bovlb / crescendo_shooter.java Secret

Last active 2 months ago • Report abuse

Code Revisions 52

Some ideas for our Crescendo shooter

crescendo_shooter.java

```
// There's been a lot of discussion on Chief Delphi about best practices for Command-based program
// Here is one thread that I found particularly interesting:
// https://www.chiefdelphi.com/t/command-based-best-practices-for-2025-community-feedback/465602?u
// I have been thinking about how we could use that pattern in our codebase.
//
// I chose to take as an example the shooter, which has three motors, two sensors, and multiple st
// Controlling the shooter has been a bit of a challenge for us, so I thought it would be a good e
//
// Read the thread linked above for more context, but what I tried to do here is:
// - Subsystems expose their state only as Triggers.
// - Subsystems expose their controls only as Command factories.
// - Commands have a single purpose and a single subsystem.
// - Triggers define when commands should run.
// - Avoid tight coupling between subsystems.
// - A simple system that works equally well in auto and teleop.
// - Use Sendable to expose the state of the robot to the dashboard.
// - Interfaces are described in terms of problem space and intention, not hardware.
// - We don't incur additional costs or inconsistency for the sake of flexibility.
//
// This is just a collection of code fragments, not a complete implementation,
// but I hope it gives you an idea of what I'm thinking.

// General background on our Crescendo in-season shooter:
// - We have a robot with a combined intake/shooter machanism.
// - The shooter has two manipulators (sets of rollers), one set at each end.
// - Rollers can be run in either direction to intake, feed, or shoot the game piece.
// - The shooter intakes from the front side only, but can shoot in either direction.
// - The shooter has a pivot that can be used for both floor intake and aiming.
// - The shooter has two sensors, one at each end, that detect the presence
//   of a game piece within the mechanism.
// - When holding a game piece and shooting, one manipulator acts as a holder/feeder,
//   and the other acts as a shooter. The manipulators (and beam-break sensors) swap
//   roles depending on the shooting direction.


// ------------
// Shooter.java
```

```
// ------------

// This subsystem provides access to the two beam-break sensors.
// It also controls the mode of the shooter, which can be idle, intaking, or shooting.
// The shooter can also be set to run forwards or backwards.
// The shooter also has emergent internal state, such as whether it has a game piece.
// These emergent states are deliberately separate from the mode, which indicates intention.
// This subsystem controls no motors.

import java.util.function.BooleanSupplier;

class Shooter extends SubsystemBase {
    // ----
    // MODE
    // ----

    private enum ShooterMode {
        IDLE, // empty or holding, possibly feeding, revving, or aiming
        INTAKING,
        SHOOTING
        // TODO: Add AMP, EJECTING
    }

    private ShooterMode m_mode = ShooterMode.IDLE;

    // Add triggers for each state; this is the only read access to the mode outside the subsystem
    public final Trigger isIdle = new Trigger(() -> m_mode == ShooterMode.IDLE);
    public final Trigger isIntaking = new Trigger(() -> m_mode == ShooterMode.INTAKING);
    public final Trigger isShooting = new Trigger(() -> m_mode == ShooterMode.SHOOTING);

    // This is the internal command factory for mode commands
    private Command setState(ShooterMode mode) {
        return runOnce(() -> {
            m_mode = mode;
        }).setName(mode.toString());
    }

    // Public command factories for each state; this is the only write access to the mode outside
    public Command stop() { return setState(ShooterMode.IDLE); }
    public Command startIntaking() { return setState(ShooterMode.INTAKING); }
    public Command startShooting() { return setState(ShooterMode.SHOOTING); }

    // ---------
    // DIRECTION
    // ---------

    private enum ShooterDirection {
        FORWARDS,
        BACKWARDS
```

```
    }

    private ShooterDirection m_direction = ShooterDirection.FORWARDS;
    // Add a trigger for one direction; this is the only read access to the direction outside the
    // This trigger is used to determine which manipulator is the shooter and which is the holder/
    // We don't need isBackwards because we can just negate isForwards; it's boolean.
