ECSE 211 - Design Principles and Methods

Software Design Document

GROUP 11
John, Durham, Alex, Ian, Ethan

Version 3.00
28th March 2017

Abstract

This document will provide a detailed overview of the software requirements and design specifications that have been used in our robot. Functions are covered with provisions to different views of the system and to design decisions made within them.

Contents

1 EDIT HISTORY **3**

2 INTRODUCTION **4**
- 2.1 Purpose . . . 4
- 2.2 Context . . . 4
- 2.3 Audience . . . 4
- 2.4 Scope . . . 4

3 FUNCTIONAL REQUIREMENTS **5**
- 3.1 Overview . . . 5
- 3.2 WiFi . . . 6
- 3.3 Odometry . . . 7
- 3.4 Odometry Correction . . . 8
- 3.5 Navigation . . . 8
- 3.6 Obstacle Avoidance . . . 8
- 3.7 Localization . . . 12
- 3.8 Ball Retrieval . . . 12
- 3.9 Field Mapping . . . 12
- 3.10 Scoring . . . 16
- 3.11 Offense . . . 16
- 3.12 Defense . . . 16

4 NON-FUNCTIONAL REQUIREMENTS **19**
- 4.1 Battery Efficiency . . . 19
- 4.2 Threading . . . 19

5 SYSTEM ARCHITECTURE **20**
- 5.1 Overview . . . 20
- 5.2 Structure . . . 21

6 AUTOMATION **22**
- 6.1 Unit Testing . . . 22
- 6.2 Continuous Integration Build . . . 22

7 GLOSSARY OF TERMS **23**

1 EDIT HISTORY

- March 4th 2017 (Version 1.0.0):
  - **John Wu:** Initial set up of document sections (including table of contents)
- March 6th 2017 (Version 1.0.1):
  - **John Wu:** Created introduction, added case diagram, added class diagram
- March 16th 2017 (Version 1.1.0):
  - **John Wu:** Updated automation section to include unit tests and CI Build
- March 23rd 2017 (Version 1.2.0):
  - **John Wu:** Created structure section in software architecture
- March 26th 2017 (Version 2.0.0):
  - **John Wu:** Wrote descriptions for non-functional requirements
- March 27th 2017 (Version 2.1.0):
  - **John Wu:** Created sequence diagrams for non-functional requirements
- March 28th 2017 (Version 3.0.0):
  - **John Wu:** Added functional requirements section

2 INTRODUCTION

2.1 Purpose

Our vehicle is a fully autonomous robot that is capable of localization, navigation, obstacle avoidance, odometry, and retrieving a ball that can be shot into a target. In addition, the robot is capable of defending all the listed capabilities. The purpose of this document is to provide a detailed overview about the software implementation used in the robot. The functionality will be covered with provisions from different views including class diagrams, sequence diagrams, and overall system architecture.

2.2 Context

See requirements document included in the reference package.

2.3 Audience

This technical document is intended for parties who are interested in maintaining, testing, or extending the software that is included with the project. It is assumed that all readers have a sufficient understanding of programming, software design, and the LeJOS API.

2.4 Scope

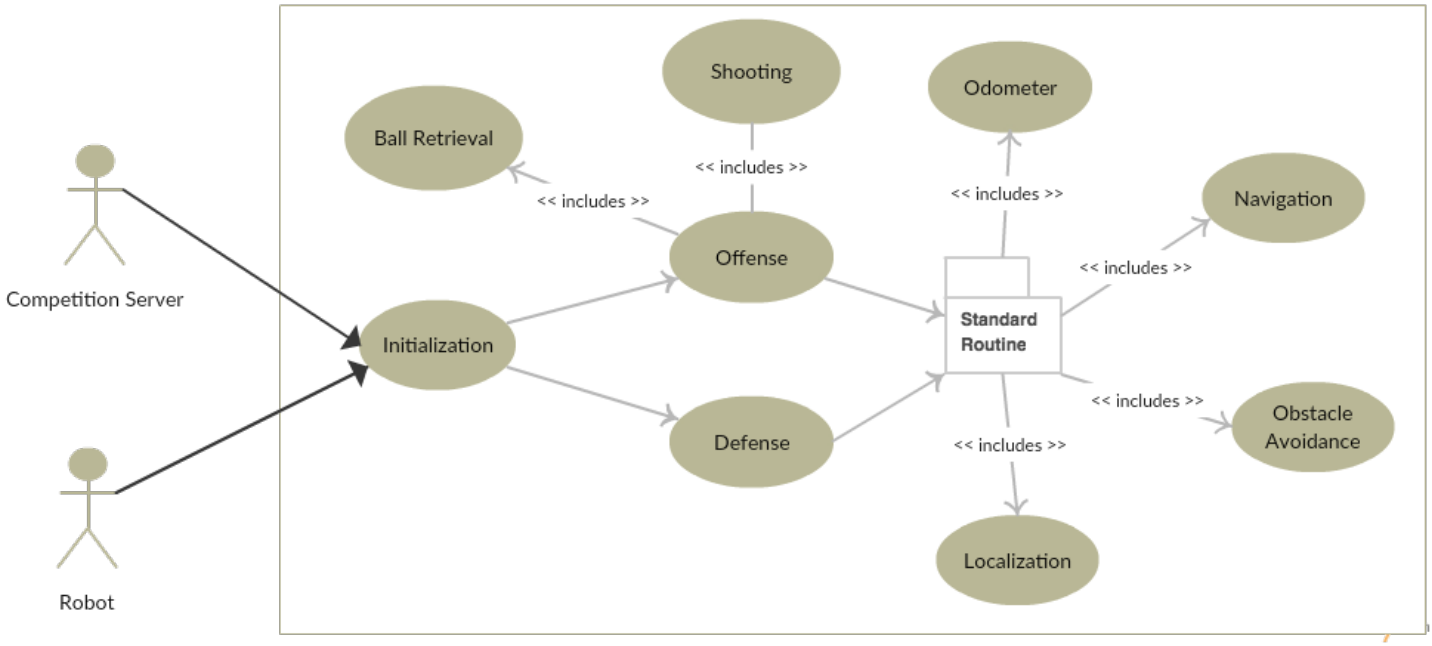
The scope of our project will be limited to only achieving the list of requirements specified by the client. Therefore, no additional features will be added.

