**Software document**

**Author:** Maxime Cardinal

**Date:** October 30th, 2018

**Version:** 5.3

**Edit history:** [Version system is WEEK#.EDIT#]

October 30, 2018 - Maxime Cardinal - Creation of the document, preliminary content - 2.1

October 30, 2018 - Spencer Handfield - Adjusted formatting for consistency with other docs and added table of contents - 2.2

November 5, 2018 - Maxime Cardinal - Modification of TravelToTree class description, Addition of the TravelToTree and travelToBridge() flowcharts - 3.1

November 5, 2018 - Spencer Handfield - Elaboration of findings and design flow/logic of certain classes - 3.2

November 5, 2018 - Irmak Pakis - Added to OdometerCorrection - 3.3

November 6, 2018 - Spencer Handfield - Reformatted certain section to attempt to better illustrate the week by week design process logic - 3.4

November 11, 2018 - Spencer Handfield - Addition of week 4 implementation/modification of software - 4.1

November 13, 2018 - Spencer Handfield - modified section to reflect project description v2.0 - 4.2

November 15, 2018 – Maxime Cardinal – Modification of classes and update of the software architecture due to beta demo results - 5.0

November 18, 2018 – Maxime cardinal – Addition of flowcharts for classes and further explanation of the Wifi class - 5.1

November 20th 2018 - Spencer Handfield - Addition of preambles to both documents to clarify documentation decision, additions to the discussions and decision processes - 5.2

November 27, 2018 - Maxime Cardinal - Modification of the TravelToTree class documentation of the DesignProjectMain flowchart - 5.3

0. PREAMBLE

The beta demo proved to be a significant milestone in more ways than one. The result of the demo we’re so significant and we received quick feedback on some very fundamental portions of our solution that had previously not been brought to our attention during the weekly meetings (see Meeting Minutes, section BETA DEMO for a complete timeline of events as well as the feedback received and excerpts of discussions stemming from such which were a leading cause in many of the changes about to be discussed). The portions that presented problems which the profs highlighted we’re so core to our robot that a major overhaul of the systems was determined necessary.

To clearly distinguish the difference between pre and post beta demo design decisions as well as for the sake of clarity and more focused/iterative analysis and discussion to take place, this second software document was decided to be an appropriate decision to make for the design process. While the previous document contains very essential iterative discussion which all still hold true to integrate the changes made post beta demo would make the document far too dense and lead to more confusion than an accurate reflection of the workflow and manner in which the changes were approached in the last two weeks of development.

**0. Preamble**

**1.0 Table of content**

2.0 Functionality

3.0 Classes

3.1 DesignProjectMain

3.2 OdometerData

3.3 Odometer

3.4 OdometerExceptions

3.5 Wifi

3.6 Localization

3.6.1 usLocalization()

3.6.2 lsLocalization()

3.7 TravelToTree

3.7.1 travelToBridge

3.7.2 travelThroughBridge

3.7.3 travelToRingSet()

3.7.4 odometerCorrection

3.8 RingRetrieval

3.8.1 reachRingSet()

3.8.2 ringGrab()

3.8.3 colorIdentification()

3.8.4 rindUnloading()

3.9 TravelToInitialPosition

3.10 Class Diagram

4.0 Progress and decision explanation TODO (revision+modification)

5.0 Improvements

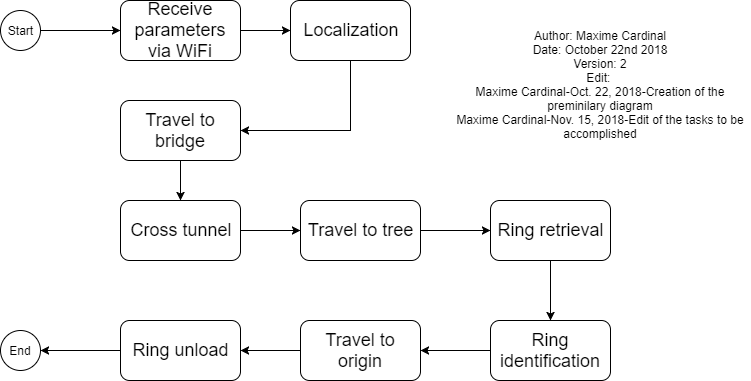
5.1 Feature

5.2 Software

6.0 Glossary of terms TODO

**2.0 Functionality**

The robot must execute many independent tasks to reach its goal. First, the robot must receive the competition parameters via Wifi and localize itself in the grid. Then, it must travel to the tunnel, cross that tunnel and reach for its team corresponding tree, while correcting its position. Afterward, it must retrieve one or many rings and identify their corresponding color. Finally, the robot must travel back to its original position via the tunnel and unload the ring(s) it retrieved (see Figure 1).



