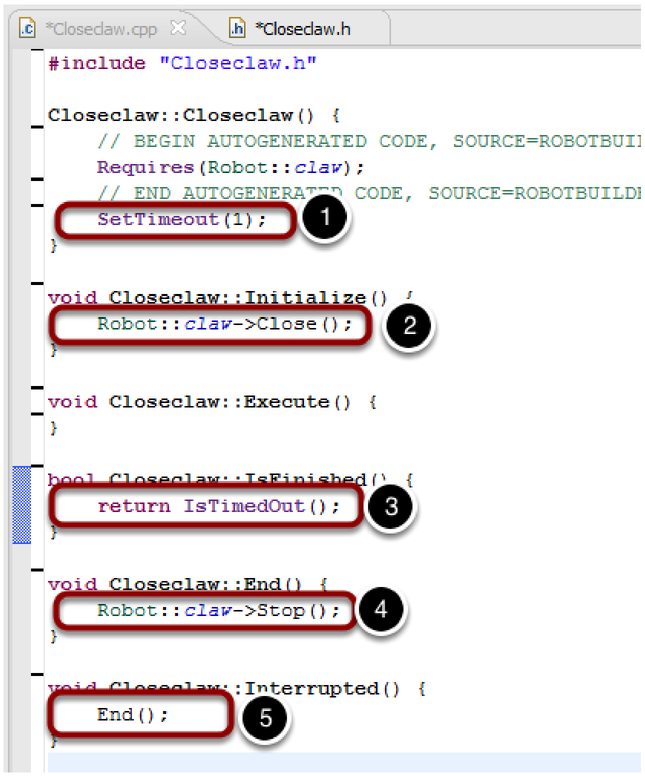
Close claw command in RobotBuilder



Property	Value
Name	Close claw
Requires	Claw

This is the definition of the Close claw command in RobotBuilder. Notice that it requires the Claw subsystem. This is explained in the next step.

Generated Closeclaw class generated in Closeclaw.cpp



```
#include "Closeclaw.h"

Closeclaw::Closeclaw() {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUI
    Requires(Robot::claw);
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILD
    SetTimeout(1);
}

void Closeclaw::Initialize() {
    Robot::claw->Close();
}

void Closeclaw::Execute() {
}

bool Closeclaw::IsFinished() {
    return IsTimedOut();
}

void Closeclaw::End() {
    Robot::claw->Stop();
}

void Closeclaw::Interrupted() {
    End();
}
```

RobotBuilder will generate the class files (header and implementation) for the Closeclaw command. The command represents the behavior of the claw, that is the operation over time. To operate this very simple claw mechanism the motor needs to operate for 1 second in the close direction. The Claw subsystem has methods to start the motor running in the right direction and to stop it. The commands responsibility is to run the motor for the correct time. The lines of code that are shown in the boxes are added to add this behavior.

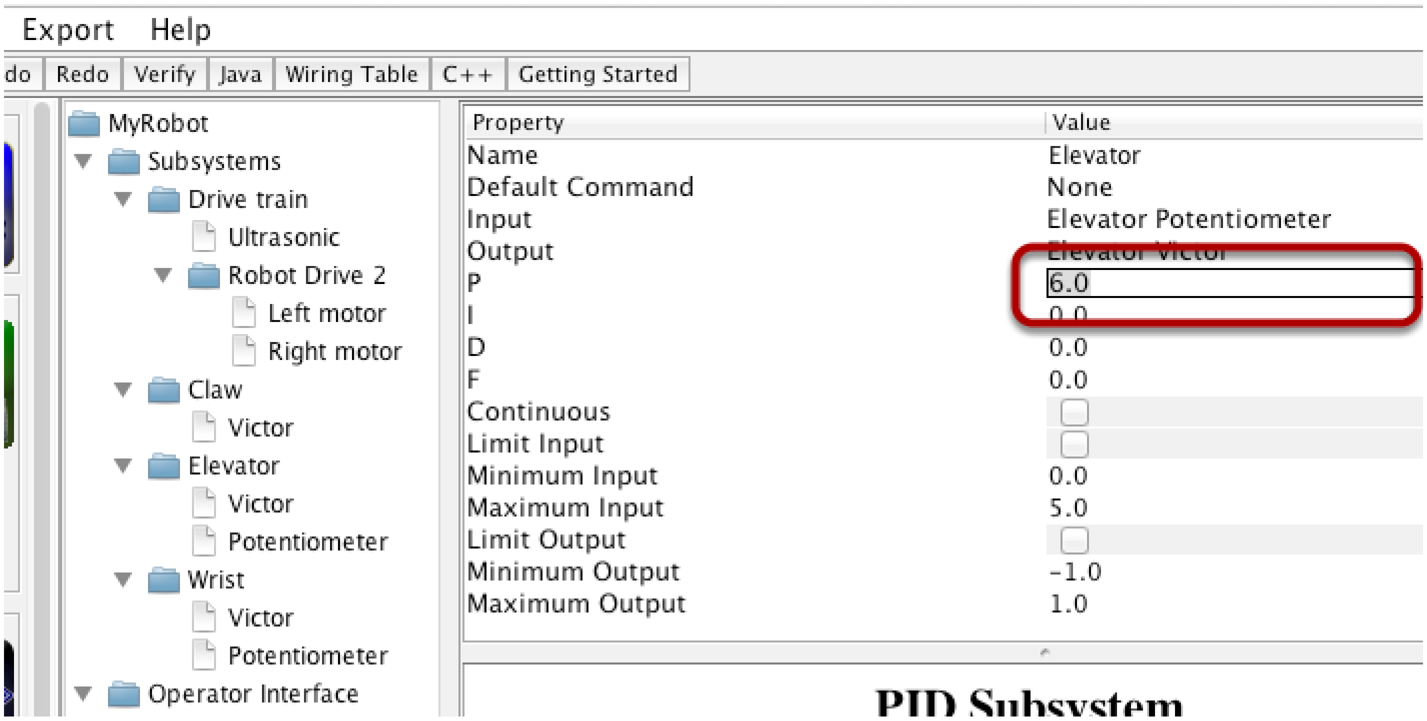
1. Set the one second timeout for this command. When the command is scheduled, a timer will be started so the one second operation can easily be tested.
2. Start the claw motor moving in the closing direction by calling the Close method that was added to the Claw subsystem.
3. This command is finished when the timer runs out which happens after one second has passed. This is the timer set in step 1.
4. The End() method is called when the command is finished and is a place to clean up. In this case, the motor is stopped since the time has run out.
5. The Interrupted() method is called is this command is interrupted if another command that also requires the Claw subsystem is scheduled before this finishes. For example, if the Closeclaw command was scheduled and running, then the Openclaw command was

scheduled it would interrupt the Openclaw command, call its Interrupted() method, and the motor would stop.

Writing the code for a PIDSubsystem in C++

PIDSubsystems use feedback to control the actuator and drive it to a particular position. In this example we use an elevator with a 10-turn potentiometer connected to it to give feedback on the height. The skeleton of the PIDSubsystem is generated by the RobotBuilder and we have to fill in the rest of the code to provide the potentiometer value and drive the motor with the output of the imbedded PIDController.

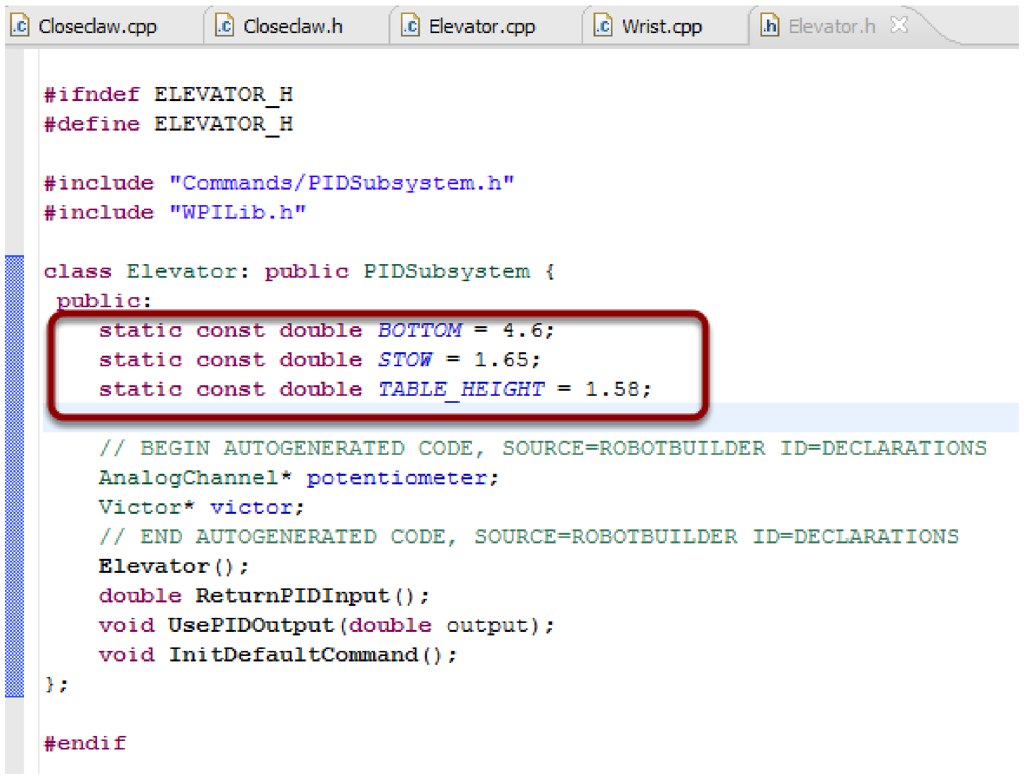
Setting the PID constants



Property	Value
Name	Elevator
Default Command	None
Input	Elevator Potentiometer
Output	Elevator Victor
P	6.0
I	0.0
D	0.0
F	0.0
Continuous	☐
Limit Input	☐
Minimum Input	0.0
Maximum Input	5.0
Limit Output	☐
Minimum Output	-1.0
Maximum Output	1.0

Make sure the Elevator PID subsystem has been created in the RobotBuiler. In the case of our elevator we use a propertional constant of 6.0 and 0 for the I and D terms. Once it's all set, generate C++ code for the project using the Export menu or the C++ toolbar menu.

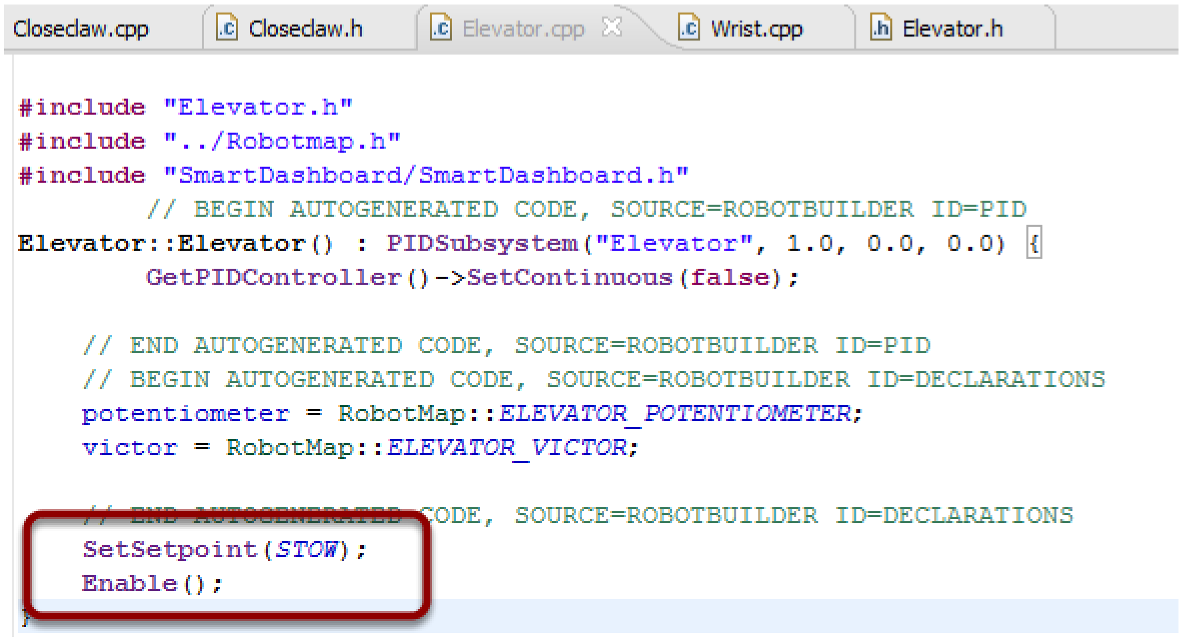
Add constants for the Elevator preset positions



```
#ifndef ELEVATOR_H
#define ELEVATOR_H

#include "Commands/PIDSubsystem.h"
#include "WPILib.h"

class Elevator: public PIDSubsystem {
 public:
    static const double BOTTOM = 4.6;
    static const double STOW = 1.65;
    static const double TABLE_HEIGHT = 1.58;

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    AnalogChannel* potentiometer;
    Victor* victor;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    Elevator();
    double ReturnPIDInput();
    void UsePIDOutput(double output);
    void InitDefaultCommand();
};

#endif
```

Elevator constants define potentiometer voltages that correspond to fixed positions on the elevator. These values can be determined using the print statements, the LiveWindow or SmartDashboard.

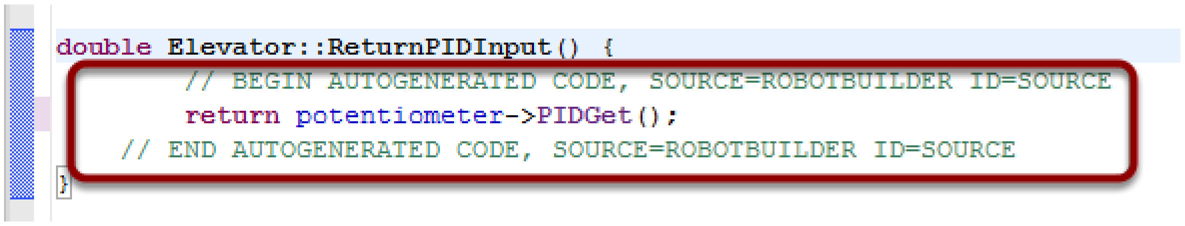
Initialize the elevator position in the Elevator constructor



```
#include "Elevator.h"
#include "../Robotmap.h"
#include "SmartDashboard/SmartDashboard.h"
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PID
Elevator::Elevator() : PIDSubsystem("Elevator", 1.0, 0.0, 0.0) {
        GetPIDController()->SetContinuous(false);

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PID
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    potentiometer = RobotMap::ELEVATOR_POTENTIOMETER;
    victor = RobotMap::ELEVATOR_VICTOR;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    SetSetpoint(STOW);
    Enable();
}
```

Set the elevator initial position so when the robot starts up it will move to that position. This will get the robot to a known starting point. Then enable the PIDController that is part of the PIDSubsystem. The elevator won't actually move until the robot itself is enabled because the motor outputs are initially off, but when the robot is enabled, the PID controller will already be running and the elevator will move to the "STOW" starting position.

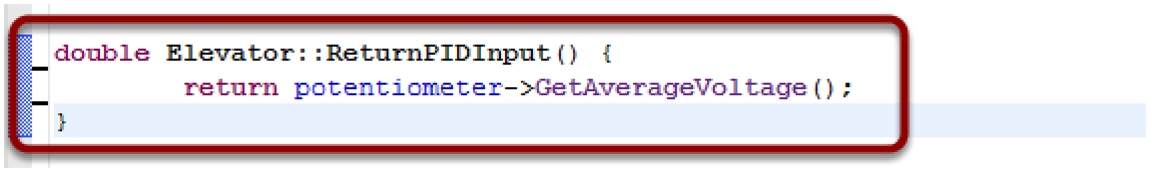
The autogenerated code to return the PID input values



```
double Elevator::ReturnPIDInput() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE
        return potentiometer->PIDGet();
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE
}
```

If you look at the RobotBuilder generated code for ReturnPIDInput() you can see that there is a //BEGIN and //END comment delimiting the return statement. \ In between //BEGIN and //END comments is where RobotBuilder will rewrite code on subsequent exports. So in general you should not change any code in that block. But, the function returns the value in raw analog units (0-1023). The setpoints are in units of Volts, so this won't work.

Set the ReturnPIDInput method to return voltage



```
double Elevator::ReturnPIDInput() {
        return potentiometer->GetAverageVoltage();
}
```

The function must be changed to return voltage. If we change the code inside the //BEGIN and //END block, it will just be overwritten next time RobotBuilder exports the file. **The solution is to remove the //BEGIN and //END comments, then make the change. This will prevent RobotBuilder from changing the code back again later.**

That's all that is required to create the Elevator PIDSubsystem in C++. To operate it with commands to actually control the motion see: Operating a PIDSubsystem from a command in C++.

