SOFTWARE DESIGN DOCUMENT

**Project:** ECSE 211 Final Design Project – Team 6
**Document Version Number**: 7.0
**Creation Date:** 22/10/2017
**Original Author:** Justin Tremblay

**Edit History:**
[18/10/2017] Justin Tremblay: First version of the document, added state machine and class diagrams.
[22/10/2017] Alex Hale: moved to Google Docs for easier collaboration; transferred existing info and diagrams
[28/10/2017] Justin Tremblay: Updated document with new diagrams, designs and systems.
[29/10/2017] Justin Tremblay: Continued working on design updates.
[30/10/2017] Alex Hale: formatting and grammar touch-ups for Week 3 submission
[5/11/2017] Alex Hale: updated Class Hierarchy with class descriptions; combined pollers into one class to reflect changes to the code
[16/11/2017] Justin Tremblay: Small changes
[19/11/2017] Alex Hale: added reference to API document
[20/11/2017] Justin Tremblay: Added localization flowchart, updated class diagram, updated descriptions and added multithreading section
[20/11/2017] Alex Hale: formatting and clarity improvements throughout entire document
[23/11/2017] Alex Hale: improved details about the functionality of the Searching algorithm
[29/11/2017] Justin Tremblay: Added clarifications, information and diagrams for final submission

## 1.0 - TABLE OF CONTENTS

2.0 - State Machine
3.0 - Model Diagram
    3.1 - FinalProject
    3.2 - MainController
    3.3 - Odometer
    3.4 - Localizer, UltrasonicLocalizer, LightLocalizer
    3.5 - Navigation
    3.6 - Zipline
    3.7 - Searcher
    3.8 - Driver
    3.9 - SensorPoller
    3.10 - SensorData
4.0 - Multithreading

NOTE: a full outline of the software structure, including the use of each class and function, can be found in the project API (included alongside this document).

## 2.0 - State Machine

The robot is controlled by a state machine that handles decision-making depending on the robot's inputs and its current goal. It is inspired from the GOAP (Goal Oriented Action Planning) system which is used in the video game industry to control the artificial intelligence of non-playable characters.

More info on GOAP: [http://alumni.media.mit.edu/~jorkin/goap.html](http://alumni.media.mit.edu/~jorkin/goap.html)

Figure 1 shows a diagram of the robot's state machine.

