**Software document**

**Author:** Maxime Cardinal

**Date:** October 30th, 2018

**Version:** 3.3

**Edit history:**

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

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

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

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

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

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

TODO: elaborate on design flow of classes/development process (descriptions all present, modifications and decisions on each are not fully yet)

**1.0 Table of contents**

2.0 Functionality

3.0 Classes

3.1 DesignProject

3.2 OdometerData

3.3 Odometer

3.4 OdometerExceptions

3.5 OdometerCorrection

3.5.1 Description

3.5.2 Week 2

3.5.3 Week 3

3.6 Localization

3.7 TravelToTree

3.7.1 Description

3.7.2 Week 3

3.7.3 Week 4

3.8 Ring Retrieval

3.9 TravelBack

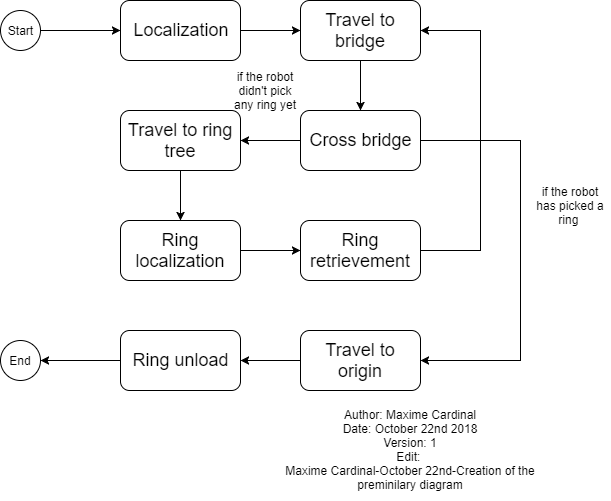
3.10 RingUnload

3.11 Class Diagrams

4.0 Glossary of terms

**2.0 Functionality**

The robot must execute many independent tasks to reach its goal. First, the robot must localize itself in the grid, then travel to the tunnel, cross that tunnel, reach for the ring tree, localize the rings and retrieve one, then travel back to its original position via the tunnel and unload the ring (see Figure 1).



***Figure 1 - Functionality flowchart***

**3.0 Classes**

**3.1 DesignProject**

The “DesignProject” class is the main class of the project. It is responsible for launching the program, initiating all sensors, all motors and some constants, as well as starting required threads. We decided to initialize all sensors and motors in this class to minimize our time consumption when starting new threads. We make use of the “.join()” method to make sure each thread is executed in the desired order.

**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 “DesignProject”.

**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 we did.

**3.5 OdometerCorrection**

**3.5.1 Description**

The OdometerCorrection class is implemented to correct the angle and the coordinates of the robot when it reached to the final point. The first week, we tried to implement the correction method by using 2 light sensors right next to wheels. This way when one of them sees the black line it stops and the other one continues until it sees the line. The problem was that, one of them would see the line after it passes the line and the other one would see the line and stop when it sees the half of it. This also created an angle which we were trying to avoid.

**3.5.2 Week 2**

In the second week, we tried another implementation of the correction. Before, implementing we made sure the light sensors are as low as possible so that it won't be detecting a larger area. This time the sensor would detect the lines and once they detect the line we would get x or y coordinates from the odometer. We would find the difference of the distance and calculate the theta that needs to be changed. We calculated theta, by finding the sine function of change in difference over the track. And then we would go back to the closest place. This method didn't work perfectly as well, because the robot was never able to set the right angle or go certain distance. We estimated, it was probably because of the wrong calculated values. Since we are still working on the design, we will test that method after the hardware design is done.

**3.5.3 Week 3**

Third week, we came up with another method, where the robot would pass both lines and correct the angle afterwards. This methods seems to be working better than the other ones. But the theta value was of by a bit. During the the week, we were changing the hardware design. The track wasn’t thigh enough, therefore each time it turned, the center of the rotation was changing, which made the the correction method work less. We are implementing a new wheel design now. We have 3 different correction method versions now. We will be testing them on the following week. As we were discussing with the group members about the reason why it wasn’t working, we thought it might be because of the stop() method. Stop method might have been causing the robot to slide. One of the methods we created, the angle is corrected without stopping.

