# Team Flame Complete Source Code (Jan. 24, 2026)

## Complete Source Code

```
"""
LEGO Robot Control System for FIRST LEGO League (FLL) team Flame #20318.

This module provides a comprehensive robot control framework built on Pybricks.

It includes:
- Robot base class for movement and sensor control
- Attachment system for modular mission-specific hardware
- Run system for executing different mission sequences based on color detection
- Runner class for coordinating mission execution

Hardware Configuration:
- Left motor: LEFT_MOTOR (clockwise)
- Right motor: RIGHT_MOTOR (counterclockwise)
- Color sensor: COLOR_SENSOR
- Left attachment motor: ATTACHMENT_LEFT
- Right attachment motor: ATTACHMENT_RIGHT

Units:
- Distance: millimeters (mm)
- Speed: millimeters per second (mm/s)
- Acceleration: millimeters per second squared (mm/s^2)

Naming schema:
- All runs are named by the mission they first complete, or their purpose.

TODO: Add attachment initialization and position reset with motor.run_until_stalled()
TODO: Break up Run sequences into smaller methods for easier debugging and maintenance
TODO: Fix detection with no color card reading and running black mission
"""

from pybricks.hubs import PrimeHub
from pybricks.pupdevices import Motor, ColorSensor
from pybricks.robotics import DriveBase
from pybricks.parameters import Port, Stop, Direction, Color, Icon
from pybricks.tools import wait, StopWatch

# ============================================================================
# COLOR CALIBRATION
# ============================================================================
# Color sensor HSV calibration values
# These override default color definitions for better detection accuracy
Color.GRAY = Color(h=180, s=11, v=24)
Color.WHITE = Color(h=40, s=3, v=81)
Color.YELLOW = Color(h=48, s=70, v=75)
Color.BLACK = Color(h=210, s=22, v=11)
Color.GREEN = Color(h=155, s=75, v=24)
Color.ORANGE = Color(h=2, s=81, v=76)
Color.LIGHT_GREEN = Color(h=85, s=56, v=46)
Color.BLUE = Color(h=213, s=77, v=31)

# Tuple of all detectable colors for the color sensor
DETECTABLE_COLORS = (
    Color.GRAY,
    Color.GREEN,
    Color.YELLOW,
    Color.WHITE,
    Color.BLACK,
    Color.ORANGE,
    Color.LIGHT_GREEN,
    Color.BLUE,
)
```

```python
# ============================================================================
# HARDWARE PORT ASSIGNMENTS
# ============================================================================
# Drive motor port assignments
LEFT_MOTOR = Port.A
RIGHT_MOTOR = Port.E

# Sensor port assignments
COLOR_SENSOR = Port.D

# Attachment motor port assignments
ATTACHMENT_LEFT = Port.F
ATTACHMENT_RIGHT = Port.B

class Robot(PrimeHub):
    """
    Main robot control class that manages all movement and sensor operations.

    This class extends PrimeHub and provides a high-level interface for:
    - Drive base movement (forward, backward, turning)
    - Color sensor reading
    - Speed and acceleration configuration

    The robot is initialized as a singleton in main() and used throughout
    the mission execution system.

    Args:
        wheel_diameter: Diameter of the wheels in millimeters
        axle_track: Distance between the centers of the two wheels in millimeters
    """

    def __init__(self, wheel_diameter: int, axle_track: int):
        super().__init__()

        # Initialize drive base with motors on configured ports
        # Left motor: clockwise direction
        # Right motor: counterclockwise direction
        self.drive_base = DriveBase(
            Motor(RIGHT_MOTOR, Direction.COUNTERCLOCKWISE),
            Motor(LEFT_MOTOR, Direction.CLOCKWISE),
            wheel_diameter,
            axle_track
        )
        self.drive_base.use_gyro(False)

        # Initialize color sensor on configured port
        self.color_sensor = ColorSensor(COLOR_SENSOR)

        # Configure detectable colors for the sensor
        self.color_sensor.detectable_colors(DETECTABLE_COLORS)

        # Perform startup checks and initialization
        self.ready_up()

    def ready_up(self):
        """
        Performs startup checks and initialization.

