Software Design Document

Project: LIBERTY

Task: SOFTWARE DESIGN

Document Version Number: 5.0
Date: 2017/11/28
Author: Bill Zhang
Editor: Andi-Camille Bakti
Edit History:https://github.com/Gabetn/DPM_01_Project_Documentation

McGill

TABLE OF CONTENTS

1.0 SUBSYSTEM DIVISION

Sub-System	Priority	Reason	Task
Localization & Correction	1	Before the robot performs any action, it needs to know its location and heading. Although they are determined by the odometer, the location must be updated and corrected at real time	Localize the robot using light and ultrasonic with wall and black lines. Correct odometry data by detecting black lines
Navigation & Zipline Traversal	2	Navigation is throughout the entire course including the traversal on the zipline. Before capturing the flag, the robot needs to go to a designated point and upon the completion, the robot needs to navigate back.	Navigate the robot between any two points including those on the zipline as well as avoid any obstacles along the way.
Capture	3	The core mission of the competition is capturing enemy's flag, but it is actually dependent on other courses of actions.	Determine the correct flag to capture in enemy's zone and beep three times to perform the capturing
WiFi	4	This sub-system is vital for the overall system because it receives all of data required, however, it is provided as a jar file externally. In terms of building the system, it does not have a high priority	Take the input coordinates from the server and pass it to the system in the correct format

