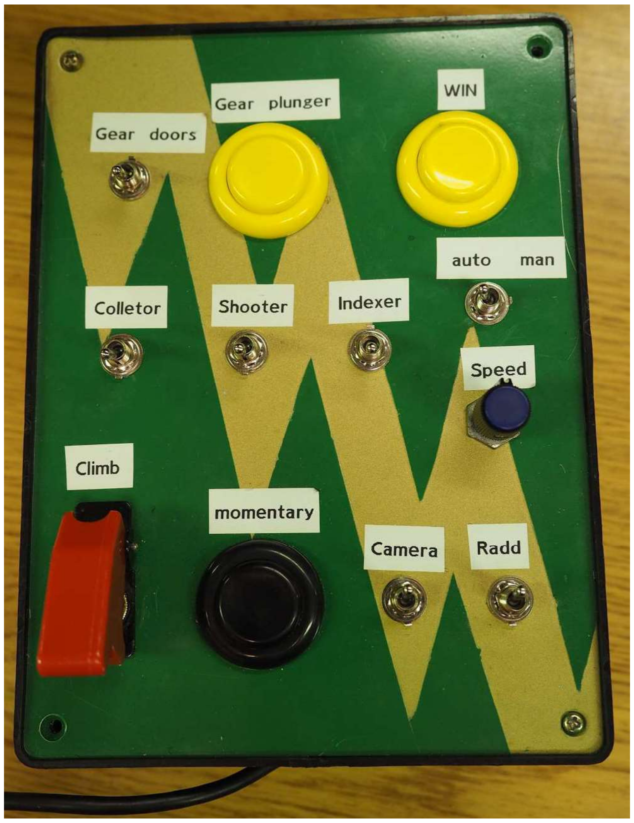
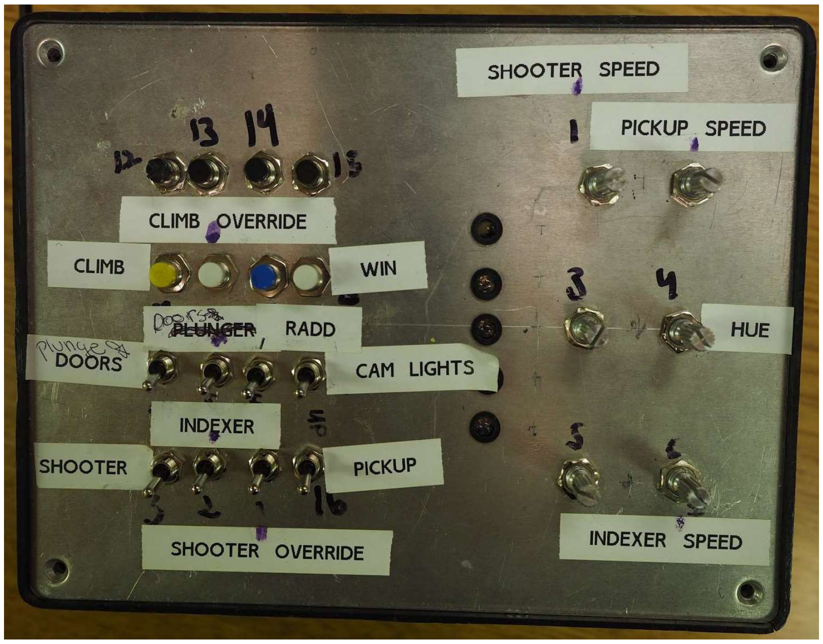
Button Box

The button box is a separate device that emulates a joystick. It allows us to organize functions effectively and have a separate person controlling those functions, not just the driver which cuts down on time delay. The button box is designed by the programming team and built by the electrical. In case of emergencies there is a spare button box which was used to design the competition button box early on. The spare button box is pictured on the right and this years is on the left.





Each button has a different function. For example the switch labeled "gear doors" opens and closes the gear doors. There are three different physical types of button on a button box: press, toggle and pot. A press button is one like the "gear plunger" button, a toggle button is on like the "gear doors" button and a pot is one like the "speed button". Pots read in a value from negative one to one inclusive with negative one on the far left. Toggle and press buttons can cause methods either in one of three ways: when pressed, while held and when released. The when pressed function causes a method to run once when the button is initially pressed. The while held function causes a method to run continuously while a button is held. The when released function causes a method to run once when the button is initially released.