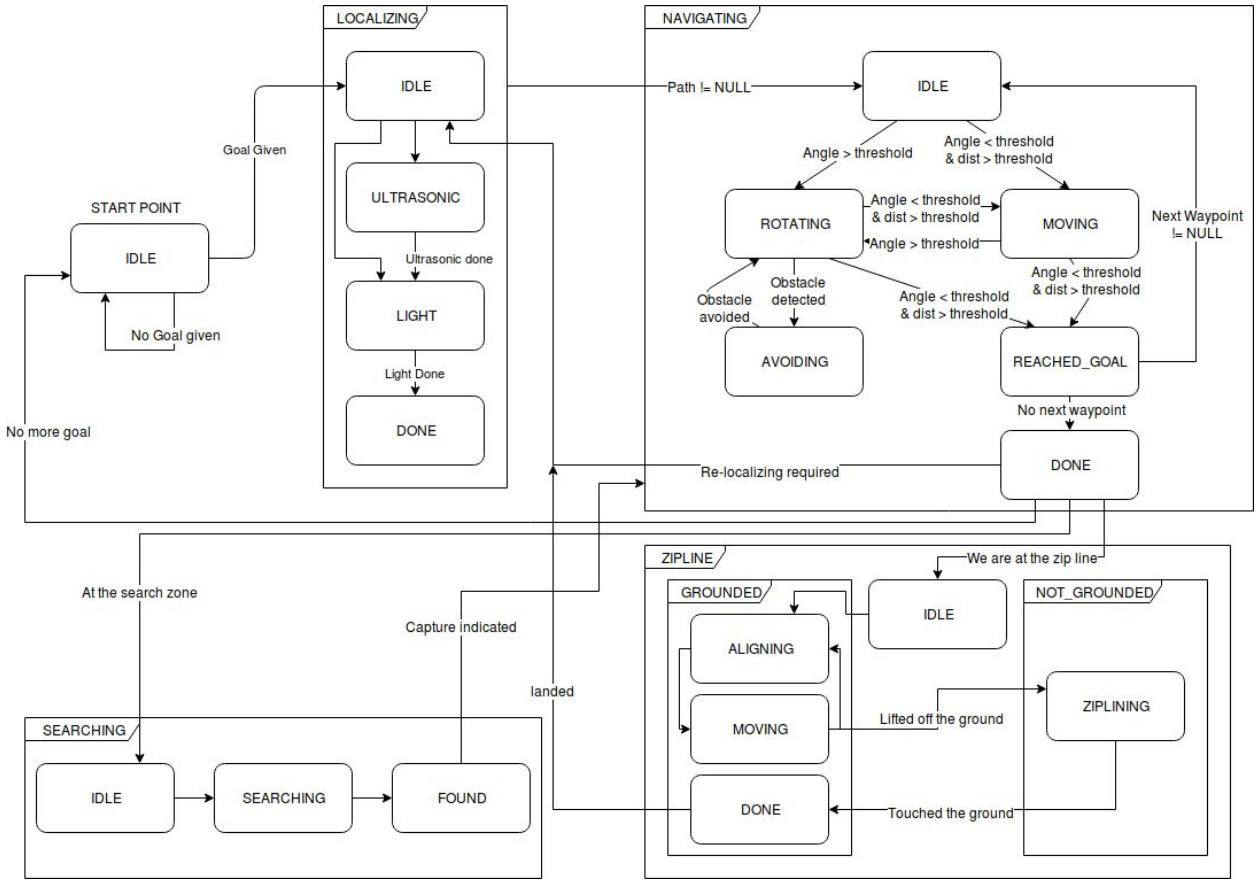


**Figure 1 - Robot State Machine**

**Example of state machine implementation:**

```java
/**
 * Root of the robot's state machine. The current state of the robot is processed at every iteration
 * of the run() loop. Depending on the current state, the process method of the corresponding subsystem
 * is called, which processes the subsystem's state.
 *
 * This structure eliminates the problem of accessing variables from multiple threads as everything
 * is essentially done is the same thread.
 */
private void process() {
  switch (cur_state) {
    case IDLE:
      cur_state = process_idle();
      break;
    case LOCALIZING:
      cur_state = process_localizing();
      break;
    case NAVIGATING:
      cur_state = process_navigating();
      break;
    case ZIPLINING:
      cur_state = process_ziplining();
      break;
    case SEARCHING:
      cur_state = process_searching();
      break;
    default: break;
  }
}
```

**Figure 2 - Sample instance of a state machine**

This pattern is used in each of the robot's subsystems. At every iteration, it runs the `process_state()` method corresponding to the current state of the system. This method handles the required actions by calling the corresponding subsystem's `process()` method and returns the next state depending on various parameters and conditions.