Operating a PIDSubsystem from a command in C++

Writing Java code for your robot

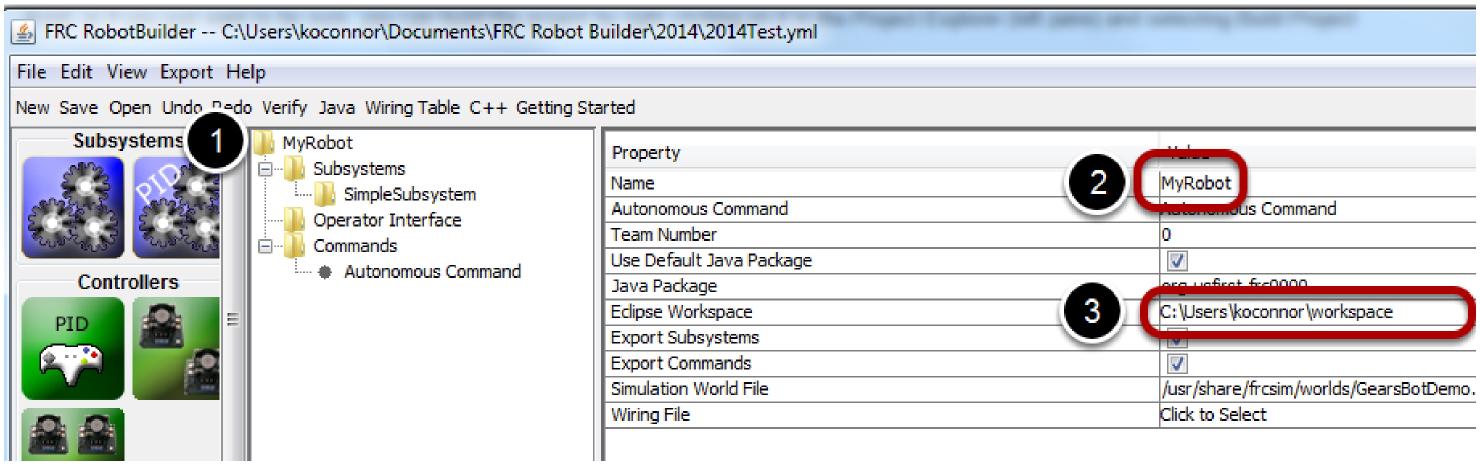
Generating Java Code for a project

After you start getting a significant part of your robot designed in RobotBuilder you can generate a Java project for use with Eclipse. The code that is generated includes project files that will let you just open the project and start adding your robot specific code. In addition, if you later make changes in RobotBuilder, you can regenerate the project again and it will not overwrite your changes. This process is described in detail below.

Prerequisite

A number of settings get configured when you create a project using the Eclipse plugins. If you have not yet created a project from within Eclipse, you should do so before proceeding (you can delete the project after). Instructions for doing this can be found in the Creating your Benchtop Test Program article.

Setting up the project properties for export

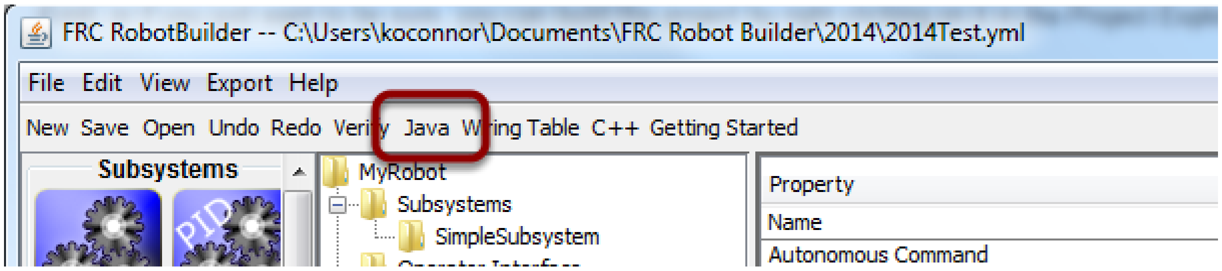

FRC RobotBuilder -- C:\Users\koconnor\Documents\FRC Robot Builder\2014\2014Test.yml
File Edit View Export Help
New Save Open Undo Redo Verify Java Wiring Table C++ Getting Started
Subsystems
PID
Controllers
PID
1
MyRobot
Subsystems
SimpleSubsystem
Operator Interface
Commands
Autonomous Command
2
3

Property	Value
Name	MyRobot
Autonomous Command	Autonomous Command
Team Number	0
Use Default Java Package	✓
Java Package	org.usfirst.frc0000
Eclipse Workspace	C:\Users\koconnor\workspace
Export Subsystems	✓
Export Commands	✓
Simulation World File	/usr/share/frcsim/worlds/GearsBotDemo.
Wiring File	Click to Select

Here is the procedure for setting up the project for Java code generation (export).

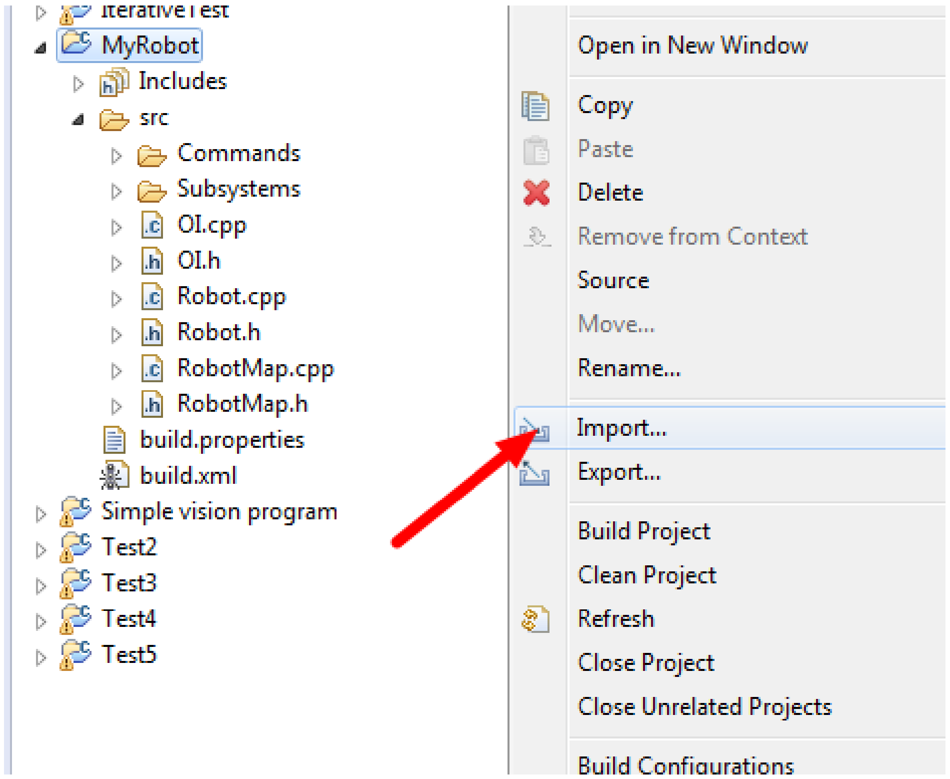
1. Select the project name in the top of the robot description to see the project properties.
2. Set the project name to something meaningful for your teams robot.
3. Set the directory where the project should be saved. This might be inside your Eclipse Workspace directory or some other folder.

Generate the project files


FRC RobotBuilder -- C:\Users\koconnor\Documents\FRC Robot Builder\2014\2014Test.yml
File Edit View Export Help
New Save Open Undo Redo Verify Java Wiring Table C++ Getting Started
Subsystems
MyRobot
Subsystems
SimpleSubsystem
Property
Name
Autonomous Command

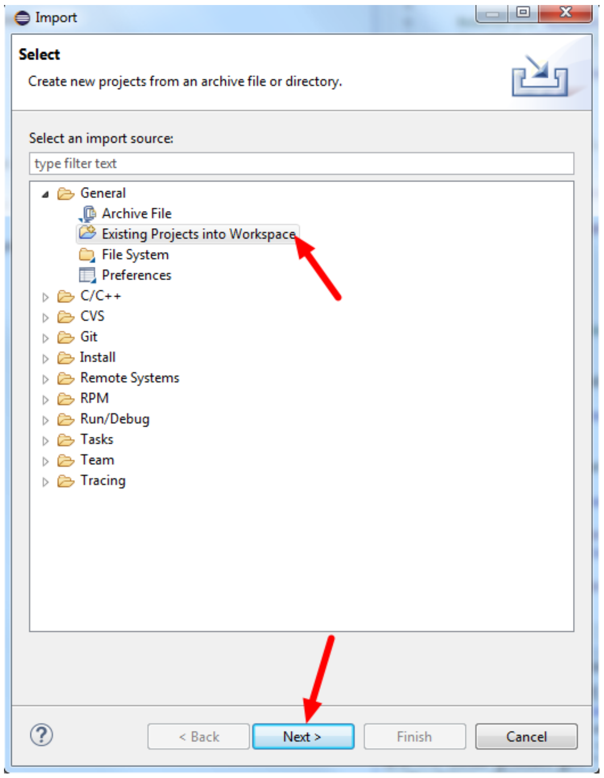
Once the location of the exported project files is defined (previous step) either click on Java from the Export menu or use the "Java" item in the toolbar to generate code to the correct location. This will generate a full project the first time the button is pressed, or it will update the project with changes on subsequent exports.

Import the project into Eclipse



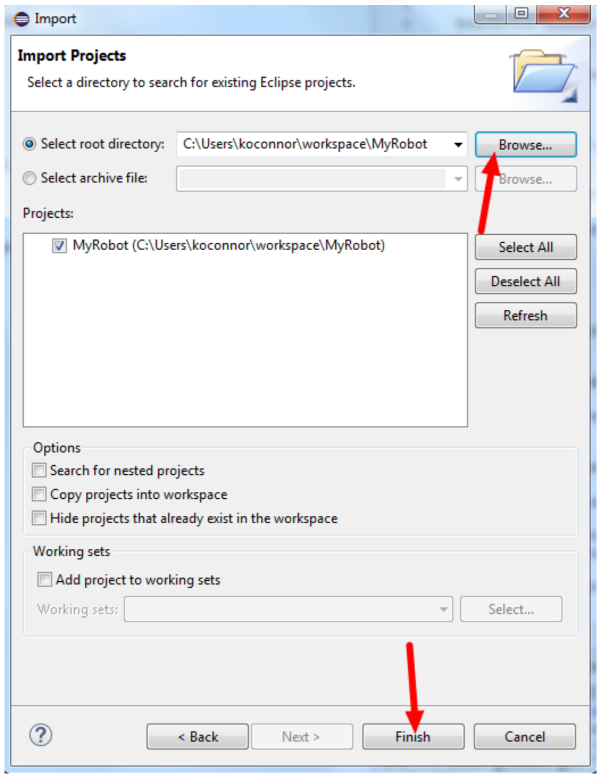
Right-click in the Project Explorer and import your project from the location set in RobotBuilder. Ideally the project has been saved in your workspace.

Project Type



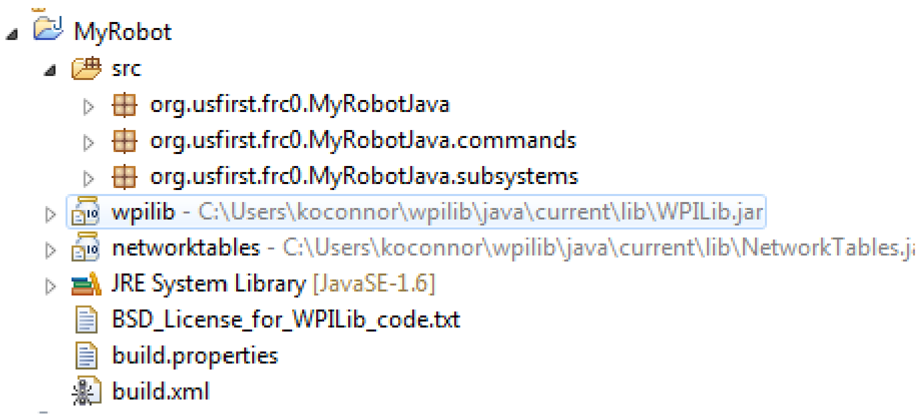
Select **Existing Projects into Workspace** then click Next.

Select directory


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory: C:\Users\koconnor\workspace\MyRobot
Browse...
Select archive file:
Browse...
Projects:
MyRobot (C:\Users\koconnor\workspace\MyRobot)
Select All
Deselect All
Refresh
Options
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel

Click **Browse..** locate the project directory created inside the workspace directory specified above, the click OK on the browse dialog and then Finish on the import window

Viewing the imported project

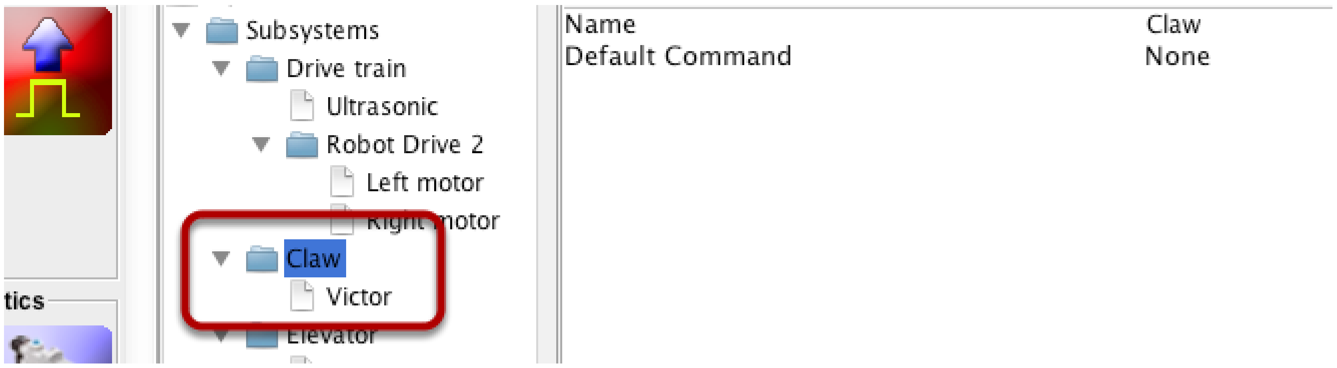

MyRobot
src
org.usfirst.frc0.MyRobotJava
org.usfirst.frc0.MyRobotJava.commands
org.usfirst.frc0.MyRobotJava.subsystems
wpilib - C:\Users\koconnor\wpilib\java\current\lib\WPILib.jar
networktables - C:\Users\koconnor\wpilib\java\current\lib\NetworkTables.j
JRE System Library [JavaSE-1.6]
BSD_License_for_WPILib_code.txt
build.properties
build.xml

You can view the project in the project explorer by double-clicking on the project name in the project explorer. Expand the src folder to view the source code. From there you can see all the project files. Your subsystems are in the Subsystems package and the commands are in the Commands package.

Writing the code for a subsystem in Java

Adding code to create an actual working subsystem is very straightforward. For simple subsystems that don't use feedback it turns out to be extremely simple. In this section we will look at an example of a Claw subsystem that operates the motor for some amount of time to open or close a claw on the robot arm.

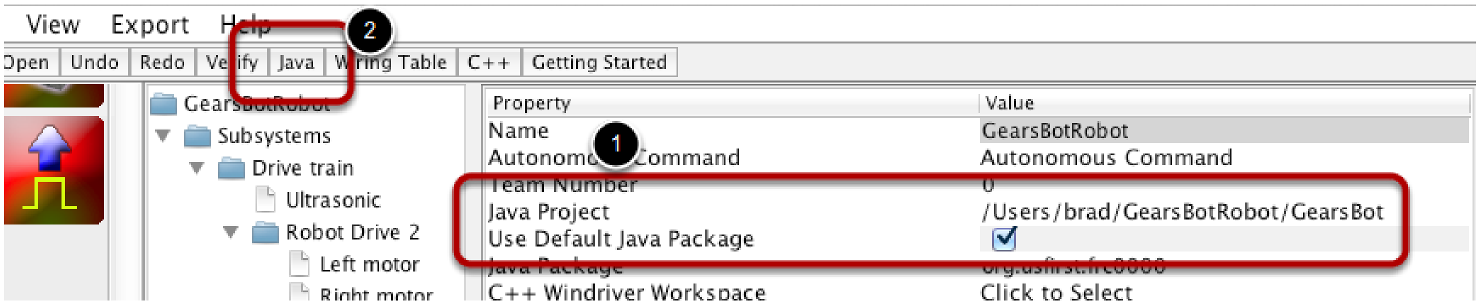
Create the subsystem



Name	Claw
Default Command	None

Be sure that the subsystem is defined in the RobotBuilder robot description. The Claw subsystem has a single Victor and no sensors since the claw motor operates for one second in either direction to open or close the claw.