        Checks:
        - Battery voltage (must exceed minimum threshold)
        - IMU (gyroscope) readiness

        Raises:
            AssertionError: If battery voltage is too low
        """
        print(f"Battery: {self.battery.voltage()} mV ({(self.battery.voltage() / 8200) * 100}% charge)")
        assert self.battery.voltage() > 6000, "Battery <6000 mV (low charge)"

        # Wait for IMU to be ready before proceeding
        while not self.imu.ready():
            pass
```

```
        print("Robot is ready")

    def move_forward(self, distance: int):
        """
        Moves the robot forward a specified distance.

        Args:
            distance: Distance to move in millimeters (positive value)
        """
        self.drive_base.straight(distance)

    def move_backward(self, distance: int):
        """
        Moves the robot backward a specified distance.

        Args:
            distance: Distance to move in millimeters (positive value)
        """
        self.drive_base.straight(-distance, Stop.BRAKE)

    def turn_left(self, angle: int):
        """
        Turns the robot left (counterclockwise) by a specified angle.

        Args:
            angle: Angle to turn in degrees (positive value)
        """
        self.drive_base.turn(-angle, Stop.BRAKE)

    def turn_right(self, angle: int):
        """
        Turns the robot right (clockwise) by a specified angle.

        Args:
            angle: Angle to turn in degrees (positive value)
        """
        self.drive_base.turn(angle, Stop.BRAKE)

    def arc_left(self, radius: float, angle: float=None, distance: float=None):
        """
        Moves the robot in a left arc (counterclockwise curve).

        Args:
            radius: Radius of the arc in millimeters (positive value)
            angle: Angle of the arc in degrees (optional)
            distance: Distance along the arc in millimeters (optional)
        """
        self.drive_base.arc(radius * -1, angle, distance)

    def arc_right(self, radius: float, angle: float=None, distance: float=None):
        """
        Moves the robot in a right arc (clockwise curve).

        Args:
            radius: Radius of the arc in millimeters (positive value)
            angle: Angle of the arc in degrees (optional)
            distance: Distance along the arc in millimeters (optional)
        """
        self.drive_base.arc(radius, angle, distance)

    def configure_straight(self, speed=None, accel=None):
        """
        Configures speed and acceleration settings for straight movement.

        Args:
            speed: Speed in mm/s (None to keep current setting)
            accel: Acceleration in mm/s^2 (None to keep current setting)
        """
        if not (speed or accel):
            return
```

```
        self.drive_base.settings(speed, accel)

    def configure_turn(self, speed=None, accel=None):
        """
        Configures speed and acceleration settings for turning movement.

        Args:
            speed: Turn rate in degrees/s (None to keep current setting)
            accel: Turn acceleration in degrees/s^2 (None to keep current setting)
        """
        if not (speed or accel):
            return

        self.drive_base.settings(turn_rate=speed, turn_acceleration=accel)

    def read_sensor(self, read_hsv: bool=False):
        """
        Reads the color sensor and returns the detected color.

        Args:
            read_hsv: If True, returns HSV tuple instead of Color object

        Returns:
            Color object or HSV tuple (h, s, v) depending on read_hsv parameter
        """
        color = Color.NONE

        if read_hsv:
            color = self.color_sensor.hsv(surface=True)
        else:
            color = self.color_sensor.color(surface=True)

        return color

    def sensor_off(self):
        """Turns off the color sensor LED lights."""
        self.color_sensor.lights.off()

    def sensor_on(self):
        """Turns on the color sensor LED lights."""
        self.color_sensor.lights.on()

    def end_early(self):
        """
        Terminates the program execution early.

        Raises:
            SystemExit: Always raised to exit the program
        """
        raise SystemExit("Ended early")
#endregion

#region Attachment Base and Defined Attachments
class Attachment:
    """
    Base class for all robot attachments.

    All mission-specific attachments must inherit from this class.
    Subclasses define their motors in a class-level 'motors' dictionary
    using the format: {"motor_id": {"port": Port.X, ...}}

    The class automatically initializes motors from the configuration
    and provides methods for controlling them with stall detection.