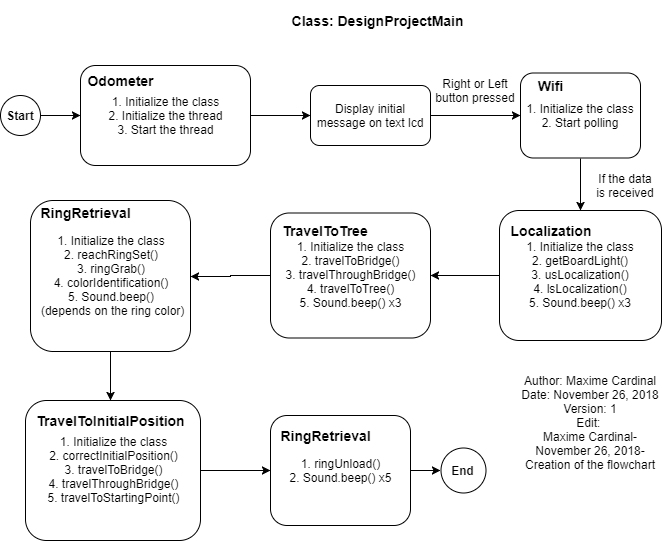
*Figure 1 - Functionality Flowchart*

The following pages further explain how those tasks are executed by the software and what classes are responsible for handling those tasks.

**3.0 Classes**

**3.1 DesignProjectMain**

The “DesignProjectMain” class is the main class of the project. It is responsible for launching the program, initializing all the sensors, all the motors and the important constants. Also, this class is responsible for starting required threads and calling required method from other classes. We decided to initialize all sensors and motors in this class to minimize our time consumption when starting new threads. Furthermore, by calling methods from other classes instead of creating threads, we minimized the number of threads required thus reducing the chances of program failure due to overhead.



*Figure 2 - DesignProjectMain Flowchart*

**3.2 OdometerData**

The “OdometerData” class is responsible of keeping track of the robot’s location and orientation. It stores and provides a save access to the odometer data. It contains methods such as “getXYT()”, “update(double dx, double dy, double dtheta)”, “setXYT(double x, double y, double theta)”, “setX(double x)”, “setY(double y)” and “setTheta(double theta)”, which can be used to access odometer data easily. This class has been reused from previous lab.

**3.3 Odometer**

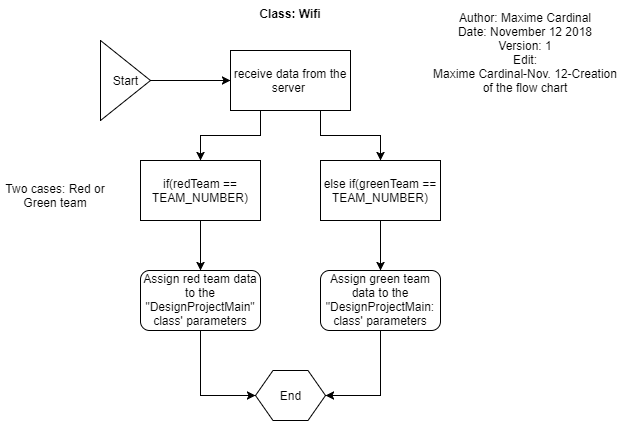
The “Odometer” class is responsible of updating the odometer data according to the robot’s wheels displacement. This class extends the “OdometerData” class and has been reused from the previous lab we did. The “Odometer” class is running as a thread by the main class “DesignProjectMain” and runs until the end of the whole program.

**3.4 OdometerExceptions**

The “OdometerExceptions” class is used to handle errors regarding the singleton pattern used for the odometer and “OdometerData”. This class has been reused from the previous lab and was provided to us.

**3.5 Wifi**

The “Wifi” class is responsible for connecting the EV3 brick to a server, receiving the game parameters and assigning the required game parameters depending on the team’s color. The “Wifi” class make use of an imported library to create a WifiConnection and receive the data from the server. Both the class and the library were provided, but we had to change the Wifi class so that it only assigns the desired parameters to the project. The Wifi class first retrieves the data from the server, then compares the robot’s team number to each parameter team number (green/red) and assigns the parameters according to the team color (See Figure 3).

****

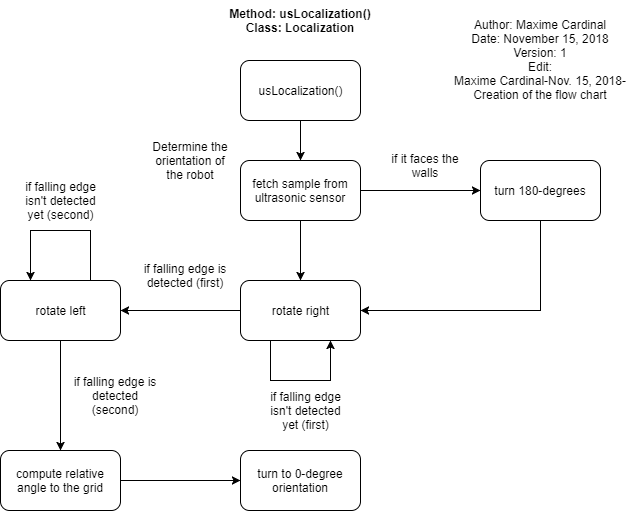
*Figure 3 - Wifi class Flowchart*

**3.6 Localization**

The “Localization” class is responsible of correcting the initial position and the orientation of the robot to be (0,0) and 0-degree respectively. This class has been implemented using the class “UltrasonicLocalizer” and “LightLocalizer” from Lab5. The “UltrasonicLocalizer” class was responsible of correcting the robot orientation, making use of an ultrasonic sensor and the “LightLocalizer” class was responsible of correcting the initial position of the robot making use of a light sensor. We decided to merge these two classes into one to minimize the time consumption of the process. By merging these classes, we reduce the number of classes needed by one, thus reducing the time needed to initialize sensors, variables and constants. The code has been further simplified to increase its readability by making use of multiple methods. This class contains is separated into two main methods: usLocalization() and lsLocalization().