    public final Trigger isForwards = new Trigger(() -> m_direction == ShooterDirection.FORWARDS);

    // This is the internal command factory for direction commands
    private Command setDirection(ShooterDirection direction) {
        return runOnce(() -> {
            m_direction = direction;
        }).setName(direction.toString());
    }

    // Public command factories for each direction and toggling; this is the only write access to

    public Command setForwards() {
        return setDirection(ShooterDirection.FORWARDS);
    }

    public Command setBackwards() {
        return setDirection(ShooterDirection.BACKWARDS);
    }

    public Command toggleDirection() {
        return Commands.either(
            setDirection(ShooterDirection.BACKWARDS),
            setDirection(ShooterDirection.FORWARDS),
            isForwards).withName("Toggle Direction");
    }

    // -------
    // SENSORS
    // -------

    // Wrap our beam-break sensors, explaining what they mean

    // These variables cache the sensor values so that we can use them in triggers
    // without having to call the sensor multiple times,
    // and ensuring we get the same value each time within an iteration.
    private boolean m_gamePieceInFront = false;
    private boolean m_gamePieceInBack = false;

    @Override
    public void periodic() {
        // Cache inputs here
        m_gamePieceInFront = !m_frontSensor.get();
        m_gamePieceInBack = !m_backSensor.get();
```

```
136     }
137
138     // This trigger will be true when there is a game piece anywhere in the shooter,
139     // and stay true for a short time after no game piece is detected.
140     // This delay allows the game piece to fully leave the shooter before we power down and stop a
141     public final Trigger hasGamePiece = new Trigger(() -> m_gamePieceInBack || m_gamePieceInFront)
142         .debounce(k_hasGamePieceDelay, Debouncer.DebounceType.kFalling);
143
144     // Combine sensors with direction to make triggers beased on their current role
145
146     public final Trigger gamePieceInHolder = new Trigger(
147         () -> (isForwards.getAsBoolean() ? m_gamePieceInBack : m_gamePieceInFront));
148
149     public final Trigger gamePieceInShooter = new Trigger(
150         () -> (isForwards.getAsBoolean() ? m_gamePieceInFront : m_gamePieceInBack));
151
152     // ----------------
153     // COMPLEX TRIGGERS
154     // ----------------
155
156     // Do we need to move the game piece away from the shooter?
157     // Only true when we're idle (not trying to shoot or intake),
158     // and the game piece is in the shooter.
159     public final Trigger shouldFeed = isIdle.and(gamePieceInShooter);
160
161     // Should we be aiming the pivot?
162     // Only true when we're idle or shooting, and we have a game piece.
163     public final Trigger shouldAim = isIdle.or(isShooting).and(hasGamePiece);
164
165     // Are we clear to shoot by spinning up the holder?
166     // Only true when we're shooting, and we have a game piece
167     // This trigger and shouldFeed cannot be true at the same time.
168     public final Trigger shouldRevHolder = isShooting.and(hasGamePiece)
169
170     // Should we spin up the shooter?
171     //
172     // When we're idle, have a game piece, but aren't feeding it
173     // or in the same circumstances when want to rev the holder.
174     // This trigger and shouldFeed cannot be true at the same time.
175     public final Trigger shouldRevShooter =
176         isIdle.and(hasGamePiece).and(shouldFeed.negate())
177         .or(shouldRevHolder);
178
179     @Override
180     public void initSendable(SendableBuilder builder) {
181         super.initSendable(builder);
182         builder.addStringProperty("State", m_mode::toString, null);
183         builder.addStringProperty("Direction", m_direction::toString, null);
184         builder.addBooleanProperty("Game Piece in Back?", () -> m_gamePieceInBack, null);
```

```
        builder.addBooleanProperty("Game Piece in Front?", () -> m_gamePieceInFront, null);
        builder.addBooleanProperty("Has Game Piece?", hasGamePiece, null);
        builder.addBooleanProperty("Game Piece in Holder?", gamePieceInHolder, null);
        builder.addBooleanProperty("Game Piece in Shooter?", gamePieceInShooter, null);
        builder.addBooleanProperty("Should Feed?", shouldFeed, null);
        builder.addBooleanProperty("Should Aim?", shouldAim, null);
        builder.addBooleanProperty("Should Rev Shooter?", shouldRevShooter, null);
        builder.addBooleanProperty("Should Rev Holder?", shouldRevHolder, null);
    }
}

// ----------------
// Manipulator.java
// ----------------

// Controls a pair of rollers at one end of the shooting mechanism.