Vision

Key Features: Angle of Incidence, Angle of Robot, Distance and Suggested Angle of Rotation

What is Vision?

Vision is shorthand for the vision recognition and processing system part of our robot, used primarily during the autonomous period. At the beginning of the build season, this was one of several options chosen in order to allow for gear placement during autonomous. In the competition robot, this code was not implemented in favor of using exact measurements which offered more precise and consistent scoring during the autonomous period. Nonetheless, vision is an almost fully developed system which allows for real time calculations on distance and angle from the gear peg.

RoboRealm

The first option which was explored was RoboRealm. A quick first draft of code was drawn up in visual basic which could find the distance and angle to the target with varying consistency, but was later discontinued in favor of the use of GRIP.

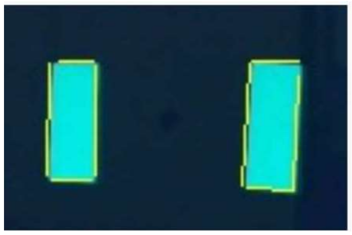
GRIP

This was the vision processing tool chosen for our final vision recognition code. In deciding whether to use GRIP or RoboRealm, we chose the more flexible option. GRIP was available to be used on both the Kangaroo, a independent processing unit separate from the roboRIO itself, as well as on the driver station.

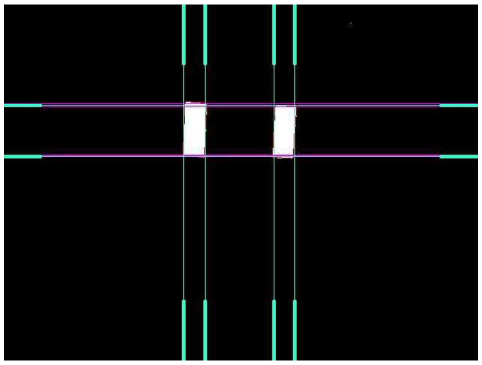
FIRST placed reflective tape equidistant on each side of the gear peg. Our focus for vision was to see those strips and be able to place a gear off of the information. To clearly differentiate the tape from the airship, we shone a green light on the tape.

Our first task was to attempt to eliminate any parts of the picture that was not the target. In order to accomplish this, we manipulated the HSL (Hue, Saturation, Lightness) values of the picture through GRIP to only show green colored shapes.

After this we processed the image to find the borders of the targets, again using GRIP. The borders were necessary for calculations involving distance and angle to the target. The end result can be seen below.



We also needed to account for situations where lights, other robots, or other general distractions create a less than optimal image to work with. Potential situations included more or less lines, or even diagonal lines. We solved this in two ways. First, the lines were sorted into line groups, as GRIP would generally find anywhere from 1 to 8 lines per edge depending on how straight the edges were. Second, a quality factor was added, in order to determine whether or not the image was reliable. This factor was based on several factors such as the number of line groups, as well as the ratio of the distances between the lines. A picture of the image with the line groups can be seen below.



The vertical line groups were considered to be the edge of the target. Using a pixel to inches conversion value, we were able to compare the the target's width to the theoretical target's widths, in order to determine the angle of the robot. The angle of the incidence of the target and the distance from the target were also calculated based on these ratios.

The suggested angle of rotation, a feature still in development at the time when it was decided not to use vision, would give an angle from 180° to -180° to tell the robot where to turn. These values would then be sent to the robot itself, and the movement would be handled in the robot code, rather than this vision code.

Although we ended up not using vision, it was a great learning experience.

Camera

Our competition robot has two cameras mounted on it. One is centered at the top of our robot on the front side. This camera is a USB camera, similar to one used to Skype on a computer. The other camera was mounted on the left side, just above the front bumpers. This camera is an IP camera, a type of camera most often used for closed-circuit surveillance. Originally the USB camera was to be used for vision because it could plug separately into the Kangaroo. The IP was to be used for driver visibility because there was no code needed for it to operate, simply an IP address.

In the end we used the USB camera for driver visibility because we ended up not using any vision code and it was more preferably placed for seeing the field. The code for adding the USB camera ended up being quite simple, but it was interesting to figure out.