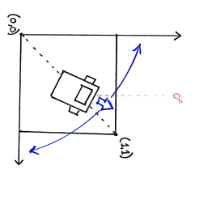
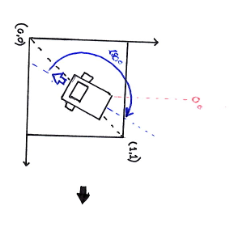
**3.6.1 usLocalization()**

The “usLocalization()” method is responsible for correcting the initial angle of the robot in the grid. To do so, this method makes use of an ultrasonic sensor to detect its position relative to the walls. Using a falling edge algorithm, the robot will compute the angle difference between the left and right falling edge to determine its orientation in the grid (see Figure 4).



*Figure 4 - usLocalization Flowchart*

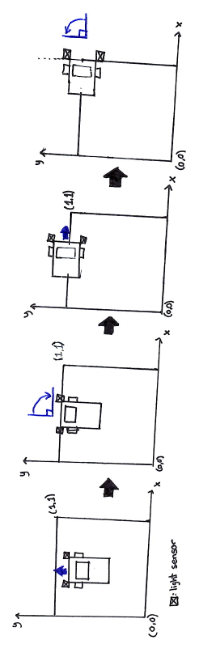
To filter the sensor samples, we restrained the falling edge detection to be from 0 to 40 cm, thus reducing the probability of detecting false falling edge at long distances. Since falling edge implementation requires the robot to be facing the outside of its starting corner, the sensor first detects if it is facing the good direction or not. If not, it will perform a 180-degree rotation before starting the falling edge detection (see Figure 5).



*Figure 5 - Falling edge: Starting orientation correction*

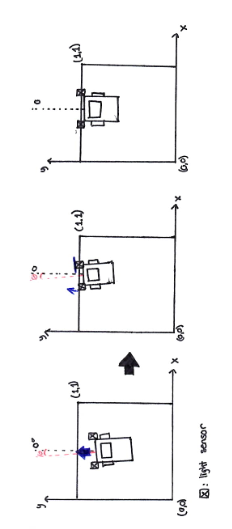
**3.6.1 lsLocalization()**

The “lsLocalization()” method is responsible for correcting the initial position of the robot to its closest grid intersection. To facilitate the understanding, the starting corner of the robot will be defined as 0 and its closest grid intersection will be defined as (1,1) for the following explanations. To correct the robot’s position, we used to only use one light sensor. At first, the light sensor localization algorithm was simple: the robot moved straight forward until it detected a line with the left light sensor, then it turned 90-degrees right and repeated the same process to correct its position to be (1,1) (see Figure 6).



*Figure 6 - initial light sensor localization algorithm*

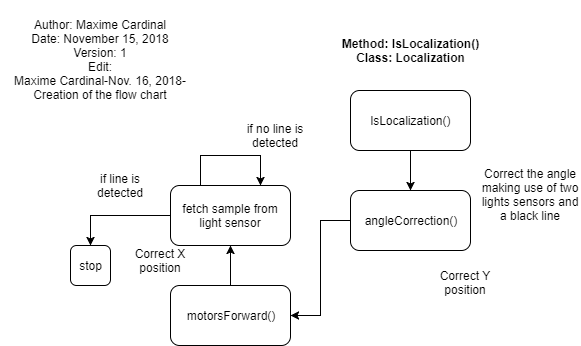
After review, it was concluded that this algorithm wasn’t a reliable position correction, as the its totally depends on the ultrasonic localization accuracy. An offset in the ultrasonic localization would directly impacts the precision of this method, which is an undesirable effect and had to be corrected. Thereby, we came up with our second light sensor localization, which made use of two light sensors and a more complex algorithm. Instead of assuming the orientation of the robot is good when crossing the lines, the robot makes use of the light sensors to correct its orientation again. The algorithm goes as follow: The robot moves forward until one of the sensors detects a line. Then, it stops the left or right motor according to the sensor that detected the line. Finally, it turns until the second sensor detects the line (see Figure 7).



*Figure 7 - Second light sensor localization algorithm*

This light sensor localization ensures that the final position and orientation of the robot

after executing the localization are respectively (1,1) and 0-degree (see Figure 8).

