Software Design Document

Version 7.02

Team 10

Mohamed Samee 260770939
Ketan Rampurkar 260732873
Marie-Lynn Mansour 260770547
Carlo D'Angelo 260803454
Hao Shu 260776361
Cristian Ciungu 260869291

Edit History

Title: Software design document edit history			
Author: Marie-Lynn Mansour, Documentation Manager			
Date Created: April 9th, 2019			
Version Number	**Date Edited**	**Author Name(s)**	**Edits Made**
7.00	April 9th	Mohamed Samee	Edited Section 4.3 and Section 6.0
7.01	April 10th	Marie-Lynn Mansour	Formatting changes made; Captioned figures; Table of contents adjusted
7.02	April 11th	Mohamed Samee, Carlo D'Angelo, and Marie-Lynn Mansour	Edited Section 4.3; Added Section 3.0 Software Constraints; Edited Sections 4.2, 4.4.10, 4.4.11, and 4.4.12; Section 7.0

1.0 Table of Contents

2.0 Software Design Process

Prior to beginning the software development, it was imperative for us to create a skeleton of the software design based on the project requirements and constraints. In order to visually demonstrate our ideas and the software design as a whole, we spent a considerable amount creating flow charts and class diagrams. The flow charts and class diagrams demonstrated the interactions between the different aspects of the code. They drove the development of the classes and methods that were specified. The diagrams enabled the software development phase to be much more efficient and organized.

3.0 Software Constraints

As outlined in the Constraints Document V2.00, there were multiple software constraints that we anticipated prior to beginning software development. Firstly, the EV3 processor cannot handle too many threads at the same time. Section 6.2 explains a particular situation involving a thread issue and how we solved it in time for the final demo. Secondly, the performance of the EV3 changes when the battery voltage is less than 8.0 Volts. Despite constant variables being declared as static or final, they are not necessarily set in stone when the battery of the EV3 is not fully charged. Furthermore, the software is restricted from performing complex calculations and is slow on integer calculations. Lastly, the accuracy of measured values from sensors is decreased. Repeatedly casting measured values can cause the inaccuracies to accumulate, causing the error to get larger.