## 3.0 - Class Diagram

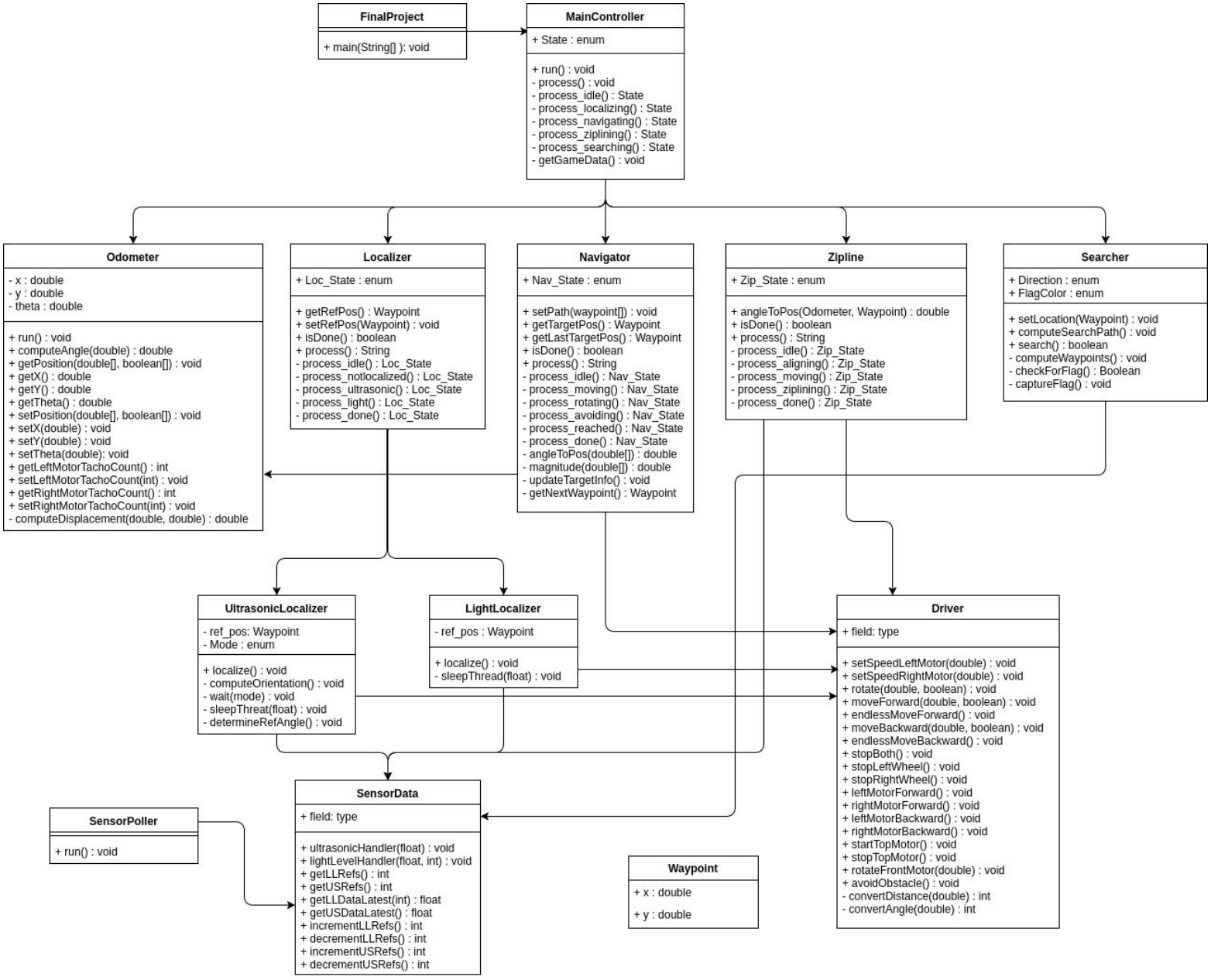


**Figure 3 - Class Hierarchy**

*Note: the Waypoint class doesn't have its own section because its only function is to provide an abstraction for the grid coordinates.*

*Note: some methods and fields are missing from this diagram in the name of compactness. For the full API documentation, refer to the included project API.*

### 3.1 - FinalProject

This is the main class of the program. It contains the main method of the program, and it groups all the constants in one place for easy tuning. It handles initializing the motors, sensors and subsystems, as well as starting all the necessary threads.

### 3.2 - MainController

This is the main control thread and the root of the state machine. It keeps track of the robot's current goal and delegates control to the appropriate subsystem depending on the current state. For example, if the robot has to move:

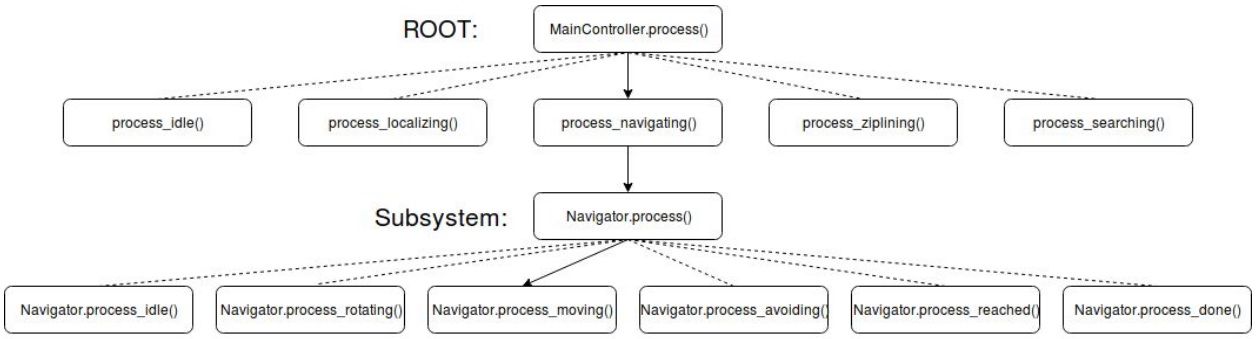


**Figure 4 - State Machine Structure**

When the `process()` method from the controller is called, the `process_navigating()` method is called This method checks for conditions before and after calling the navigator's own `process()` method, which will branch to `process_moving()`. This will then check for other conditions and tell the driver class to move forward, stop, and return the next state. This code structure allows the robot to continuously re-evaluate its status and environment to make decisions on what to do next.

When the MainController's `start()` method is called, it immediately attempts to establish a connection with the game server and waits for input before moving on to the execution of its tasks.

Upon receiving the game data from the server, the MainController will internally plan most of the paths the robot will take in order to reach its objective. This makes it so the robot can execute its tasks without wasting resources on computing paths at the same time.

### 3.3 - Odometer

This class keeps track of the robot's position and orientation on the playing field. It uses the motors' tachometers and simple mathematics to determine displacement. It continuously runs in its own thread.

The odometer works starts by getting the variation of rotation of both wheels via comparison of the previous tachometer and current tachometer counts. The displacement of each wheel can then be computed using this formula:

$$d = \frac{wheelradius * \pi * \Delta_{rotation}}{180}$$

Knowing the displacement of each wheel, the variation in orientation of the robot can then be computed:

$$\Delta\theta = \frac{d_R - d_L}{wheelbase}$$

The distance travelled can also be found:

$$dist = \frac{d_R + d_L}{2}$$

Trigonometry is then used to determine the movement in the x-axis and the y-axis. At the end of all the computations, the new x, y, and $\Theta$ are sent to the odometer to update its position and orientation.

### 3.4 - Localizer, UltrasonicLocalizer, LightLocalizer

These classes determine the orientation and position of the robot at the start of the game and when the robot comes off the zipline. It is divided in two parts:

- Ultrasonic Localizer: uses the ultrasonic sensor and two walls to determine the orientation of the robot, the positive x-axis being 0 degrees.
- Light Localizer: uses two colour sensors (see Hardware Design section 8.0) and the orientation angle to determine the robot's position with respect to a reference position.

When localizing, the main controller first determines if ultrasonic localization is necessary (this amounts to determining whether the robot is in a corner). If the robot is in a corner, a reading is taken from the ultrasonic sensor to determine whether rising edge or falling edge ultrasonic localization is required. If localization is taking place in a corner, the robot aligns itself to a reference angle. If localization is taking place in the middle of the field, the robot aligns itself to 45º, backs up, then aligns itself to a reference angle. The reference angle can be 0°, 90°, 180° or 270°, depending on the position of the robot. The reference angle is used to orient the robot in a direction perpendicular to the line it is looking for. Once aligned, robot then starts moving

forward. When a colour sensor detects a line, the motor on the corresponding side of the robot is stopped. The opposite side of the robot continues to move forward until the remaining colour sensor detects the line, at which point that motor is stopped. Once both wheels are stopped, the corresponding coordinate is set, the orientation is corrected in the odometer, and the robot backs up, turns, and repeats the procedure for the other line.

In the case where one or both sensors fail to detect the line, or the robot had already passed the line prior to localizing, a failsafe procedure kicks in. If both sensors haven't detected the line after four seconds of forward motion, the robot will start going backwards at a slightly higher speed to return to the line. If one sensor detects the line but the other has not, the moving motor will change direction after four seconds and return the corresponding side of the robot to the line. This process repeats until both sensors detect the line.