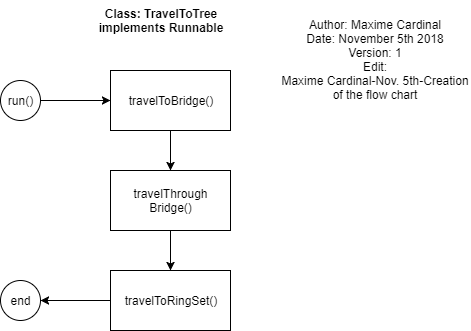
**3.6 Localization**

The “Localization” class is responsible of correcting the initial position and orientation of the robot to be (0,0) and 0-degree starting from the y-axis. 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 threads needed by on, thus reducing the time needed to initialize sensors, variables and constants. This class is running as a thread by the main class “DesignProject”. The code has been further simplified to increase its readability by making use of multiple methods.

**3.7 TravelToTree**

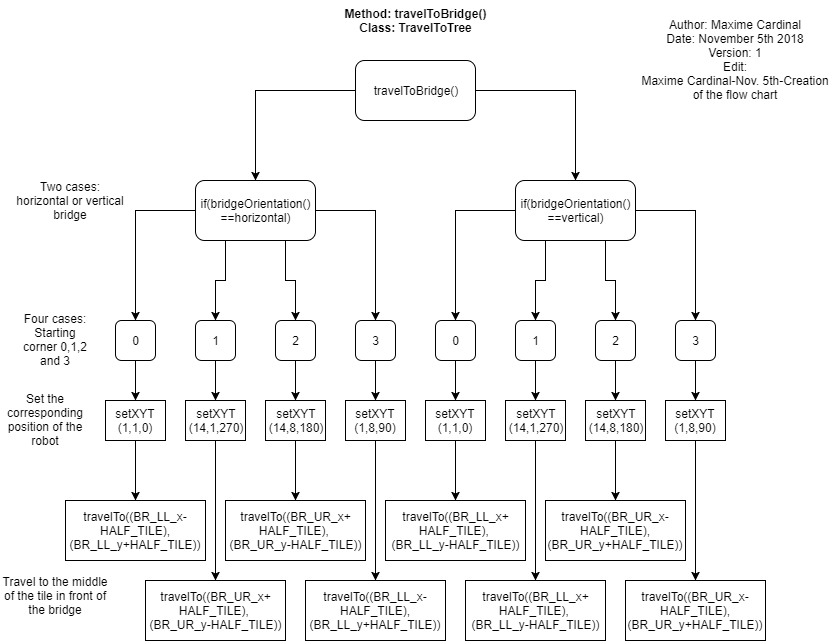
**3.7.1 Description**

The “TravelToTree” class will be responsible for the travelling of the robot from the origin to the tunnel, crossing the tunnel and guiding the robot to the ring tree. This class will be separated in three different tasks and will stop once each of them is executed. To implement this class, we will make use of methods implemented in previous labs such as “travelTo(double x, double y)”. It will require 2 light sensors, which will be used for position correction, and 2 large EV3 motors used for travelling. This class is running as a thread by the main class “DesignProject”. The main methods of this class are travelToBridge(), travelThroughBridge() and travelToRingSet() (see Figure 2).



***Figure 2 - “TravelToTree” class flowchart***

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 3). The method supports both overlapping and non-overlapping bridges.



***Figure 3 - “travelToBridge()” method flowchart***

**3.7.2 Week 3**

TravelTo was largely the focus of software for week 3.

As seen in the flow chart, the amount of cases that would be present in the final project are relatively limited and can neatly be all accounted for in a case by case basis. The premise of the class was based on the flexibility to handle the data received from the wifi class. This proper integration will be further elaborated upon when the wifi class is released.

