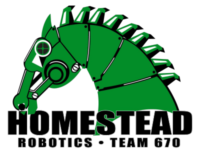
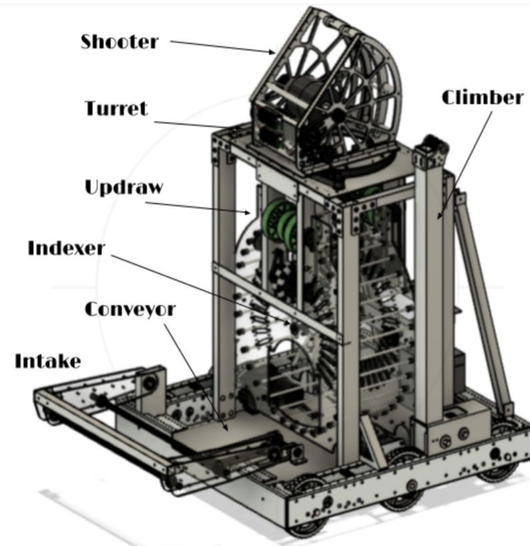
Background - Game & Our Robot

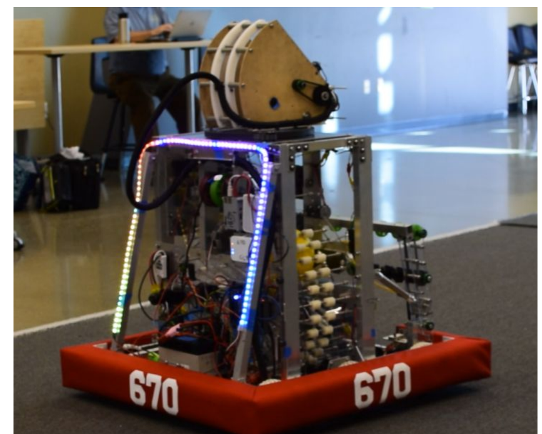


The 2020 FRC game, Infinite Recharge, involves two alliances of three teams each, with each team controlling one robot. The robots perform tasks like shooting Power Cells (balls) into different goals and climbing at the end of a match, to earn points for the alliance. Also, the robot must follow height and weight limits, and cannot control more than 5 balls at a time.

Our goal was to build a competitive robot, capable of winning the FIRST Championship. To maximize our rank from qualification matches, we needed to have a fast cycle time when acquiring and scoring power cells.



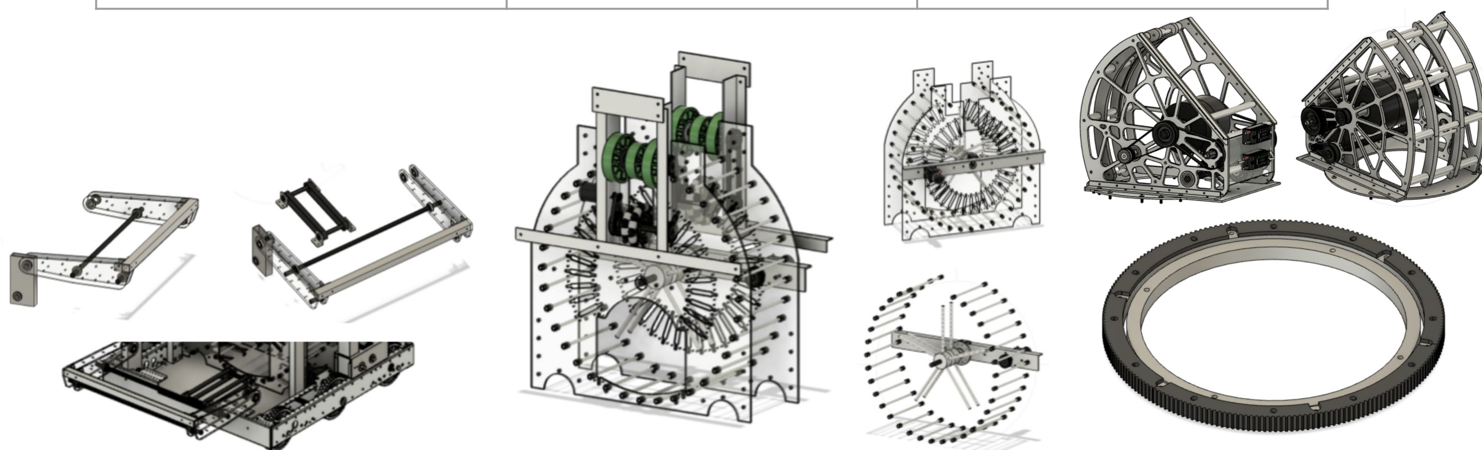
For that reason, we chose to acquire and store up to the maximum of 5 balls at a time using our intake and indexer (vertical revolver), then align & shoot quickly using our uptake, turret, and shooter.