3 FUNCTIONAL REQUIREMENTS

3.1 Overview

Shown below is a use case diagram of our system. In order to initialize, the robot must be connected to retrieve parameters from the competition server. Based on the parameters given, the robot will either play offense or defense. Regardless of its position, the robot will go through it standard routines which includes localization, odometer, navigation, and obstacle avoidance. On offense, the robot will have additional cases for retrieving the ball and shooting it. On defense, the robot will rely on its standard routines to properly defend.

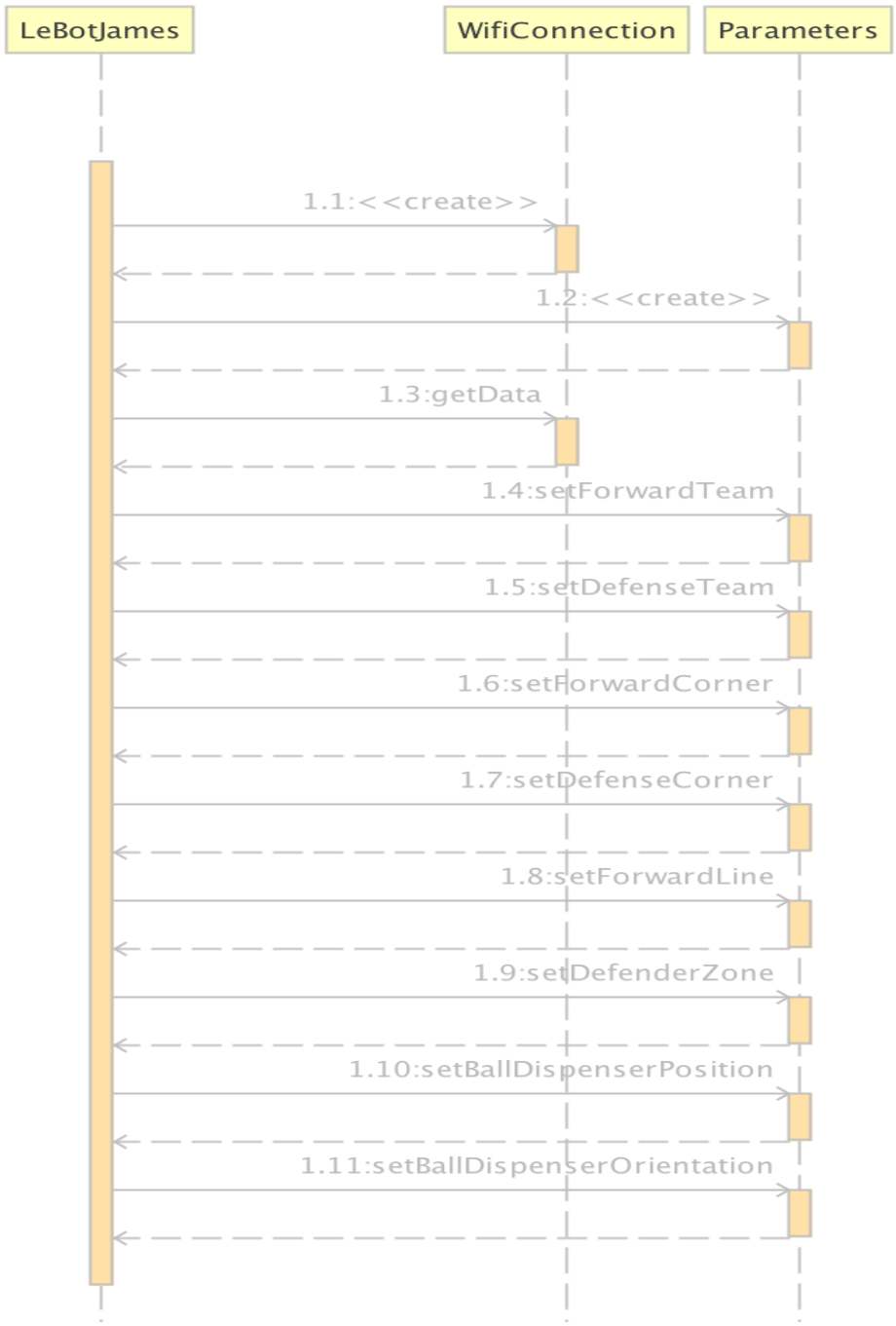
Figure 1: Use Case Diagram



3.2 WiFi

Upon starting the software, the robot will wait for parameters to be received from the server through the WiFi before beginning the proper functions. Before uploading the software, the IP Address of the network must be specified through the WifiProperties.java class. Upon retrieving parameters, LeBot will then localize and proceed to play either defense or offense.