    Attributes:
        motors: Dictionary mapping motor IDs to their configurations.
                Must include "port" key for each motor.
        robot: Reference to the Robot instance
    """

    motors = {
        # Use "id": {"param": value} syntax to define motors
```

```
        # Example: "arm": {"port": ATTACHMENT_LEFT, "reset_angle": False}
    }

    def _get_motor(self, motor: str) -> Motor:
        """
        Retrieves a Motor instance by its ID.

        Args:
            motor: String ID of the motor as defined in motors dictionary

        Returns:
            Motor instance for the specified motor ID

        Raises:
            KeyError: If the specified motor ID does not exist in the motors dictionary
        """
        return type(self).motors[motor]

    def __init__(self, robot: Robot):
        """
        Initializes the attachment with a robot reference.

        Args:
            robot: Robot instance to attach to
        """
        self.robot = robot
        self._init_motors()

    def _init_motors(self, duty_limit: int=20, initialization_speed: int=-200):
        """
        Initializes all motors defined in the motors dictionary.

        Converts motor configuration dictionaries into Motor instances.
        Each motor configuration must include a "port" parameter.

        Raises:
            KeyError: If a motor configuration is missing the "port" parameter
            TypeError: If motor configuration parameters are invalid

        TODO: Add configuration and initializaiton with motor.run_until_stalled().
        """
        for motor_id, configuration in self.motors.items():
            if "port" not in configuration:
                raise KeyError(f"Motor \"{motor_id}\" must specify a \"port\" parameter")
            try:
                self.motors[motor_id] = Motor(**configuration)
            except TypeError as exception:
                raise TypeError(f"Invalid motor configuration for \"{motor_id}\": {exception}")

    def target(self, motor: str, angle: int, speed: int=300):
        """
        Moves a motor to a specific target angle with stall detection.

        The motor runs to the target angle and stops if it stalls.
        This method blocks until the movement completes or stalls.

        Args:
            motor: Motor ID string
            angle: Target angle in degrees
            speed: Motor speed in degrees per second (default: 300)
        """
        motor = self._get_motor(motor)
        motor.run_target(speed, angle, wait=False)

        # Monitor for stall condition and stop if detected
        while not motor.done():
            if motor.stalled():
                print("[!] Stall detection triggered! Killing movement and continuing...")
                motor.brake()
                break

    def angle(self, motor: str, angle: int, speed: int=300):
```

```
        """
        Rotates a motor by a relative angle with stall detection.

        The motor rotates by the specified angle relative to its current position.
        Stops if the motor stalls (indicating obstruction).
        This method blocks until the movement completes or stalls.

        Args:
            motor: Motor ID string
            angle: Relative angle to rotate in degrees (positive or negative)
            speed: Motor speed in degrees per second (default: 300)
        """
        motor = self._get_motor(motor)
        motor.run_angle(speed, angle, wait=False)

        # Monitor for stall condition or excessive load
        while not motor.done():
            if motor.stalled():
                print("Motor stalled. Killing movement and continuing")
                motor.brake()
                break

    def reset(self, motor: str):
        """
        Resets a motor to its zero position.

        Args:
            motor: Motor ID string
        """
        self.target(motor, 0)

    def zero_out(self, motor: str):
        """
        Resets a motor's angle counter to zero without moving the motor.

        Args:
            motor: Motor ID string
        """
        self._get_motor(motor).reset_angle(0)

    def lock(self, motor: str):
        """
        Locks a motor in its current position using hold mode.

        Args:
            motor: Motor ID string
        """
        self._get_motor(motor).hold()

    def unlock(self, motor: str):
        """
        Unlocks a motor, allowing it to move freely.

        Args:
            motor: Motor ID string
        """
        self._get_motor(motor).stop()

class BrushAndReveal(Attachment):
    """
    Attachment for Surface Brushing and Map Reveal missions.

