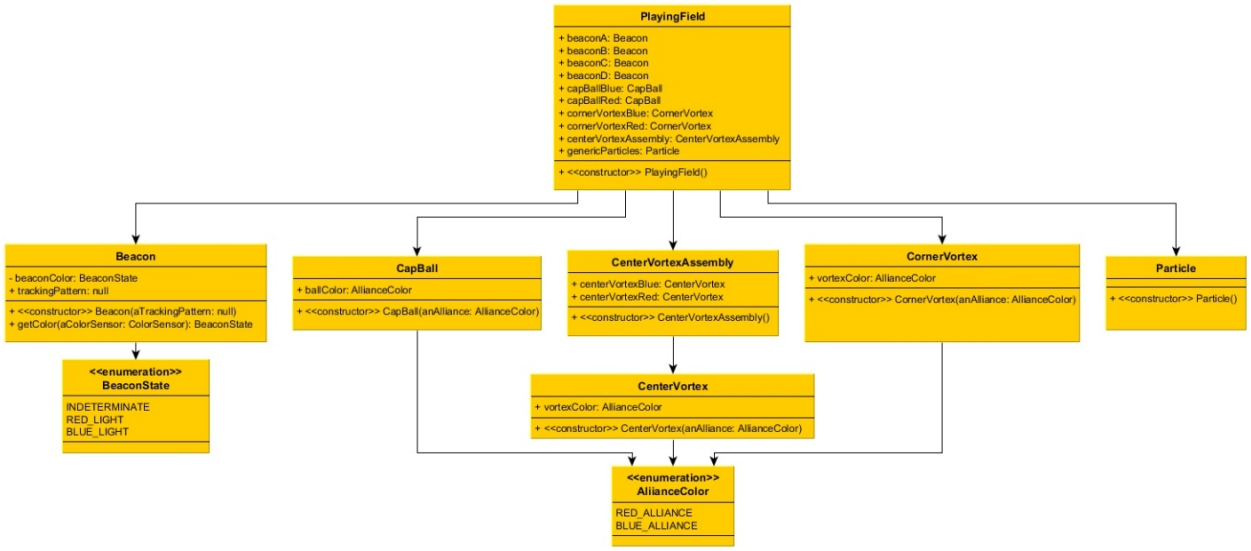
Section IV Engineering (Software)

Playing Field Abstraction

Data Model:



Code Description and Considerations:

- The abstraction of the field will allow us to reuse common references for field components and interface with field components in different opmodes and classes in a consistent manner.
- Specifically, the enumerated classes will allow for us to reference human-readable, specific constants rather than representing colors with integers, booleans, or other types that have no explicit meaning in regards to field elements.

Playing Field Abstraction

Class File for Playing Field:

```
public class PlayingField {

    public Beacon beaconA;
    public Beacon beaconB;
    public Beacon beaconC;
    public Beacon beaconD;
    public CapBall capBallBlue;
    public CapBall capBallRed;
    public CornerVortex cornerVortexBlue;
    public CornerVortex cornerVortexRed;
    public CenterVortexAssembly centerVortexAssembly;
    public Particle genericParticles;

    public PlayingField() {
        beaconA = new Beacon();
        beaconB = new Beacon();
        beaconC = new Beacon();
        beaconD = new Beacon();
        capBallBlue = new CapBall(BLUE_ALLIANCE);
        capBallRed = new CapBall(RED_ALLIANCE);
        cornerVortexBlue = new CornerVortex(BLUE_ALLIANCE);
        cornerVortexRed = new CornerVortex(RED_ALLIANCE);
        centerVortexAssembly = new CenterVortexAssembly();
        genericParticles = new Particle();
    }

}
```

Class File for Beacon:

```
public class Beacon {
    private BeaconState beaconColor;

    public Beacon() {
        beaconColor = BeaconState.INDETERMINATE;
    }

    public BeaconState getColor(ColorSensor aColorSensor) {
        beaconColor = BeaconState.INDETERMINATE;
        return beaconColor;
    }

}
```

Enum Type for Beacon State:

```
public enum BeaconState {
    INDETERMINATE,
    RED_ALLIANCE,
    BLUE_ALLIANCE
}
```

Class File for Particle:

```
public class Particle {

    public Particle() {

    }

}
```

Playing Field Abstraction

Class File for Cap Ball:

```
public class CapBall {

    public AllianceColor ballColor;

    public CapBall(AllianceColor anAlliance){
        ballColor = anAlliance;
    }

}
```

Class File for Center Vortex Assembly:

```
public class CenterVortexAssembly {

    public CenterVortex centerVortexBlue;
    public CenterVortex centerVortexRed;

    public CenterVortexAssembly() {
        centerVortexBlue = new CenterVortex(BLUE_ALLIANCE);
        centerVortexRed = new CenterVortex(RED_ALLIANCE);
    }

}
```

Class File for Center Vortex Goal:

```
public class CenterVortex {

    public AllianceColor vortexColor;

    public CenterVortex(AllianceColor anAlliance) {
        vortexColor = anAlliance;
    }

}
```

Class File for Corner Vortex Goal:

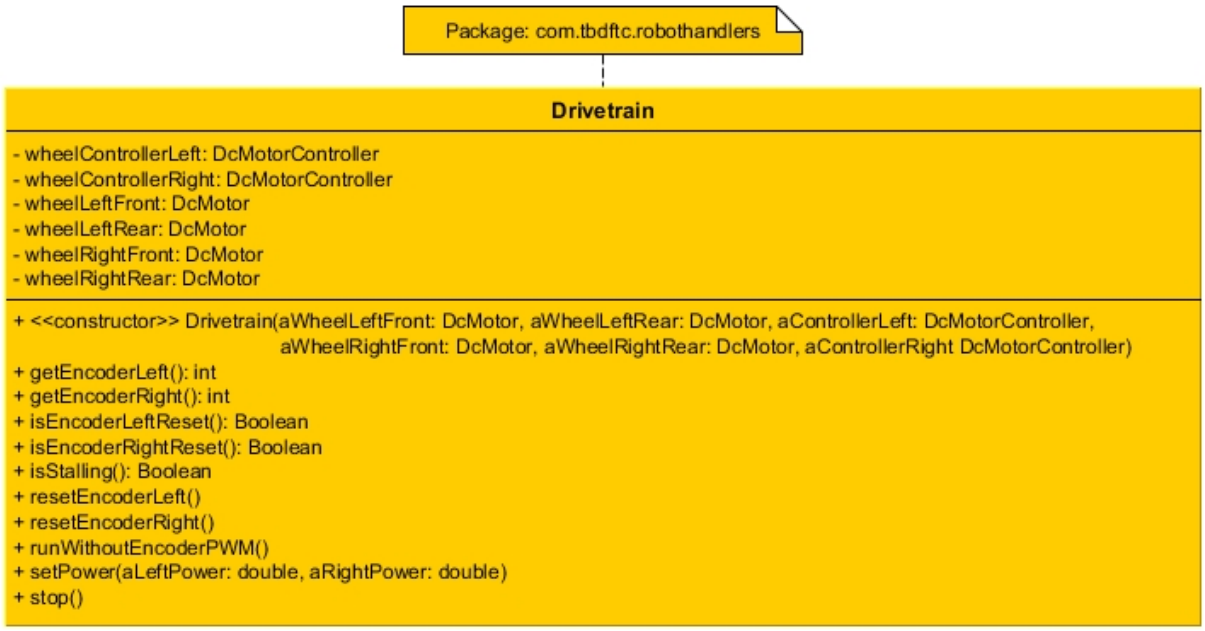
```
public class CornerVortex {

    public AllianceColor vortexColor;

    public CornerVortex(AllianceColor anAlliance) {
        vortexColor = anAlliance;
    }

}
```

Enum Type for Alliance Color:

```
public enum AllianceColor {
    RED_ALLIANCE,
    BLUE_ALLIANCE;
}
```

Basic 4WD Drivetrain Class

Data Model:



Code Description and Considerations:

- To allow us to test our basic drivetrain system, we created this class; however, we only implemented the constructor, the set power method, and the stop method.
- The above methods allow for basic control in both teleoperated and autonomous paradigms.
- All other methods, as outlined in the above diagram, still need to be implemented. These more advanced-use methods include getting encoder values, resetting encoders, detecting if encoders are reset, detecting if the motors are stalling, and driving without pulse width modulation control.
- This class is designed to be implemented in an iterative/looping opmode environment to achieve desired functionality, especially with future encoder sporadic value correction and stall detection implementation.

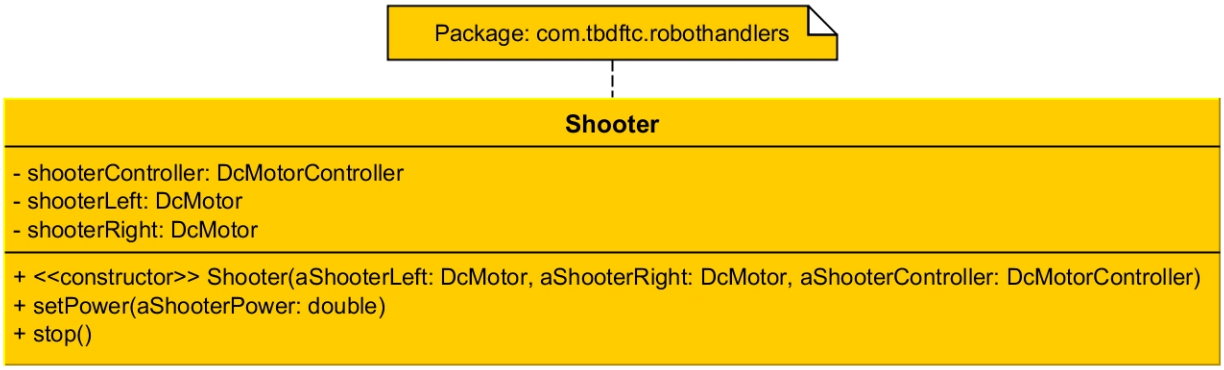
Basic 4WD Drivetrain Class

Class File:

```
public class Drivetrain {

    private DcMotorController wheelControllerLeft;
    private DcMotorController wheelControllerRight;
    private DcMotor wheelLeftFront;
    private DcMotor wheelLeftRear;
    private DcMotor wheelRightFront;
    private DcMotor wheelRightRear;

    public Drivetrain(DcMotor aWheelLeftFront, DcMotor aWheelLeftRear,
                      DcMotorController aControllerLeft, DcMotor aWheelRightFront,
                      DcMotor aWheelRightRear, DcMotorController aControllerRight) {
        wheelControllerLeft = aControllerLeft;
        wheelControllerRight = aControllerRight;
        wheelLeftFront = aWheelLeftFront;
        wheelLeftRear = aWheelLeftRear;
        wheelRightFront = aWheelRightFront;
        wheelRightRear = aWheelRightRear;

        wheelRightFront.setDirection(REVERSE);
        wheelRightRear.setDirection(REVERSE);
    }

    //TODO: Implement all other methods.

    public void setPower(double aLeftPower, double aRightPower) {
        wheelLeftFront.setPower(aLeftPower);
        wheelLeftRear.setPower(aLeftPower);
        wheelRightFront.setPower(aRightPower);
        wheelRightRear.setPower(aRightPower);
    }

    public void stop() {
        wheelLeftFront.setPower(0.0);
        wheelLeftRear.setPower(0.0);
        wheelRightFront.setPower(0.0);
        wheelRightRear.setPower(0.0);
    }

}
```

Basic Particle Shooter Class

Data Model:



Code Description and Considerations:

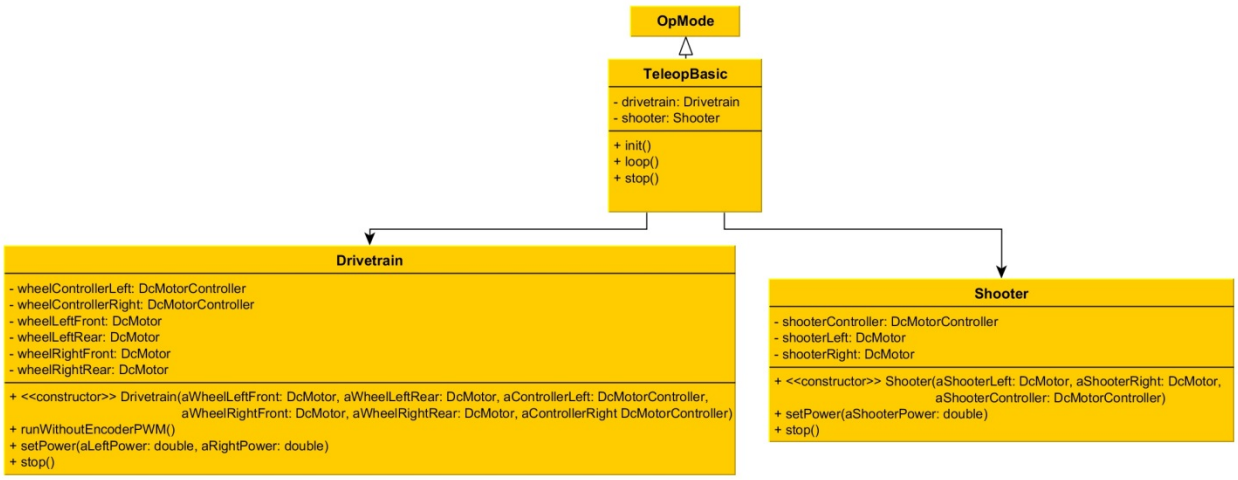
- To allow us to test our basic particle launcher, we created this class with basic motor movement.
- The above methods allow for basic control in both teleoperated and autonomous paradigms.
- Going forward, this class will contain a servo attribute so that we can flick balls into our particle shooter to ensure controllability in launching balls.
- Additionally, we plan to later implement encoders and PID control to ensure that the motors are synchronized, which will help us launch particles straight out while minimizing the risk of launching the valuable particles crooked and off-center relative to the vortices.

Class File:

```
public class Shooter {

    private DcMotorController shooterController;
    private DcMotor shooterLeft;
    private DcMotor shooterRight;

    public Shooter(DcMotor aShooterLeft, DcMotor aShooterRight, DcMotorController aShooterController) {
        shooterController = aShooterController;
        shooterLeft = aShooterLeft;
        shooterRight = aShooterRight;

        shooterLeft.setDirection(REVERSE);
    }

    public void setPower(double aShooterPower) {
        shooterLeft.setPower(aShooterPower);
        shooterRight.setPower(aShooterPower);
    }

    public void stop() {
        shooterLeft.setPower(0.0);
        shooterRight.setPower(0.0);
    }

}
```

Basic Teleop with Driving and Shooting

Data Model:



Code Description and Considerations:

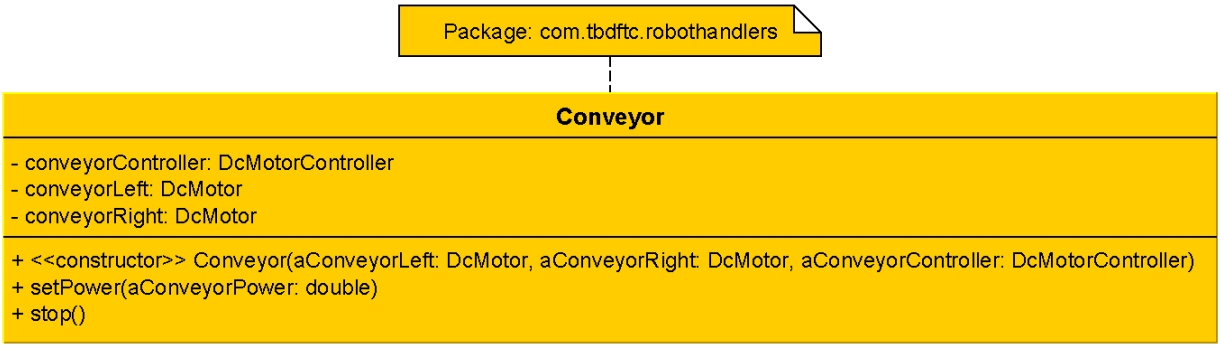
- To allow us to test our current robot, we developed this opmode that implements the drivetrain and shooter classes described on the previous two pages.

