ECSE 211 Soft Design Document

Project Group 12

# Version: 1.0.0

# Date: Wednesday, March 14, 2018

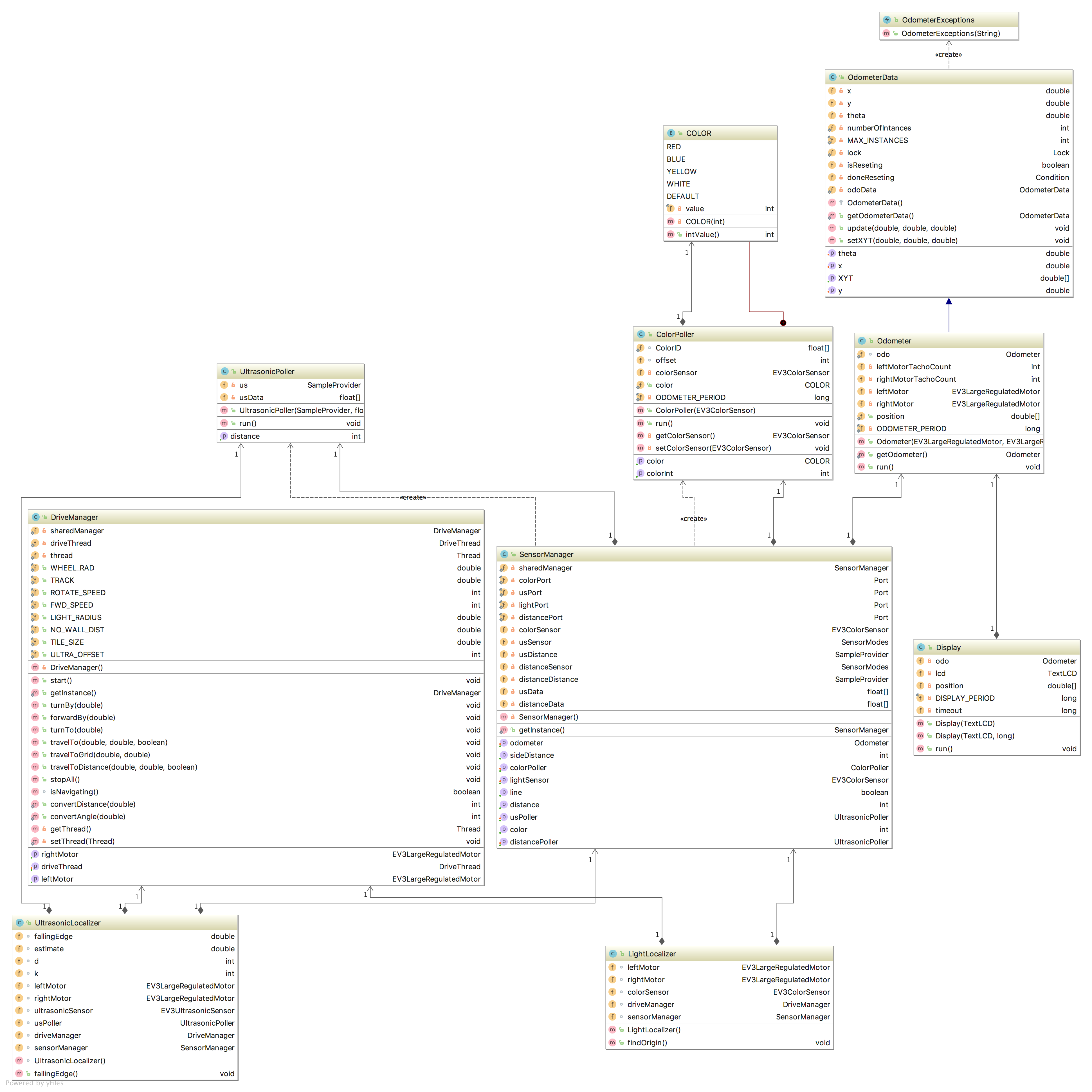
# Modified By: Connor Fowlie, 260687955

# ChangeLog:

* Initial version of document.

