```java
package org.whitneyrobotics.ftc.teamcode.Subsystems;

import com.qualcomm.robotcore.util.ElapsedTime;
import org.firstinspires.ftc.robotcore.external.navigation.DistanceUnit;

/**
 * Automates a full FTC scoring cycle using a finite state machine.
 *
 * This class sequences intake and outtake mechanisms in a deterministic order to
 * reduce driver workload and improve consistency during matches. Each state enforces
 * timing delays to prevent mechanical interference, and intake completion is gated
 * using a REV 2m Distance Sensor to confirm that a game piece has been successfully acquired.
 *
 * Usage:

 * - Call toggle() to start automation (and to manually step through states when active)
 * - Call update() every loop to advance states automatically when timing and sensor conditions are met
 * - Call stop() to cancel and move the robot into a safe posture
 */
public class CycleAutomationStateMachine {

    public enum CycleMode { INTAKE, OUTTAKE }
    public enum IntakeState { IDLE, HORIZON_OUT, INTAKE_DOWN, COMPLETE }
    public enum OuttakeState { IDLE, HORIZON_IN, WRISTUP, SERVO_SWAP, ELBOW_FLIP, COMPLETE }

    private final RobotImpl robot;

    // Current high-level mode and sub-state
    private CycleMode currentMode = CycleMode.INTAKE;
    private IntakeState currentIntakeState = IntakeState.IDLE;
    private OuttakeState currentOuttakeState = OuttakeState.IDLE;

    // Whether automation is currently running
    private boolean active = false;

    // FTC timer used to measure time spent in the current state
    private final ElapsedTime stateTimer = new ElapsedTime();

    // Intake timing delays
    private double delayIdleIntake = 0.0;
    private double delayHorizonOutIntake = 0.1;
    private double delayIntakeDown = 0.1;

    // Outtake timing delays
    private double delayIdleOuttake = 0.0;
    private double delayHorizonIn = 0.1;
    private double delayWristUp = 1.5;
    private double delayServoSwap = 1.0;
    private double delayElbowFlip = 1.0;

    // Failsafe timeout to avoid getting stuck in a state
    private double stateTimeout = 3.5;

```

```java
    // Distance sensor gating parameters (REV 2m Distance Sensor)
    private double minTimeBeforeSensorGate = 0.08;
    private double pieceDistanceThresholdCm = 4.0;

    public CycleAutomationStateMachine(RobotImpl robot) {
        this.robot = robot;
        stateTimer.reset();
    }

    public void toggle() {
        // Start automation if inactive
        if (!active) {
            active = true;
            stateTimer.reset(); // start timing current state
            return;
        }

        // Manual step-through when active, useful for debugging
        if (currentMode == CycleMode.INTAKE && currentIntakeState != IntakeState.COMPLETE) {
            advanceIntakeState();
        } else if (currentMode == CycleMode.OUTTAKE && currentOuttakeState != OuttakeState.COMPLETE) {
            advanceOuttakeState();
        } else {
            switchMode();
        }
    }

    public void update() {
        if (!active) return;

        // Stop automation if a state takes too long
        if (stateTimer.seconds() >= stateTimeout) {
            stop();
            return;
        }

        if (currentMode == CycleMode.INTAKE) {
            double delayVal = getDelayForCurrentIntakeState();

            // Wait until minimum delay has elapsed
            if (stateTimer.seconds() >= delayVal) {

                // Require a detected game piece before completing intake
                if (currentIntakeState == IntakeState.INTAKE_DOWN) {
                    if (stateTimer.seconds() < minTimeBeforeSensorGate) return;
                    if (!isPieceDetected()) return;
                }

                advanceIntakeState();
            }
        } else {
            double delayVal = getDelayForCurrentOuttakeState();

            // Advance outtake states purely by timing
            if (stateTimer.seconds() >= delayVal) {
                advanceOuttakeState();
            }
        }
```

```
111         }
112
113         public void stop() {
114             active = false;
115
116             // Move mechanisms into a safe, non-interfering posture
117             robot.horizontalServo.currentState = HorizontalServo.Positions.RETRACT;
118             robot.intakeWrist.position = ClawWrist.ClawState.UP;
119             robot.elbowWrist.position = ElbowWrist.ElbowState.DOWN;
120
121             stateTimer.reset();
122         }
123
124         public void reset() {
125             // Fully reset automation state
126             active = false;
127             currentMode = CycleMode.INTAKE;
128             currentIntakeState = IntakeState.IDLE;
129             currentOuttakeState = OuttakeState.IDLE;
130             stateTimer.reset();
131         }
132
133         private boolean isPieceDetected() {
134             // Detect possession by checking distance to the nearest object at the