Class File:

```
@TeleOp(name="Tele: Basic", group="Tele")
public class TeleopBasic extends OpMode {

    private Drivetrain drivetrain;
    private Shooter shooter;

    @Override
    public void init() {
        drivetrain = new Drivetrain(hardwareMap.dcMotor.get("wheelLeftFront"),
hardwareMap.dcMotor.get("wheelLeftRear"), hardwareMap.dcMotorController.get("wheelsLeft"),
hardwareMap.dcMotor.get("wheelRightFront"), hardwareMap.dcMotor.get("wheelRightRear"),
hardwareMap.dcMotorController.get("wheelsRight"));

        shooter = new Shooter(hardwareMap.dcMotor.get("shooterLeft"),
hardwareMap.dcMotor.get("shooterRight"), hardwareMap.dcMotorController.get("particleShooter"));
    }

    @Override
    public void loop() {
        drivetrain.setPower(gamepad1.left_stick_y, gamepad1.right_stick_y);
        shooter.setPower(gamepad2.right_stick_y);
        telemetry.addData("1 Runtime", getRuntime());
    }

    @Override
    public void stop() {
        drivetrain.stop();
        shooter.stop();
    }

}
```

Basic Particle Conveyor Class

Data Model:



Code Description and Considerations:

- To allow us to test our basic particle conveyor, we created this class.
- This control was assigned to the second gamepad's left stick y-axis.
- Given our testing and discussion with the hardware team, it is unlikely that we will use this conveyor design in our final robot, so it is unlikely that this class will be implemented.

Class File:

```
public class Conveyor {

    private DcMotorController conveyorController;
    private DcMotor conveyorLeft;
    private DcMotor conveyorRight;

    public Conveyor(DcMotor aConveyorLeft, DcMotor aConveyorRight, DcMotorController
aConveyorController) {
        conveyorController = aConveyorController;
        conveyorLeft = aConveyorLeft;
        conveyorRight = aConveyorRight;

        conveyorRight.setDirection(REVERSE);
    }

    public void setPower(double aConveyorPower) {
        conveyorLeft.setPower(aConveyorPower);
        conveyorRight.setPower(aConveyorPower);
    }

    public void stop() {
        conveyorLeft.setPower(0.0);
        conveyorRight.setPower(0.0);
    }

}
```

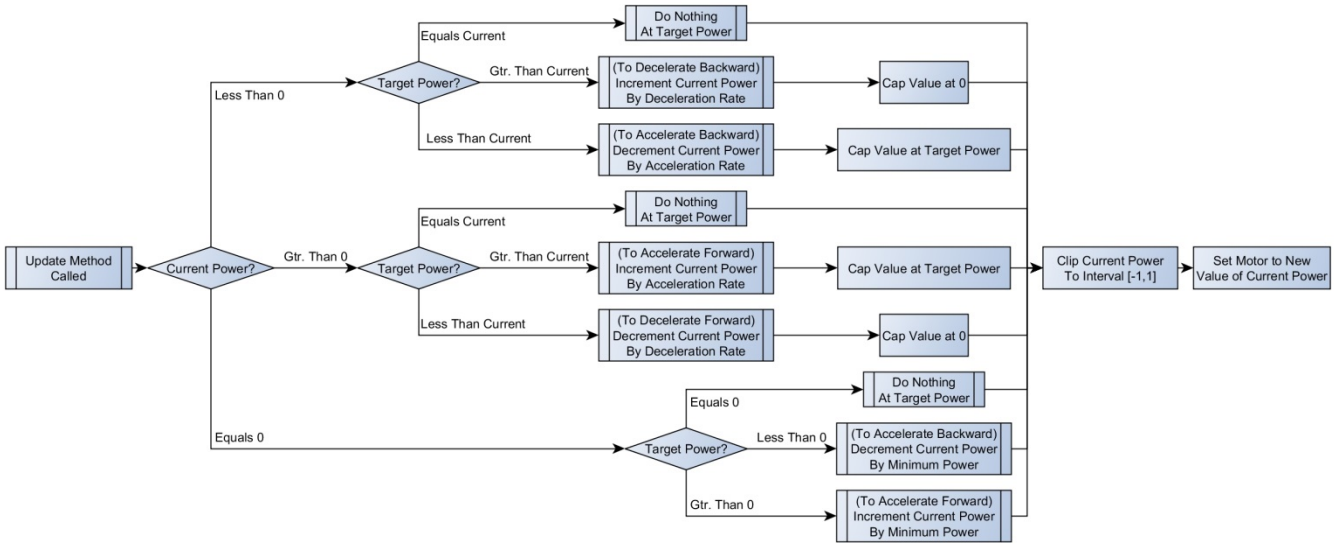
DC Motor Acceleration Control

General Overview:

- Given the sensitivity of our motor gearboxes and our desire to reduce strain on the motor, there was a clear need for acceleration control.
- This control, as explained below, is broken down into three segments: the accelerated motor class, the thread for updating accelerated motors, and the implementation with the drivetrain.

Accelerated Motor Data Model:

Accelerated Motor Flowchart:



DC Motor Acceleration Control

Accelerated Motor Code Description and Considerations:

- This class essentially wraps the SDK-provided DcMotor class. All of the functions are the same except instead of setting a motor power directly, an acceleration rate and a target speed are set and then the update method is called to increment or decrement the motor towards the target power.
- Because of the need for the update method to be called repeatedly, it is called from a loop in a separate thread while the main thread updates the target power through the opmode paradigm.
- The programmer inputs a motor's acceleration rate in motor power units (from -1 to 1) per second. Then, the software computes the rate of change using the inputted rate and the update period defined in the looping motor acceleration control thread.
- There is also a minimum power parameter that ensures the motor starts at a speed that allows the robot to move and limits stalling strain on the motors.
- The stopMotorHard method is used to ensure that the motors come to a complete stop upon the end of a match and prevent us from incurring penalties.
- Two separate rates are used for adjusting power, an acceleration rate and a deceleration rate. The conditions during which each is used is outlined in the chart below:

		Which is Greater? (Target or Current Power)	
		Target Power Greater	**Current Power Greater**
Sign of Velocity	**Positive**	Accelerate Forwards	Decelerate Forwards
	Negative	Decelerate Backwards	Accelerate Backwards

Accelerated Motor Class File:

```
public class DcMotorAccelerated {

    private static final double MAX_MOTOR_POWER = 1;
    private static final double MIN_MOTOR_POWER = -1;

    private DcMotor acceleratedMotor;
    private double accelerationRateSpeedPerHardwareTick;
    private double currentPower;
    private double decelerationRateSpeedPerHardwareTick;
    private double minimumPower;
    private double targetPower;

    public DcMotorAccelerated(DcMotor aMotor, double anAccelerationRateMotorSpeedPerSecond, double
aDecelerationRateMotorSpeedPerSecond, double aMinimumPower) {
        acceleratedMotor = aMotor;
        setAccelerationRates(anAccelerationRateMotorSpeedPerSecond,
aDecelerationRateMotorSpeedPerSecond);
        setMinPower(aMinimumPower);
    }

    public double getCurrentPower() { return acceleratedMotor.getPower(); }

    public synchronized double getTargetPower() { return targetPower; }
```

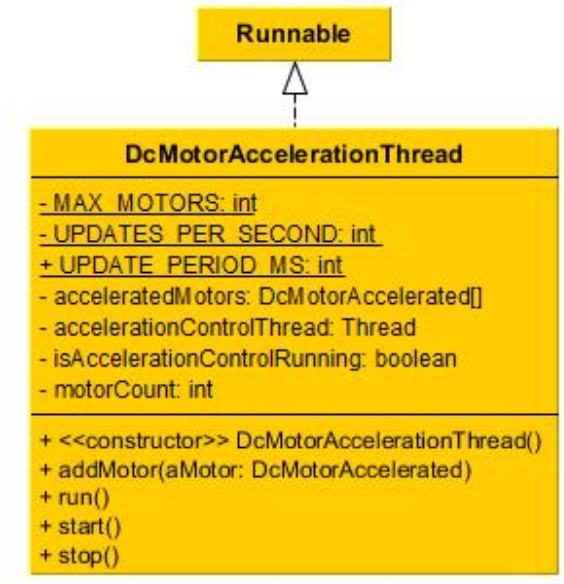
DC Motor Acceleration Control

Accelerated Motor Class File (Continued):

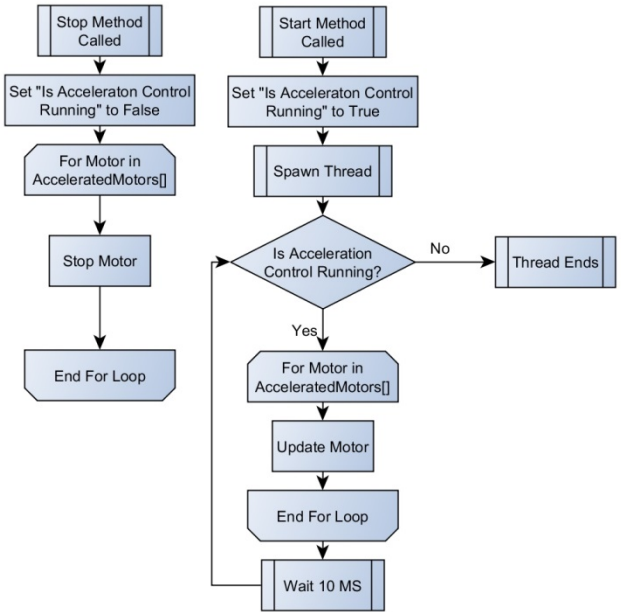
```
    public synchronized void setAccelerationRates(double anAccelerationRateMotorSpeedPerSecond, double
aDecelerationRateMotorSpeedPerSecond) {
        accelerationRateSpeedPerHardwareTick = anAccelerationRateMotorSpeedPerSecond / (1000 /
DcMotorAccelerationThread.UPDATE_PERIOD_MS);
        decelerationRateSpeedPerHardwareTick = aDecelerationRateMotorSpeedPerSecond / (1000 /
DcMotorAccelerationThread.UPDATE_PERIOD_MS);
    }

    public synchronized void setMinPower(double aMinPower) { minimumPower = aMinPower; }

    public synchronized void setTargetPower(double aTargetPower) { targetPower = aTargetPower; }

    public void stopMotorHard() { acceleratedMotor.setPower(0.0); }

    public synchronized void update() {
        if(currentPower > 0) {
            //Accelerating.
            if(currentPower < targetPower) {
                currentPower += accelerationRateSpeedPerHardwareTick;
                currentPower = Math.min(currentPower, targetPower);
            }
            //Decelerating.
            else if(currentPower > targetPower) {
                currentPower -= decelerationRateSpeedPerHardwareTick;
                currentPower = Math.max(currentPower, 0);
            }
            else {
                //Do nothing.
            }
        }
        else if(currentPower < 0){
            //Decelerating.
            if(currentPower < targetPower) {
                currentPower += decelerationRateSpeedPerHardwareTick;
                currentPower = Math.min(currentPower, 0);
            }
            //Accelerating.
            else if(currentPower > targetPower) {
                currentPower -= accelerationRateSpeedPerHardwareTick;
                currentPower = Math.max(currentPower, targetPower);
            }
            else {
                //Do nothing.
            }
        }
        else {
            //Start moving forward.
            if(targetPower > 0.0) {
                currentPower += Math.max(minimumPower, accelerationRateSpeedPerHardwareTick);
            }
            //Start moving backward.
            else if(targetPower < 0.0) {
                currentPower -= Math.max(minimumPower, accelerationRateSpeedPerHardwareTick);
            }
            else {
                //Don't move.
            }
        }

        currentPower = Math.min(currentPower, MAX_MOTOR_POWER);
        currentPower = Math.max(currentPower, MIN_MOTOR_POWER);
        acceleratedMotor.setPower(currentPower);
    }

}
```

DC Motor Acceleration Control

Accel. Control Thread Data Model:



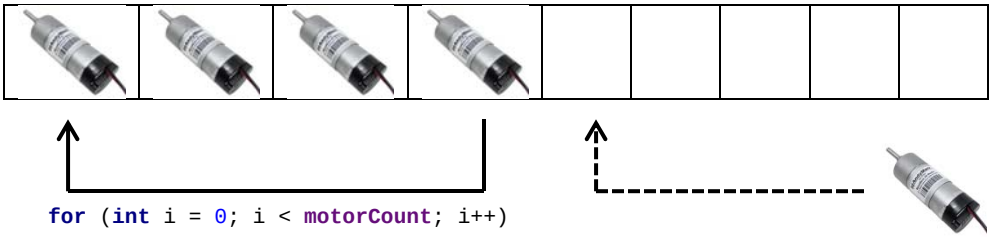
Accel. Control Thread Flowchart:



Acceleration Control Thread Code Description and Considerations:

- Upon instantiating this class, it is used by using the addMotor method, which adds an accelerated motor object, as described on the previous pages, to an array.
- Once the start method is called, if no other motor acceleration control thread exists, a new thread is created that calls the update method, as described on the previous pages, on the accelerated motor objects in the aforementioned array through a for loop.
- This looping behavior continues until the stop method is called, which kills the thread and sets each motor in the array to zero power.
- To ensure that the above for loops aren't calling empty array items, there is a motor count variable that increments each time a motor. This is then used in the for loop to keep it in bounds of the valid array elements.

acceleratedMotors[]



```
for (int i = 0; i < motorCount; i++)
```

```
acceleratedMotors[motorCount++]
 = aMotor;
```

DC Motor Acceleration Control

Acceleration Control Thread Class File:

```
public class DcMotorAccelerationThread implements Runnable {

    private static final int MAX_MOTORS = 8;
    private static final int UPDATES_PER_SECOND = 100;
    public static final int UPDATE_PERIOD_MS = 1000 / UPDATES_PER_SECOND;

    private DcMotorAccelerated[] acceleratedMotors = new DcMotorAccelerated[MAX_MOTORS];
    private boolean isAccelerationControlRunning = false;
    private int motorCount = 0;
    private Thread accelerationControlThread;

    public DcMotorAccelerationThread() {

    }

    public void addMotor(DcMotorAccelerated aMotor) {
        acceleratedMotors[motorCount++] = aMotor;
    }

    @Override
    public synchronized void run() {
        while (isAccelerationControlRunning) {
            for (int i = 0; i < motorCount; i++) {
                acceleratedMotors[i].update();
            }

            try {
                Thread.sleep(UPDATE_PERIOD_MS);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public synchronized void start() {
        if(!isAccelerationControlRunning) {
            isAccelerationControlRunning = true;
            accelerationControlThread = new Thread(this);
            accelerationControlThread.start();
        }
    }

    public void stop() {
        isAccelerationControlRunning = false;

        for (int i = 0; i < motorCount; i++) {
            acceleratedMotors[i].stopMotorHard();
        }
    }

}
```