Subsystem Highlights

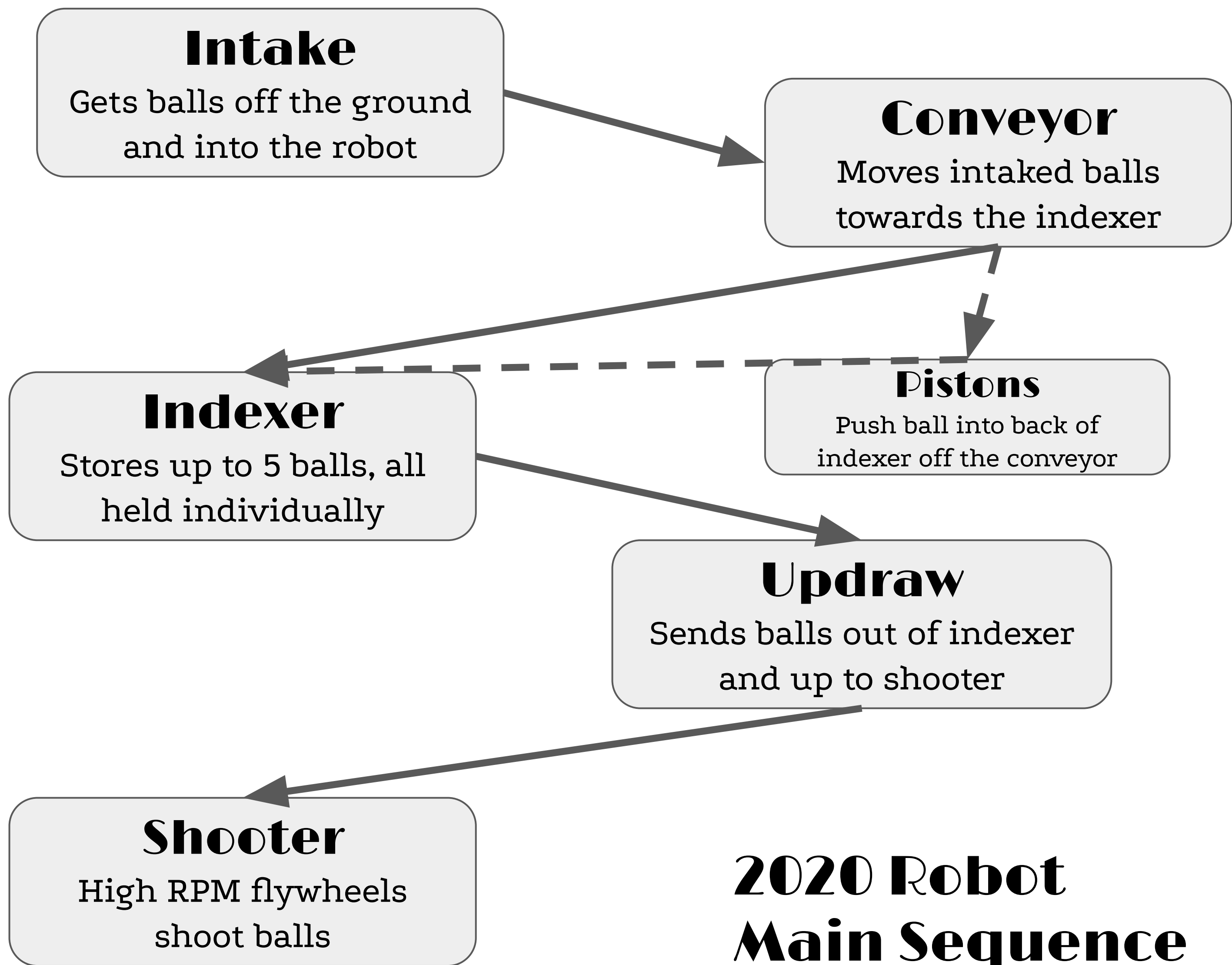
A number of interrelated subsystems on the robot are constantly communicating and are dependent on each other to complete tasks successfully. For example, the intake, conveyor and indexer work together to acquire a ball and store it in one of the chambers of the indexer. In order to function, each of our robot's subsystems has various motors, motor controllers and additional sensors, which need to be handled.