4.0 Preliminary Software Architecture

4.1 Flow Charts

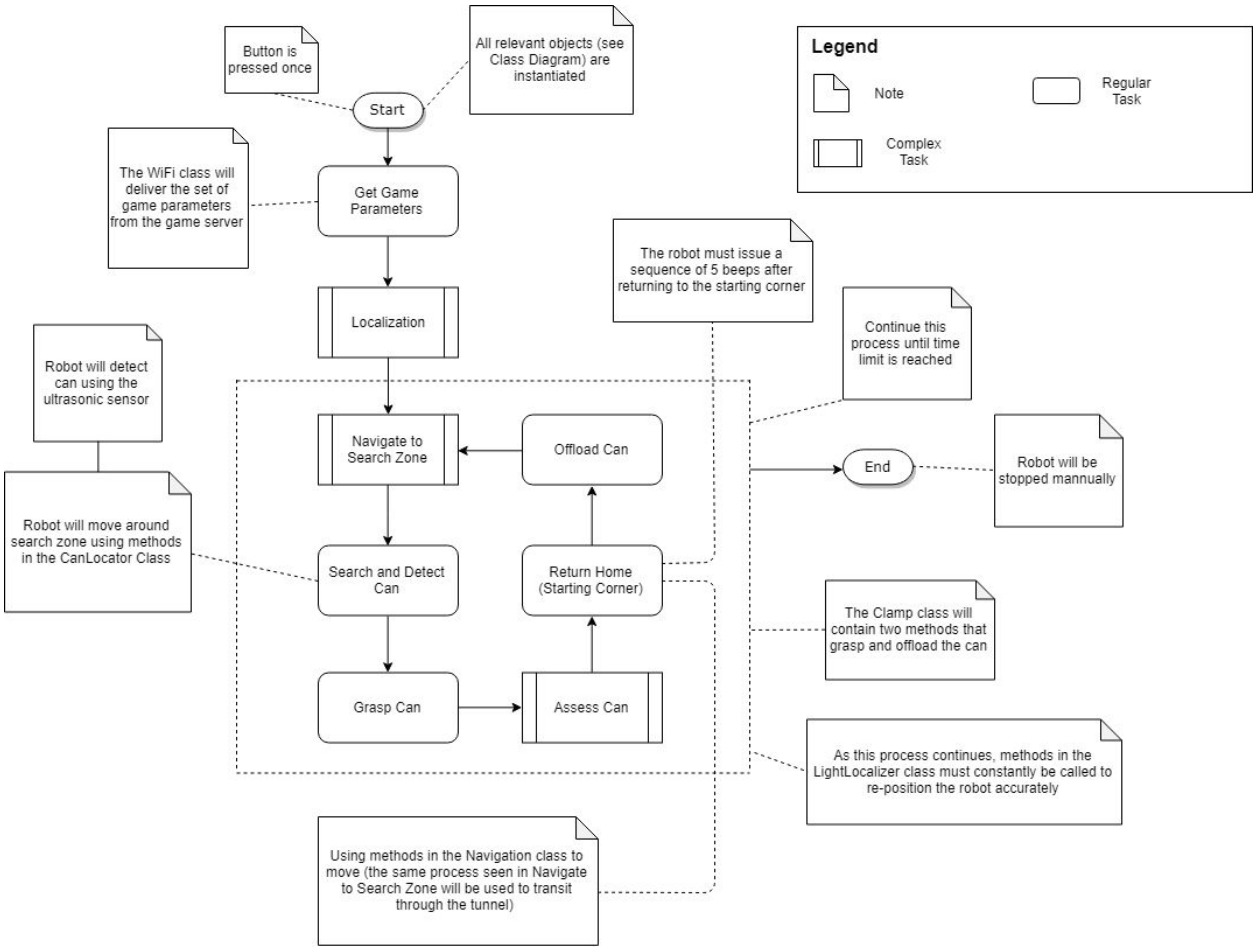

Figure 1: Overall System

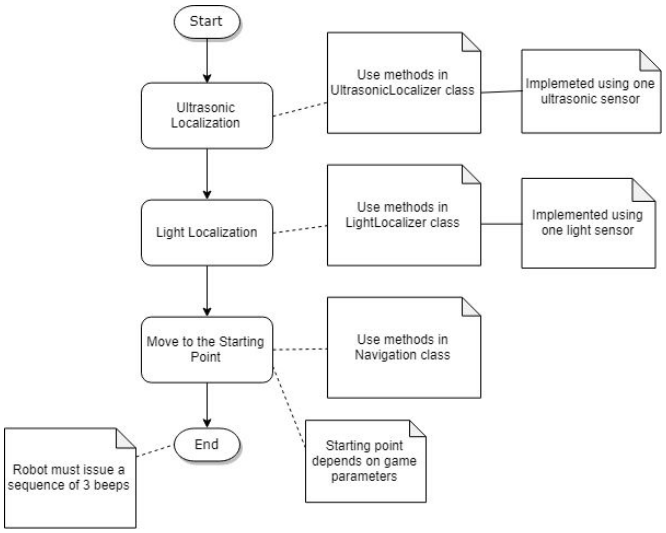

Figure 2: Localization

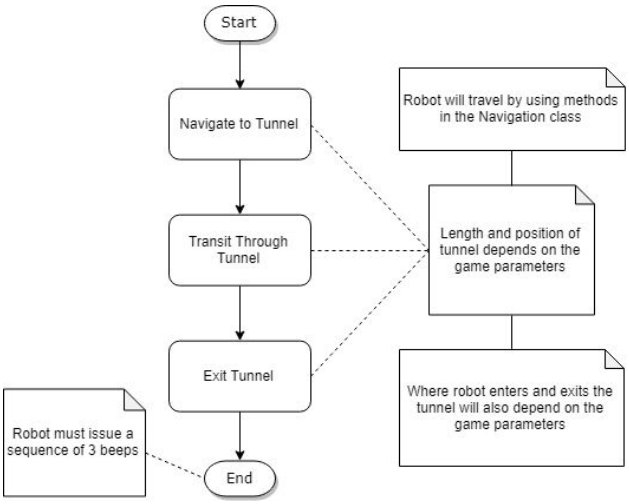

Figure 3: Navigate To Search Zone

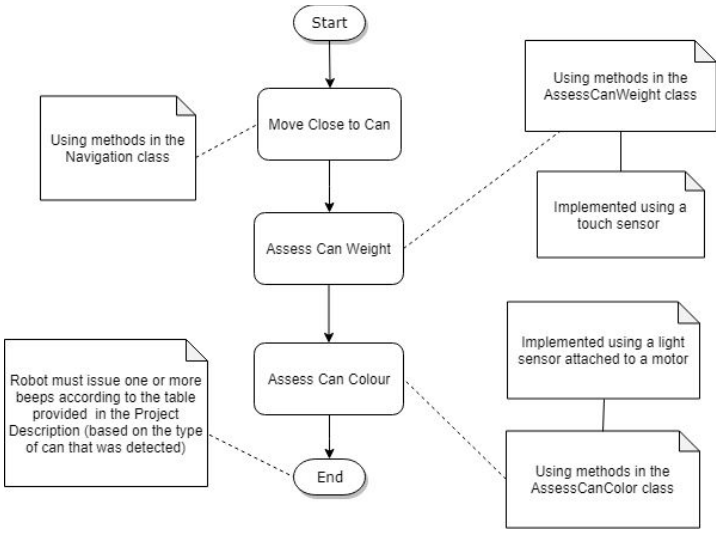

Figure 4: Assess Can

4.2 Class Diagram

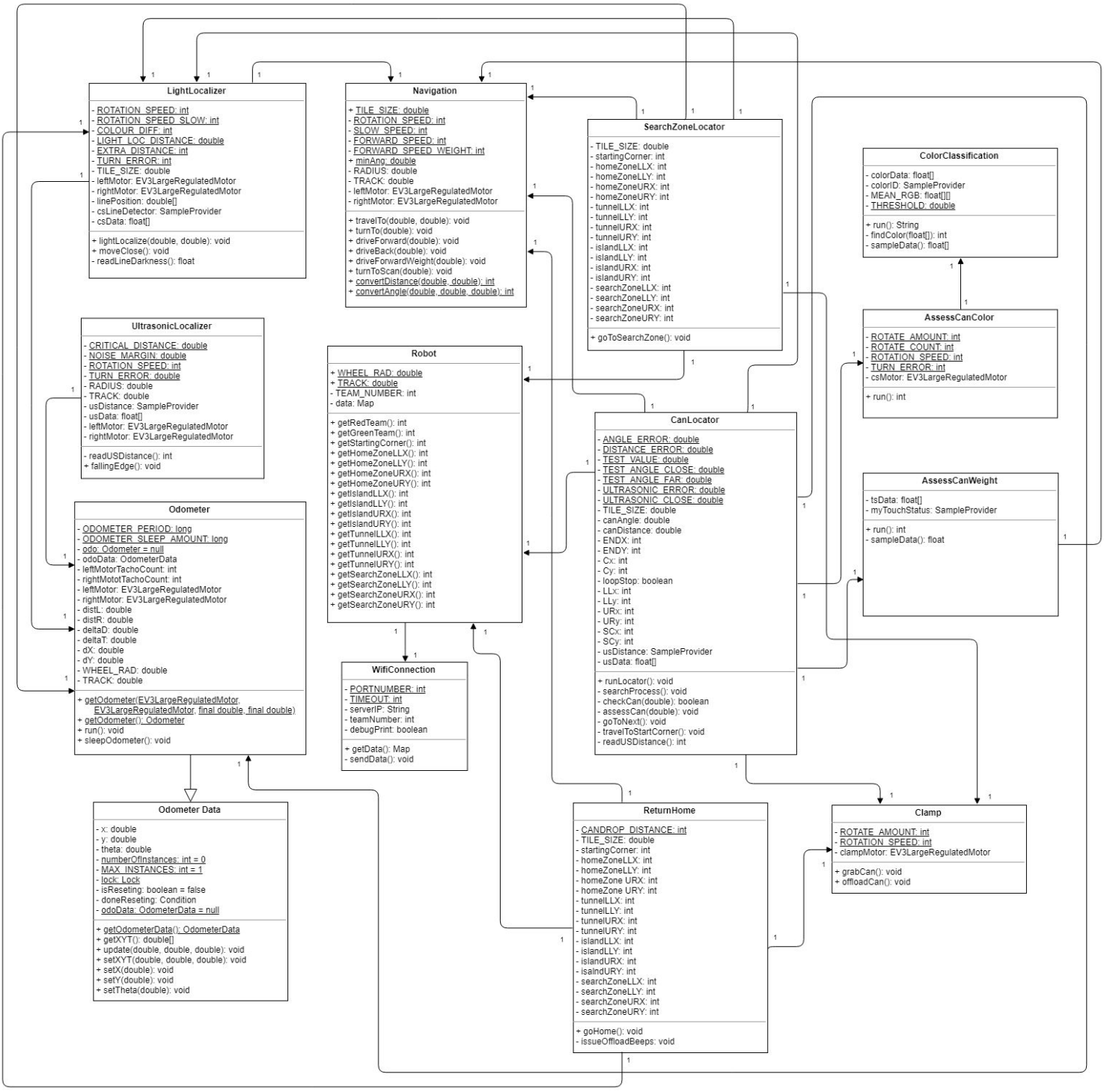

Figure 5: Class Diagram

4.3 Class Descriptions

4.3.1 AssessCanColor

The AssessCanColor class calls on methods from ColorClassification and uses the run() method to identity can colors by taking the mode of many samples. This method returns integers corresponding to the color of the can.

- **`public int run()`**
  The only method in the `AssessCanColor` class is the `run()` method. It contains two arraylists: `canColor` and `frequency`. The method begins by rotating the motor and calling another `run()` method from the `ColorClassification` class to fetch data from the front color sensor. As the motor rotates the sensor, the color scanned from each sample is stored into the `canColor` arraylist. Once sampling is done, the frequency of each color is checked. For instance, if blue was scanned 5 times, it will record that into the `frequency` arraylist, then it checks the number of times green was scanned and add it to the `frequency` list, and so on. Note that the exact frequencies are checked in the order of blue, green, yellow, and finally red (explained in the third paragraph).

  Once all frequencies are recorded, the `frequency` arraylist is passed into the `Collections.max()` method, which returns the maximum value contained in the arraylist. Finally, the `indexOf()` method is called on the `frequency` arraylist with the argument as the maximum value obtained, and then incremented by one (explained below).

  The way the PDF associated the integers with the colors is as follows: blue (1), green (2), yellow (3), and red (4). So since blue is checked first, it is added to index zero of the list. For simplification, assume blue had the highest frequency. The result of `indexOf()` will then be the index that contains this value, which is zero since blue is always the first color to be added to the list. The index given is then incremented by one `( 0 + 1 = 1)`. The method will therefore return one, which corresponds to blue as defined in the PDF.

4.3.2 AssessCanWeight

The AssessCanWeight class uses a method to identity can weight by receiving a signal from the touch sensor. If the can is heavy, the touch sensor will be triggered.

- **`public int run()`**
  The `run()` method calls the `fetchSample()` method, but casts its `float` value returned into an `int`. Casting the value fetched into an `int` makes it easier to compare with the choices of colors, since all color comparisons used in Lab 5 and the Beta Demo were compared with integers (i.e blue → 1, etc…).

- **`private float sampleData()`**

This method uses the `fetchSample()` method to see if the touch sensor has been triggered or not. The `fetchSample()` for a touch sensor will return a 1 if it has been triggered, otherwise it will return 0. Since the instance of the buffer is a `float` , the binary number will be casted into a `float` (i.e either 0.00 or 1.00).

4.3.3 CanLocator

The CanLocator class begins running once the EV3 has reached the lower left corner of the search zone. It calls on methods from other classes (AssessCanWeight, AssessCanColor, etc.) while running the search algorithm to assess the can color and weight. It then clamps the can and calls `goHome()` from the ReturnHome class to return to the starting zone.

- **`public void runLocator()`**

  This method is what runs the search algorithm of the code. It begins by initializing the start corner values (`SCx, SCy`) and the current coordinate values (`Cx, Cy`) to the lower left x and y respectively (see `goToNext()` method below for `Cx` and `Cy` explanation). Before the algorithm is run, the odometer values are compared to the upper right coordinates of the search zone, since the EV3 travels to either the upper right or the lower left depending on which corner is closest to the EV3 once it exits the tunnel. If the EV3 is indeed at the upper right, then the `SCx, Cx` are initialized to `URx`, and `SCy, Cy` are initialized to `URy`. The condition also calls `turnTo(180)` from the Navigation class, since the EV3 would be localized and facing away from the search zone at the upper right corner, as shown in *figure 6*. The algorithm is run using a `while` loop.

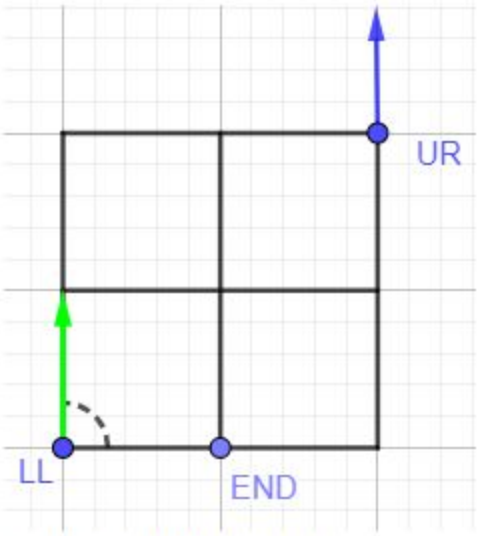

Figure 6: EV3 localized at UR (upper right of the search zone) with the blue vector showing the orientation of the robot after the localization.

Using *figure 6*, with the assumption of the EV3 starting at LL, the algorithm functions as follows: The EV3 travels on the border and clockwise around the search zone scanning each tile for cans. It scans 90º clockwise and checks for a can within that tile. If no can is detected, the EV3 heads to the next tile and checks for cans. Otherwise, it would assess the can scanned and return to its starting corner (lower left in this case).

A `while` loop begins to run with the complement of the `boolean loopStop`. `loopStop` is initially set to `false`, therefore allowing the loop to run. Once the EV3 finds and assesses a can, `loopStop` is set to `true` upon which the loop breaks, allowing `runLocator()` to return. Contained in the `while` loop are three conditions:

- `if(Cx == ENDX && Cy == ENDY)`

The first condition runs if no cans were found in the zone. The `if` condition checks whether or not the EV3 has arrived at the (`ENDX`,`ENDY`) coordinate (i.e if it traveled around an empty search zone). Note that with the assumption of the EV3 starting at the lower left corner:

```
ENDX = LLx + 1
ENDY = LLY
```

This is because the first tile would have already been scanned.

If the EV3 has completed its search, `travelTo()` and `turnTo()` from the Navigation class are called to drive the EV3 back to its start corner (lower left or upper right). `lightLocalize()` is called after every `travelTo()` call to keep the odometer values correct.

- `else if(!checkCan(90))`

If the EV3 has not reached the END coordinates, the complement of `checkCan()` is called as a condition to scan the current tile for cans. If `checkCan()` returns false, the `goToNext()` method will be called to drive the EV3 to the next tile.

- `else`

If a can has been detected, the `searchProcess()` method is called to assess and retrieve the can.

❖ **`private void searchProcess()`**

The `searchProcess()` method is called once a can has been detected. It calls on `assessCan()` with the argument `canDistance`, which is the distance required to travel in order to assess the can. The `canDistance` is the reading from the ultrasonic sensor subtracted by a `TEST_VALUE`, which was determined experimentally. The `TEST_VALUE` is subtracted because we do not want the EV3 to travel exactly to the can, but to have the can in the proximity of the clamping motor.

Once `assessCan()` returns, the EV3 uses the `driveBack()` method to return to its last intersection point, then calls `travelToStartCorner()` to begin its journey back to the starting zone.

- **`private boolean checkCan(double angle)`**
  The `checkCan()` method is called when an intersection of a tile is reached, and the EV3 is ready to search for cans. It returns `true` if a can has been spotted, otherwise returns `false`. The method takes in an argument `angle`, which is always inputted as 90º, since that is the angle required to fully scan a tile as shown below in *figure 7*.

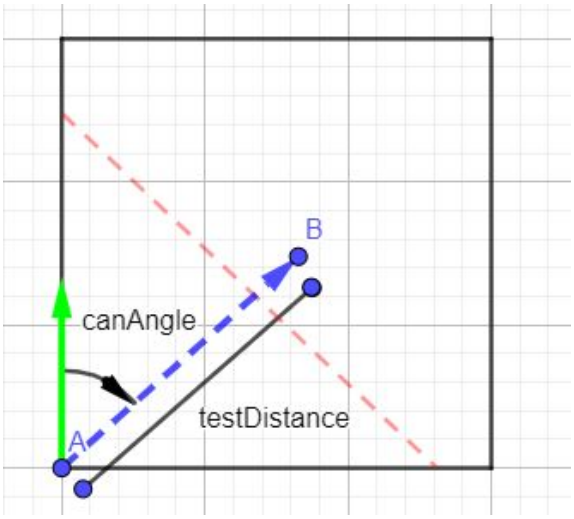

Figure 7: EV3 begins at point A facing the green vector. The EV3 scans clockwise until an object is detected or a 90º turn is completed. The can detected is shown at point B.

The method begins by resetting the `canAngle` to zero, then initializing `currentAngle` with the odometer angle. The method then calls `turnToScan()` with `angle` as an argument to begin scanning the tile. A `while` loop runs in parallel comparing the US distance measured with a tile size minus an error. This error is taken from a tile size because the US sensor protrudes in front of the point of rotation (as shown in *figure 7*), therefore, to assure it only scans within a tile, the distance from the center of rotation to the US sensor needs to be subtracted. If the distance returned from the US sensor is less than a tile, it is clear that a can has been detected. However, if the motors complete the 90º turn and then stop, the EV3 uses `turnTo()` with a negative 90º argument to fix its orientation before returning false.

Once a can is detected, the motors are halted and the measured distance is saved in `testDistance`, the distance the EV3 needs to travel to assess the can. Considering our US sensor placement (not exactly in the middle of the EV3) as well as the range of the sensor being a cone, angles of adjustments are required to

face the EV3 ahead of the can. Note that these values were determined through numerous tests.The `testDistance` is then compared to `ULTRASONIC_CLOSE` to determine whether the can spotted is in the first half of the tile (the red diagonal shown in *figure 7*). This is because the closer the can, the higher the angle of adjustment required. Therefore, if `testDistance` is greater, `TEST_ANGLE_FAR` is used as an argument for `turnTo()`, Otherwise, `TEST_ANGLE_FAR` is used.

Once adjusted, the `canAng` is calculated. In *figure 7*, the green vector is when the `currentAngle` was recorded, while the current odometer angle is taken at the blue vector (when the can is detected). The difference between them results in `canAng`, which is used to re-align the EV3 with the edge once the can has been grabbed and assessed. One more `if` statement checks if the difference is negative with a magnitude greater than 110º, then adds 360º to `canAng` so that the EV3 does not take the long turn around. This case was written keeping in mind the odometer angle might be off when at the first edge. For instance, if the `currentAngle` read 356º but the current odometer angle reads 30º, then the difference would be -326º. Adding 360º to the difference will yield 34º, which is then adjusted by subtracting `ANGLE_ERROR` (an experimentally determined value) from 34º. Finally, the method returns true.