Other Features

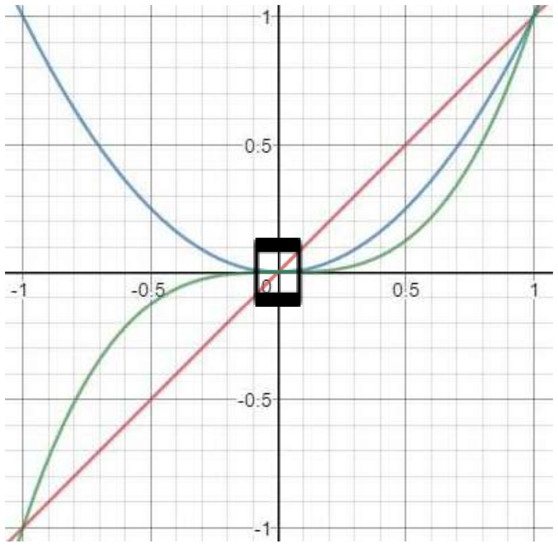
Key Features: Gyro, Deadband, Delinearization, CAN Lights, RADD Display and Timers

Deadband

Deadbands are a code feature that create a dead zone on the joystick. If the joystick is within the zone, the robot doesn’t move. Deadbands allow for white noise on a joystick. Joysticks often read small non-zero values even when they are not touched. Deadbands are important to keep robots from moving when they are not supposed to be. Our robots deadband is set to 0.1, so white noise is discounted and fine control is not lost.

Delinearization

Delinearization replaces the normal linear joystick curve with a power curve. Since some fine control is lost through deadbanding, delinearization regains some of that control. Delinearization also increases fine control at low speeds. Delinearization takes the joystick input and raises it to a power. Since every output of a joystick is less than one the delinearization output is always smaller. It’s important to make sure the delinearization power is odd however, or the robot will no longer be able to drive backwards. Below is a graph of what the joystick output and robot input would look like with delinearization. On the x-axis is the joystick output. Robot input lies on the y-axis. The red curve is without delinearization. The blue curve is delinearization with a power of two. The green curve is delinearization with a power of three. The black box shows the deadband area. Due to driver preferences, the final delinearization power was one.



CAN Lights

CAN lights are strips of special lights that take input from the robot in order to change their color and pattern. We use the CAN lights to display diagnostic information about our robot to our drive team. During the game the lights default to our current alliances' color. The lights also display green while climbing, orange while shooting and purple while delivering a gear.

There is also code that displays a rainbow of colors based on the gyro's position. This code was passed over in favor of the diagnostics mentioned above, but it was a notable learning experience. This process involved a conversion from HSL to RGB. This part of the code taught us all about HSL (Hue, Saturation, Lightness) and RGB's (Red, Green, Blue) different numeric display of colors. It was also a learning experience for a slightly more complicated chaining of if statements and comparisons.

RADD Display

Our RADD display is a text display on the back of our robot. We use it to display information about our robot and sponsors during a match. The display receives 4 digital inputs to choose which message to display. The inputs are set by converting a number from 0 to 15 into binary, and using each digit to set a single input to on or off. This feature was a learning experience in base ten to binary conversions and digital inputs.

The displays are selected based on various features of the robot. For instance during Autonomous the RADD display shows the message "Look Mom, No Hands!". Other specific messages correspond with functions of the robot such as climbing or shooting. When none of the specific functions are in use, the RADD display shows one of eight messages selected randomly. These eight messages contain our sponsors list, FIRST related messages and other silly messages such as "This Space for Rent".

Timers

Timers were absolutely necessary for our code to operate effectively. Instead of creating many loops that hamper the code's readability we use timers. A timer is started as part of the first actions done when a method is called. These timers are started before the methods action begins. Any timer is checked in the is finished method. Is finished is checked every time the action is executed and will return true or false. A true will end the action. Timers are a crucial part of the is finished methods. For example, they account for overcoming initial friction in drive to current function and allow our climbing systems override to operate both instantaneously and continuously.

Spike

A spike is a type of electrical relay. It has three states: off, forward, and back. We use a spike to turn our RADD display on or off. The coding for a spike is unique because it has three possible values: off, forward and reverse. It was also valuable to learn how to set up spikes in the code.

Servo

Servos are rotating devices that have a limited range. This means that a servo can rotate to a certain angle, but not more. Due to space limitations, there was an issue with feeding balls into our indexer. Mechanical could attach an agitator to one side, but not the other due to motor placement. The servo was meant to agitate the other side of the ball container. The servo was never used, as a more effective design was found by mechanical. However, the code for the servo was, again, a learning experience. Servos are unique in their declaration in the code and in how they are given input. The experience using servos was invaluable despite the lack of use.