**

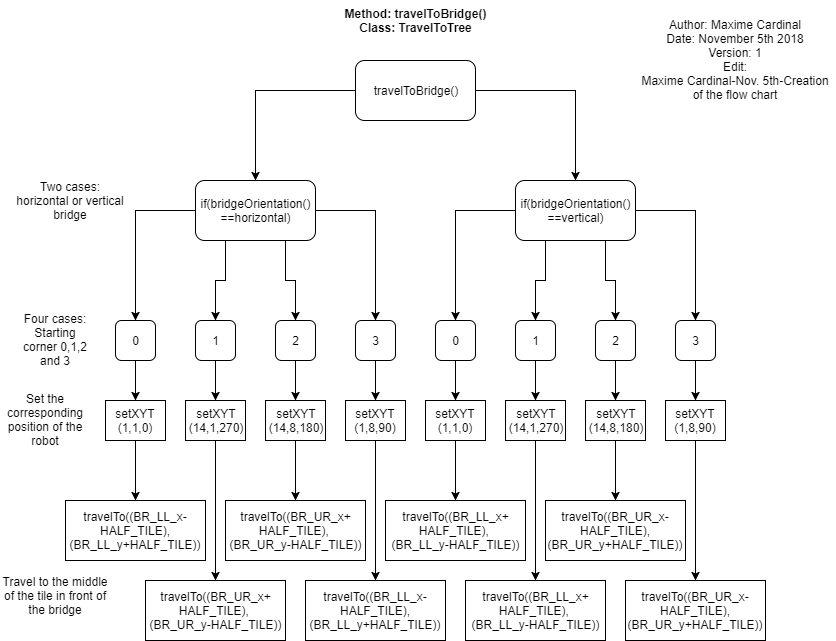
*Figure 8 - lsLocalization Flowchart*

**3.7 TravelToTree**

The “TravelToTree” class contains the methods responsible for travelling the robot from its starting corner to the ring set. To simplify the implementation and readability of the class, we divided the tasks the robot has to accomplish into specific methods, which are travelToBridge(), travelThroughBridge() and travelToRingSet(). Also, this class contains an essential method called “odometerCorrection()” which is responsible for correcting the x, y and theta of the odometer. This class requires 2 light sensors, which will be used for position correction, and 2 large EV3 motors used for travelling.

**3.7.1 travelToBridge()**

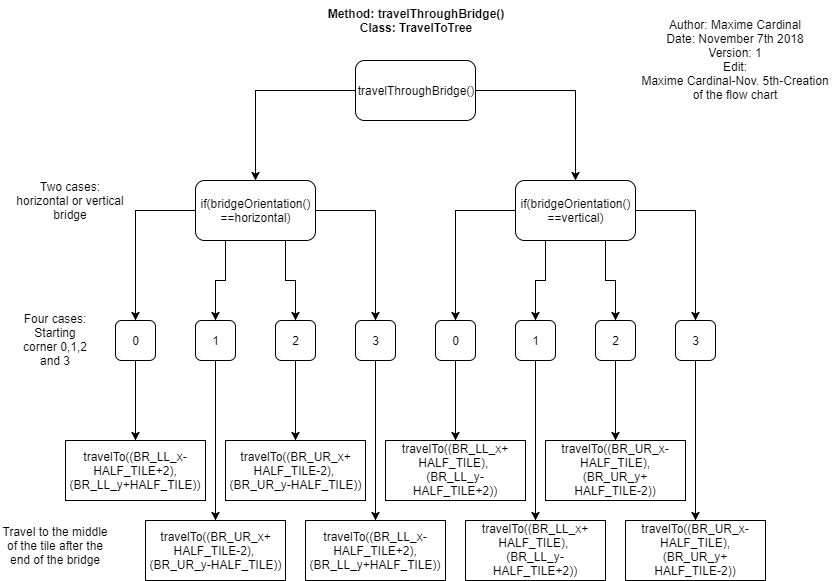
The “travelToBridge()” method is responsible for navigating the robot from its initial position to the tile right in front of the bridge entrance. This method first considers the orientation of the bridge (horizontal vs vertical), then consider the starting corner of the robot (0,1,2 or 3) and travels the robot according to that information (see Figure 9). The method supports both overlapping and non-overlapping bridges. Also, this method calculates the size of the bridge according to the lower left corner and upper right corner coordinates of it.



*Figure 9 - travelToBridge() Flowchart*

**3.7.2 travelThroughBridge()**

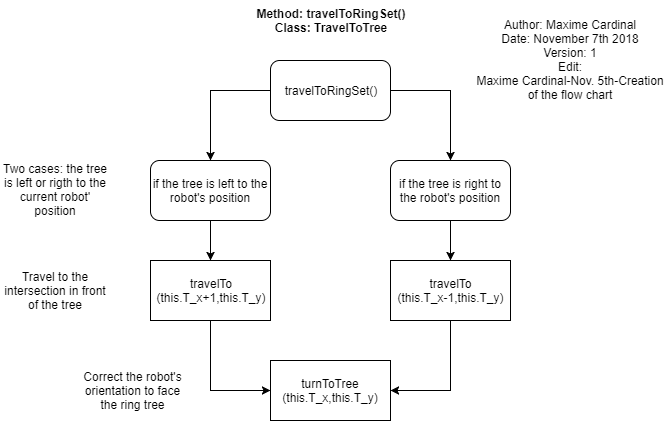
The “travelThroughBridge()” method is responsible for travelling the robot through the tunnel and stopping it at the middle of the tile right next to the tunnel’s exit. This method has to consider the orientation of the bridge, as it must travel in the x axis if the bridge is horizontal or in the y axis if the bridge is vertical. Then, it must consider the starting corner of the robot, as it changes the direction in which it must travel (+x/-x or +y/-y). Also, it must consider the size of bridge calculated by the travelToBridge() method, as it changes the displacement the robot must execute to cross the bride. The following flow chart describes this algorithm (see Figure 10).