The placement of the bridge, in an of itself within the context of the staging represents only two possible orientations given the 2D nature of the field. To simplify things, we have designed the code to evaluate the bridge as either a vertical or horizontal entity. The direction of travel is left up to other classes and will be expanded upon further but in terms of the movements based on the coordinates of the grid, the only thing that matters is the orientation. The latter can be determined simply by reading in the coordinates of the bridge to see along which side it is longer and appropriately setting the boolean for the case to enter.

The travelTo method relies heavily on the assumption that regardless of the corner it will handle, the localization of the robot has been effectively carried out, see 3.5, as all the numbers acquired from the wifi class are passed into travelTo and operated on as defined by the origin after localization.

The actual movement of the robot is done via a sort of brute force movement calculation. The robot however, must first orient itself in the proper direction. It utilizes a turnTo method within the travelTo class which as of right now, was recycled from previous lab iterations wherein it simply translates the values provided to it into radians the wheels may rotate in place. The value passed to the turn to stem from within the cases of the travelTo operation where the total distance the robot must travel, based on the coordinates passed to it and the determined location of itself on the grid are also applied to rudimentary arithmetic and the minimal angle (done using some if statements to catch fringe cases) of turning calculation done in a previous lab. Upon facing the right direction it travels in a straight line to its objective, which at this current point is right in front of the entrance of the bridge based on its coordinates

However, one small caveat that was noticed is as it stands, the robot directs itself to the corner of the tile where the bridge is located. Yet when taking into account the real world constraints of the situation, the robot must in fact arrive at the tile in front of the bridge and in the middle of such, not along any of the grid lines so that it may then go through the tunnel.

The current implementation of the class has the skeleton of the architecture we have determined to be most appropriate in approaching the problem and has laid the foundation for being able to determine where it finds itself in regards to the whole of the area after localization as well as accommodating for any situations that the bridge placement can bring about. The travelTo and turnTo operate well in their methodology and in the proper circumstances thanks to the aforementioned techniques and now must simply be put to use with appropriate values which are fed in through the wifi class or coded in to address constraints such as aligning itself with the bridge. They are currently being worked on but due to resource restriction in terms of the approaching deadline, will be discussed in the upcoming week after testing their efficiency similarly to how the previous components were approached. There remains to do the traversing of the bridge, the guiding to the tree and the return to initial point (those two methods and the one done this week in reverse) but as it stands now, the direction and consensus of the team was that they will not be too resource intensive to implement since the foundation has already been created this week and the wifi class has been uploaded.

Finally, one large obstacle in terms of the overall travelTo concept which will be carried into next week following very last minute testing during week 3 is the way in which it travels to objects. As mentioned previously the robot calculates the shortest path (a straight line) and crosses over tiles arbitrarily for that distance until arriving at its destination. It was seen that this presents a large amount of inaccuracies compared to the other option which will be elaborated upon further: travelling strictly on grid lines. It may be longer distance to travel but given the observed inaccuracies, travelling along the grid lines in a very linear fashion allows for constant correction with 3.5 OdometerCorrection.

**3.7.3 Week 4**

TODO: diagonal vs grid line driver

TODO: crossing bridge (accuracy of robot with tight size constraints as it currently stands)

TODO: travel to tree (same issues in the methodology as others?)

TODO: implications to accuracy of ring retrieval

TODO: travelto implications in travelback?

**3.8 RingRetrieval**

The “RingRetrieval” class will be responsible of localizing the rings in the tree, retrieving a ring from the tree and identifying the color of the ring. This class will be separated in three different tasks and will stop once each of them is executed. To implement this class, we will make use of methods and constants used in the ring identification class from Lab5. It will require a light sensor, which will identify the ring color, and a large EV3 motor, which will lift an arm for the ring retrieval. This class is running as a thread by the main class “DesignProject”.