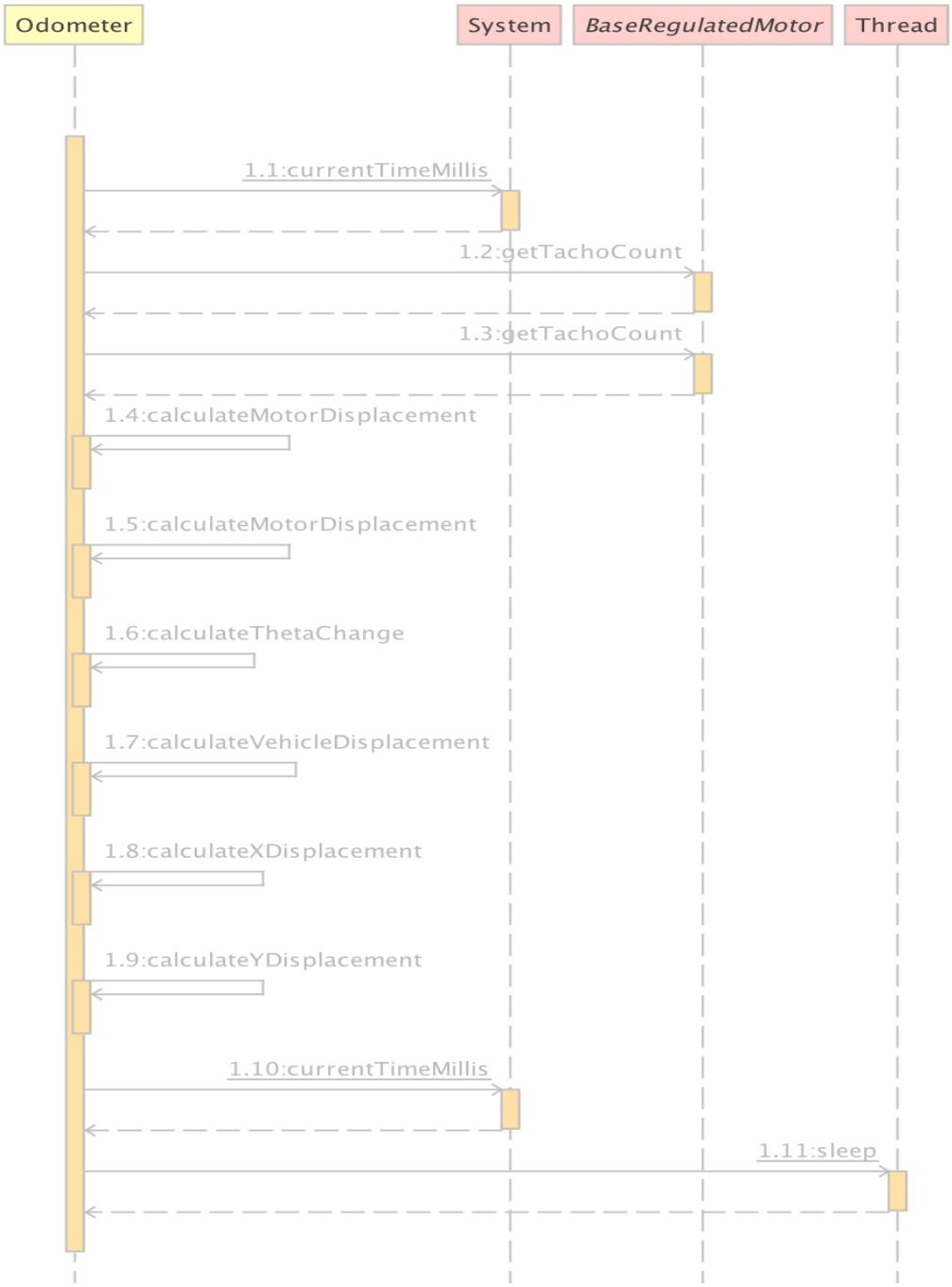
Figure 2: WiFi Sequence Diagram



3.3 Odometry

The Odometer controller class is a continuous thread that processes the odometry of LeBot. It relies on the tacho counts from the motors of the EV3 hardware to calculate its position at all times. Currently the odometer updates its values every 10 milliseconds which can be modified through the TimeConstants.java resource class.

Figure 3: Odometry Sequence Diagram



3.4 Odometry Correction

The odometer relies on theoretic calculations to determine its position, but there are many factors that can vary the actual result of the position of the robot. This includes slip in the vehicle, imbalances in the center of gravity causing it drift in a certain direction, and decreases in performance due to low battery.

In order to accommodate for these errors that accumulate over time, the robot uses its surroundings to implement odometry correction. Currently the robot only navigates in movements that are perpendicular to the lines (see section 3.5) and corrects its position every time it passes it. It uses two light sensors on each side of the robot (see hardware document) that aligns itself if one sensor reaches the line before the other. Upon passing each line, the robot references the Field Mapper utility class (see section 3.10), to correct to its actual position on the field.

3.5 Navigation

In order to have more precise odometry and utilize odometry correction, LeBot James navigates only in movements perpendicular to the grid lines. When traveling to a specific point on the field, the navigation is broken up in components by only moving one square at a time. At each square, the robot uses a greedy algorithm along with a priority queue and the field mapping to determine its next move, which considers the following parameters:

- The distance to the final destination
- Previous moves in order to avoid cycles
- Optimization for minimal turning angle

Upon determining the possible moves, the robot will try and make each one in order of its priority. If the robot been in the square previously, then it will immediately drive to the square. If not, then it will begin by checking the mapping to ensure there was no existing obstacle or boundaries. It will scan the square for any obstacles (see section 3.6) if that data as not been determined. If any obstacles or boundaries are found, then the robot will try the next highest priority. This approach for navigation is called recursively until the final destination has been reached.

3.6 Obstacle Avoidance

In order to avoid obstacles, the robot uses the ultrasonic sensors placed on the front of the vehicle to determine if there are obstacles in the square it is trying to approach. To check for obstacles, it simply does a full radius scan of the square and either marks the whole square as an obstacle or not an obstacle in the field mapping. After the the first initial scan of a square, the result is mapped and the square is never needed to be scanned again. Over time, the robot will eventually learn the field and ultimately not need to scan any more squares, increasing the speed of our navigation. The obstacle mapper class also helps the robot learn the field faster (see section 3.7).