    Motors:
        - brush: Dropping attachment for brush retrieval
        - arm: Arm raising mechanism for map reveal attachment
    """
    motors = {
        "brush": {
            "port": ATTACHMENT_LEFT,
            "reset_angle": False
        },
        "arm": {
```

```
            "port": ATTACHMENT_RIGHT,
            "reset_angle": False,
            "gears": [[12, 20], [8, 24]],
            "positive_direction": Direction.COUNTERCLOCKWISE
        }
    }

    def initialize_positions(self):
        """
        Initialize the position of the attachments.

        TODO: Move this into the base attachment class.
        """
        self._get_motor("brush").run_until_stalled(-100, duty_limit=20)
        self._get_motor("arm").run_until_stalled(-50, duty_limit=30)

        self.zero_out("brush")
        self.zero_out("arm")

    def get_brush(self):
        """
        Executes a brush retrieval sequence.

        Lowers and raises brush attachment to complete the brush retrieval sequence.
        Includes a brief wait between movements for smooth operation.
        """
        self.angle("brush", 425)
        wait(100)
        self.target("brush", 0, speed=150)

    def lower_arm(self):
        """Lowers the arm to its operating position."""
        self.angle("arm", 95)

    def raise_arm(self):
        """Raises the arm to its retracted position."""
        self.angle("arm", -95, speed=150)

class ShipDropper(Attachment):
    """
    Attachment for Salvage Operation and Site Marking mission.

    Controls a rotational flag dropping mechanism to deliver the flag.

    Motors:
        - flag: Flag dropping mechanism
    """
    motors = {
        "flag": {
            "port": ATTACHMENT_LEFT,
            "reset_angle": False,
            "positive_direction": Direction.COUNTERCLOCKWISE,
            "gears": [20, 20]
        }
    }

    def drop_flag(self):
        """
        Executes the flag dropping sequence.

        Rotates the flag mechanism forward then backward to drop and reset
        the flag holder. Includes a brief wait for the flag to drop.
        """
        self.angle("flag", 130)
        wait(100)
        self.angle("flag", -130)

class ForgeAndSilo(Attachment):
    """
    Attachment for Silo, Forge, Heavy Lifting, and Who Lived Here? missions.

    Controls both a hammer mechanism and an arm for targeted missions.
```

```
    Motors:
        - hammer: Hammer mechanism for Silo activation
        - arm: Arm for Heavy Lifting mission
    """
    motors = {
        "hammer": {
            "port": ATTACHMENT_RIGHT,
            "reset_angle": False,
            "gears": [20, 20]
        },
        "arm": {
            "port": ATTACHMENT_LEFT,
            "reset_angle": False,
            "gears": [[12, 20], [8, 28]],
            "positive_direction": Direction.COUNTERCLOCKWISE
        }
    }

    def initialize(self):
        """
        Initializes the attachment's hammer and arm motors.

        Both attachments are raised to their fully upright positions.
        """
        self._get_motor("hammer").run_until_stalled(-100, Stop.COAST, 35)
        self.zero_out("hammer")
        print("hammer initialized")

        self._get_motor("arm").run_until_stalled(200, Stop.COAST, 15)
        self.zero_out("arm")

    def raise_arm(self, angle: int=120):
        """
        Raises the arm by rotating a specified angle.

        Args:
            angle: Angle in degrees to raise the arm (default: 120)
        """
        self.angle("arm", angle)
        print("Raised arm")

    def lower_arm(self, angle: int=120):
        """
        Lowers the arm by rotating a specified angle.

        Args:
            angle: Angle in degrees to lower the arm (default: -120)
        """
        self.angle("arm", -angle)

    def raise_hammer(self):
        """
        Raises the hammer to its starting position.

        Prepares the hammer for a slam action.
        """
        self.angle("hammer", -115)

    def slam_hammer(self):
        """
        Executes a hammer slam action for the Silo mission.

        Slams the hammer down at high speed then raises it
        back up. This action activates the silo mechanism.
        """
        self.angle("hammer", 115, speed=700)
        wait(100)
        self.raise_hammer()

class MineshaftAttachment(Attachment):
    """
```

```
    Attachment for Mineshaft Retrieval and Careful Recovery missions.