Table 1: Subsystem priority list and tasks

2.0 CLASS DIAGRAM AND INTERACTIONS

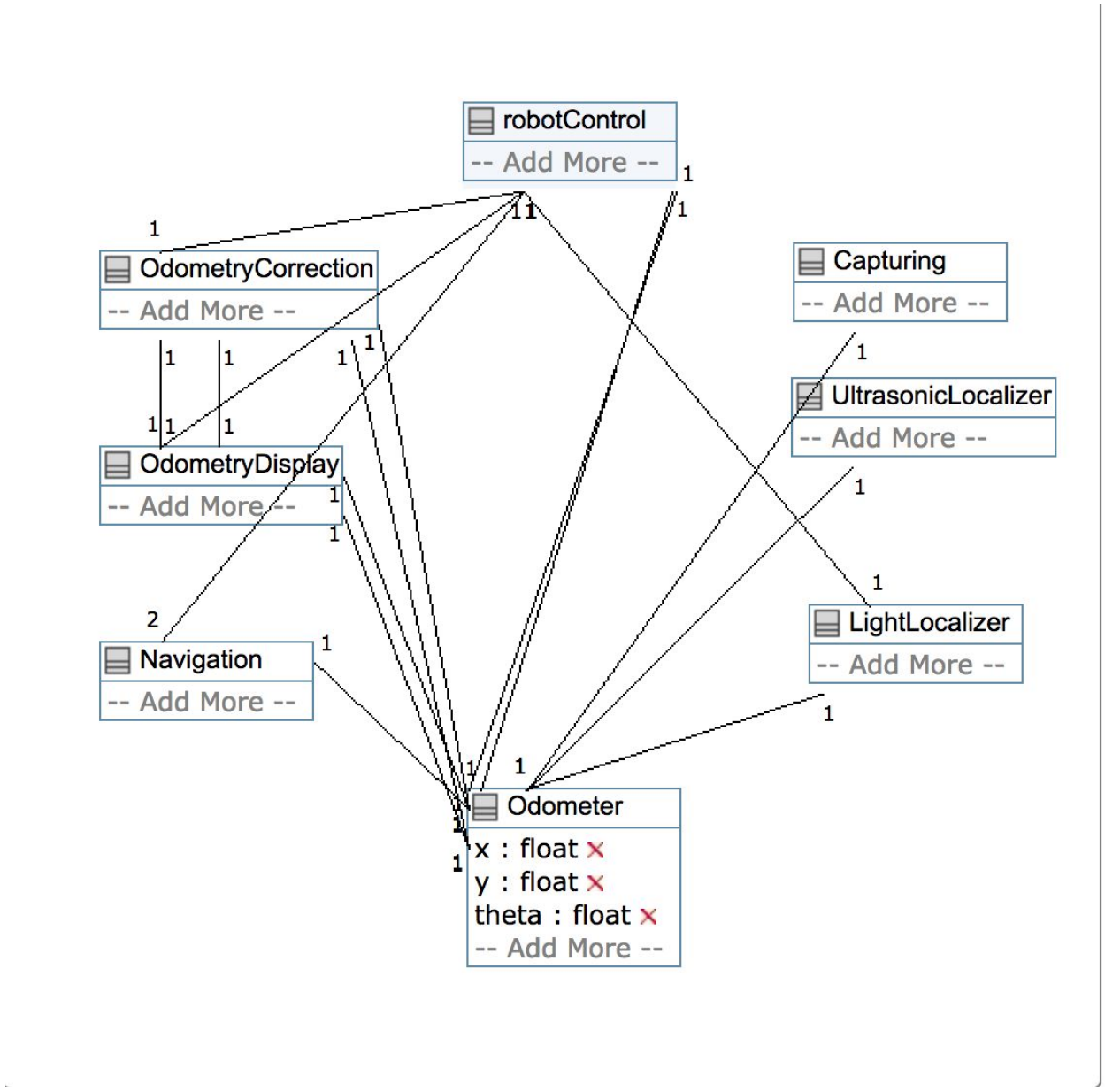


Figure 1: Class Diagram of the system

Each subsystem handles a specific task as mentioned in ***SOFT - SOFTWARE DESIGN; 1.0:***

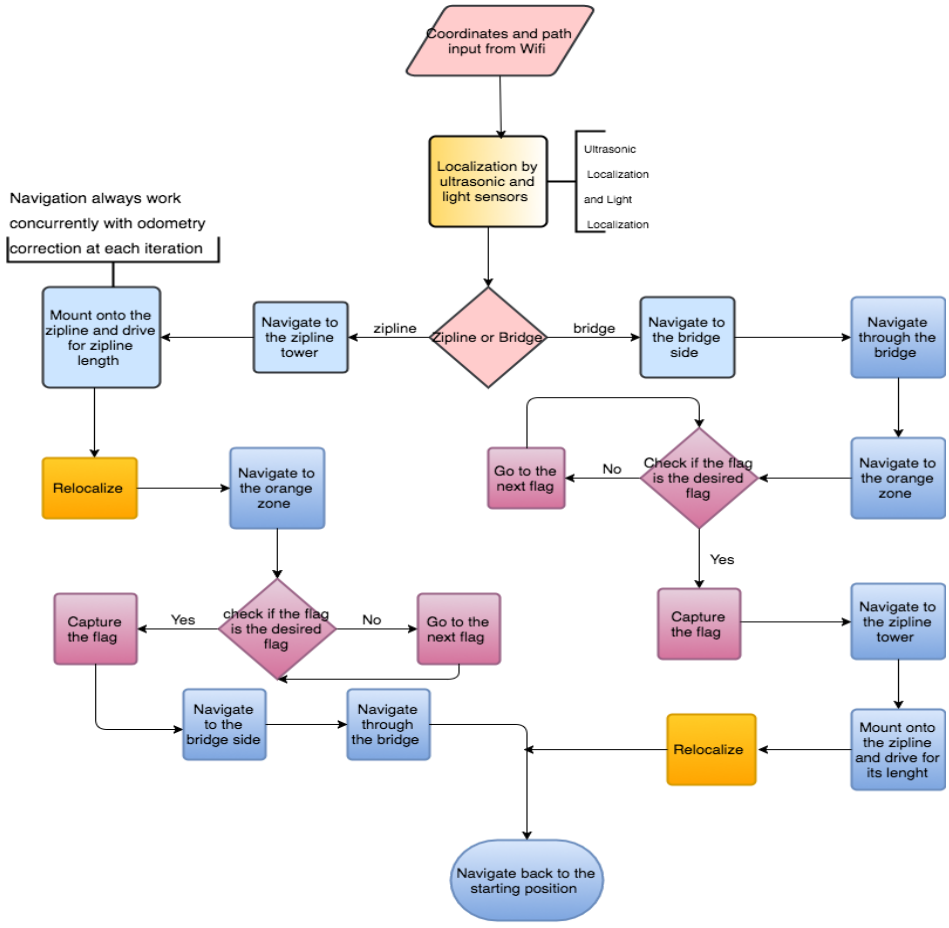
- **Robot Control:** This is the main class that handles the overall logic of the system and calls the other classes when needed. The main method is written in this class.
- **Localization** (*Ultrasonic* and *Light*): These classes performs the ultrasonic localization and light localization to ensure the accuracy of x, y and θ at the start of the round.
- **Navigation:** This class handles the displacement between two points and generates a path based on the points given. Two instances of this class should be running in the control class. One is for the two bottom motors for driving the system, and the other is for the zip-line traversal motor and the logic determining whether the system has landed.
- **Odometry Correction:** This class corrects the x, y coordinates and the angle θ using the two bottom light sensors when encountering a black line.
- **Capturing:** This class determines if the flag is in fact the enemy's flag, beeping 3 times to capture it.