Subsystem	**Motors, Motor Controllers**	**Sensors**
Drivebase	4 Motors, 4 Motor Controllers	NavX(9 axis navigation sensor), Integrated encoder
Intake	2 Motors, 2 Motor Controllers	Current Sensor
Indexer + Updraw	3 Motors, 2 Motor Controllers	Time of Flight Sensor, Through Bore encoder, Current Sensor Integrated encoder
Turret	1 Motor, 1 Motor Controller	Hall effect sensors, Integrated encoder
Shooter	2 Motors, 2 Motor Controllers	Integrated encoder
Climber	1 Motor, 1 Motor Controller	Integrated encoder, Current Sensor



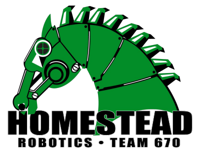
HOMESTEAD
ROBOTICS • TEAM 670

The Problem:
Dealing with complex systems and sequences



These are the steps that need to work together in order to get a ball from the ground to the shooter. Before we run the sequences of actions to move balls, checking to see if the systems are functioning as they should would help prevent the situation where an issue jams up the whole pathway and breaks more things as it tries to continue executing a task. **As our systems got more complex, it would be really helpful to have something which could automatically manage checking the state/health of subsystems and handling whether an action should be done or not.** This way, if a sensor was disconnected or a motor controller malfunctioned, the robot code as a whole wouldn't completely crash, and the system would prevent actions associated with that missing element from being scheduled to happen.

Our Solution: MustangSystem



Robot actions, their necessary preconditions, and tracking subsystem states can be handled by doing a lot of if-else logic, but that can get really unwieldy. It is also complicated, difficult to maintain, and can lead to a lot of code repetition and confusion. The way we solved this problem was by creating MustangSystem, a framework which builds on the WPILibJ library (the standard control system library used by most FRC teams) and command based programming framework, with which commands, manipulating subsystems that meet their health requirements, represent robot actions that are run with the Scheduler.

MustangSubsystem

- Defines states for a part of the robot
- Ex: Red/ Yellow/ Green
- State of susbystem affects which commands can be run

Requires

Checks

MustangSystem

A codebase built on the WPiLibJ Library used to schedule commands which require certain subsystems at specific target states

MustangCommand Base

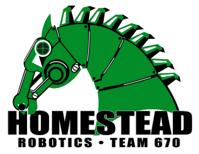
- Robot Action
- Defines the subsystems needed and what state they must be in for a command to run

MustangScheduler

- Expanded on WPiLib Scheduler
- Does not run a command if current state is not the same as target state