// We have two instances of this subsystem, one for each end.
// The manipulator can be set to intake, feed, or shoot the game piece.

class Manipulator extends SubsystemBase {
    // Constants for the manipulator
    private enum ManipulatorState {
        STOPPED(0, false),
        INTAKING(k_intakeVelocity, false),
        SHOOTING(k_shootingVelocity, true),
        FEEDING(k_feedVelocity, true);

        private final double m_velocity;
        private final boolean m_reverses;

        ManipulatorState(double velocity, boolean reverses) {
            m_velocity = velocity;
            m_reverses = reverses;
        }

        public double velocity(BooleanSupplier forwards) {
            if (m_reverses) {
                return forwards.getAsBoolean() ? -m_velocity : m_velocity;
            } else {
                return m_velocity;
            }
        }
    }

    // Internal command factory for setting the velocity
    private Command setVelocity(ManipulatorState state) {
        return run(() -> {
            m_setpoint = state.velocity(m_forwards);
            // Use on-controller PID to control the velocity here
```

```
            m_pid.setReference(m_setpoint);
        }).setName(state.name());
    }

    // Public command factories for each velocity
    // These are the only write access to the velocity outside the subsystem
    public Command stop() { return setVelocity(ManipulatorState.STOPPED); }
    public Command setIntaking() { return setVelocity(ManipulatorState.INTAKING); }
    public Command setShooting() { return setVelocity(ManipulatorState.SHOOTING); }
    public Command setFeeding() { return setVelocity(ManipulatorState.FEEDING); }

    // Is the manipulator near the desired speed?
    private boolean isReady() {
        return MathUtil.isNear(m_velocity, m_setpoint, k_tolerance);
    }

    // Add a trigger for isReady;
    // Debounce it so that it doesn't flicker when the game piece hits the rollers
    // This is the only read access to the velocity outside the subsystem.
    // TODO: Coonsider caching.
    public final Trigger isReady = new Trigger(this::isReady).debounce(k_isReadyDelay, Debouncer.D

    @Override
    public void initSendable(SendableBuilder builder) {
        super.initSendable(builder);
        builder.addDoubleProperty("Velocity (m/s)", () -> m_velocity, null);
        builder.addDoubleProperty("Setpoint (m/s)", () -> m_setpoint, null);
        builder.addBooleanProperty("Is Ready?", isReady, null);
    }

    private final BooleanSupplier m_forwards;
    private double m_velocity = 0;

    // Constructor is configured based on whether this is the front or back manipulator
    // All other details are determined internally.
    Manipulator(boolean isFront, BooleanSupplier forwards) {
        m_forwards = forwards;
        // Set up motors, encoders, and PID controllers here
        setName(isFront ? "Front Manipulator" : "Back Manipulator");
        // No longer need to know which manipulator we are.
    }

    @Override
    public void periodic() {
        // Cache inputs here
        m_velocity = m_encoder.getVelocity();
    }
}
```

```
// ----------
// Pivot.java
// ----------

// The pivot subsystem controls the angle of the shooter.
// The pivot can be set to various angles, such as intaking, home, or aiming.

class Pivot extends SubsystemBase {
    private final DoubleSupplier m_distance;
    private final BooleanSupplier m_forwards;

    public Pivot(DoubleSupplier distance, BooleanSupplier forwards) {
        m_distance = distance;
        m_forwards = forwards;
        // Set up motors, encoders, and PID controllers here
    }

    // Internal command factory for setting the angle (in radians)
    private Command setAngle(DoubleSupplier angle) {
        return run(() -> {
            m_setpoint = angle.getAsDouble();
            // Use m_setpoint to control the pivot angle here
            m_feedback = m_feedbackController.calculate(m_position, m_setpoint);
            m_feedforward = m_feedforwardController.calculate(m_position, m_setpoint);
            m_power = MathUtil.clamp(feedforward + feedback, -1.0, 1.0);
            m_motor.setPower(m_power);
        });
    }

    // Public command factories for each angle; these are the only write access to the angle outsi

    // Bring the front manipulator close to the floor
    public Command setIntaking() {
        return setAngle(() -> k_intakeAngle).withName("Intaking");
    }

    // Go to the home (fully retracted) position
    public Command setHome() {
        return setAngle(() -> k_homeAngle).withName("Home");
    }

    // Set the pivot angle based on the distance to the target and the shooting direction
    // Aiming is done using a lookup table.