DC Motor Acceleration Control

Drivetrain Implementation Description and Considerations:

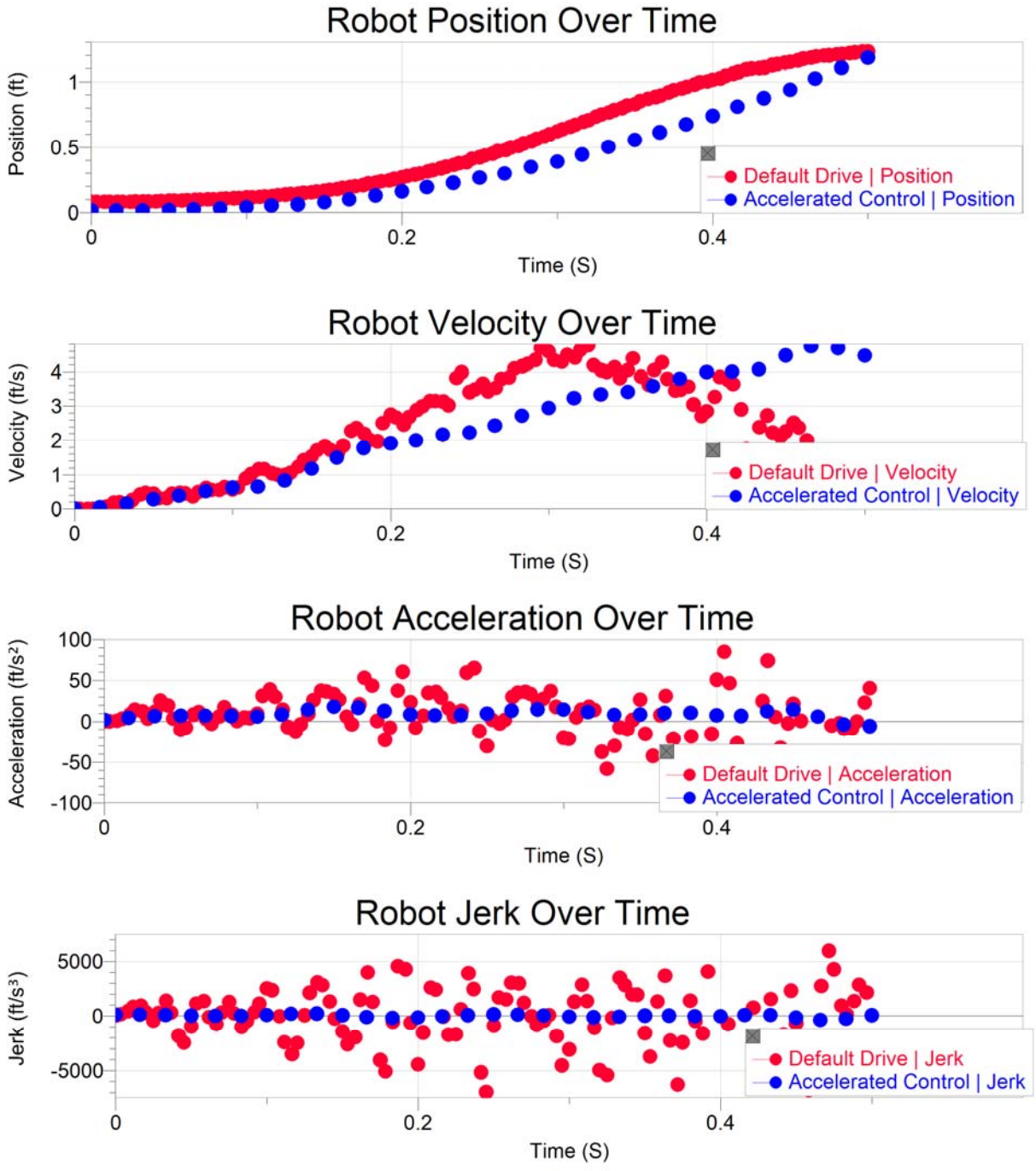
- Rather than using the traditional setPower method, the motors in the drivetrain are assigned target powers through the setTargetPower method. These target powers inform that power the motor should accelerate to.
- Constants are used in the drivetrain class to set the acceleration rate, deceleration rate, and minimum power across all drive motors.

Drivetrain Implementation Class File:

```
public class Drivetrain {

    private static final double WHEEL_ACCEL_SPEED_PER_SECOND = 1.5;
    private static final double WHEEL_DECEL_SPEED_PER_SECOND = 2.5;
    private static final double WHEEL_MINIMUM_POWER = 0.3;

    private DcMotorAccelerationThread wheelAccelerationThread;
    private DcMotorController wheelControllerLeft;
    private DcMotorController wheelControllerRight;
    private DcMotorAccelerated wheelLeftFront;
    private DcMotorAccelerated wheelLeftRear;
    private DcMotorAccelerated wheelRightFront;
    private DcMotorAccelerated wheelRightRear;

    public Drivetrain(DcMotor aWheelLeftFront, DcMotor aWheelLeftRear, DcMotorController
aControllerLeft,
                      DcMotor aWheelRightFront, DcMotor aWheelRightRear, DcMotorController
aControllerRight) {
        wheelControllerLeft = aControllerLeft;
        wheelControllerRight = aControllerRight;
        wheelLeftFront = new DcMotorAccelerated(aWheelLeftFront, WHEEL_ACCEL_SPEED_PER_SECOND,
WHEEL_DECEL_SPEED_PER_SECOND, WHEEL_MINIMUM_POWER);
        wheelLeftRear = new DcMotorAccelerated(aWheelLeftRear, WHEEL_ACCEL_SPEED_PER_SECOND,
WHEEL_DECEL_SPEED_PER_SECOND, WHEEL_MINIMUM_POWER);
        wheelRightFront = new DcMotorAccelerated(aWheelRightFront, WHEEL_ACCEL_SPEED_PER_SECOND,
WHEEL_DECEL_SPEED_PER_SECOND, WHEEL_MINIMUM_POWER);
        wheelRightRear = new DcMotorAccelerated(aWheelRightRear, WHEEL_ACCEL_SPEED_PER_SECOND,
WHEEL_DECEL_SPEED_PER_SECOND, WHEEL_MINIMUM_POWER);

        wheelRightFront.setDirection(REVERSE);
        wheelRightRear.setDirection(REVERSE);
        runWithoutEncoderPWM();

        wheelAccelerationThread = new DcMotorAccelerationThread();
        wheelAccelerationThread.addMotor(wheelLeftFront);
        wheelAccelerationThread.addMotor(wheelLeftRear);
        wheelAccelerationThread.addMotor(wheelRightFront);
        wheelAccelerationThread.addMotor(wheelRightRear);
        wheelAccelerationThread.start();
    }

    public void setPower(double aLeftPower, double aRightPower) {
        wheelLeftFront.setTargetPower(aLeftPower);
        wheelLeftRear.setTargetPower(aLeftPower);
        wheelRightFront.setTargetPower(aRightPower);
        wheelRightRear.setTargetPower(aRightPower);
    }

    public void stop() {
        wheelAccelerationThread.stop();
    }

}
```

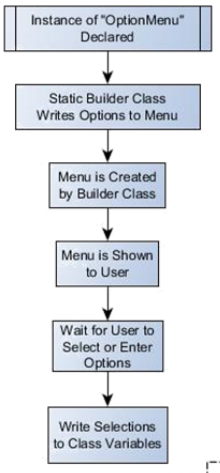
DC Motor Acceleration Control

Graphs:



Autonomous Options/Parameters Menu

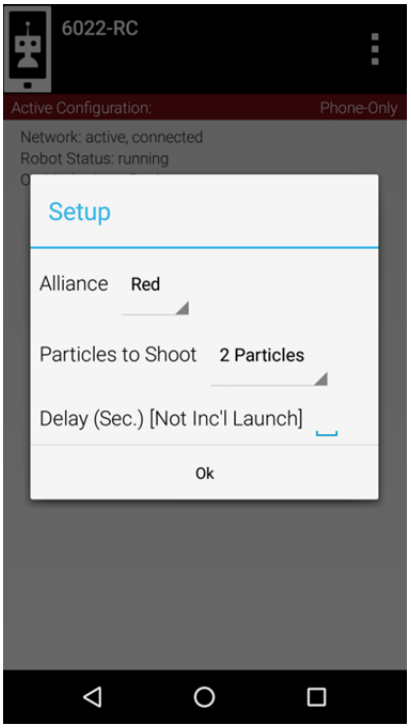
Flowchart:



Data Model:

Code Description and Considerations:

- Using the LASA FTC Library, we implemented the option menu selection package in order to give us the flexibly to specify options or parameters for autonomous routines. (In fact, LASA, the team that developed this, was our first select partner during the Cascade Effect World Championship!)
- Currently, we use an alliance color option for mirroring our drive path for each side of the field.
- There is also an option for choosing how many particles to shoot. Although we would normally shoot 2, we could decrease this to save time and more quickly get to the beacons.
- Finally, we have a time delay parameter to give us flexibility with other teams in order to allow us to work with other teams effectively in autonomous.
- This code structure allows for future parameters and options to be added as needed.

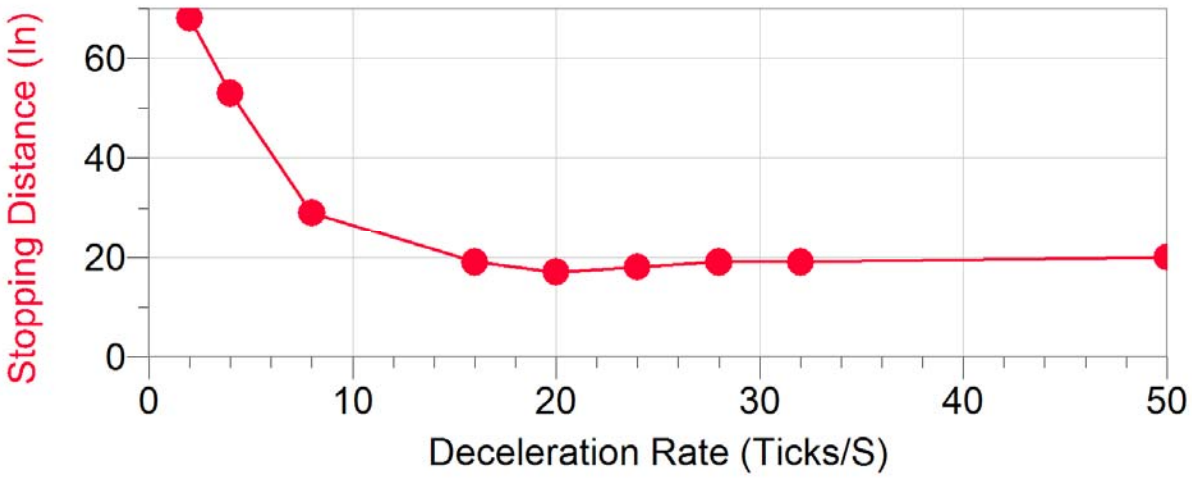


DC Motor Acceleration Tuning

Experiment Description and Considerations:

- To minimize stopping distance, and maintain controllability of our drivetrain, we needed to tune our motor deceleration rate.
    - Stopping distance is minimized when the motors are decelerating as fast as possible so that they aren't slipping or skidding. Having a high deceleration rate is ideal, but once the wheels skid, then it will take longer to stop due to the loss of friction with the ground.
    - In order to find our ideal deceleration rate, we measured the stopping distance without any control algorithm as a control metric. This turned out to be 20 inches.
    - Then, we varied the deceleration rate, as shown in the graph.
    - As expected, the stopping distance initially decreases as the deceleration rate increases; however, once the minimum is reached, the stopping distance will increase again as it asymptotically approaches the control value.
- From our data, this turned out to be a deceleration rate of 20 motor power units per second, which decreases stopping to 17 inches.

Graph:

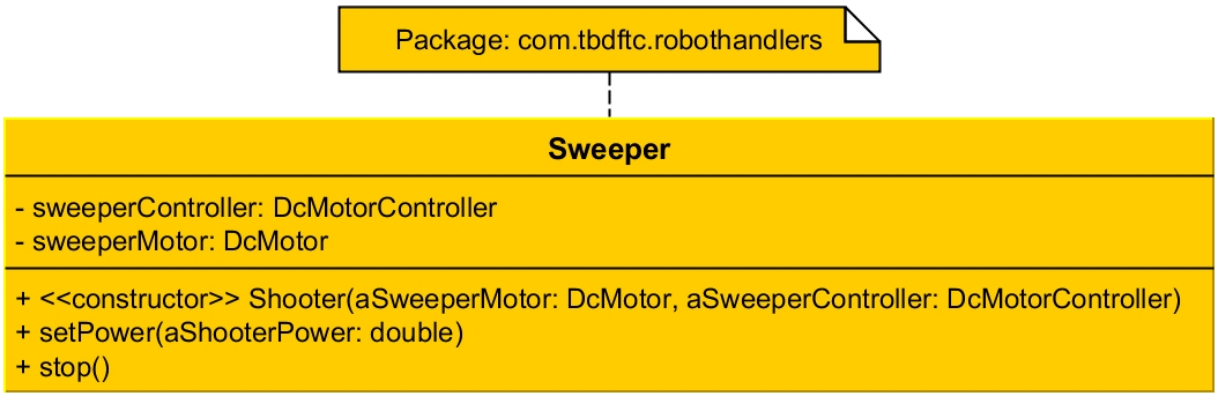


Minor Code Revisions and Hotfixes:

- Implemented a maximum power so that the driver can more easily maintain control.
- Change the initial movement so that it jumps to the minimum power instead of incrementing by the small delta-velocity value and then jumping to minimum power.

Basic Particle Sweeper Class

Data Model:



Code Description and Considerations:

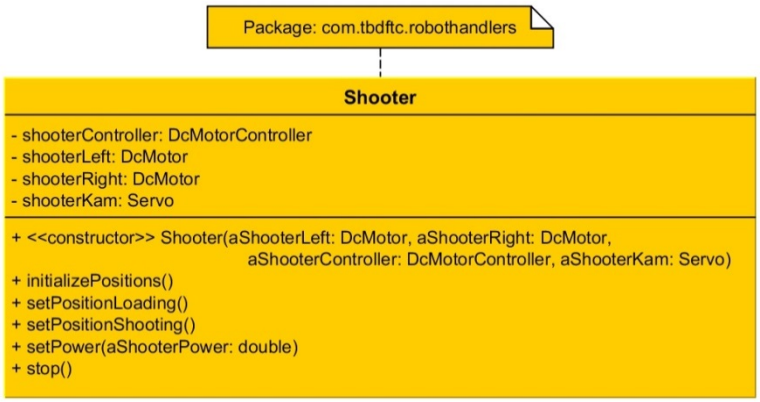
- This class allows us to test our prototype particle sweeper.
- The control is currently assigned to the gamepad's right stick y-axis, but we may change this in the future.
- The sweeper only spins at 80% of its maximum power to improve particle handling and collection.