Checks, Schedules, Runs

Our Solution: MustangSubsystemBase

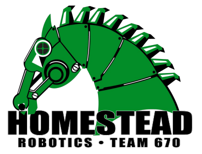


A MustangSubsystem can be in 1 of 4 possible Health states: GREEN, YELLOW, RED, or UNKNOWN (as in, we'll need to recalculate it). When we code each subsystem of the robot, we define what these states mean with regard to the specific system. For example, if 1 motor is disconnected on the drivebase, the drivebase could still function (although not optimally) and thus would be in "YELLOW" state, while if 1 motor was disconnected on the shooter, that would mean that the shooter could not function normally. Thus the shooter would be defined to be in "RED" state.

```java
public abstract class MustangSubsystemBase extends SubsystemBase {

    protected HealthState lastHealthState;
    private boolean failedLastTime = false;

    private static NetworkTableInstance instance = NetworkTableInstance.getDefault();
    private static NetworkTable table = instance.getTable("/SmartDashboard");

    /**
     * Creates a new MustangSubsystemBase. By default, the subsystem's initial
     * health state is UNKNOWN (ID 0).
     */
    public MustangSubsystemBase() {
        RobotContainer.addSubsystem(this);
        this.lastHealthState = HealthState.UNKNOWN;
    }
```

```java
  /**
   * Represents possible conditions a MustangSubsystemBase can be in. Each
   * MustangSubsystemBase should define what the States mean for it specifically.
   * The default state is UNKNOWN, before the subsystem is first "used".
   */
  public enum HealthState {
      UNKNOWN(0), GREEN(1), YELLOW(2), RED(3);

      private final int ID;

      HealthState(int id) {
          ID = id;
      }

      /**
       * Gets the ID of the state.
       */
      public int getId() {
          return ID;
      }

  }
```

Our Solution: MustangSubsystemBase

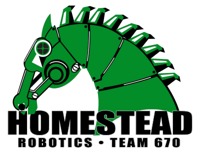



```java
/**
 * Calculates the current state of the subsystem.
 */
public abstract HealthState checkHealth();

public void initDefaultCommand(MustangCommand command) {
    CommandScheduler.getInstance().setDefaultCommand(this, (CommandBase) command);
}
```

We have the method checkHealth() which needs to be defined for every subsystem, and is responsible for specifying what the GREEN, YELLOW, and RED states mean for each part of the robot. Here's an example below from our Indexer subsystem, which has 2 motor controllers: a SparkMax and a TalonSRX, and a Time of Flight sensor, to measure the distance between the ball and the wall of the indexer. If all 3 of these parts are working and ready to go, the state of the indexer is GREEN (it's 'healthy') while if there's a problem with only the sensor, the indexer and updraw can still function, thus the state of the system is YELLOW. However if the motor that rotates the indexer, or the motors that run the updraw, have issues, then the indexer system is not usable, and it's in the RED state.

```java
@Override
public HealthState checkHealth() {
    // if either the rotator or updraw breaks, we can't use the indexer anymore.
    // Same deal if the indexer is jammed (but that's recoverable).
    if (isSparkMaxErrored(rotator) || isPhoenixControllerErrored(updraw) || indexerIsJammed) {
        return HealthState.RED;
    }
    // if the ToF sensor breaks but nothing else,
    // the next option would be manual control -- not a fatal issue
    if (indexerIntakeSensor == null || !indexerIntakeSensor.isHealthy()) {
        return HealthState.YELLOW;
    }
    return HealthState.GREEN;
}
```

Our Solution: MustangSubsystemBase



Initially, we don't know the state of subsystems, so right away we'll determine the subsystems' current states. Periodically, we will continue to update and recalculate the states so the system can run commands based on the state of the robot. Driver and operator will also be notified if there are any issues, so they will avoid using a subsystem if it is broken.