Figure 4: Odometer Correction Sequence Diagram

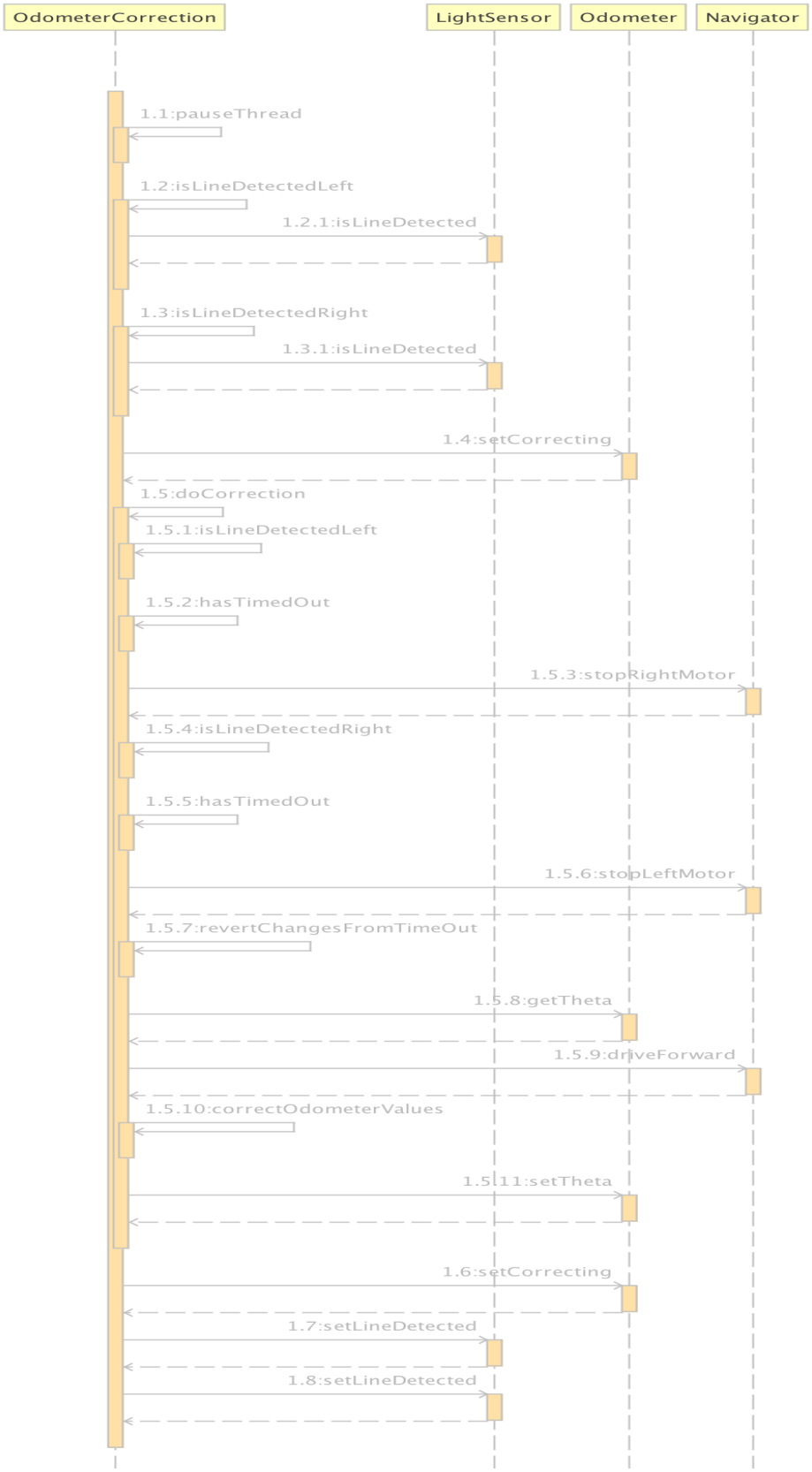


Figure 5: Navigation Sequence Diagram

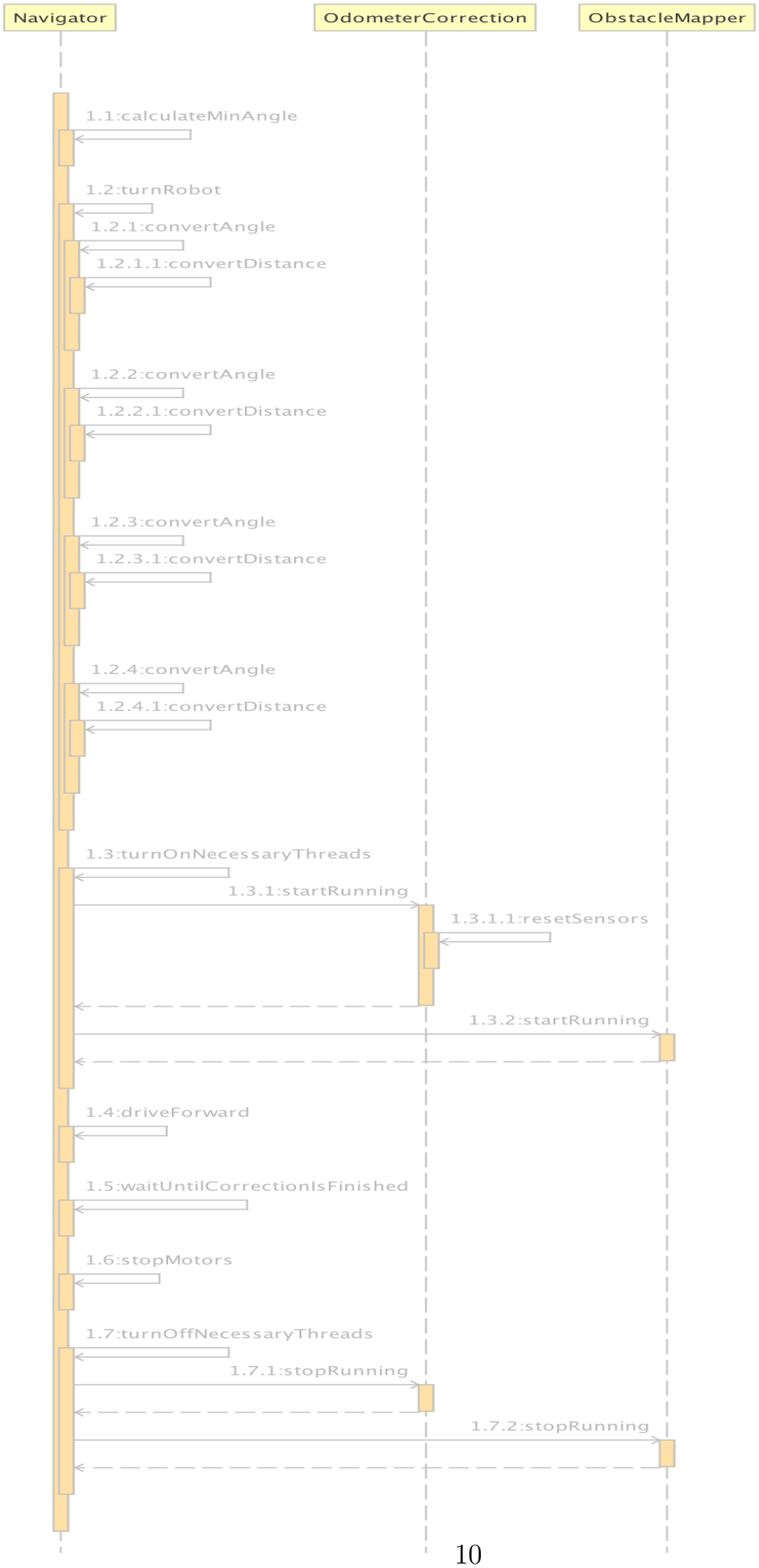
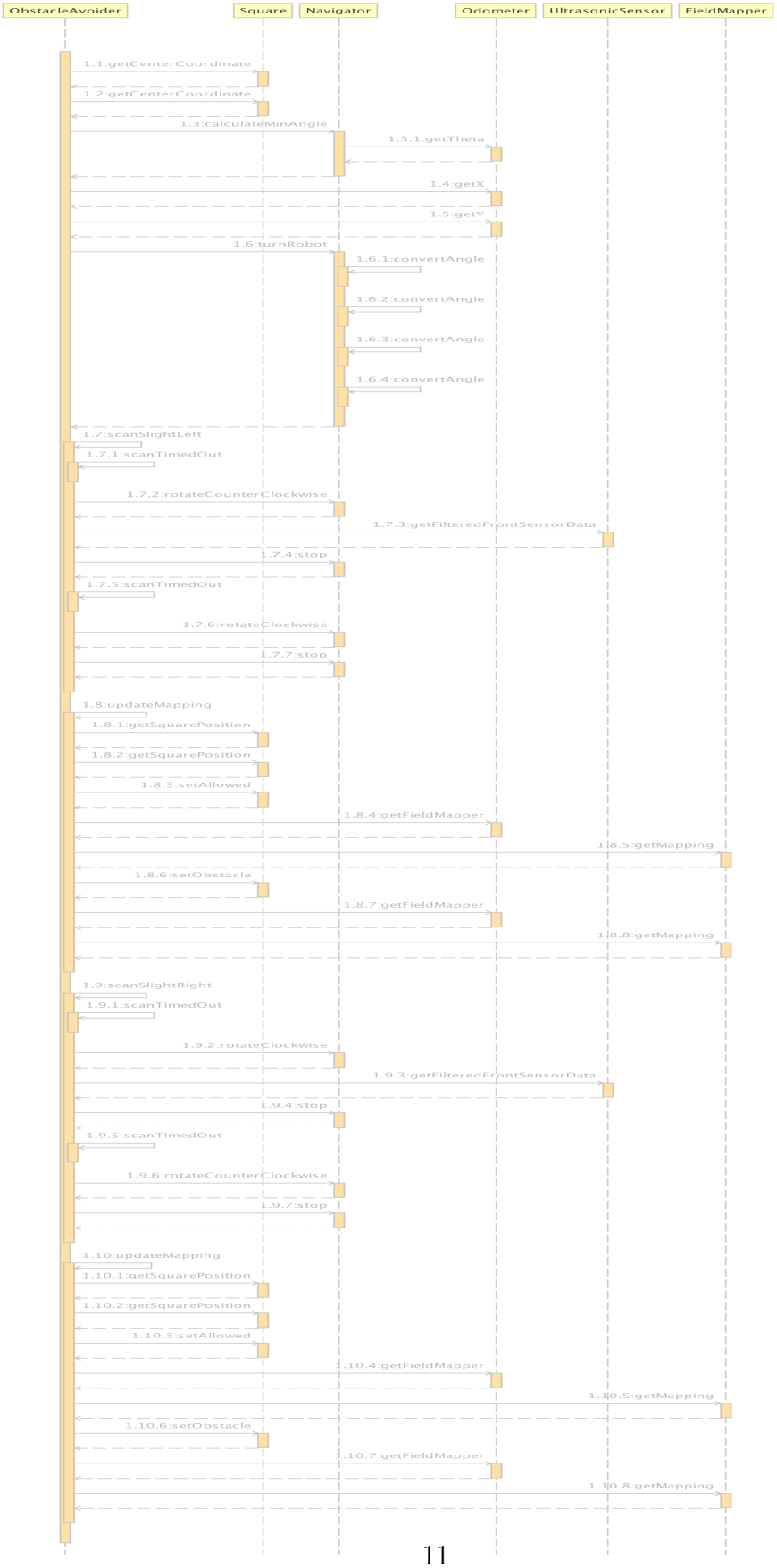


