

Software Design Document

GROUP 11

John, Durham, Alex, Ian, Ethan

Version 4.00

30th 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	4
2	INTRODUCTION	5
2.1	Purpose	5
2.2	Context	5
2.3	Audience	5
2.4	Scope	5
3	FUNCTIONAL REQUIREMENTS	6
3.1	Overview	6
3.2	WiFi	7
3.3	Odometry	8
3.4	Odometry Correction	9
3.5	Navigation	9
3.6	Obstacle Avoidance	9
3.7	Obstacle Mapping	13
3.8	Localization	13
3.9	Ball Retrieval	13
3.10	Field Mapping	13
3.11	Scoring	18
3.12	Offense	18
3.13	Defense	18
4	NON-FUNCTIONAL REQUIREMENTS	21
4.1	Battery Efficiency	21
4.2	Threading	21
5	SYSTEM ARCHITECTURE	22
5.1	Overview	22
5.2	Structure	23
5.3	Classes	23
5.3.1	LeBotJames.java	23
5.3.2	Odometer.java	24
5.3.3	OdometerCorrection.java	26
5.3.4	Navigator.java	26
5.3.5	ObstacleAvoider.java	28
5.3.6	ObstacleMapper.java	28
5.3.7	Localizer.java	29
5.3.8	Launcher.java	29
5.3.9	BallRetriever.java	30
5.3.10	LightSensor.java	31
5.3.11	Square.java	31
5.3.12	UltrasonicSensor.java	32

5.3.13	FieldConstants.java	33
5.3.14	NavigationConstants.java	34
5.3.15	RobotConstants.java	34
5.3.16	ShootingConstants.java	34
5.3.17	ThresholdConstants.java	35
5.3.18	TimeConstants.java	36
5.3.19	FieldMapper.java	36
5.3.20	WifiConnection.java	37
5.3.21	WifiProperties.java	38
5.3.22	Parameters.java	38
6	AUTOMATION	39
6.1	Unit Testing	39
6.2	Continuous Integration Build	40
7	GLOSSARY OF TERMS	40

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
- March 29th 2017 (Version 4.0.0):
 - **John Wu:** Added class diagrams 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

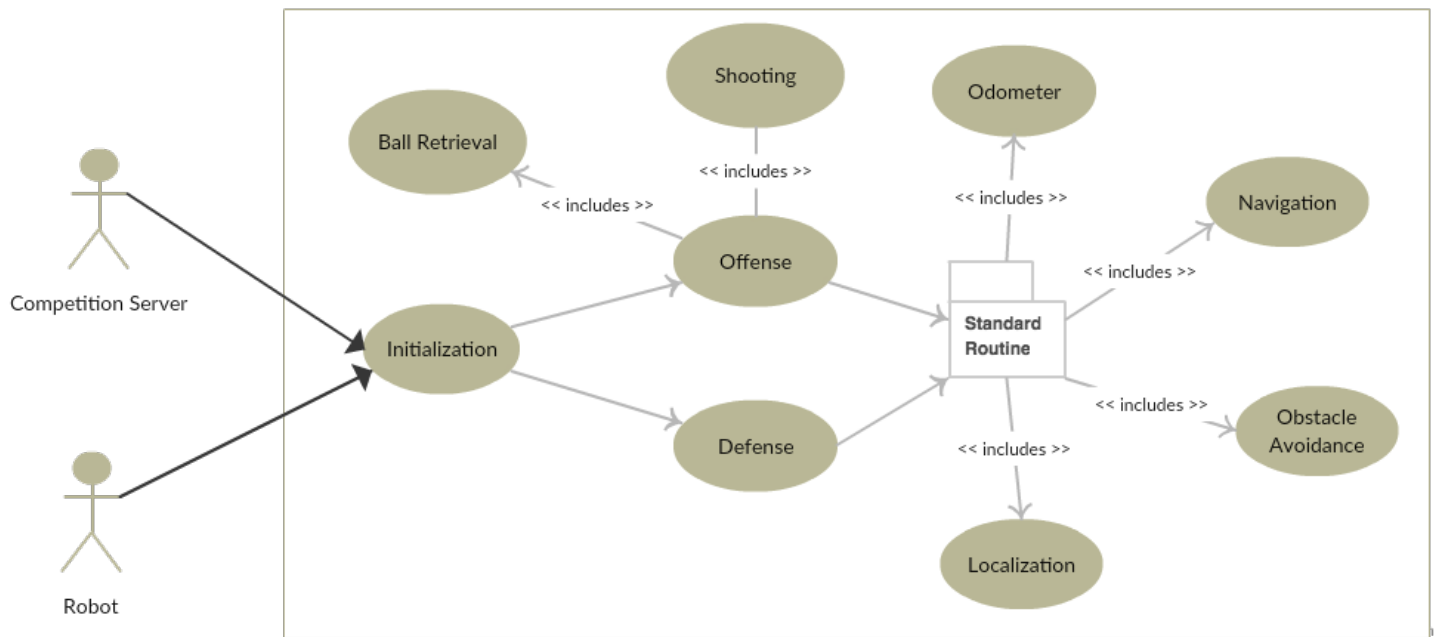
The scope of our project will be limited to only achieving the list of requirements specified by the client. Therefore, no additional features will be added.

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 its 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.