    Controls a grabber mechanism and an arm for retrieving artifacts
    from the mineshaft.

    Motors:
        - grabber: Grabbing mechanism for artifact retrieval
        - arm: Lifting mechanism to raise mineshaft track
    """
    motors = {
        "grabber": {
            "port": ATTACHMENT_LEFT,
            "positive_direction": Direction.COUNTERCLOCKWISE,
            "reset_angle": False,
            "gears": [12, 20]
        },
        "arm": {
            "port": ATTACHMENT_RIGHT,
            "positive_direction": Direction.COUNTERCLOCKWISE,
            "reset_angle": False,
            "gears": [12, 20]
        }
    }

    def init(self):
        """
        Initializes the arm to its desired position.
        """

        self._get_motor("arm").run_until_stalled(-100, duty_limit=20)
        self.zero_out("arm")
        wait(100)
        self.target("arm", 35, speed=500)
        print("Arm is initialized!")

    def drop_arm(self, angle: int, speed: int=300):
        """
        Lowers the arm by a specified angle.

        Args:
            angle: Angle in degrees to lower the arm (positive value)
            speed: Motor speed in degrees per second (default: 300)
        """
        self.angle("arm", angle * -1, speed)

    def raise_arm(self, angle: int, speed: int=300):
        """
        Raises the arm by a specified angle.

        Args:
            angle: Angle in degrees to raise the arm (positive value)
            speed: Motor speed in degrees per second (default: 300)
        """
        self.angle("arm", angle, speed)

    def drop_grabber(self, angle: int, speed: int=300):
        """
        Lowers the grabber by a specified angle.

        Args:
            angle: Angle in degrees to lower the grabber (positive value)
            speed: Motor speed in degrees per second (default: 300)
        """
        self.angle("grabber", angle, speed)

    def raise_grabber(self, angle: int, speed: int=300):
        """
        Raises the grabber by a specified angle.

        Args:
            angle: Angle in degrees to raise the grabber (positive value)
            speed: Motor speed in degrees per second (default: 300)
```

```
        """
        self.angle("grabber", angle * -1, speed)

    def grab_artifact(self):
        """
        Complete sequence for the Careful Recovery mission.

        This method should:
        1. Lower arm to position grabber near artifact
        2. Activate grabber to secure artifact
        3. Raise arm to remove artifact from mineshaft
        """
        # Lower arm to grab the artifact
        # Raise arm to remove the artifact
        raise NotImplementedError("Make sure to implement grab_artifact before using it")

class TesterAttachment(Attachment):
    """
    Testing attachment for development and debugging.

    Provides two motors for testing motor configurations and movements
    without affecting mission-specific attachments.

    Motors:
        - motor1: Left attachment motor (Port F), counterclockwise direction
        - motor2: Right attachment motor (Port B), clockwise direction
    """
    motors = {
        "left": {
            "port": ATTACHMENT_LEFT,
            "positive_direction": Direction.COUNTERCLOCKWISE,
            "reset_angle": False
        },
        "right": {
            "port": ATTACHMENT_RIGHT,
            "positive_direction": Direction.CLOCKWISE,
            "reset_angle": False
        }
    }
#endregion

#region Run Base, Runner, and Defined Runs
class Run:
    """
    Base class for mission runs.

    All mission-specific run classes must inherit from this class.
    Each subclass defines a color that triggers its execution and
    implements the execute() method with the mission sequence.

    Attributes:
        color: Color object that triggers this run (set in subclasses)
        debug_enabled: Boolean flag for enabling debug logging
        flags: Dictionary of configuration flags for the run
    """
    color = None

    def __init__(self, flags={}):
        """
        Initializes a run instance.

        Args:
            flags: Dictionary of configuration flags (default: empty dict)
        """
        self.debug_enabled = False
        self.flags = flags

    def execute(self, robot: Robot):
        """
        Executes the mission sequence for this run.

        This method must be implemented by subclasses to define
```

```
        the specific mission sequence.