Figure 6: Obstacle Avoiding Sequence Diagram



3.7 Localization

Localization of the robot uses a falling edge technique to detect the the wall directly to the left of the robot. Once the wall is found, it then uses the front ultrasonic sensor to poll and store data while continuing to rotate counterclockwise until the wall has no longer been detected. Upon finishing rotation, it uses the data to compute the derivatives in order to determine the two minimums. Using these values, the starting position of the LeBot is determined and used to center itself on the starting square.

3.8 Ball Retrieval

In order to retrieve the ball, LeBot begins by traveling to the closest square to the ball dispenser approach which is determined by referencing the field mapper (see section 3.10). Upon approaching the square, all odometry correction is turned off and the robot carefully drives to the line in which the ball dispenser straddles. After aligning itself, the robot then carefully rotates the launcher into the position to retrieve the ball and moves back to the square it was previously at. From there it will continue its navigation to the shooting positions.

3.9 Field Mapping

As a utility for more accurate and faster navigation, the software includes a field mapper class that stores data about its surroundings and allowing it to map and ultimately learn the field over time. Currently, the field mapper stores the following data about each square:

- Position in the overall field map
- Coordinates of the north, south, east, and west lines
- Coordinates of the center of the square
- Whether the square is allowed or not based on boundaries
- Whether the square has an obstacle
- Whether the square is a shooting position