**3.9 TravelBack**

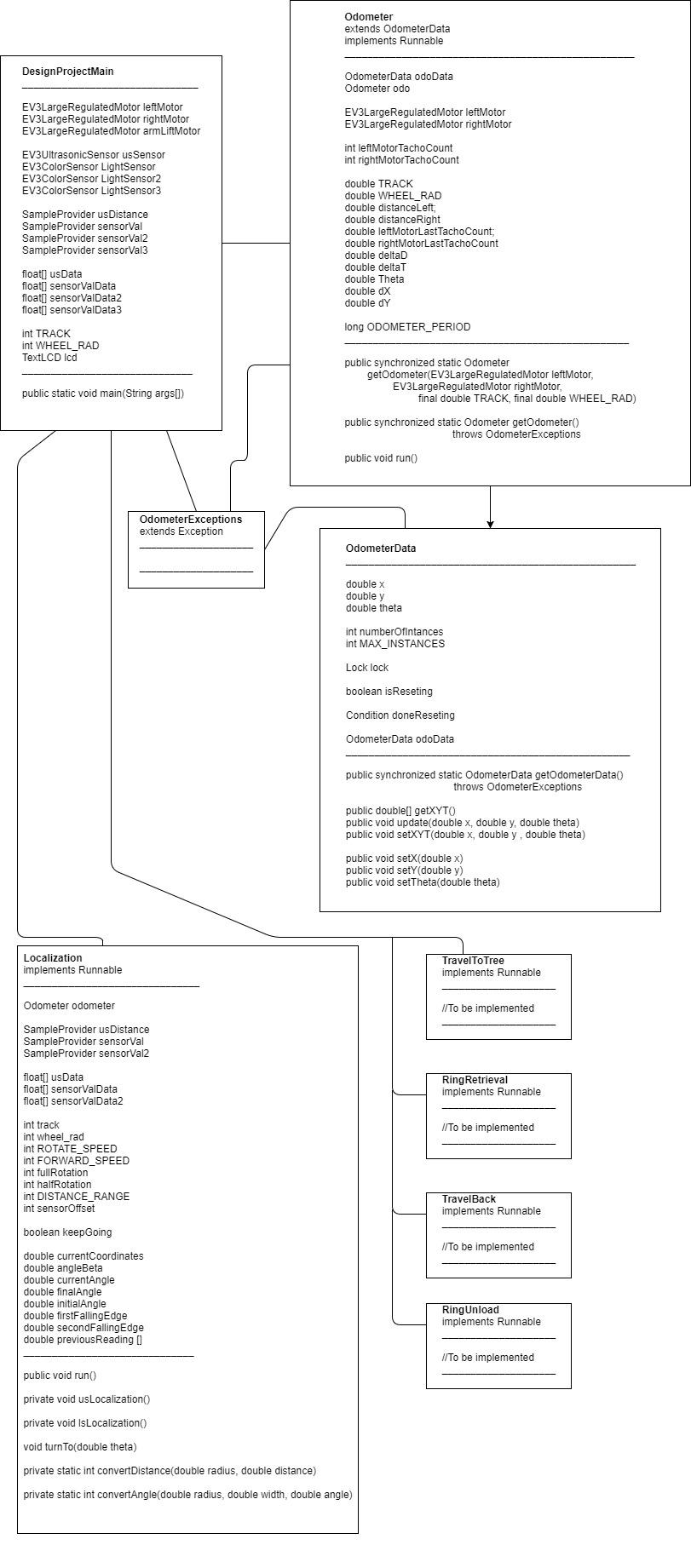
The “TravelBack” class will be responsible of travelling the robot back to the point (0,0) of the grid. It will execute the opposite function as TravelToTree. To implement this class, we will make use of methods implemented in previous labs such as “travelTo(double x, double y)”. It will require 2 light sensors, which will be used for position correction, and 2 large EV3 motors used for travelling. This class is running as a thread by the main class DesignProject.