- **`private void assessCan(double distance)`**
  This method clamps the can, then checks its color and weight. It begins by driving the EV3 closer to the can using the argument `distance` with the `driveForward()` method. The `grabCan()` method is called from the Clamp class to hold the can, then the EV3 drives back a distance $\delta = \frac{2}{3} * \texttt{TILE\_SIZE}$ using `driveBack()` to assess the weight. Note that `distance` $> \delta$, were δ was determined experimentally.

  To assess the weight, the EV3 uses `driveForwardWeight()` with the argument δ and runs a `while` loop that only breaks when both motors have stopped (i.e the EV3 has traveled the distance δ). In the loop, `run()` from the assessCanWeight class is called and the `int heavy` is initialized to its result. `heavy` is then repeatedly OR'd (using the bitwise operation) with its previous value and the `run()` value. This way, if the touch sensor detected a heavy can for at least one instance, the can is categorized as heavy. This conclusion was drawn after testing concluded that light cans can only trigger the touch sensor at unreasonably high speed.

  Since there are different beep variations for heavy and light cans, there is an `if` condition that checks whether the can was heavy or not. Once the weight is assessed, the respective switch statement calls `run()` from the assessCanColor class to determine the color of the can. The switch statement contains cases with varying numbers of beeps depending on the color of the can. The `playTone()`

method was used instead of a regular `beep()` to control the duration of the beep (heavier cans have longer duration).

- **`private void goToNext()`**
  This method uses `travelTo()` to drive the EV3 from one tile to the next around the search zone. It also calls `turnTo()` and `lightLocalize()` at the halfway point of each edge, and at the corners of the search zone to minimize odometer errors at all times.

  Before the `goToNext()` explanation proceeds, let us discuss what the `Cx` and `Cy` variables accomplish. Put simply, they keep track of the coordinates the EV3 is currently at. For instance, assume that the search zone is empty and the EV3 begins at the LL point shown by Figure 6 in the `runLocator()` method above. The `Cx` and `Cy` are initially set to `LLx` and `LLy` respectively. If the EV3 scans the first tile and no cans were detected, `Cy` is incremented by one, then the robot travels to that coordinate (which will be `(LLx,LLy+1)`). The EV3 will continue to travel upwards until `Cy = URy`, then `Cx` will begin to increment in the same manner. Once the EV3 reaches the `UR` corner, `Cy` will begin to decrement to travel down the third edge. Finally, the `Cx` will decrement when traveling on the fourth edge. After incrementing the `C` values, the EV3 checks if it has reached any of the four corners of the search zone. If so, it uses the `turnTo()` method to adjust the light sensor into the third quadrant (see `lightLocalize()` method), then calls `lightLocalize()` at that corner.

  The green vector represents the direction the EV3 is facing and the zero degree angle. The EV3 knows at which edge it's currently traveling on using the odometer angle reading:

  When close to 0^o, it is on the first edge:

```
odo.getXYT()[2] >= 360-ANGLE_ERROR || odo.getXYT()[2] <= 0+ANGLE_ERROR
```

  When close to 90^o, it is on the second edge:

```
odo.getXYT()[2] >= 90-ANGLE_ERROR && odo.getXYT()[2] <= 90+ANGLE_ERROR
```

  The same goes for the third and fourth edge. Note that checking for the first edge is a little different since the odometer overlaps the angle (goes from 359.9 to 0).

  The EV3 uses these edge conditions to check whether the middle of an edge has been reached (using: $(URx+LLx)\frac{1}{2}$) so that the EV3 localizes at that point too. If an edge is uneven in length (i.e 3 tiles), the EV3 would localize at the rounded value (i.e 2) for the second and fourth edge, and at the floored value (i.e 1) for the first and third edge. The `Math.round()` method is used to round the value, while flooring is done just by casing the the midpoint found into an `int`. The rounding is because the EV3 ends its search on the END corner (which is

(LLx+1,LLY)), so if the floored value was used, the EV3 would localize at the END point due to the `goToNext()`, then once more because it has reached the END point (included in the `runLocator()` method). To remove this redundancy, the second and fourth edge localization is shifted by one tile.

- **`private void travelToStartCorner()`**
  Once a can is clamped and assessed, `travelToStartCorner()` is called to drive the EV3 on and around the search zone border back to its start corner. As mentioned in the `checkCan()` method, `turnTo()` is used with the negative argument `canAng` to reorient the EV3 parallel to the edge it's currently situated on. The start corner is then compared with the upper right corner, since navigating to the lower left or upper right is not done the same way. For the explanations below, use *figure 8* as a reference.

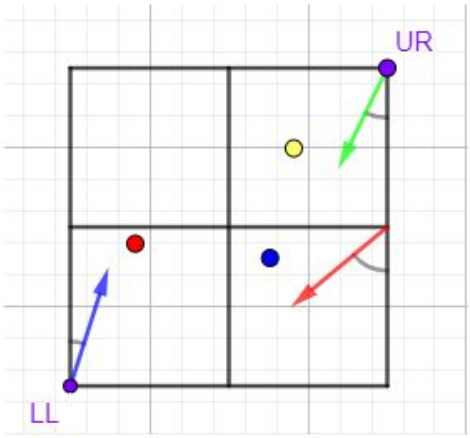

Figure 8: EV3 at different locations of the search zone.

If the start corner was the upper right corner, then the EV3 would be on the third edge of the zone before starting its search (i.e the angle of the odometer reads a value close to 180º) as shown in t*figure 8*. If a can is spotted from the red vector position (i.e at the second tile), the EV3 will assess it then call `turnTo()` and `lightLocalize()`. Once localized, the EV3 uses `travelTo()` to head to UR, then `turnto()` and `lightLocalize()` are called once more to correct the odometer values. Otherwise, if the can was spotted at the first tile (green vector position), there is no need for localizing twice since the EV3 is already at UR. Therefore, `turnTo()` and `lightLocalize()` are only called once. Finally, if a can was spotted on the first, second, or fourth edge, the EV3 would have to navigate to the zone corners until it has reached UR. For instance, if the EV3 is at the blue vector when it spotted the can, `turnTo()` and `lightLocalize()` are called, then `travelTo()` is used to navigate the robot to the upper left search zone coordinates. Once there, `turnTo()` and `lightLocalize()` are called again, then `travelTo()` is called with the arguments `URx` and `URy`. Finally, `turnTo()` and `lightLocalize()` are called to localize at UR.

Similarly, if the EV3 started at the lower left corner and a can was spotted at the first edge (i.e EV3 angle oriented at 0º), it would have two conditions. One, if the EV3 is at the first tile and a can is spotted, only localize once. Otherwise, the EV3 would localize at the intersection the can was assessed, then travel to the lower

left and localize once more. At the second, third, and fourth edge, the method would function with the same logic as explained in the previous paragraph. A combination of `turnTo()`, `lightLocalize()`, and `travelTo()` are used to navigate the EV3 on the border of the search zone to the lower left corner, then one final localization is done once at LL to correct the odometer. Once the EV3 arrives at the starting corner, `loopStop` is set to true so that the search algorithm ((`runLocator()` loop) is broken, and the `goHome()` method is allowed to run.

- **`private int readUSDistance()`**
  This method calls the `fetchSample()` method on the Sample Provider instance of the ultrasonic sensor, which stores the fetched data (in meters) into the buffer instance (`usDistance`). The sample distance is then extracted from the buffer, multiplied by 100 to convert it into centimeters, and finally, casted into an `int` before it is returned. If multiple samples are required, `readUSDistancecan()` can be called in a loop or as an argument of a loop depending on the requirement.

4.3.4 Clamp

Clamp class calls a method that uses a regulated motor to hold onto a can after it has been assessed by color/weight.

- **`public void grabCan()`**
  This method sets the speed and acceleration of the motor responsible for clamping, then uses the `rotate()` method to rotate it until it has held onto the can. The rotation angle and speed were experimentally determined.

- **`public void offloadCan()`**
  Similarly to `grabCan()`, the `offloadCan()` method rotates the motor with the same speed and acceleration, but opposite direction to unload the can.

4.3.5 ColorClassification

The ColorClassification class uses methods to fetch many samples and identifies the color of each sample using RGB values.

- **`public String run()`**
  The `run()` method calls both `findColor()` and `sampleData()` in order to determine the color of a can scanned. Once these methods return, the value returned is an `int` that is associated with a color. The value is then mapped to the color it corresponds to, and a `string` is returned with the name of the color identified.

- **`private int findColor(float[] colorData)`**

This method is called by the `run()` method to determine the color of the sample scanned. The buffer is passed as an argument, containing the RGB values of the sample. Before the RGB values are compared, they are normalized by dividing each R, G, and B value separately with the euclidean distance (`eucDistance`). These normalized values are then used to initialize the float variables `nR`, `nG`, and `nB` respectively. The `eucDistance` is determined using the formula below:

$$eucDistance = \sqrt{\alpha^2 + \beta^2 + \gamma^2}$$

$\alpha \rightarrow$ *Red value of sample RGB*
$\beta \rightarrow$ *Green value of sample RGB*
$\gamma \rightarrow$ *Blue value of sample RGB*

The method will then run a loop that normalizes the `MEAN_RGB` value of one color and subtract it from each normalized value (`nR,nG,nB`). The absolute value of the difference is then stored in the variables `deltaR`, `deltaG`, and `deltaB` respectively, and these delta values are compared with the THRESHOLD, which can also be seen as an error margin. If all three delta values fall below 35% error, the color compared is considered as the sample color identified, and an integer corresponding to that color is returned . Otherwise, the loop repeats using the `MEAN_RGB` of the next color, and if none of the colors match, an integer (4) that is not corresponding to any color is returned.

- **`private float[] sampleData()`**
  This method calls the `fetchSample()` method on the Sample Provider instance of the color sensor, which stores the fetched data (light intensity) in the buffer instance (`colorData`). This sample is later used to be compared to the `MEAN_RGB` value that was experimentally determined in order to determine the color scanned.

4.3.6 LightLocalizer

The LightLocalizer class calls on its methods to reorient itself to face the zero degree angle as defined in the coordinate plane, as well as correct the odometer (x,y) values.

- **`public void lightLocalize(double pointX, double pointY)`**
  `lightLocalize()` uses a color sensor that runs in red mode (measures light intensity reflected back) to detect the gridlines of the playfield in order for the EV3 to reorient itself at the zero degree angle and correct the odometer x and y values to the arguments `pointX` and `pointY` respectively. *Figure 9* shows an example of how the localization would function. Points A and B represent the position of the light sensor and the center of rotation of the EV3 respectively. Segment AB is the `LIGHT_LOC_DISTANCE`, the distance measured from the

center of rotation (the center of the wheel base) to the light sensor in the back. Note that the zero angle is in the direction of the y axis (blue vector with point C).

In order for the EV3 to localize correctly, the robot's light sensor must be in the third quadrant, since that would only require the use of one formula rather than the need to create multiple cases depending on the sensor's position. As shown below in *figure 9*, the light sensor (point A) is indeed in the third quadrant.