3.0 CLASS DEPENDENCY

Class Name	**Dependency**
Robot Control	All other classes
Localization	Robot Control & Odometer
Navigation	Robot Control & Odometer
Odometry Correction	Robot Control & Odometer
Avoidance	Robot Control & Odometer
Capture	Robot Control
Wifi Communication	Robot Control
Odometry Display	Odometry Correction, Odometer, Robot Control
Odometer	Robot Control

Table 2: List of Class dependencies

4.0 OVERALL SOFTWARE WORKFLOW



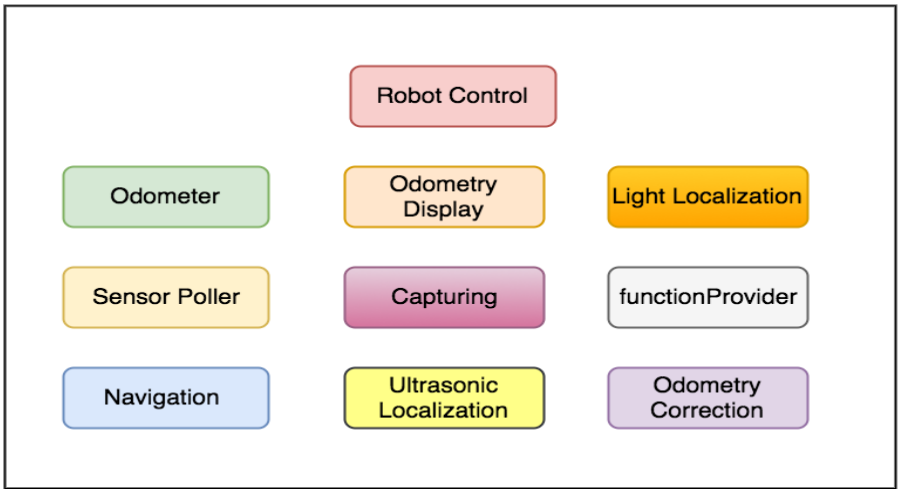


Figure 2: Flowchart of main software organisation

5.0 CLASS HIERARCHY

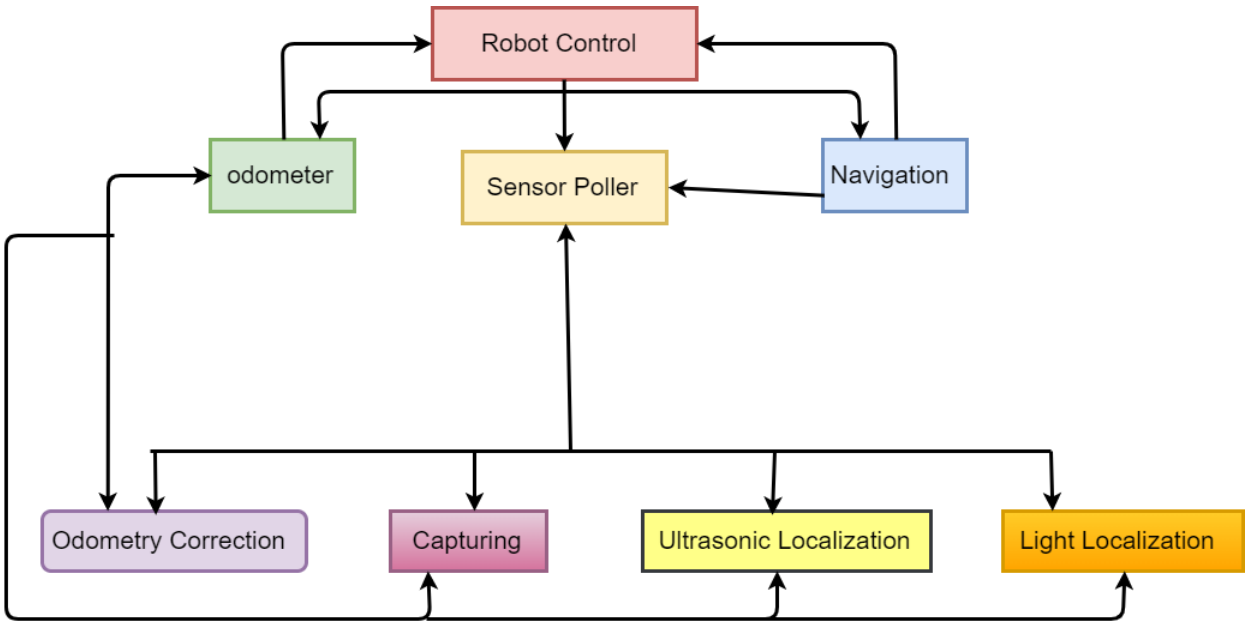
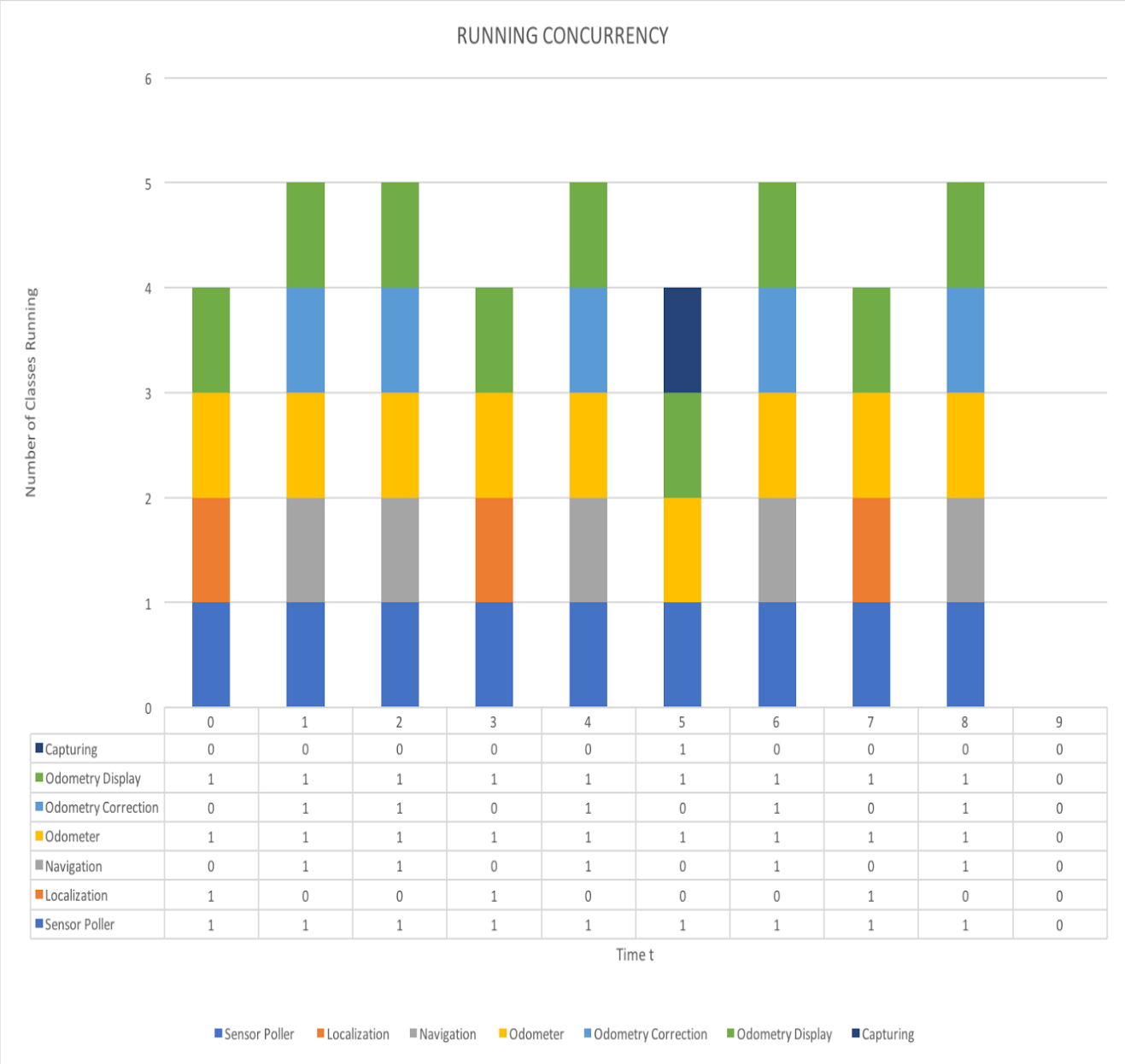