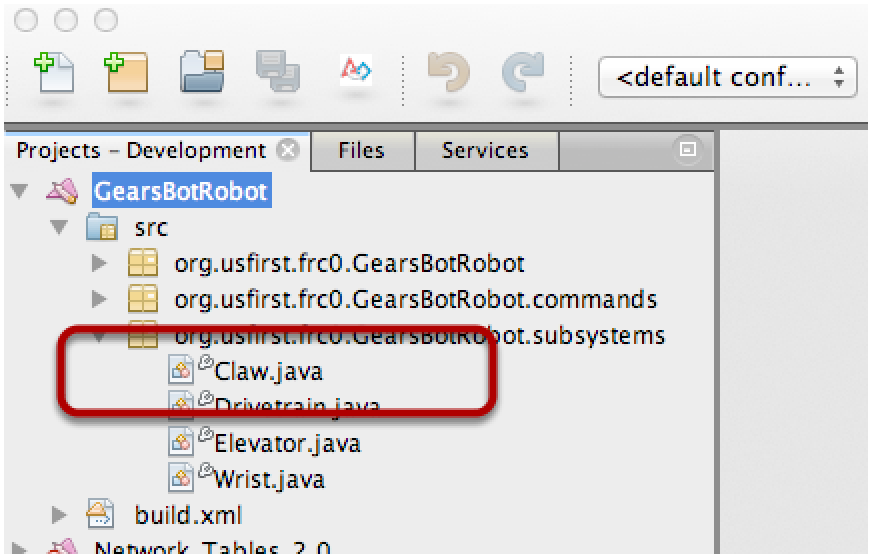
Generate code for the project



Property	Value
Name	GearsBotRobot
Autonomous Command	Autonomous Command
Team Number	0
Java Project	/Users/brad/GearsBotRobot/GearsBot
Use Default Java Package	✓
Java Package	org.usfirst.frc0000
C++ Windriver Workspace	Click to Select

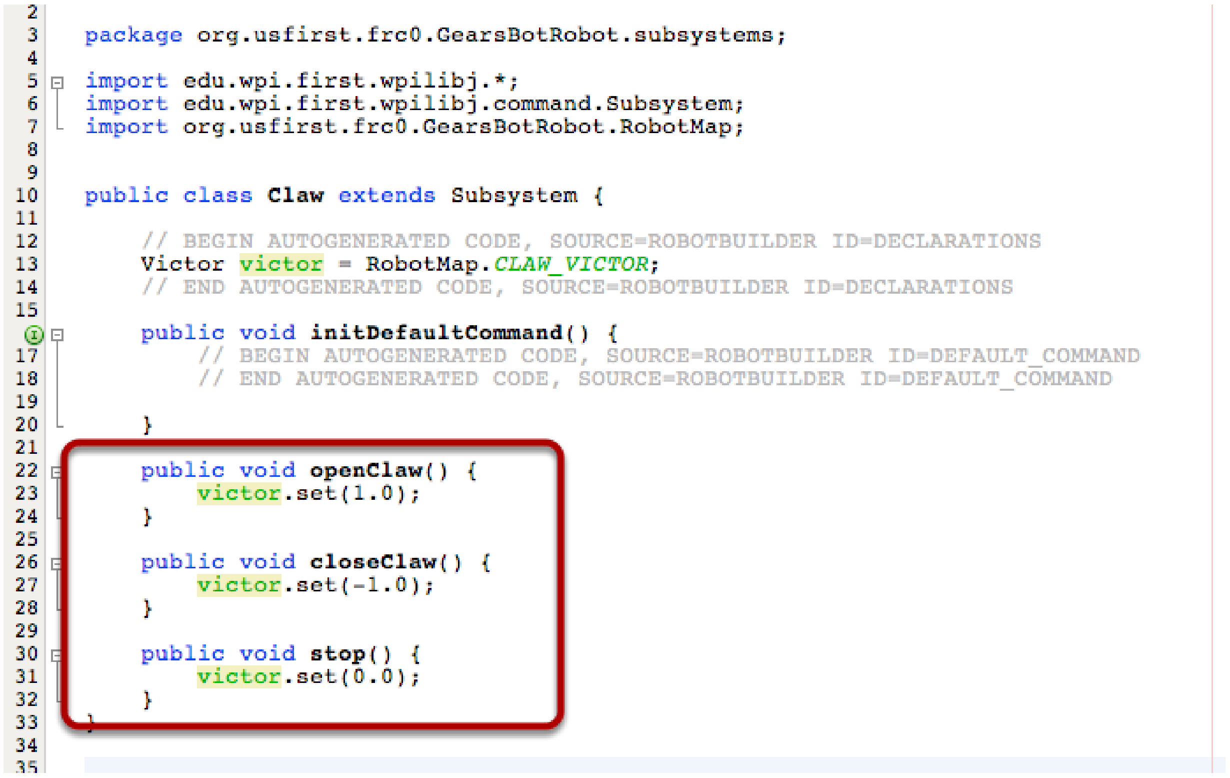
Verify that the java project location is set up (1) and generate code for the robot project (2).

Open the project in Netbeans



Open the generated project in Netbeans and notice the subystems package containing each of the subsystem files. Open the Claw.java file to add code that will open and close the claw.

Add methods to open, close, and stop the claw



```java

package org.usfirst.frc0.GearsBotRobot.subsystems;

import edu.wpi.first.wpilibj.*;
import edu.wpi.first.wpilibj.command.Subsystem;
import org.usfirst.frc0.GearsBotRobot.RobotMap;


public class Claw extends Subsystem {

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    Victor victor = RobotMap.CLAW_VICTOR;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    public void initDefaultCommand() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND
        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND

    }

    public void openClaw() {
        victor.set(1.0);
    }

    public void closeClaw() {
        victor.set(-1.0);
    }

    public void stop() {
        victor.set(0.0);
    }
}

```

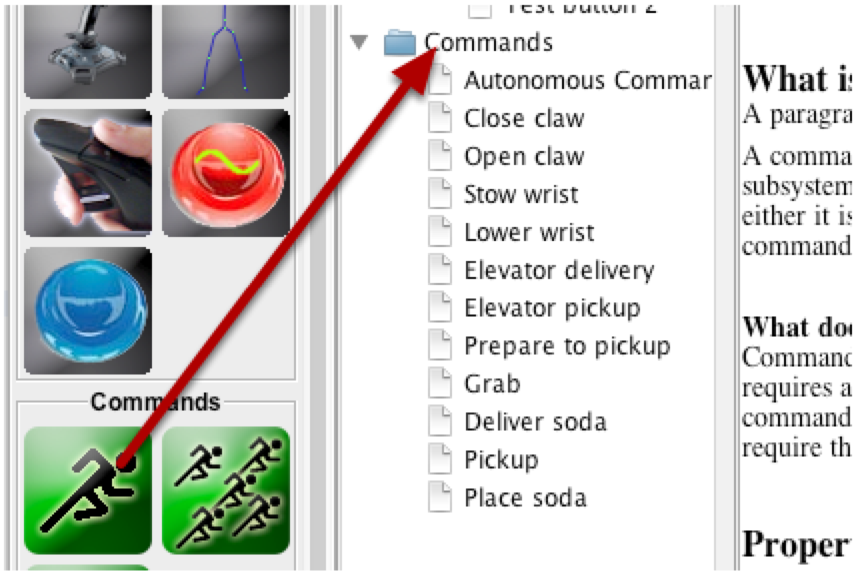
Add methods to the claw.java that will open, close, and stop the claw from moving. Those will be used by commands that actually operate the claw. The comments have been removed from this file to make it easier to see the changes for this document. Notice that a member variable called "victor" is created by RobotBuilder so it can be used throughout the subsystem. Each of your dragged-in palette items will have a member variable with the name given in RobotBuilder.

See: Writing the code for a command in Java to see how to get this Claw subsystem to operate using commands. See: Writing the code for a PIDSubystem in Java to write the code for a more complex subsystem with feedback (PIDSubsystem).

Writing the code for a simple command in Java

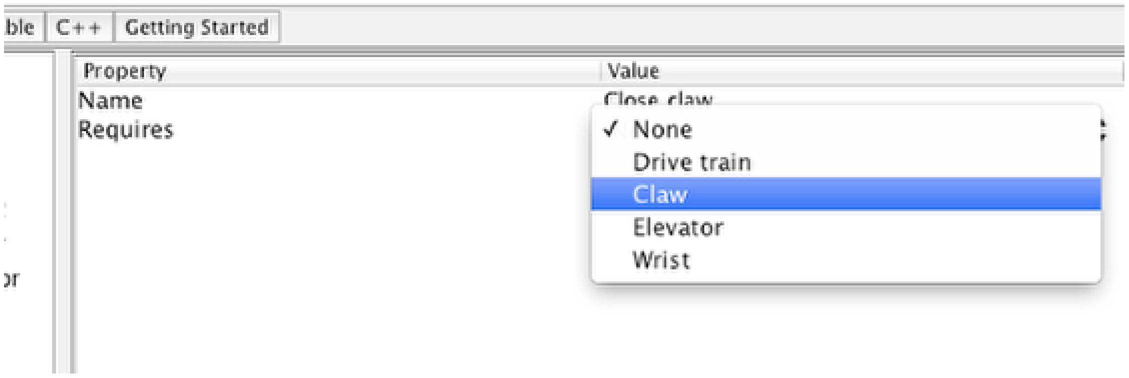
Subsystem classes get the mechanisms on your robot moving, but to get it to stop at the right time and sequence through more complex operations you write Commands. Previously in Writing the code for a subsystem in Java we developed the code for the Claw subsystem on a robot to start the claw opening, closing, or to stop moving. Now we will write the code for a command that will actually run the Claw motor for the right time to get the claw to open and close. Our claw example is a very simple mechanism where we run the motor for 1 second to open it or 0.9 seconds to close it.

Adding a command



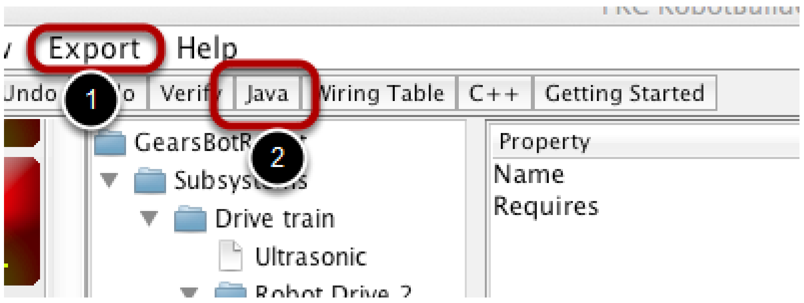
To create a command drag the command icon from the palette to the Commands folder in the robot description. This will create a command with a default name, then rename the command to be something meaningful, in this case "Open claw" or "Close claw".

Setting the Requires property to the correct subsystem



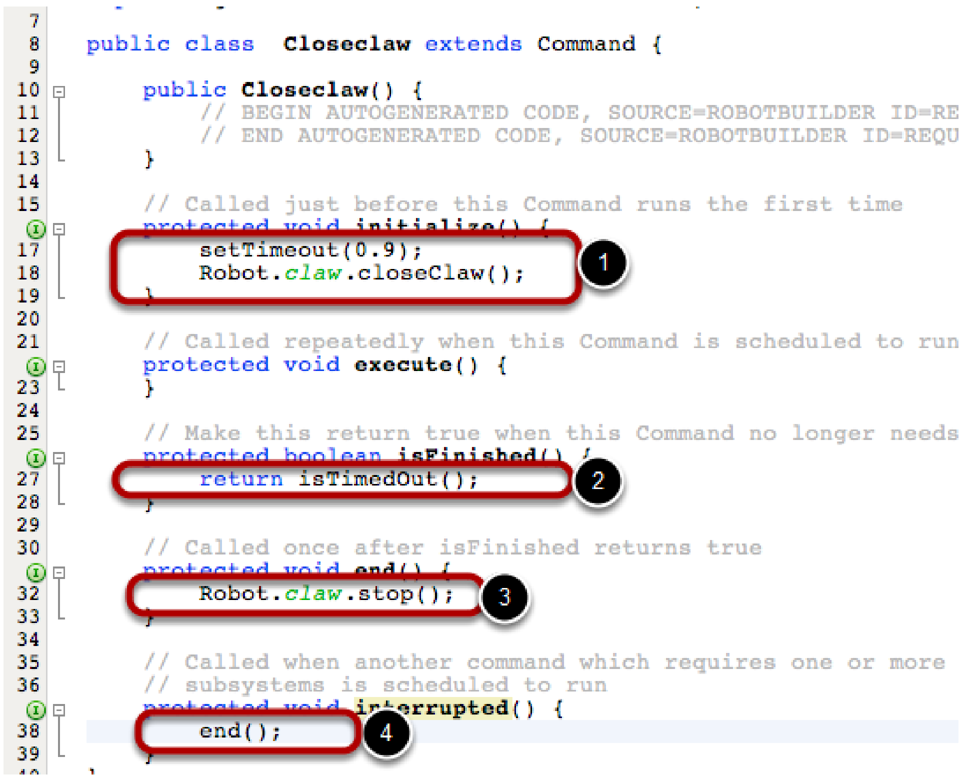
Set the Requires propert to the subsystem that this command is controlling. In this case, the "Close claw" command controls the Claw subsystem. If the "Close claw" command is scheduled while another command that uses the Claw is also running, the "Close claw" command will preempt the other command and start. For example, if the "Open claw" was running, then the robot operator decided that they really wanted to close it, since they both require the Claw, the second command (Close claw) would cancel the running open command.

Generate code for the project



Using either the Export menu (1) or the Java toolbar item (2), generate the code for the project.

Write the code to close the claw



```java
public class  Closeclaw extends Command {

    public Closeclaw() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=RE
        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQU
    }

    // Called just before this Command runs the first time
    protected void initialize() {
        setTimeout(0.9);
        Robot.claw.closeClaw();
    }

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
    }

    // Make this return true when this Command no longer needs
    protected boolean isFinished() {
        return isTimedOut();
    }

    // Called once after isFinished returns true
    protected void end() {
        Robot.claw.stop();
    }

    // Called when another command which requires one or more
    // subsystems is scheduled to run
    protected void interrupted() {
        end();
    }
```

Add these 5 lines of code to the Closeclaw command to it can be used:

1. The claw needs to run in the close direction for 0.9 seconds to completely get closed. Setting a timeout initializes the timer for this command. Each command can have a single timer that can be used for timing operations or timeouts to make sure that commands don't get stuck in the case of a broken sensor. Then start the claw closing. Notice that we only need to start the claw closing once, so having it in the initialize method is sufficient.
2. The claw should continue closing until the timer runs out. The command has an isTimedOut() method that returns true if the timer that was set in the initialize() method is done.
3. In the end() method stop the claw from moving. This is called when the isFinished() method returns true (the timer runs out in this case).
4. The interrupted() method is called when this command is preempted by another command using the Claw subsystem. In this case, we just call the end() method to stop the claw from moving.

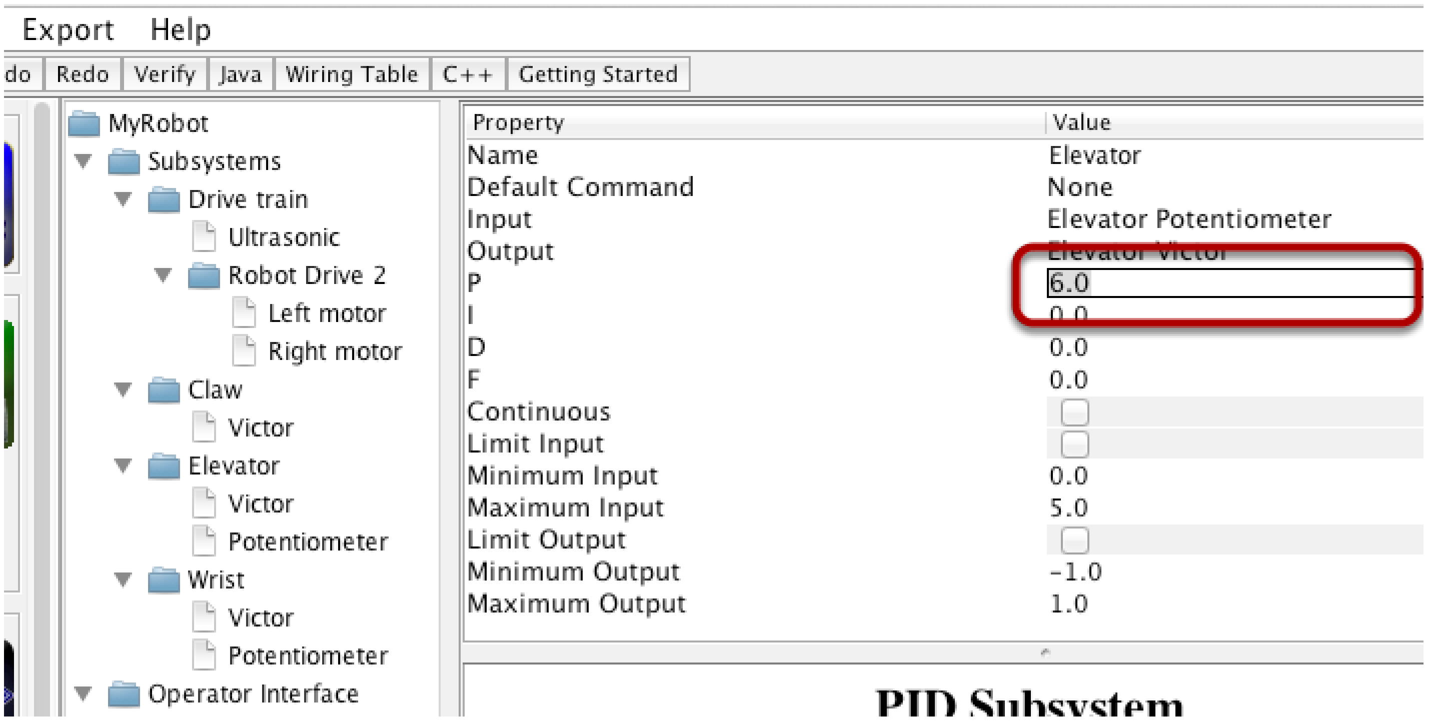
That's all that's required to get the claw to run when the command is run. Notice that you can refer to any subsystem by using the class name Robot since subsystem references are automatically generated as static variables. For example, "Robot.claw.closeClaw()" gives you access to the claw class and it's methods.

