SOFTWARE DESIGN DOCUMENT

Project: ECSE 211 Final Design Project – Team 6
Document Version Number: 3.2
Date: 22/10/2017
Original Author: Justin Tremblay
Edit History:
[18/10/2017] Justin Tremblay: First version of the document, added state machine and class diagrams.
[22/10/2017] Alex Hale: moved to Google Docs for easier collaboration; transferred existing info and diagrams
[28/10/2017] Justin Tremblay: Updated document with new diagrams, designs and systems.
[29/10/2017] Justin Tremblay: Continued working on design updates.
[30/10/2017] Alex Hale: formatting and grammar touch-ups for Week 3 submission

1.0 - TABLE OF CONTENTS

2.0 - State Machine

The robot is controlled by a state machine that handles decision-making depending on the robot's inputs and its current goal. It is inspired from the GOAP (Goal Oriented Action Planning) system which is used in the video game industry to control the artificial intelligence of non-playable characters.

More info on GOAP: http://alumni.media.mit.edu/~jorkin/goap.html

Figure 1 depicts a diagram of the robot's state machine.

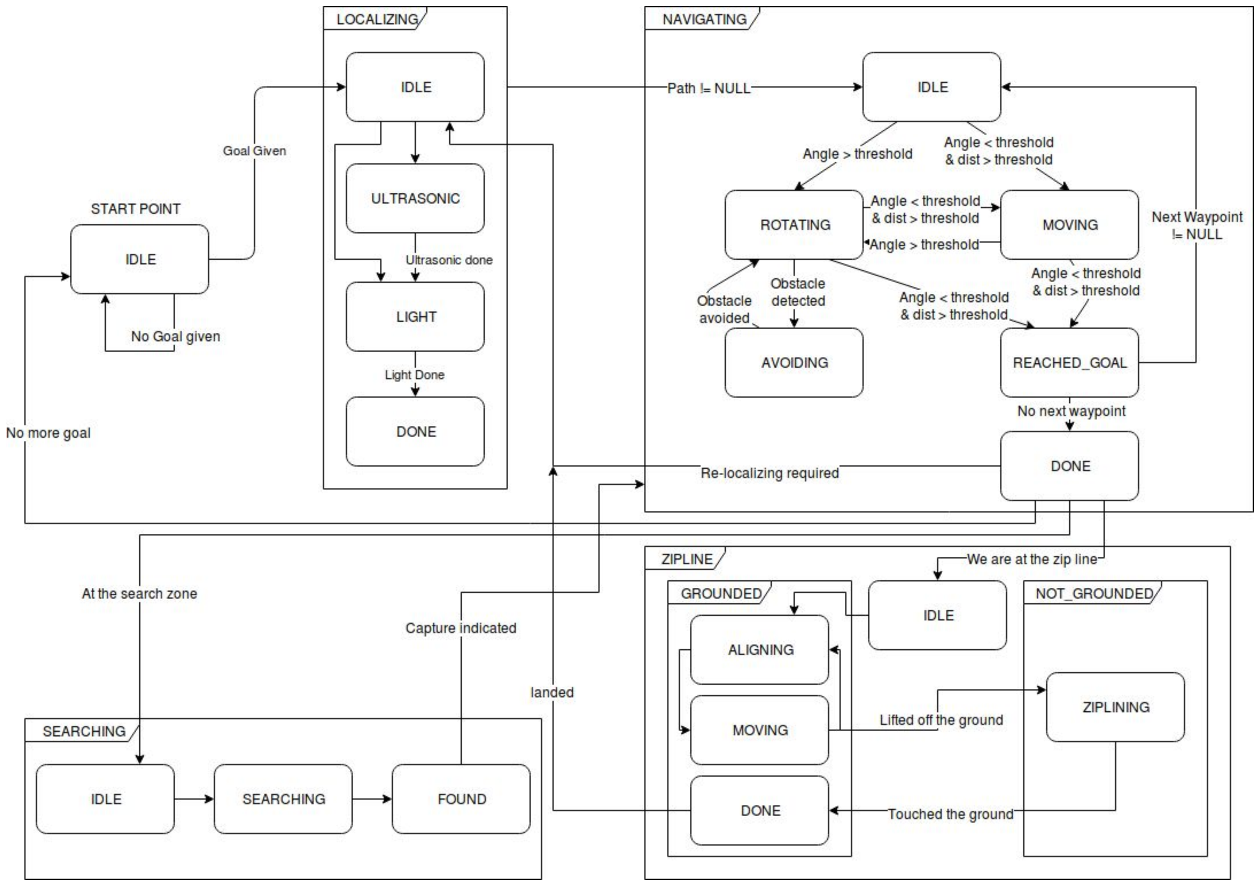
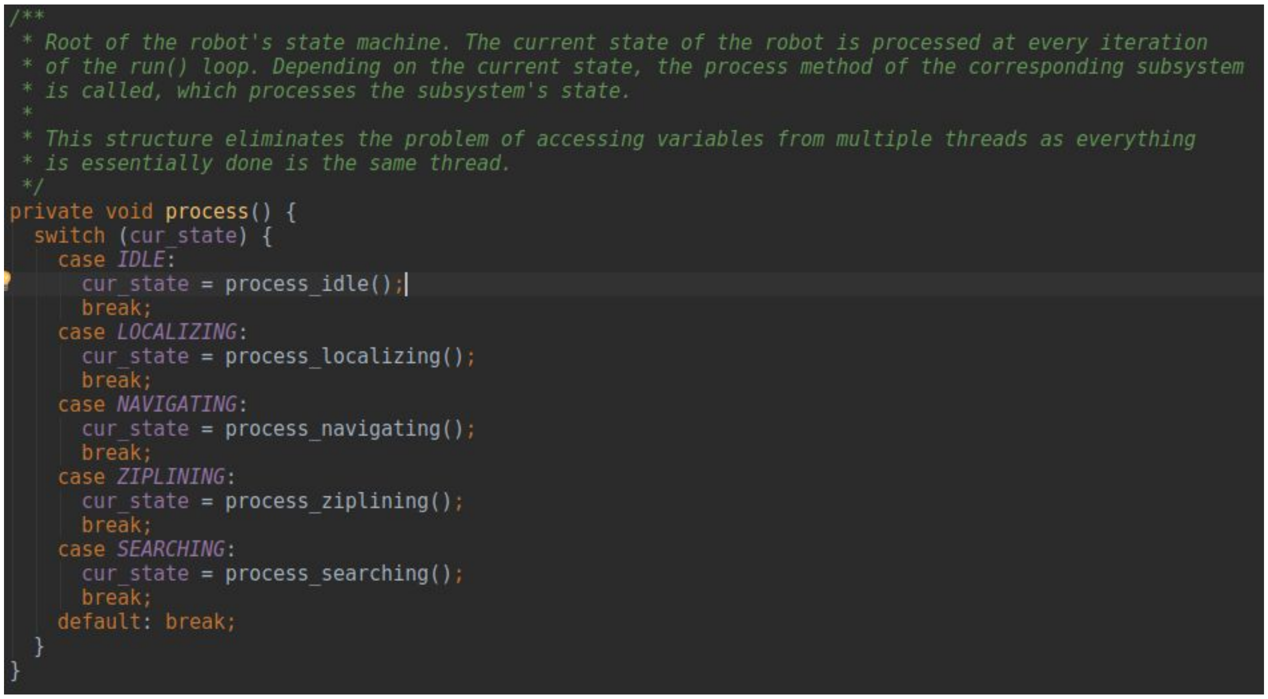


Figure 1 - Robot State Machine

Example of state machine implementation:



```
/**
 * Root of the robot's state machine. The current state of the robot is processed at every iteration
 * of the run() loop. Depending on the current state, the process method of the corresponding subsystem
 * is called, which processes the subsystem's state.
 *
 * This structure eliminates the problem of accessing variables from multiple threads as everything
 * is essentially done is the same thread.
 */
private void process() {
  switch (cur_state) {
    case IDLE:
      cur_state = process_idle();
      break;
    case LOCALIZING:
      cur_state = process_localizing();
      break;
    case NAVIGATING:
      cur_state = process_navigating();
      break;
    case ZIPLINING:
      cur_state = process_ziplining();
      break;
    case SEARCHING:
      cur_state = process_searching();
      break;
    default: break;
  }
}
```

Figure 2 - Example instance of a state machine

This pattern is used in each of the systems. At every iteration, it runs the `process_state()` method corresponding to the current state of the system. This method handles the required actions by calling the corresponding subsystem's `process()` method and returns the next state depending on various parameters and conditions.