*Figure 10 - travelThroughBridge() Flowchart*

**3.7.3 travelToRingSet()**

The “travelToRingSet()” method is responsible for travelling the robot from the exit of the tunnel to the closest grid intersection from the ring set. Then, it also corrects the orientation of the robot so that it exactly faces the ring set after the method execution. Thus, this method is the last step before the ring retrieval. The following flow chart describes the method algorithm (see Figure 11).



*Figure 11 - travelToRingSet() Flowchart*

**3.7.3 odometerCorrection()**

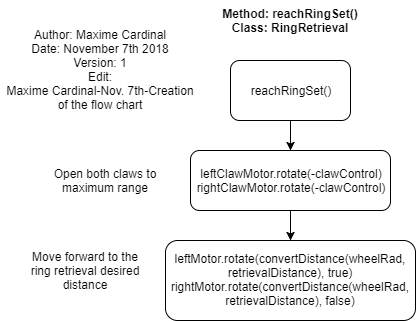
The “odometerCorrection()” method is responsible for correction the x and y coordinates of the odometer as well as the theta by using a grid intersection as a reference point. The correction process is similar as the angle correction used in the “lsLocalization()” (see Figure 7). However, since we want to correct not only the angle but the x and y coordinates of the odometer, the robot executes two consecutive corrections, one in the x and one in the y axis, to reposition itself at the grid intersection. Using this method, we can make sure our robot is properly travelling to the desired destination as we correct its odometer multiple times during the travelling process. Since some of the travelling tasks required more precision, the robot corrects its odometer before entering the bridge, after exiting the bridge and before retrieving the ring.

**3.8 RingRetrieval**

The “RingRetrieval” class contains the methods responsible of retrieving a ring from the tree, identifying the color of the ring and unloading the ring. To implement this class, we made use of methods and constants from the ring identification class of Lab5.  It requires a light sensor, which will identify the ring color, and two large EV3 motor, which control the claws for the ring retrieval. This class is separated in four main methods: reachRingSet(), ringGrab(), colorIdentification() and ringUnloading().

**3.8.1 reachRingSet()**

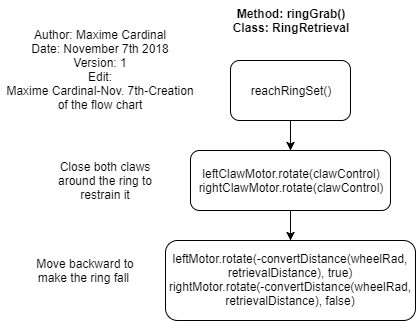
The “reachRingSet()” method is responsible for opening the robot’s arm to prepare it for ring retrieval and forwarding the robot to the desired retrieval distance from the tree (See Figure 12).



*Figure 12 - reachRingSet() Flowchart*

**3.8.2 ringGrab()**

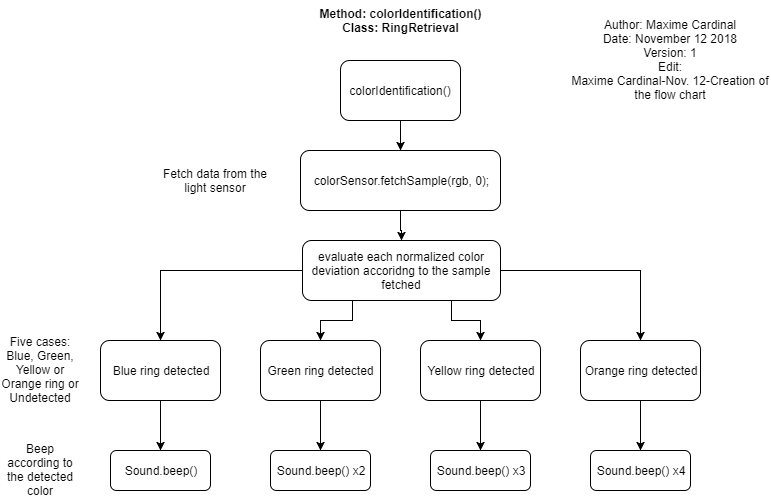
The “ringGrab” method is responsible for closing the claws to the required angle for the ring retrieval and moving the robot backward to the closest grid intersection. While doing so, the robot retrieves the ring from the tree as it is restrained by the claws. The following flowchart describes the algorithm of the method (See Figure 13).



*Figure 13 - ringGrab() Flowchart*

**3.8.3 colorIdentification()**

The “colorIdentification” method is responsible for identifying the retrieved ring’s color, by making use of a light sensor. This method implementation was easy, as it was already implemented during the Lab 5 (See Figure 14). We made use of normalization when identifying the ring’s color, as it is a more suitable algorithm for the method. With normalization of the samples, the color identification is independent of the lighting, thus offers more capabilities to the robot.