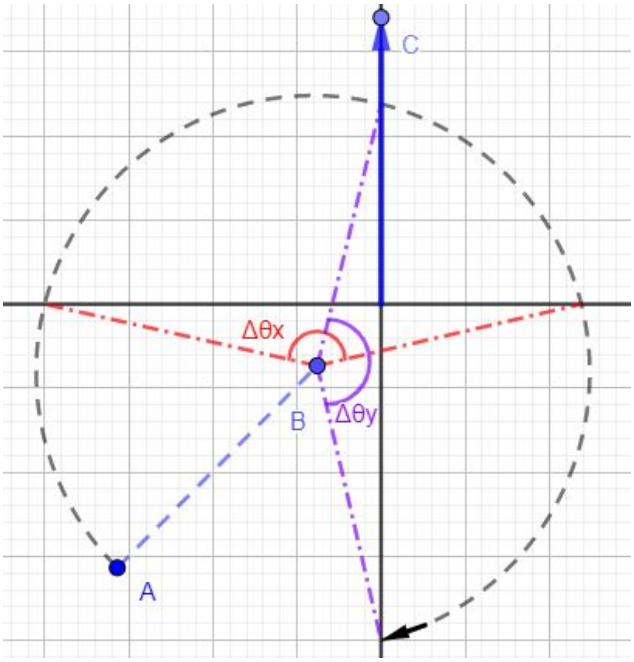

Figure 9: Light Localization x and y correction.

The method begins by calling `readLineDarkness()` once and stores its result in `firstReading`. Since the EV3 is oriented around the 45° line of the third quadrant, then the sample reading is the tile color of the playfield. It will be used later to determine whether a gridline has been crossed or not. The method then runs a loop that would rotate the EV3 clockwise at a relatively high speed while recording angles in the `linePosition` array at every grid line detection. A line is detected if the percent error of the current light intensity measured and the `firstReading` is greater than 35%. Once the third grid line has been detected, the motors are set to lower speeds so that when the fourth line is detected, the motors are halted and the EV3 does not stop too far from its intended orientation (at the fourth grid line).

Using *figure 9*, we can see that the angles stored, when the first and third lines were detected, belong to $\Delta\theta_x$ (i.e taking their difference will result in $\Delta\theta_x$). Similarly, the second and fourth belong to $\Delta\theta_y$. These two angles (`angleX` $\rightarrow \Delta\theta_x$ and `angleY` $\rightarrow \Delta\theta_y$) are then used to calculate the x and y changes using the following formulas (can be derived from the diagram above):

$$\Delta x = -LIGHT\_LOC\_DISTANCE * cos(\frac{\Delta\theta_y}{2})$$
$$\Delta y = -LIGHT\_LOC\_DISTANCE * cos(\frac{\Delta\theta_x}{2})$$

Calculating the angle to fix the orientation is also given by (assuming no turn error):

$$\texttt{deltaA} = 90^o - \frac{\Delta\theta_y}{2}$$

To prove the above the formula, we use *figure 10* below. Note that the blue vector represents the final orientation of the EV3 (i.e at the fourth grid line), and the following hold: $\Delta\psi = \frac{\Delta\theta_y}{2}$ and $\texttt{deltaA} = \Delta\varphi_1 = \Delta\varphi_2 = 90^o - \Delta\psi$. For simplicity, let $\Delta\theta_y = 120^o$.

If we apply the above formula:

$$\Delta\psi = \frac{\Delta\theta_y}{2} = \frac{120}{2} = 60^o$$
$$\Delta\varphi_1 = 90^o - \Delta\psi = 30^o$$

From *figure 10*, it is clear that adding the correction $\Delta\varphi_1$ to the current position of the EV3 (center of rotation at point B and facing the blue vector), it will correct its angle so that the EV3 is parallel to the y axis.

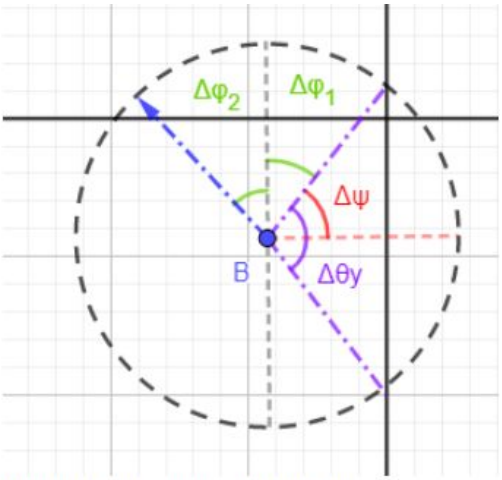

Figure 10: Light Localization with EV3 center of rotation at point B.

Once `deltaA` is calculated, the `turnTo()` method is called with that angle as the argument to correct the orientation. The odometer `setXYT()` method is called with the arguments `deltaA, pointX,` and `pointY` with the Δx and Δy values added to their respective variables. Note that `pointX` and `pointY` have to be multiplied by `TILE_SIZE` before the addition with their respective Δ values since they are in coordinate units and not distance (odometer runs on centimeters!). `travelTo()` is then called with the arguments `pointX` and `pointY` so that the EV3 travels to the localized (x,y) coordinate. Since the EV3 always travels to the point at an angle, the `turnTo()` method must be called again with the negative argument of the `minAng` used in `travelTo()` so that the EV3 is aligned parallel with the zero degree angle. The odometer then uses the `setXYT()` once more before the motors are halted.

- **`public void moveClose()`**

When the EV3 is first sent the coordinates of the zones in the playfield, it uses the ultrasonic sensor to localize close to the zero degree angle. Once it has returned, the EV3 needs to be close enough to detect four gridlines around the point it is trying to localize to, as well as make sure that the light sensor in the back is positioned in the third quadrant (see method above). The `moveClose()` method is only used during that initial localization process. It rotates the EV3 45º using the `turnTo()` method, then drives it forward using `convertDistance()` method from the navigator class and the `rotate()` motor method.

- **`private float readLineDarkness()`**
  The `readLineDarkness()` method returns a scaled value of the sample light intensity reflected back. It uses the sensor mounted at the back of the EV3 to detect the playground gridlines when localizing. The method calls `fetchSample()` on the instance of the sample provider and saves it in the buffer `csData`. It then returns the the sample from the buffer, but scaled by 1000 for accurate comparisons when needed.

4.3.7 Navigation

The Navigation class contains methods that help the EV3 navigate around the playfield.

- **`public static int convertDistance(double RADIUS, double distance)`**
  `convertDistance()` takes in the wheel radius of the EV3, which was measured, and the distance the EV3 needs to travel. It uses these values by taking the ratio $\frac{distance}{RADIUS}$ and multiplying it by $\frac{180^o}{\pi}$ to convert the distance into degrees for the motors to use. The answer is casted into an `int` before returning the method.

  Note that both `convertDistance()` and `convertAngle()` return integers, which is the main cause of the EV3's inaccuracy. The values that are used for calculation are all `double`, but are truncated into `int` since the motors methods only take `int` values for speed.

- **`public static int convertAngle(double RADIUS, double width, double angle)`**

  Equations:

  $\alpha \rightarrow arclength = r \times \theta$

  $r \rightarrow$ *radius of wheel base*
  $\theta \rightarrow$ *angle in radians*
  $\phi \rightarrow$ *angle in degrees*

  $$\beta \rightarrow \frac{\pi \times width \times \phi}{360^o}$$

`convertAngle()` is mainly used because the EV3 designs always change. Note that the method takes in `RADIUS` (measured radius of wheels), `width` (the measured length of the wheelbase from one center of the wheel to the other), and `angle` (the angle desired for rotation). Clearly, as the wheelbase of the EV3 increases (i.e the distance between the two motors increases), turning will require the EV3 motors to travel a larger distance (or more rotations). Therefore, this method is used to convert the angle required to turn into a distance that depends on the wheelbase, and finally back to an angle that coincides with the design of the EV3.

The method passes equation β, which is in the form of a distance (see next paragraph), as an argument to `convertDistance()`. `convertDistance()` will return the amount of rotation in degrees required to travel that distance (β). Finally, `convertAngle()` returns that same angle that was returned by `convertDistance()` casted as an `int`. Note that the EV3 is subject to errors also due to this method truncation from `double` to `int`.

Shown below is how β is in the form of a distance that each wheel need to travel. Note that ϕ *and* θ are in different units (see equations above).

$$\beta = \frac{\pi \times width \times \phi}{360^{o}} = \frac{\pi \times 2 \times r \times \phi}{360^{o}} = \frac{\pi \times r \times \phi}{180^{o}} = r \times \theta = arclength$$

- **`public void travelTo(double x, double y)`**

`travelTo()` is used to travel to a certain (x,y) coordinate, where `x` and `y` are given as arguments. First, the method saves the current odometer readings of x,y, and θ into temporary values (`currentX`, `currentY`, and `currentA`). Note that the (x,y) saved values of the odometer reading are in centimeters and <u>not in tile units</u> (1 coordinate unit = 1 tile = 30.48 cm). Next, `deltaX` and `deltaY` are computed by subtracting the the initial values (the saved `currentX` and `currentX` values) from the final values (the argument values). Keep in mind the final `x` and `y` values are scaled into centimeters before the subtraction is done, since scaling the coordinates into centimeters will allow for precise results from calculation as well as the use of the `driveForward()` method (called at the end of `travelTo()`) requires a distance in centimeters to travel. Therefore, both `delta` variables contain the Δx and Δy values in centimeters.

Then, the angle the EV3 will be required to turn to is calculated in the following lines below:

```
minAng = -currentA + Math.atan2(deltaX, deltaY) * 180 / Math.PI;

if (minAng < 0 && (Math.abs(minAng) >= 180)) {
        minAng += 360;
} else if (minAng > 0 && (Math.abs(minAng) >= 180)){
        minAng -= 360;
```

```
}
```

currentA is subtracted so that the EV3 aligns itself with the 0^o angle. The `Math.atan2()` method then computes the angle (`minAng`) and multiplies it by $\frac{180^o}{\pi}$ to convert radians into degrees. However, `minAng` is not in the range of $-180^o \leq minAng \leq 180^o$. Meaning, the EV3 might end up taking a longer turn to get to the orientation required. Therefore, if the angle is less than or equal to -180^o, add 360^o degrees. Otherwise, if the angle is greater than or equal to 180^o, subtract 360^o degrees. These conditions will allow the EV3 to turn a smaller angle. For instance, if the angle calculated was -270^o, then the `if` condition would run, and the `minAng` will now be 90^o.
Note that `Math.atan2()` was used instead of the regular `Math.atan()` since the regular arctan method does not account for the second and third quadrant. For instance, if `Math.atan()` was used on two values that should return 135^o, it would return -45^o.

Once `minAng` is calculated, `turnTo()` is called with the `minAng` argument. Then, the `distance` local variable is initialized to the result of `Math.hypot()`, which is called on the `deltaX` and `deltaY` variables to calculate the distance required to travel to the point (`x,y`). Finally, `distance` is passed as an argument to `driveForward()`, which drives the EV3 to the (x,y) coordinates.

- **`public void turnTo(double theta)`**

`turnTo()` takes in an angle as argument and rotates the EV3 that many degrees. Using the `convertAngle()` method, it converts the angle given in degrees to the correct angle required to turn based on the wheelbase (see `convertAngle()` method above). It then passes the converted angle as an argument for both `motor.rotate()` methods (one for each wheel). Since the convention for a positive angle is clockwise, the left wheel rotates forward and the right wheel rotates backwards. A negative argument will switch the direction of rotation (i.e the EV3 will rotate counter clockwise).

The clockwise direction chosen to be positive mean that the first motor called to rotate is the left one. A `true` argument is passed into the `rotate()` method along with the angle, while the right motor argument is `false`. These `boolean` arguments, when set to `true`, allow the motor thread to run in parallel with the code it precedes. For instance:

```
leftMotor.rotate(45, false);
rightMotor.rotate(-45, false);
```

If the EV3 runs on the lines written above, it will rotate the left motor 45^{o} first, then stop the left motor and begin to rotate the right motor 45^{o}. This is known as "blocking". The correct way of implementation used is:

```
leftMotor.rotate(45, true);
rightMotor.rotate(-45, false);
```

This will allow the left motor to run simultaneously with the right one. Note that since the right motor is blocking, the code below the right motor will not run until the motor has stopped.