intake
135             double cm = robot.intakeDistance.getDistance(DistanceUnit.CM);
136             return cm > 0 && cm < pieceDistanceThresholdCm;
137         }
138
139         private double getDelayForCurrentIntakeState() {
140             switch (currentIntakeState) {
141                 case IDLE:
142                     return delayIdleIntake;
143                 case HORIZON_OUT:
144                     return delayHorizonOutIntake;
145                 case INTAKE_DOWN:
146                     return delayIntakeDown;
147                 case COMPLETE:
148                     return 0;
149             }
150             return 0;
151         }
152
153         private double getDelayForCurrentOuttakeState() {
154             switch (currentOuttakeState) {
155                 case IDLE:
156                     return delayIdleOuttake;
157                 case HORIZON_IN:
158                     return delayHorizonIn;
159                 case WRISTUP:
160                     return delayWristUp;
161                 case SERVO_SWAP:
162                     return delayServoSwap;
163                 case ELBOW_FLIP:
164                     return delayElbowFlip;
165                 case COMPLETE:
166                     return 0;
167             }
168             return 0;
169         }
```

```
170
171       private void advanceIntakeState() {
172           switch (currentIntakeState) {
173               case IDLE:
174                   currentIntakeState = IntakeState.HORIZON_OUT;
175                   // Extend and open to prepare for acquisition
176                   robot.horizontalServo.currentState =
HorizontalServo.Positions.EXTEND;
177                   robot.OuttakeServo.currentState = outtakeServo.Positions.OPEN;
178                   robot.elbowWrist.position = ElbowWrist.ElbowState.DOWN;
179                   break;
180
181               case HORIZON_OUT:
182                   currentIntakeState = IntakeState.INTAKE_DOWN;
183                   // Lower wrist to engage the game piece
184                   robot.intakeWrist.position = ClawWrist.ClawState.DOWN;
185                   break;
186
187               case INTAKE_DOWN:
188                   currentIntakeState = IntakeState.COMPLETE;
189                   // Intake cycle complete only after distance confirmation
190                   break;
191
192               case COMPLETE:
193                   break;
194
195               default:
196                   break;
197           }
198
199           // Restart timer for the new intake state
200           stateTimer.reset();
201       }
202
203       private void advanceOuttakeState() {
204           switch (currentOuttakeState) {
205               case IDLE:
206                   currentOuttakeState = OuttakeState.HORIZON_IN;
207                   // Retract intake assembly before lifting or transferring
208                   robot.horizontalServo.currentState =
HorizontalServo.Positions.RETRACT;
209                   break;
210
211               case HORIZON_IN:
212                   currentOuttakeState = OuttakeState.WRISTUP;
213                   // Raise intake wrist to clear the transfer path
214                   robot.intakeWrist.position = ClawWrist.ClawState.UP;
215                   break;
216
217               case WRISTUP:
218                   currentOuttakeState = OuttakeState.SERVO_SWAP;
219                   // Transfer the game piece from intake to outtake
220                   robot.OuttakeServo.currentState = outtakeServo.Positions.CLOSED;
221                   robot.intakeServo.currentState = intakeServo.Positions.OPEN;
222                   break;
223
224               case SERVO_SWAP:
225                   currentOuttakeState = OuttakeState.ELBOW_FLIP;
226                   // Move outtake into scoring orientation
227                   robot.elbowWrist.position = ElbowWrist.ElbowState.UP;
```

```java
                break;

            case ELBOW_FLIP:
                currentOuttakeState = OuttakeState.COMPLETE;
                break;

            case COMPLETE:
                break;

            default:
                break;
        }

        // Restart timer for the new outtake state
        stateTimer.reset();
    }

    private void switchMode() {
        // Swap between intake and outtake cycles
        if (currentMode == CycleMode.INTAKE) {
            currentMode = CycleMode.OUTTAKE;
            currentOuttakeState = OuttakeState.IDLE;
        } else {
            currentMode = CycleMode.INTAKE;
            currentIntakeState = IntakeState.IDLE;
        }

        stateTimer.reset();
    }

    public String getState() {
        if (currentMode == CycleMode.INTAKE) {
            return "INTAKE: " + currentIntakeState.name();
        } else {
            return "OUTTAKE: " + currentOuttakeState.name();
        }
    }

    public boolean isActive() {
        return active;
    }
}
```