Figure 3: Class Hierarchy

There are three basic layers in the class structure in terms of hierarchy. The top layer is in charge of controlling the robot (Robot Control) which is a managing class that is responsible for calling the other threads and other classes in the system and the overall process. It is also the core of controller in our software architecture. The second layer is composed of the odometer and the sensor poller. The odometer layer is placed here because of the multiple classes that are dependent on it. The sensor poller class is where the instances for all the sensors are created. Due to the limited numbers of port, each light sensor or ultrasonic sensor serves multiple purposes in different classes. However, if the same sensor object is defined more than once in different places then this causes an error in the system. To prevent this a sensor poller class is introduced to create instances for the sensor objects and pass the sensor data as parameters to the classes that need them. However, for some classes that needs changing the state of the sensor, for example, the capturing class needs to change the state on the light sensor to determine the when to check the flag as well as its color. In this case, the light sensor is initiated in the class. Also, there might be some sensors that are exclusive to some classes. Therefore, sensors are also allowed to be created within a particular class. The third layer includes the functional classes. They deliver the different functionalities that are required, and these classes are managed by robot control and sensor poller. They are the core of model in the software architecture.

6.0 SOFTWARE CONCURRENCY & PROCESSES



	0	1	2	3	4	5	6	7	8	9
Capturing	0	0	0	0	0	1	0	0	0	0
Odometry Display	1	1	1	1	1	1	1	1	1	0
Odometry Correction	0	1	1	0	1	0	1	0	1	0
Odometer	1	1	1	1	1	1	1	1	1	0
Navigation	0	1	1	0	1	0	1	0	1	0
Localization	1	0	0	1	0	0	0	1	0	0
Sensor Poller	1	1	1	1	1	1	1	1	1	0

Time t (period)	Action
0	Localize around the starting position and determine the team

1	Navigate to the waypoint
2	Traversal the zipline or navigate through the bridge based on the team
3	Re-localize on the other side
4	Navigate to the orange zone
5	Capture the flag
6	Navigate back to the other side via zipline or bridge based on the team
7	Re-localize on the other side
8	Navigate back to the starting point
9	System stops

Table 3: Actions over Time

7.0 FUNCTIONAL LOGICS

This section shows the logic and approach that are used for the system and listed by the order of classes.

7.1 Capturing

The key challenge of capturing a flag is how to determine the correct flag to capture and avoid the incorrect flags. The problem is solved by using a changing-mode light sensor. A light sensor must be used to determine the color of the flag however, the light intensity is not solely determined by the color of the object, it is also determined by the distance between the sensor and object. The approach is that the Red Mode in the sensor is used to determine where to detect the color as the robot is approaching a block and the RGB Mode is used to detect the color. A point of closeness is determined by a series of trials to decide when to change the sensor mode from detecting distance to detecting color. A detailed explanation regarding sensor characteristics can be found in the test documents. Once a block is determined to be incorrect, the robot will turn and use its extended arm to sweep away the block. If the block color is correct, the robot will beep for three times.

7.2 Light Localizer

Although this class is used during the lab period, the light localizer is still completely re-designed because the location of the light sensor no longer allows the implementation of the given algorithm in the localization lab. Two light sensors are placed next to the wheels in order to improve the performance of odometry correction. For the detail of this design, please refer to the hardware design document. Therefore, a new algorithm for light localization must be developed. The approach that is used is to utilize the time difference between two light sensors detecting a black line to determine the error of the robot. The localizer keeps tracks of the lines it detects to determine the x and y coordinates at the end of the localization. When there is a time gap between detecting the same black line on two sensors, the side that first detects the line will slow down and the other will speed up to make sure two sides get to the black line at the same time to adjust the heading. When both wheels are on the black line, the localizer will update the corresponding x or y coordinate. To finish the entire light localization process, the robot will perform this on both vertical and horizontal directions.

7.3 Navigation

The navigation is largely inherited from the lab period. The majority of the code can be implemented into the project. There are two features are added or changed. The first is to change the navigation course from polar to Cartesian movement. That means the movement is changed from the minimal distance to line following to accommodate the odometry correction algorithm. It is achieved by calculating the differences between the destination coordinates and current coordinates both horizontal and vertically, and roll the wheel in the vertical and horizontal direction respectively. It is also required to avoid any obstacle along the way. The coordinates of the obstacles are given, therefore, when there is an obstacle along the vertical direction, the robot will roll in the horizontal direction first to avoid the obstacle and vice versa. The avoidance algorithm is basically once an obstacle is found along one direction, the robot will roll in the other direction.