    // The lookup table is different for forwards and backwards shooting.
    public Command setAiming() {
        return setAngle(() -> {
            boolean forwards = m_forwards.getAsBoolean();
            InterpolatingDoubleTreeMap angles = forwards ? m_forwardsAngles : m_backwardsAngles;
            double distance = m_distance.getAsDouble(); // distance to speaker in metres
```

```
            double angle = angles.get(distance);
            return angle;
        }).withName("Aiming");
    }

    // Is the pivot near the desired angle and stationary?
    private boolean isReady() {
        return MathUtil.isNear(m_position, m_setpoint, k_positionTolerance)
            && MathUtil.isNear(m_velocity, 0, k_velocityTolerance);
    }

    // Add a trigger for isReady; debounce it so that it doesn't flicker while we're shooting
    // This is the only read access to the angle and angular velocity outside the subsystem.
    // TODO: Coonsider caching.
    public final Trigger isReady = new Trigger(this::isReady).debounce(k_isReadyDelay, Debouncer.D

    @Override
    public void periodic() {
        // Cache inputs here
        m_position = m_encoder.getPosition();
        m_velocity = m_encoder.getVelocity();
    }

    @Override
    public void initSendable(SendableBuilder builder) {
        super.initSendable(builder);
        builder.addDoubleProperty("Setpoint (degrees)",
                () -> Math.toDegrees(m_setpoint), null);
        builder.addDoubleProperty("Angle (degrees)",
                () -> Math.toDegrees(m_position), null);
        builder.addDoubleProperty("Angular Velocity (degrees/s)",
                () -> Math.toDegrees(m_velocity), null);
        builder.addDoubleProperty("Feedforward", () -> m_feedforward, null);
        builder.addDoubleProperty("Feedback", () -> m_feedback, null);
        builder.addDoubleProperty("Power", () -> m_power, null);
        builder.addBooleanProperty("Is Ready?", isReady, null);
        addChild(m_feedbackController);
    }
}

// --------------------
// RobotContainer.java
// --------------------

// The RobotContainer is the glue that holds everything together.
// It is responsible for wiring up the subsystems, triggers, and commands.
// It also exposes the state of the robot to the dashboard.
// In particular, it is responsible for tying together information from multiple subsystems
// and for commanding subsystems to act in concert.
```

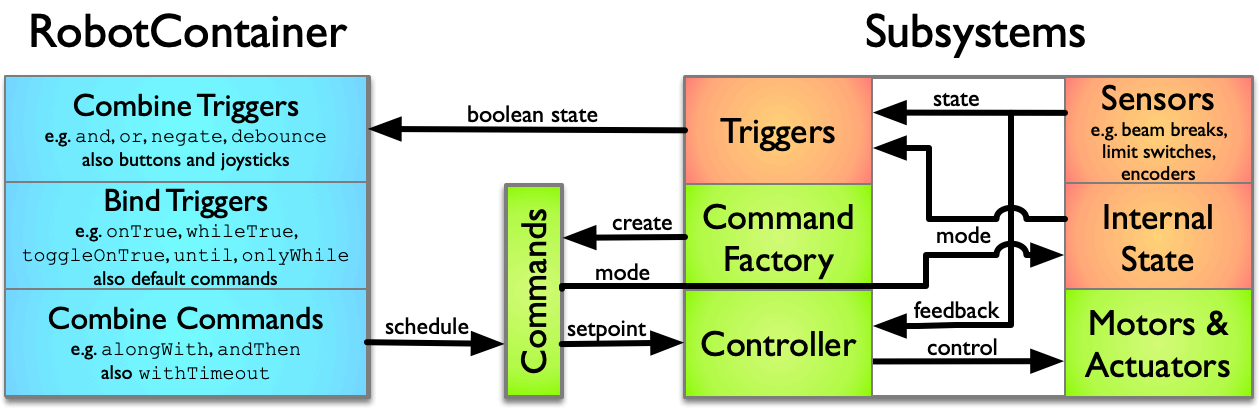
```
// In this way, the subsystems are not tightly coupled to each other, and commands are simple.