        Args:
            robot: Robot instance to control

        Raises:
            NotImplementedError: If called on base class
        """
        raise NotImplementedError("Base class was called")

    def toggle_debug(self):
        """Toggles debug logging on or off."""
        self.debug_enabled = not self.debug_enabled

    def log(self, message: str):
        """
        Logs a debug message if debug mode is enabled.

        Args:
            message: Debug message string to log
        """
        if not self.debug_enabled:
            return

        print("Log: " + message)

class Runner:
    """
    Coordinates mission execution based on color detection.

    The Runner class manages a list of available runs and executes
    the appropriate mission sequence based on the color detected
    by the robot's color sensor.

    Attributes:
        runs: List of Run class types (not instances) that can be executed
    """

    def __init__(self, runs: list[Run]):
        """
        Initializes the runner with a list of available runs.

        Args:
            runs: List of Run class types (e.g., [Gray, Yellow, Green])
        """
        self.runs = runs

    def begin(self, robot: Robot, read_sensor_only: bool=False, read_hsv: bool=False):
        """
        Begins mission execution by detecting color and running the matching mission.

        Process:
        1. Turns on green light to indicate readiness
        2. Activates color sensor and reads color
        3. Finds matching run class based on detected color
        4. Executes the run's mission sequence
        5. Handles errors if no matching run is found

        Args:
            robot: Robot instance to control
            read_sensor_only: If True, only reads and prints color without executing (for debugging)
            read_hsv: If True, reads color as HSV values instead of Color object

        Raises:
            LookupError: If no run matches the detected color
        """
        # Read color from sensor
        robot.sensor_on()
        wait(100)  # Brief wait for sensor to stabilize
        print(robot.color_sensor.reflection())
        color = robot.read_sensor(read_hsv=read_hsv)
        print("Read color: " + str(color))
```

```
        # Debug mode: only read color, don't execute
        if read_sensor_only:
            return

        # Find and execute matching run
        for run in self.runs:
            if run.color == color:
                robot.sensor_off()

                # Instantiate and execute the run
                timer = StopWatch()
                run().execute(robot)
                print(f"Timing complete: took {timer.time() / 1000} seconds")

                print("====== Run complete! ======")
                return

        # Error handling: no matching run found
        robot.light.on(Color.RED)
        robot.display.icon(Icon.FALSE)
        wait(2000)
        raise LookupError(f"No run associated with color: {color}")

class MineshaftRetrieval_Old(Run):
    """
    Run 3: Missions 8, 9, and 10 - Mineshaft Retrieval, Tip the Scales, and What's On Sale missions.

    Mission sequence:
    1. Moves towards mineshaft
    2. Releases the cart
    3. Moves towards Tip the Scales and activates it
    4. Moves and triggers What's On Sale
    """
    color = Color.LIGHT_GREEN

    def execute(self, robot: Robot):
        mineshaft_attachment = MineshaftAttachment(robot)
        mineshaft_attachment.init()

        # Move towards mineshaft
        robot.move_forward(655)
        robot.turn_right(90)
        robot.move_forward(402.5) # previously 412.5mm
        robot.turn_left(87) # previously 90*
        #robot.move_backward(10)

        # Release the cart
        # TODO change speed of this
        mineshaft_attachment.raise_arm(180)
        #mineshaft_attachment.drop_arm(70)
        #mineshaft_attachment.raise_arm(90)
        robot.move_forward(35)

        # Move towards Tip the Scales
        robot.turn_right(87)
        robot.move_forward(530)
        robot.turn_right(90)
        robot.move_backward(20)
        # Bring down Tip the Scales
        mineshaft_attachment.drop_arm(160)
        mineshaft_attachment.raise_arm(160)

        # Move & trigger What's On Sale
        #robot.move_backward(15)
        robot.arc_right(120, -65)
        robot.turn_left(10)
        robot.move_forward(400)
        robot.turn_right(40)
        #robot.turn_left(45)
        robot.move_forward(780)
        # Return to blue home
```

```
        #robot.move_backward(150)
        #robot.turn_left(45)
        #robot.move_forward(350)
        #robot.turn_right(75)
        #robot.move_forward(750)

class SalvageOperation(Run):
    """
    Run 1: Mission 12 - Salvage Operation and Site Marking mission.

