Team O-Ryon: Draconis Code Overview

We chose to use WPILib Java with the command-subsystem framework on our robot. This design requires impressive amounts of boilerplate code and many classes up front, but offers easy extensibility as mechanisms change and complexity increases as the build season goes on. The [WPI command-based programming manual](http://wpilib.screenstepslive.com/s/3120/m/7952) and the rest of the pages on the [WPI 2014 FRC Control System](http://wpilib.screenstepslive.com/s/3120) site are great resources for teams training new members or trying to learn the command-based structure. Here we present a description of the files in our project for reference when teams are designing their own code, and an arcade drive visualizer that teams can modify and use to test the effects of different drivetrain controls.

Classes:

* HW: This class simply lists all the components plugged into cRIO modules or the digital sidecar. We keep this list in one place and then reference it in the subsystems so any electrical changes are quick fixes.
* Main: The main class can be very simple on a command-based robot. All that teams need to change from the template is their autonomous command and sending any data to the SmartDashboard.
* OI: OI stands for Operator Interface. This class has objects representing the joysticks and the buttons on the joysticks. It then ties these buttons to commands, so the function of each button can be changed on short notice.
* commands.CommandBase: The CommandBase class lists the robot’s subsystems so the commands (which extend this class) can easily use them.
* commands.DriveForwards: This is our autonomous command. Given a time, the robot simply drives for that amount of time. 5 points!
* commands.arm.ArmDown,

commands.arm.ArmUp,

commands.arm.ArmStage2Up,

commands.arm.ArmStage2Down: These commands are mostly boilerplate, and just call methods to control pneumatics. Having commands for them lets us tie them to buttons and execute them easily.

* commands.compress.ToggleCompressor: This command turns the compressor on and off depending on its current state. Teams may want to send the state of the compressor to the Driver Station over the SmartDashboard, as it is difficult to determine whether it is running from across the field.
* commands.drive.LerpDrive: The linearly interpolated arcade drive scheme implemented here allows a linear turning response at right speeds with no clipping. Please see [this thread](http://www.chiefdelphi.com/forums/showthread.php?p=1202236) on Chief Delphi, Ether’s whitepaper linked there, and the included arcade drive visualizer.
* commands.arm.variIntake.

commands.flip.variFlip: These commands always run and set power for the intake and flipper based on button inputs. These commands are an artifact of time pressure and ease of implementation. You should instead make individual commands for each state of your systems, such as a command for intake in and a command for intake out. These commands should then be tied to whileHeld() button inputs.

* subsystems.CompressorSub,

subsystems.Drivetrain,

subsystems.Flipper,

subsystems.Intake: In the command-based structure, subsystems should provide objects representing the motors, sensors, and other hardware on your robot and the most basic methods to access and set those objects. Manipulation of these objects via these methods should take place in commands.

* utils.BetterDrive: We created this extension of RobotDrive to gain control over smoothing sudden changes in joystick inputs.
* utils.Dashboard: This class puts all the data we want to send to the SmartDashboard in one place.
* utils.GamePad: This class allows us to use the Logitech gamepad in either mode (determined by a switch on the back) to control the robot. Mode D allows use of the Y digital pad.