public class RobotContainer implements Sendable {
    // How far are we fron the target speaker?
    private double getSpeakerDistance() {
        Pose2d speaker = m_apriltags.getTagPose(m_alliance == Alliance.RED ? 4 : 7).toPose2d();
        return m_poseEstimator.getEstimatedPosition().getTranslation().getDistance(speaker.getTran
    }

    // How far is the speaker from the robot?
    // Cache answer in a variable for consistency and efficiency.
    private double m_speakerDistance = 0; // metres
    private boolean m_forwards = true;

    // Debounce the falling edge to avoid flickering when we're on the edge of the range
    // and to allow us to complete the shot before we stop shooting.
    // TODO: Does the range depend on whether we're shooting forwards or backwards?
    private final Trigger isInShootingRange =
        newTrigger(() -> (m_speakerDistance < k_shootingRange))
            .debounce(k_isInShootingRangeDelay, Debouncer.DebounceType.kFalling);

    // Subsystems have no coupling with each other; they are only coupled to the RobotContainer
    private final Manipulator m_frontManipulator = new Manipulator(true, ()->m_forwards);
    private final Manipulator m_backManipulator = new Manipulator(false, ()->m_forwards);
    private final Pivot m_pivot = new Pivot(()->m_speakerDistance, ()->m_forwards);
    private final Shooter m_shooter = new Shooter();

    // Cache the manipulators based on the shooting direction
    private Manipulator m_shooterManipulator = m_frontManipulator;
    private Manipulator m_holderManipulator = m_backManipulator;

    // Called from Robot.robotPeriodic before CommandScheduler.run()
    public void updateInputCaches() {
        m_speakerDistance = getSpeakerDistance();
        m_isForwards = m_shooter.isForwards.getAsBoolean();
        m_shooterManipulator = m_isForwards ? m_frontManipulator : m_backManipulator;
        m_holderManipulator = m_isForwards ? m_backManipulator : m_frontManipulator;
    }

    // Bind buttons to commands here; teleop only
    private void configureButtonBindings() {
        // This button controls whether we are intaking
        // We may stop early if we detect a game piece
        // Debounce to stop flickering if game controller is noisy
        intake_button.debounce(k_buttonDebounce)
            .whileTrue(m_shooter.startIntaking()
                .finallyDo(m_shooter.stop()));

        // This button controls whether we are shooting
```

```
430         // Note that releasing the button will not stop the shooter
431         // Instead we keep shooting until the game piece is out
432         // TODO: Consider adding a timeout to stop shooting if we jam.
433         shooting_button.onTrue(m_shooter.startShooting());
434 
435         // This button controls the shooting direction, toggling between forwards and backwards
436         // TODO: Consider disabling this button while shooting
437         shooting_direction_button.onTrue(m_shooter.toggleDirection());
438     }
439 
440     // Bind other triggers to commands here; both auto and teleop
441     private void configureTriggerBindings() {
442         // While in intaking mode, run intaking commands
443         m_shooter.isIntaking
444             .whileTrue(m_frontManipulator.setIntaking()
445                 .alongWith(m_backManipulator.setIntaking()
446                 .alongWith(m_pivot.setIntaking())).withName("Intaking"));
447 
448         // In both auto and teleop, we want to stop intaking when we have a gamepiece.
449         // This is because we want to retract the intake as soon as possible
450         // It doesn't mean that we stop feeding the game piece.
451         // Short delay to ensure the game piece is fully inside.
