Software Design Document

Project: DPM Final Project

Document Version Number: 4.0

Author: Zakaria ESSADAOUI

Carl El Khoury

EDIT HISTORY:

[29/10/2018] Zakaria: Created the document + added flow chart.

[30/10/2018] Zakaria: Added UMLs and class hierarchy+ comments.

[05/11/2018] Zakaria: Added the Week 3 subsection on the document.

[10/11/2018] Carl: Added the Week 4 subsection on the document.

[13/11/2018] Zakaria: Added the flow chart for the line detection

[13/11/2018] Carl: Added the flow chart for the ring grabbing routine

[18/11/2018] Zakaria: Re-structured the document + Added the pollers, color detection, ring grabbing and final project sections

[19/11/2018] Zakaria: Added the tree search and tunnel traversal algorithm+ final project flow chart

Tables of contents:

A. Overview:

This document constitutes the software documents which includes a description of all the classes used as well as descriptions for the algorithms that were used in some of the important methods. It is organized by sub-systems. We will include flow charts and UML diagrams to help communicate the desired idea to the reader.

B. Odometer:

The odometer is used to keep track of the robot at all time. For this system, we re-used the code from the Lab2. However, we deleted the "OdometryCorrection" class since it will not be need in for the final project.

Here we can see a summary of the methods used by this sub-system:

-Odometer class:

Description: This class is the skeleton of the odometer class.

```
public synchronized static Odometer getOdometer(EV3LargeRegulatedMotor leftMotor,
EV3LargeRegulatedMotor rightMotor, final double TRACK, final double WHEEL_RAD)
```

Type: public synchronized static
Parameters: EV3LargeRegulatedMotor leftMotor, EV3LargeRegulatedMotor rightMotor, final double TRACK, final double WHEEL_RAD
Throws: OdometerException
Return: Odometer

Description: This method is meant to ensure only one instance of the odometer is used throughout the code.

```
public synchronized static Odometer getOdometer()
```

Type: public synchronized static
Parameters: None
Return: Odometer
Throws: OdometerException (if no previous odometer exists)

Description: This class is meant to return the existing Odometer Object. It is meant to be used only if an odometer object has been created

```
public void run()
```

Type: public
Parameters: None
Return: None

Description: This method calculates the change in X, Y and Theta and updates them. This is the same method used in the slides with some changes to the angles units.

-OdometerData

Description: This class stores and provides thread safe access to the odometer data.

```
public synchronized static OdometerData getOdometerData()
```

Description: OdometerData factory. Returns an OdometerData instance and makes sure that only one instance is ever created. If the user tries to instantiate multiple objects, the method throws a MultipleOdometerDataException.

```
public double[] getXYT()
```

Description: Return the Odometer data. Writes the current position and orientation of the robot onto the odoData array. odoData[0] =x odoData[1] = y odoData[2] = theta

```
public void update(double dx, double dy, double dtheta)
```

Description: Adds dx, dy and dtheta to the current values of x, y and theta, respectively. Useful for odometry.

```
public void setXYT(double x, double y, double theta)
```

Description : Overrides the values of x, y and theta. Used for odometry correction.

```
public void setX(double x)
```

Description: Overrides x. Use for odometry correction.

```
public void setY(double y)
```

Description: Overrides y. Use for odometry correction.

```
public void setTheta(double theta)
```
Description: Overrides theta. Use for odometry correction.

C. Navigation:

This sub-system takes care of all the navigation related tasks. It is mainly re-used from Lab 3. The basic structure of that system can be shown in the following UML diagram:

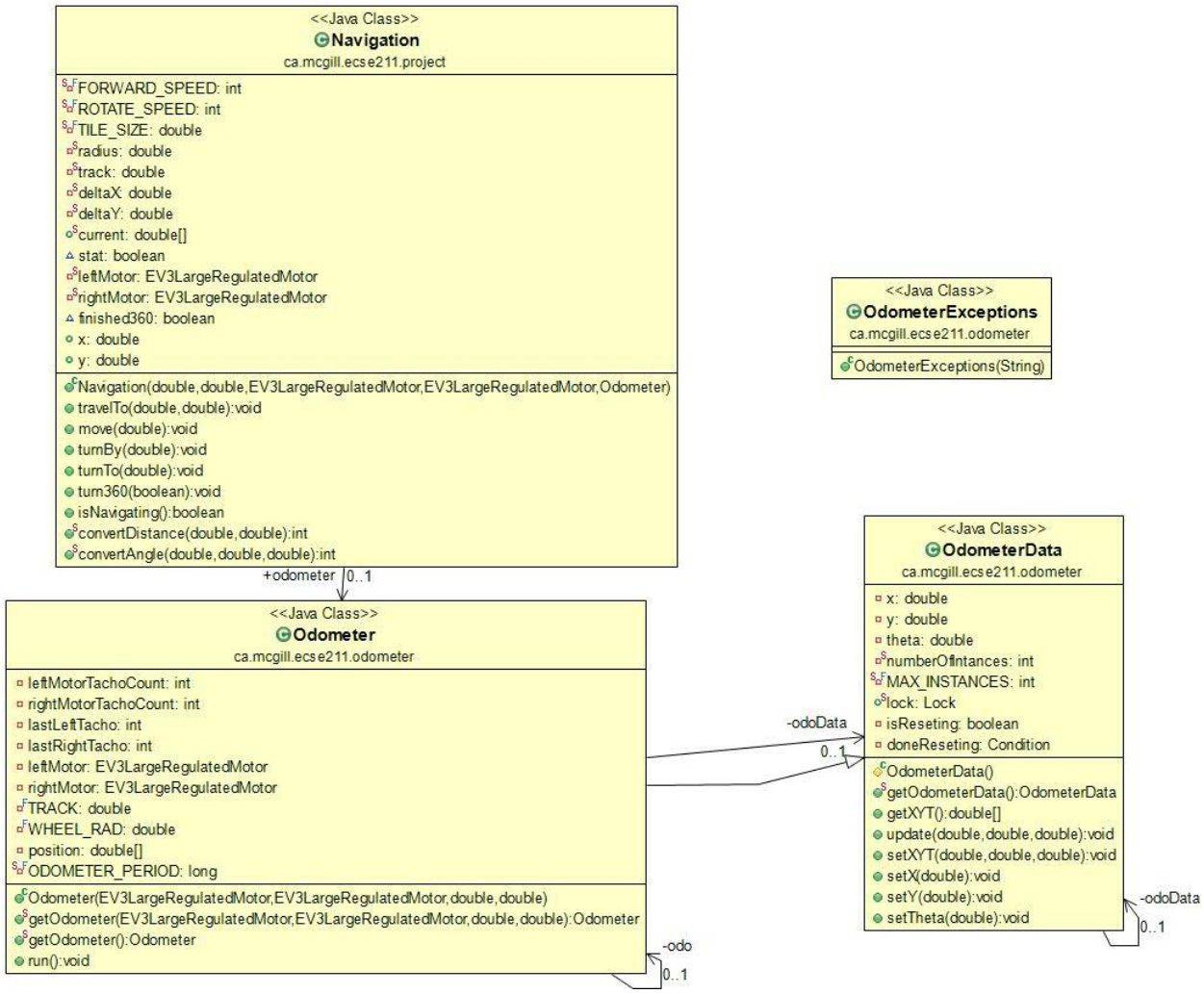


Figure 1: UML diagram from the Navigation Lab

It always interacts with odometer class to get the position of the robot on the grid to know by how much it should move. In our final project, this system will be used in a similar fashion in order to navigate to the tree and back to our starting corner.

Here we can see a summary of the methods used by this sub-system:

Description: This class is responsible for the calculations of distance and angle to reach desired coordinate by taking in current position information and running set calculations on the info to solve for needed variables.

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

Description: This method makes the robot travel to a (x,y).It first measures get the position of the robot from the odometer. Then it calculates the angle it should be at and the distance it needs to move by.

```
public void move(double distance)
```

Description: This method makes the robot move by the distance that was passed to it

```
public void turnBy(double theta)
```

Description: This method makes the robot turn (clockwise) by the angle that was passed to it

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

Description: This method make the robot turn to a certain angle. It also make sure that it uses the minimal angle to turn.

```
public void turn360(boolean clockWise)
```

Description: Instructs the robot to spin 360 degrees in a direction specified by the boolean argument. A boolean true argument would result in clockwise 360 degree turn.

```
public boolean isNavigating()
```

Description: This method returns whether the robot is navigating or not

```
public static int convertDistance(double radius, double distance)
```

Description: This method computes the angle that must be passed to the motor using the radius of the wheel and the distance we want the robot to cross.

```
public static int convertAngle(double radius, double width, double angle)
```

Description: This method calculates the angle by which the motor must turn in order for the robot to turn by a certain angle.

D. Light poller:

This sub-system takes care of initializing a light poller which is used for line detect. It basically gets data for light sensor and checks for drops in the readings which indicates that that sensor is on a line.

Method description:

```
public boolean line_detected(boolean first)
```

Description: This method detects if a line was crossed using a differential filter (i.e takes a past reading and compare it with a new reading).

```
private int fetch()
```
Description: This method is used to get data from the light sensor, the data is pulled from the sample provider, scaled by a factor of 1000 then returned.

E. Ultrasonic poller:

This sub-system is used for polling data the ultrasonic sensor. It has a feature for detecting false negatives (no object detected) by using a filter.

Method summary:

```
public int getDistance()
```

Description: This method returns the distance between the sensor and the closest object in front of it in centimeters.

```
public int process(int distance)
```

Description: This is the filter we are using to deal with false negatives, when a distance of more than 255 is detected, we check 20 times that it is indeed 255.

F. Localization

This system is used to perform the ultrasonic and light localization. We re-used part of the code for Lab 4. In that lab, we had two ways of performing the ultrasonic localization (falling edge and rising edge). Since we don't use both ways and falling edge works better in our design, we decided to delete the rising edge part. For the light localization, we decided to change the design to use two light sensors instead of one, we noticed that this gave us more precision. It also allowed us to perform light localization in different positions of the grid without worrying about errors. We also decided to add poller classes to have cleaner code structure. By doing so, we would have three pollers for these tasks, "LeftLightPoller", "RightLightPoller", "UltrasonicPoller".

After making all the changes above mentioned, we ended up with the following structure for this subsystem:

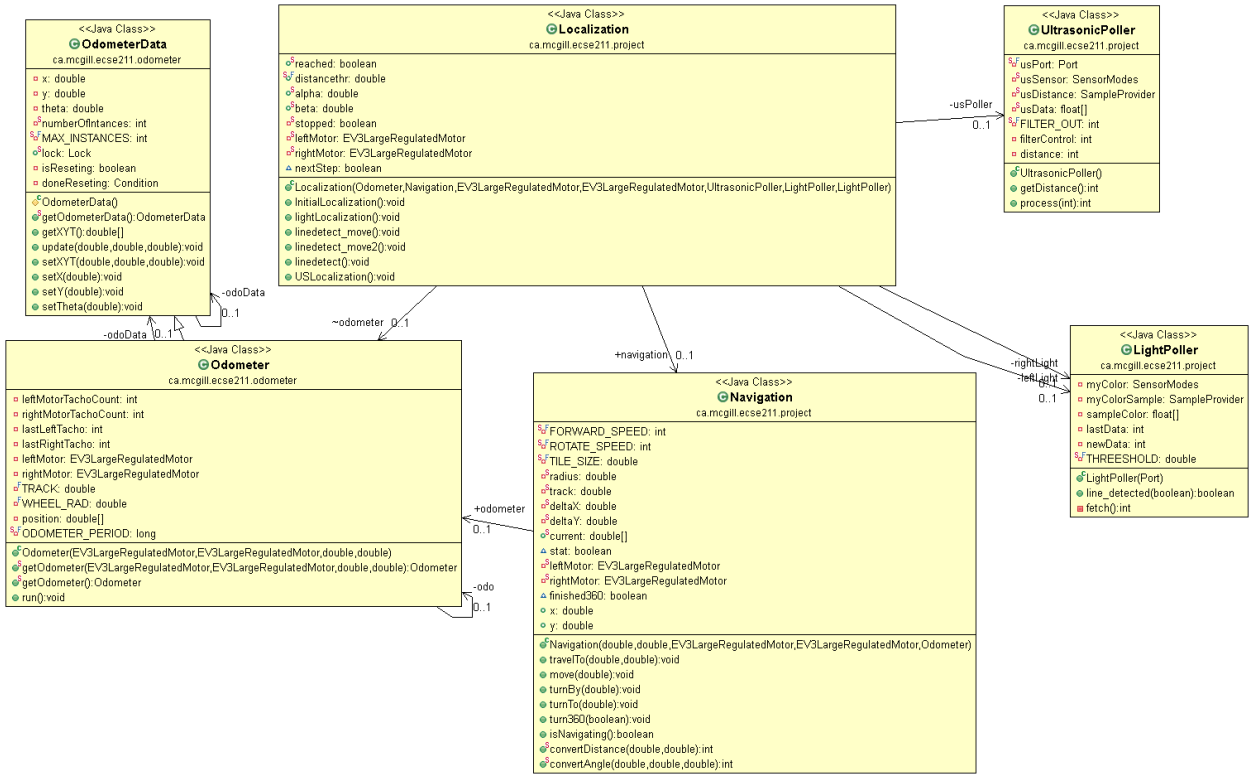


Figure 2: UML diagram for the Localization subsystem

One of the main methods used in this subsection was the line detection. We included the following flow chart that explains how does the algorithm used work:

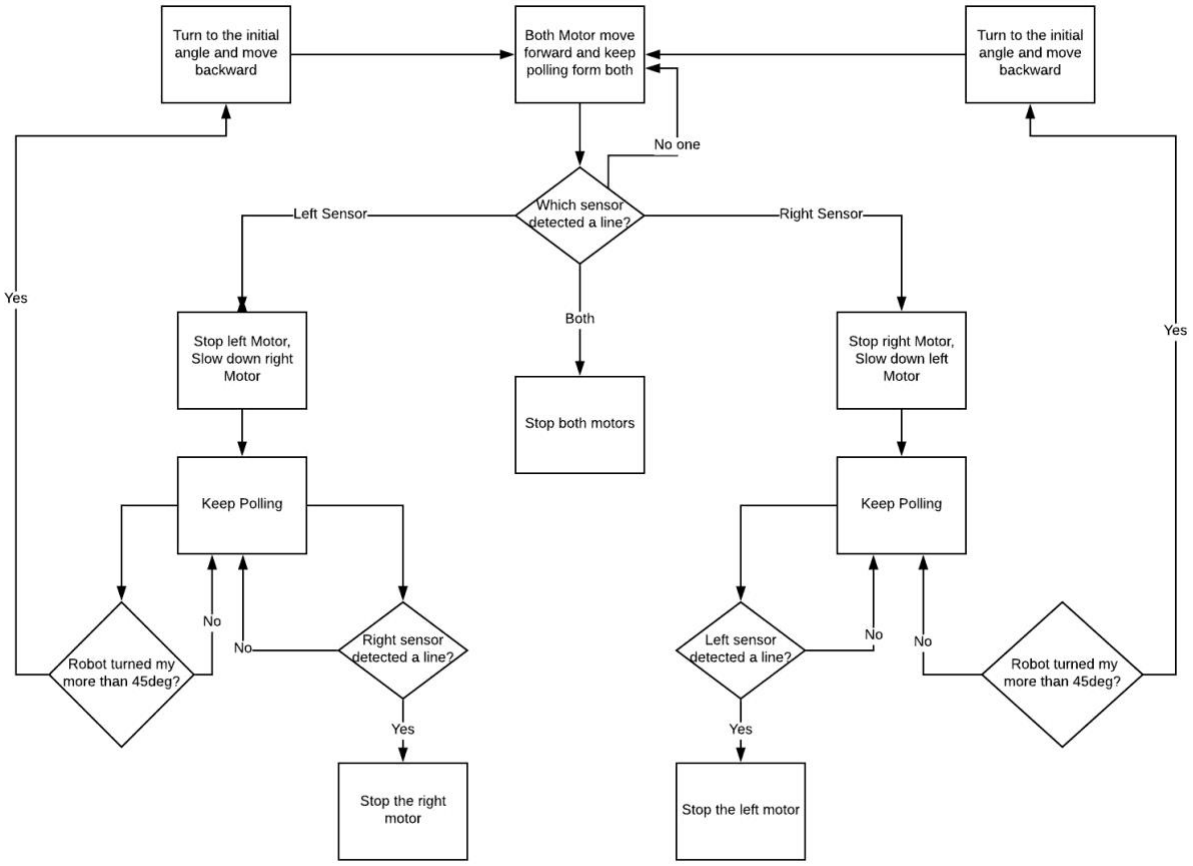


Figure 3: Flow chart for the line detection method

As shown in the flow chart above, we deal with extreme cases where one of the pollers fails to detect a line by reorienting the robot in its initial orientation (before the method start) and moving backward, then we call line detection again.