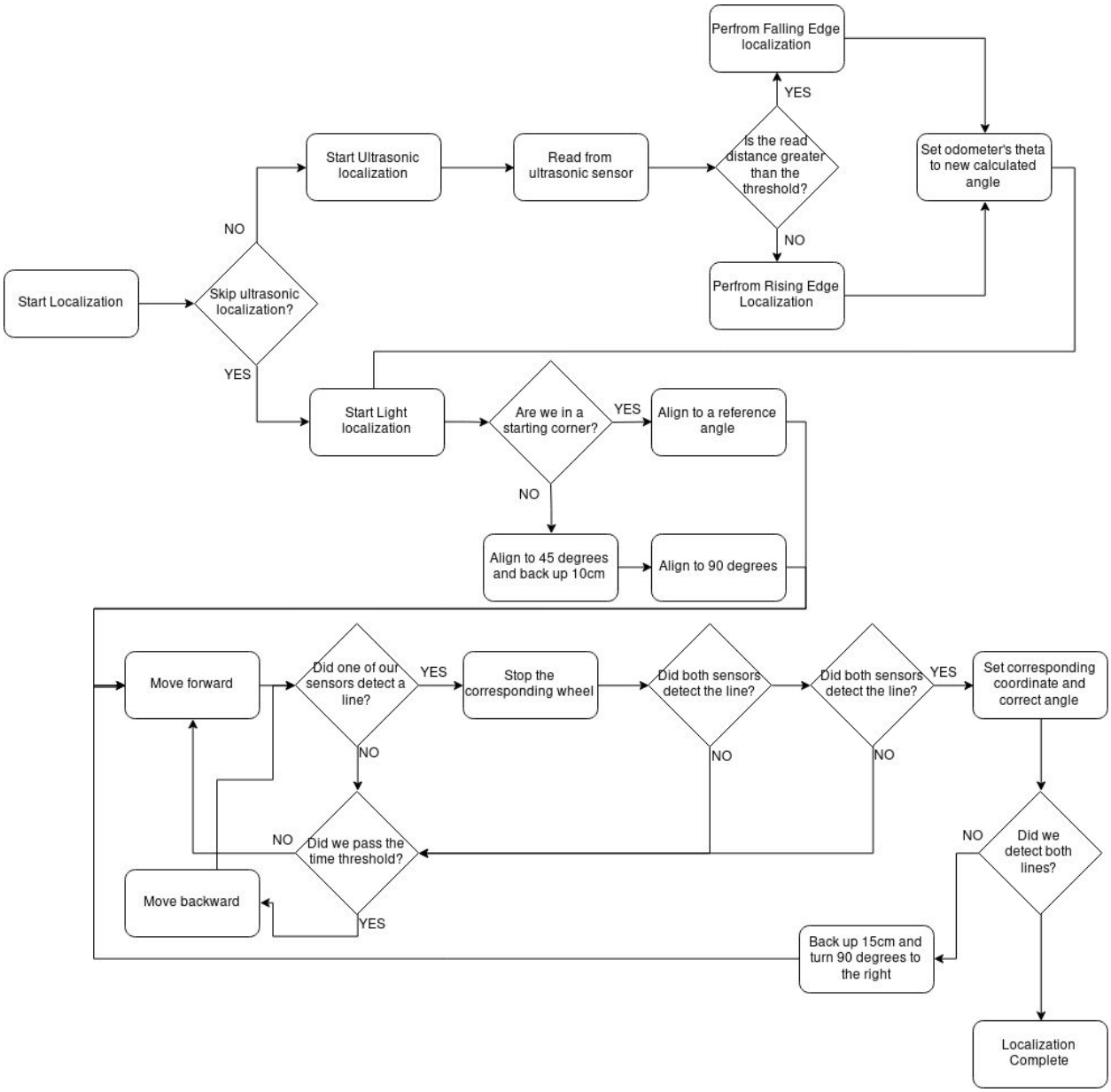


**Figure 5 - Localization Flowchart**

### 3.5 - Navigation

This class moves the robot through a set of waypoints on the grid and, should they be encountered, avoids obstacles.