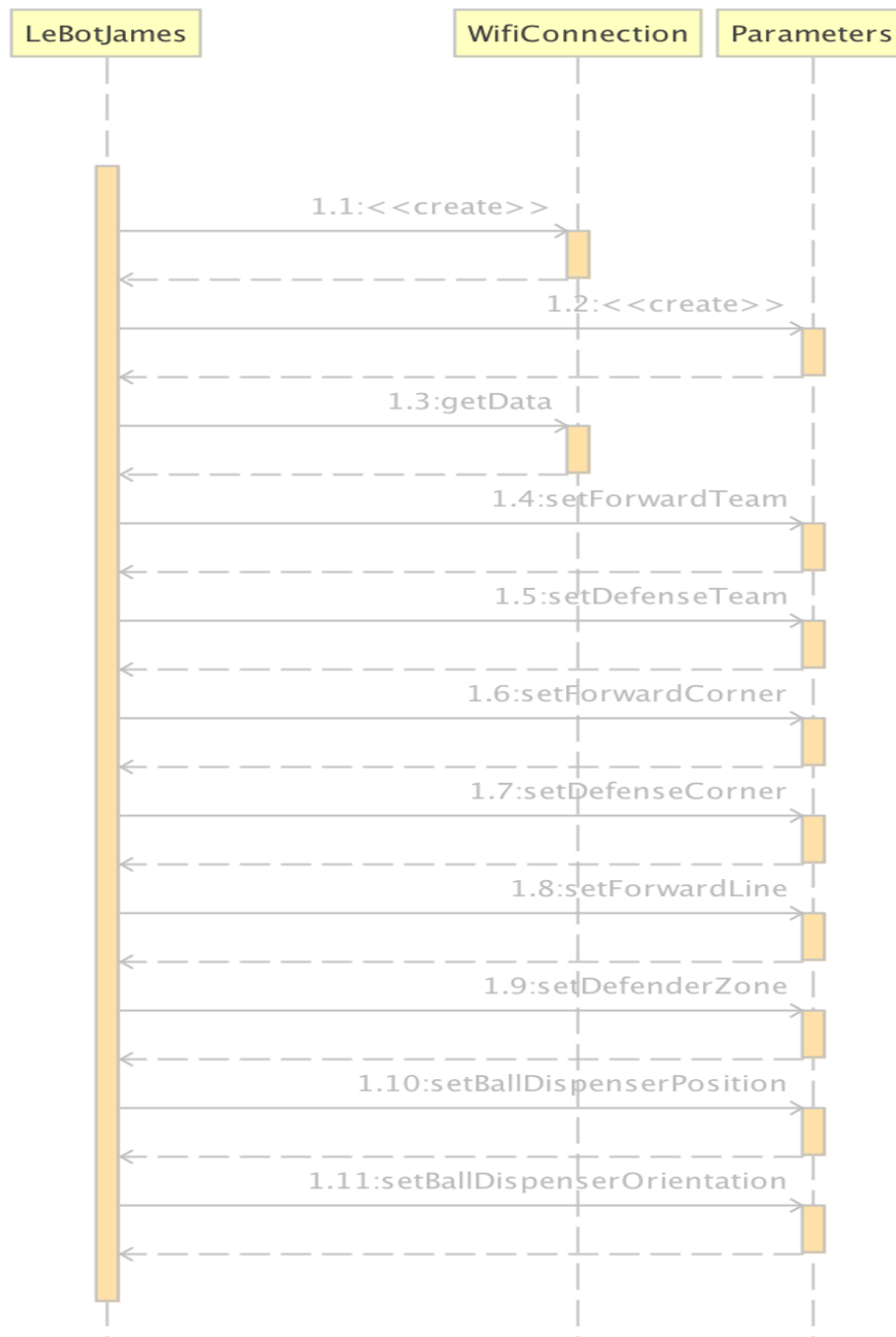
Figure 1: Use Case Diagram



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