Class File:

```
public class Sweeper {

    private DcMotorController sweeperController;
    private DcMotor sweeperMotor;

    public Sweeper(DcMotor aSweeperMotor, DcMotorController aSweeperController) {
        sweeperController = aSweeperController;
        sweeperMotor = aSweeperMotor;
    }

    public void setPower(double aSweeperPower) {
        sweeperMotor.setPower(aSweeperPower * 0.8);
    }

    public void stop() {
        sweeperMotor.setPower(0.0);
    }

}
```

Particle Shooter Class with Flicker Servo

Data Model:



Code Description and Considerations:

- In order to allow us to shoot the particles we collect, we added a servo to our shooting device that flicks a ball into the launcher motors.
- To implement this servo, there are two constants that represent positions: one for loading and one for shooting.
- Then, we coded three methods. Two of these set the servo to the desired position using the aforementioned constant values. The third is for initializing the robot and currently only calls the loading position method. This is separate in case our initialization routine is ever differentiated.

Class File:

```
public class Shooter {

    private static final double POSITION_KAM_LOADING = 0.48;
    private static final double POSITION_KAM_SHOOTING = 0.61;

    private DcMotorController shooterController;
    private DcMotor shooterLeft;
    private DcMotor shooterRight;
    private Servo shooterKam; //Named after the man, the myth, the legend: Kameron Fry.

    public Shooter(DcMotor aShooterLeft, DcMotor aShooterRight,
                   DcMotorController aShooterController, Servo aShooterKam) {
        shooterController = aShooterController;
        shooterLeft = aShooterLeft;
        shooterRight = aShooterRight;
        shooterKam = aShooterKam;
        shooterRight.setDirection(REVERSE);
        initializePositions();
    }

    public void initializePositions() { setPositionLoading(); }

    public void setPositionLoading() { shooterKam.setPosition(POSITION_KAM_LOADING); }

    public void setPositionShooting() { shooterKam.setPosition(POSITION_KAM_SHOOTING); }

    public void setPower(double aShooterPower) {
        shooterLeft.setPower(aShooterPower);
        shooterRight.setPower(aShooterPower);
    }

    public void stop() {
        shooterLeft.setPower(0.0);
        shooterRight.setPower(0.0);
    }

}
```

Turning Speed with Motor Acceleration

Code Description and Considerations:

- With the motor acceleration control code in place as it is, our drive team noted that the robot struggled to turn quickly, despite linear driving working well.
- In order to remedy this, there are now two sets of acceleration/decelerating rates: one for driving straight and one for turning.
- To speed up turning, the acceleration and deceleration rates for turning are higher than those of driving straight, as shown in the code below.
- To switch between the two acceleration modes, when the joystick input is being sent through the set power method, if the powers of each side have the same signs, then the robot is driving straight and the acceleration rate is set to go straight. When the powers of each side have opposite signs, then the rates for turning are used.
    - To test for same signs, this Boolean is evaluated:
    - `(aLeftPower < 0) == (aRightPower < 0)`
- As a minor change, the method for setting acceleration rate is now private to ensure that it is set by class methods only.

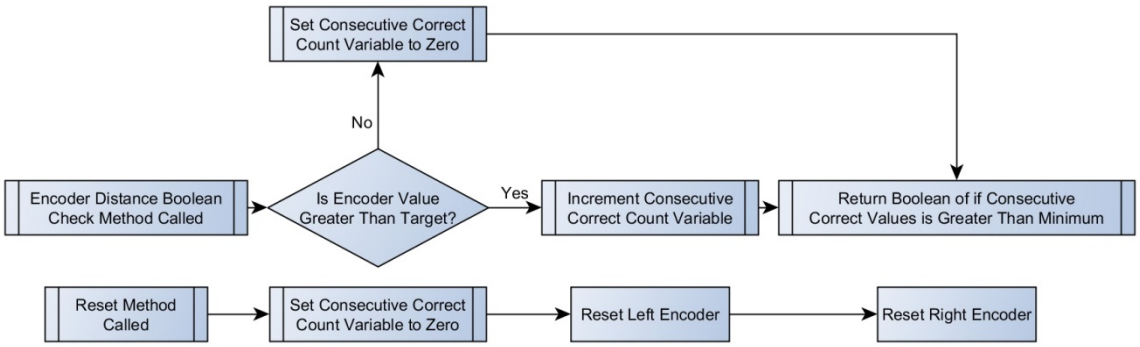
Modified Class File Methods:

```
private static final double WHEEL_ACCEL_SPEED_PER_SECOND_STRAIGHT = 1.5;
private static final double WHEEL_DECEL_SPEED_PER_SECOND_STRAIGHT = 20;
private static final double WHEEL_ACCEL_SPEED_PER_SECOND_TURNING = 20;
private static final double WHEEL_DECEL_SPEED_PER_SECOND_TURNING = 20;

public void setPower(double aLeftPower, double aRightPower) {

    if((aLeftPower < 0) == (aRightPower < 0)) {
        setAccelerationRate(WHEEL_ACCEL_SPEED_PER_SECOND_STRAIGHT,
                            WHEEL_DECEL_SPEED_PER_SECOND_STRAIGHT);
    }
    else {
        setAccelerationRate(WHEEL_ACCEL_SPEED_PER_SECOND_TURNING,
                            WHEEL_DECEL_SPEED_PER_SECOND_TURNING);
    }

    wheelLeftFront.setTargetPower(aLeftPower);
    wheelLeftRear.setTargetPower(aLeftPower);
    wheelRightFront.setTargetPower(aRightPower);
    wheelRightRear.setTargetPower(aRightPower);

}

private void setAccelerationRate(double anAcceleration, double aDeceleration){

    wheelLeftFront.setAccelerationRates(anAcceleration, aDeceleration);
    wheelLeftRear.setAccelerationRates(anAcceleration, aDeceleration);
    wheelRightFront.setAccelerationRates(anAcceleration, aDeceleration);
    wheelRightRear.setAccelerationRates(anAcceleration, aDeceleration);

}
```

Drive Encoder Implementation

Data Model:

Sporadic Value Correction Flowchart:



Drive Encoder Implementation

Code Description and Considerations:

Accessing Encoder via Controller Objects:

- In the FTC SDK, reading the encoder value of a motor from a DC motor object is more resource intensive as the DC motor object's method calls its controller attribute to obtain the value.
- Therefore, to improve encoder access efficiency, the drivetrain class has attributes for both drive controllers and uses encoder port constants to read the encoder values directly.

Averaging Values to Improve Accuracy:

- To give a more precise encoder value and remedy any effects of PID tuning or off-center driving, the two sides' encoder values are averaged when the get encoder magnitude method is called.
- When averaging, this method uses absolute values to ensure that both are positive.
- However, it is important to consider direction and to reset encoders when changing direction.

Sporadic Value Correction:

- In some cases, the motor encoder may return too big or too small of a value, which can result in code jumping ahead in a loop or logic condition. To prevent this, the autonomous code uses a Boolean method to determine if the encoders are past a target value.
- When checking to see if the encoders have gone the proper distance, a variable is incremented if it is the proper distance and is set to zero if it falls short. Once this consecutive correct value variable is greater than or equal to five, the method returns true so that the program can safely proceed knowing that five consecutive motor encoder readings were above the threshold.
- It is required that the "are encoders past" Boolean method is used in an iterative loop, such as an opmode, to ensure that the consecutive correct value passing count is updated.

Virtual Encoder Reset to Reduce Latency:

- Given that the ModernRobotics motor controllers take quite a bit of time to change from run mode to encoder reset mode and back, we decided to implement virtual encoder reset methods that set a variable to the current position. This variable is then subtracted from the actual position when reading distance, thus giving a relative distance since the last reset.
- This is much less time consuming than waiting for the latent native motor controller reset.
- To start with zeroed encoders, the encoders are reset in the initialization before being set to their normal run mode.

Future Improvements:

- The stall detection method needs to be implemented, but it can be copied from last year's code.
- If time permits, these methods will be improved to utilize both encoders on each side of the robot. This will provide redundancy because if the first encoder fails to read properly, the code can detect the issue and return the value from the second encoder on that side.

Drive Encoder Implementation

Class File:

```
public class Drivetrain {

    private static final int ENCODER_PORT_1 = 1;
    private static final int ENCODER_PORT_2 = 2;
    private static final double WHEEL_ACCEL_SPEED_PER_SECOND_STRAIGHT = 1.5;
    private static final double WHEEL_DECEL_SPEED_PER_SECOND_STRAIGHT = 20;
    private static final double WHEEL_ACCEL_SPEED_PER_SECOND_TURNING = 20;
    private static final double WHEEL_DECEL_SPEED_PER_SECOND_TURNING = 20;
    private static final double WHEEL_MINIMUM_POWER = 0.4;
    private static final double WHEEL_MAXIMUM_POWER = 1.0;

    private int driveEncoderCorrectionPassings;
    private final int driveEncoderCorrectionThreshold = 5;
    private int virtualEncoderZeroLeft;
    private int virtualEncoderZeroRight;

    private DcMotorAccelerationThread wheelAccelerationThread;
    private DcMotorController wheelControllerLeft;
    private DcMotorController wheelControllerRight;
    private DcMotorAccelerated wheelLeftFront;
    private DcMotorAccelerated wheelLeftRear;
    private DcMotorAccelerated wheelRightFront;
    private DcMotorAccelerated wheelRightRear;

    public Drivetrain(DcMotor aWheelLeftFront, DcMotor aWheelLeftRear,
                      DcMotorController aControllerLeft, DcMotor aWheelRightFront,
                      DcMotor aWheelRightRear, DcMotorController aControllerRight) {
        //Method code omitted from this page because it has been already documented earlier.
    }

    public int getEncoderLeft() {
        return wheelControllerLeft.getMotorCurrentPosition(ENCODER_PORT_1) - virtualEncoderZeroLeft;
    }

    public int getEncoderRight() {
        return wheelControllerRight.getMotorCurrentPosition(ENCODER_PORT_1) - virtualEncoderZeroRight;
    }

    public int getEncodersMagnitude() {
        return (Math.abs(getEncoderLeft()) + Math.abs(getEncoderRight())) / 2;
    }

    public boolean isDriveEncodersPast(int aDistance) {
        if(Math.abs(getEncodersMagnitude()) >= aDistance) driveEncoderCorrectionPassings++;
        else driveEncoderCorrectionPassings = 0;
        return driveEncoderCorrectionPassings >= driveEncoderCorrectionThreshold;
    }

    public boolean isEncoderLeftReset() {
        return Math.abs(getEncoderLeft()) <= 10;
    }

    public boolean isEncoderRightReset() {
        return Math.abs(getEncoderRight()) <= 10;
    }

    public boolean isEncodersReset() {
        return isEncoderLeftReset() && isEncoderRightReset();
    }

    public boolean isStalling() {
        //TODO: Implement isStalling.
    }

    private void resetEncoderLeftVirtually() {
        virtualEncoderZeroLeft = wheelControllerLeft.getMotorCurrentPosition(ENCODER_PORT_1);
    }
```

Drive Encoder Implementation

Class File (Continued):

```
    private void resetEncoderRightVirtually() {
        virtualEncoderZeroRight = wheelControllerRight.getMotorCurrentPosition(ENCODER_PORT_1);
    }

    public void resetEncoders() {
        driveEncoderCorrectionPassings = 0;

        resetEncoderLeftVirtually();
        resetEncoderRightVirtually();
    }

    public void resetEncodersPhysically() {
        driveEncoderCorrectionPassings = 0;
        wheelControllerLeft.setMotorMode(ENCODER_PORT_1, DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        wheelControllerRight.setMotorMode(ENCODER_PORT_1, DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    }

    public void runWithoutEncoderPWM() {
        //Method code omitted from this page because it has been already documented earlier.
    }

    public void setPower(double aLeftPower, double aRightPower) {
        //Method code omitted from this page because it has been already documented earlier.
    }

    private void setAccelerationRate(double anAcceleration, double aDeceleration){
        //Method code omitted from this page because it has been already documented earlier.
    }

    public void stop() {
        //Method code omitted from this page because it has been already documented earlier.
    }

}
```

Particle Shooter PID Implementation

Code Description and Considerations:

- One of the problems we encountered with the particle shooter was that if the ball were loaded unevenly, the motor speeds would change and the particle would spin and veer off-course.
- In order to remedy this, the ModernRobotics controller-internal PID loop was enabled. To do this, the controller is set to “run with encoders” mode in the constructor of the shooter handler.
- This is useful as it processes the encoder values and calculates new powers on the controller, freeing up computation power and memory on the robot controller app.
- As a result of this change, the shooter now spins at a constant speed regardless of motor load changes or battery voltage.
- Another minor tweak with the shooter was in the teleop code. To ensure that particles are always being launched as high and far as possible, we made it so that once the analog stick is past a certain threshold, it will run the shooter at full speed. This is beneficial as it eliminates the potential of driver error in the heat of a match.

Modified Class File Constructor:

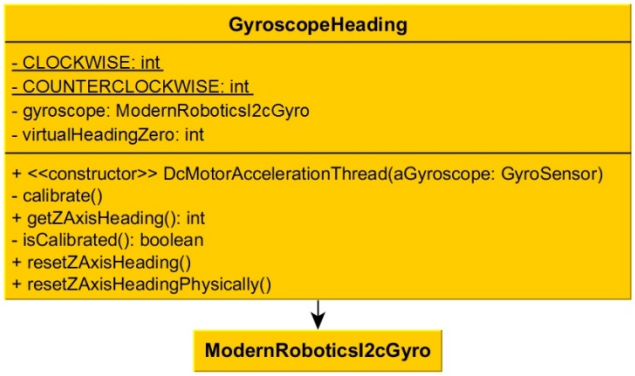
```
public Shooter(DcMotor aShooterLeft, DcMotor aShooterRight,
               DcMotorController aShooterController, Servo aShooterKam) {
    shooterController = aShooterController;
    shooterLeft = aShooterLeft;
    shooterRight = aShooterRight;
    shooterKam = aShooterKam;

    shooterLeft.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
    shooterRight.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
    shooterRight.setDirection(REVERSE);

    initializePositions();
}
```

Modified Teleop Implementation:

```
private static final double STICK_DIGITAL_THRESHOLD = 0.25;

public void loop() {
    //Axes on analog sticks are reversed so that up is positive rather than negative.
    if(-gamepad2.left_stick_y > STICK_DIGITAL_THRESHOLD) shooter.setPower(1.0);
    else if(-gamepad2.left_stick_y < -STICK_DIGITAL_THRESHOLD) shooter.setPower(-1.0);
    else shooter.setPower(0.0);

    if(gamepad2.a) shooter.setPositionShooting();
    else shooter.setPositionLoading();
}
```

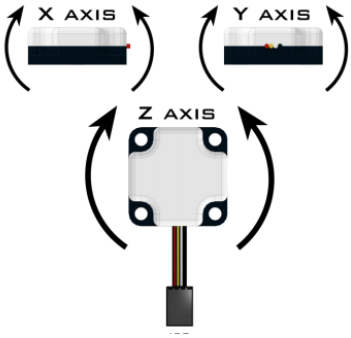
Gyroscope Wrapper Class

Data Model:



Code Description and Considerations:

- Due to the latent nature of interfacing directly with the ModernRobotics integrating gyroscope, we created this wrapper class.
- This wrapper's constructor takes in a gyroscope object from the hardware map and casts it into the ModernRobotics gyroscope type to provide its specific interfaces.
- In the constructor method, the gyroscope is calibrated. When this wrapper is implemented in an opmode, this would be in the initialization method. Therefore, the robot must already have been set down and must remain still during the initialization.
- As the native reset method sends an I2C command to the gyroscope directly, there is significant latency. To reduce this, resetting the gyroscope with the wrapper sets the "virtual zero" variable to the current integrated z value. Then, when the gyro is read with the wrapper, the "virtual zero" is subtracted from the integrated z value to provide the heading relative to the last reset.
- The integrated z value is used as it provides an absolute heading (-∞° to ∞°) as opposed to the get heading method, which provides a Cartesian heading (0° to 359°).
- As shown below to the right, the z axis is used as it corresponds with the robot turn direction.