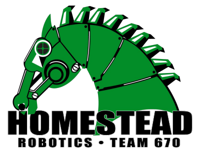
```java
@Override
public void periodic() {
    HealthState lastHealth = getHealth(false);
    if (lastHealth == HealthState.GREEN) {
        if (failedLastTime) {
            MustangNotifications
                    .notify("Health state for " + this.getName() + " is: " + lastHealth + ". Enabling Periodic");
            failedLastTime = false;
        }
        mustangPeriodic();
    } else {
        if (!failedLastTime) {
            MustangNotifications.reportError(
                    "Health state for " + this.getName() + " is: " + lastHealth + ". Disabling Periodic");
            failedLastTime = true;
        }
    }
}
```

```
subsystems
    Climber.java                 1
    ColorWheelSpinner.java
    Conveyor.java                1
    DriveBase.java               1
    Indexer.java                 8
    Intake.java                  1
    LEDSubsystem.java
    MustangSubsystemBase.java
    Shooter.java                 2
    SparkMaxRotatingSubsys...    1
    TunableSubsystem.java
    Turret.java
```

If someone tried to use the subsystem after its failure, the request would be denied and the command will not run.

All of the subsystems in our robot are represented by subtypes of this object. When we code robot actions with Commands, we call on these MustangSubsystemBase objects.

Our Solution: MustangCommand



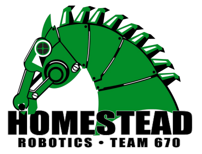
Now that we have a way of representing the parts of the robot, we can represent actions that those parts can do, in code. WPILib has the command based structure, where Commands representing robot actions are set to run by the Scheduler. For our system, an action can only run if the subsystems it uses are all in a usable state, with the requirements defined by the action. So, each MustangCommand keeps a list of the subsystems it needs and the minimum health state that each subsystem must be in in order to properly execute the action.

```
public interface MustangCommand{

    /**
     * @return A Map containing the minimum health condition for each subsystem that this Command requires
     */
    public Map<MustangSubsystemBase, MustangSubsystemBase.HealthState> getHealthRequirements();
```

For example, below we have a MustangCommand for rotating the indexer by 1 chamber. The only subsystem we need for doing this is the indexer, and we've previously defined the state of the indexer as GREEN if everything is working, and YELLOW if only the Time of Flight sensor (used when intaking) is not working. Since we can rotate the indexer without that sensor, the minimum health state of the indexer needed for this command to run without issues is YELLOW, and we add that to the map.

```
public class RotateToNextChamber extends CommandBase implements MustangCommand {

    private Indexer indexer;
    private Map<MustangSubsystemBase, HealthState> healthReqs;

    private boolean isForward;

    public RotateToNextChamber(Indexer indexer, boolean isForward) {
        this.isForward = isForward;
        this.indexer = indexer;
        addRequirements(indexer);
        healthReqs = new HashMap<MustangSubsystemBase, HealthState>();
        healthReqs.put(indexer, HealthState.YELLOW); // This Command works without Time of Flight sensor
    }
```

Our Solution: MustangCommand

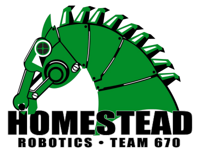


We can also represent more complex sequences of actions that use many subsystems together, by extending the CommandGroup classes from WPILib and still implementing MustangCommand. For example, here's the code for the sequence to intake a ball and move it into the bottom chamber of the indexer. To do this, we need the intake, conveyor belt, and indexer to all be in working order, with their health state being GREEN. So we add this condition to the map in the CommandGroup. Then we can go ahead and add the actions (MustangCommands) that make up this sequence in the constructor.