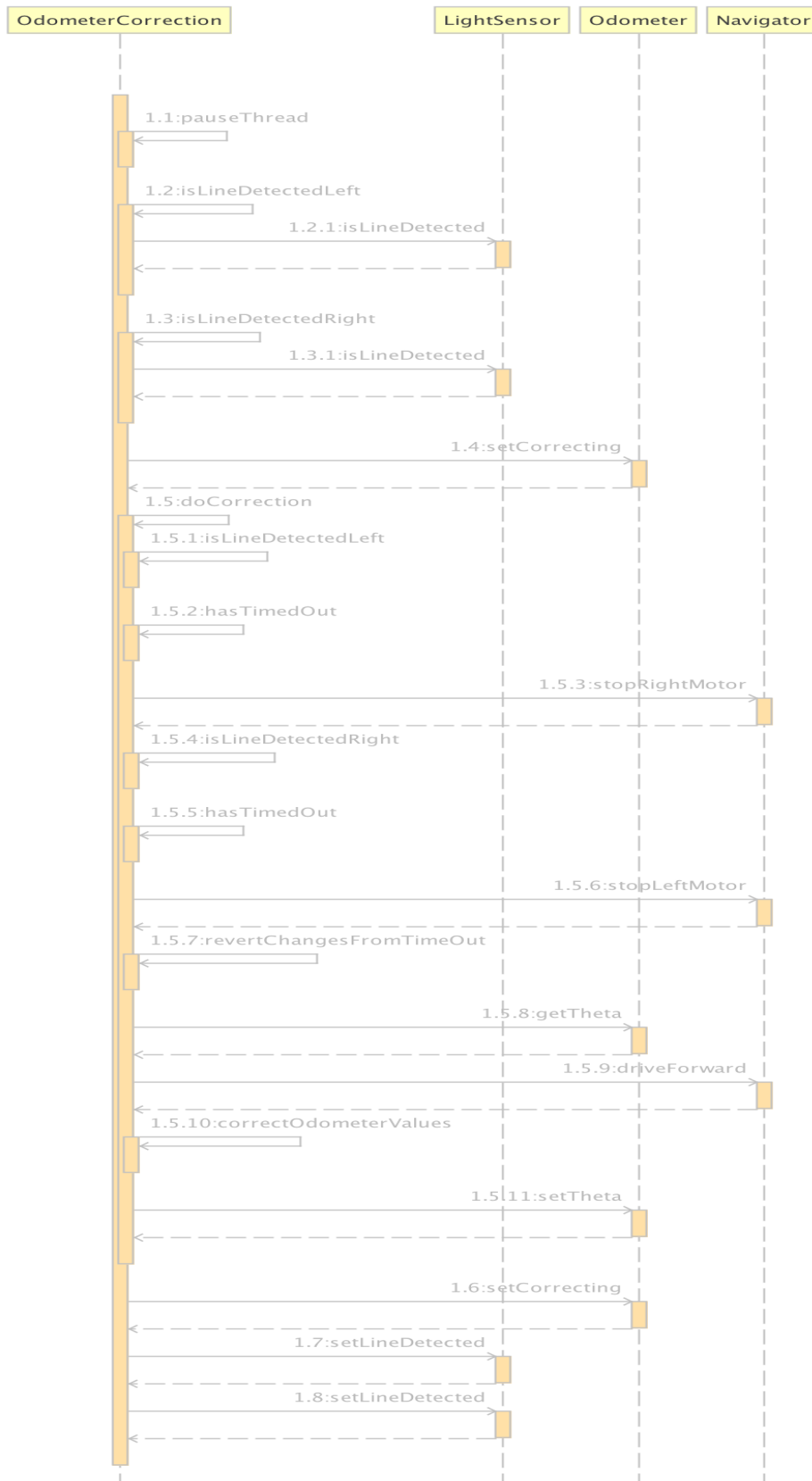


Figure 5: Navigation Sequence Diagram

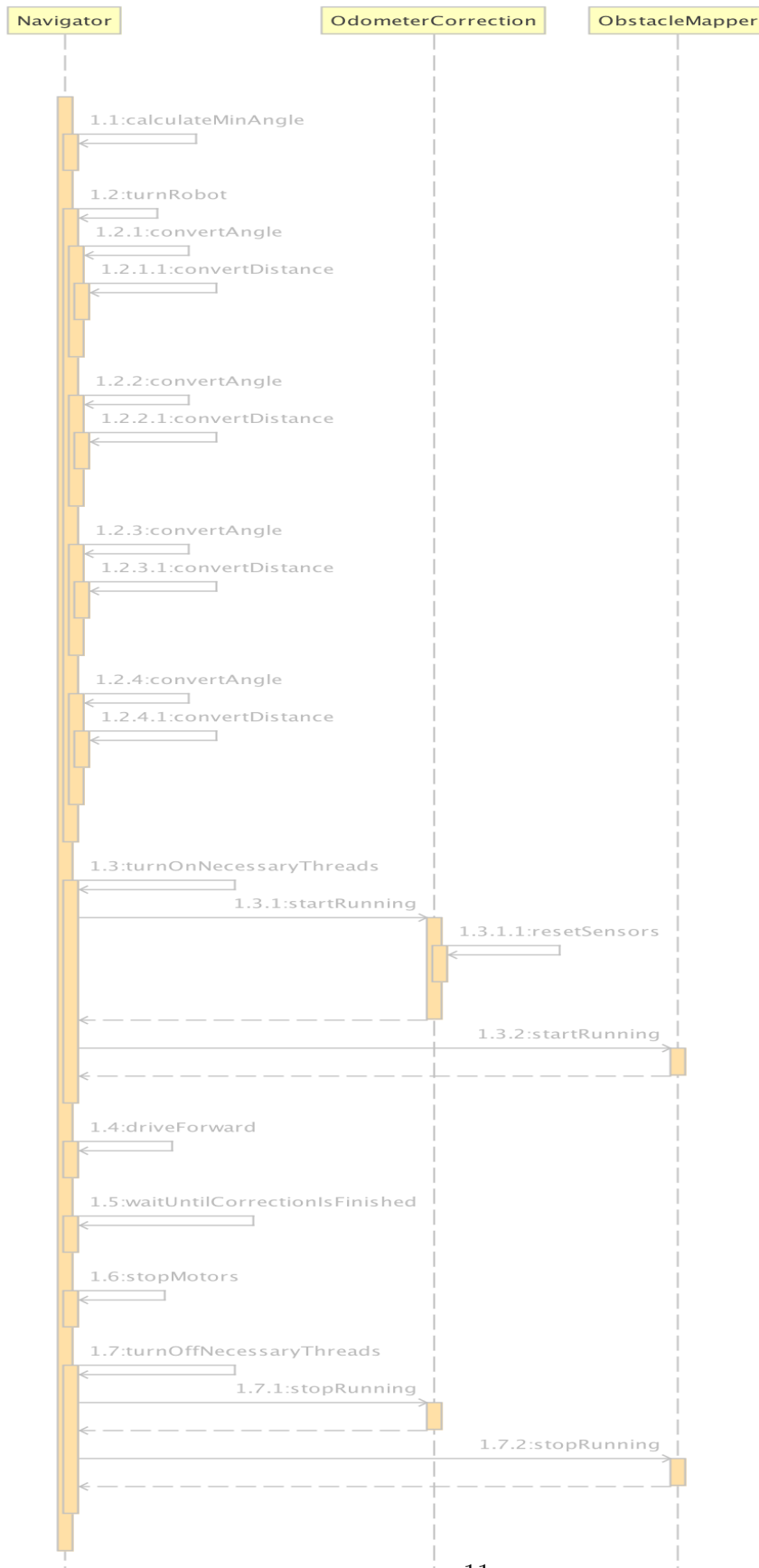
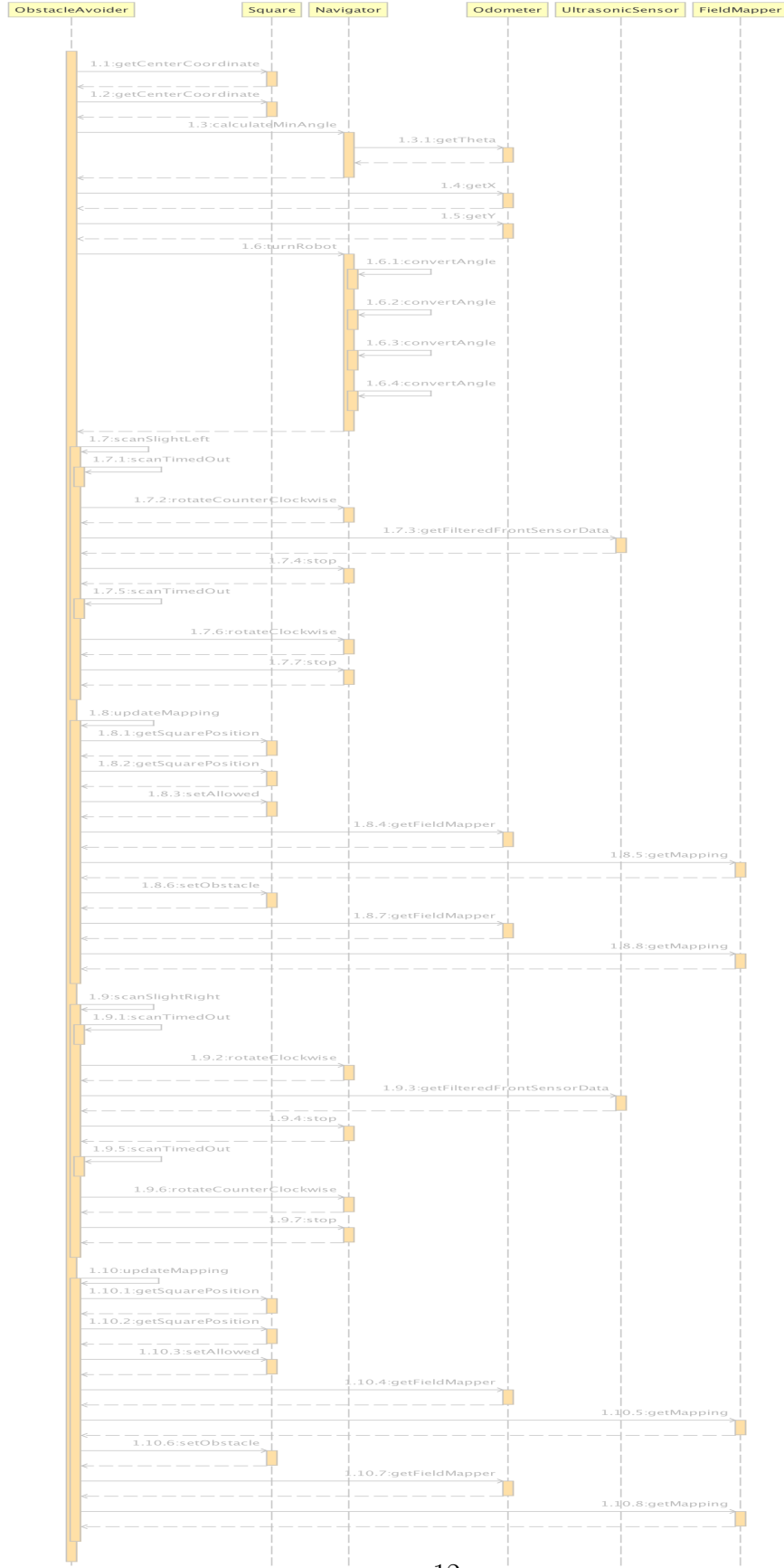