This command can be part of a more complex Command Group (see: Creating a command that runs other commands) or run from an operator interface button such as a joystick button (see: Connecting the operator interface to a command).

Writing the code for a PIDSubsystem in Java

PIDSubsystems use feedback to control the actuator and drive it to a particular position. In this example we use an elevator with a 10-turn potentiometer connected to it to give feedback on the height. The skeleton of the PIDSubsystem is generated by the RobotBuilder and we have to fill in the rest of the code to provide the potentiometer value and drive the motor with the output of the imbedded PIDController.

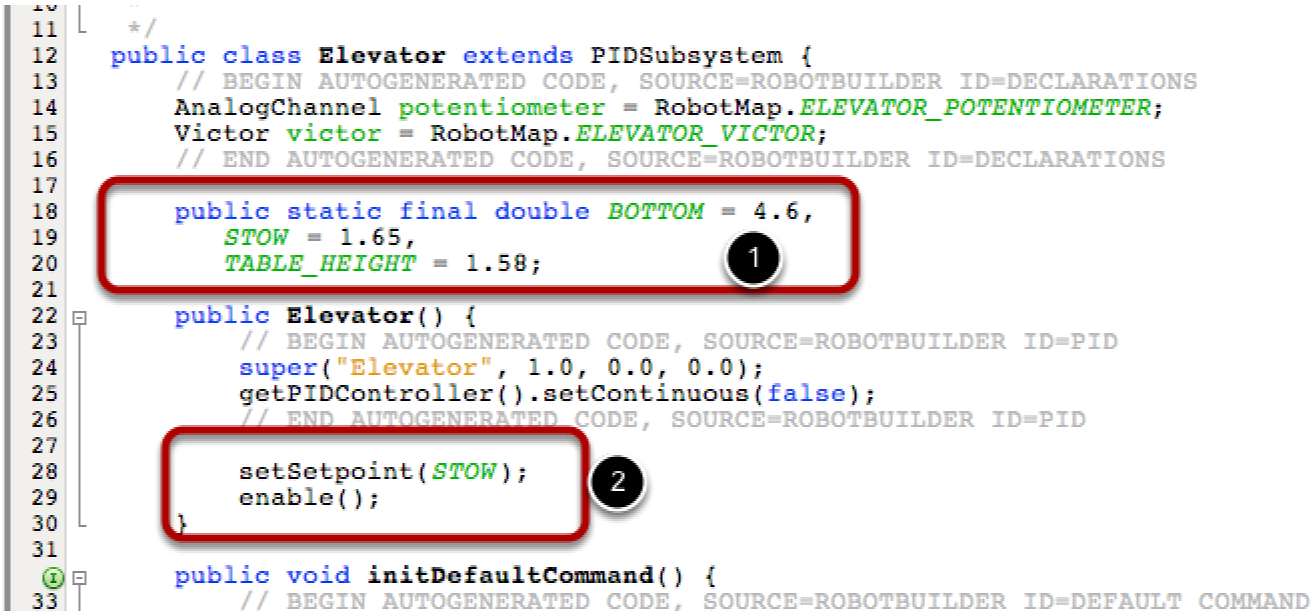
Setting the PID constants



Property	Value
Name	Elevator
Default Command	None
Input	Elevator Potentiometer
Output	Elevator Victor
P	6.0
I	0.0
D	0.0
F	0.0
Continuous	☐
Limit Input	☐
Minimum Input	0.0
Maximum Input	5.0
Limit Output	☐
Minimum Output	-1.0
Maximum Output	1.0

Make sure the Elevator PID subsystem has been created in the RobotBuiler. In the case of our elevator we use a propertional constant of 6.0 and 0 for the I and D terms. Once it's all set, generate Java code for the project using the Export menu or the Java toolbar menu.

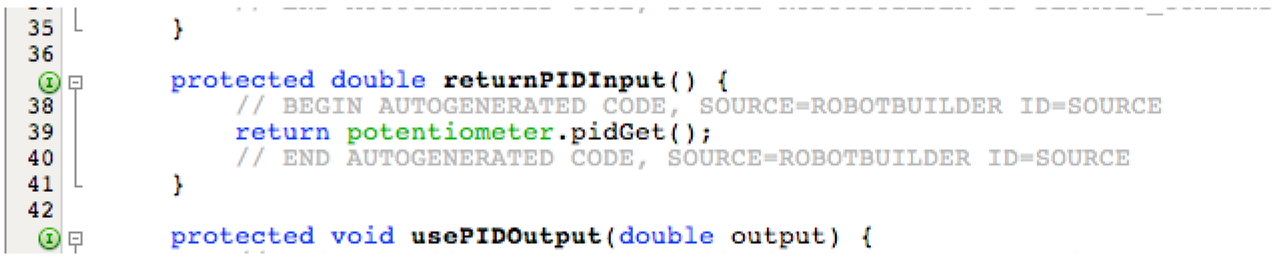
Add constants for the Elevator preset positions and enable the PID controller



```
    */
public class Elevator extends PIDSubsystem {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    AnalogChannel potentiometer = RobotMap.ELEVATOR_POTENTIOMETER;
    Victor victor = RobotMap.ELEVATOR_VICTOR;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    public static final double BOTTOM = 4.6,
       STOW = 1.65,
       TABLE_HEIGHT = 1.58;

    public Elevator() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PID
        super("Elevator", 1.0, 0.0, 0.0);
        getPIDController().setContinuous(false);
        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PID

        setSetpoint(STOW);
        enable();
    }

    public void initDefaultCommand() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND
```

To make it easier to drive the elevator to preset positions, we added preset positions for the bottom, stow, and table height. Then the elevator is set to the STOW position by setting the PID setpoint and the PID controller is enabled. This will cause the elevator to move to the stowed position when the robot is enabled.

Look at the autogenerated code from RobotBuilder for returnPIDInput



```
    }

    protected double returnPIDInput() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE
        return potentiometer.pidGet();
        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE
    }

    protected void usePIDOutput(double output) {
```

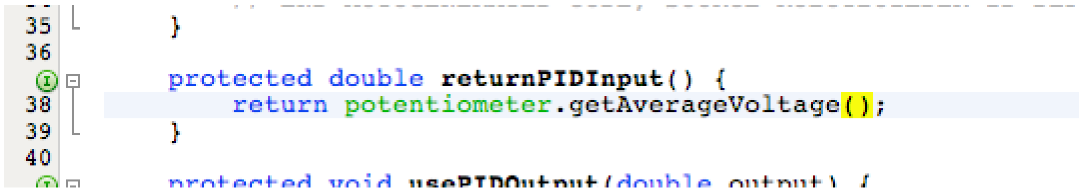
The returnPIDInput() method is used to set the value of the sensor that is providing the feedback for the PID controller. In this case, the code is automatically generated and returns the potentiometer raw analog input value (a number that ranges from 0-1023). In our case we would like the PID controller to be based on the average voltage read by the analog input for the potentiometer, not the raw value.

If we just change the line:

return potentiometer.pidGet();

it will be overwritten by RobotBuilder next time we export to Java. You can tell which lines are automatically generated by looking at the "//BEGIN AUTOGENERATED CODE" and "//END AUTOGENERATED CODE" comments. Any code inbetween those markers will be overwritten next time RobotBuilder is run. You're free to change anything outside of those blocks.

Use the avarage voltage for the PID input



```
    }

    protected double returnPIDInput() {
        return potentiometer.getAverageVoltage();
    }
```

To get around the problem from the last step, the comment blocks can be removed. Then if the line is changed as shown, it will no longer be overwritten by RobotBuilder.

Remember, if we just wanted to add code to a method it could be added safely outside of the comment blocks.

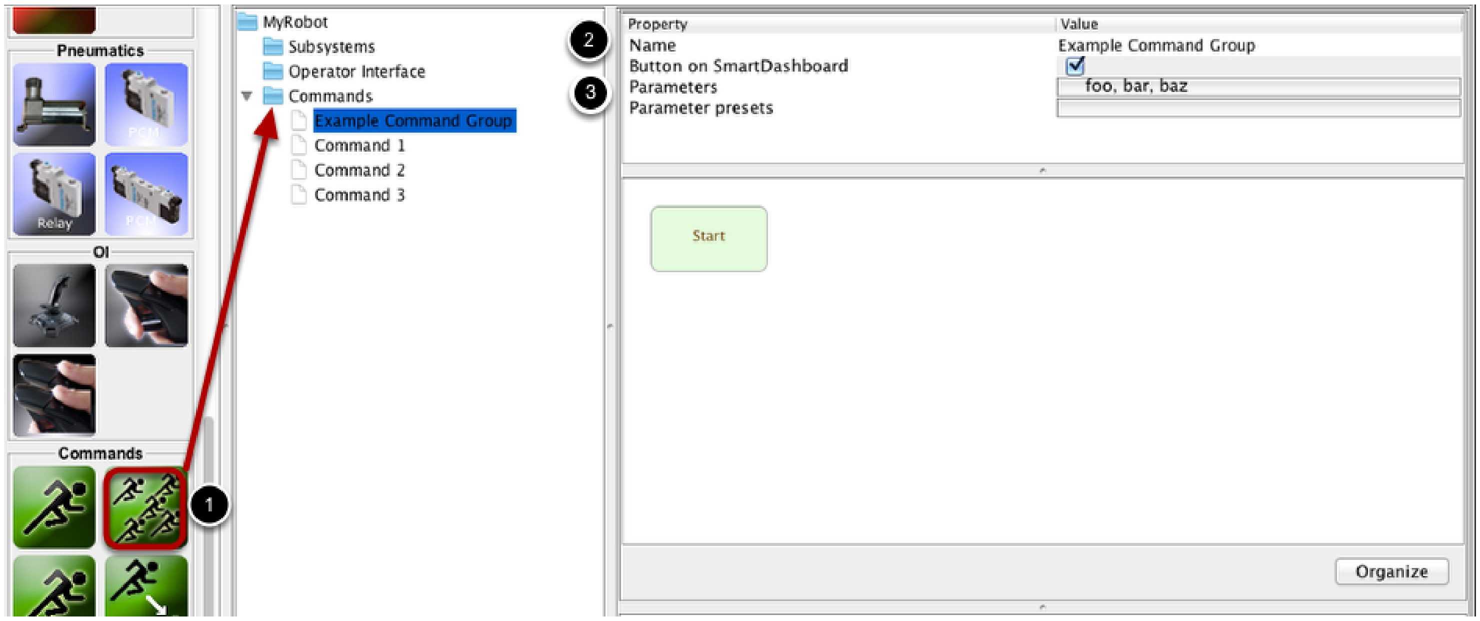
That's all that is required to create the Elevator PIDSubsystem in Java. To operate it with commands to actually control the motion see: Operating a PIDSubsystem from a command

Advanced techniques

Creating a command that runs other commands

Often you will want to run multiple commands, one after another to enable more complex behaviors in your program. Once each of the individual commands have been debugged, you can create a CommandGroup. A CommandGroup is a named set of commands that may be executed sequentially or in parallel.

Creating a Command Group

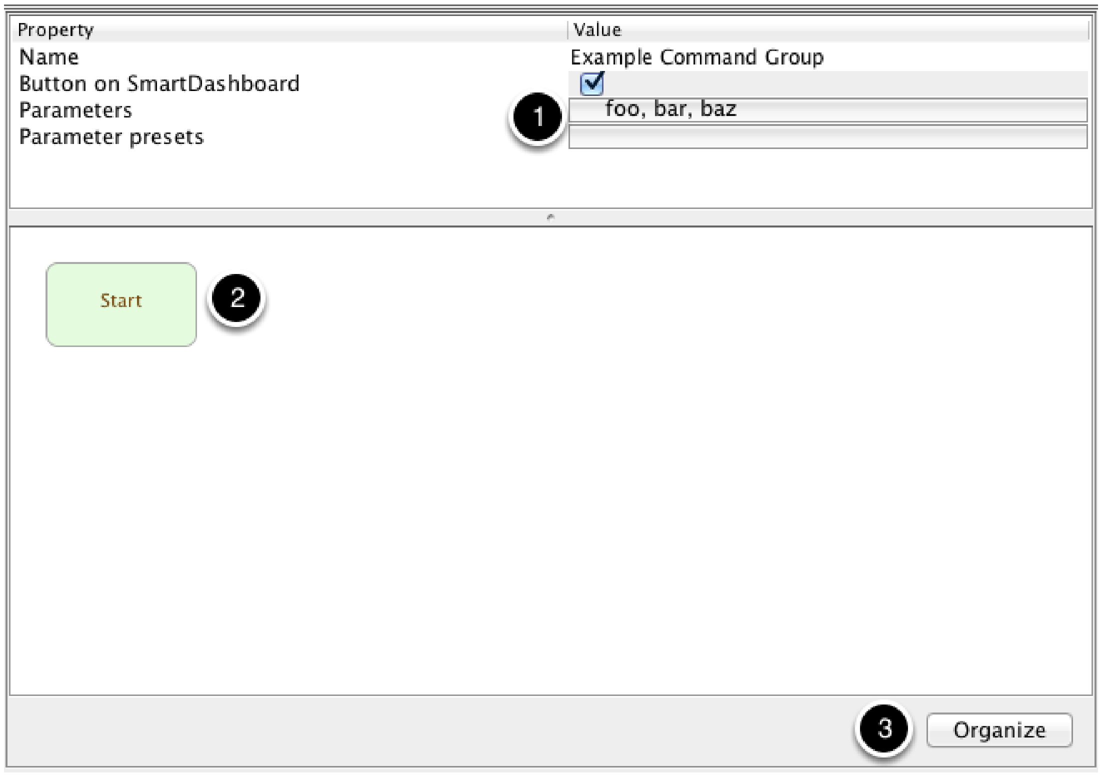


Property	Value
Name	Example Command Group
Button on SmartDashboard	✓
Parameters	foo, bar, baz
Parameter presets	

To create a CommandGroup:

1. Drag the command group from the palette to Commands folder in the robot description
2. Name the command group so that is has a meaningful name
3. Give the group any parameters it needs and presets for those parameters, just like with a normal command. These can be changed later at any time

Overview of the editor



Property	Value
Name	Example Command Group
Button on SmartDashboard	✓
Parameters	foo, bar, baz
Parameter presets	

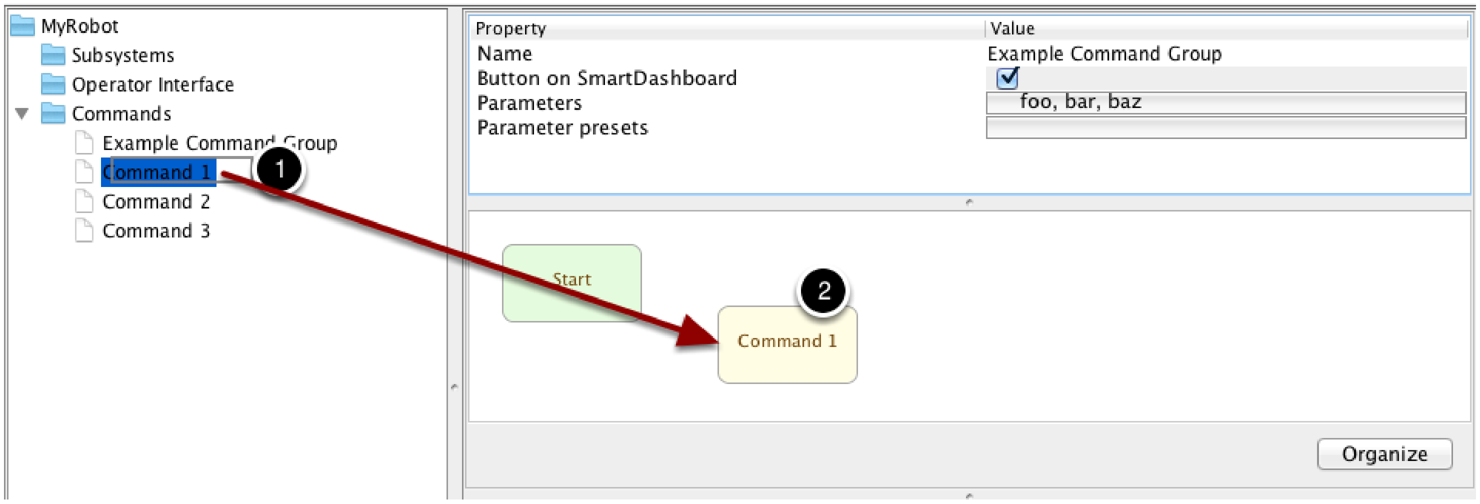
The elements of the command group editor are:

1. The properties of the command group.
2. The editor where commands are placed and connected.
3. The organize button will automatically organize all the commands in the editor

There are also a few differences between sequential and parallel commands in the editor.