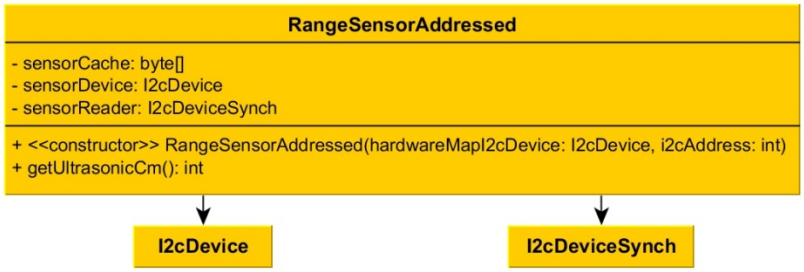


Class File:

```
public class GyroscopeHeading {

    private static final int CLOCKWISE = -1;
    private static final int COUNTERCLOCKWISE = 1;

    private ModernRoboticsI2cGyro gyroscope;
    private int virtualHeadingZero;

    public GyroscopeHeading(GyroSensor aGyroscope) {
        gyroscope = (ModernRoboticsI2cGyro) aGyroscope;
        virtualHeadingZero = 0;
        calibrate();
    }

    private void calibrate() { gyroscope.calibrate(); }

    public int getZAxisHeading() {
        return CLOCKWISE * (gyroscope.getIntegratedZValue() - virtualHeadingZero);
    }

    private boolean isCalibrated() { return !gyroscope.isCalibrating(); }

    public void resetZAxisHeading() { virtualHeadingZero = gyroscope.getIntegratedZValue(); }

    public void resetZAxisHeadingPhysically() { gyroscope.resetZAxisIntegrator(); }

}
```

Range Sensor Wrapper Class

Data Model:



I2C Device Register:

Register	Function
0x00	Sensor Firmware Revision
0x01	Manufacturer Code
0x02	Sensor ID Code
0x03	Unused
0x04	Ultrasonic Reading (cm)
0x05	Raw Optical Distance Reading

Code Description and Considerations:

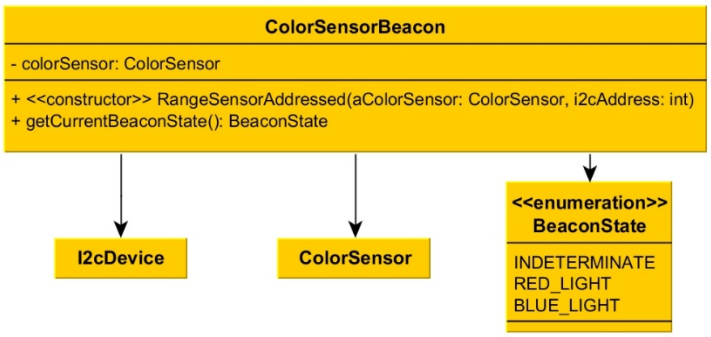
- When implementing two ModernRobotics range sensors, we noticed that the I2C address setting method was not working, despite changing the address of the sensor through Core Device Discovery and then implementing it in the code as such.
- To allow us to use two range sensors, this wrapper uses parameters: a generic I2C device and an I2C address. This allows for two sensors to be used, assuming their addresses are different.
- Based on the SDK code of using sensors, this code maps the I2C device and engages the sensor.
- Then, when the "get ultrasonic value in centimeters" method is used, the class uses the sensor reader instance to read the specified register. If this wrapper ever incorporates the optical distance sensor, the byte array can be made longer to read the next register, which is optical distance sensor.
- To properly sign the value, a bitwise operator is used (`& 0xFF`) to sign the sensor value from 0 cm to 256 cm instead of from -128 cm to 128 cm.
- As shown in the table above, the 0x04 register corresponds to the ultrasonic reading.

Class File:

```
public class RangeSensorAddressed {

    private byte[] sensorCache;
    private I2cDevice sensorDevice;
    private I2cDeviceSynch sensorReader;

    public RangeSensorAddressed(I2cDevice hardwareMapI2cDevice, int i2cAddress) {
        sensorDevice = hardwareMapI2cDevice;
        sensorReader = new I2cDeviceSynchImpl(sensorDevice, I2cAddr.create8bit(i2cAddress), false);
        sensorReader.engage();
    }

    public int getUltrasonicCm() {
        sensorCache = sensorReader.read(0x04, 1);
        return sensorCache[0] & 0xFF;
    }

}
```

Beacon Color Sensor Wrapper Class

Data Model:



Code Description and Considerations:

- To better implement the ModernRobotics color sensor, this wrapper class is used to return the beacon color read by the sensor rather than a raw value or color number.
- To allow us to use multiple color sensors in the future, this wrapper takes in an I2C address and sets the object's color sensor parameter to that I2C address on the Core Device Module.
- This wrapper also turns off the sensor's built-in LED. This ensures that the light returned from the beacon is detected rather than white reflected light.
- When the "get beacon color" method is used, the red and blue channels of the color sensor are compared. If red is greater than blue, then red light is returned and vice-versa. If both the red and blue channels are equal or too near-zero, then indeterminate beacon color is returned.
- Colors are returned in the form of an enumeration from the field abstraction.

Class File:

```
public class ColorSensorBeacon {

    private ColorSensor colorSensor;

    public ColorSensorBeacon(ColorSensor aColorSensor, int i2cAddress) {
        colorSensor = aColorSensor;
        colorSensor.setI2cAddress(I2cAddr.create8bit(i2cAddress));
        colorSensor.enableLed(false);
    }

    public BeaconState getCurrentBeaconState() {
        if(colorSensor.red() > colorSensor.blue())  return RED_LIGHT;
        else if(colorSensor.blue() > colorSensor.red()) return BLUE_LIGHT;
        else return INDETERMINATE;
    }

}
```

Drivetrain Acceleration Control Bypass

Code Description and Considerations:

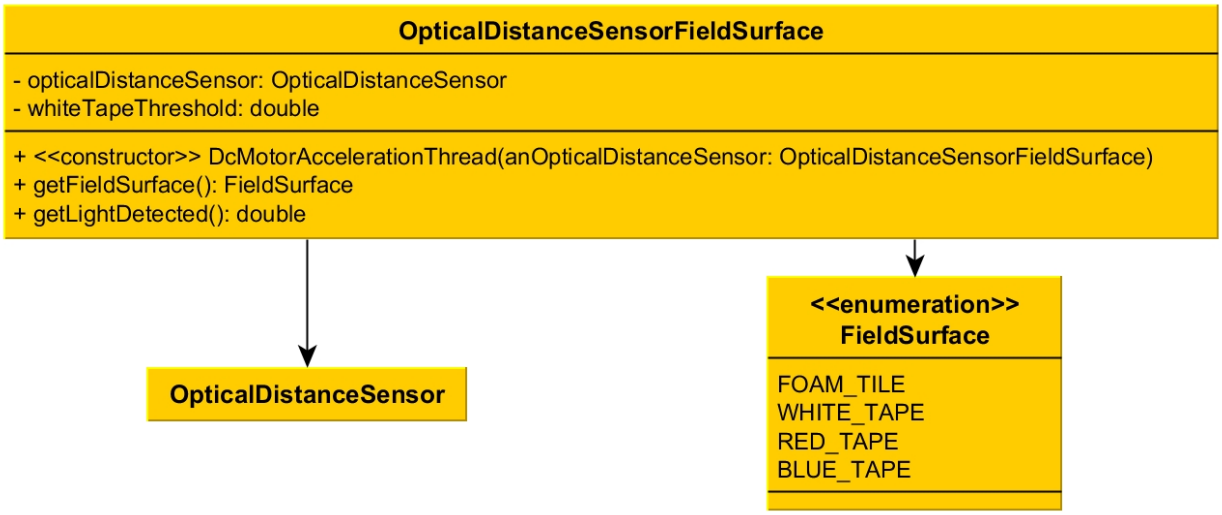
- While developing autonomous, a frequent problem was that when driving at slower speeds, the motor acceleration control either took too long to get the robot to full speed or would not power the motor due to the value being below the minimum.
- To get past this, there is now a method for driving without acceleration control.
- This is used only in autonomous when the robot is at slower speeds. Acceleration control remains the same as it has been in teleop.
- When the "set power without accelerating" method is called, a parameter for each side's power is passed to the four accelerated motor objects, which sets the target power, current power, and physical motor power to the parameter value. This therefore bypasses the acceleration control and allows for more fine-tune driving in autonomous.

Modified Class File Methods:

```
public class DcMotorAccelerated {

    public synchronized void setDirectPower(double aPower) {
        currentPower = aPower;
        setTargetPower(aPower);
        acceleratedMotor.setPower(aPower);
    }

}


public class Drivetrain {

    public void brake() {
        wheelLeftFront.stopMotorHard();
        wheelLeftRear.stopMotorHard();
        wheelRightFront.stopMotorHard();
        wheelRightRear.stopMotorHard();
    }
    public void setPowerWithoutAcceleration(double aLeftPower, double aRightPower) {
        wheelLeftFront.setDirectPower(aLeftPower);
        wheelLeftRear.setDirectPower(aLeftPower);
        wheelRightFront.setDirectPower(aRightPower);
        wheelRightRear.setDirectPower(aRightPower);
    }

}
```

Field Optical Sensor Wrapper Class

Data Model:



Code Description and Considerations:

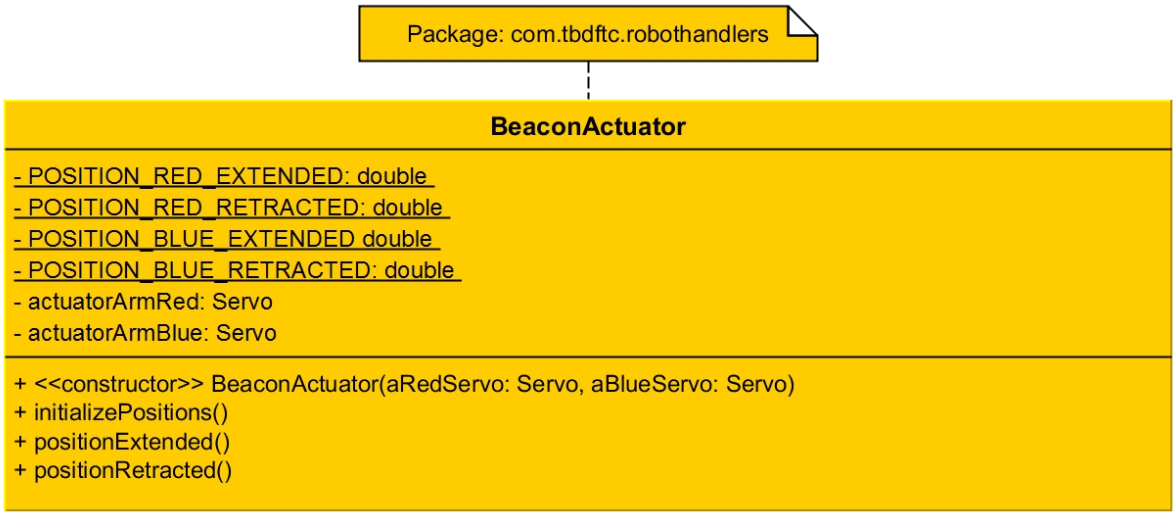
- In our current autonomous program, we are using the white line on the field as a reference point to zero out encoders, so we needed a way to sense the white line.
- To implement this, we used the optical distance sensor through a wrapper class.
- This class creates a white tape threshold by reading the value of the field during initialization and adds 0.025 to that.
- Then, if the light detected is greater than the threshold, the class returns that the field surface is white tape. If it is less than the threshold, the class returns that the field is the foam tile.
- One benefit of using the optical distance sensor over a color sensor is that an ODS takes up an analog port, which we have plenty of, compared to an I2C port, which we have almost filled.

Class File:

```
public class OpticalDistanceSensorFieldSurface {

    private OpticalDistanceSensor opticalDistanceSensor;
    private double whiteTapeThreshold;

    public OpticalDistanceSensorFieldSurface(OpticalDistanceSensor aOpticalDistanceSensor) {
        opticalDistanceSensor = aOpticalDistanceSensor;
        whiteTapeThreshold = opticalDistanceSensor.getLightDetected() + 0.025;
    }

    public FieldSurface getFieldSurface() {
        if(opticalDistanceSensor.getLightDetected() > whiteTapeThreshold) return WHITE_TAPE;
        else return FOAM_TILE;
    }

    public double getLightDetected() {
        return opticalDistanceSensor.getLightDetected();
    }

}
```

Beacon Actuator Class

Data Model:



Code Description and Considerations:

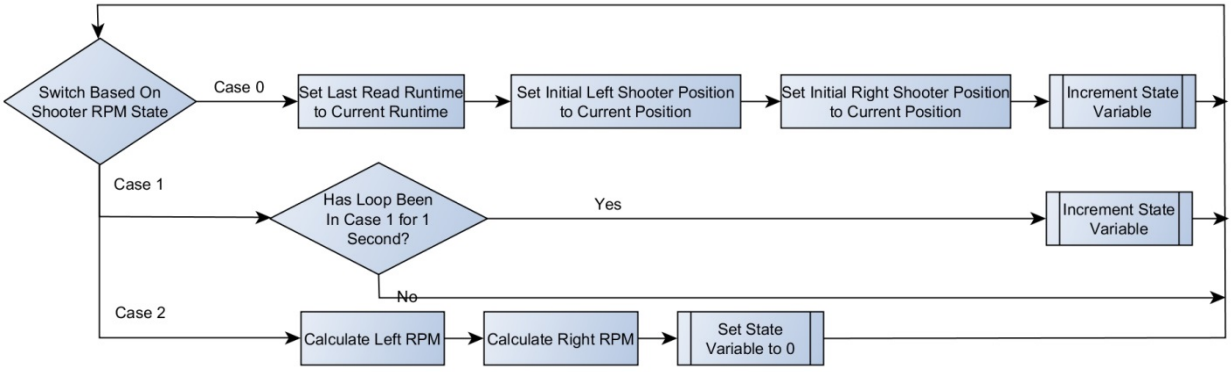
- This class controls the two arms that press the beacons in autonomous.
- Because only one servo is plugged in at a time, both are set using the same method calls and the one that is plugged in will then move to the appropriate value.
- The servo positions are set as constants that can be easily changed in the class file.