Figure 6: Obstacle Avoiding Sequence Diagram



3.7 Obstacle Mapping

In order to increase the speed at which the robot learns the field, an additional ultrasonic sensor is placed on the left side of the vehicle that continuously detects obstacles to add to the field mapping. The mapper is limited to mapping only adjacent squares and uses a mean filter to determine in order to avoid mapping false objects.

3.8 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.

3.9 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.

3.10 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: Obstacle Mapping Sequence Diagram

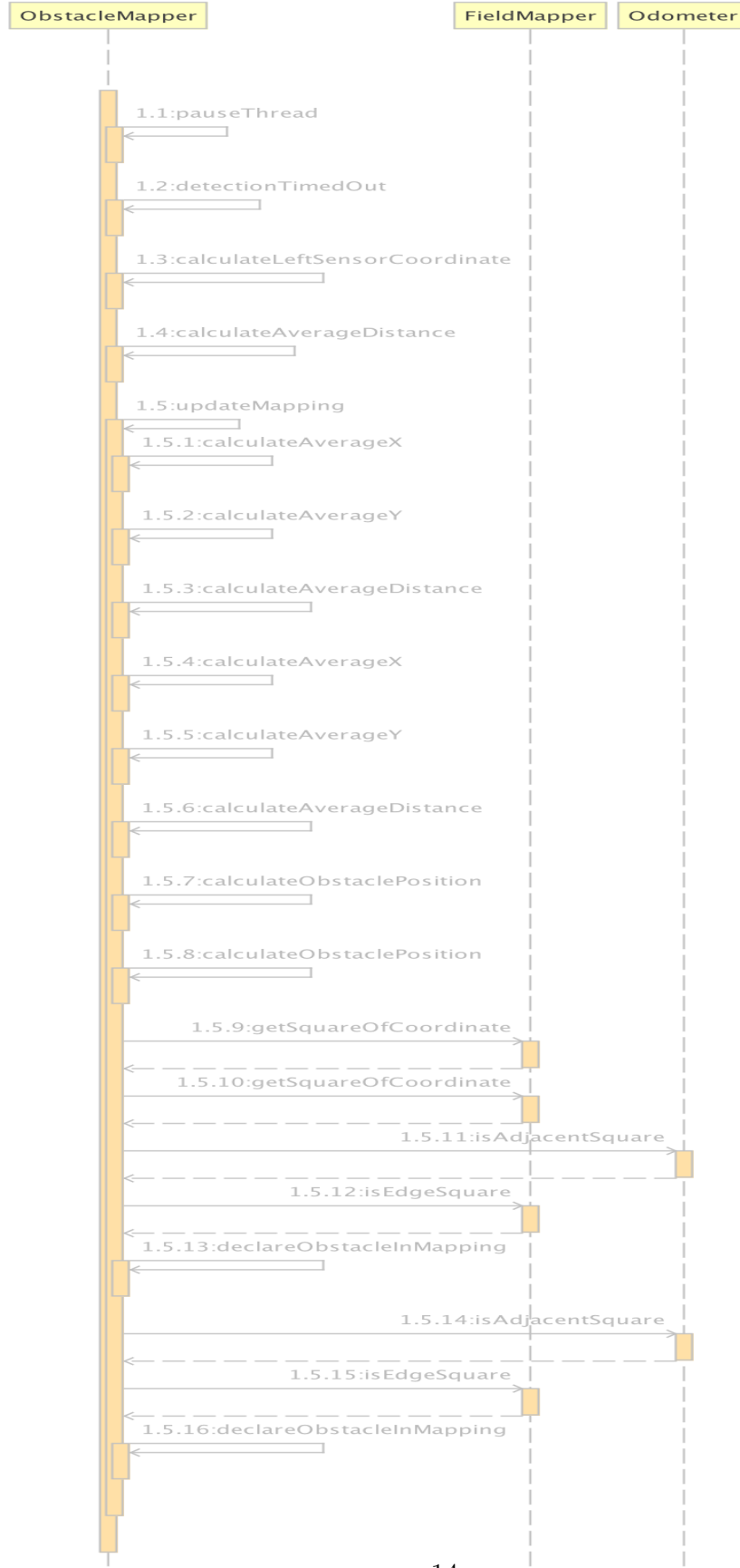


Figure 8: Localization Sequence Diagram

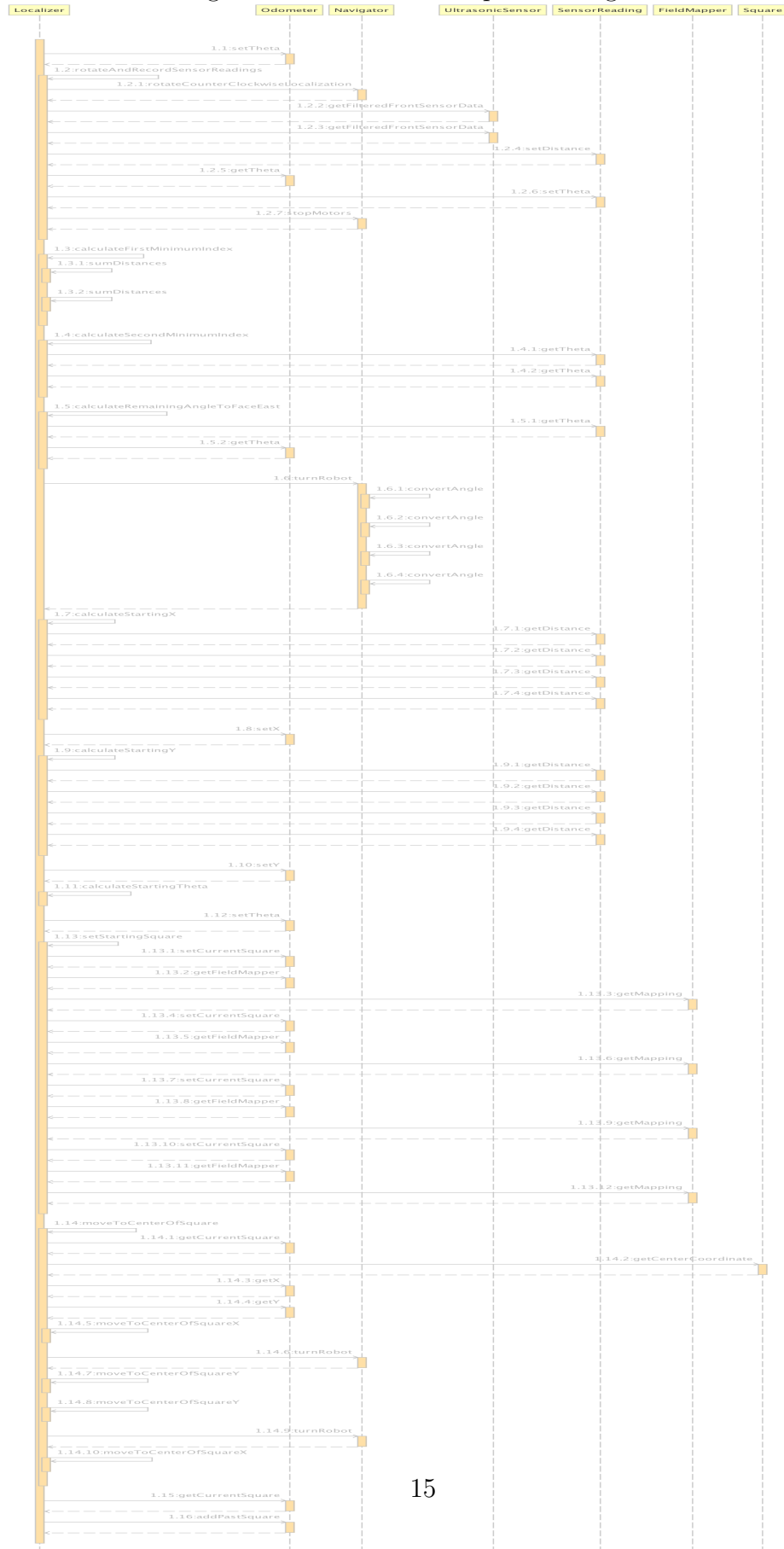


Figure 9: Ball Retrieval Sequence Diagram

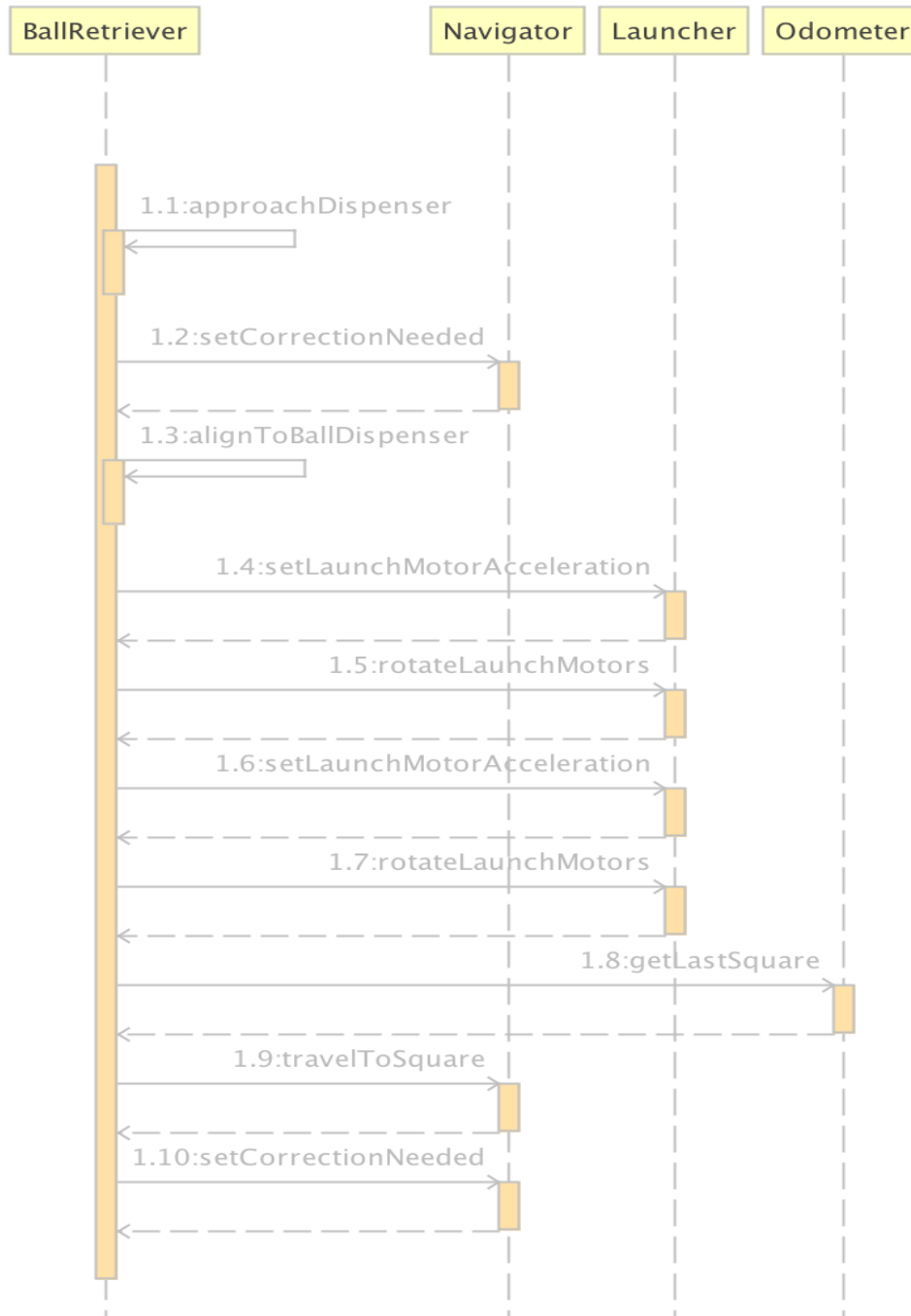


Figure 10: Field Mapping Sequence Diagram



3.11 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.

3.12 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.

3.13 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 11: Shooting Sequence Diagram