- A command running sequentially can have only one command run sequentially after it
- A command running in parallel cannot have any commands run after it
- Sequential commands are colored light green and are placed at the top of the editor when it's organized
- Parallel commands are colored light blue and prefer to be placed below sequential commands

Adding commands



Property	Value
Name	Example Command Group
Button on SmartDashboard	✓
Parameters	foo, bar, baz
Parameter presets	

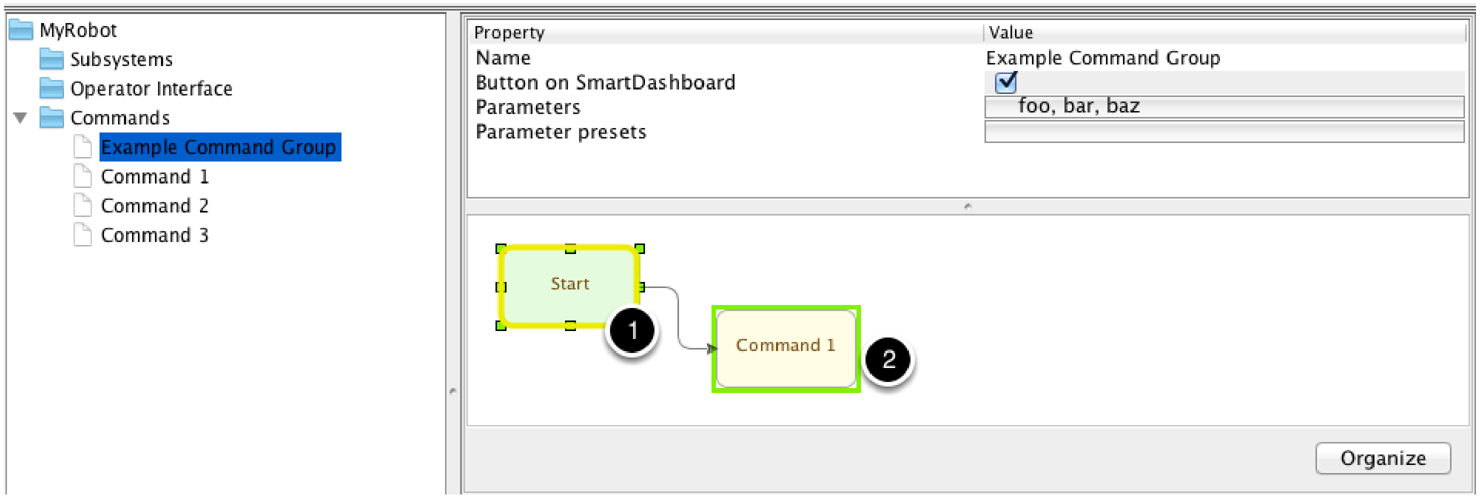
An empty command group isn't very useful, so to add commands to it:

1. Click a command in the palette ("Command 1" in this example)
2. Drag it somewhere into the graph editor

Alternatively, you can right-click anywhere on the editor and choose the command from the dropdown, or right click a command and choose a command to automatically be placed after the command you click.

Now you have a command in the group! But it's not connected to anything, so RobotBuilder doesn't know how it's supposed to behave.

Connecting commands



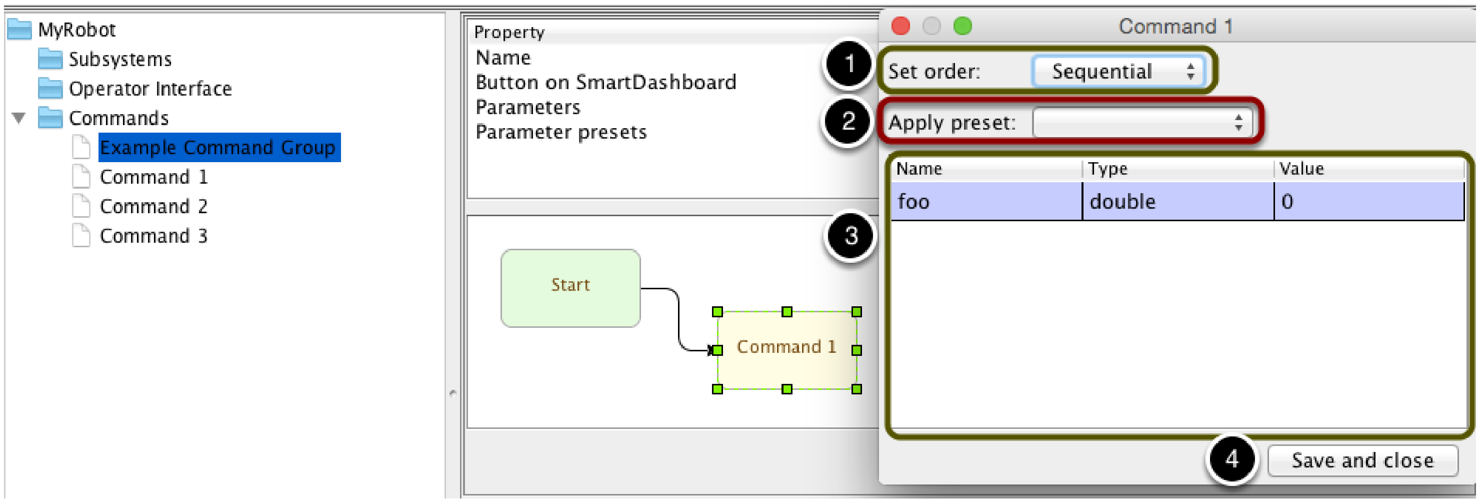
Property	Value
Name	Example Command Group
Button on SmartDashboard	✓
Parameters	foo, bar, baz
Parameter presets	

To connect commands and control when commands run in the group,

1. Click the center of the box for the previous command (or "Start" if you want it to run at the beginning). In this example, "Command 1" is the first command to run "Start" is clicked.
2. When a yellow box appears around the first box, drag your mouse to the command to run after it. This will create an arrow from the first box to your mouse
3. When a yellow box appears around the second box, you can release your mouse. The arrow will connect the two boxes and RobotBuilder will set the sequence of commands.

You can skip this step if you right-click a box and add a command through the menu -- Robotbuilder will automatically connect them.

Editing commands



Name	Type	Value
foo	double	0

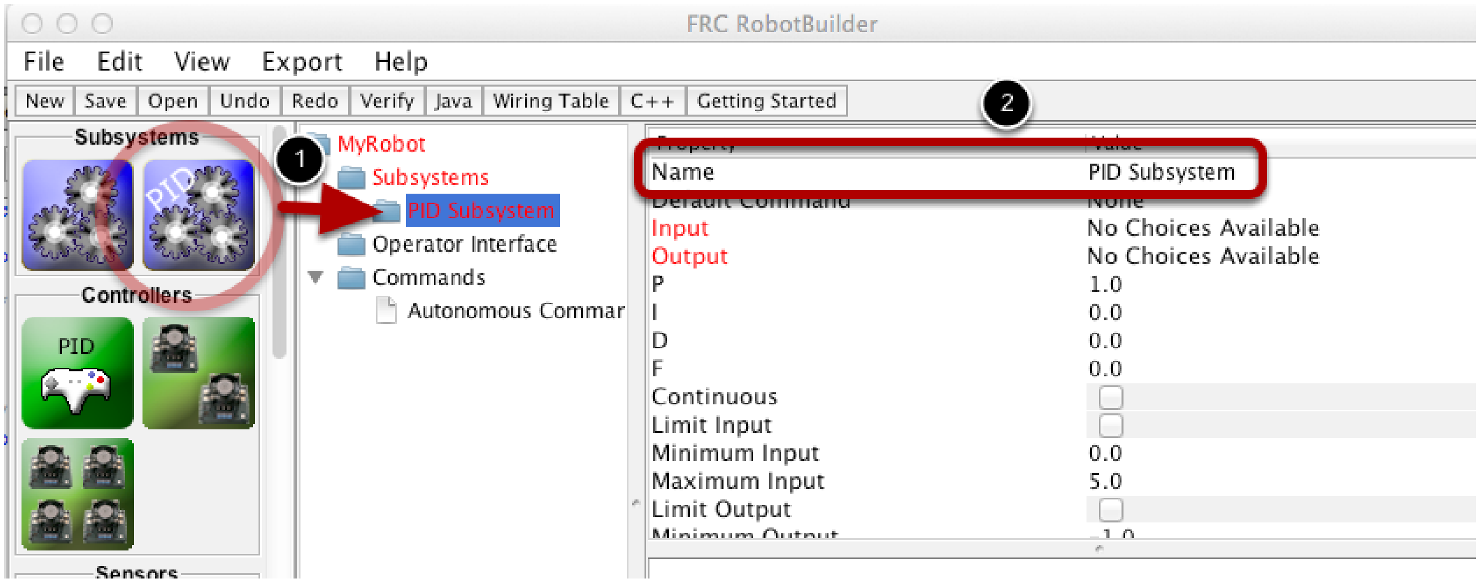
Double click a command to bring up the editor window.

1. Commands you drag in are automatically sequential. To change a command between running sequentially and in parallel, simply double-click on the command you want to edit and choose the order you want from the dropdown. You can also right-click on the command and choose "Set Parallel" or "Set Sequential" from the popup menu.
2. If the command has been configured to have presets, you can select one from this dropdown and Robotbuilder will fill out the values of the parameters.
3. The editor also allows you to edit the values of parameters that have been added to the command. Double-click the cell in the "Value" column to change it. The values of parameters can either be passed literally (e.g. the value for "foo" has been hardcoded to 0), or as a reference to a variable, method, or expression in the code. A reference starts with a dollar sign ('$'); everything after the dollar sign will be inserted into the code generated by RobotBuilder *exactly how it's typed*, so it's easy to misstype something and have the code not compile. The user is responsible for creating proper imports/ include statements and declaring variables and methods.
4. When you are done with editing, press the "Save and close" button to exit the editor and save the changes you've made to the command. If you don't want to save, press the close window button on the top of the window.

Using PIDSubsystems to control actuators with feedback from sensors

More advanced subsystems will use sensors for feedback to get guaranteed results for operations like setting elevator heights or wrist angles. The PIDSubsystem has a built-in PIDController to automatically set the correct setpoints for these types of mechanisms.

Create a PIDSubsystem



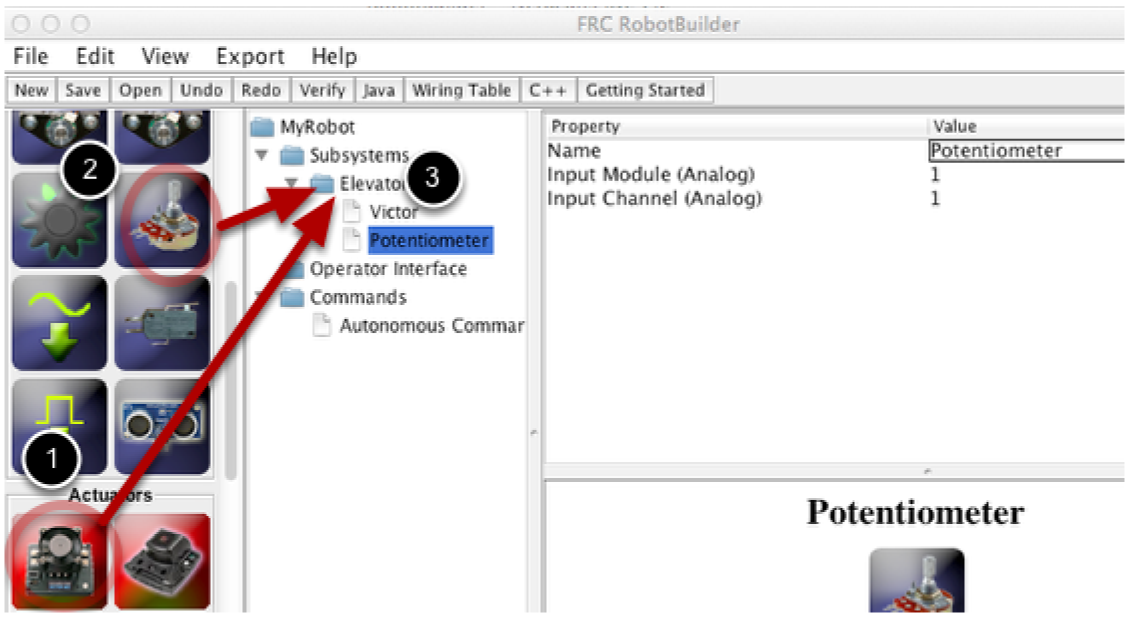
Property	Value
Name	PID Subsystem
Default Command	None
Input	No Choices Available
Output	No Choices Available
P	1.0
I	0.0
D	0.0
F	0.0
Continuous	☐
Limit Input	☐
Minimum Input	0.0
Maximum Input	5.0
Limit Output	☐
Minimum Output	-1.0

Creating a subsystem that uses feedback to control the position or speed of a mechanism is very easy.

1. Drag a PIDSubsystem from the palette to the Subsystems folder in the robot description
2. Rename the PID Subsystem to a more meaningful name for the subsystem

Notice that some of the parts of the robot description have turned red. This indicates that these components (the PIDSubsystem) haven't been completed and need to be filled in. The properties that are either missing or incorrect are shown in red.

Adding sensors and actuators to the PID Subsystem

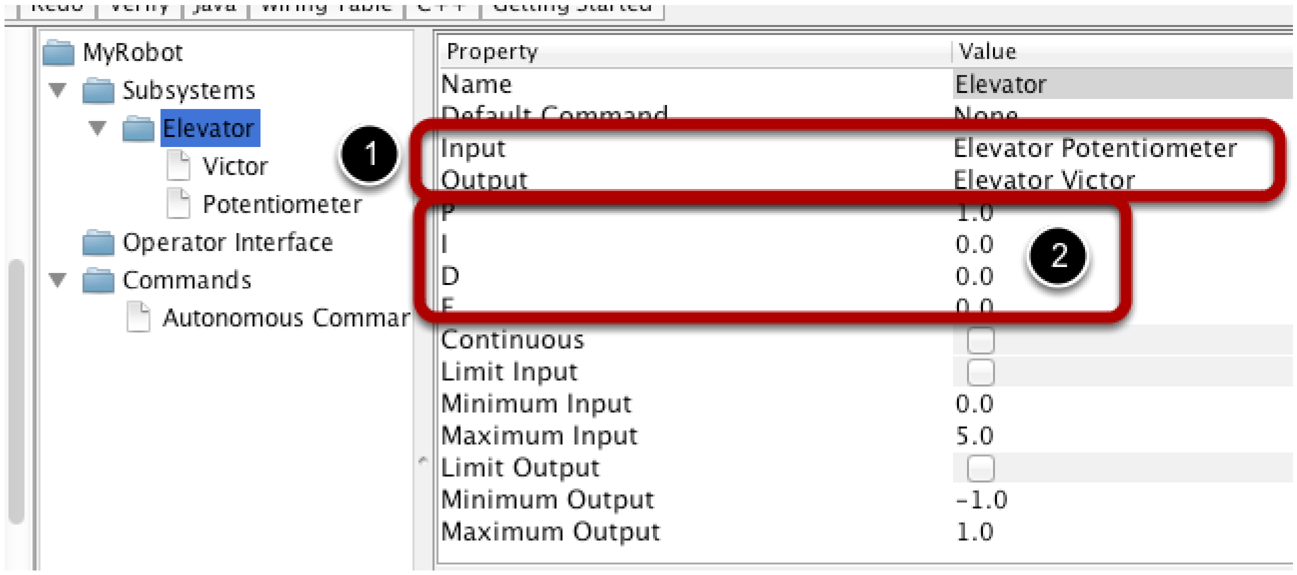


Property	Value
Name	Potentiometer
Input Module (Analog)	1
Input Channel (Analog)	1

Add the missing components for the PIDSubsystem

1. Drag in the actuator (a speed controller) to the particular subsystem - in this case the Elevator
2. Drag the sensor that will be used for feedback to the subsystem, in this case the sensor is a potentiometer that might give elevator height feedback.

Fill in the PIDSubsystem parameters to get the correct operation of the mechanism



Property	Value
Name	Elevator
Default Command	None
Input	Elevator Potentiometer
Output	Elevator Victor
P	1.0
I	0.0
D	0.0
F	0.0
Continuous	☐
Limit Input	☐
Minimum Input	0.0
Maximum Input	5.0
Limit Output	☐
Minimum Output	-1.0
Maximum Output	1.0

There a number of parameters for the PIDSubsystem but only a few need to be filled in for most cases

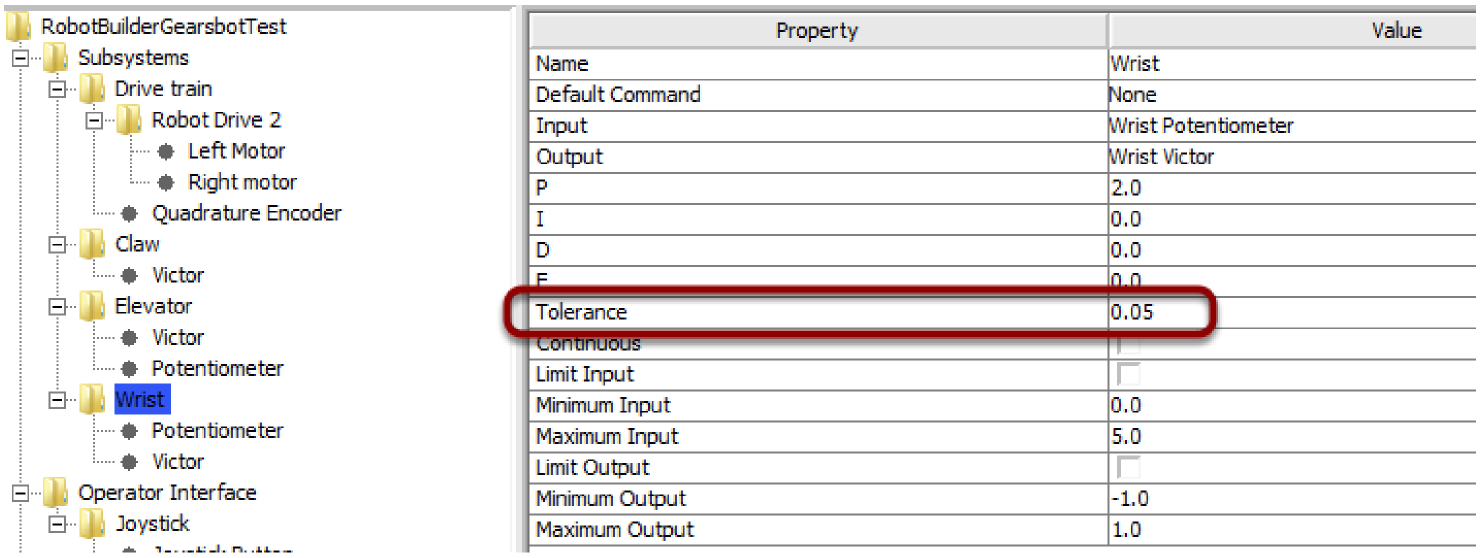
1. The Input and Output compents will have been filled in automatically from the previous step when the actuator and sensor were dragged into the PIDSubsystem
2. The P, I, and D values need to be filled in to get the desired sensitivity and stability of the component

See: Writing the code for a PIDSubystem in Java and Writing the code for a PIDSubsystem in C++

Setpoint command

A common use case in robot programs is to drive an actuator to a particular angle or position that is measured using a potentiometer or encoder. This happens so often that there is a shortcut in RobotBuilder to do this task. It is called the Setpoint command and it's one of the choices on the palette or the right-click context menu that can be inserted under "Commands".