We included the following methods description:

```
public void InitialLocalization()
```

Description: This method performs the initial localization. It first calls the ultrasonic localization method, then it calls the light localization method and returns.

```
public void lightLocalization()
```

Description: This method is used to perform our light localization Make the robot move forward until both sensors detect the line turn by 90degress and does the same in that direction.

```
public void linedetect()
```

Description: This method makes the robot move forward until one sensor detect a black line. At that point, only the wheel from the other side keep turning until the it detect a line. It also include mechanisms to deal with extreme cases.

```
public void linedetect_move()
```

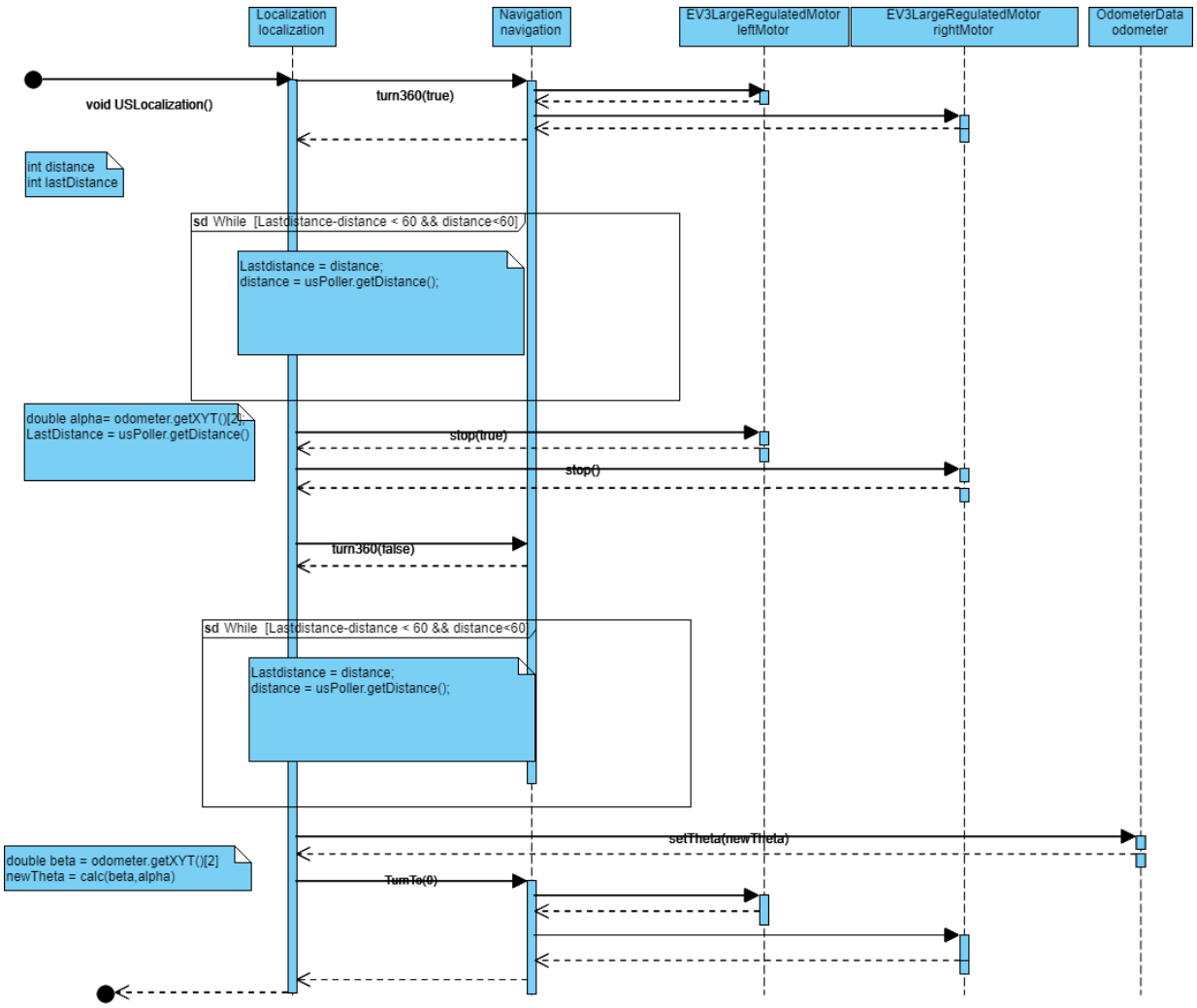
Description: This method make the robot detect a line (using line detect) then move forward by the wheel to sensors offset in order to make the wheels exactly at the line.

```
public void linedetect_move2()
```

Description: This method first makes the robot move backward, then calls line detection, then makes the robot move forward in order to make the wheel directly on the line. It used to make sure we detect a line when we might think that the sensors are already in front of the lines.

```
public void USLocalization()
```

Sequence diagram representing the simplified version of USLocalization method calls:



Description: This method reads distance values from the ultrasonic sensor and detects rapid changes in the distances during movement. Depending on the direction of the change in value, it is considered a rising or falling edge. Two edges are detected during the sequence and the robots angle at the moment of each detection is used to calculate the angle needed to accurately orient the robot at 0 degrees.

G. Color detection

This class is used to detect ring's color. To create this class, we first had to acquire the RGB values from the rings in order to calibrate the sensor. Then, we computed all the normalized means to get be able to detect the rings.