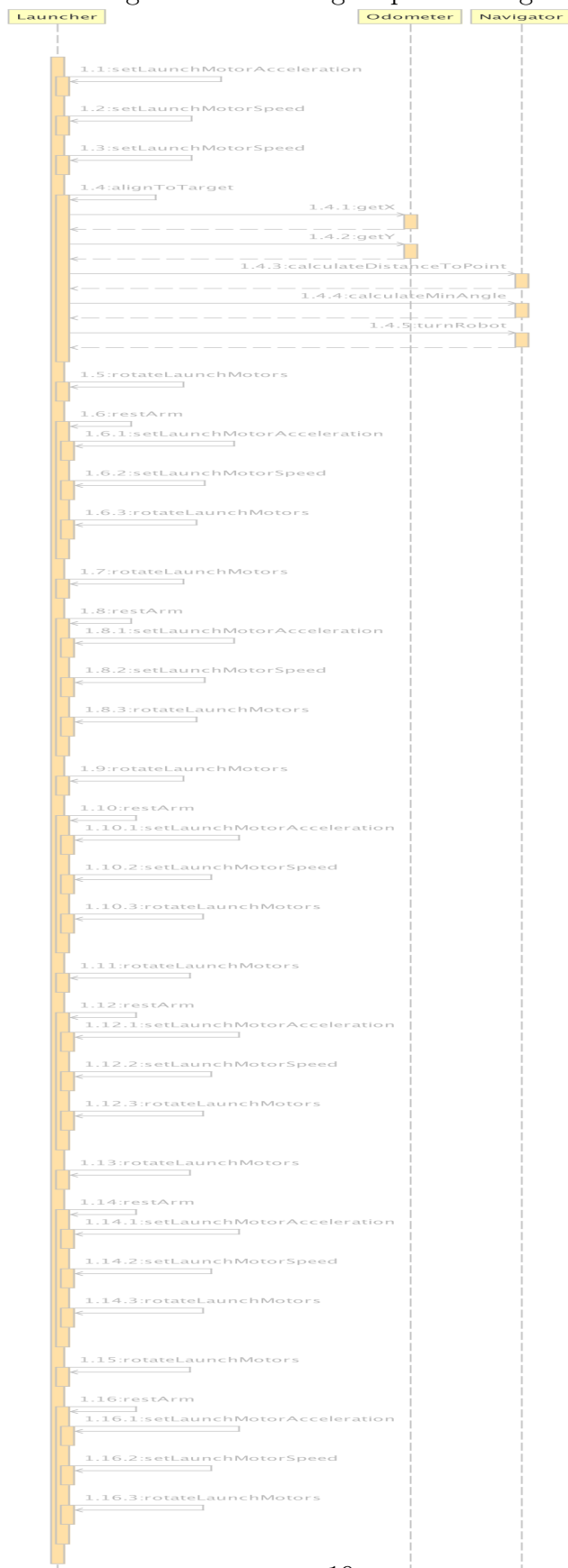


Figure 12: Offense Sequence Diagram

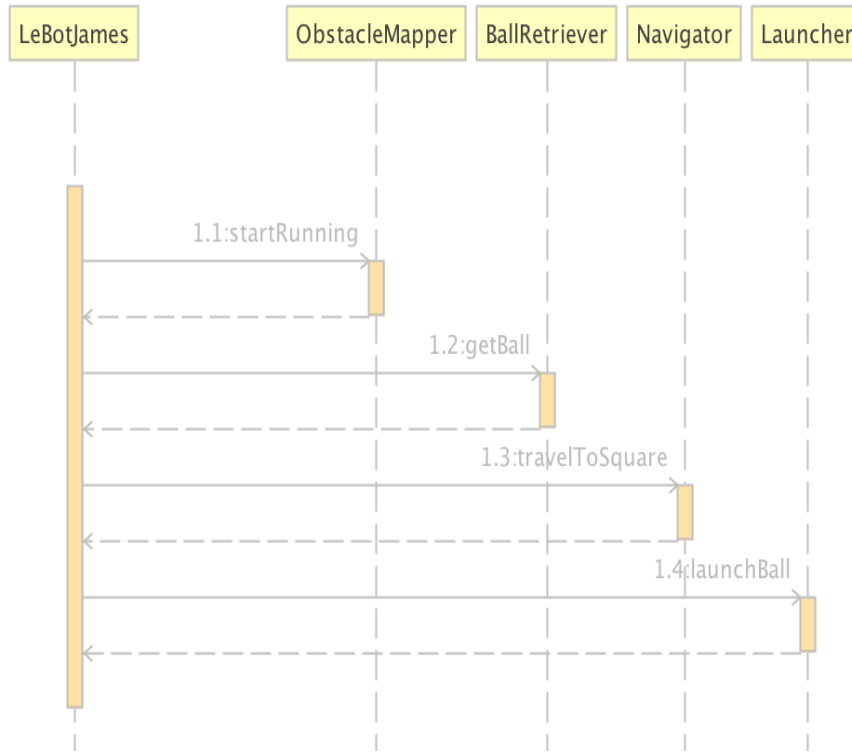
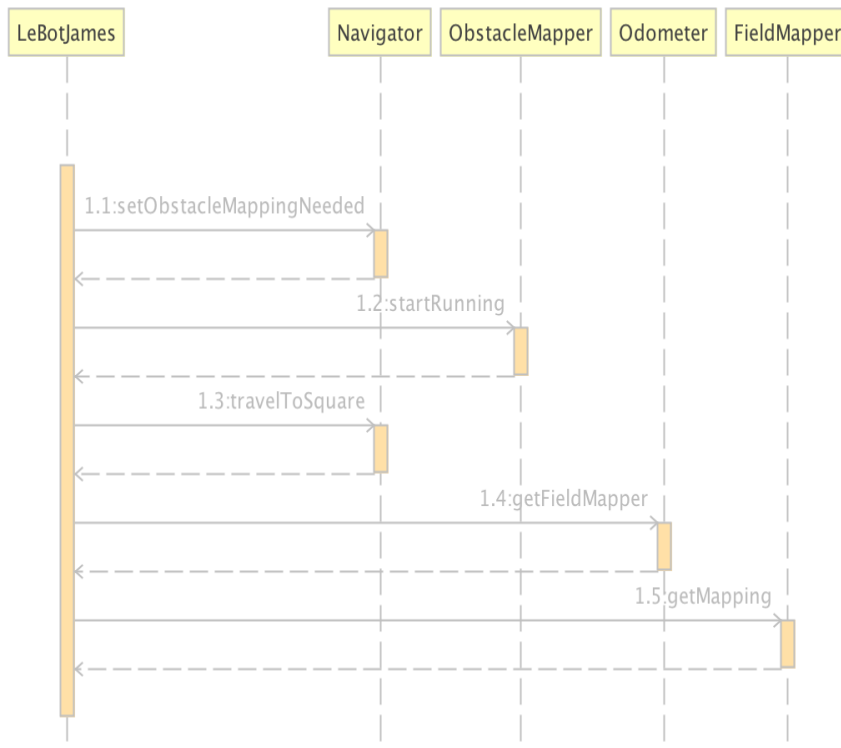


Figure 13: Defense Sequence Diagram



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.

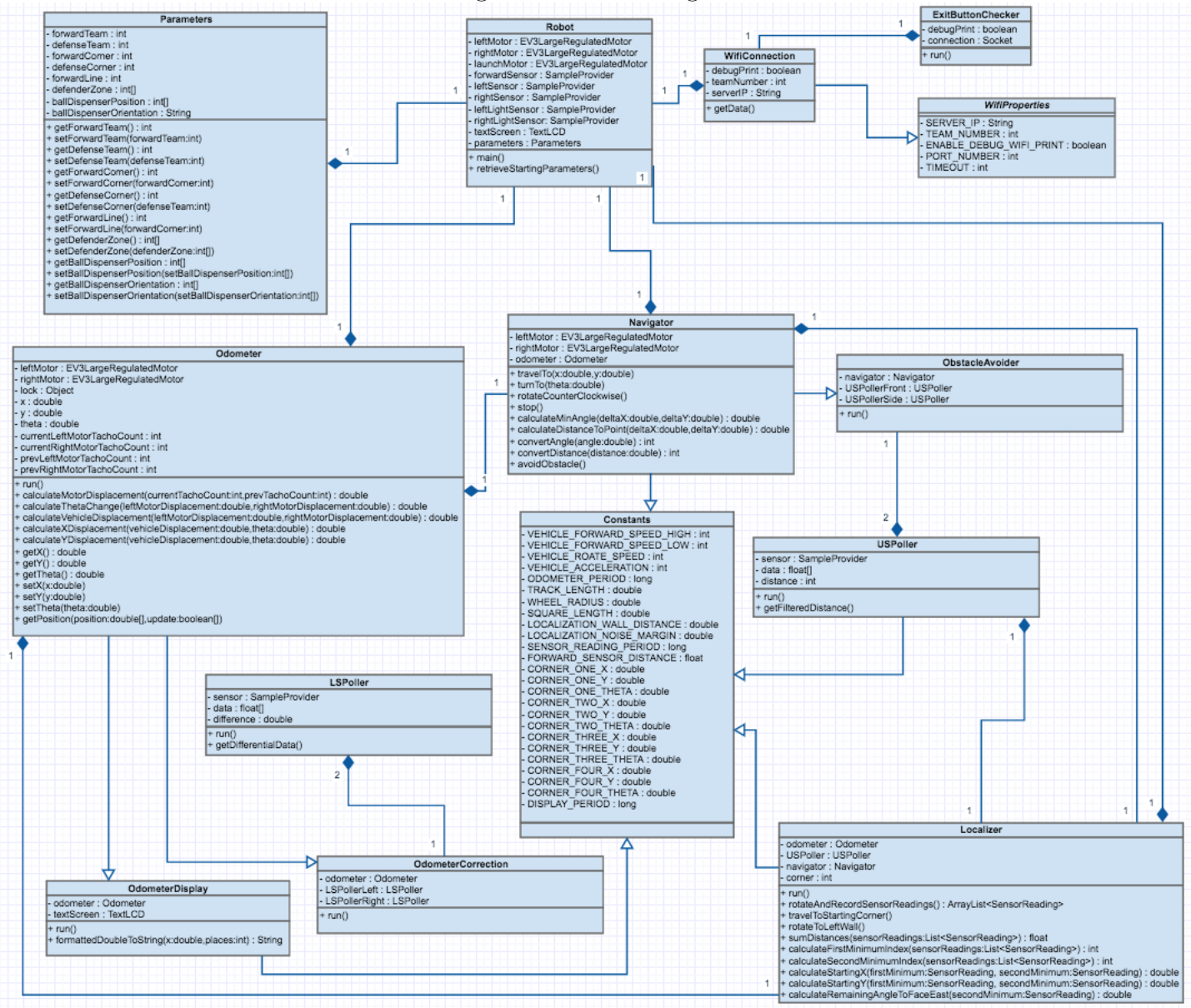
Future improvements to the software must take great caution with managing threads and ensuring safe usage at all times.