Solenoids

Solenoids are another type of electrical relay. They were used to operate the shifters in the code. Solenoids are unique because they only exist in on and off. It was also a unique experience to learn how to set up solenoids in the code.

Ultrasonic

An ultrasonic sensor uses pulses of sound to find its distance away from an object. We considered using an ultrasonic sensor to find the proper distance for placing a gear on the airship. However, this was replaced by our drive-to-current feature that allows us to stop when we hit a wall. Despite the fact an ultrasonic sensor was never used, it was a learning experince code wise. It was unique to code because of the values it returns. The ultrasonic sensor we used returned values in volts, so we had to find the robots conversion from feet to volts and when to use it in the code.

Gyroscope

A gyroscope, or gyro, is a device that returns an angular direction from -180 to 180 inclusive on three different planes. These planes are officially called pitch, yaw and roll. Yaw returns the angle of the robot if it were placed inside a circle drawn on the ground. Pitch returns the angle of the robot if it were placed inside a circle that went around the robot from front to back. Roll returns the angle of the robot if it were placed inside a circle that went around the robot from left to right. We only utilized the yaw values returned from our gyro.

Like most parts, gyros are produced by many different companies. The gyro we use is a navX gyro. The primary reason The Charge uses a navX gyro is because of its accuracy. Gyros accumulate error over time. For instance at the end of a match the gyro might read zero degrees as two degrees. navX gyros accumulate much less error than other gyros we have tried over the years, under five degrees of error instead of close to twenty. This ensures accuracy at any point in the match.

Our gyro is primarily used in our turning functions, see page 2. We also considered using the gyro to drive the robot in a straight line during game's teleoperated period. However, with testing-based time constraints, we decided to average the joysticks inputs instead.

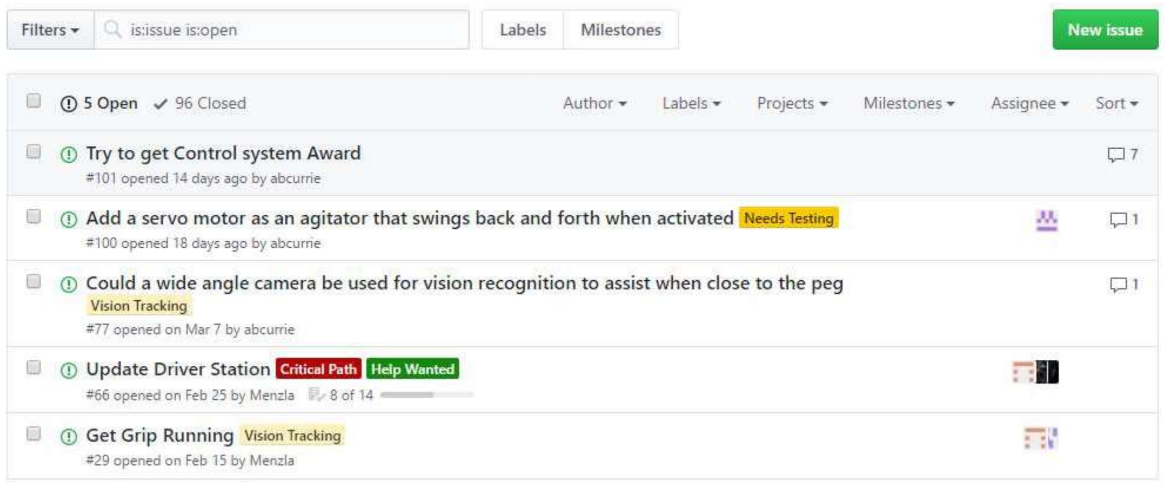
Code Structure

Key Features: SourceTree, GitHub, Eclipse

GitHub

Our programming team consists of ten people, all working on the same code. It gets difficult to communicate between all of us. One of our solutions to that problem is our use of GitHub. GitHub is an online website. We use it to store all of our code in a mutually accessible location. Every year the team creates a repository that will store our code for the season. A handy feature of these repositories is that they allow us to reuse parts of code from previous years by fulfilling FIRST requirement that any old code that we want to use again must be published before then next build season.

Each repository also has a feature called issues. Issues gather everything that needs to be done in the code to one location. They also let everyone see everything and pick what they would like to work on. Below is our current issues screen. We have completed 96 different issues this season. The icon on the right is who has chosen that issues to work on, this can be many or one person. The comments within each issue also allow for dialogue and documentation.