Class File:

```
public class BeaconActuator {

    private static final double POSITION_RED_EXTENDED = 0.21;
    private static final double POSITION_RED_RETRACTED = 0.71;
    private static final double POSITION_BLUE_EXTENDED = 0.21;
    private static final double POSITION_BLUE_RETRACTED = 0.71;
    Servo actuatorArmRed;
    Servo actuatorArmBlue;

    public BeaconActuator(Servo aRedServo, Servo aBlueServo) {
        actuatorArmRed = aRedServo;
        actuatorArmBlue = aBlueServo;
        initializePositions();
    }

    public void initializePositions() { positionRetracted(); }

    public void positionExtended() {
        actuatorArmRed.setPosition(POSITION_RED_EXTENDED);
        actuatorArmBlue.setPosition(POSITION_BLUE_EXTENDED);
    }

    public void positionRetracted() {
        actuatorArmRed.setPosition(POSITION_RED_RETRACTED);
        actuatorArmBlue.setPosition(POSITION_BLUE_RETRACTED);
    }

}
```

Shooter RPM Readings via Telemetry

Flowchart:



Code Description and Considerations:

- To allow the drive team to know the speed of the shooters so that they can better know where to shoot from, this state machine reads the encoders and calculates the RPM every second.
- To convert motor ticks per second to RPM, the following formula is used:

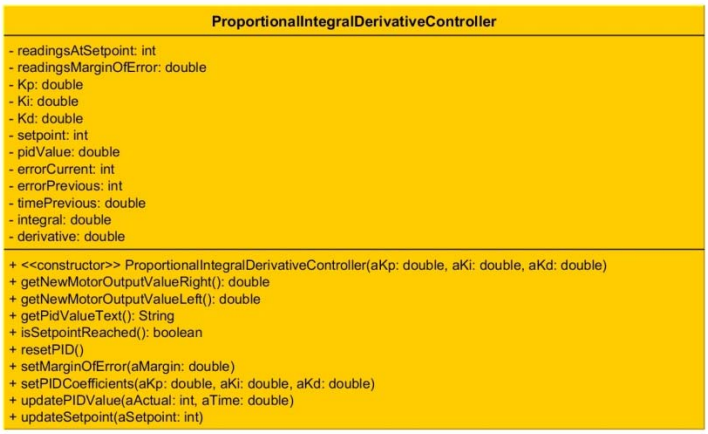
$$\frac{1\ Rotation}{1\ Minute} = \frac{1\ Encoder\ Tick}{1\ Second} * \frac{60\ Seconds}{1\ Minute} * \frac{1\ Rotation}{1440\ Encoder\ Ticks} * \frac{4\ (Gearbox)}{1\ (Gearbox)} \approx \frac{1\ Encoder\ Tick}{1\ Second} * \mathbf{0.167}$$

New Class File Code:

```
public class TeleopBasic extends OpMode {
    private double lastReadRuntime;
    private int shooterStateMachineFlow;
    private int lastShooterPositionLeft;
    private int lastShooterPositionRight;
    private double shooterRpmLeft;
    private double shooterRpmRight;

    @Override
    public void loop() {
        switch(shooterStateMachineFlow) {
            case 0:
                lastReadRuntime = getRuntime();
                lastShooterPositionLeft = shooter.getEncoderLeft();
                lastShooterPositionRight = shooter.getEncoderRight();
                shooterStateMachineFlow++;
                break;
            case 1:
                if(getRuntime() - lastReadRuntime > 1.0) shooterStateMachineFlow++;
                break;
            case 2:
                //From 60 Sec/Min / 1440 Ticks/Rev * 4:1 Gearbox
                shooterRpmLeft = (shooter.getEncoderLeft() - lastShooterPositionLeft) * 0.1667;
                shooterRpmRight = (shooter.getEncoderRight() - lastShooterPositionRight) * 0.1667;
                shooterStateMachineFlow = 0;
                break;
        }

        telemetry.addData("Shooter RPM (L)", shooterRpmLeft);
        telemetry.addData("Shooter RPM (R)", shooterRpmRight);
    }

}
```

Custom PID Controller Loop

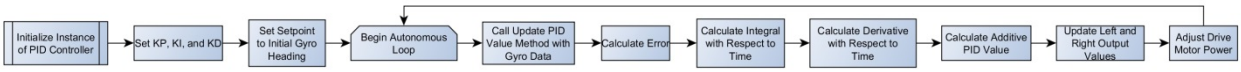
Data Model:



Zeigler-Nichols Tuning Table:

Control Type	K_p	T_i	T_d
P	$0.5K_u$	-	-
PI	$0.45K_u$	$T_u/1.2$	-
PD	$0.8K_u$	-	$T_u/8$
classic PID	$0.60K_u$	$T_u/2$	$T_u/8$
Pessen Integral Rule	$0.7K_u$	$T_u/2.5$	$3T_u/20$
Some Overshoot	$0.33K_u$	$T_u/2$	$T_u/3$
No Overshoot	$0.2K_u$	$T_u/2$	$T_u/3$

Flowchart:



Code Description and Considerations:

- We decided to write our own proportional-integral-derivative loop because there was no compact Java PID library that fit our needs and we thought it would be fun to write our own. ☺
- To tune our PID controller, we used the Ziegler-Nichols method, as shown in the table above.
- For turning, we chose to use proportional-derivative control only because proportionality and differentiability are useful for stabilization, whereas adding an integral component to the error would result in unnecessary swerve and overshoot.
- For driving straight, we use proportional control only so that it is responsive and more stable.
- The use of a PID loop with respect to a gyroscope is especially useful in this autonomous context, as it helps in the event if we get knocked off course by driving on debris or being hit by a robot running a defensive autonomous program.

Class File:

```
public class ProportionalIntegralDerivativeController {
    private double readingsMarginOfError;
    private double Kp;
    private double Ki;
    private double Kd;
    private double setpoint;
    private double pidValue;
    private double errorCurrent;
    private double errorPrevious;
    private double timePrevious;
    private double integral;
    private double derivative;
```

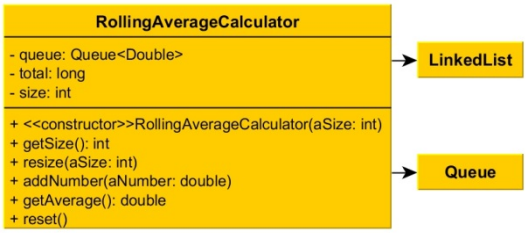
Custom PID Controller Loop

Class File (Continued):

```
    public ProportionalIntegralDerivativeController(double aKp, double aKi, double aKd) {
        setPidCoefficients(aKp, aKi, aKd);
        updateSetpoint(0);
        readingsMarginOfError = 2.0;
        resetPid(0);
    }

    public double getNewMotorOutputValueRight(double aMotorPower) {
        if(Math.abs(pidValue) > 0.01) return Range.clip(aMotorPower - pidValue, -1, 1);
        else return aMotorPower;
    }

    public double getNewMotorOutputValueLeft(double aMotorPower) {
        if(Math.abs(pidValue) > 0.01) return Range.clip(aMotorPower + pidValue, -1, 1);
        else return aMotorPower;
    }

    public boolean isSetpointReahced() {
        return readingsAtSetpoint >= SETPOINT_REACHED_READINGS_THRESHOLD;
    }

    public void resetPid(double aRuntime) {
        pidValue = 0;
        readingsAtSetpoint = 0;
        timePrevious = aRuntime;
        errorCurrent = 0;
        errorPrevious = 0;
        integral = 0;
        derivative = 0;
    }

    public void setMarginOfError(double aMargin) { readingsMarginOfError = aMargin; }

    public void setPidCoefficients(double aKp, double aKi, double aKd) {
        Kp = aKp;
        Ki = aKi;
        Kd = aKd;
    }

    public void updatePidValue(double aActual, double aRuntime) {
        errorCurrent = setpoint - aActual;
        integral += (errorCurrent * (aRuntime - timePrevious));
        derivative = (errorCurrent - errorPrevious) / (aRuntime - timePrevious);

        pidValue = Kp*errorCurrent + Ki*integral + Kd*derivative;
        errorPrevious = errorCurrent;
        timePrevious = aRuntime;
        if(aActual < (setpoint + readingsMarginOfError) && aActual > (setpoint -
readingsMarginOfError)) readingsAtSetpoint++;
        else readingsAtSetpoint = 0;
    }

    public void updateSetpoint(int aSetpoint) {
        setpoint = aSetpoint;
        readingsAtSetpoint = 0;
    }

}
```

Custom Rolling Average Calculator

Data Model:



Code Description and Considerations:

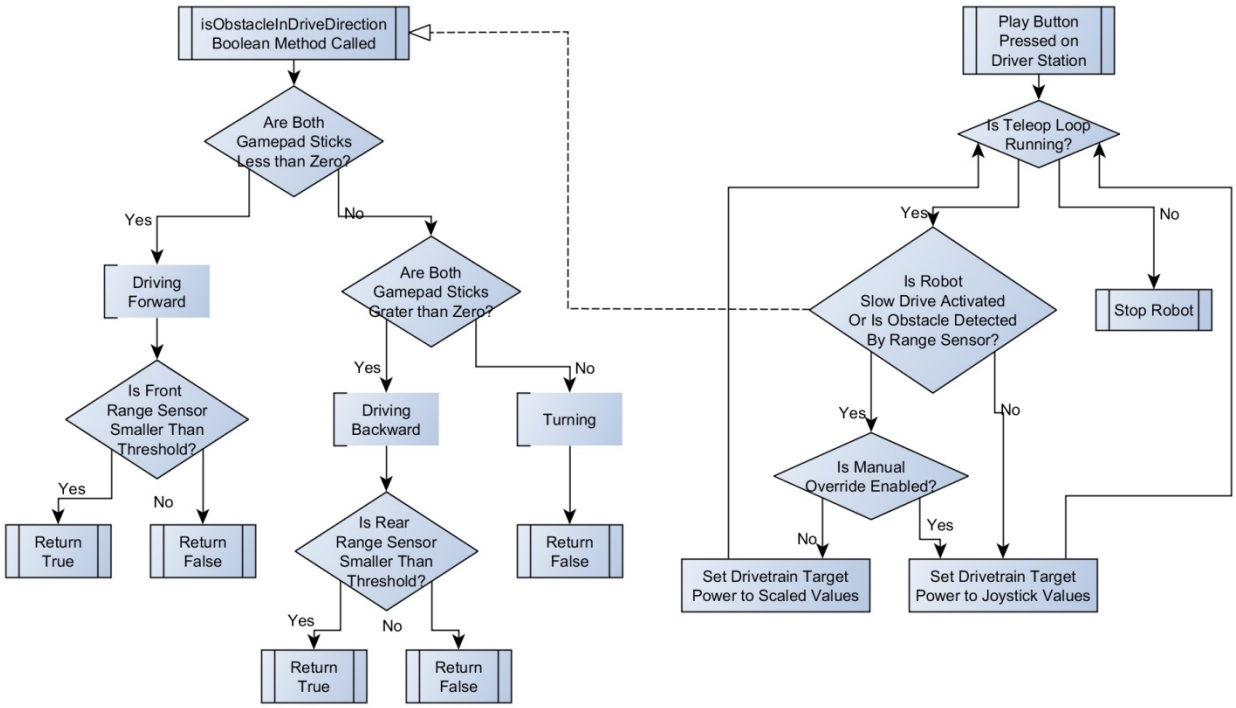
- When implementing the gyroscope into our PID control loop, we found that the derivative value would often read zero or sporadically change due to the gyroscope only returning an integer.
- To fix this, we tried using the native rolling average class, but this also only returns an integer.
- So, we wrote our own custom rolling average calculator that supports doubles.
- This allows us to average the last few gyroscope headings and approximate a decimal value, especially when it is on the verge of changing between two headings.

Class File:

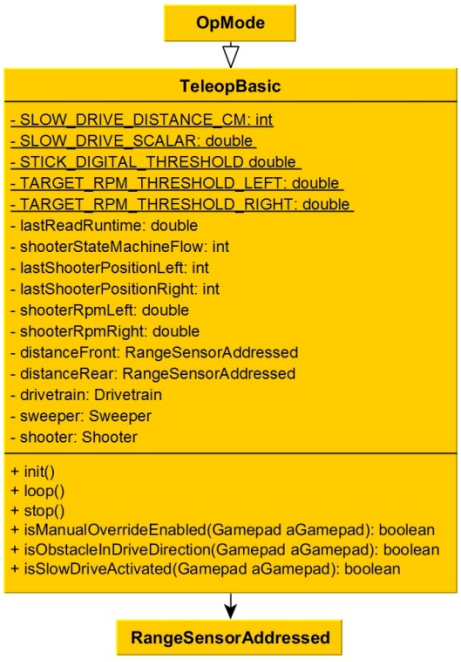
```
public class RollingAverageCalculator {
    private final Queue<Double> queue = new LinkedList();
    private long total;
    private int size;

    public RollingAverageCalculator(int size) {
        this.resize(size);
    }

    public int size() {
        return size;
    }

    public void resize(int aSize) {
        size = aSize;
        queue.clear();
    }

    public void addNumber(double aNumber) {
        if(queue.size() >= size) {
            double last = queue.remove();
            total -= (long)last;
        }

        queue.add(Double.valueOf(aNumber));
        total += (long)aNumber;
    }

    public double getAverage() {
        return queue.isEmpty()?0:(double)(this.total / (long)this.queue.size());
    }

    public void reset() {
        queue.clear();
    }
}
```

Ultrasonic Autonomous Robot Slowdown for Teleop Driving

Flowchart:



Data Model:



Code Description and Considerations:

- In order to improve controllability in the teleop period, this sensor system takes advantage of multiple range sensors to detect obstacles, such as a wall or other robot, and slow down to prevent hard collisions.
- This collision detection system works by enabling a slow drive mode if either an object is detected or slow drive is manually enabled.
- There is a manual override to allow for full-speed driving in the unlikely event of a sensor failure.
- To detect if an object is in the way, the teleop class file has a method that returns a Boolean reflecting whether or not there is an obstacle. This method uses the driver gamepad as a parameter to read the current driving direction. Then, the corresponding ultrasonic sensor is read and is checked to see if the sensor is below a user-specified constant threshold.