5 SYSTEM ARCHITECTURE

5.1 Overview

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

Figure 14: 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.

5.3 Classes

This section will cover each class in depth and its purpose within the software. The details of this section are an extension to the higher level software architecture show in section 5.1.












































5.3.1 LeBotJames.java

The LeBot James class is the main method of the robot which executes at the beginning of the program.

LeBotJames		
 leftMotor	EV3LargeRegulatedMotor	
 rightMotor	EV3LargeRegulatedMotor	
 leftLaunchMotor	EV3LargeRegulatedMotor	
 rightLaunchMotor	EV3LargeRegulatedMotor	
 forwardUltrasonicSensor	SampleProvider	
 leftUltrasonicSensor	SampleProvider	
 leftColorSensor	SampleProvider	
 rightColorSensor	SampleProvider	
 parameters	Parameters	
 main(String[])	void	
 playOffense(Navigator, Odometer, ObstacleMapper, BallRetriever, Launcher)	void	
 playDefense(Navigator, Odometer, ObstacleMapper)	void	
 retrieveStartingParameters()	void	
 getStartingCorner(Parameters)	int	

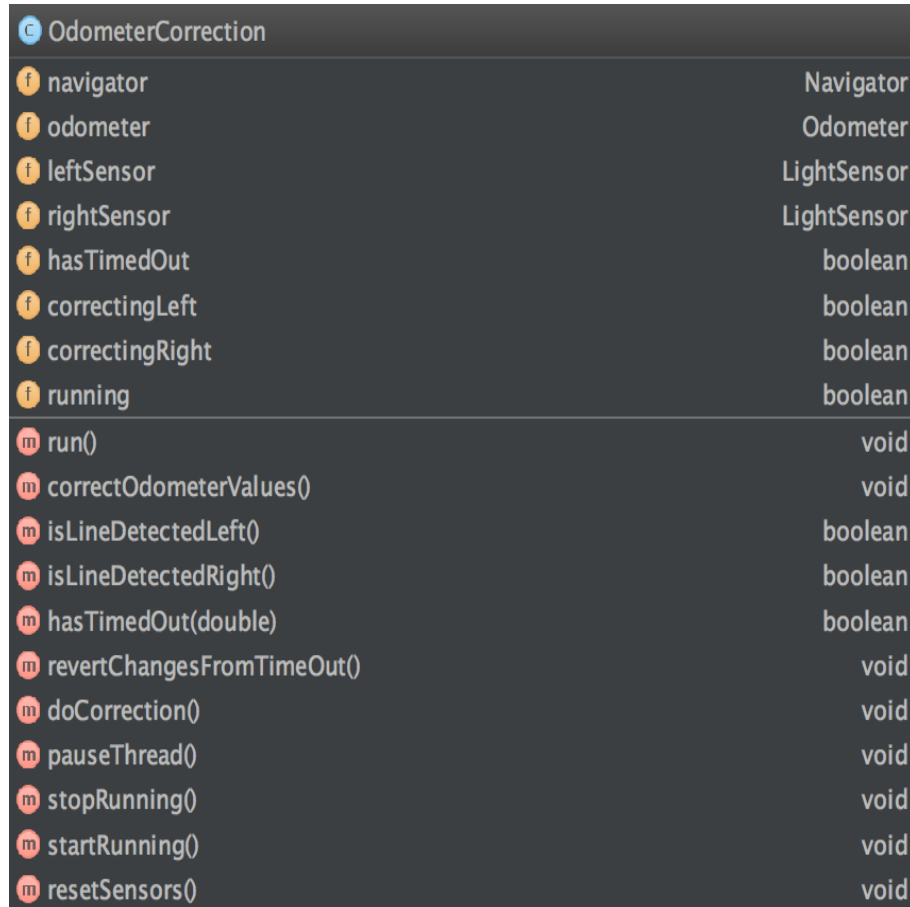
5.3.2 Odometer.java





















The odometer class is a controller used to determine the robot's true position at all times.

 Odometer	
 leftMotor	EV3LargeRegulatedMotor
 rightMotor	EV3LargeRegulatedMotor
 fieldMapper	FieldMapper
 lock	Object
 x	double
 y	double
 theta	double
 currentLeftMotorTachoCount	int
 currentRightMotorTachoCount	int
 prevLeftMotorTachoCount	int
 prevRightMotorTachoCount	int
 correcting	boolean
 currentSquare	Square
 pastSquares	ArrayList<Square>
 run()	void
 calculateMotorDisplacement(int, int)	double
 calculateThetaChange(double, double)	double
 calculateVehicleDisplacement(double, double)	double
 calculateXDisplacement(double, double)	double
 calculateYDisplacement(double, double)	double
 getX()	double
 getPastSquares()	ArrayList<Square>
 getLastSquare()	Square
 addPastSquare(Square)	void
 getY()	double
 getTheta()	double
 setX(double)	void
 setY(double)	void
 setTheta(double)	void
 updatePosition(double[], boolean[])	void
 isCorrecting()	boolean
 setCorrecting(boolean)	void
 getFieldMapper()	FieldMapper
 getCurrentSquare()	Square
 getEastSquare()	Square
 getWestSquare()	Square
 getNorthSquare()	Square
 getSouthSquare()	Square
 setCurrentSquare(Square)	void
 getCurrentDirection()	String
 getCurrentDirectionTheta()	double
 isAdjacentSquare(Square)	boolean

5.3.3 OdometerCorrection.java

The odometer correction class is a controller used to correct the odometer readings based on its surroundings.



	OdometerCorrection	
	navigator	Navigator
	odometer	Odometer
	leftSensor	LightSensor
	rightSensor	LightSensor
	hasTimedOut	boolean
	correctingLeft	boolean
	correctingRight	boolean
	running	boolean
<hr/>		
	run()	void
	correctOdometerValues()	void
	isLineDetectedLeft()	boolean
	isLineDetectedRight()	boolean
	hasTimedOut(double)	boolean
	revertChangesFromTimeOut()	void
	doCorrection()	void
	pauseThread()	void
	stopRunning()	void
	startRunning()	void
	resetSensors()	void

5.3.4 Navigator.java

The navigator class is a controller used to control all movements of the robot around the field.

Navigator	
odometer	Odometer
leftMotor	EV3LargeRegulatedMotor
rightMotor	EV3LargeRegulatedMotor
odometerCorrection	OdometerCorrection
obstacleAvoider	ObstacleAvoider
obstacleMapper	ObstacleMapper
correctionNeeded	boolean
obstacleMappingNeeded	boolean
recentMoves	ArrayList<Square>
movesInCurrentDirection	int
travelToSquare(Square)	void
makeBestMoves(Square)	void
getPossibleMoves(Square)	Stack<Square>
travelToX(double)	void
travelToY(double)	void
travelToXBackward(double)	void
travelToYBackward(double)	void
moveSquareX(int)	boolean
moveSquareY(int)	boolean
isMovePossible(Square)	boolean
isSquareAllowed(int, int)	boolean
getNextSquare()	Square
turnRobot(double)	void
rotateLeftMotorForwardSlow()	void
rotateRightMotorForwardSlow()	void
rotateLeftMotorBackward()	void
rotateRightMotorBackward()	void
rotateLeftMotorBackwardSlow()	void
rotateRightMotorBackwardSlow()	void
driveForward()	void
driveForwardSlow()	void
driveBackward()	void
rotateCounterClockwise()	void
rotateClockwise()	void
rotateCounterClockwiseLocalization()	void
rotateCounterClockwiseLocalizationFast()	void
stopMotors()	void
stopLeftMotor()	void
stopRightMotor()	void
calculateMinAngle(double, double)	double
calculateDistanceToPoint(double, double)	double
convertAngle(double)	int
convertDistance(double)	int
waitUntilCorrectionIsFinished()	void
setCorrectionNeeded(boolean)	void
setOdometerCorrection(OdometerCorrection)	void
stop()	void
stopFast()	void
getComponentDistances(Square)	int[]
turnToSquare(Square)	void
moveDistance(double)	void
setObstacleMappingNeeded(boolean)	void
turnOnNecessaryThreads()	void
turnOffNecessaryThreads()	void