Try to get Control system Award — 7
#101 opened 14 days ago by abcurrie

Add a servo motor as an agitator that swings back and forth when activated Needs Testing — 1
#100 opened 18 days ago by abcurrie

Could a wide angle camera be used for vision recognition to assist when close to the peg Vision Tracking — 1
#77 opened on Mar 7 by abcurrie

Update Driver Station Critical Path Help Wanted
#66 opened on Feb 25 by Menzla 8 of 14

Get Grip Running Vision Tracking
#29 opened on Feb 15 by Menzla

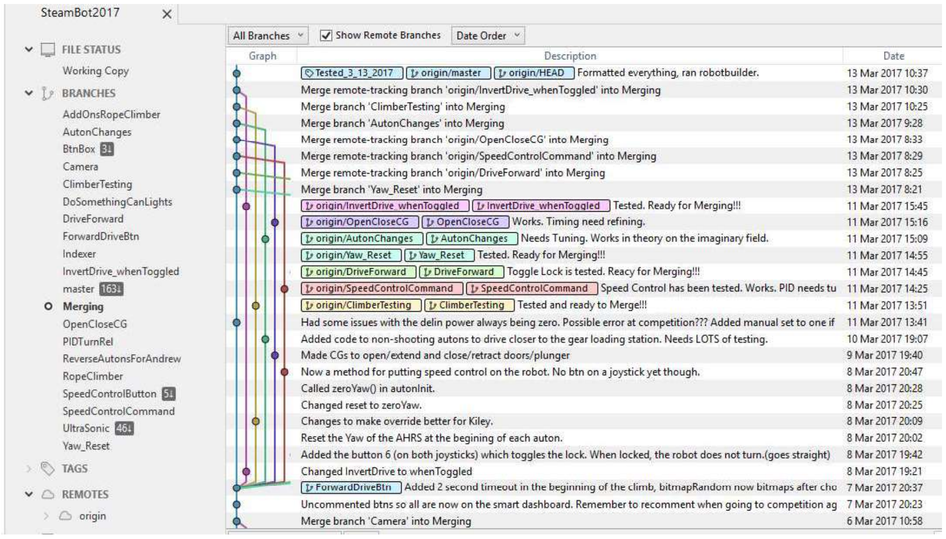
SourceTree

SourceTree is a third party software that is made to work with GitHub and other online code storage sites. SourceTree provides a few major features that we love, namely branching, merging and committing.

Branching allows each individual to branch off of the main code. This allows everyone to make as many changes to the code as they want without worrying about permanently messing up something that already works. Branching also allows extensive testing of changes, without unintended side effects.

Merging is the opposite of branching. During merging all the branches are brought together in one thread. After the process we have one branch with all the code changes.

Commits are another great feature. A commit is when any changes to the code are put back on the internet. It doesn't mean that the branch done or ready for testing, but it allows another person to pick up where someone else left off. On SourceTree it is also possible to go back to previous commits if something becomes terribly messed up. Below and on the right is a picture of our SourceTree window from early March. The lines represent branches and the dots commits. On the left is GitHub's tracking of all of our commits during the season.



SteamBot2017

FILE STATUS
- Working Copy

BRANCHES
- AddOnsRopeClimber
- AutonChanges
- BtnBox 3↓
- Camera
- ClimberTesting
- DoSomethingCanLights
- DriveForward
- ForwardDriveBtn
- Indexer
- InvertDrive_whenToggled
- master 163↓
- **Merging**
- OpenCloseCG
- PIDTurnRel
- ReverseAutonsForAndrew
- RopeClimber
- SpeedControlButton 5↓
- SpeedControlCommand
- UltraSonic 46↓
- Yaw_Reset