452         m_shooter.isIntaking
453             .and(m_shooter.hasGamePiece)
454             .debounce(k_intakingDelay, Debouncer.DebounceType.kRising)
455             .onTrue(m_shooter.stop());
456 
457         // If the game piece needs to be fed, run feeding commands on both rollers
458         // This moves the game piece away from the shooter and into the holder
459         m_shooter.shouldFeed
460             .whileTrue(
461                 m_frontManipulator.setFeeding()
462                 .alongWith(m_backManipulator.setFeeding()).withName("Feeding"));
463 
464         // In both auto and teleop, if we're aimable and in range, aim the pivot
465         m_shooter.shouldAim
466             .and(isInShootingRange)
467             .whileTrue(m_pivot.setAiming());
468 
469         // In both auto and teleop, spin up the shooter when we're ready to rev and in-range
470         m_shooter.shouldRevShooter
471             .and(isInShootingRange)
472             .whileTrue(m_shooterManipulator.setShooting());
473 
474         // When everything is ready to shoot, spin up the holder/feeder as well
475         // This will push the game piece into the shooter
476         m_shooter.shouldRevHolder
477             .and(m_pivot.isReady)
478             .and(m_shooterManipulator.isReady)
```

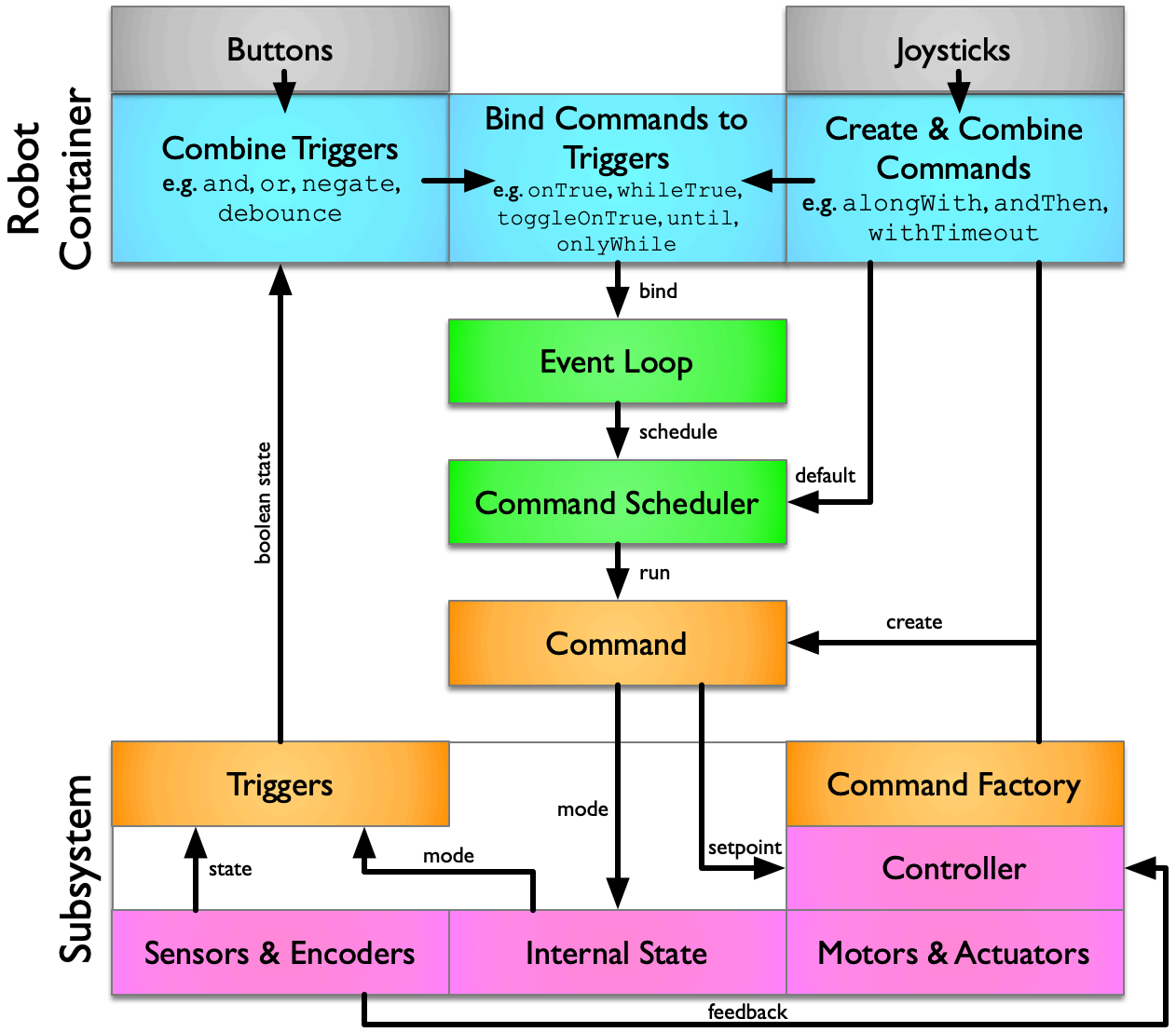
```
            .whileTrue(m_holderManipulator.setShooting());

        // In both auto and teleop, stop shooting when we don't have a gamepiece or we go out of r
        m_shooter.isShooting
            .and(m_shooter.hasGamepiece.negate())
            .or(isInShootingRange.negate())
            .onTrue(m_shooter.stop());

        // Reset to known state when we're disabled
        // This ensures we enter auto or teleop in a consistent state, no matter what state we wer
        RobotModeTriggers.disabled().onTrue(
            m_shooter.stop()
            .andThen(m_shooter.setForwards())
            .runsWhenDisabled().withName("Reset Shooter"));
    }

    // Here we define some named commands that we can use in auto
    private void configureNamedCommands() {
        // These commands only use the Shooter subsystem and,
        // while they do indirectly affect the manipulators and the pivot,
        // they do so through triggers that depend on the shooter internal state.