- **`public void turnToScan(double theta)`**
  Unlike the `turnTo()` method, `turnToScan()` uses the exact same code, with the exception of the second motor also passed a `true` argument rather than a `false`. `turnToScan()` is used only during the searching process, since the EV3 has to rotate while reading data from the Ultrasonic (US) sensor at the same time. In order for the EV3 to scan for cans as it rotates 90^{o}, the sensor has to begin running instantly after the motors begin their rotation. This is possible when both motors are passed `true` arguments, allowing the motor threads to run in parallel with the rest of the code. Hence, both motors rotate simultaneously as the US sensor fetches the data.

- **`public void driveForward(double distance)`**
  This method allows the EV3 to drive forward a certain distance. It takes the `distance` argument and converts it into an angle for rotation using `convertDistance()`. `driveForward()` uses the similar lines of code as the `turnTo()` method, but note that the direction of rotation has changed. The left motor rotates forward as in the previous method, however the right motor now also rotates forward allowing the EV3 to drive in the forward direction. The right motor is also blocking using the `false` argument given as in the `turnTo()` method. This allows for the movement to finish before proceeding to the rest of the code.

- **`public void driveBack(double distance)`**
  This method functions the exact same as the `driveForward()` with the exception of both motors rotating backwards rather than forwards. I.e the motors have negative `rotate()` arguments for the distance.

- **`public void driveForwardWeight(double distance)`**
  `driveForwardWeight()` is used when a can is detected. It uses the same code as the `driveForward()` method with the exception of both motors passed the argument `true`. This allows the EV3 to run the motor threads in parallel with the loop of the touch sensor that is fetching data.

4.3.8 Odometer

The Odometer class contains a thread that constantly updates the (x,y) values of the EV3 as it navigates around the playfield.

- **`public void run()`**

  The `run()` method of the odometer uses a `while(true)` loop to constantly calculate and update the distance the EV3 travels using the tacho count of the wheels, as well as its orientation in space. It is one of the only threads running in parallel along with the other motor threads that drive the EV3 forward.

  The method begins by getting the tacho counts of the left and right motors, then storing them in separate variables. These tacho counts are retrieved using the `getTachoCount()` method, which returns an int angle. The distance traveled by each wheel is then calculated using the formula below:

$$dist = motorTachoCount * r \frac{\pi}{180}$$

  $dist$ is calculated twice (as `distL` and `distR`, since there are two wheels) with each calling their respective $motorTachoCount$. Radius r belongs to the wheel and is measured from its center to the edge of the rubber tire. Note that the $\frac{\pi}{180}$ is multiplied in order to convert the arclength ($motorTachoCount * r$) into distance, since the $motorTachoCount$ is in degrees. Once `distL` and `distR` are calculated, the average of both is taken and stored in `deltaD`. The change in orientation is calculated by subtracting the `distR` from `distL`, then dividing the result by the `TRACK`, and storing it in `deltaT`. The order of subtraction is important, since the convention for an increasing angle is clockwise. Therefore, $\Delta distance = distL - distR$.

  Let dX, dY, deltaT, deltaD, be Δx, Δy, $\Delta\theta$, Δd respectively. *Figure 11* shows a scenario of how the x and y values are calculated (note that the direction of x and y are written with respect to the convention defined in the Final Design PDF). Since Δd and $\Delta\theta$ have been calculated, then using trigonometry yields:

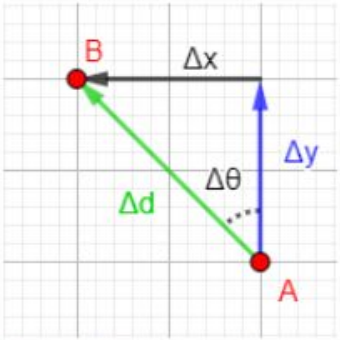

Figure 11: Odometer calculation when EV3 starts at point A. Orientation is shown by the blue vector and direction Of heading is shown by the green vector.

$$\Delta x = \Delta d * sin\,(current\ odometer\ angle + \Delta\theta)$$
$$\Delta y = \Delta d * cos\,(current\ odometer\ angle + \Delta\theta)$$

Note that the odometer angle is displayed in degrees, so when calculating Δx *and* Δy, the current odometer angle is passed as an argument to `Math.toRadians()` before the addition with $\Delta\theta$. Once Δx and Δy are calculated, $\Delta\theta$ is converted into degrees using `Math.toDegrees()` and all three variables (Δx, Δy, $\Delta\theta$) are passed as arguments to the `update()` method.

The `update()` method from the OdometerData class is called on the odometer instance to add the small changes in values to the current odometer values. The `resetTachoCount()` method is called once `update()` returns so that the next iteration does not use an accumulated number of revolutions to update the odometer. Note that the method is `void`, therefore returns nothing.

4.3.9 OdometerData

The OdometerData class helps the EV3 keep track of its current position and angle on the playfield. It imports the Lock and Condition classes to halt all threads if any set method is called.

- **`public double[] getXYT()`**
  The `getXYT()` allows the utilization of the odometer values x, y, and the angle. The method declares an array of three doubles and stores the `x, y`, and `theta` values into the array respectively, upon which the array is returned.

- **`public void update(double dx, double dy, double dtheta)`**
  This is the method used by the odometer to keep track of its position in space. The `set()` methods are only used to correct the odometer. `update()` adds the small changes Δx, Δy, $\Delta\theta$ (`dx, dy, dtheta`) to the current `x, y`, and `theta` values. The method is called in the thread of the odometer, allowing the EV3 to constantly update its position and orientation as it drives around.

- **`public void setXYT(double x, double y, double theta)`**
  This method sets the odometer current `x, y`, and `theta` values to the `x, y`, and `theta` arguments inputted.

- **`public void setX(double x)`**
  This method sets the current x value (in centimeters) of the odometer to the argument `x` given.

- **`public void setY(double y)`**
  This method sets the odometer current y value (in centimeters) to the argument `y` given.

- **`public void setTheta(double theta)`**

This method sets the odometer angle `theta` (in degrees) to the argument `theta` given.

4.3.10 ReturnHome

Once the can search and can assessment are complete, the `goHome()` method in this class guides the robot back to the starting corner, and signals when to unload the can. The class diagram above (see Section 3.2) shows that the ReturnHome class is associated with the Robot, Navigation, LightLocalizer, and Clamp classes. Like the SearchZoneLocator class (see Section 3.3.12), the game parameters (e.g., coordinates of starting zone and coordinates of tunnel) must be delivered to the robot in order for it to navigate the play field correctly. This process is done in the constructor of this class, which calls the methods found in the Robot class (see Section 3.3.11).

- **`public void goHome()`**
  Similar to the SearchZoneLocator class, since the robot can theoretically be associated to any of the four starting corners, there exists no universal algorithm that can guide the robot back to its initial starting point. Therefore, four different cases must be accounted for (one for each corner), which is what is done in this method.

  First, methods from the Navigation class (see Section 3.3.7) are called in this method to move the robot on the playing field. These methods enable the robot to travel through the tunnel and return to its starting corner.

  Then, when the robot reaches its starting corner, it calls the `issueOffloadBeeps()` method before offloading the can to signal that it has returned to its starting corner.

  Finally, in order to offload the can, the robot travels to the inside of its starting corner using methods in the Navigation class and releases the clamp. The clamp is released by calling the `offloadCan()` method in the Clamp class (see Section 3.3.4).

  Note: As the robot travels on the playing field, the `lightLocalize()` method found in the LightLocalizer class (see Section 3.3.6) must be called to correct the odometer's readings.

- **`public void issueOffloadBeeps()`**
  This method calls the static method `beep()` found in the Sound class five times to issue five beeps.

4.3.11 Robot

The Robot class contains methods that fetch and access the specified game parameters (e.g., starting corner) associated with our robot. By creating an instance of the `WifiConnection` class and calling the `getData()` method, we are able to save the

entire collection of game parameters delivered by the game server to an instance of the `Map` class (implemented in the constructor of the `Robot` class). Then, the methods created in the Robot class can access this `Map` instance and get the correct game parameters associated with our robot (implemented by comparing our assigned team number to the team numbers linked to the *RedTeam* and *GreenTeam* game parameters).

- **`public int getRedTeam()`**
  This method returns the team number associated with the *RedTeam*.

- **`public int getGreenTeam()`**
  This method returns the team number associated with the *GreenTeam*.

- **`public int getStartingCorner()`**
  This method returns the starting corner associated with the team's assigned color (i.e., *RedTeam* or *GreenTeam*).

- **`public int getHomeZoneLLX()`**
  This method returns the x-coordinate of the starting zone's lower left corner. The starting zone (*Red* or *Green*) depends on the team's assigned color.

- **`public int getHomeZoneLLY()`**
  This method returns the y-coordinate of the starting zone's lower left corner. The starting zone (*Red* or *Green*) depends on the team's assigned color.

- **`public int getHomeZoneURX()`**
  This method returns the x-coordinate of the starting zone's upper right corner. The starting zone (*Red* or *Green*) depends on the team's assigned color.

- **`public int getHomeZoneURY()`**
  This method returns the y-coordinate of the starting zone's upper right corner. The starting zone (*Red* or *Green*) depends on the team's assigned color.

- **`public int getIslandLLX()`**
  This method returns the x-coordinate of the island's lower left corner.

- **`public int getIslandLLY()`**
  This method returns the y-coordinate of the island's lower left corner.

- **`public int getIslandURX()`**
  This method returns the x-coordinate of the island's upper right corner.

- **`public int getIslandURY()`**
  This method returns the y-coordinate of the island's upper right corner.

- **`public int getTunnelLLX()`**

This method returns the x-coordinate of the tunnel's lower left corner. The tunnel (*Red* or *Green*) depends on the team's assigned color.

- **`public int getTunnelLLY()`**
  This method returns the y-coordinate of the tunnel's lower left corner. The tunnel (*Red* or *Green*) depends on the team's assigned color.

- **`public int getTunnelURX()`**
  This method returns the x-coordinate of the tunnel's upper right corner. The tunnel (*Red* or *Green*) depends on the team's assigned color.

- **`public int getTunnelURY()`**
  This method returns the y-coordinate of the tunnel's upper right corner. The tunnel (*Red* or *Green*) depends on the team's assigned color.

- **`public int getSearchZoneLLX()`**
  This method returns the x-coordinate of the search zone's lower left corner. The search zone (*Red* or *Green*) depends on the team's assigned color.

- **`public int getSearchZoneLLY()`**
  This method returns the y-coordinate of the search zone's lower left corner. The search zone (*Red* or *Green*) depends on the team's assigned color.

- **`public int getSearchZoneURX()`**
  This method returns the x-coordinate of the search zone's upper right corner. The search zone (*Red* or *Green*) depends on the team's assigned color.

- **`public int getSearchZoneURY()`**
  This method returns the y-coordinate of the search zone's upper right corner. The search zone (*Red* or *Green*) depends on the team's assigned color.

4.3.12 SearchZoneLocator

This class contains the method that guides the robot to the tunnel, through the tunnel, and to the lower left corner/upper right corner of the search zone. The class diagram above (see Section 3.2) shows that the SearchZoneLocator class is associated with the Robot, Navigation, LightLocalizer, Odometer and Clamp classes. In order to navigate around the play field correctly, the game parameters (e.g., coordinates of starting zone and coordinates of tunnel) must be delivered to the robot. This process is done in the constructor of this class, which calls the methods found in the Robot class (see Section 4.3.11).

- **`public void goToSearchZone()`**
  Since, in theory, the robot can start from any corner, there exists no universal algorithm that can guide the robot to the tunnel and to the search zone. Therefore,

four different cases must be accounted for (one for each corner), which is what is done in this method.