5.3.5 ObstacleAvoider.java

The obstacle avoider class is a controller used to assist with obstacle avoidance and mapping by scanning squares for obstacles.

c ObstacleAvoider	
f frontSensor	UltrasonicSensor
f odometer	Odometer
f navigator	Navigator
m setNavigator(Navigator)	void
m scanSquare(Square)	boolean
m scanSlightLeft()	boolean
m scanSlightRight()	boolean
m scanTimedOut(double)	boolean
m updateMapping(Square)	void




















5.3.6 ObstacleMapper.java

The obstacle mapper class is a controller used to continuously map the field while the robot navigates

c ObstacleMapper	
f leftSensor	UltrasonicSensor
f odometer	Odometer
f running	boolean
m run()	void
m pauseThread()	void
m stopRunning()	void
m startRunning()	void
m detectionTimedOut(double)	boolean
m calculateAverageDistance(List<SensorReading>)	double
m calculateAverageX(List<SensorReading>)	double
m calculateAverageY(List<SensorReading>)	double
m updateMapping(ArrayList<SensorReading>)	void
m declareObstacleInMapping(Square)	void
m calculateObstaclePosition(double, double, double, double)	double[]
m calculateLeftSensorCoordinate(double, double)	double[]

5.3.7 Localizer.java

The localizer class is a controller used to determine the initial position of the robot at the beginning of execution.

 Localizer	
 odometer	Odometer
 ultrasonicSensor	UltrasonicSensor
 navigator	Navigator
 corner	int
 start()	void
 rotateAndRecordSensorReadings()	ArrayList<SensorReading>
 rotateToLeftWall()	void
 sumDistances(List<SensorReading>)	float
 calculateFirstMinimumIndex(ArrayList<SensorReading>)	int
 calculateSecondMinimumIndex(ArrayList<SensorReading>, int)	int
 calculateStartingX(SensorReading, SensorReading)	double
 calculateStartingY(SensorReading, SensorReading)	double
 calculateStartingTheta()	double
 setStartingSquare()	void
 moveToCenterOfSquare()	void
 moveToCenterOfSquareX(double[], double)	void
 moveToCenterOfSquareY(double[], double)	void
 calculateRemainingAngleToFaceEast(SensorReading)	double

5.3.8 Launcher.java

The launcher class is a controller used to align and shoot at the target during offensive methods.

Launcher	
f navigator	Navigator
f odometer	Odometer
f leftLaunchMotor	EV3LargeRegulatedMotor
f rightLaunchMotor	EV3LargeRegulatedMotor
m launchBall()	void
m retractArm()	void
m restArm(int)	void
m setLaunchMotorAcceleration(int)	void
m setLaunchMotorSpeed(int)	void
m rotateLaunchMotors(int)	void
m stopLaunchMotors()	void
m alignToTarget()	double








5.3.9 BallRetriever.java

The ball retriever class is a controller used to navigate to the ball dispenser to retrieve a ball during offensive methods.

BallRetriever	
f launcher	Launcher
f navigator	Navigator
f odometer	Odometer
f odometerCorrection	OdometerCorrection
f alignmentTimedOut	boolean
f aligningLeft	boolean
f aligningRight	boolean
m getBall()	void
m approachDispenser()	void
m chooseApproach()	Square
m alignToBallDispenser()	void
m moveToLine()	void
m alignToLine()	void
m hasTimedOut(double)	boolean
m holdTimedOut(double)	boolean
m revertChangesFromTimeOut()	void

5.3.10 LightSensor.java

The light sensor class is an object that polls and returns data from the light sensors through the LeJOS API.

 LightSensor	
 sensor	SampleProvider
 data	float[]
 lineDetected	boolean
 run()	void
 isLineDetected()	boolean
 setLineDetected(boolean)	void

5.3.11 Square.java

The square class is an object that represents a square on the field and stores all information relevant to a specific square.

Square		
f	squarePosition	int[]
f	centerCoordinate	double[]
f	allowed	boolean
f	obstacle	boolean
t	shootingPosition	boolean
f	northLine	double
f	southLine	double
f	eastLine	double
f	westLine	double
m	equals(Object)	boolean
m	hashCode()	int
m	getSquarePositionArray(int, int)	int[]
m	getSquarePosition()	int[]
m	getCenterCoordinate()	double[]
m	setCenterCoordinate(double[])	void
m	isAllowed()	boolean
m	setAllowed(boolean)	void
m	isObstacle()	boolean
m	setObstacle(boolean)	void
m	isShootingPosition()	boolean
m	setShootingPosition(boolean)	void
m	getNorthLine()	double
m	setNorthLine(double)	void
m	getSouthLine()	double
m	setSouthLine(double)	void
m	getEastLine()	double
m	setEastLine(double)	void
m	getWestLine()	double
m	setWestLine(double)	void

5.3.12 UltrasonicSensor.java

The ultrasonic sensor class is an object that polls and returns data from the ultrasonic sensors through the LeJOS API.

C UltrasonicSensor		
f	sensor	SampleProvider
f	data	float[]
f	running	boolean
m	run()	void
m	getFilteredFrontSensorData()	float
m	getFilteredLeftSensorData()	float

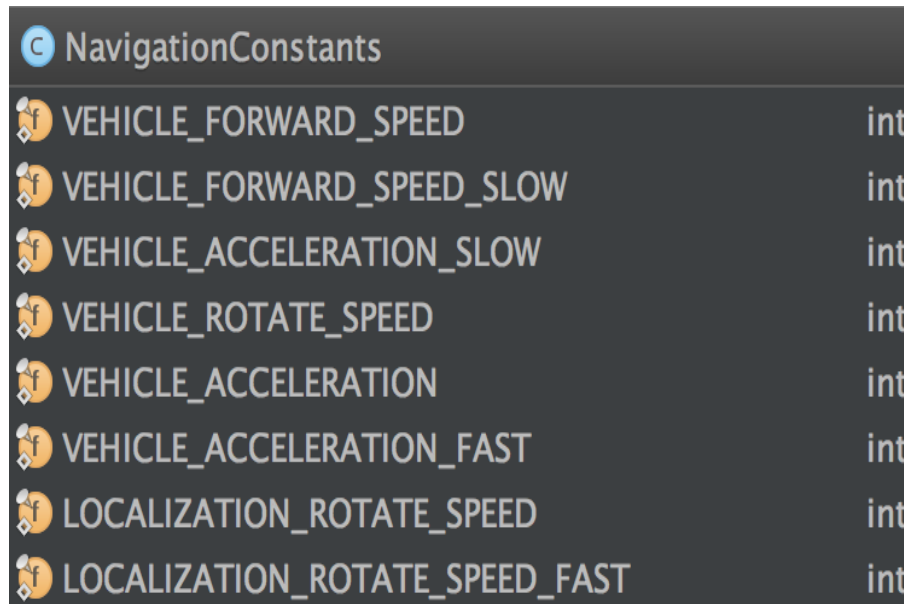
5.3.13 FieldConstants.java

The field constants class is a resource that stores all constants related to the mapping of the field.

C FieldConstants		
f	SQUARE_LENGTH	double
f	CORNER_ONE_X	double
f	CORNER_ONE_Y	double
f	CORNER_ONE_THETA	double
f	CORNER_TWO_X	double
f	CORNER_TWO_Y	double
f	CORNER_TWO_THETA	double
f	CORNER_THREE_X	double
f	CORNER_THREE_Y	double
f	CORNER_THREE_THETA	double
f	CORNER_FOUR_X	double
f	CORNER_FOUR_Y	double
f	CORNER_FOUR_THETA	double
f	TARGET_CENTER_Y	int
f	TARGET_CENTER_X	int
f	TARGET_CENTER_X_COORDINATE	double
f	TARGET_CENTER_Y_COORDINATE	double

5.3.14 NavigationConstants.java

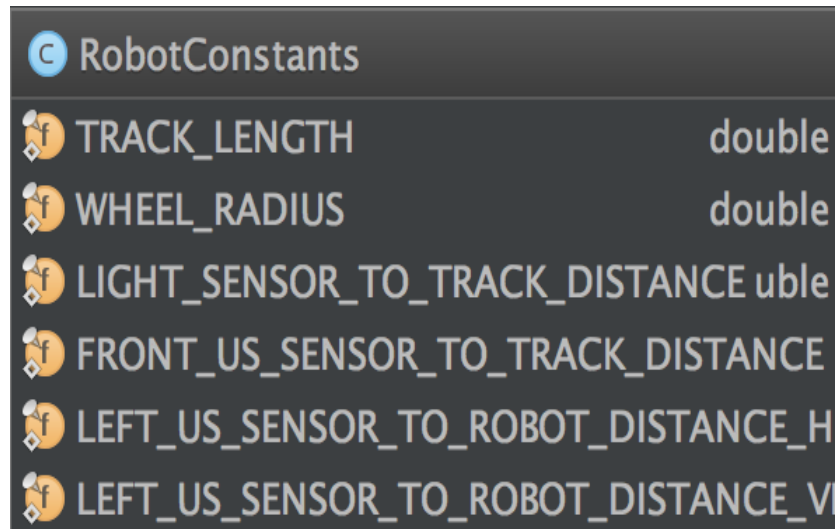
The navigation constants class is a resource that stores all constants related to the navigation of the field.