    Mission sequence:
    1. Moves forward to ship position
    2. Drops the flag
    3. Returns to starting position
    """
    color = Color.BLACK

    def execute(self, robot: Robot):
        ship_dropper = ShipDropper(robot)

        robot.move_forward(467)
        ship_dropper.drop_flag()
        robot.move_backward(467)

class SurfaceBrushing(Run):
    """
    Run 2: Missions 1 and 2 - Surface Brushing and Map Reveal mission.

    Mission sequence:
    1. Lowers arm
    2. Moves forward to position
    3. Executes brush action
    4. Returns to starting position
    """
    color = Color.GRAY

    def execute(self, robot: Robot):
        dropper = BrushAndReveal(robot)
        dropper.initialize_positions()

        self.get_brush(robot, dropper)
        self.map_reveal(robot, dropper)
        self.return_to_home(robot)

    def get_brush(self, robot: Robot, dropper: BrushAndReveal):
        robot.move_forward(545) # TODO: decrease speed of this
        dropper.get_brush()

    def map_reveal(self, robot: Robot, dropper: BrushAndReveal):
        dropper.lower_arm()
        robot.move_forward(135)
        dropper.raise_arm()

    def return_to_home(self, robot: Robot):
        robot.move_backward(680)

class MineshaftRetrieval(Run):
    """
    Run 3: Missions 8, 9, and 10 - Mineshaft Retrieval, Tip the Scales, and What's On Sale missions.

    Mission sequence:
    1. Moves towards mineshaft
    2. Releases the cart
    3. Moves towards Tip the Scales and activates it
    4. Moves and triggers What's On Sale
    """
    color = Color.LIGHT_GREEN

    def execute(self, robot: Robot):
        mineshaft_attachment = MineshaftAttachment(robot)
        mineshaft_attachment.init()
```

```
        # Move towards mineshaft
        robot.move_forward(655)
        robot.turn_right(90)
        robot.move_forward(406) # previously 412.5mm
        robot.turn_left(87) # previously 90*
        wait(200)
        robot.move_forward(10)
        wait(200)
        #robot.move_backward(10)

        # Release the cart
        # TODO change speed of this
        mineshaft_attachment.raise_arm(75)
        wait(200)
        robot.move_forward(60)
        wait(200)
        mineshaft_attachment.raise_arm(15)
        wait(200)
        robot.move_forward(35)
        wait(200)
        mineshaft_attachment.drop_grabber(65)
        wait(200)
        robot.move_forward(20)
        wait(200)
        mineshaft_attachment.drop_grabber(115, speed=1000)
        robot.end_early()
        #mineshaft_attachment.drop_arm(70)
        #mineshaft_attachment.raise_arm(90)

        # Move towards Tip the Scales
        robot.turn_right(87)
        robot.move_forward(530)
        robot.turn_right(90)
        robot.move_backward(20)
        # Bring down Tip the Scales
        mineshaft_attachment.drop_arm(160)
        mineshaft_attachment.raise_arm(160)

        # Move & trigger What's On Sale
        #robot.move_backward(15)
        robot.arc_right(120, -65)
        robot.turn_left(10)
        robot.move_forward(400)
        robot.turn_right(40)
        #robot.turn_left(45)
        robot.move_forward(780)
        # Return to blue home
        #robot.move_backward(150)
        #robot.turn_left(45)
        #robot.move_forward(350)
        #robot.turn_right(75)
        #robot.move_forward(750)

class Silo(Run):
    """
    Run 4: Missions 5, 6, and 7 - Silo, Forge, Heavy Lifting, and Who Lived Here? missions.