Method summary:

```
public int detect ()
```

Description: This method returns the color detected.

```
public static float[] fetch()
```

Description: This method gets the data from the sensor

```
public static int findMatch (float array[])
```

Description: This method returns the corresponding color for the RGB values that were passed to it If the RGB doesn't correspond to any known colors, it returns 5. To decide what to returns, it first computes the normalized RGB values of the array passed, then it gets the color with the smallest Euclidian distance compared to the RGB values passed to it. After that, it makes sure that the normalized RGB values are close enough (within a 0.10% window) to the means of the color with the smallest Euclidean distance. If the color fails this tests, the method returns 5, otherwise it returns that color.

H. Ring grabbing

This is the subsystem that takes care of detecting and grabbing a ring. This class interacts with both the Navigation and ColorDetection as shown in the following UML diagram:

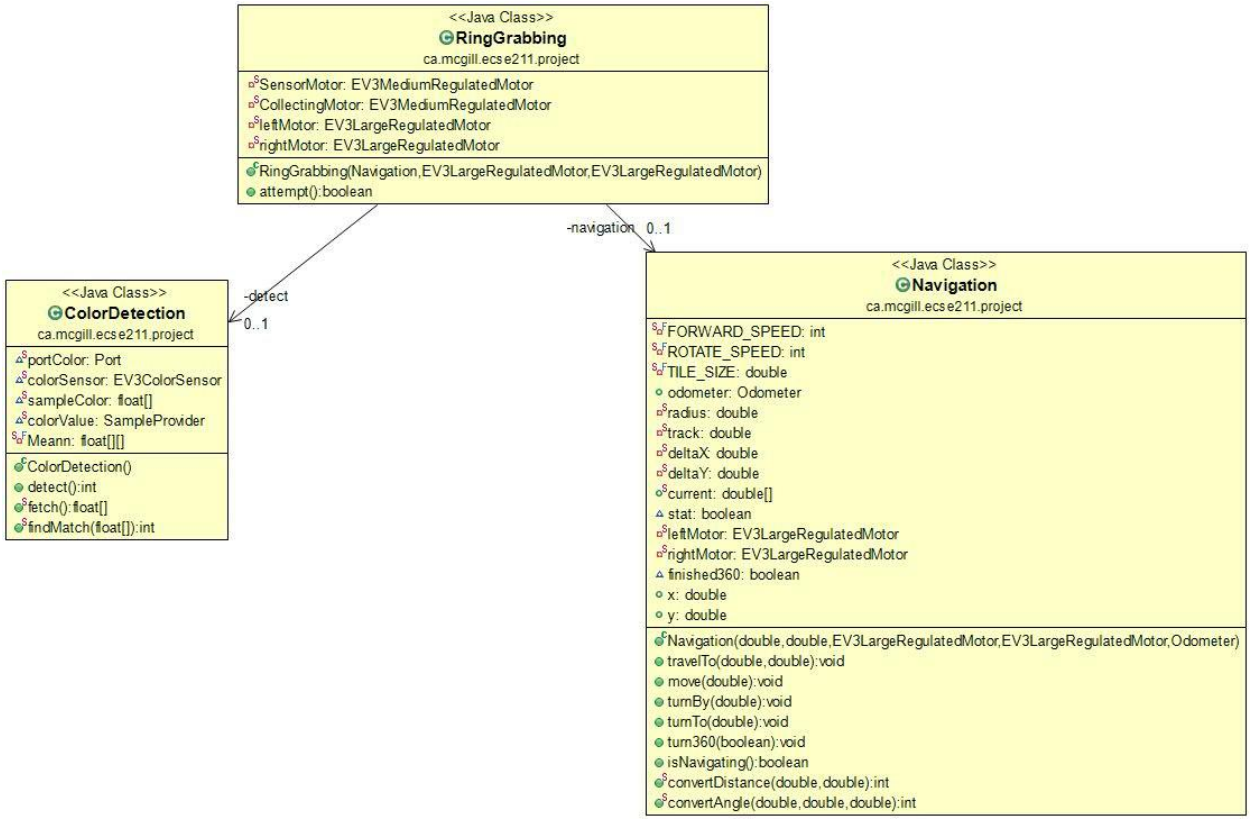


Figure 4: UML diagram for the ring grabbing sub system

Basically, this class has one method which performs the operations summarized in the following flow chart:

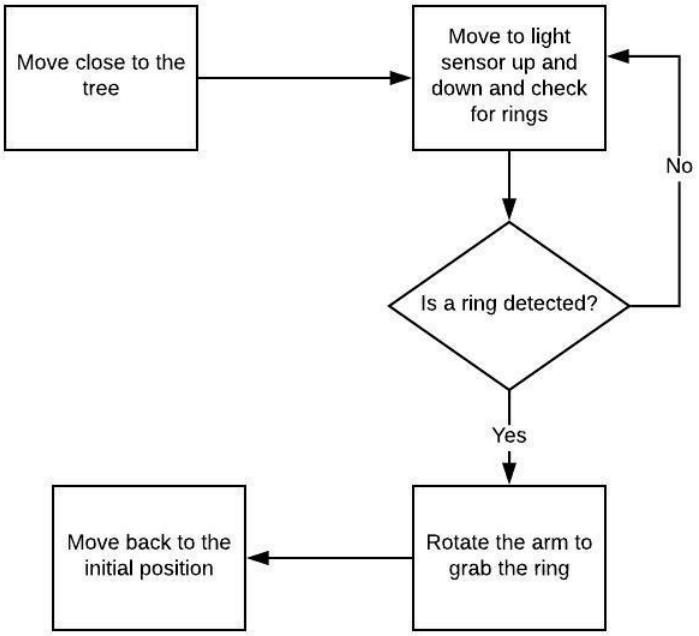


Figure 5: Flow chart for the ring grabbing mechanism

Method summary:

```
public boolean attempt()
```

Description: This method checks for a ring in the side that the robot is facing. It makes the robot go close to the tree, then starts scanning for a ring while making the sensor move up and down. If a ring is detected, the robots grab it and beeps according to the ring's color and backs of. Otherwise, the robot just backs of.