*Figure 14 - colorIdentification() Flowchart*

**3.8.4 ringUnloading()**

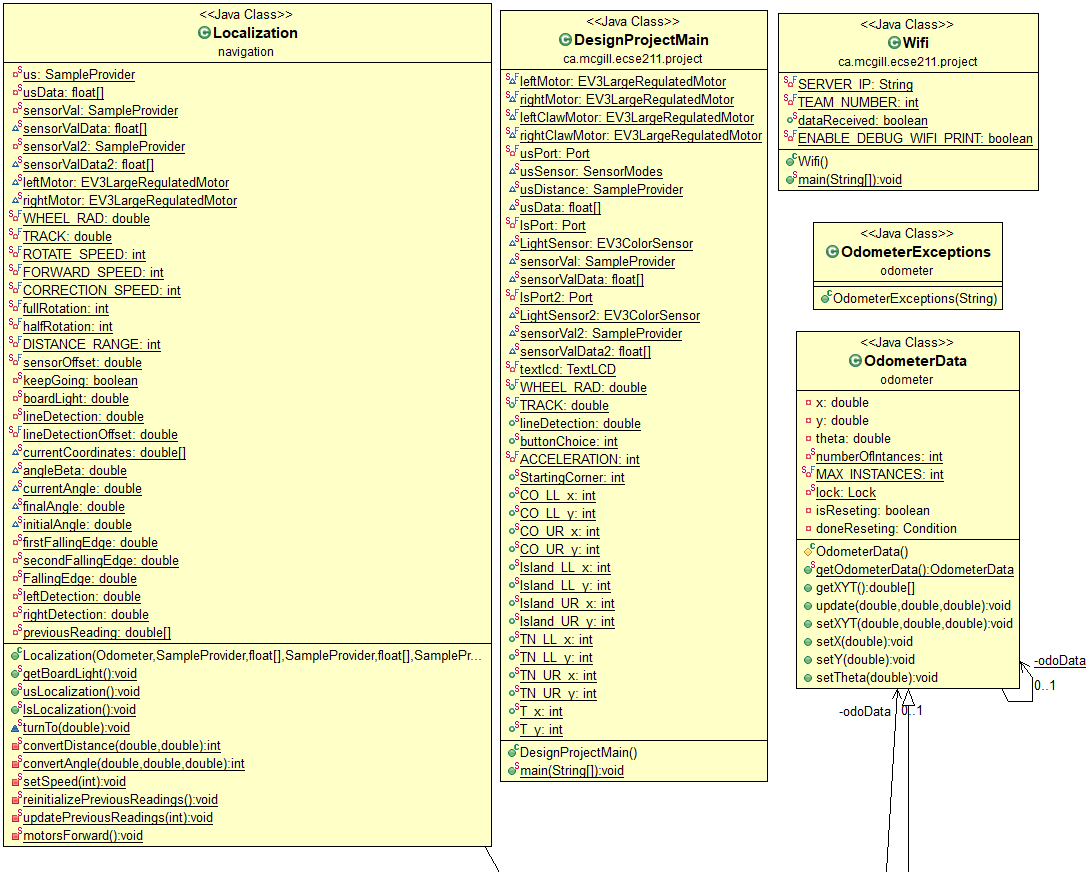
The rindUnloading() method is responsible for dropping the retrieved ring in the starting corner of the robot. Since the only thing preventing the retrieved ring from falling is the two claws, opening both of them at the same thing is enough for the ring to drop. Thus, the claw motors are rotating backward to drop the ring.

**3.9 TravelToInitialPosition**

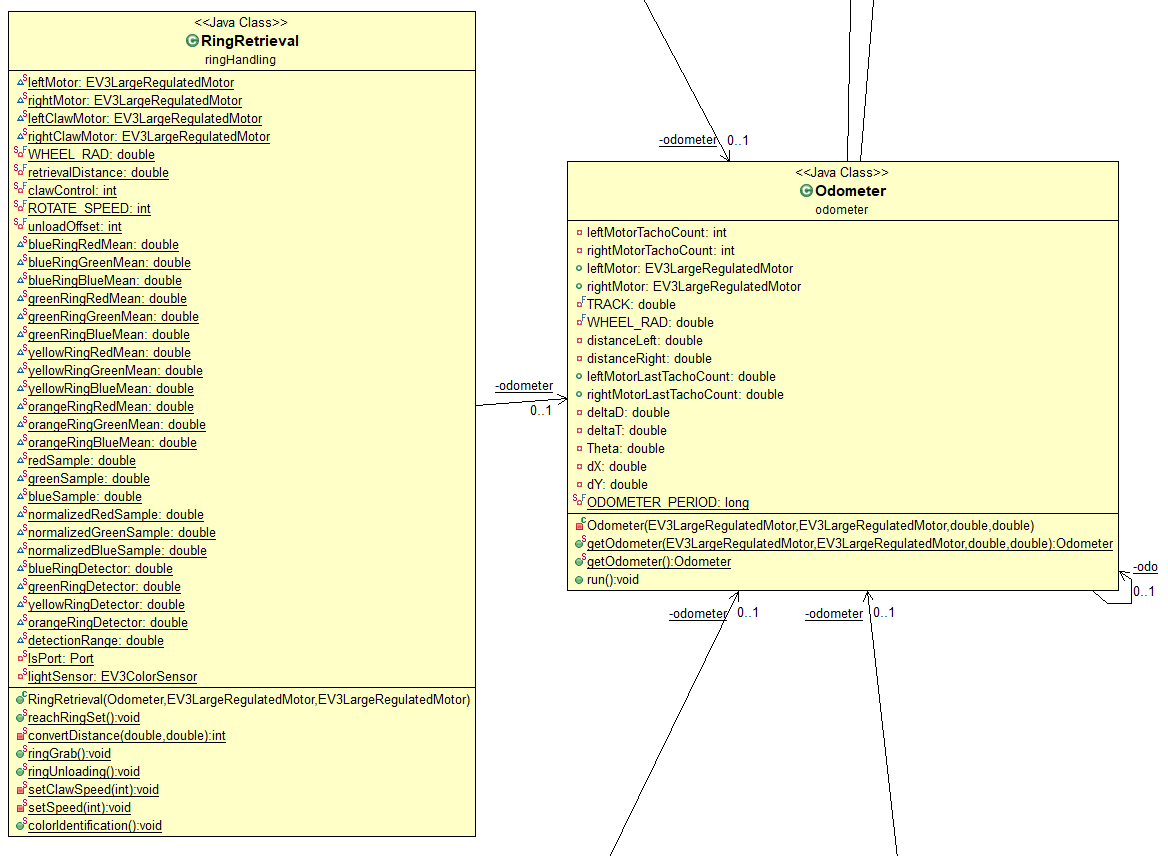
The “TravelToInitialPosition” class contains the method responsible for travelling the robot from the tree to its starting corner. This method basically consist of the same methods seen in the “TravelToTree” class, but executed in the opposite direction. Thus, the robot travels to the tunnel’s entrance, crosses the tunnel and reach for its starting corner while correcting itself. As for the “TravelToTree” class, “TravelToInitialPosition” makes use of two large EV3 motors to move the robot and two light sensors for odometry correction.