Ultrasonic Autonomous Robot Slowdown for Teleop Driving

Modified Class File Methods:

```
@Override
public void loop() {
    //Note that axes on analog sticks are reversed so that up is positive rather than negative.

    //Drivetrain at slower speed.
    if(isSlowDriveActivated(gamepad1) || (isObstacleInDriveDirection(gamepad1)) &&
!isManualOverrideEnabled(gamepad1)) {
        drivetrain.setMinimumMotorPower(0.1);
        drivetrain.setPower(-gamepad1.left_stick_y * SLOW_DRIVE_SCALAR, -gamepad1.right_stick_y *
SLOW_DRIVE_SCALAR);
    }
    //Drivetrain at regular speed.
    else {
        drivetrain.setMinimumMotorPower(0.35);
        drivetrain.setPower(-gamepad1.left_stick_y, -gamepad1.right_stick_y);
    }

    //Other code omitted as it is already documented in other entries.
}

public boolean isManualOverrideEnabled(Gamepad aGamepad) {
    return (aGamepad.right_bumper || aGamepad.left_bumper);
}

public boolean isObstacleInDriveDirection(Gamepad aGamepad) {
    //Driving forward.
    if(aGamepad.left_stick_y < 0 && aGamepad.right_stick_y < 0) {
        return distanceFront.getUltrasonicCm() < SLOW_DRIVE_DISTANCE_CM;
    }
    //Driving backward.
    else if(aGamepad.left_stick_y > 0 && aGamepad.right_stick_y > 0) {
        return distanceRear.getUltrasonicCm() < SLOW_DRIVE_DISTANCE_CM;
    }
    //Turning.
    else {
        return false;
    }

}

public boolean isSlowDriveActivated(Gamepad aGamepad) {
    return (aGamepad.left_trigger > STICK_DIGITAL_THRESHOLD || aGamepad.right_trigger >
STICK_DIGITAL_THRESHOLD);
}
```

Turning Deadband Calculator Opmode for PID Tuning

Code Description and Considerations:

- A critical part of our turning in autonomous relies on PID control. To ensure that this control is accurate, there is deadband compensation so that and change in the PID value has an effect on the turning without stalling and straining the drive motors.
- However, this deadband changes on different field surfaces or base flooring. Therefore, we needed a way to calibrate this deadband value quickly.
- This opmode allows for deadband calculation by measuring the initial gyro rate and slowly ramping up the motor powers in an on-the-spot turn until motion is achieved. The final motor power is then coded in as the deadband value.

Opmode Class File Loop:

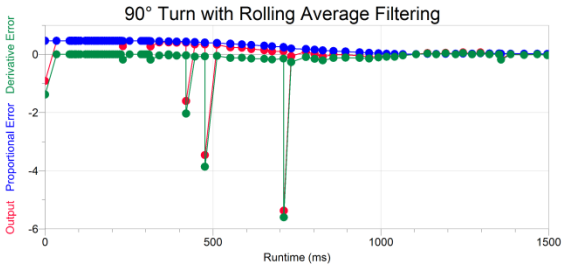
```
public void loop() {

    if(!isDeadbandFound) {
        switch(speedIncrementStateMachineFlow) {
            case 0:
                runtime.reset();
                speedIncrementStateMachineFlow++;
                break;
            case 1:
                if(runtime.seconds() > 0.1) speedIncrementStateMachineFlow++;
                break;
            case 2:
                currentMotorPower += 0.01;
                speedIncrementStateMachineFlow = 0;
                break;
        }

        if(Math.abs(initialGyroRate - gyroscope.getZAxisRate()) > 500) {
            measuredDeadband = currentMotorPower;
            deadandPassings++;
        }
        else {
            deadandPassings = 0;
        }

        if(deadandPassings > 7) isDeadbandFound = true;

        drivetrain.setPowerWithoutAcceleration(-currentMotorPower, currentMotorPower);
    }

    else {
        if(gamepad1.a) drivetrain.setPowerWithoutAcceleration(-measuredDeadband, measuredDeadband);
        else drivetrain.brake();
    }

    telemetry.addData("Current Power", currentMotorPower);
    telemetry.addData("Current Gyro Rate", gyroscope.getZAxisRate());
    telemetry.addData("Initial Gyro Rate", initialGyroRate);
    telemetry.addData("Measured Deadband", measuredDeadband);

}
```

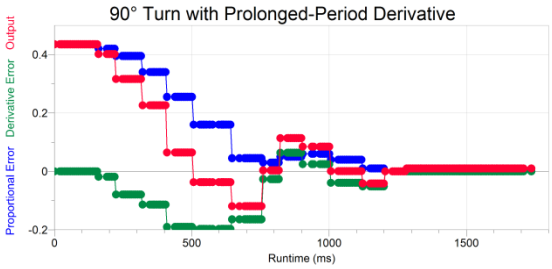
Improved PID Controller Differentiation

Code Description and Considerations:

- Given that the ModernRobotics gyroscope only returns an integer value with no decimal accuracy, when the PID controller took the derivative of the readings between each update, we would often get zero and then spike up to a large magnitude for one loop iteration before returning to zero.
- This led to jerky turning, so we improved our differentiation algorithm to only take a derivative when the gyroscope's heading value changes.
- This has led to much smoother turning, as outlined in the graphs to the right:

Graphs:



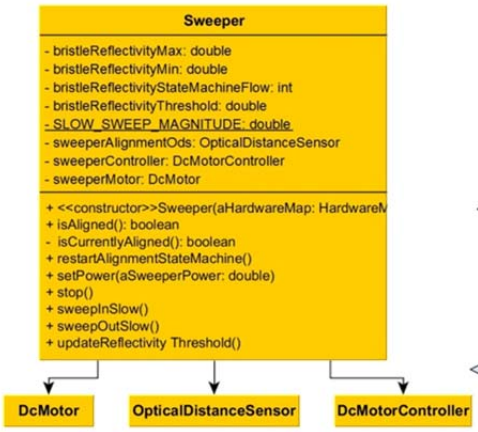


Modified Class File Methods:

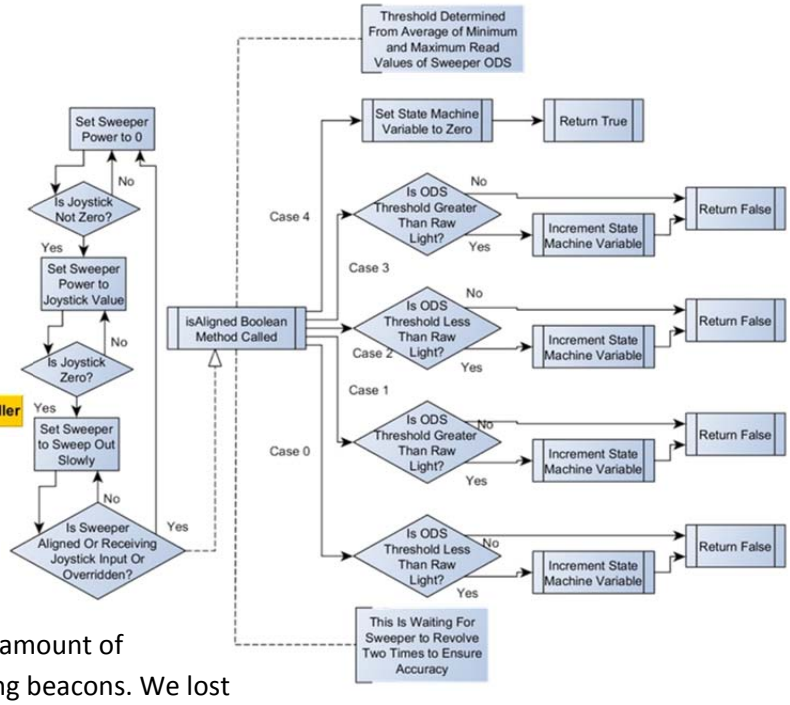
```
public void updatePidValue(double aActual, double aRuntime) {
    errorCurrent = setpoint - aActual;
    integral += (errorCurrent * (aRuntime - timePrevious));

    if(derivativeActualPrevious != aActual) {
        if(errorCurrent == 0 || errorPrevious == 0) derivative = 0;
        else derivative = (errorCurrent - derivativeErrorPrevious) / (aRuntime -
derivativeTimePrevious);

        derivativeActualPrevious = aActual;
        derivativeErrorPrevious = errorCurrent;
        derivativeTimePrevious = aRuntime;
    }

    pidValue = Kp*errorCurrent + Ki*integral + Kd*derivative;

    errorPrevious = errorCurrent;
    timePrevious = aRuntime;

    if(aActual < (setpoint + readingsMarginOfError) && aActual > (setpoint - readingsMarginOfError))
readingsAtSetpoint++;
    else readingsAtSetpoint = 0;
}
```

Particle Sweeper Automatic Alignment

Data Model:



Flowchart:



Code Description and Considerations:

- During our past few competitions, we noticed that we were wasting a significant amount of time in endgame while pressing beacons. We lost time because we needed to manually line up our sweeper so that the bristles wouldn't be pointing towards the wall and interfering with our beacon pushing mechanism.
- To address this, we added an optical distance sensor with a bristle covered with aluminum tape.
- Then, in our robot hander for the sweeper, we have a state machine that determines whether or not the bristle is lined up. In our teleop loop, we have another state machine that rotates the sweeper slowly until it is aligned with the sensor. The robot rotates the sweeper twice to ensure that the bristles have reduced angular momentum and don't overshoot the sensor.
- The threshold is calculated by averaging the maximum and minimum sensor readings during that match. This allows the code to be adaptive at any light level.

Class File:

```
public class Sweeper {

    private static final double SLOW_SWEEP_MAGNITUDE = 0.175;

    private double bristleReflectivityMax;
    private double bristleReflectivityMin;
    private int bristleReflectivityStateMachineFlow;
    private double bristleReflectivityThreshold;
    private OpticalDistanceSensor sweeperAlignmentOds;
    private DcMotorController sweeperController;
    private DcMotor sweeperMotor;
```

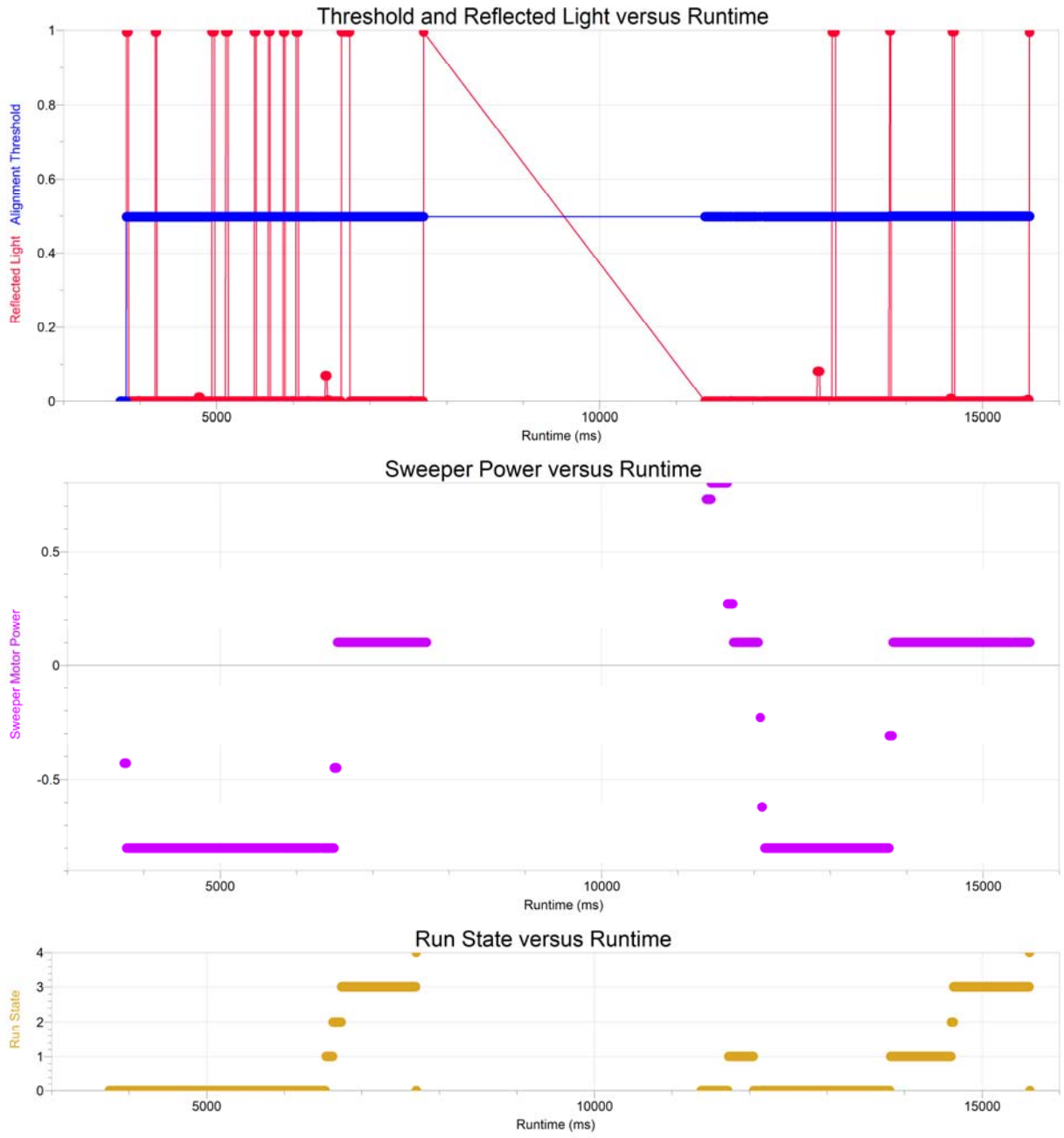
Particle Sweeper Automatic Alignment

Class File (Continued):

```
    public Sweeper(HardwareMap aHardwareMap) {
        sweeperAlignmentOds = aHardwareMap.opticalDistanceSensor.get("particleSweeperOds");
        sweeperController = aHardwareMap.dcMotorController.get("particleSweeperController");
        sweeperMotor = aHardwareMap.dcMotor.get("particleSweeper");

        bristleReflectivityMax = 0.0;
        bristleReflectivityMin = 1.0;
        bristleReflectivityStateMachineFlow = 0;
        bristleReflectivityThreshold = 0.5;
    }

    public boolean isAligned() {
        switch (bristleReflectivityStateMachineFlow) {
            case 0:
                //Should start of out of alignment.
                if(!isCurrentlyAligned()) bristleReflectivityStateMachineFlow++;
                return false;
            case 1:
                //Wait for first reflective pass.
                if(isCurrentlyAligned()) bristleReflectivityStateMachineFlow++;
                return false;
            case 2:
                //Sweep one more revolution to ensure accuracy.
                if(!isCurrentlyAligned()) bristleReflectivityStateMachineFlow++;
                return false;
            case 3:
                //Wait for final reflection.
                if(isCurrentlyAligned()) bristleReflectivityStateMachineFlow++;
                return false;
            default:
                //Return true, but go to initial alignment state for next use.
                bristleReflectivityStateMachineFlow = 0;
                return true;
        }
    }

    private boolean isCurrentlyAligned() {
        return sweeperAlignmentOds.getRawLightDetected() > bristleReflectivityThreshold;
    }

    public void restartAlignmentStateMachine() { bristleReflectivityStateMachineFlow = 0; }

    public void setPower(double aSweeperPower) {
        updateReflectivityThreshold();
        sweeperMotor.setPower(aSweeperPower * 0.8);
    }

    public void stop() { sweeperMotor.setPower(0.0); }

    public void sweepInSlow() { setPower(-SLOW_SWEEP_MAGNITUDE); }

    public void sweepOutSlow() { setPower(SLOW_SWEEP_MAGNITUDE); }

    protected void updateReflectivityThreshold() {
        bristleReflectivityMax = Math.max(bristleReflectivityMax,
sweeperAlignmentOds.getLightDetected());
        bristleReflectivityMin = Math.min(bristleReflectivityMin,
sweeperAlignmentOds.getLightDetected());
        bristleReflectivityThreshold = 0.5 * (bristleReflectivityMax + bristleReflectivityMin);
    }

}
```