Figure 7: Localization Sequence Diagram

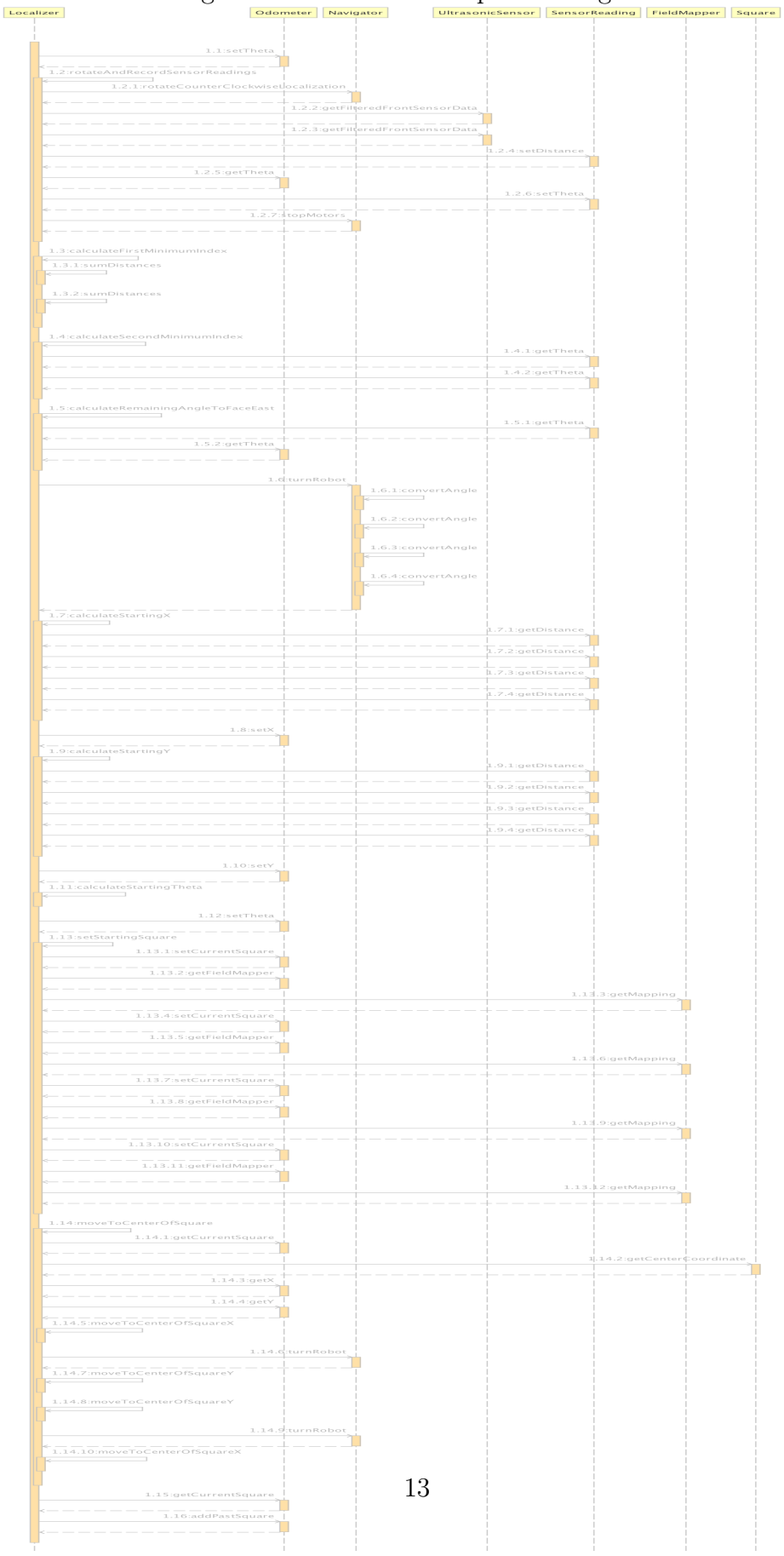


Figure 8: Ball Retrieval Sequence Diagram

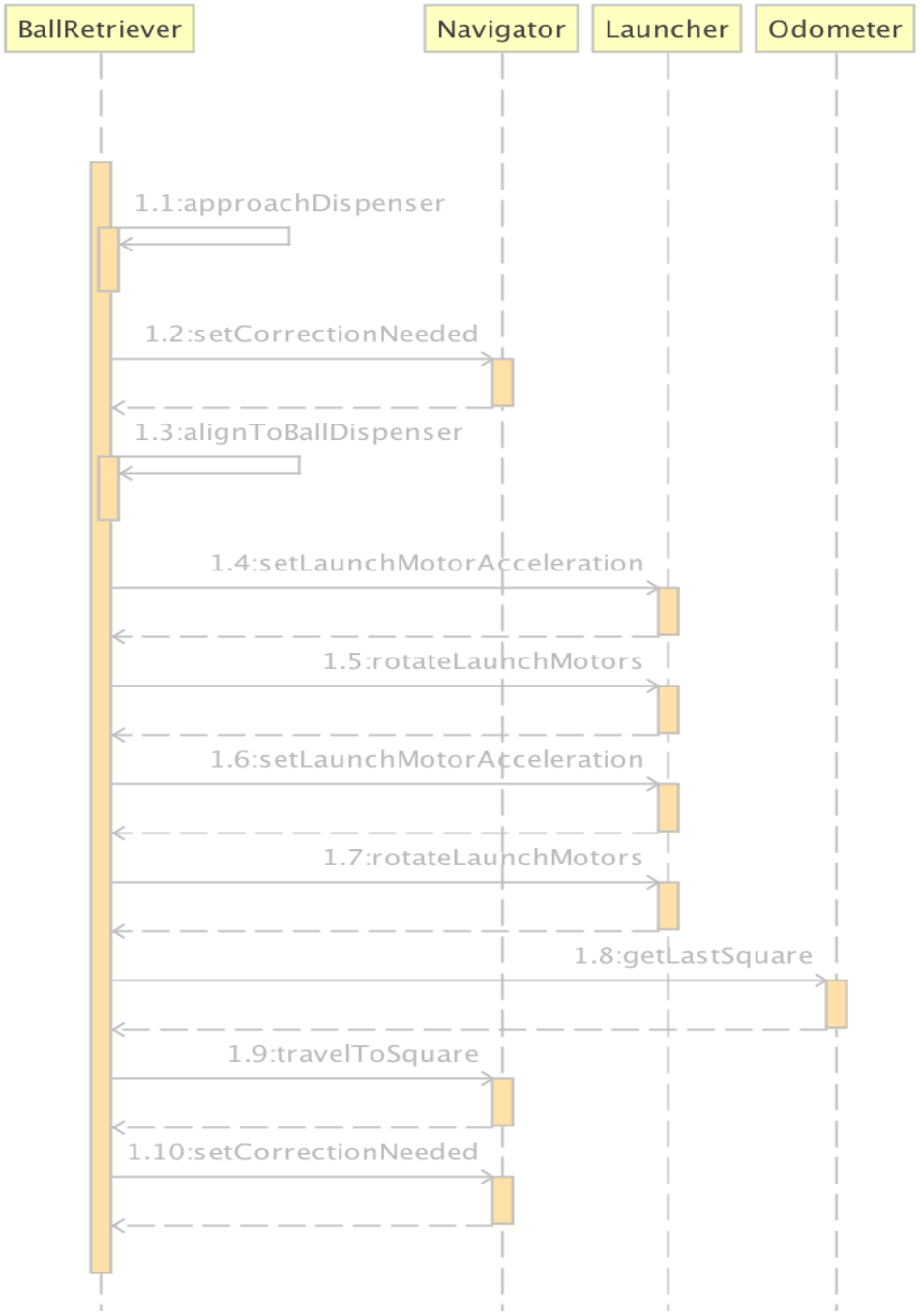


Figure 9: Field Mapping Sequence Diagram

3.10 Scoring

In order to score, the robot references the field map to determine what the ideal position to shoot from is (see section 3.10). Each position has a different calibrated release angle and launcher rotation speed. Upon reaching one of the shooting positions, the robot simply aligns to the goal, sets its shooting variables to the calibrated values, and shoots the ball.