**3.10 RingUnload**

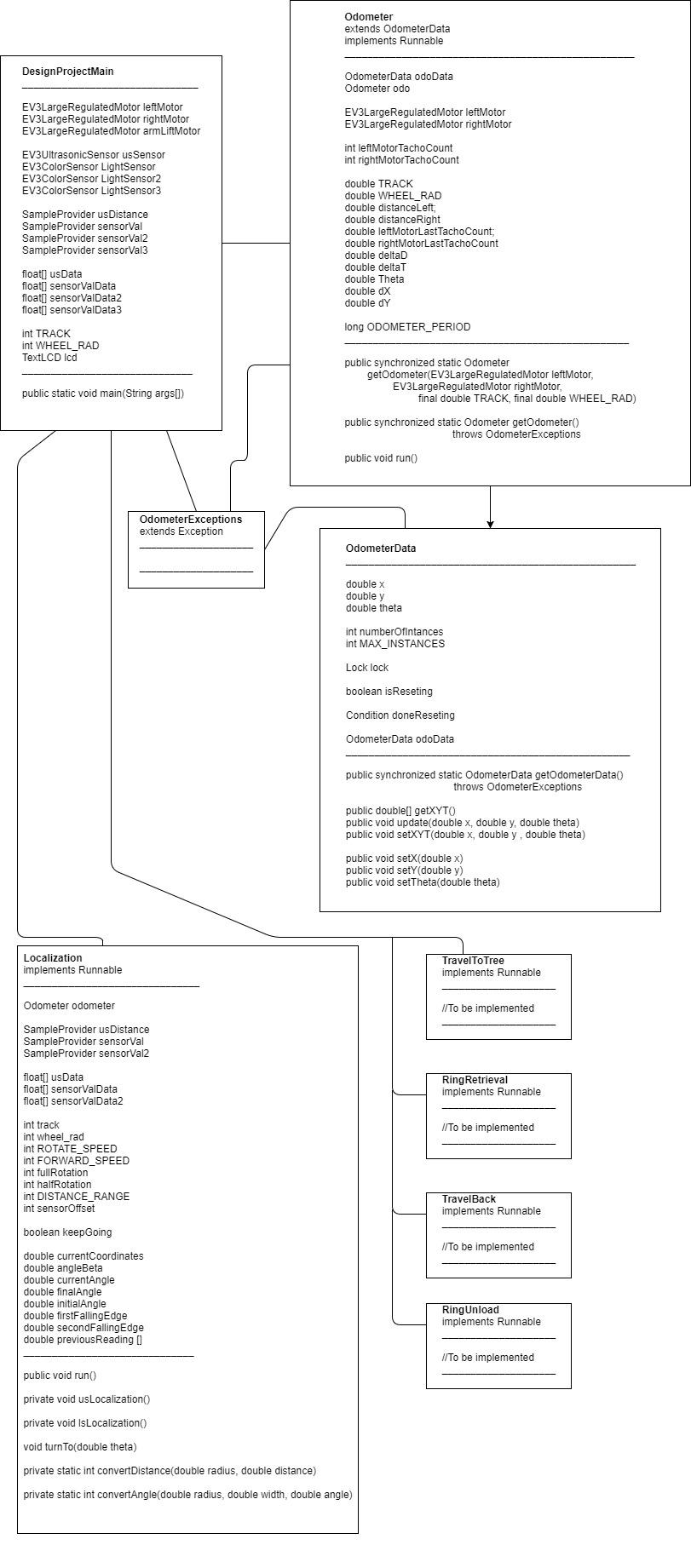
The “RingUnload” class will be responsible of unloading the retrieved ring in the starting corner of the grid. It will require a large EV3 motor to control the arm and drop the ring. This class is running as a thread by the main class “DesignProject”.

**3.11 Class diagram**

The following figure (Figure 2) represents how the classes will interact with each other and shows their content.



***Figure 4 - Class diagram (Part a)***



***Figure 5 - Class diagram (Part b)***

**4.0 Glossary of Terms**

TODO