TAGS

REMOTES
- origin

All Branches | ✓ Show Remote Branches | Date Order

Description	Date
Tested_3_13_2017 origin/master origin/HEAD Formatted everything, ran robotbuilder.	13 Mar 2017 10:37
Merge remote-tracking branch 'origin/InvertDrive_whenToggled' into Merging	13 Mar 2017 10:30
Merge branch 'ClimberTesting' into Merging	13 Mar 2017 10:25
Merge branch 'AutonChanges' into Merging	13 Mar 2017 9:28
Merge remote-tracking branch 'origin/OpenCloseCG' into Merging	13 Mar 2017 8:33
Merge remote-tracking branch 'origin/SpeedControlCommand' into Merging	13 Mar 2017 8:29
Merge remote-tracking branch 'origin/DriveForward' into Merging	13 Mar 2017 8:25
Merge branch 'Yaw_Reset' into Merging	13 Mar 2017 8:21
origin/InvertDrive_whenToggled InvertDrive_whenToggled Tested. Ready for Merging!!!	11 Mar 2017 15:45
origin/OpenCloseCG OpenCloseCG Works. Timing need refining.	11 Mar 2017 15:16
origin/AutonChanges AutonChanges Needs Tuning. Works in theory on the imaginary field.	11 Mar 2017 15:09
origin/Yaw_Reset Yaw_Reset Tested. Ready for Merging!!!	11 Mar 2017 14:55
origin/DriveForward DriveForward Toggle Lock is tested. Reacy for Merging!!!	11 Mar 2017 14:45
origin/SpeedControlCommand SpeedControlCommand Speed Control has been tested. Works. PID needs tu	11 Mar 2017 14:25
origin/ClimberTesting ClimberTesting Tested and ready to Merge!!!	11 Mar 2017 13:51
Had some issues with the delin power always being zero. Possible error at competition??? Added manual set to one if	11 Mar 2017 13:41
Added code to non-shooting autons to drive closer to the gear loading station. Needs LOTS of testing.	10 Mar 2017 19:07
Made CGs to open/extend and close/retract doors/plunger	9 Mar 2017 19:40
Now a method for putting speed control on the robot. No btn on a joystick yet though.	8 Mar 2017 20:47
Called zeroYaw() in autonInit.	8 Mar 2017 20:28
Changed reset to zeroYaw.	8 Mar 2017 20:25
Changes to make override better for Kiley.	8 Mar 2017 20:09
Reset the Yaw of the AHRS at the begining of each auton.	8 Mar 2017 20:02
Added the button 6 (on both joysticks) which toggles the lock. When locked, the robot does not turn.(goes straight)	8 Mar 2017 19:42
Changed InvertDrive to whenToggled	8 Mar 2017 19:21
ForwardDriveBtn Added 2 second timeout in the beginning of the climb, bitmapRandom now bitmaps after cho	7 Mar 2017 20:37
Uncommented btns so all are now on the smart dashboard. Remember to recomment when going to competition ag	7 Mar 2017 20:23
Merge branch 'Camera' into Merging	6 Mar 2017 10:58

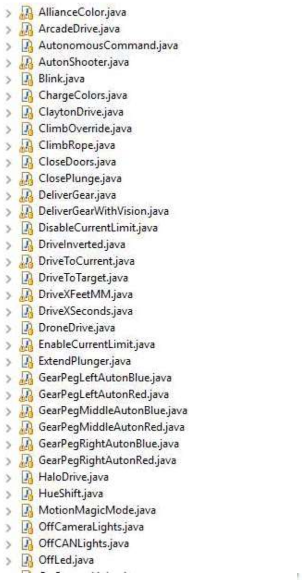
Jan 8, 2017 – Apr 9, 2017
Contributions to master, excluding merge commits
Contributions: Commits
Menzla
JessicaKline
wittcd
abcurrie
Kaedon2619
Yand1