    Mission sequence:
    1. Moves to forge position
    2. Executes multiple hammer slams for silo activation
    3. Navigates to additional mission positions
    4. Returns to starting position
    """
    color = Color.GREEN

    def execute(self, robot: Robot):
        forge_attachment = ForgeAndSilo(robot)

        self.silo(robot, forge_attachment)
        self.forge(robot)
        self.millstone(robot, forge_attachment)
        self.who_lived_here_and_finish(robot)
```

```
    def silo(self, robot: Robot, forge_attachment: ForgeAndSilo):
        forge_attachment.initialize()
        forge_attachment.lock("arm")
        # Move to Silo position
        robot.move_forward(480)

        # Execute 5 hammer slams for Silo mission
        for _ in range(4):
            forge_attachment.slam_hammer()
            wait(100)

    def forge(self, robot: Robot):
        robot.configure_straight(400, 1200)
        robot.configure_turn(90, 1200)
        robot.move_forward(174)
        robot.turn_right(41)

    def millstone(self, robot: Robot, forge_attachment: ForgeAndSilo):
        forge_attachment.lower_arm(120) # previously: 117 deg
        robot.move_forward(50)
        wait(500) # ms
        robot.move_forward(30)
        forge_attachment.raise_arm(90)
        robot.move_backward(88)
        robot.turn_right(41)
        robot.move_forward(180)
        robot.turn_left(10)
        robot.move_forward(120)
        robot.move_backward(300)

    def who_lived_here_and_finish(self, robot: Robot):
        robot.turn_left(85)
        robot.move_forward(120)
        robot.turn_left(38)
        robot.move_forward(15)
        robot.move_backward(15)
        robot.turn_right(25)
        robot.move_backward(770)

class CrossTable(Run):
    """
    Run 5: Cross-table navigation - Blue to Red home.

    High-speed cross-table movement for missions requiring
    the robot to traverse from one side of the table to the other.
    """
    color = Color.WHITE

    def execute(self, robot: Robot):
        # Configure for high-speed cross-table movement
        robot.configure_straight(1000, 1200)
        robot.move_forward(1800)

class Forum(Run):
    """
    Run 6: Mission 14 - Forum delivery.

    Simple forward and backward movement to push forum pieces in place.
    """
    color = Color.ORANGE

    def execute(self, robot: Robot):
        robot.configure_straight(500, 800)
        robot.move_forward(450)
        robot.move_backward(450)

class Test(Run):
    """
    Test Run: Development and testing purposes.

    No assigned missions. Used for testing attachments
```

```
    and robot functionality during development.
    """
    color = Color.YELLOW

    def execute(self, robot: Robot):
        brush_attachment = BrushAndReveal(robot)
        # brush_attachment.initialize_positions()

        # brush_attachment.lower_arm()

        # wait(200)

        brush_attachment.raise_arm()

class Demo(Run):
    """
    Demo Run: Demonstration purposes.

    Tests hammer and arm functionality with basic movements.
    Used for demonstrations for judges.
    """
    color = Color.BLUE

    def execute(self, robot: Robot):
        forge_attachment = ForgeAndSilo(robot)
        forge_attachment.initialize()
        robot.move_forward(100)

        # Execute multiple hammer slams for Silo mission demonstration
        for _ in range(3):
            forge_attachment.slam_hammer()
            wait(100)

        robot.move_backward(100)
#endregion

def main():
    """
    Main entry point for the robot program.

    Initializes the robot with physical specifications:
    - Wheel diameter: 61.1 mm
    - Axle track: 104.3 mm

    Configures the robot for mission execution:
    - Enables gyroscope for improved navigation accuracy
    - Sets default movement speed and acceleration parameters

    Registers all available mission runs and starts the runner,
    which will detect the color cartridge and execute the appropriate mission.
    """
    # Initialize robot with wheel diameter and axle track measurements
    robot = Robot(61.1, 104.3)

    # Enable gyroscope for improved turning accuracy
    robot.drive_base.use_gyro(True)

    # Configure default movement settings
    robot.configure_straight(500, 1200)
    robot.configure_turn(90, 1200)

    # Register all available mission runs
    known_runs = [
        SalvageOperation,
        SurfaceBrushing,
        MineshaftRetrieval_Old,
        Silo,
        CrossTable,
        Forum,
        Test,
        Demo,
    ]
```

```
    # Start the runner to detect color and execute mission
    Runner(known_runs).begin(robot)
    wait(500)

if __name__ == "__main__":
    main()
```