7.4 Odometry Correction

To simplify the threading management, the odometry correction only corrects the odometer reading. No physical correction of the robot is done in this class. Instead, the navigation will automatically correct the position of the robot over the course based on the odometer. The approach to determine x and y coordinate is very similar to the approach in the light localization. Additionally, the heading of the robot is being tracked to determine the direction of the line that is detected. Therefore, it is possible to determine the x and y coordinates based on the number of lines the sensor detected in each direction. With the help of two sensors, it improves the correction by better determining the current x and y values. The heading (theta) is corrected by this class as well by calculating the error between the right and right wheels. Then a trigonometry is used to determine the offset in theta as taught during the lectures. In the end, the theta is updated by the offset that is calculated.

7.5 Ultrasonic Localizer

The ultrasonic localizer is adopted from the lab period since the location of the ultrasonic sensor is unchanged. The same algorithm can be used. That means that placing the robot along the diagonal and using either falling edge or rising edge to determine the point at the diagonal in order to determine the heading or coordinate of the robot. To see the detail, please refer to the localization lab slide. For this project, the falling edge routine is selected because the falling edge generally performs better after a series of testing. The rationale behind this might be that the falling edge routine is scanning through the wall and detecting the local extrema. Detecting along the wall is uniform, so the reading is more accurate than detecting beyond the wall like the rising edge routine. For a detailed rationale, please refer to Localization Tutorial.

7.6 Odometer

The odometer is a core data class which means that this class is there to provide necessary odometry data to support all the classes above. The code base and most of the methods in the class are provided by McGill ECSE 211 TA team. The complete class is written by Lab Team 1 with the logic provided in ECSE 211 during the Lab 3 period. This class determines the position of the robot by updating its x-coordinate, y-coordinate and heading, denoted as x, y and theta throughout the program. All of these are calculated and updated by adding the change to the previous data. The total displacement is calculated by the average of the displacement on the right motor and left motor. Then the changes on x and y directions are calculated with the total displacement multiplied by the trigonometry with respect to the horizontal and vertical direction. The change in heading is calculated by the difference between the left and the right displacement divided by the wheelbase which is the distance between the centers of the right and left wheel. For a detailed rationale behind this, please refer to Lejos documentation at Odometry-lejos-EV3-Jan-27-2016.

Only functional classes are listed for approach analysis and explanation. The supporting classes including the main class named *robotControl* are just there to arrange the software processes and concurrency as well as passing data which is shown in Section 6 of this document. There are 2 other methods named zipline and bridge in *robotControl* besides main. The zipline method uses navigation and light localization to perform the zipline traversal process. The process and logic of this can be found in Section 6 of this document or in the source code. The bridge method uses odometer and navigation to cross the virtue bridge. The logic of this is to navigate to the starting point of the bridge first. Then continue to the turning points of the bridge named SV_UR in the example diagram in Design Description. From the turning point, the robot navigates to the end of the bridge and performs a light localization. Also, a sensor logger is to create a separated text file in the project directory and record the data from the sensors aboard for debugging and analytical purposes. To learn more about supporting classes, please refer to their commenting in the source code or the JavaDoc.

8.0 CLASS DESIGN RATIONALE

The core principle of our design is reusability because the reusable parts from the previous lab period have been tested for multiple times, hence these parts could offer great reliability to the system. Therefore, code base of the system is modelled from the class structure from the previous labs. Furthermore, the content of some classes is reused from the lab period, such as odometer, ultrasonic localizer and some methods in navigation. By implementing these reusable codes, a lot of resources is saved as well as improved the stability of the system. However, it is inevitable that there must be some changes to the existing system to accommodate the new features. For example, odometry correction is required over a long distance. The correction algorithm is based on the detection of black lines. If the robot is moving with the minimal distance between points, known as the polar movement, it is impossible for the correction program to properly function due to the failure to detect lines. Hence, the polar coordinate movement in the navigation class is switched to the Cartesian coordinate movement which is basically a line following algorithm.

In conclusion, our software design focuses on reusing the parts as much as possible from the previous lab to ensure the reliability and stability of the system and but actively updates the program to adapt to new features. The general design rationale is to simply add the new feature on top of the existing parts instead of creating an entirely new bespoke system for the new features and requirements.

9.0 SOFTWARE STATUS

Class	Workload (line)	Completion
robotControl	348	100%
Navigation	387	100%
Ligth Localization	451	100%
Ultrasonic Location	297	100%
OdometryCorrection	228	100%
SensorPoller	117	100%
Capturing	251	100%
Odometer	314	100%
sensorLogger	50	100%
Total	2443	100%

Table 4: Software progress and workload.