Start with a PIDSubsystem

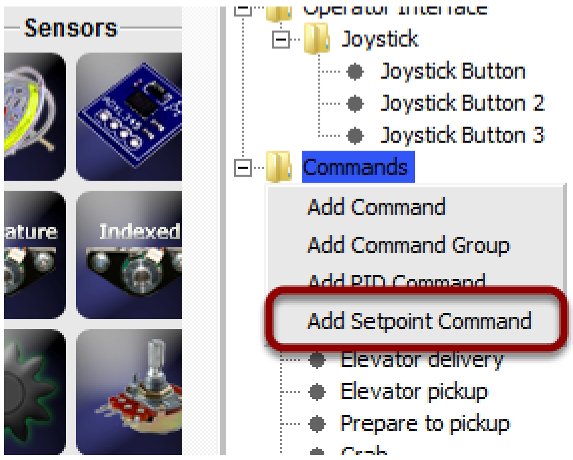


Property	Value
Name	Wrist
Default Command	None
Input	Wrist Potentiometer
Output	Wrist Victor
P	2.0
I	0.0
D	0.0
F	0.0
Tolerance	0.05
Continuous	
Limit Input	
Minimum Input	0.0
Maximum Input	5.0
Limit Output	
Minimum Output	-1.0
Maximum Output	1.0

Suppose in a robot there is a wrist joint with a potentiometer that measures the angle. First create a PIDSubsystem that include the motor that moves the wrist joint and the potentiometer that measures the angle. The PIDSubsystem should have all the PID constants filled in and working properly.

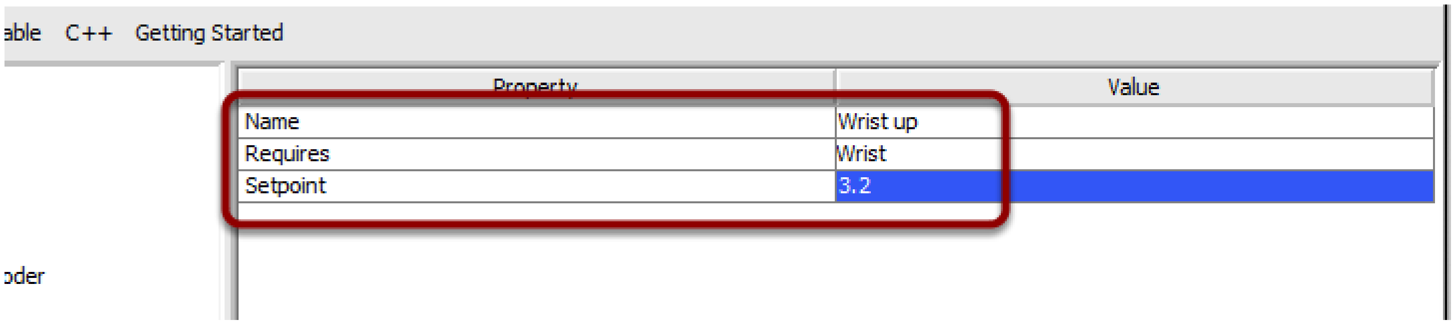
It is important to set the Tolerance parameter. This controls how far off the current value can be from the setpoint and be considered on target. This is the criteria that the SetpointCommand uses to move onto the next command.

Creating the Setpoint Command



Right-click on the Commands folder in the palette and select "Add Setpoint command".

Setpoint Command parameters



Property	Value
Name	Wrist up
Requires	Wrist
Setpoint	3.2

Fill in the name of the new command. The Requires field is the PIDSubsystem that is being driven to a setpoint and the Setpoint parameter is the setpoint value for the PIDSubsystem. There is no need to fill in any code for this command, it is automatically created by RobotBuilder.

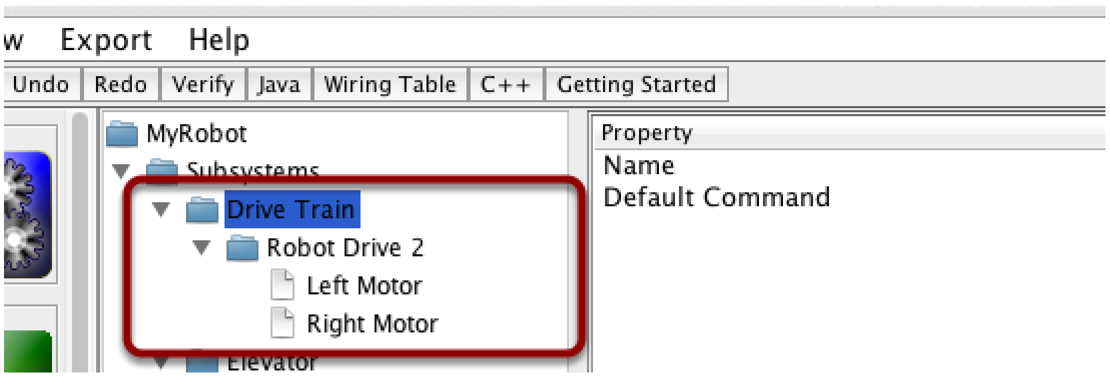
Whenever this command is scheduled, it will automatically drive the subsystem to the specified setpoint. When the setpoint is reached within the tolerance specified in the PIDSubsystem, the command ends and the next command starts. It is important to specify a tolerance in the PIDSubsystem or this command might never end because the tolerance is not achieved.

Driving the robot with tank drive and joysticks

A common use case is to have a joystick that should drive some actuators that are part of a subsystem. The problem is that the joystick is created in the OI class and the motors to be controlled are in the subsystem. The idea is to create a command that, when scheduled, reads input from the joystick and calls a method that is created on the subsystem that drives the motors.

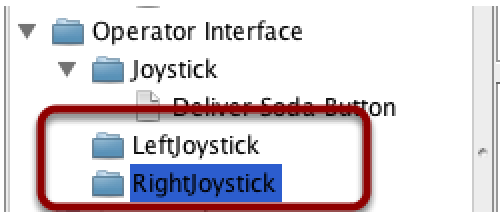
In this example a drive base subsystem is shown that is operated in tank drive using a pair of joysticks.

Create a Drive Train subsystem



Create a subsystem called Drive Train. Its responsibility will be to handle the driving for the robot base. Inside the Drive Train is a Robot Drive object for a two motor drive robot (in this case). There is a left motor and right motor as part of the Robot Drive 2 class.

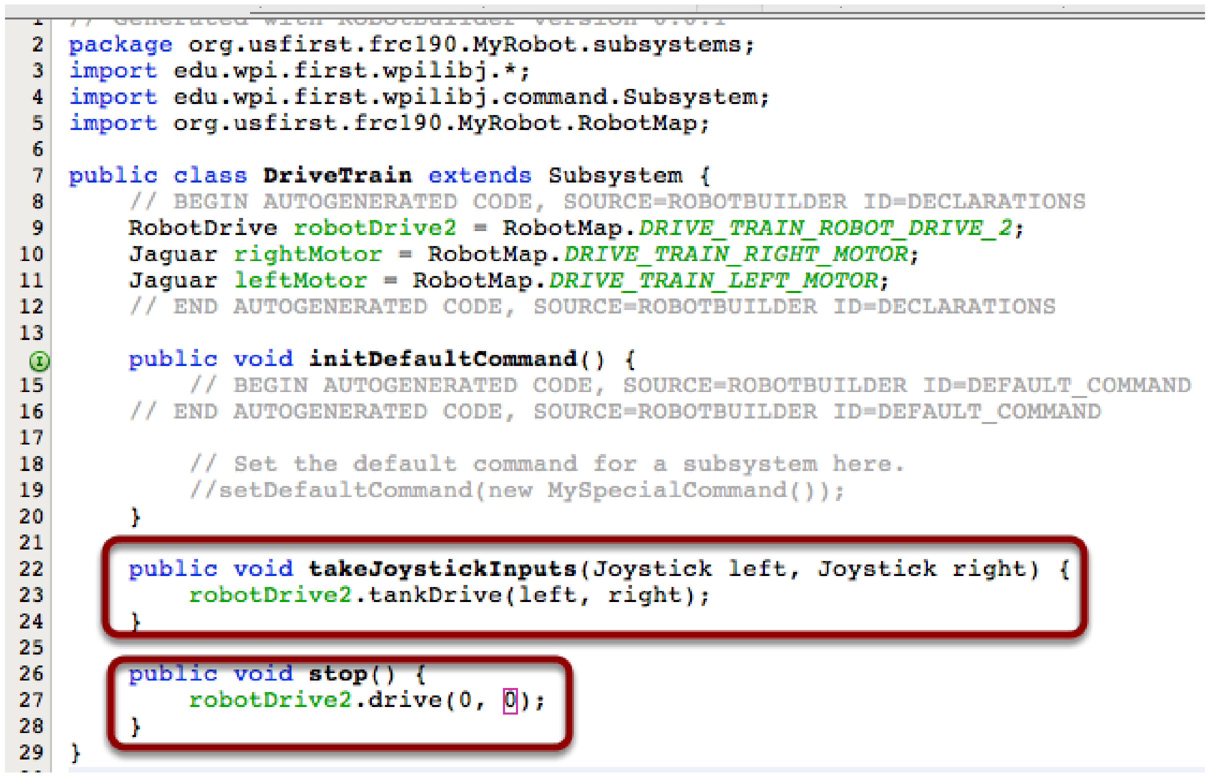
Add the joysticks to the Operator Interface



Add two joysticks to the Operator Interface, one is the left stick and the other is the right stick. The y-axis on the two joysticks are used to drive the robots left and right sides.

Note: be sure to export your program to C++ or Java before continuing to the next step.

Create a method to write the motors on the subsystem

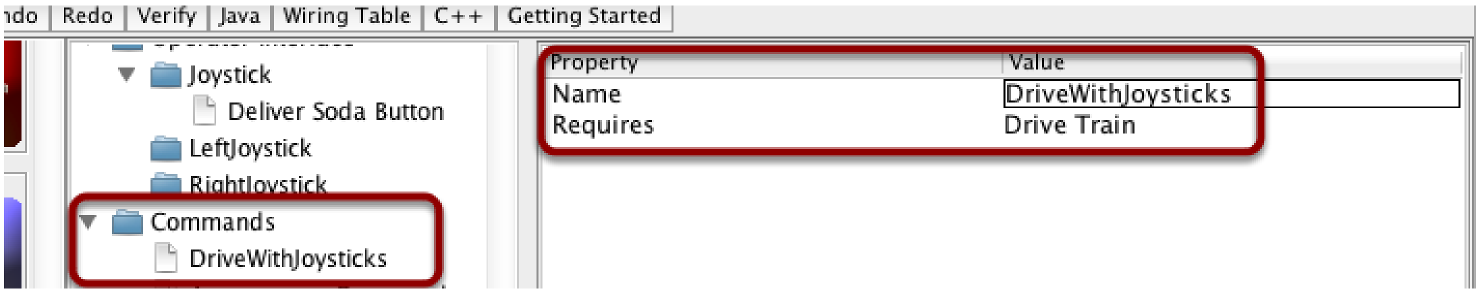


```java
package org.usfirst.frc190.MyRobot.subsystems;
import edu.wpi.first.wpilibj.*;
import edu.wpi.first.wpilibj.command.Subsystem;
import org.usfirst.frc190.MyRobot.RobotMap;

public class DriveTrain extends Subsystem {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    RobotDrive robotDrive2 = RobotMap.DRIVE_TRAIN_ROBOT_DRIVE_2;
    Jaguar rightMotor = RobotMap.DRIVE_TRAIN_RIGHT_MOTOR;
    Jaguar leftMotor = RobotMap.DRIVE_TRAIN_LEFT_MOTOR;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    public void initDefaultCommand() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT_COMMAND

        // Set the default command for a subsystem here.
        //setDefaultCommand(new MySpecialCommand());
    }

    public void takeJoystickInputs(Joystick left, Joystick right) {
        robotDrive2.tankDrive(left, right);
    }

    public void stop() {
        robotDrive2.drive(0, 0);
    }
}
```

Create a method that takes the joystick inputs, in this case the the left and right driver joystick. The values are passed to the RobotDrive object that in turn does tank steering using the joystick values. Also create a method called stop() that stops the robot from driving, this might come in handy later.

Note: the extra RobotBuilder comments have been removed to format the example for the documentation.

Create a command that reads the joystick values and calls the subsystem method

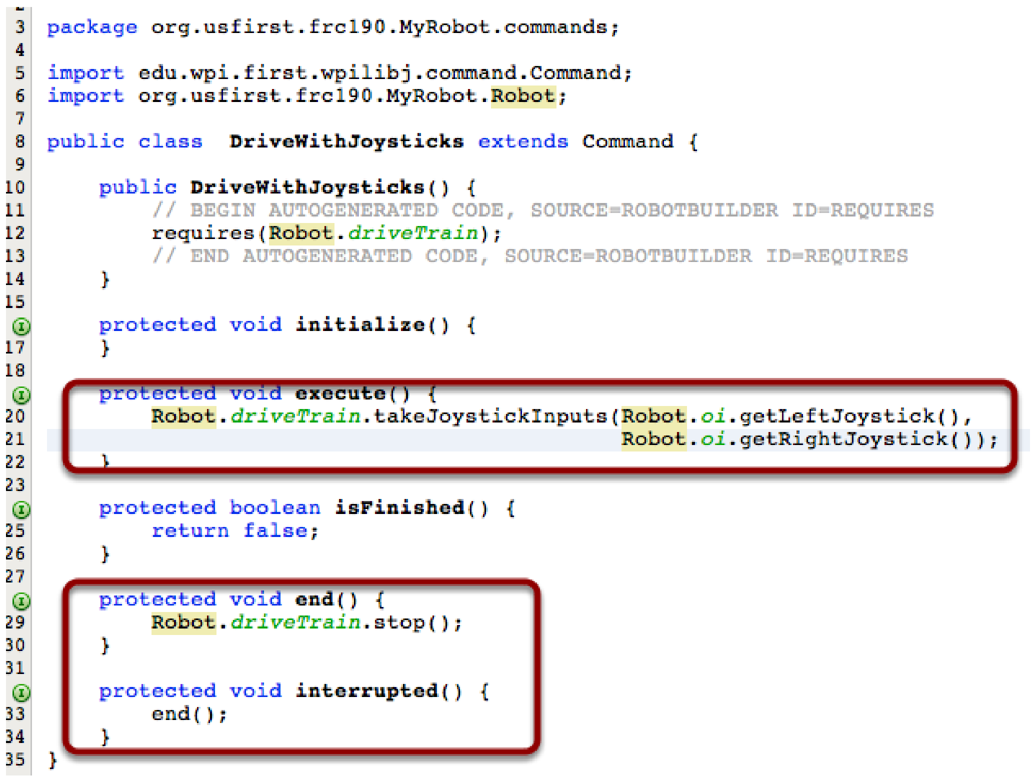


Property	Value
Name	DriveWithJoysticks
Requires	Drive Train

Create a command, in this case called DriveWithJoysticks. Its purpose will be to read the joystick values and send them to the Drive Base subsystem. Notice that this command Requires the Drive Train subsystem. This will cause it to stop running whenever anything else tries to use the Drive Train.

Note: be sure to export your program to C++ or Java before continuing to the next step.