CommandGroups like this are how we represent our sequences in code for connecting and transitioning between the steps on the path the ball takes, from getting a ball from off the ground to shooting it out. We also use them in our autonomous routines.

```java
public class IntakeBallToIndexer extends SequentialCommandGroup implements MustangCommand {

    private Map<MustangSubsystemBase, HealthState> healthReqs;

    public IntakeBallToIndexer(Intake intake, Conveyor conveyor, Indexer indexer) {
        addRequirements(intake, conveyor, indexer);
        healthReqs = new HashMap<MustangSubsystemBase, HealthState>();
        healthReqs.put(intake, HealthState.GREEN);
        healthReqs.put(conveyor, HealthState.GREEN);
        healthReqs.put(indexer, HealthState.GREEN);
        if (!intake.isDeployed()) {
            addCommands(new DeployIntake(true, intake));
        }
        addCommands(
                new RotateToIntakePosition(indexer),
                new ParallelCommandGroup(
                    new RunIntake(false, intake),
                    new RunConveyor(false, conveyor)
                )
        );
    }

    @Override
    public Map<MustangSubsystemBase, HealthState> getHealthRequirements() {
        return healthReqs;
    }


}
```

Our Solution: MustangScheduler



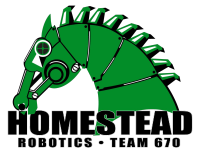
Now that we have our commands which have specific subsystems and their target states as requirements in order to run, we need something to run the commands. We use the MustangScheduler to schedule/run our commands which is slightly different from WPiLib's CommandScheduler. In addition to the CommandScheduler's functions, MustangScheduler rechecks the current health state for all the subsystems that are required to run a specific function. If any of the subsystems are not at the desired/target state, the command is not allowed to run and the user is notified

```
for (MustangCommand a_command : commands) {

    CommandBase m_command = (CommandBase) a_command;
    try {
        Map<MustangSubsystemBase, MustangSubsystemBase.HealthState> requirements = ((MustangCommand) (m_command))
                .getHealthRequirements();

        if (requirements != null) {
            for (MustangSubsystemBase s : requirements.keySet()) {
                MustangSubsystemBase.HealthState healthReq = requirements.get(s);
                if (s != null && healthReq != null) {
                    HealthState currentHealth = s.getHealth(false);
                    if (currentHealth.getId() > healthReq.getId()) {
                        MustangNotifications.reportWarning(
                                "%s not run because of health issue! Required health: %s, Actual health: %s",
                                m_command.getName(), healthReq, currentHealth);
                        return;
                    }
                }
            }
        }
        this.currentCommand = m_command;
        scheduler.schedule(currentCommand);
        Logger.consoleLog("Command scheduled: %s", this.currentCommand.getName());
    } finally {
        this.currentCommand = null;
    }
}
```

Finally, commands can be scheduled easily this way. This specific one is trying to schedule the ExtendClimer command with the climber subsystem as its requirement. Ultimately, this command extends the climber if the subsystem is healthy or at its target state.

```
private void extendClimber() {
    MustangScheduler.getInstance().schedule(new ExtendClimber(climber));
}
```

Conclusion



Summary

As our robots' systems became more complex, it was clear that something to keep track of the health of robot subsystems and determine whether actions could be run would be immensely helpful. Such a development would prevent issues that could happen if one part of the robot fails and affects other interconnected systems and the procedures that use them. To deal with this challenge, we created MustangSystem, a framework built on the preexisting WPILib Java library which checks the state of a subsystem before executing commands that rely on it, providing a safe way to manage and carry out robot actions.

Benefits

MustangSystem **helps us model the state of our robot** by tracking the health of its parts. It intelligently stops commands that cannot be safely run at the time, and allows for alternative actions to happen if defined. This system avoids the messiness of sprinkling if-else statements everywhere, and provides a solution which **makes writing organized and readable code easier**. With MustangSystem, **we get a good separation of concerns** between subsystems, commands and scheduler. It allows us to have **greater control over robot subsystems and actions** without completely replacing the base of WPILibJ, and the foundation it provides means that we will not need to worry about forgetting to implement checks everywhere.

Going Forward

With MustangSystem being a new innovation this year, there is still a lot of room for further development and improvement. For example, we wrote a SparkMaxRotatingSubsystem subclass which represented a mechanism with a rotator motor and SparkMax controller. In the future, more such subclasses based on MustangSubsystem could be written to represent other common types of robot mechanisms, to reduce repeating code and improve organization.