3.0 - Class Diagram

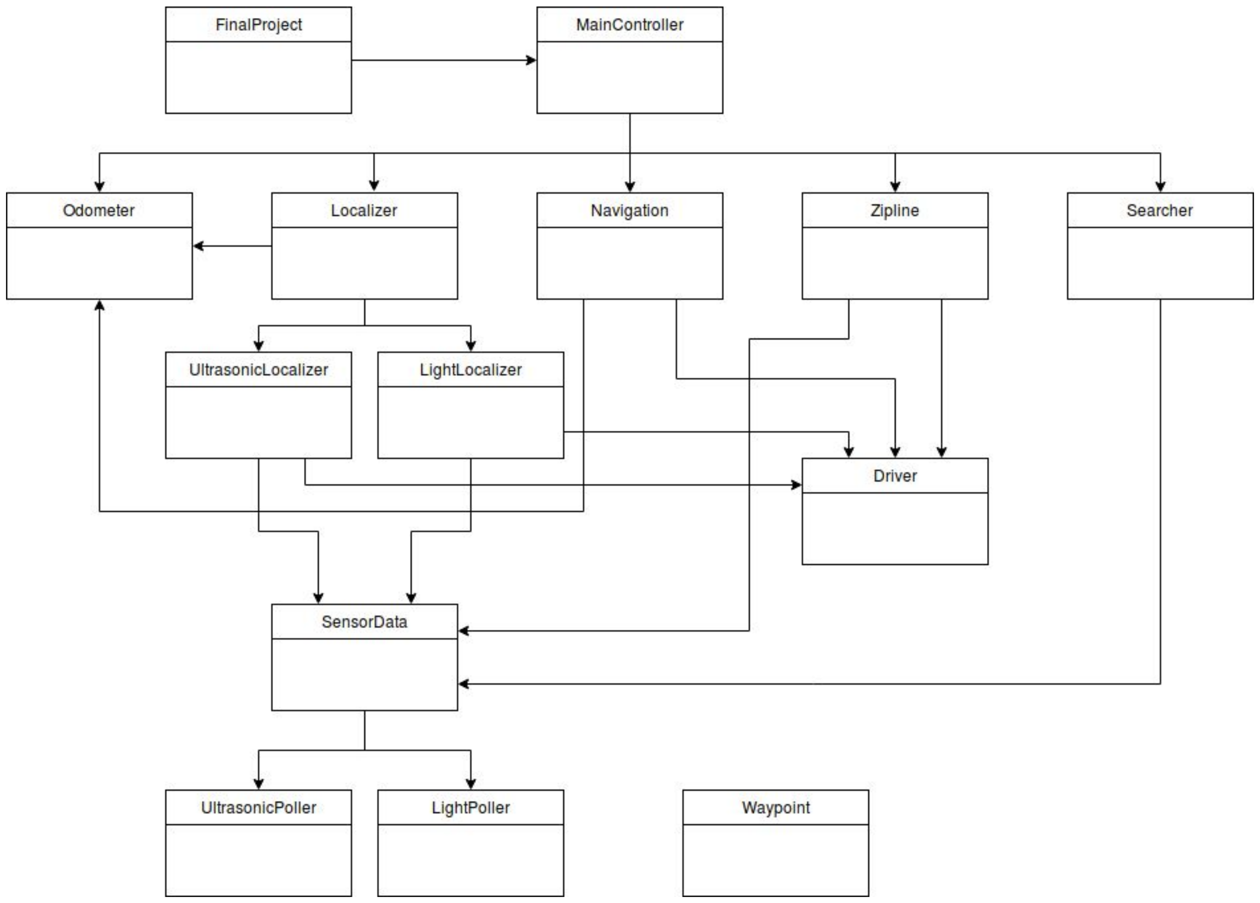


Figure 3 - Class Hierarchy

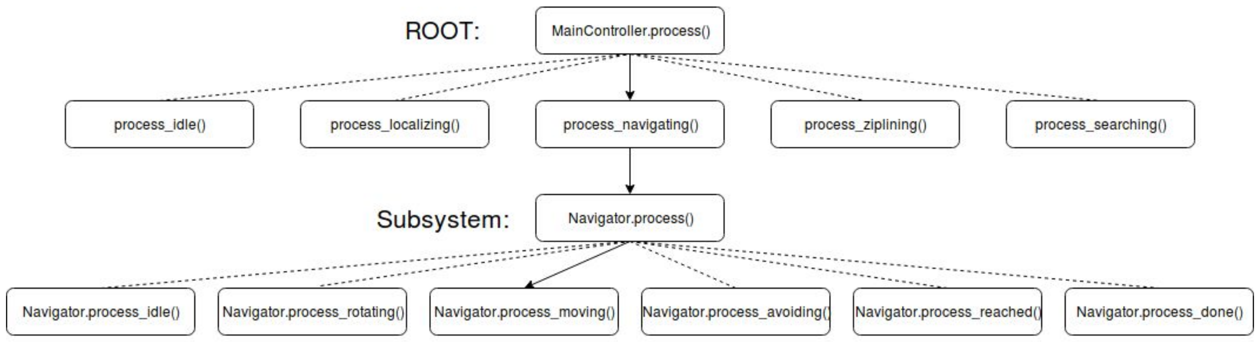
Note: the Waypoint class doesn't have its own section since its only function is to provide an abstraction for the grid coordinates.

3.1 - FinalProject

This is the main class of the program. It contains the main method of the program, and it groups all the constants in one place for easy tuning. It also handles initializing the motors, sensors and subsystems, as well as starting all the necessary threads.

3.2 - MainController

This is the main control thread and the root of the state machine. It keeps track of the robot's current goal and delegates control to the appropriate subsystem depending on the current state. For example, if the robot has to move:



When the `process()` method from the controller is called, the `process_navigating()` method is called This method checks for conditions before and after calling the navigator's own `process()` method, which will branch to `process_moving()`. This will then check for other conditions and tell the driver class to move forward, stop, and return the next state.

When the MainController's `start()` method is called, it immediately attempts to establish a connection with the game server and waits for input before moving on to the execution of its tasks.

3.3 - Odometer

Keeps track of the robot's position and orientation on the playing field. It uses the motors' tachometers and simple mathematics to determine displacement. It continuously runs in its own thread.

3.4 - Localizer, UltrasonicLocalizer, LightLocalizer

Handles determining the orientation and position of the robot at the start of the game and when the robot comes off the zipline. It is divided in two parts:

- Ultrasonic Localizer: uses the ultrasonic sensor and two walls to determine the orientation of the robot, the positive x-axis being 0 degrees.