3.11 Offense

The offense essentially combines all of the above functionalities together to recursively retrieving the ball from the dispenser navigating to the optimal position to shoot.

3.12 Defense

The defensive strategy of LeBot James simply involves traveling into the defender zone and recursively moving back and forth between the squares in front of the goal.

Figure 10: Shooting Sequence Diagram

Figure 11: Offense Sequence Diagram

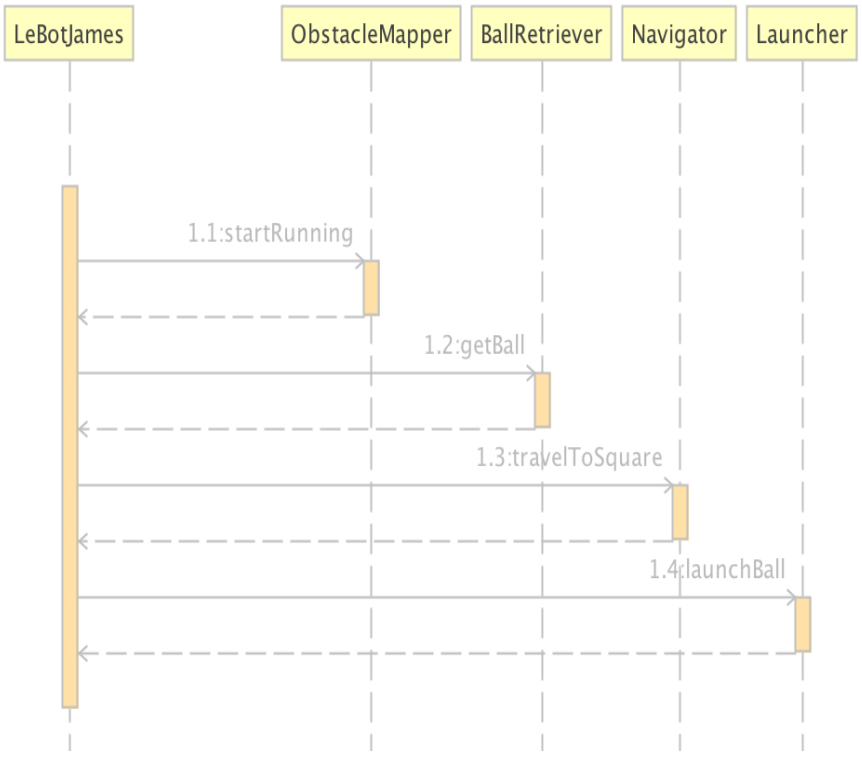
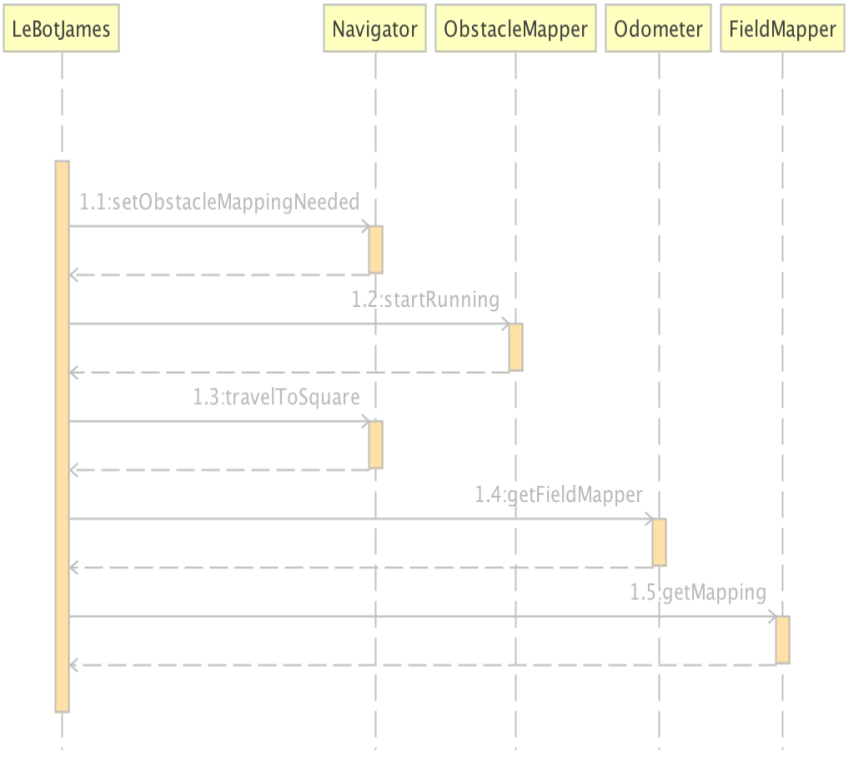


Figure 12: Defense Sequence Diagram



4 NON-FUNCTIONAL REQUIREMENTS

4.1 Battery Efficiency

It is important to note that the performance of the robot relies highly on battery efficiency. This means that over time the performance of the robot may decrease as the power in the battery drops. Currently the design accounts for any of these deficiencies by catching any errors and executing backup plans, which include the following:

- In the the odometry correction, if a light sensor fails to detect a line, then the robot is position is reverted back to its original point before correction begins.
- In localization, if there are not enough (or insufficient) data to calculate the starting positions it will recursively execute the same function.

Should there be any future updates to the software, it is important to continue thinking about scenerios where battery life can play a factor and account for this in the design.

4.2 Threading

A unique feature to using the LeJOS API is that it allows for multi-threading through Java. Although advantageous, this feature can also cause a lot of trouble throughout the software if not used properly. Odometer, light sensor, ultrasonic sensor, and emergency stopper threads currently run continuously throughout the execution of the program. Odometer correction and field mapper classes are safely turned on and off throughout the program through methods that have limited access by other classes.