First, for each case, the `setXYT()` method (see Section 3.3.9) found in the Odometer class must be called because, following its initial localization, the robot may face different directions (i.e., North, South, West or East) depending on its starting corner. Furthermore, when this process is done, the clamp must be closed by calling the `offloadCan()` method in the Clamp class (see Section 3.3.4). In doing so, the robot will be able to traverse through the tunnel without any issues.

Then, methods from the Navigation class (see Section 3.3.7) are called in this method to move the robot on the playing field. These methods enable the robot to travel through the tunnel and to the search zone.

Finally, when the robot reaches the lower left corner/upper right corner of the search zone, the clamp is reopened by calling the `grabCan()` method in the Clamp class in preparation for the can grabbing.

Note: As the robot travels on the playing field, the `lightLocalize()` method found in the LightLocalizer class (see Section 3.3.6) must be called to correct the odometer's readings.

4.3.13 UltrasonicLocalizer

This class contains a method that runs only at the start corner to align the EV3 with the zero angle defined in the coordinate plane of the playfield.

- **`public void fallingEdge()`**
  This method can only be used when the EV3 is in between two walls (see *figure 12*). It uses a falling-edge detection procedure to reorient the EV3 as close to the zero degree angle following the convention on the coordinate grid (0^o is set at the positive y-axis).

  The method begins by declaring three angles (`angleA`, `angleB`, `turningAngle`) and then using the following lines of code:

```
while (readUSDistance() < CRITICAL_DISTANCE + NOISE_MARGIN){
      leftMotor.forward();
      rightMotor.backward();
}

while (readUSDistance() > CRITICAL_DISTANCE){
      leftMotor.forward();
      rightMotor.backward();
}
```

  The first `while` loop will spin the EV3 clockwise so that the Ultrasonic (US) sensor is facing outward (i.e away from the wall), since the

`readUSDistance()` method will return a value less than the `CRITICAL_DISTANCE` when facing the wall. Due to the US sensor containing lots of spikes in the sample data fetched, the `NOISE_MARGIN` is used to assure that the EV3 US sensor has exceeded the `CRITICAL_DISTANCE` range so that no errors occur while scanning for the next falling edge. Once the US sensor faces outward, the next loop will rotate the EV3 clockwise until the first falling edge is detected. `angleA` will then be initialized to the current odometer angle reading, and the EV3 will use two loops similar to the ones shown above, however, the motor directions are reversed so that the EV3 rotates counter clockwise. Once the second falling edge is detected, the motors are halted and `angleB` is initialized to the current odometer reading.

In order to apply the correction turn to the EV3, both recorded angles have to be compared with one another. This is because the correction turn applied to the EV3 when it starts facing the wall would differ if the EV3 started localizing away from the wall (i.e the magnitude and direction would differ). The two conditions are:

```
        if (angleA < angleB) {
                turningAngle = (360 - angleB) + ((angleA + angleB) / 2) -
225 + TURN_ERROR;
        } else if (angleA > angleB) {
        turningAngle = -angleB + (angleA + angleB) / 2 - 45 + TURN_ERROR;
        }
```

The difference is when `angleA < angleB`, a 180º shift is added. It is easier to see why the shift it done with a sample calculation (see example below). Finally, the EV3 uses the `turnTo()` with the argument `turningAngle`, then sets the odometer angle to zero.

Figure 12 below shows an example of how the US localization works. Although multiple assumptions are made here to make explanation and understanding trouble-free, many tests were conducted in order to determine constants such as `CRITICAL_DISTANCE`, `NOISE_MARGIN`, and `TURN_ERROR`. To make it simple, the EV3 is positioned facing the zero degree (red vector) along with the `CRITICAL_DISTANCE` (segments CD and BD) initialized to a value such that the US sensor will be perpendicular to the wall (facing it exactly) when sensing that distance. Theoretically, the EV3 should end up at the same position once it has run the code, since that is the zero degree convention.

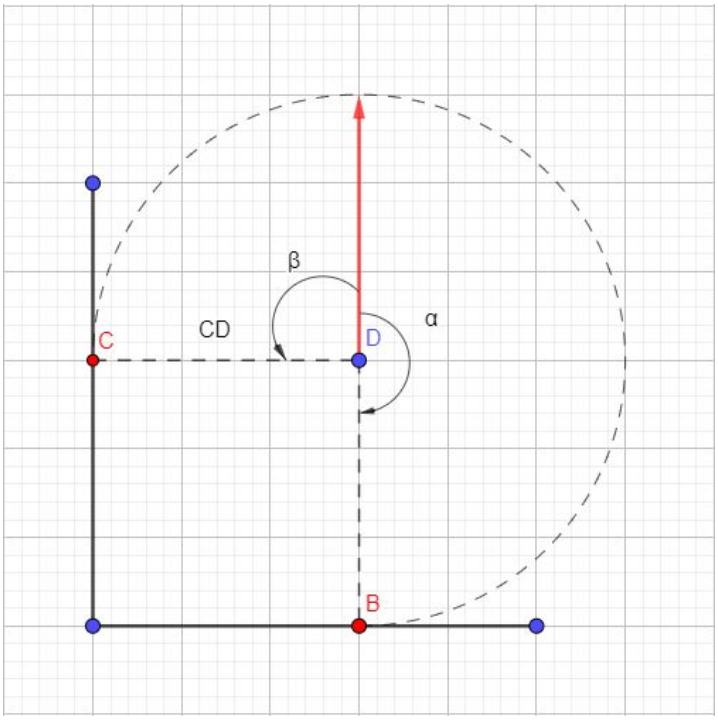

Figure 12: Ultrasonic Localization

The first angle to be recorded will be α, which we can easily conclude to be 180^o. Next, the EV3 will spin the other way and calculate angle $\beta = 360 - 90 = 270^o$. Using the conditions stated above, and assuming there is no turn error:

$$\alpha < \beta \Rightarrow turningAngle = (360 - \beta) + \frac{\alpha+\beta}{2} - 225 = 90^o$$

Note that once β was recorded, the motors are halted. Therefore, the correction of `turningAngle` is applied from that point (point C in this example). Clearly, from *figure 12*, applying a 90^o turn will bring the EV3 back to the zero angle position. If the $\alpha > \beta$ condition was used, the result for `turningAngle` would be -90º, which would place the EV3 facing the opposite direction of the red vector (an exact shift of 180º).

- **`private int readUSDistance()`**
  Refer to Section 4.3.3.