- Light Localizer: uses two color sensors and the newly found orientation angle to determine its position with respect to a reference position.

When localizing, the main controller first determines if it has to skip ultrasonic localization (this amounts to determining whether the robot is in a corner). If it is not skipping the ultrasonic, it calls the UltrasonicLocalizer's `localize()` method, which will do either rising or falling edge localization to determine its orientation. It then calls the LightLocalizer's `localize()` method, which will perform localization using two colour sensors. The process ends with the robot aligning itself to the grid lines to determine its position relative to the reference position that is was given.

3.5 - Navigation

Handles navigating the robot through a set of waypoints on the grid, as well as avoiding obstacles when they are encountered.

It also works in tandem with the searcher in order to navigate through the searching zone while looking for the flag.

3.6 - Zipline

Handles moving the robot on the zipline. It uses one of the colour sensors to determine whether or not the robot is touching the ground or not. When the robot lands, it moves forward and re-localizes.

3.7 - Searcher

Works alongside the navigator to look for the "flag" and identify it. Indicates capture upon successfully identifying the "flag".

3.8 - Driver

Handles moving the robot. It is used in every system that requires the robot to move somewhere.

3.9 - UltrasonicPoller and LightPoller

Used by most systems for various purposes. They handle collecting samples and sending them to the SensorData class for processing and easier accessing.

3.10 - SensorData

This class gets the data from the sensors and processes it. It also facilitates access to the data for other subsystems.