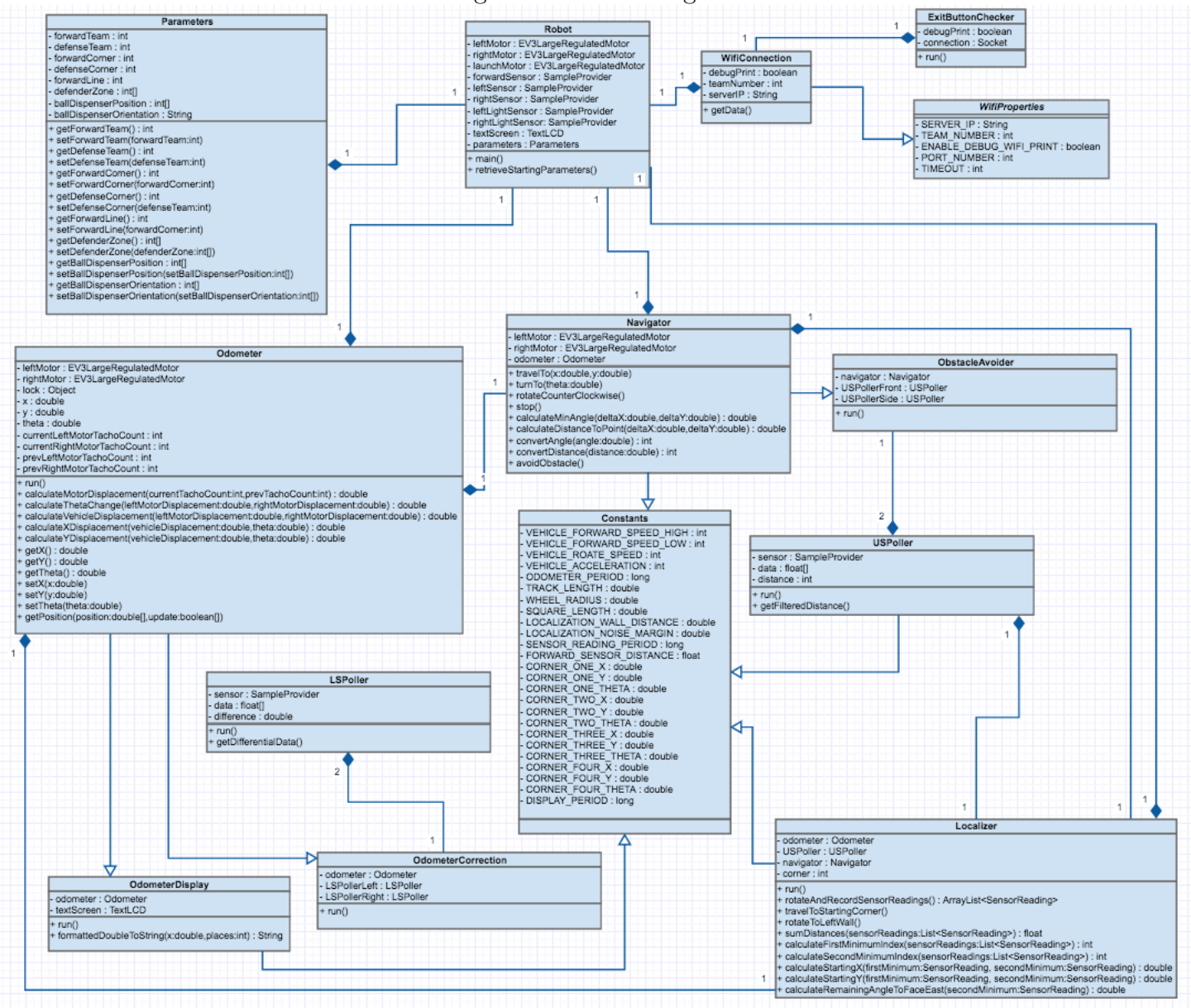
Future improvements to the software must take great caution with managing threads and ensuring safe usage at all times.

5 SYSTEM ARCHITECTURE

5.1 Overview

The following class diagram is a high level overview of the software architecture built into the system.

Figure 13: Class Diagram



5.2 Structure

As shown in the above diagram, the software of the robot can seem quite complex. However, each class in the system can be split into one of the five structural software packages. These structural components include controller, object, resource, utility, and wifi packages.

- **Controller Package:** The role of the controller classes are to simply control the robot and perform all actions in the standard routine package. This includes localizing, navigation, odometer, shooting, correction, obstacle avoidance, etc.
- **Object Package:** The classes in the object package represent components of either the field or robot. In most cases these objects are custom software layers built on top of the EV3 API and used to access output values in the hardware. In other cases, these are simply plain old java object (POJO) templates.
- **Resource Package:** Classes in the resource package store all the static constants throughout the software. It provides a central referencing system for easy refactoring and keeps all variables consistent throughout the code.
- **Utility Package:** The utility package holds various helper classes that can be used by the robot in various miscellaneous situations.
- **Wifi Package:** The wifi package includes classes that are used to connect to the wifi server in order to receive commands from a central server.

6 AUTOMATION

6.1 Unit Testing

In order to ensure the code works as expected, the software is bundled with a set of unit tests. These unit tests are written with the JUnit framework and covers all the logic involved in the functioning of the robot. These tests should be ran each time a change is made to make sure no prior code has been broken in the update process.

6.2 Continuous Integration Build

The software also includes a continuous integration build file which uses the apache ant command line tool. Upon new changes to the software, the following ant commands can be ran in the terminal:

- **ant clean** - cleans the current build directory, deleting any cached files
- **ant compile** - compiles all the code and puts them into a build directory (has a dependency on ant clean)
- **ant test-compile** - compiles all the testing classes and puts them into a build directory (has a dependency on ant compile)
- **ant test** - runs all the unit tests that are packaged in the software (has a dependency on ant test-compile)

The CI build is also integrated into the remote Git Hub repository through the Travis CI plug-in. Upon pushing any code to the remote repository, all the build commands will be ran to ensure nothing has been broken and printing out a stack trace if something went wrong. This is allows for easy collaboration and quick development for any future updates to the software.

7 GLOSSARY OF TERMS

- Continuous Integration - a process used in agile development which automates tests and allows for quick implementation from multiple developers.
- Field Mapping - the process of recording details about the surroundings of the robot
- Unit Testing - the process of testing the logic of individual methods in the software
- User Case Diagram - a diagram used to describe a set of high level actions a system can perform in collaboration from external users.
- Sequence Diagram - a diagram used to describe how objects operate with one another.
- Class Diagram - a diagram that shows the attributes, operations, and relationships that one class has with another.
- LeJOS API - the application programming interface layer that is used to control the physical hardware of the robot.