Eclipse

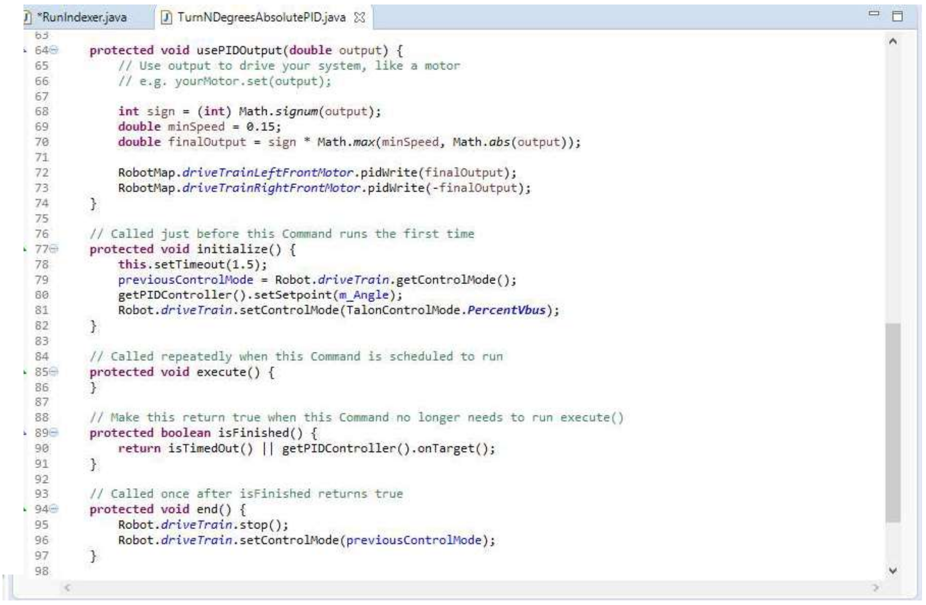
Eclipse is a free software that is often used as a coding platform. Eclipse can be linked to SourceTree easily, making it the perfect platform for us to use. One more unique thing about our code is that is command based. Commands are similar to classes. Command based code makes it easier to work with buttons and cuts down on loops. With a FIRST plug-in called Robot Builder, a command based system becomes simple. Robot Builder allows us to organize all elements of the code in one area. Each subsystem has its motors in its folder. Each joystick input has its buttons on it. Adding new commands or buttons is as simple as a right click and filling out a few boxes. Robot Builder then creates auto-generated code that makes it simple to code such functions.

Another plus of a command based system is organization. Each command does a few things and any heavy methods are contained in the subsystems.

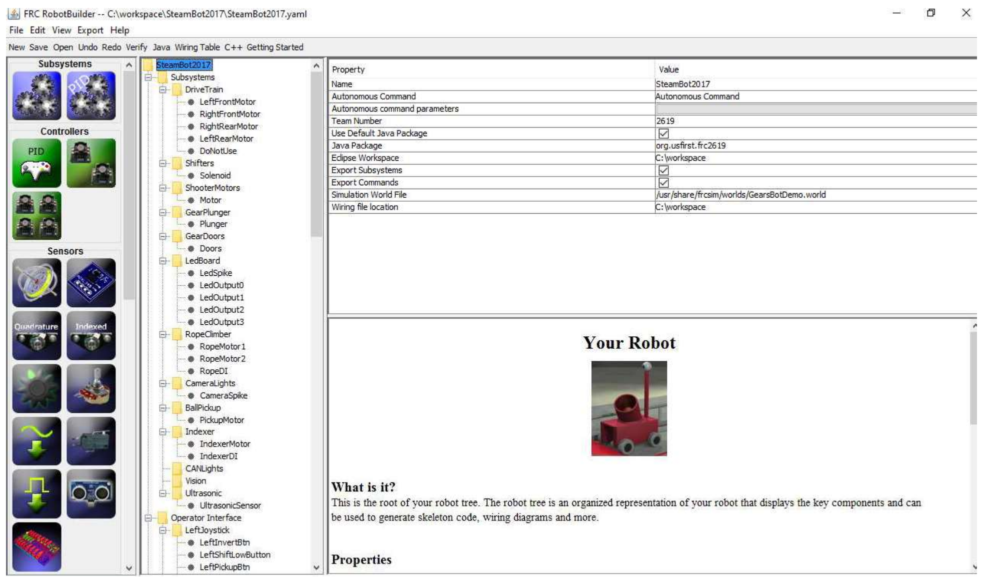
Command based systems also bring something called a command group. Command groups run multiple commands sequentially or in parallel. We use command groups for something like autonomous or opening our doors and plunging.



```
AllianceColor.java
ArcadeDrive.java
AutonomousCommand.java
AutonShooter.java
Blink.java
ChargeColors.java
ClaytonDrive.java
ClimbOverride.java
ClimbRope.java
CloseDoors.java
ClosePlunge.java
DeliverGear.java
DeliverGearWithVision.java
DisableCurrentLimit.java
DriveInverted.java
DriveToCurrent.java
DriveToTarget.java
DriveXFeetMM.java
DriveXSeconds.java
DroneDrive.java
EnableCurrentLimit.java
ExtendPlunger.java
GearPegLeftAutonBlue.java
GearPegLeftAutonRed.java
GearPegMiddleAutonBlue.java
GearPegMiddleAutonRed.java
GearPegRightAutonBlue.java
GearPegRightAutonRed.java
HaloDrive.java
HueShift.java
MotionMagicMode.java
OffCameraLights.java
OffCANLights.java
OffLed.java
```



```java
    protected void usePIDOutput(double output) {
        // Use output to drive your system, like a motor
        // e.g. yourMotor.set(output);

        int sign = (int) Math.signum(output);
        double minSpeed = 0.15;
        double finalOutput = sign * Math.max(minSpeed, Math.abs(output));

        RobotMap.driveTrainLeftFrontMotor.pidWrite(finalOutput);
        RobotMap.driveTrainRightFrontMotor.pidWrite(-finalOutput);
    }

    // Called just before this Command runs the first time
    protected void initialize() {
        this.setTimeout(1.5);
        previousControlMode = Robot.driveTrain.getControlMode();
        getPIDController().setSetpoint(m_Angle);
        Robot.driveTrain.setControlMode(TalonControlMode.PercentVbus);
    }

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

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return isTimedOut() || getPIDController().onTarget();
    }

    // Called once after isFinished returns true
    protected void end() {
        Robot.driveTrain.stop();
        Robot.driveTrain.setControlMode(previousControlMode);
    }
```