**3.10 Class diagram**

The following diagram (see figure 15, 16 and 17) shows how the classes are linked.

****

*Figure 15 - Software class diagram (part 1)*

****

*Figure 16 - Software class diagram (part 2)*

****

*Figure 17 - Software class diagram (part 3)*

**4.0 Decisions and discussions**

**TODO:** decision for travelling to ring set, small components concerning ring retrieval, why sometimes even this correction misses detections

**4.1 Thread vs methods**

As mentioned in the Meeting Minutes document in the beta demo section, the light sensors were unable to detect a line to locate and correct to resulting in it going in a completely wrong direction and failing the demo. There were two leading reasons to this which was the method we were using for line detection which could largely be affected by ambient light and elaborated upon further later, however there was also a possibility that it stemmed from something that had been previously observed during testing stemming from threading. The EV3 brick is very limited in its computing power, particularly in terms of the amount of threads it’s able to handle at once. It had previously been observed that when too many threads are running, if a light sensor crosses a line to begin correction, the thread reads in the data and begins operations, however the brick may become overloaded and when the other light sensor calls a thread of its own to also contribute to the correction, the system overloads the hardware and the latter thread simply does not run, which resulted possibly in the lack of detection of the line.

The devised way to prevent this (as well as modifying our light sensor methodology) was to limit to an extreme extent the amount of threading our robot undertakes. Instead we shifted to a system of using many more methods rather than threads so there is less concurrent operations to overload the hardware. As mentioned in many of the class descriptions above, a vast amount of the functionality was shifted from being classes threaded together to calling methods at appropriate times. This effectively eliminated one potential source of error from the observed results and feedback of the beta demo.

This also made the code much more legible since with less concurrent run() commands and more method calls in sequential order, the software processes were much easier to follow rather than keep going tracking of many simultaneous actions. It was tested and works, meaning the methods function as expected and present a solution to keep. For example, instead of previously threading many localization functionalities, they’ve now been converted to respective methods done in sequence which removes a lot of unnecessary overhead computing cost for the limited brick and showcases the effectiveness of converting from threads to methods.

**4.2 Adjusting Localization to constraints**

As it was discovered following the beta demo, our understanding of the localization requirements was slightly shifted. The main motivators for the description and changes described in the localization classes and method in section 3 stem from the accuracy of the localizing and the time constraint of 20 seconds. Previously the robot orientated itself and located itself approximately to the desired corner, however as said previously it was often viewed to be inaccurate. The solution to this was to undertake a correction on the corner after correcting. However, the beep emitted by the robot was done before this correction, which by the requirements of the project would signal the end of all localization and signal the beginning of travelling to the tunnel, not starting another correction before travelling. We could not simply beep after the further correction since it would then break the 30 second limit. We took this opportunity following that feedback to improve our localizing process (combined with the new method methodology of software implementation).

The first change made was to accelerate the speed of localizing with the ultrasonic sensor. This portion never provided much problem, the correction that had to be done within the localizing and beep sound had to be modified from our original design. Instead of driving past the lines and returning, now the correction to the corner was simplified to almost mimic the correction during travel where we use both light sensors and only do a single correction per axis (which was not the case previously). This drastically shortened the time for localizing while not compromising the accuracy of the solution as well as fitting within the criteria which was brought to our attention following the beta demo.