I. Final project

This is the main class we are using to perform all the task needed. It responsible for creating instance of all the class used and calling their methods when needed. We are using all the sub-systems explained above either on their own or with interaction with each others.

The hierarchy of the full project can be summarized in the following UML diagram:

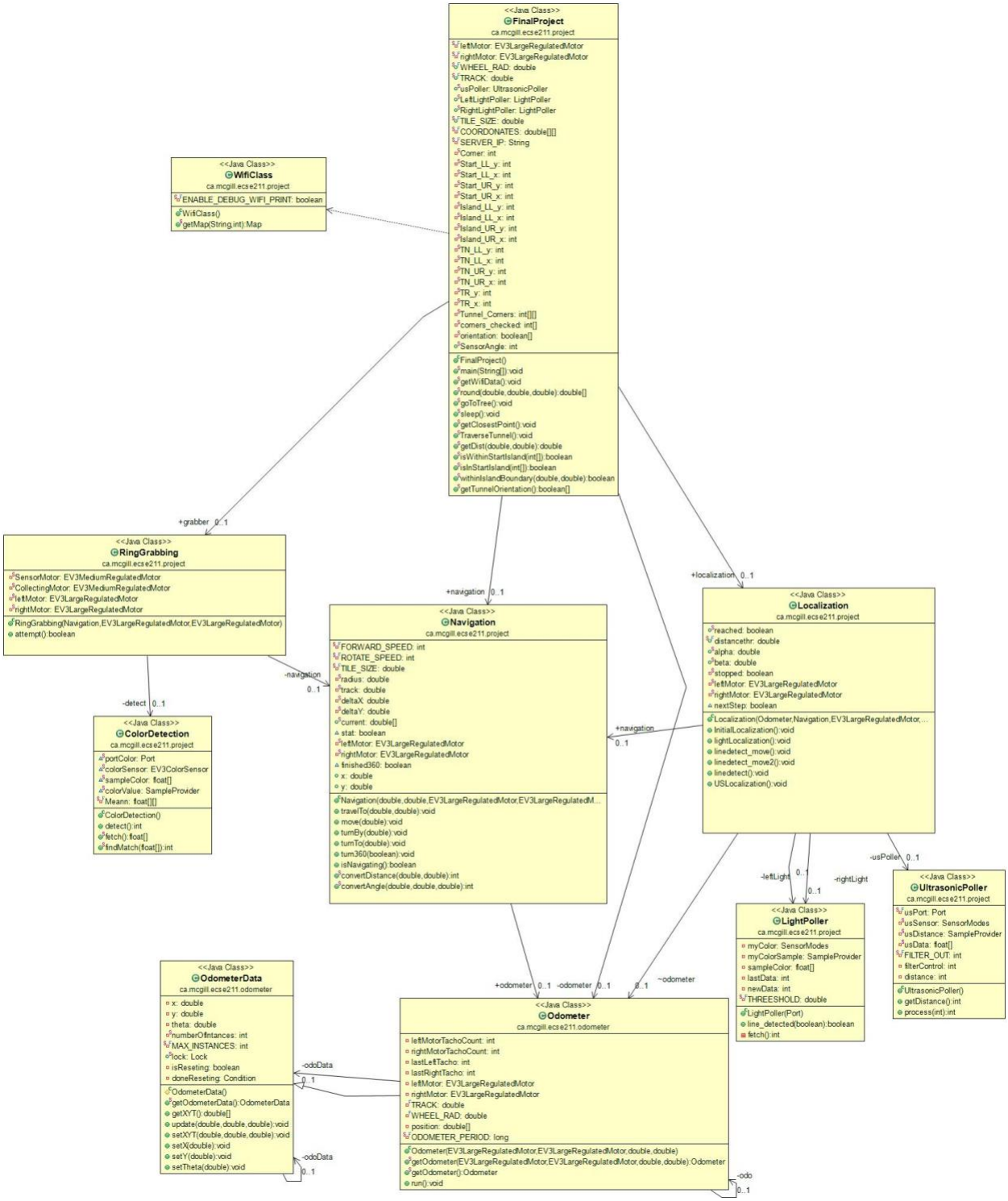
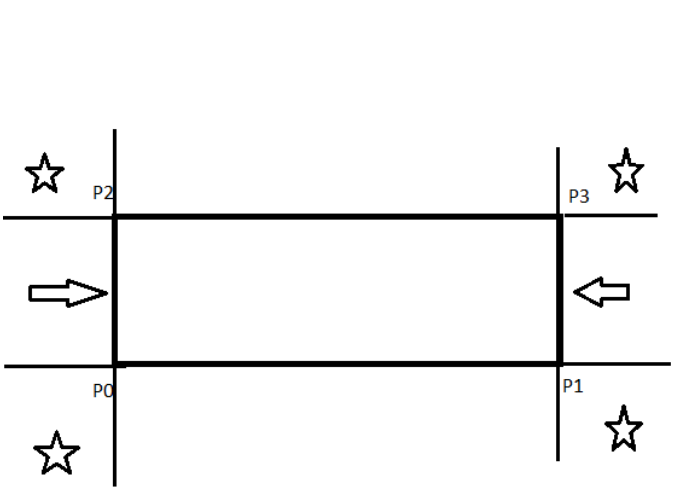