NavigationConstants	
VEHICLE_FORWARD_SPEED	int
VEHICLE_FORWARD_SPEED_SLOW	int
VEHICLE_ACCELERATION_SLOW	int
VEHICLE_ROTATE_SPEED	int
VEHICLE_ACCELERATION	int
VEHICLE_ACCELERATION_FAST	int
LOCALIZATION_ROTATE_SPEED	int
LOCALIZATION_ROTATE_SPEED_FAST	int

5.3.15 RobotConstants.java













The robot constants class is a resource that stores all constants related to the hardware of the robot.



RobotConstants	
TRACK_LENGTH	double
WHEEL_RADIUS	double
LIGHT_SENSOR_TO_TRACK_DISTANCE	double
FRONT_US_SENSOR_TO_TRACK_DISTANCE	double
LEFT_US_SENSOR_TO_ROBOT_DISTANCE_H	double
LEFT_US_SENSOR_TO_ROBOT_DISTANCE_V	double









5.3.16 ShootingConstants.java

The shooting constants class is a resource that stores all constants related to shooting the ball while on offense.

C ShootingConstants	
 BALL_RETRIEVAL_ANGLE	int
 BALL_LOWERING_ACCELERATION	int
 LAUNCH_MOTOR_ACCELERATION	int
 LAUNCH_MOTOR_RETRACTION_ACCELERATION	int
 LAUNCH_MOTOR_RETRACTION_SPEED	int
 LAUNCH_RETRACTION_ROM	int
 LAUNCH_ROM_4	int
 LAUNCH_ROM_5	int
 LAUNCH_ROM_6	int
 LAUNCH_ROM_7	int
 LAUNCH_ROM_8	int
 LAUNCH_ROM_MAX	int

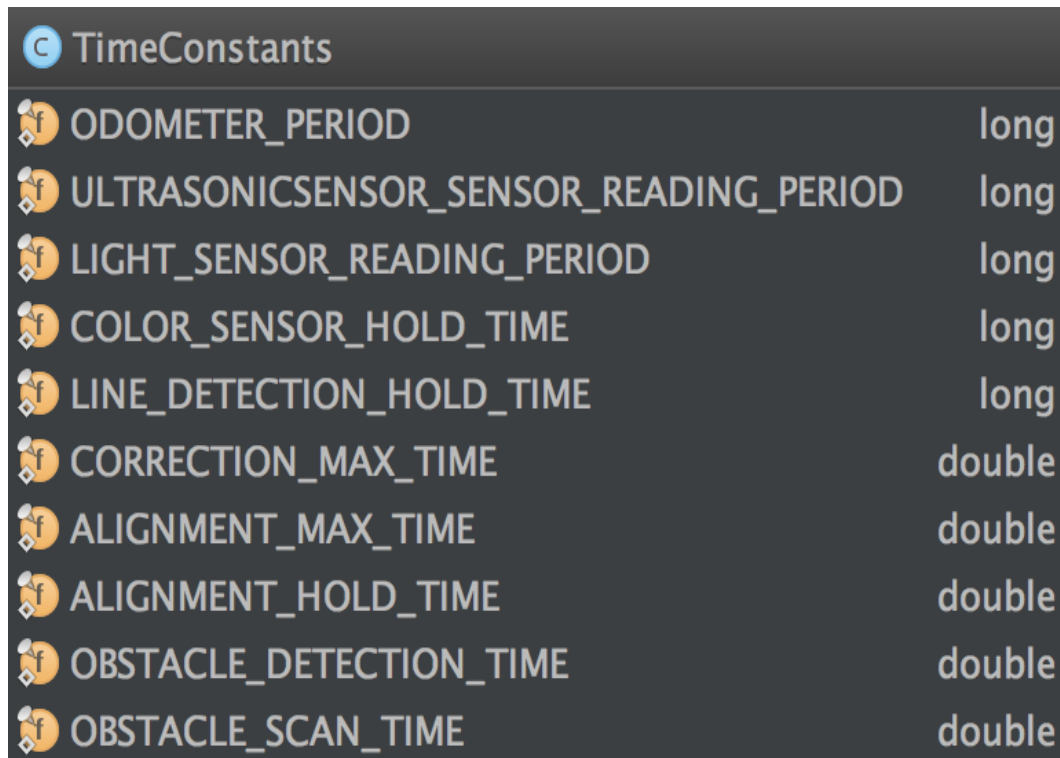
5.3.17 ThresholdConstants.java











The threshold constants class is a resource that stores all constants related to specific thresholds during execution.

C ThresholdConstants	
 LOCALIZATION_WALL_DISTANCE	double
 LOCALIZATION_NOISE_MARGIN	double
 ULTRASONICSENSOR_MAX_DISTANCE	int
 LINE_DETECTION	double
 POINT_REACHED	double
 OBSTACLE_TRACKING	double
 COORDINATE_IN_SQUARE	double
 BALL_RETRIEVAL_DISTANCE	double

5.3.18 TimeConstants.java


























The time constants class is a resource that stores all constants related to the timing of execution



C TimeConstants		
	ODOMETER_PERIOD	long
	ULTRASONICSENSOR_SENSOR_READING_PERIOD	long
	LIGHT_SENSOR_READING_PERIOD	long
	COLOR_SENSOR_HOLD_TIME	long
	LINE_DETECTION_HOLD_TIME	long
	CORRECTION_MAX_TIME	double
	ALIGNMENT_MAX_TIME	double
	ALIGNMENT_HOLD_TIME	double
	OBSTACLE_DETECTION_TIME	double
	OBSTACLE_SCAN_TIME	double






5.3.19 FieldMapper.java

The field mapper class is a utility that stores all information relative to the field for machine learning purposes.

 FieldMapper	
 squares	Square[][]
 ballDispenserApproach	Square[]
 parameters	Parameters
 mapField()	void
 isOffense()	boolean
 calculateCenterCoordinate(Square)	double[]
 isInGoalRegion(Square)	boolean
 isInOffenseRegion(Square)	boolean
 isInDefenseRegion(Square)	boolean
 isSquareAllowed(Square)	boolean
 isInitialObstacle(Square)	boolean
 isBallDispenser(Square)	boolean
 isShootingPosition(Square)	boolean
 getNorthLine(Square)	double
 getSouthLine(Square)	double
 getEastLine(Square)	double
 getWestLine(Square)	double
 calculateBallDispenserApproach()	Square[]
 getBallDispenserApproach(int)	Square
 getMapping()	Square[][]
 getParameters()	Parameters
 getSquareOfCoordinate(double, double)	Square
 isCoordinateInSquare(double, double, Square)	boolean
 isEdgeSquare(Square)	boolean







5.3.20 WifiConnection.java

The WiFi connection class connects with the competition server in order to receive starting parameters.

	WifiConnection	
	debugPrint	boolean
	teamNumber	int
	serverIP	String
	getData()	Map

5.3.21 WifiProperties.java

The WiFi properties class is stores all the constants used in the WiFi connection.

	WifiProperties	
	SERVER_IP	String
	TEAM_NUMBER	int
	ENABLE_DEBUG_WIFI_PRINT	boolean
	PORT_NUMBER	int
	TIMEOUT	int

5.3.22 Parameters.java

The parameters class is stores the parameters given by the server into a POJO format for easier access.

Parameters	
f forwardTeam	int
f defenseTeam	int
f forwardCorner	int
f defenseCorner	int
f forwardLine	int
f defenderZone	int[]
f ballDispenserPosition	int[]
f ballDispenserOrientation	String
m getForwardTeam()	int
m setForwardTeam(int)	void
m getDefenseTeam()	int
m setDefenseTeam(int)	void
m getForwardCorner()	int
m setForwardCorner(int)	void
m getDefenseCorner()	int
m setDefenseCorner(int)	void
m getForwardLine()	int
m setForwardLine(int)	void
m getDefenderZone()	int[]
m setDefenderZone(int[])	void
m getBallDispenserPosition()	int[]
m setBallDispenserPosition(int[])	void
m getBallDispenserOrientation()	String
m setBallDispenserOrientation(String)	void

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 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.