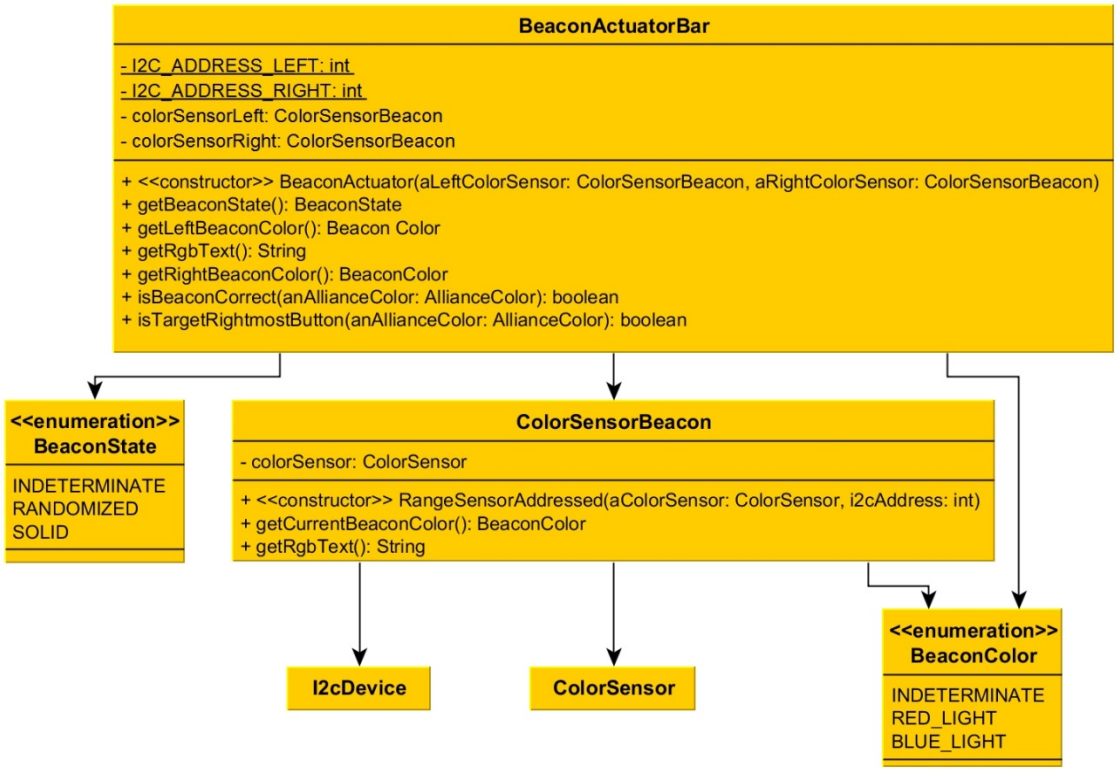
Particle Sweeper Automatic Alignment

Graphs:



Beacon Actuator Bar Class for Implementation of 2 Color Sensors

Data Model:



Code Description and Considerations:

- A problem that we faced was that although the robot could identify which side of the beacon to press with a single color sensor, the robot could not know when the beacon was actuated or if it were successfully hit.
- To remedy this, we developed the above class that encapsulates two beacon color sensor objects (*See Page ES-32)* that allows us to determine if the beacon is randomized, a solid color, or indeterminate.
- This allows the robot to still actively pivot towards the proper button on the beacon.
- In addition, the robot can now stop pressing once both the left and right color sensor read that both sides are lit to the same color and it is the correct color before driving to the next beacon. This is more reliable than trying to determine when to stop pressing through the use of a range sensor.
- This class also has two parameterized constants that allow us to define the I2C addresses of each color sensor with ease.

Beacon Actuator Bar Class for Implementation of 2 Color Sensors

Class File:

```
public class BeaconActuatorBar {

    private static final int I2C_ADDRESS_LEFT = 0x3e;
    private static final int I2C_ADDRESS_RIGHT = 0x3c;

    private ColorSensorBeacon colorSensorLeft;
    private ColorSensorBeacon colorSensorRight;

    public BeaconActuatorBar(HardwareMap aHardwareMap) {
        colorSensorLeft = new ColorSensorBeacon(aHardwareMap.colorSensor.get("beaconColorLeft"),
I2C_ADDRESS_LEFT);
        colorSensorRight = new ColorSensorBeacon(aHardwareMap.colorSensor.get("beaconColorRight"),
I2C_ADDRESS_RIGHT);
    }

    public BeaconState getBeaconState() {
        if(colorSensorLeft.getCurrentBeaconColor() == BeaconColor.INDETERMINATE ||
colorSensorRight.getCurrentBeaconColor() == BeaconColor.INDETERMINATE) return
BeaconState.INDETERMINATE;
        else if(colorSensorRight.getCurrentBeaconColor() == colorSensorLeft.getCurrentBeaconColor())
return BeaconState.SOLID;
        else return BeaconState.RANDOMIZED;
    }

    public BeaconColor getLeftBeaconColor() {
        return colorSensorLeft.getCurrentBeaconColor();
    }

    public String getRgbText() {
        return colorSensorLeft.getRgbText() + " | " + colorSensorRight.getRgbText();
    }

    public BeaconColor getRightBeaconColor() {
        return colorSensorRight.getCurrentBeaconColor();
    }

    public boolean isBeaconCorrect(AllianceColor anAllianceColor) {
        if(!(getBeaconState() == BeaconState.SOLID)) return false;

        if(anAllianceColor == AllianceColor.RED_ALLIANCE && getLeftBeaconColor() ==
BeaconColor.RED_LIGHT && getRightBeaconColor() == BeaconColor.RED_LIGHT) return true;
        else return anAllianceColor == AllianceColor.BLUE_ALLIANCE && getLeftBeaconColor() ==
BeaconColor.BLUE_LIGHT && getRightBeaconColor() == BeaconColor.BLUE_LIGHT;
    }

    public boolean isTargetRightmostButton(AllianceColor anAllianceColor) {
        return anAllianceColor == AllianceColor.RED_ALLIANCE && getRightBeaconColor() == RED_LIGHT ||
anAllianceColor == AllianceColor.BLUE_ALLIANCE && getRightBeaconColor() == BLUE_LIGHT;
    }

}
```

Referencing HardwareMap in Robot Handler Classes with Parameterization

Code Description and Considerations:

- In order to organize our existing code and also allow us to easier make modifications in the future, we adjusted all of our robot handler classes so that they receive the opmode's hardware map as a parameter rather than multiple motor, servo, controller or sensor objects.
- Then, in the method constructor, the hardware map parameter is used to obtain the objects for any robot hardware or sensors.
- This is beneficial as this allows us to change any hardware configuration name in one place rather than in each opmode. In addition, it makes our code look much clearer.

Constructor Method for Drivetrain:

```
public Drivetrain(HardwareMap aHardwareMap) {
    wheelControllerLeft = aHardwareMap.dcMotorController.get("wheelsLeft");
    wheelControllerRight = aHardwareMap.dcMotorController.get("wheelsRight");
    wheelLeftFront = new DcMotorAccelerated(aHardwareMap.dcMotor.get("wheelLeftFront"),
WHEEL_ACCEL_SPEED_PER_SECOND_STRAIGHT, WHEEL_DECEL_SPEED_PER_SECOND_STRAIGHT, WHEEL_MINIMUM_POWER,
WHEEL_MAXIMUM_POWER);
    wheelLeftRear = new DcMotorAccelerated(aHardwareMap.dcMotor.get("wheelLeftRear"),
WHEEL_ACCEL_SPEED_PER_SECOND_STRAIGHT, WHEEL_DECEL_SPEED_PER_SECOND_STRAIGHT, WHEEL_MINIMUM_POWER,
WHEEL_MAXIMUM_POWER);
    wheelRightFront = new DcMotorAccelerated(aHardwareMap.dcMotor.get("wheelRightFront"),
WHEEL_ACCEL_SPEED_PER_SECOND_STRAIGHT, WHEEL_DECEL_SPEED_PER_SECOND_STRAIGHT, WHEEL_MINIMUM_POWER,
WHEEL_MAXIMUM_POWER);
    wheelRightRear = new DcMotorAccelerated(aHardwareMap.dcMotor.get("wheelRightRear"),
WHEEL_ACCEL_SPEED_PER_SECOND_STRAIGHT, WHEEL_DECEL_SPEED_PER_SECOND_STRAIGHT, WHEEL_MINIMUM_POWER,
WHEEL_MAXIMUM_POWER);

    wheelRightFront.setDirection(REVERSE);
    wheelRightRear.setDirection(REVERSE);
    runWithoutEncoderPWM();

    driveEncoderCorrectionPassings = 0;
    measuredPidTurningBasePower = DEFAULT_PID_TURNING_BASE_POWER;

    wheelAccelerationThread = new DcMotorAccelerationThread();
    wheelAccelerationThread.addMotor(wheelLeftFront);
    wheelAccelerationThread.addMotor(wheelLeftRear);
    wheelAccelerationThread.addMotor(wheelRightFront);
    wheelAccelerationThread.addMotor(wheelRightRear);
    wheelAccelerationThread.start();
}
```

Referencing HardwareMap in Robot Handler Classes with Parameterization

Constructor Method for Shooter:

```
public Shooter(HardwareMap aHardwareMap) {
    shooterController = aHardwareMap.dcMotorController.get("particleShooter");
    shooterLeft = aHardwareMap.dcMotor.get("shooterLeft");
    shooterRight = aHardwareMap.dcMotor.get("shooterRight");
    shooterKam = aHardwareMap.servo.get("shooterFlicker");

    shooterLeft.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
    shooterRight.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
    shooterRight.setDirection(REVERSE);

    initializePositions();
}
```

Constructor Method for Sweeper:

```
public Sweeper(HardwareMap aHardwareMap) {
    sweeperAlignmentOds = aHardwareMap.opticalDistanceSensor.get("particleSweeperOds");
    sweeperController = aHardwareMap.dcMotorController.get("particleSweeperController");
    sweeperMotor = aHardwareMap.dcMotor.get("particleSweeper");

    bristleReflectivityMax = 0.0;
    bristleReflectivityMin = 1.0;
    bristleReflectivityStateMachineFlow = 0;
    bristleReflectivityThreshold = 0.5;
}
```

Constructor Method for Beacon Actuator Bar:

```
public BeaconActuatorBar(HardwareMap aHardwareMap) {
    colorSensorLeft = new ColorSensorBeacon(aHardwareMap.colorSensor.get("beaconColorLeft"),
I2C_ADDRESS_LEFT);
    colorSensorRight = new ColorSensorBeacon(aHardwareMap.colorSensor.get("beaconColorRight"),
I2C_ADDRESS_RIGHT);
}
```

Disabling Teleop Shooting when Motors Below Target RPM

Code Description and Considerations:

- To eliminate the risk of shooting a ball before the pitching wheels reach their target RPM, this code only allows for the shooter servo to be actuated unless the wheels are at their target RPM.
- There is a manual override using the bumpers to allow shooting if there were an encoder failure.
- The RPM is determined using the same calculations for the RPM telemetry display, as outlined on Page ES-36.

Modified Class File Methods:

```
@Override
public void loop() {
    //Note that axes on analog sticks are reversed so that up is positive rather than negative.

    //Shooter flicker.
    if(gamepad2.a && (Math.abs(shooterRpmLeft) > TARGET_RPM_THRESHOLD_LEFT &&
          Math.abs(shooterRpmRight) > TARGET_RPM_THRESHOLD_RIGHT)) shooter.setPositionShooting();
    else if(gamepad2.a && isManualOverrideEnabled(gamepad2)) shooter.setPositionShooting();
    else shooter.setPositionLoading();

}

private boolean isSlowDriveActivated(Gamepad aGamepad) {
    return (aGamepad.left_trigger > STICK_DIGITAL_THRESHOLD ||
                    aGamepad.right_trigger > STICK_DIGITAL_THRESHOLD);
}
```

On/Off Toggle for Sweeper Alignment

Code Description and Considerations:

- After a few rounds of driving practice, our drive team realized that the automatic sweeper alignment could be detrimental as it would allow us to either accidentally sweep in opponent particles or push our particles out of the way.
- To remedy this, the alignment function can now be toggled on and off via the joystick. The status of the alignment system is displayed on the driver station phone via telemetry.

Modified Class File Methods:

```
private int sweeperAlignmentToggleStateMachineFlow;
private boolean sweeperAutoAlignmentEnabled;

@Override
public void loop() {
    //Sweeper motor.
    switch(sweeperStateMachineFlow) {
        case 0:
            sweeper.stop();
            if(Math.abs(gamepad2.right_stick_y) > STICK_DIGITAL_THRESHOLD) sweeperStateMachineFlow++;
            break;
        case 1:
            if(gamepad2.left_trigger > STICK_DIGITAL_THRESHOLD || gamepad2.right_trigger >
STICK_DIGITAL_THRESHOLD) sweeperScalarCurrent = SWEEPER_SCALAR_SLOW;
            else sweeperScalarCurrent = 1.0;
            if(Math.abs(gamepad2.right_stick_y) > STICK_DIGITAL_THRESHOLD) sweeper.setPower(-
gamepad2.right_stick_y * sweeperScalarCurrent);
            else sweeperStateMachineFlow++;
            break;
        case 2:
            if(sweeper.isAligned() || Math.abs(gamepad2.right_stick_y) > STICK_DIGITAL_THRESHOLD ||
isManualOverrideEnabled(gamepad2) || !sweeperAutoAlignmentEnabled) sweeperStateMachineFlow++;
            if(Math.abs(gamepad2.right_stick_y) > STICK_DIGITAL_THRESHOLD ||
isManualOverrideEnabled(gamepad2)) sweeper.restartAlignmentStateMachine();
            sweeper.sweepInSlow();
            break;
        default:
            sweeperStateMachineFlow = 0;
            break;
    }
    //Sweeper alignment system toggle.
    switch(sweeperAlignmentToggleStateMachineFlow) {
        case 0:
            if(gamepad2.x) sweeperAlignmentToggleStateMachineFlow++;
            break;
        case 1:
            if(!gamepad2.x) sweeperAlignmentToggleStateMachineFlow++;
            break;
        case 2:
            sweeperAutoAlignmentEnabled = !sweeperAutoAlignmentEnabled;
            sweeperAlignmentToggleStateMachineFlow = 0;
            break;
    }
    //Driver station telemetry.
    telemetry.addData("Sweeper", getTelemetryEnabledDisabledDisplay("Auto-Align",
sweeperAutoAlignmentEnabled));
}

private String getTelemetryEnabledDisabledDisplay(String aLabel, boolean aToggle) {
    if(aToggle) return aLabel + " Enabled";
    else return aLabel + " Disabled";
}
```