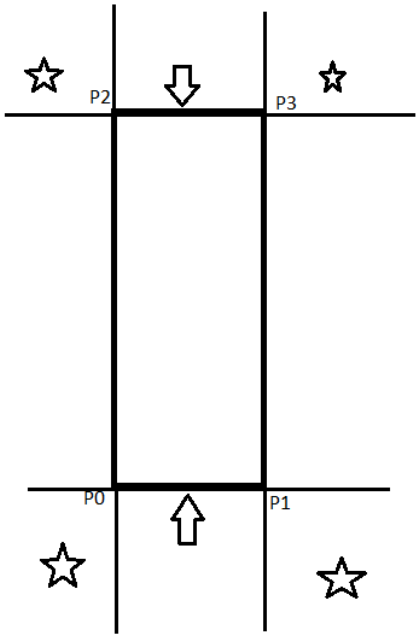
Figure 6: Full Project's UML

Tunnel traversal algorithm:

In order to cross the tunnel, we used the following idea:



Case where the tunnel is horizontal



Case where the tunnel is vertical

First, we define four points (P0, P1, P2, P3) which are the corners of the tunnel as shown above. Then, depending on the which points are within the starting Island, we can conclude on both the orientation of the tunnel (Horizontal or Vertical) and the direction at which we will travel it the first time (UP OR DOWN/ LEFT OR RIGHT).

Once we have this information, we go to the closest corner of the tunnel, specifically to one of the points labeled with a star. We then perform a line detection at the line next to that point. We then travel to one of the points pointed to by an arrow (depending on the case) and perform a line detection to fix our heading. We would then be perfectly aligned with the tunnel. After that, we traverse the tunnel and perform a light localization after that.

In order to come back, we only need to change the direction of the traversal to the opposite (either true to false OR false to true) and call the same method.

With this algorithm, we take care of the four cases and depending on the actual run, we will choose the right sequence of operations to do.

Tree search and ring grabbing algorithm:

In order to look for rings in the tree, we first define the points (P0, P1, P2, P3) as shown in the figure below. The red circle defines the tree. After that, we go to the closest point to from the robot's initial location that is accessible (not in the boundary of the island). We perform a light localization at that point in order to be perfectly aligned with the tree. After that, we use the ring grabbing algorithm that was explained in section H. Once that operation is done, we check if there are any other accessible corners and we go check for rings over there. We also take into account cases where the next point is facing the actual point (P0 with P2 and P1 with P3). Since the tree is between the two points, we first travel to an intermediate point first before travelling to the target point.

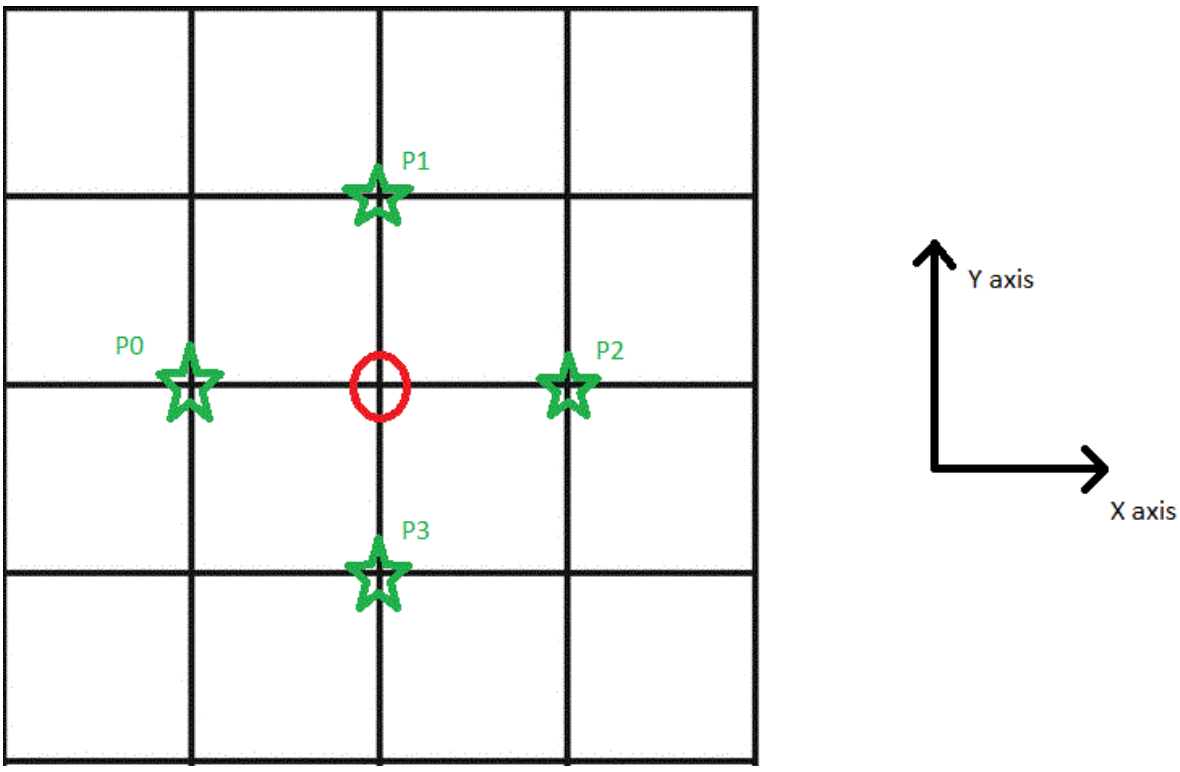


Figure 7: Tree searching scenario

All the steps that being executed through each run can be summarized in the following flow chat:

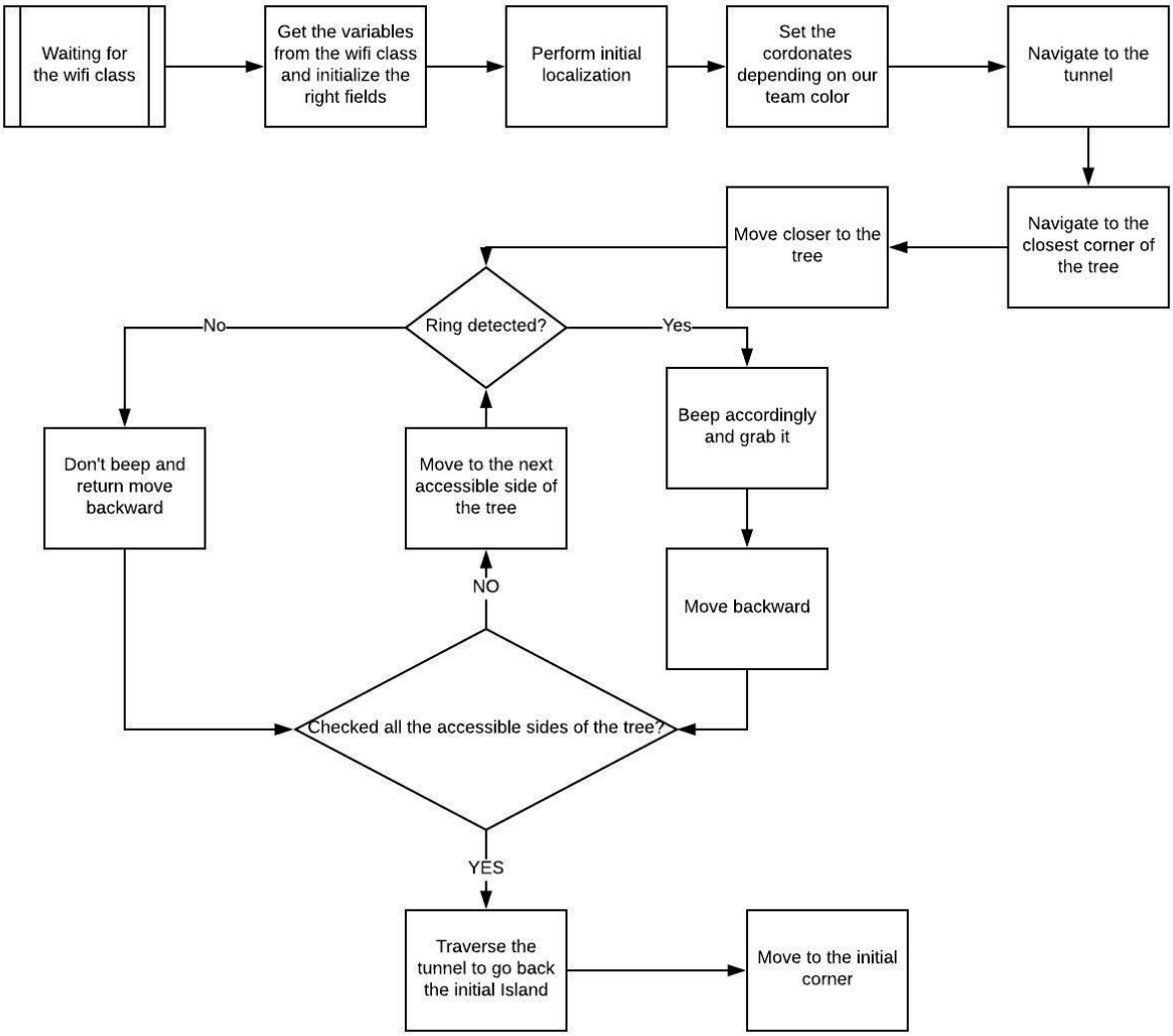


Figure 8: Full project's flow chart

Method summary:

```
public static void getWifiData()
```

Description: This method gets data from the WIFI class and initiate all the appropriate fields. First, it checks which team we are (Red or Green) and initiate the Islands and Tree parameters accordingly.

```
public static void goToTree()
```

Description: This method makes the robot go to closest point next to the tree first. Then checks for rings in all the sides of the tree that are accessible (not next to a wall or the water). At each corner, it calls the attempt method from the RingGrabbing class.

```
public static void sleep()
```

Description: This method makes the robot sleep for one second.

```
public static void getClosestPoint()
```

Description: When called, this method checks for the closest corner of the tree from the robot's position. Then it fills the corners_checked[corner] with 1. It only checks for available corners, i.e. corners that are nor a wall nor in water.

```
public static void TraverseTunnel()
```

Description: This method makes the robot traverse the tunnel and localize at the other end of the tunnel. It follows the algorithm explained previously.

```
public static double getDist(double deltaX, double deltaY)
```

Description: This method computes the Euclidean distance between two points

```
public static boolean isWithinStartIsland(int XY[])
```

Description: This method is checks if a point with coordinates (x,y) is within the start island (island in which the robot is positioned at the beginning).

```
public static boolean isInStartIsland(int XY[])
```

Description: This method checks if a point with coordinates (x,y) is in the starting island (island in which the robot is positioned at the beginning), excluding the boundaries.

```
public static boolean withinIslandBoundary(double x, double y)
```

Description: This method checks if a point is in the island. A point is not considered in the island if it is located in its boundaries.

```
public static boolean[] getTunnelOrientation()
```

Description: This method checks for the tunnel's orientation (horizontal or vertical) and for the direction at which we want to cross it (UP OR DOWN OR LEFT OR RIGHT).

State diagram: explaining the method calls and the steps of the final project class.