5.4 Sequence Diagrams

4.4.1 Navigation Class - travelTo(double, double)

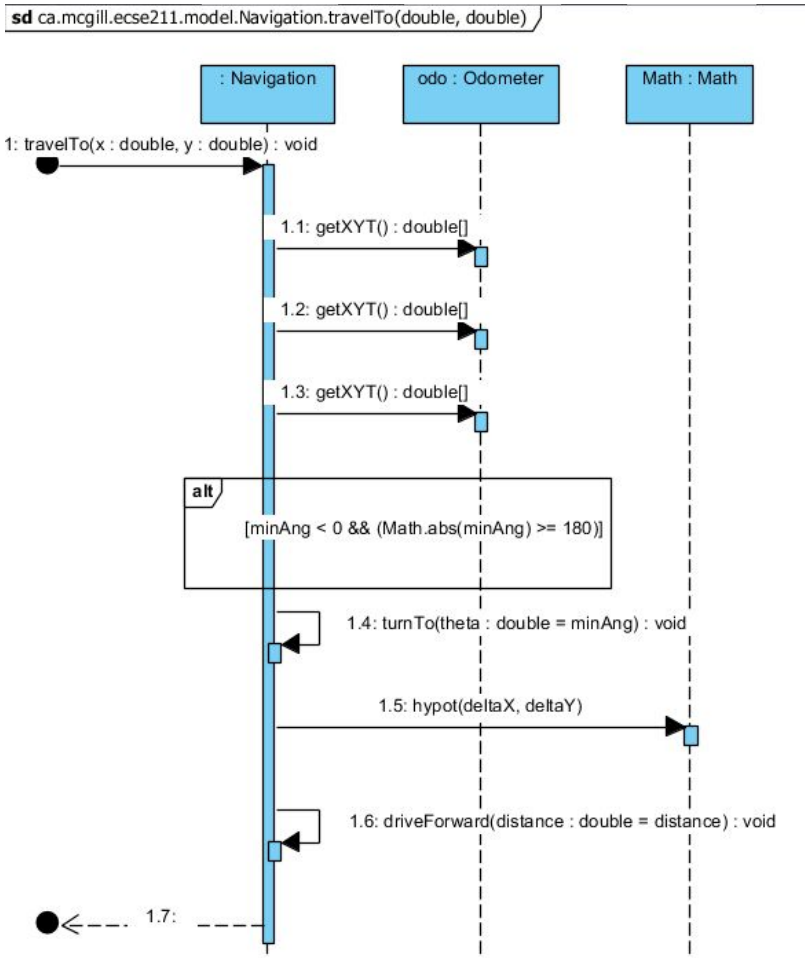

Figure 13: Navigation Class, travelTo(double, double) sequence diagram

4.4.2 LightLocalizer Class - lightLocalize(double, double)

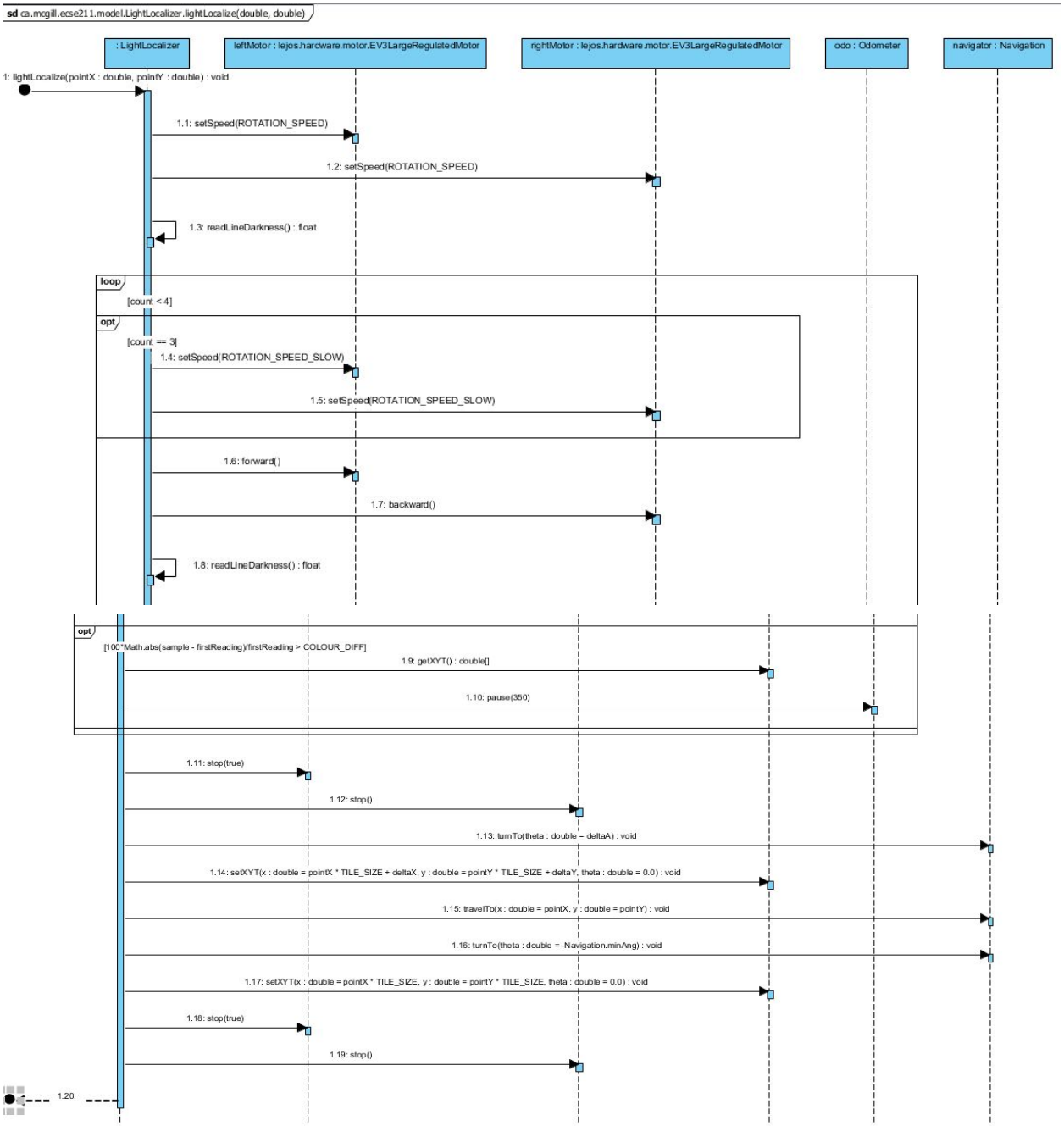

Figure 14: LightLocalizer Class, lightLocalize(double, double) sequence diagram

4.4.3 UltrasonicLocalizer Class - fallingEdge()

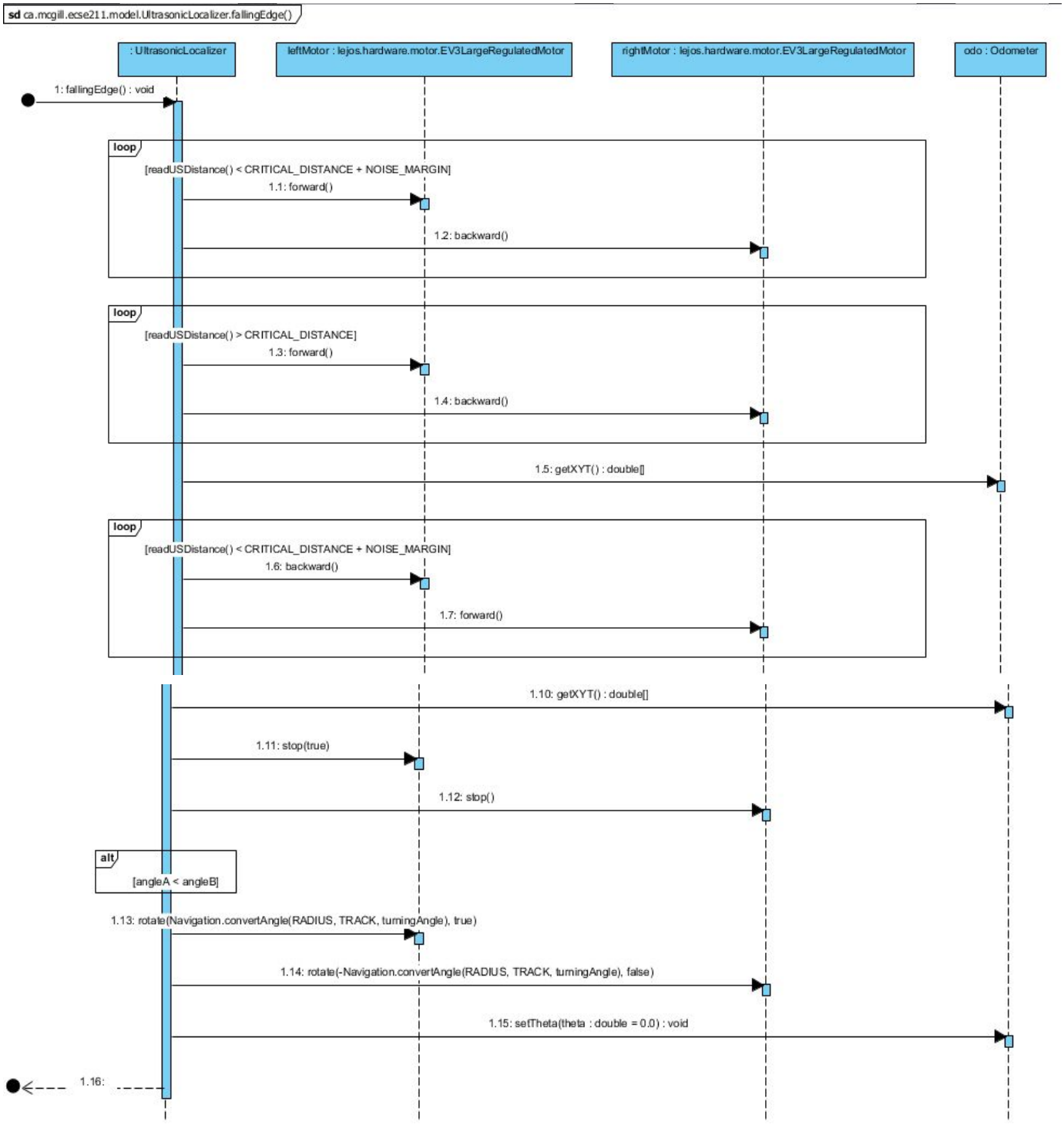

Figure 15: UltrasonicLocalizer Class, fallingEdge() sequence diagram

4.4.4 AssessCanColor Class - AssessCanColor.run()

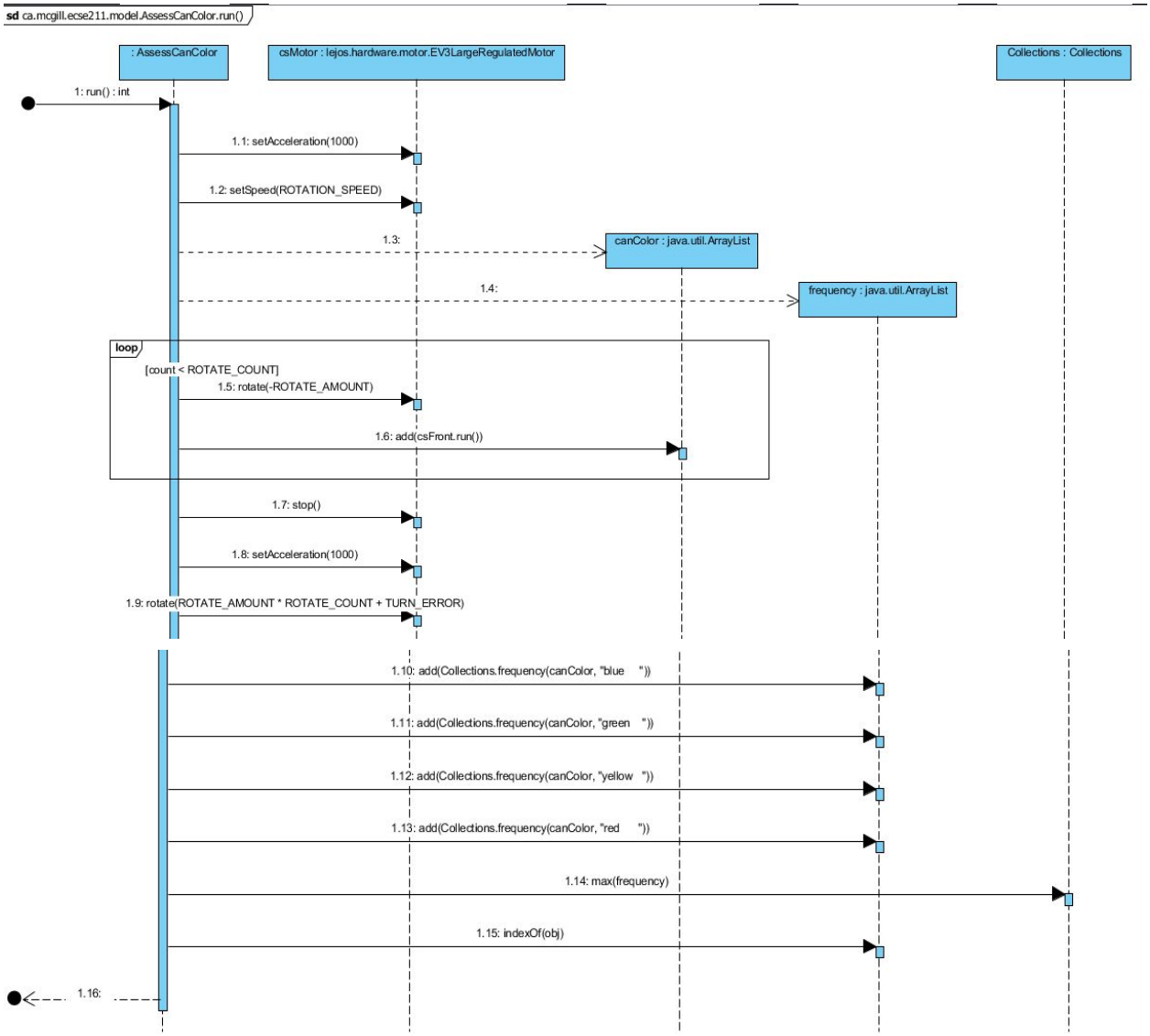

Figure 16: AssessCanColor Class, run() sequence diagram

4.4.5 Odometer Class - Odometer.run()

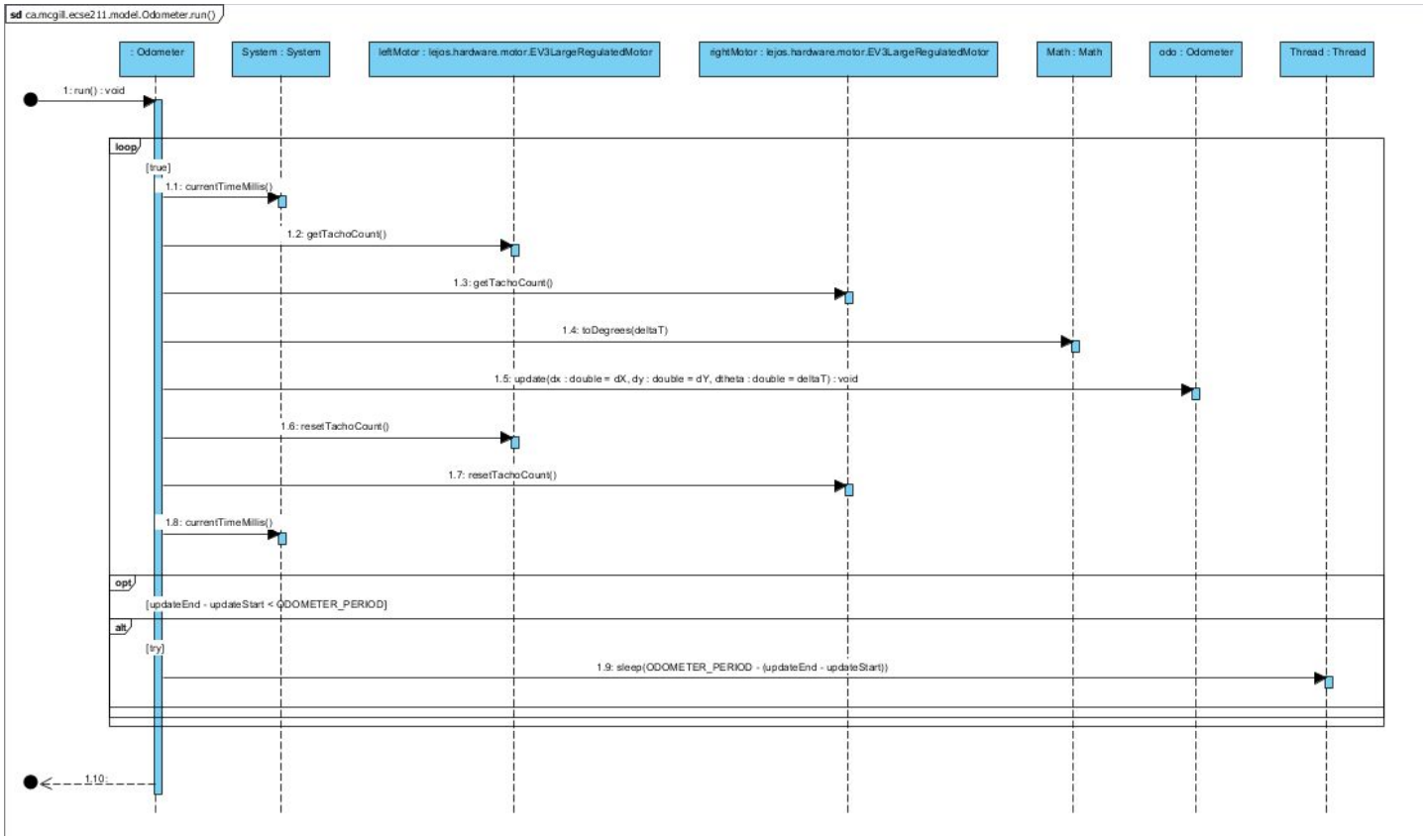

Figure 17: Odometer Class, run() sequence diagram

4.4.6 Robot Class - Robot.getStartingCorner()

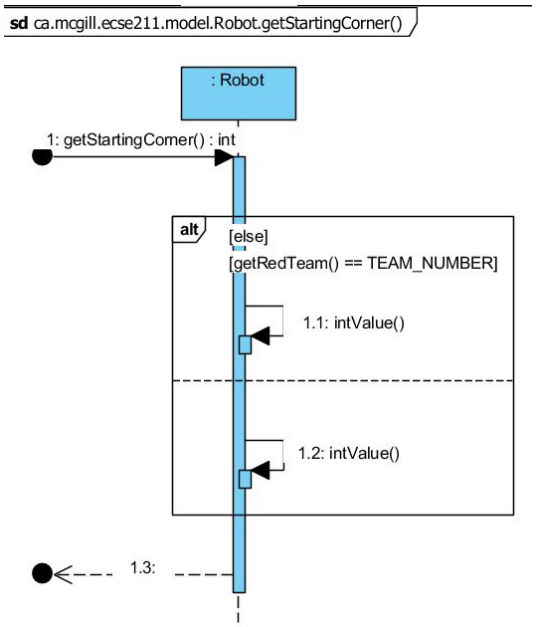

Figure 18: Robot Class, getStartingCorner() sequence diagram

5.0 Javadocs

Please refer to the html files in the Javadocs folder for the complete API.

6.0 Further Improvements

6.1 Lab 5

The completion of lab five was a success, however many of the calculations were inaccurate and required reworking. First, the odometer was inconsistent, which led to many navigation errors as the EV3 traveled around the zone. This issue was fixed after the demo by localizing more often (i.e calling `lightLocalize()` at every corner of the search zone). Next, the target can parameter made it difficult to navigate around the search zone due to the numerous cases that had to be written. For instance, *figure 19* shows a scenario with the target color set to blue and the EV3 spotting the yellow can first. If the yellow can is wrong, it needs to call `insideDodge()`, which only drives the EV3 back and rotates it 90º to the left so that it is oriented parallel with the zone edge again. It then spots the red can and assesses it, then calls the `borderDodge()` method, which will navigate the EV3 to the outside of the

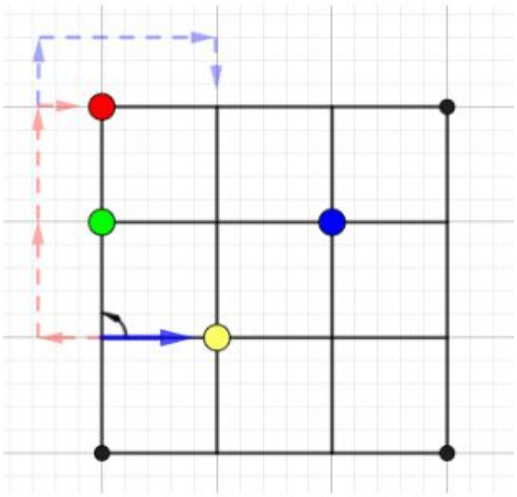

Figure 19: Lab 5 Search diagram with the blue can as a target color.

search zone in order for it to avoid the can. Once `borderDodge()` returns, the EV3 will spot and assess the green can. Since the EV3 is on the outside, the `outsideDodge()` method is called to avoid the green can and get back onto the border of the zone. This also applied to the `travelToUR()` methods, because spotting the right colored can from the outside is not the same as spotting the can on the border or inside the zone. Although these methods functioned as intended, the EV3 would deviate as it traveled, since the methods were hard-coded because of odometer inconsistency (i.e commanding the EV3 to drive back one tile, turn left, travel up, etc..). Once the odometer was fixed, methods such as `travelTo()` were used to allow for more accurate and consistent navigation. Finally, the choice of removing the clamp caused the color classification to be inconsistent. Since the EV3 is not oriented at the exact same angle everytime it spots a can, the color sensor would either be too close or too far, which will lead to different identification results. Adding the clamp allowed the EV3 to hold the can at a certain position so that the color sensor can measure consistent data.

6.2 Beto Demo

Although all the fixes above were implemented during the beta phase, it was not a success due to a thread bug. The robot localized at corner zero in less than thirty seconds, then began to head to the tunnel. Just as it finished localizing next to the tunnel, the left motor increased its speed for a few milliseconds. One main cause of this issue was having the LCD display thread running in parallel with the motors and the odometer. Since the CPU is alternating between instructions of the odometer, display, and the motors, it eventually noticed the left motor was behind on its rotation, and ended up ramping the current into the motor to help it catch up. The solution to this problem was removing the LCD display, since it was not required for the final (only helped with testing), as well as halting the odometer for a few milliseconds (~50) using `Thread.sleep()` whenever the motors were called initially so that both start up at the same time. Another cause might have been due to a few variables declared as `static`, taking up space in memory.

Despite the thread issues, the search and color classification worked as intended. One final modification was the implementation of the `travelToStartCorner()` method. The beta demo required the EV3 to travel to the upper right once the can was detected, however, the final demo required the can to be returned to the starting zone, hence replacing the old `travelToUR()` method with `travelToStartCorner()`.

6.3 Final Demo

The preparation for the final demo did not take many change. The cases for the three other start corners (1,2,3) were written, as well as the `goHome()` method for each. During our second demo run, the lower left was at the border of the grid, however it was not closer to the EV3 when compared to the upper right corner. If it was, the EV3 would have attempted to travel there, eventually colliding with the wall. During the third run, we noticed that the `TEST_ANGLE_FAR` was not accurate because the EV3 battery had gone down below the testing threshold, which affected the EV3's navigation and turning. Our first and last runs were successful, however the EV3 localized too many times and the robot did not make it back in time. The closest attempt was the last, were the EV3

was localizing to offload the can, and as it finished localizing, the time ran out. The project was an overall success since all four runs had the EV3 localize in time, navigate to the search zone, and begin searching for a can.

7.0 Evolution of the Software Document

7.1 Version 1.02