Navigation uses an array of waypoints. The class starts by getting the next waypoint in the path and computing the angle and distance to it using simple vector mathematics and trigonometry. The robot will then either start moving to the target waypoint or rotate toward it, depending on the current orientation. While moving, the robot continually re-evaluates its angle to the target waypoint. If the angle grows higher than the allowed error threshold, the robot pauses, adjusts its orientation, then continues moving. This allows for relatively precise navigation across short to medium distances. When the robot reaches its destination, it fetches the next waypoint from the array. If the fetch returns “null”, the robot is done navigating.

When an obstacle is encountered, the navigator first confirms that this obstacle needs to be avoided by comparing its current orientation to its destination. If avoidance is required, the ultrasonic sensor is rotated to a 45° orientation using the front motor and the navigator goes into the AVOIDING state.The driver’s `avoidObstacle()` method is called to implement a P-Controller. The robot moves around the obstacle until the obstacle is no longer between the robot and its destination. The state machine then returns to NAVIGATING and the ultrasonic sensor is set back to its original forward position.

The Navigator class is implemented using the same tree-like state machine structure as the MainController class. At every iteration, the `process()` method is called, which then moves into the `process_x()` method (where `x` is the robot’s current state). This method then evaluates various parameters (e.g.angle to target waypoint, distance to target waypoint) to determine what to do and what state to return.

For example: if the angle to the target position is less than the error threshold and the distance is greater than the distance threshold, the navigator will move into the MOVING state, which will drive the robot toward the waypoint. While moving, if the angle to the target waypoint becomes higher than the error threshold, the robot will stop, rotate to face the target waypoint, and start moving again. This allows for very precise navigation. In the specific case where the robot’s angle to the target waypoint becomes very high, instead of doing long turns, the robot will back up, adjust its angle and then start moving forward again. This avoids accumulating too much error in the odometer angle.

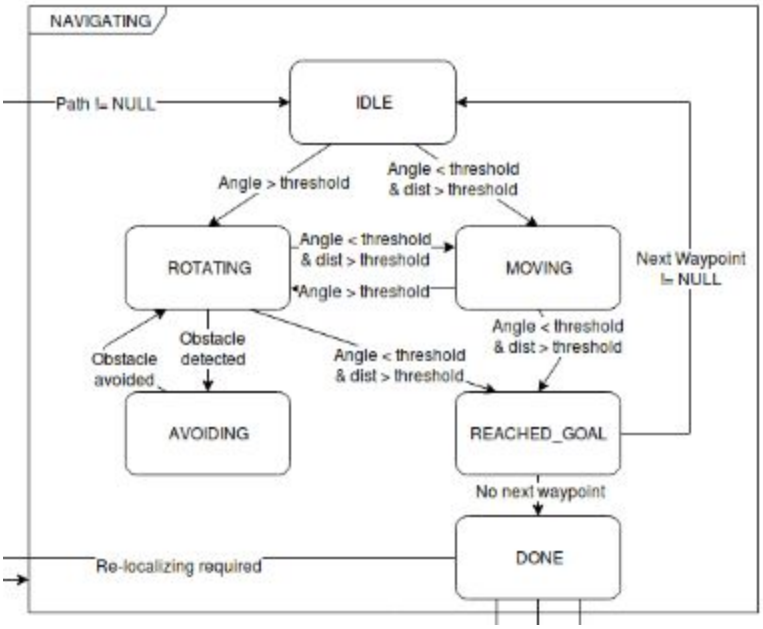


**Figure 6 - Navigator state machine**

### 3.6 - Zipline

This class moves the robot on the zipline. It uses one of the colour sensors (see Hardware Design section 8.0) to determine whether or not the robot is touching the ground. When the robot lands, it moves forward and re-localizes.

The class works by aligning to the zipline and moving forward for a set distance while blocking the thread's execution. When the robot has travelled this set distance, the robot can then detect whether or not it is on the ground or not. It then keeps polling the color sensor to detect when it touches the ground again.