        // Hence they do not interfere with the default commands and triggers of the other subsyst
        // We use the asProxy() method to ensure that the command is not interrupted by other comm
        // In this way, default commands and triggers will work normally during autonomous.

        NamedCommands.registerCommand("Intake",
            m_shooter.startIntaking().asProxy()
                .until(m_shooter.hasGamePiece)
                .withTimeout(k_intakingTimeout)
                .withName("Intake"));

        NamedCommands.registerCommand("Shoot Forwards",
            m_shooter.setForwards().asProxy()
                .withName("Shoot Forwards"));

        NamedCommands.registerCommand("Shoot Backwards",
            m_shooter.setBackwards().asProxy()
                .withName("Shoot Backwards"));

        NamedCommands.registerCommand("Shoot",
            m_shooter.startShooting().asProxy()
                .until(m_shooter.isIdle)
                .withName("Shoot"));
    }

    private void configureDashboard() {
        // Add Sendable objects to the dashboard
        SendableRegistry.addLW(this);
    }
```

```
528
529     RobotContainer() {
530         // Default commands
531         m_frontManipulator.setDefaultCommand(m_frontManipulator.stop());
532         m_backManipulator.setDefaultCommand(m_backManipulator.stop());
533         m_pivot.setDefaultCommand(m_pivot.setHome());
534
535         configureButtonBindings();
536         configureTriggerBindings();
537         configureNamedCommands();
538         configureDashboard();
539     }
540
541     @Override
542     public void initSendable(SendableBuilder builder) {
543         builder.addBooleanProperty("In Shooting Range?", isInShootingRange, null);
544         builder.addDoubleProperty("Speaker Distance (m)", () -> m_speakerDistance, null);
545     }
```

bovlb commented on Sep 15, 2024 — Author



bovlb commented on Sep 16, 2024 — Author


Buttons
Joysticks
Robot Container
Combine Triggers
e.g. and, or, negate, debounce
Bind Commands to Triggers
e.g. onTrue, whileTrue, toggleOnTrue, until, onlyWhile
Create & Combine Commands
e.g. alongWith, andThen, withTimeout
bind
Event Loop
schedule
Command Scheduler
default
run
Command
create
boolean state
Subsystem
Triggers
Command Factory
mode
setpoint
state
mode
Controller
Sensors & Encoders
Internal State
Motors & Actuators
feedback