FRC RobotBuilder -- C:\workspace\SteamBot2017\SteamBot2017.yaml
File Edit View Export Help
New Save Open Undo Redo Verify Java Wiring Table C++ Getting Started
Subsystems
PID
Controllers
PID
Sensors
Quadrature
Indexed
Your Robot

```
SteamBot2017
  Subsystems
    DriveTrain
      LeftFrontMotor
      RightFrontMotor
      RightRearMotor
      LeftRearMotor
      DoNotUse
    Shifters
      Solenoid
    ShooterMotors
      Motor
    GearPlunger
      Plunger
    GearDoors
      Doors
    LedBoard
      LedSpike
      LedOutput0
      LedOutput1
      LedOutput2
      LedOutput3
    RopeClimber
      RopeMotor1
      RopeMotor2
      RopeDI
    CameraLights
      CameraSpike
    BallPickup
      PickupMotor
    Indexer
      IndexerMotor
      IndexerDI
    CANLights
    Vision
    Ultrasonic
      UltrasonicSensor
  Operator Interface
    LeftJoystick
      LeftInvertBtn
      LeftShiftLowButton
      LeftPickupBtn
```

Property	Value
Name	SteamBot2017
Autonomous Command	Autonomous Command
Autonomous command parameters	
Team Number	2619
Use Default Java Package	✓
Java Package	org.usfirst.frc2619
Eclipse Workspace	C:\workspace
Export Subsystems	✓
Export Commands	✓
Simulation World File	/usr/share/frcsim/worlds/GearsBotDemo.world
Wiring file location	C:\workspace

What is it?
This is the root of your robot tree. The robot tree is an organized representation of your robot that displays the key components and can be used to generate skeleton code, wiring diagrams and more.

Properties

Top Left: Portion of our list of commands in Eclipse.

Top Right: Example of code in Eclipse. This code is specifically for our absolute turning command. The UsePID method is unique to this type of command and does not appear in most.

Bottom: Robot Builder. On the side are the different subsystems and the start of the joysticks and their buttons.

Testing

Key Features: Ply-Bot, Practice Bot and Competition Bot

Ply-Bot

As much as we like to pretend our code is perfect every time, it rarely is. Thus we conduct extensive testing. Luckily we don't have to wait until mechanical builds a robot to do that. Our first stage of testing takes place on our ply-bot. Ply-bot is just a drive train and some plywood holding electronics, but it serves perfectly for early stage testing. We use ply-bot to test functions like set distance driving, set angle turning and different joystick configurations for driving. Ply-bot serves as the platform for teaching new programmers about the robot and coding the robot. Ply-bot is pictured on the far left at the bottom of the page.

Practice Bot

Ply-bot is only good until a certain stage. As the code becomes more specialized towards the season's game, we move towards the practice bot. Practice bot is nearly identical to our competition bot. Practice bot started as a way for mechanical to prototype without using out of bag time, but became very helpful to programming. Practice bot is the way we can test, debug and fix code without taking valuable out of bag time. This is the robot that subsystem code and tuning was done on.

Competition Bot

This is our actual competing robot that goes in the bag. The only things tested on this robot were distances and angles specific to the field. It was still necessary to the overall process, as it executes the code on the field during games. Competition bot is pictured on the far right at the bottom of the page.