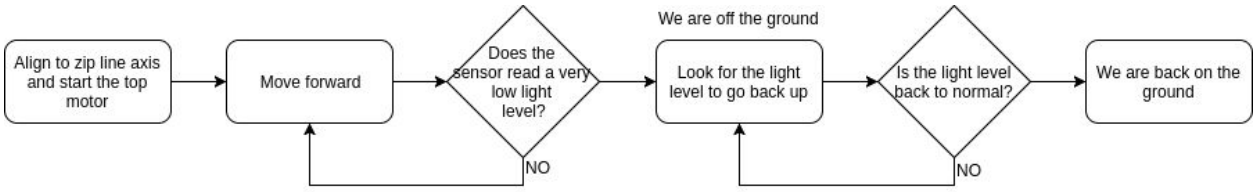


**Figure 7 - Zip Line Flowchart**

### 3.7 - Searcher

This class is invoked once the robot arrives at the searching area. A path is developed around the searching area, and the robot slowly follows this path while looking into the searching zone with the ultrasonic sensor. When a flag is detected, the robot turns and moves directly into the

searching zone, approaching the flag so the colour sensor can read the flag's colour. If the flag is the correct colour, the robot indicates capture with three beeps, then leaves the searching zone and passes control to the Navigator class. If the flag is not the correct colour, the robot returns to the path along the exterior of the searching zone and resumes its search for flags.

The searching process is reliable, but slow, and it is quite possible that the game will run out of time before the robot finds the required flag. To gain the maximum possible number of points in the match, the robot's searching time is capped at 20 seconds. If the robot has not found the block in this time, it will pass control to the navigator to return back to the starting corner.

### 3.8 - Driver

This class is used in every system that requires the robot to move. It supports moving forward and backward, as well as rotation. It also moves the zip line and sensor motors.

For moving forward, backward and rotating, motor synchronization has been implemented to ensure that there is close to no delay between the two motors when moving the motors.

To account for the weight distribution of the robot that is slightly to the right because of the zip line motor (see Hardware Design, section 8.0), the right motor's speed is always multiplied by a small amount to make sure the robot moves in a straight line. This amount is also divided from the right wheel's displacement in the odometer to account for this change.

### 3.9 - SensorPoller

This class collects samples from each sensor and sends them to the SensorData class for processing and easier accessibility.

It is running in its own thread to ensure that it is not slowed down by other systems going into computationally expensive algorithms. It also only pushes data to the SensorData class when the references to the desired sensor(s) are higher than zero.

### 3.10 - SensorData

This class retrieves sensor readings from SensorPoller and processes them. It also facilitates easier access to the data for other subsystems.

The SensorData class works by counting the references to itself, only getting data when at least one other object is referring to it. This means that the robot only gets and processes sensor data when it is needed, saving system resources and using them for more important computations.

When the robot gets the data from the sensor poller, it stores the data from each sensor in a circular array of twenty elements. It can also compute the moving derivative of the data in case it is needed.

### 4.0 - Multithreading

To ensure that the robot can perform all of its tasks while continuously updating its current position and reading from the sensors, the software must be divided into multiple threads to allow systems to run in parallel. However, since the EV3 brick has very limited resources, it is important to properly design the software so that the brick is not overloaded with concurrent threads.

The software design went through many iterations to determine the maximum number of concurrently running threads. Initially, the system had seven threads since every subsystem was implemented as a thread. Following the results of Laboratory 5, the number of threads was reduced to five, creating a major performance improvement. The challenge was to minimize the number of threads while ensuring that each critical system of the software had its own dedicated thread. This guarantees that critical systems cannot be slowed down by another system on the same thread.

The final design only uses four threads, one being the display thread, which can be disabled during competition. The first thread is the main controller thread, which allows the entire state machine to be run from only one thread due to its tree-like structure. The second thread is the odometer thread. It is required that the odometer has its own thread so the robot's position and orientation can be continuously updated without hindering its ability to perform other tasks. The third thread is used to poll the robot's sensors. The SensorPoller only sends data if the references to SensorData are greater than zero, ensuring that resources aren't wasted.