Add the code for the command to do the actual driving

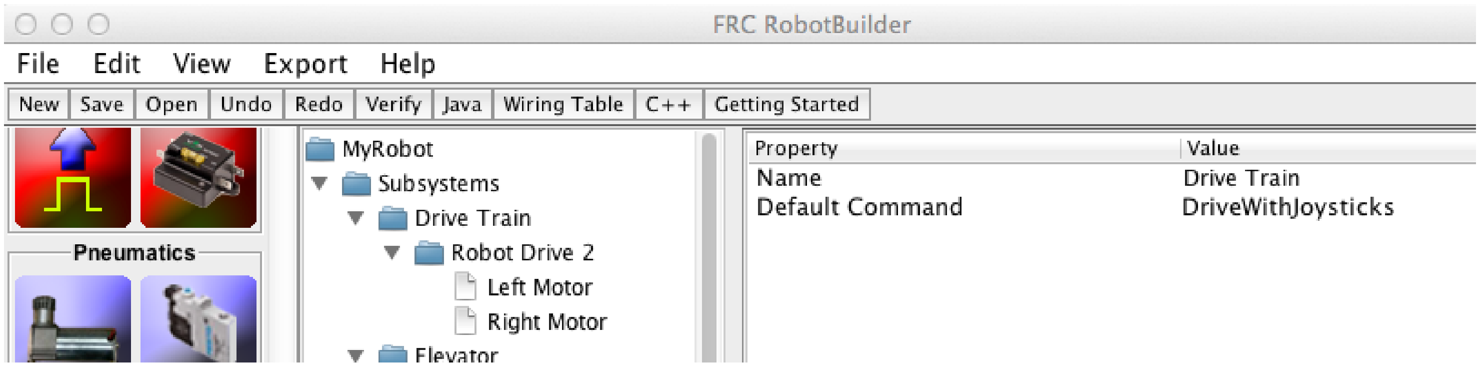


```
package org.usfirst.frc190.MyRobot.commands;

import edu.wpi.first.wpilibj.command.Command;
import org.usfirst.frc190.MyRobot.Robot;

public class  DriveWithJoysticks extends Command {

    public DriveWithJoysticks() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
        requires(Robot.driveTrain);
        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
    }

    protected void initialize() {
    }

    protected void execute() {
        Robot.driveTrain.takeJoystickInputs(Robot.oi.getLeftJoystick(),
                                            Robot.oi.getRightJoystick());
    }

    protected boolean isFinished() {
        return false;
    }

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

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

Add code to the execute method to do the actual driving. All that is needed is to get the Joystick objects for the left and right drive joysticks and pass them to the Drive Train subsystem. The subsystem just uses them for the tank steering method on its RobotDrive object. And we get tank steering.

We also filled in the end() and interrupted methods so that when this command is interrupted or stopped, the motors will be stopped as a safety precaution.

Make the command the "default command" for the subsystem



Property	Value
Name	Drive Train
Default Command	DriveWithJoysticks

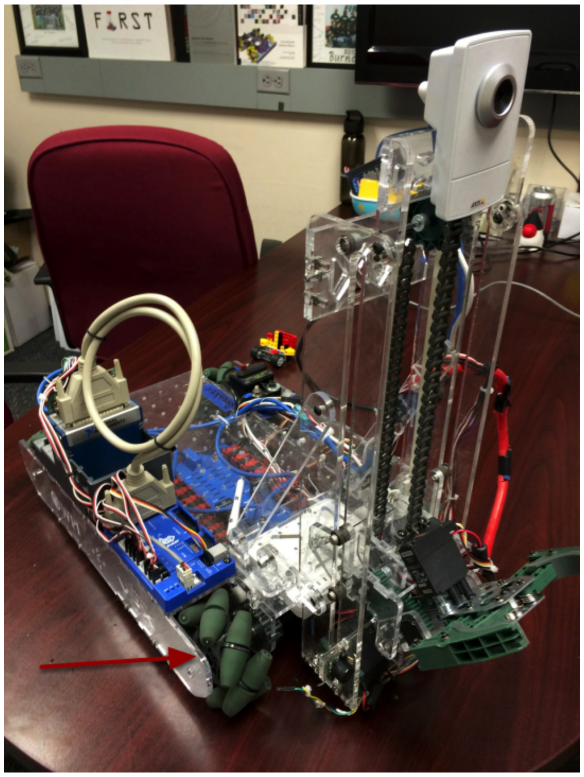
The last step is to make the DriveWIthJoysticks command be the "Default Command" for the Drive Train subsystem. This means that whenever no other command is using the Drive Train, the Joysticks will be in control. This is probably the desirable behavior. When the autonomous code is running, it will also require the drive train and interrupt the "DriveWithJoystick" command. When the autonomous code is finished, the DriveWithJoysticks command will restart automatically (because it is the default command), and the operators will be back in control. If you write any code that does teleop automatic driving, those commands should also "require" the DriveTrain so that they too will interrupt the DriveWithJoysticks command and have full control.

Note: be sure to export your program to C++ or Java before continuing.

Driving a robot using Mecanum drive

Mecanum drive is a method of driving using specially designed wheels that allow the robot to drive in any direction without changing the orientation of the robot. A robot with a conventional drivetrain (4 or six wheels) must turn in the direction it needs to drive. A mecanum robot can move in any direction without first turning and is called a holonomic drive.

Mecanum wheels



The wheels shown in this robot have rollers that cause the forces from driving to be applied at a 45 degree angle rather than straight forward as in the case of a conventional drive. You might guess that varying the speed of the wheels results in travel in any direction. You can look up how mecanum wheels work on various web sites on the internet.

Code for driving with mecanum wheels

```
#include "WPILib.h"
/**
```

```
 * Simplest program to drive a robot with mecanum drive using a single Logitech
 * Extreme 3D Pro joystick and 4 drive motors connected as follows:
 *   - Digital Sidecar 1:
 *     - PWM 1 - Connected to front left drive motor
 *     - PWM 2 - Connected to rear left drive motor
 *     - PWM 3 - Connected to front right drive motor
 *     - PWM 4 - Connected to rear right drive motor
 */
class MecanumDefaultCode : public IterativeRobot
{
        RobotDrive *m_robotDrive;                // RobotDrive object using PWM 1-4 for
drive motors
        Joystick *m_driveStick;                        // Joystick object on USB port 1
(mecanum drive)
public:
        /**
         * Constructor for this "MecanumDefaultCode" Class.
         */
        MecanumDefaultCode(void)
        {
                // Create a RobotDrive object using PWMS 1, 2, 3, and 4
                m_robotDrive = new RobotDrive(1, 2, 3, 4);
                // Define joystick being used at USB port #1 on the Drivers Station
                m_driveStick = new Joystick(1);
                // Twist is on Axis 3 for the Extreme 3D Pro
                m_driveStick->SetAxisChannel(Joystick::kTwistAxis, 3);
        }
        /**
         * Gets called once for each new packet from the DS.
         */
        void TeleopPeriodic(void)
        {
                m_robotDrive->MecanumDrive_Cartesian(m_driveStick->GetX(), m_driveStick-
>GetY(), m_driveStick->GetTwist());
        }
};
START_ROBOT_CLASS(MecanumDefaultCode);
```

Here's a sample program that shows the minimum code to drive using a single joystick and mecanum wheels. It uses the RobotDrive object that is available in both C++ and Java so even though this example is in C++ similar code will work in Java. The idea is to create the RobotDrive object with 4 PWM ports for the 4 speed controllers on the robot. The joystick XY position represents a direction vector that the robot should follow regardless of its orientation. The twist axis on the joystick represents the rate of rotation for the robot while it's driving.

Thanks to FRC Team 2468 in Austin, TX for developing this example.

Updating the program for field-oriented driving

I would be remiss in not mentioning that is a 4th parameter to the MecanumDrive_Cartesian() method that is the angle returned from a Gyro sensor. This will adjust the rotation value supplied, in this case, from the twist axis of the joystick to be relative to the field rather than relative to the robot. This is particularly useful with mecanum drive since, for the purposes of steering, the robot really has no front, back or sides. It can go in any direction. Adding the angle in degrees from a gyro object will cause the robot to move away from the drivers when the joystick is pushed forwards, and towards the drivers when it is pulled towards them - regardless of what direction the robot is facing!

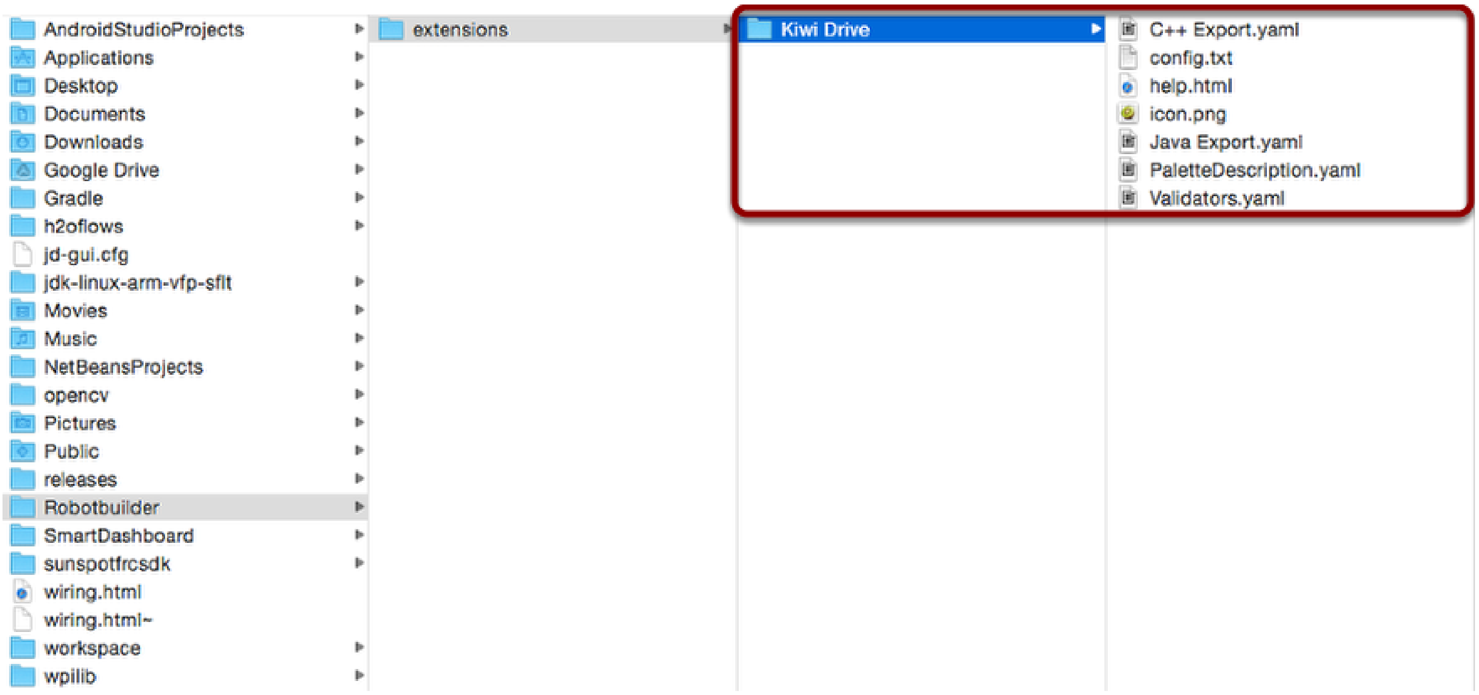
The use of field-oriented driving makes often makes the robot much easier to drive, especially compared to a "robot-oriented" drive system where the controls are reversed when the robot is facing the drivers.

Just remember to get the gyro angle each time MecanumDrive_Caresian() is called.

Adding custom components

RobotBuilder works very well for creating robot programs that just use WPILib for motors, controllers, and sensors. But for teams that use custom classes, RobotBuilder doesn't have any support for those classes, so a few steps need to be taken to use them in RobotBuilder.

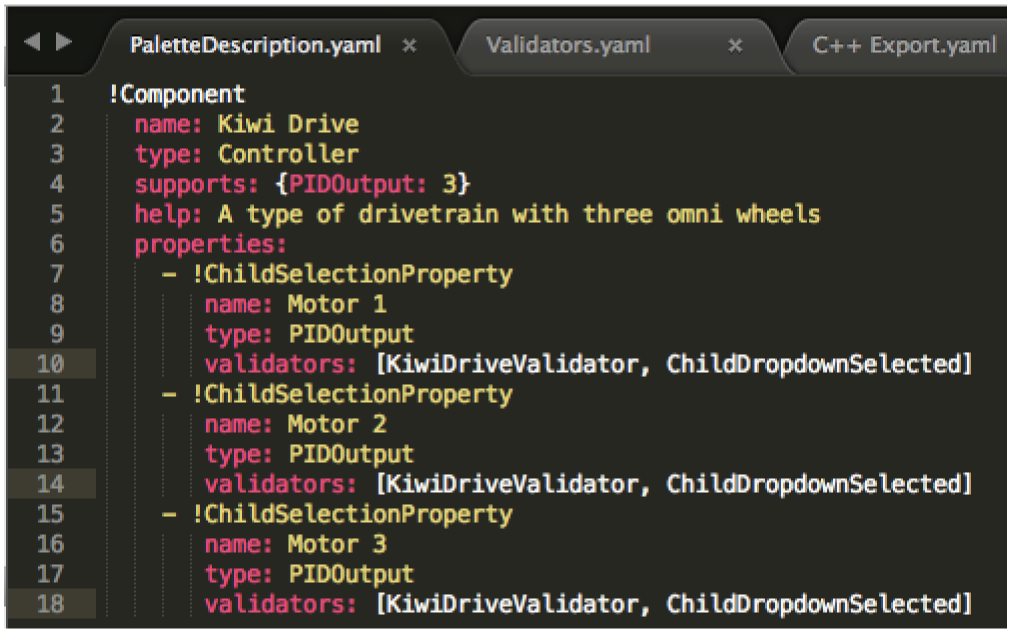
The structure of a custom component



Custom components all go in $USER_HOME/Robotbuilder/extensions. On Linux and Mac, $USER_HOME will be /Users/yourusername/. On Windows, it will be C:\Users\yourusername\

There are seven files and one folder that are needed for a custom component. The folder contains the files describing the component and how to export it. It should have the same name as the component (e.g."Kiwi Drive" for a kiwi drive controller, "Robot Drive 6" for a six-motor drive controller, etc.). The files should have the same names and extensions as the ones shown here. Other files can be in the folder along with these seven, but the seven must be present for RobotBuilder to recognize the custom component.

PaletteDescription.yaml



```yaml
!Component
  name: Kiwi Drive
  type: Controller
  supports: {PIDOutput: 3}
  help: A type of drivetrain with three omni wheels
  properties:
    - !ChildSelectionProperty
      name: Motor 1
      type: PIDOutput
      validators: [KiwiDriveValidator, ChildDropdownSelected]
    - !ChildSelectionProperty
      name: Motor 2
      type: PIDOutput
      validators: [KiwiDriveValidator, ChildDropdownSelected]
    - !ChildSelectionProperty
      name: Motor 3
      type: PIDOutput
      validators: [KiwiDriveValidator, ChildDropdownSelected]
```

Line-by-line:

- !Component: Declares the beginning of a new component
- name: The name of the component. This is what will show up in the palette/tree -- this should also be the same as the name of the containing folder
- type: the type of the component (these will be explained in depth later on)
- supports: a map of the amount of each type of component this can support. Motor controllers in RobotBuilder are all PIDOutputs, so a kiwi drive can support three PIDOutputs. If a component doesn't support anything (such as sensors or motor controllers), just leave this line out
- help: a short string that gives a helpful message when one of these components is hovered over
- properties: a list of the properties of this component. In this kiwi drive example, there are three very similar properties, one for each motor. A ChildSelectionProperty allows the user to choose a component of the given type from the subcomponents of the one being edited (so here, they would show a dropdown asking for a PIDOutput - i.e. a motor controller - that has been added to the kiwi drive)

The types of component RobotBuilder supports (these are case-sensitive):

- Command

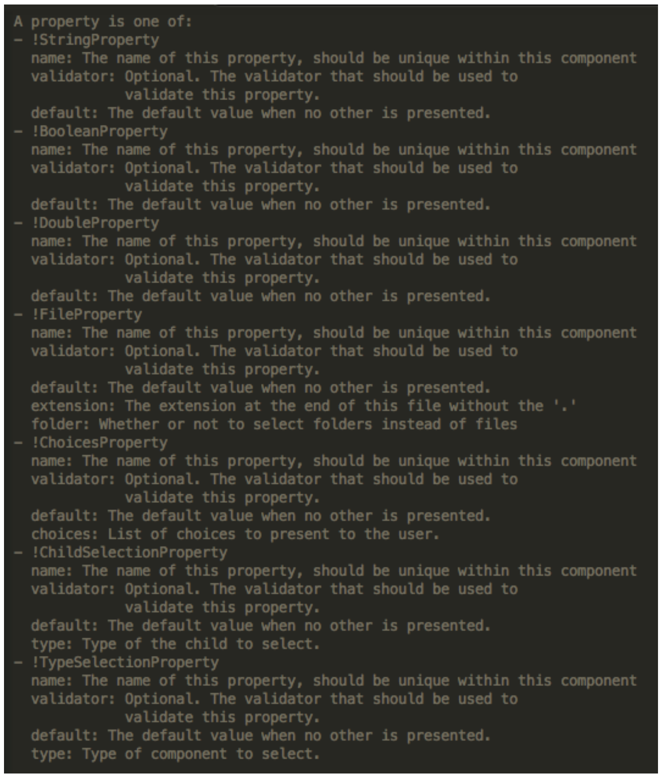
- Subsystem
- PIDOutput (speed controller)
- PIDSource (sensor that implements PIDSource e.g. analog potentiometer, encoder)
- Sensor (sensor that does not implement PIDSource e.g. limit switch)
- Controller (robot drive, PID controller, etc.)
- Actuator (an output that is not a motor, e.g. solenoid, servo)
- Joystick
- Joystick Button

Properties

The properties relevant for a custom component:

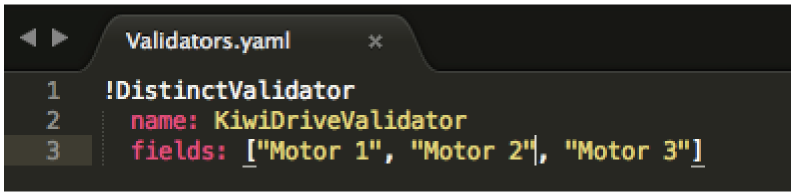
- StringProperty: used when a component needs a string e.g. the name of the component
- BooleanProperty: used when a component needs a boolean value e.g. putting a button on the SmartDashboard
- DoubleProperty: used when a component needs a number value e.g. PID constantsChoicesProperty
- ChildSelectionProperty: used when you need to choose a child component e.g. speed controllers in a RobotDrive
- TypeSelectionProperty: used when you need to choose any component of the given type from anywhere in the program e.g. input and output for a PID command

The fields for each property are described below:



```
A property is one of:
- !StringProperty
  name: The name of this property, should be unique within this component
  validator: Optional. The validator that should be used to
             validate this property.
  default: The default value when no other is presented.
- !BooleanProperty
  name: The name of this property, should be unique within this component
  validator: Optional. The validator that should be used to
             validate this property.
  default: The default value when no other is presented.
- !DoubleProperty
  name: The name of this property, should be unique within this component
  validator: Optional. The validator that should be used to
             validate this property.
  default: The default value when no other is presented.
- !FileProperty
  name: The name of this property, should be unique within this component
  validator: Optional. The validator that should be used to
             validate this property.
  default: The default value when no other is presented.
  extension: The extension at the end of this file without the '.'
  folder: Whether or not to select folders instead of files
- !ChoicesProperty
  name: The name of this property, should be unique within this component
  validator: Optional. The validator that should be used to
             validate this property.
  default: The default value when no other is presented.
  choices: List of choices to present to the user.
- !ChildSelectionProperty
  name: The name of this property, should be unique within this component
  validator: Optional. The validator that should be used to
             validate this property.
  default: The default value when no other is presented.
  type: Type of the child to select.
- !TypeSelectionProperty
  name: The name of this property, should be unique within this component
  validator: Optional. The validator that should be used to
             validate this property.
  default: The default value when no other is presented.
  type: Type of component to select.
```

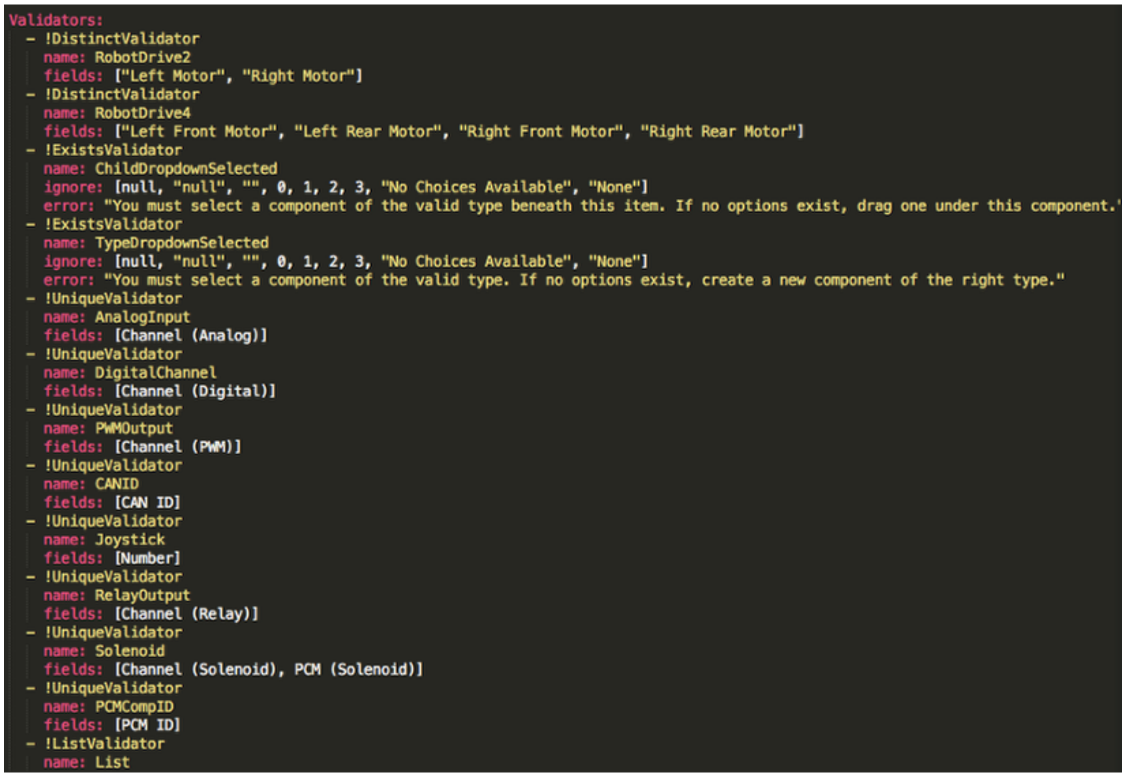
Validators.yaml



```
!DistinctValidator
  name: KiwiDriveValidator
  fields: ["Motor 1", "Motor 2", "Motor 3"]
```

You may have noticed "KiwiDriveValidator" in the validators entry of each of the motor properties in PaletteDescription.yaml. It's not a built-in validator, so it had to be defined in Validators.yaml. This example validator is very simple - it just makes sure that each of the named fields has a different value than the others.

Built in validators and validator types

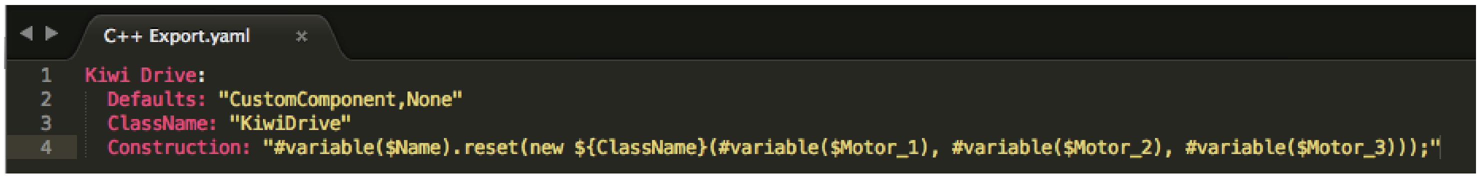


```
Validators:
  - !DistinctValidator
    name: RobotDrive2
    fields: ["Left Motor", "Right Motor"]
  - !DistinctValidator
    name: RobotDrive4
    fields: ["Left Front Motor", "Left Rear Motor", "Right Front Motor", "Right Rear Motor"]
  - !ExistsValidator
    name: ChildDropdownSelected
    ignore: [null, "null", "", 0, 1, 2, 3, "No Choices Available", "None"]
    error: "You must select a component of the valid type beneath this item. If no options exist, drag one under this component.'
  - !ExistsValidator
    name: TypeDropdownSelected
    ignore: [null, "null", "", 0, 1, 2, 3, "No Choices Available", "None"]
    error: "You must select a component of the valid type. If no options exist, create a new component of the right type."
  - !UniqueValidator
    name: AnalogInput
    fields: [Channel (Analog)]
  - !UniqueValidator
    name: DigitalChannel
    fields: [Channel (Digital)]
  - !UniqueValidator
    name: PWMOutput
    fields: [Channel (PWM)]
  - !UniqueValidator
    name: CANID
    fields: [CAN ID]
  - !UniqueValidator
    name: Joystick
    fields: [Number]
  - !UniqueValidator
    name: RelayOutput
    fields: [Channel (Relay)]
  - !UniqueValidator
    name: Solenoid
    fields: [Channel (Solenoid), PCM (Solenoid)]
  - !UniqueValidator
    name: PCMCompID
    fields: [PCM ID]
  - !ListValidator
    name: List
```

The built-in validators are very useful (especially the UniqueValidators for port/channel use), but sometimes a custom validator is needed, like in the previous step

- DistinctValidator: Makes sure the values of each of the given fields are unique
- ExistsValidator: Makes sure that a value has been set for the property using this validator
- UniqueValidator: Makes sure that the value for the property is unique globally for the given fields
- ListValidator: Makes sure that all the values in a list property are valid

C++ Export.yaml



C++ Export.yaml

```
Kiwi Drive:
  Defaults: "CustomComponent,None"
  ClassName: "KiwiDrive"
  Construction: "#variable($Name).reset(new ${ClassName}(#variable($Motor_1), #variable($Motor_2), #variable($Motor_3)));"
```

A line-by-line breakdown of the file:

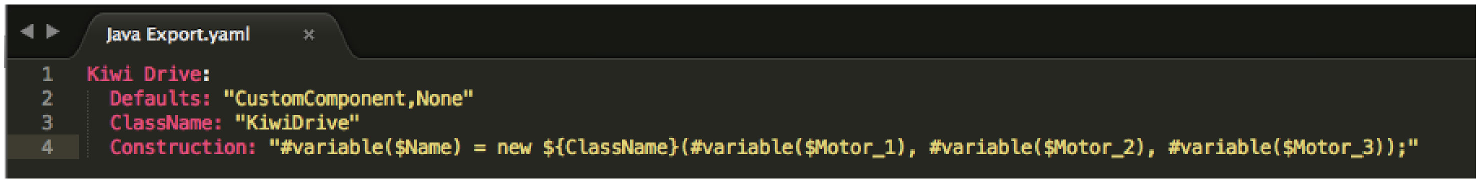
1. Kiwi Drive: the name of the component being exported. This is the same as the name set in PaletteDescription.yaml, and the name of the folder containing this file
2. Defaults: provides some default values for includes needed by this component, the name of the class, a construction template, and more. The CustomComponent default adds an include for "Custom/${ClassName}.h" to every generated file that uses the component (e.g. RobotDrive.h would have #include "Custom/KiwiDrive.h" at the top of the file)
3. ClassName: the name of the custom class you're adding.
4. Construction: an instruction for how the component should be constructed. Variables will be replaced with their values ("${ClassName}" will be replaced with "KiwiDrive"), then macros will be evaluated (for example, "#variable($Name)" may be replaced with "drivebaseKiwiDrive").

This example expects a KiwiDrive class with the constructor

```
KiwiDrive(shared_ptr<SpeedController>, shared_ptr<SpeedController>,
shared_ptr<SpeedController>)
```

If your team uses Java, this file can be empty.

Java Export.yaml



```
Kiwi Drive:
  Defaults: "CustomComponent,None"
  ClassName: "KiwiDrive"
  Construction: "#variable($Name) = new ${ClassName}(#variable($Motor_1), #variable($Motor_2), #variable($Motor_3));"
```

Very similar to the C++ export file; the only difference should be the Construction line. This example expects a KiwiDrive class with the constructor

```
KiwiDrive(SpeedController, SpeedController, SpeedController)
```

If your team uses C++, this file can be empty.

Using macros and variables

Macros are simple functions that RobotBuilder uses to turn variables into text that will be inserted into generated code. They always start with the "#" symbol, and have a syntax similar to functions:

<macro_name>(arg0, arg1, arg2, ...). The only macro you'll probably need to use is "#variable(component_name)"

"#variable" takes a string, usually the a variable defined somewhere (i.e. "Name" is the name given to the component in RobotBuilder, such as "Arm Motor"), and turns it into the name of a variable defined in the generated code. For example, #variable("Arm Motor") results in the string "ArmMotor"

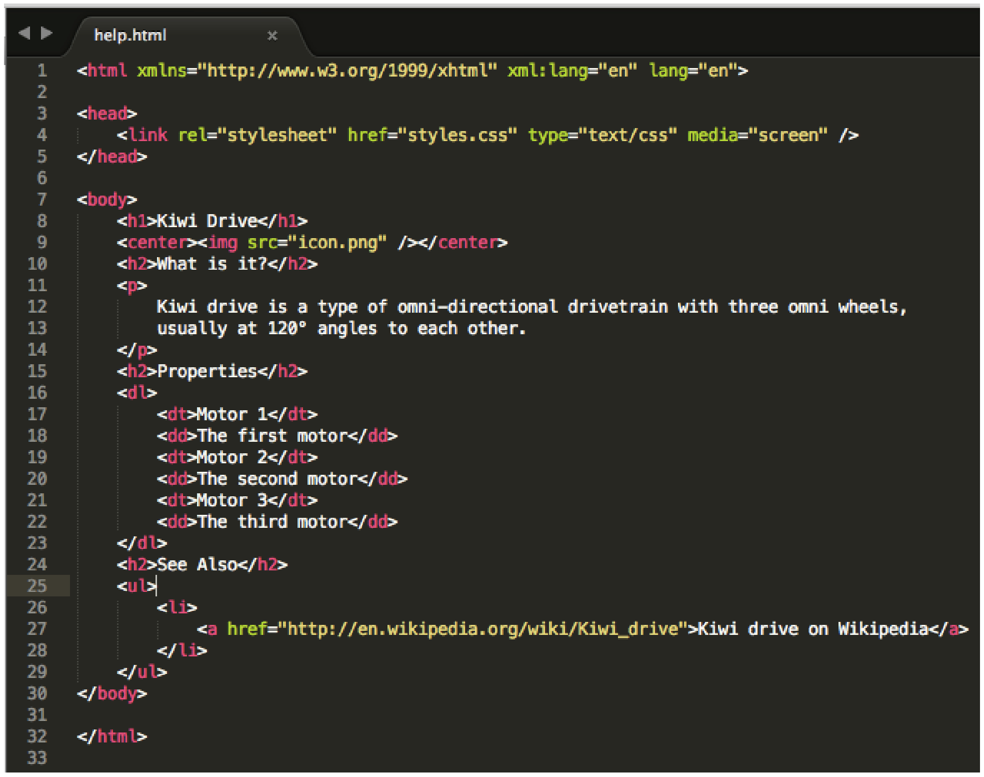
Variables are referenced by placing a dollar sign ("$") in front of the variable name, which an optionally be placed inside curly braces to easily distinguish the variable from other text in the file. When the file is parsed, the dollar sign, variable name, and curly braces are replaced with the value of the variable (e.g. "${ClassName}" is replaced with "KiwiDrive").

Variables are either component properties (e.g. "Motor 1", "Motor 2", "Motor 3" in the kiwi drive example), or one of the following:

1. Short_Name: the name given to the component in the editor panel in RobotBuilder
2. Name: the full name of the component. If the component is in a subsystem, this will be the short name appended to the name of the subsystem
3. Export: The name of the file this component should be created in, if any. This should be "RobotMap" for components like actuators, controllers, and sensors; or "OI" for things like gamepads or other custom OI components. Note that the "CustomComponent" default will export to the RobotMap.
4. Import: Files that need to be included or imported for this component to be able to be used.
5. Declaration: an instruction, similar to Construction, for how to declare a variable of this component type. This is taken care of by the default "None"
6. Construction: an instruction for how to create a new instance of this component
7. LiveWindow: an instruction for how to add this component to the LiveWindow
8. Extra: instructions for any extra functions or method calls for this component to behave correctly, such as encoders needing to set the encoding type.
9. Prototype (C++ only): The prototype for a function to be created in the file the component is declared in, typically a getter in the OI class
10. Function: A function to be created in the file the component is declared in, typically a getter in the OI class
11. PID: An instruction for how to get the PID output of the component, if it has one (e.g. "#variable($Short_Name)->PIDGet()")
12. ClassName: The name of the class that the component represents (e.g. "KiwiDrive" or "Joystick")

If you have variables with spaces in the name (such as "Motor 1", "Right Front Motor", etc.), the spaces need to be replaced with underscores when using them in the export files.

help.html

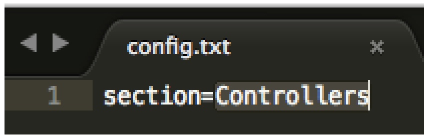


help.html

```html
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
    <link rel="stylesheet" href="styles.css" type="text/css" media="screen" />
</head>

<body>
    <h1>Kiwi Drive</h1>
    <center><img src="icon.png" /></center>
    <h2>What is it?</h2>
    <p>
        Kiwi drive is a type of omni-directional drivetrain with three omni wheels,
        usually at 120° angles to each other.
    </p>
    <h2>Properties</h2>
    <dl>
        <dt>Motor 1</dt>
        <dd>The first motor</dd>
        <dt>Motor 2</dt>
        <dd>The second motor</dd>
        <dt>Motor 3</dt>
        <dd>The third motor</dd>
    </dl>
    <h2>See Also</h2>
    <ul>
        <li>
            <a href="http://en.wikipedia.org/wiki/Kiwi_drive">Kiwi drive on Wikipedia</a>
        </li>
    </ul>
</body>

</html>

```

A HTML file giving information on the component. It is better to have this be as detailed as possible, though it certainly isn't necessary if the programmer(s) are familiar enough with the component, or if it's so simple that there's little point in a detailed description.

config.txt



config.txt

```
section=Controllers
```

A configuration file to hold miscellaneous information about the component. Currently, this only has the section of the palette to put the component in.

The sections of the palette (these are case sensitive):

- Subsystems
- Controllers
- Sensors
- Actuators
- Pneumatics
- OI
- Commands

icon.png

The icon that shows up in the palette and the help page. This should be a 64x64 .png file. It should use the color scheme and general style of the section it's in to avoid visual clutter, but this is entirely optional.