**4.3 TravelTo; XY vs Diagonal**

There is a balancing act to strike in all aspects of travelling with the robot between the speed of execution that travelling diagonally the shortest distance to a desired location and the accuracy that travelling only along axis with many more corrections undertaken. Initially, most travel methods and classes implemented a hard coded, calculated shortest distance to travel in a certain direction and while it worked to a certain extent, following the beta demo, like many other things it was put to question. The beta demo also failed partly due to a single correction/secondary localization near the tunnel being responsible for ensuring the robot maintains proper location, yet if this fails, the entire system fails. While we are starting to feel the weight of the time constraint requirement, however regardless of the longer time it would take to operate along the axis trajectories, to fail a single correction following diagonal travelling would be more damaging than accomplishing more successfully but slowly. A compromise was made that the travel to tunnel method would be along the axis and the travel to tree function would be a discussion reserved for next week following the observations of this decision.

TravelToBridge() will therefore be implemented along the XY axis, with periodic correction at each intersection. To reduce slightly the time this would take, the speed between intersections was decided to be increased, however, when the correction begins (within a certain range of the intersection based on known distances travelled) it would slow down to allow for the accuracy of the correction. The previous versions of this didn’t work mainly due to our observed and potential inaccuracies due to a diagonal method of travel and this compromise was deemed a solution that struck a fair middle ground of accuracy and time to great effect.

After a revision of the correction algorithm, we the travelling of the robot was changed back to diagonal travelling. Even though the X and Y axis travelling is more accurate than the diagonal one, we observed that the first travelling method is way slower than we thought. We then had to decide between having a more accurate travelling or a faster one. After further testing of the diagonal travelling we concluded that its accuracy was acceptable enough and that we would rather cut on the accuracy of the travelling than cutting on the precision of the travelling.

**4.4 Correction decision**

While it was a topic of discussion with the TA in terms of the method to implement for the odometer correction, our observations after have shown that the correction is very accurate if the speed is appropriate. With the decision being made in the travel to portions of the software, it was deemed necessary to slow down the robot near intersections so it may accurately correct. This correction became more impactful following the decision to travel along the axis. Therefore with this proven accuracy and simplification of other components, the decision was made to remain with our current design for correction. However the light sensors would be less likely to miss any lines due to the new method implementation over threads. On top of this, there would be multiple checks for correction rather than a single. To summarize we decided to stay with a wheel pausing at the line, waiting for the other to align appropriately and then restarting in sync rather than calculating the difference of separation the wheels have after scanning the line and rotating to adjust.

**4.5 Light sensors**

One of the most pressing issues we had to address this week, especially based on the tones of the professors when they saw our beta demo, was that up to this point we had assume that an absolute value threshold for detecting colors would be sufficient throughout the project. However, as mentioned often in the main docs, the ambient light is a very significant factor, particularly during the demos when we are no longer in the controlled environment of the lab. A priority for week 5 was therefore to implement a new method of detecting lines with the light sensors. Two options were available to us, one which was a continuous while loop which took in the current values (regardless of the color) and if it was detected that there was a significant change, it would mean the robot was detecting a different color based on the difference of input ID, and would therefore count as a line. This was an interesting idea but building off the logic we had of avoiding too many concurrent activities during operation, it was decided against. It was also decided against because the original idea we had as to how we may implement a differential color correction was simpler to implement in our current software versions and worked well. This method at the very start of the demo, while initializing all the values, reads in the current color code of the tile it sits on multiple times and assembles an average. Therefore regardless of light conditions or the color of the floor, there will always be a base value associated to the color of the tile, whatever it might be. Then, this value will be compared to any readings the light sensor might make. Should this exceed a certain boundary, it would also mean that there was a color change in the form of crossing a line. This differential system, may be tested to determine what the best difference to look for might be and also avoids any ambient light or changing board conditions. This was the main source of error our previous design faced during the beta demo in a different light condition. It was crucial to implement this change since the location and adjusting of odometry are central to the proper operation of the tasks, no matter the circumstances which surround those tasks. The problem with our beta demo and line detection was primarily due to detecting lines using their absolute values which in shifting light conditions were heavily falsified resulting in a failed demo, however with scanning the values, whatever they might be, and using a differential threshold for the latter to detection lines, we’ve developed a solution that has been observed to function as wanted under the varying conditions.

**5.0 Improvements**

**5.1 Features**

Object Avoidance

As the robot could encounter another robot during the demo, it would be relevant to add an avoidance method so that it can avoid the incoming robot and continue its tasks afterward. By making use of the ultrasonic sensor, the robot could detect when something is getting too close and avoid it.

**5.2 Software**

Ring Retrieval

The ring retrieval process doesn’t take into account whether or not the robot has retrieved a ring and returns to the origin no matter what happened. By implementing a method that verifies if the robot successfully retrieved a ring, the robot could move to another side of the tree if it didn’t pick any ring and re-do the ring retrieval process. That way, the robot is almost certain of travelling back to the origin with a ring in its claws.

Software Speed

Since the software is not optimal and its complexity isn’t minimized, the robot is sometimes victim of some lags. By optimizing the software complexity, which means reducing the number of variables, methods and classes required to accomplish the tasks, the robot would be lag free and thus increase the chances of success of the design.

**6.0 Glossary**