Version 1.02, created in the first week of the project, of the software document was the primary stages of our software development. We were focused on understanding the project requirements and software constraints that we would possibly face during the design process (as explained in Section 3.0). We created initial flow charts for the overall system as well as the Localization, Navigation, and Assess Can functions. The first week of the project overlapped with Lab 5 and therefore most of our efforts were focused towards succeeding in the lab. However, it provided a good starting point for the project; the base of our initial flow charts and preliminary software architecture were inspired by Lab 5. We also performed some basic software testing for the Color Classification function (Testing Document, Section 4.1.1 Color Classification Test Version 1.0) as the code had already been developed for this function as a part of Lab 5. The color classification function proved to be reliable in the majority of cases, however more tests were performed later to ensure its accuracy in identifying blue versus green cans.

7.2 Version 2.00

Version 2.00 of the software document barely had any changes except for adding the flow chart for the Assess Can function. During this week, which was week 2, we were more focused on the hardware design process and discussing how to transform the information on the flowcharts to actual software development.

7.3 Version 3.03

In week 3, we completed one task for the software document, but arguably the most important one: the creation of our class diagram. For version 3.03 of the software document, the software team fully assessed the requirements of the project and developed a comprehensive class diagram, detailing the classes and their methods and associate it with other classes. Since we had a skeleton for all the classes, we decided to javadoc all the methods without implementing all of them which assisted us in doing the preliminary API docs. The creation of the class diagram took several days to complete, however it provided a solid base for the beginning of the actual writing of code which properly started in Week 4.

7.4 Version 4.01

Version 4.01 of the software document marked the beginning of the code development for the final project. During this week, we were preparing for the beta demo. We properly defined the software design process and begun writing class descriptions for the various methods and classes present in our code. Since a lot of the software still had yet to be developed, the class descriptions section was initially a skeleton, later filled in during the last few weeks as the software was being finalized. Many of the software functions were

tested during this week as we agreed that the earlier tests were performed the better. The testing took place in order to identify potential issues and inaccuracies in the robot's performance of the functions that had been developed at this stage in the project. During this week, the Navigation function of the robot was tested (Navigation Test Version 1.0) to assess its accuracy. The results showed that the navigation function was mostly accurate, however it needed further testing. The Light Localization function was also tested (Testing Document, Section 4.4.1 Light Localization Test Version 1.0) to assess its accuracy. The results proved that it was accurate and needed a bit more testing, which was conducted in the coming weeks. Furthermore, the Obstacle Traversal function of the robot was tested (Testing Document, Section 4.5.1 Obstacle Traversal Test Version 1.0) to determine whether the robot was able to travel through tunnels and bridges successfully.

7.5 Version 5.02

Version 5.02 of the software document involved the refinement of the flow charts and the class diagram. Furthermore, we continued to develop the class descriptions section. We broke each of the classes down into their respective methods in order to explain the function and purpose of each one and how they connect together. We also developed sequence diagrams for a few of the significant methods in our classes. Furthermore, during week 5, the Ultrasonic Localization Angle (Testing Document, Version 4.3.1 Ultrasonic Localization Angle Test Version 1.0) was tested to assess the accuracy of the Localization class in terms of the final angle post-localization.

7.6 Version 6.01

Version 6.01 of the software document was an indication of the finalization of our software development. The class and method descriptions were almost fully completed and we assessed possible future improvements in terms of Lab 5 and the Beta Demo. In addition, as we were finalizing our software in preparation for the final demo, further tests was conducted on the software functions in order to assess their proper functionality. The Color Classification function was tested a second time (Testing Document, Section 4.1.2 Color Classification Test Version 2.0) in order to ensure its 100% accuracy in time for the final demo. Furthermore, the Navigation function was tested a second time (Testing Document, Section 4.2.2 Navigation Test Version 2.0) to assess its accuracy. We also re-tested the Ultrasonic Localization Angle (Testing Document, Section 4.3.2 Ultrasonic Localization Angle Test Version 2.0) to assess the accuracy of the angle post-localization and determine if an angle offset needs to be implemented. Additionally, the Light Localization function (Testing Document, Section 4.4.2 Light Localization Test Version 2.0) was re-tested to assess its accuracy. The Obstacle Traversal Test was done again (Testing Document, Section 4.5.2 Obstacle Traversal Test Version 2.0) to ensure the robot could travel through tunnels and bridges. Lastly, a Can Detection Test was performed (Testing Document, Section 4.6 Can Detection Test) to assess whether the robot can detect cans in terms of its turning angle.

7.7 Version 7.02

Version 7.02 of the software document was the finalization of the class diagram, the sequence diagrams, and the class and method descriptions. After the final demo, we discussed improvements for the future. Version 7.02 represents the final version of the software design document.