# Software Architecture

Architecture WIP. This design allows for sensors and drive to operate independent of the main project file on separate threads. This allows for fully asynchronous operation of the projects main java file and any other operations. This also allows for custom callback functions for asynchronous completion blocks at the end of specified operations (such as certain drive functions).



# Managers and Singletons

The two biggest objects within this architecture are the SensorManager and DriveManager singletons. Instantiated at first call, these managers spin up thread independently of the main for all sensors and drive functions. This allows these managers to be statically called from anywhere, and return values for any sensor or motor in a meaningful way by using static helper methods. This in turn, eliminates the need to pass any motor or sensor variables around to controller files, as they can be accessed via the singleton.

## Singleton Calling Example:

*// Init shared Managers***final** DriveManager driveManager = DriveManager.*getInstance*();  
**final** SensorManager sensorManager = SensorManager.*getInstance*();

## Singleton Use Example:

*// Set Odometer value and stop***driveManager**.stopAll();  
**sensorManager**.getOdometer().setTheta(0);

# Sensors & Localization

Sensors are all controller via the SensorManager as mentioned above. All sensors are declared as private class variables within the sensor manager, and then accessed and set via field encapsulation. They are initialized when the sensor manager is in its init() method, which because it is a singleton ensures all sensors are only initialized in one place.

## SensorManager Initialization:

*// Ultrasonic Sensor***private** SensorModes **usSensor** = **new** EV3UltrasonicSensor(***usPort***); *// usSensor is the instance***private** SampleProvider **usDistance** = **usSensor**.getMode(**"Distance"**);  
**private** SensorModes **distanceSensor** = **new** EV3UltrasonicSensor(***distancePort***); *// usSensor is the instance***private** SampleProvider **distanceDistance** = **distanceSensor**.getMode(**"Distance"**);*// usDistance provides samples from***private float**[] **usData** = **new float**[**usDistance**.sampleSize()]; *// usData is the buffer in which data are***private** UltrasonicPoller **usPoller**;  
**private float**[] **distanceData** = **new float**[**distanceDistance**.sampleSize()]; *// usData is the buffer in which data are***private** UltrasonicPoller **distancePoller**;  
  
**private** SensorManager() **throws** OdometerExceptions {  
   
 *// Odometer* **this**.setOdometer(Odometer.*getOdometer*());  
 Thread odoThread = **new** Thread(getOdometer());  
 odoThread.start();  
  
 *// Ultrasonic Sensor* **this**.setUsPoller(**new** UltrasonicPoller(**usDistance**, **usData**));  
 **this**.getUsPoller().start();  
   
 **this**.setDistancePoller(**new** UltrasonicPoller(**distanceDistance**, **distanceData**));  
 **this**.getDistancePoller().start();  
  
 *// Color Sensor* setColorPoller(**new** ColorPoller(**colorSensor**));  
 getColorPoller().start();  
  
 *// Light Sensor* setLightSensor(**new** EV3ColorSensor(***lightPort***));  
}

## Helper Methods

*/\*  
 \* Returns Forward Ultrasonic Sensors distance  
 \*/***public int** getDistance()

*/\*  
 \* Returns Side Ultrasonic Sensors distance  
 \*/***public int** getSideDistance()

*/\*  
 \* Returns true if currently over a line  
 \*/***public boolean** getLine()  
  
*/\*  
 \* Returns the Color Sensors current color  
 \*/***public int** getColor()

# Drive Use

Drive use is similar to the SensorManager, with the biggest difference being the DriveThread interface. This interface allows for all drive code to be run asyc by extending Thread() and adding a custom completion callback function, that can be defined when the drive thread is created. Although multiple Drivethreads can be created, it is unadvised as they will asynchronously attempt to control the motors (unless Semaphores are used to lock and sync which threads have control, this will possibly be added to DriveManger later). DriveManager currently only supports a single drivethread and is encapsulated as such. The completion block will be run asynchronously at the end of the run method in the drive thread.

## Helper Methods

*/\*\*  
 \* This method causes the robot to travel to the absolute field location (x,y),  
 \* specified in tile points.  
 \*   
 \** ***@param x*** *\** ***@param y*** *\** ***@throws*** *OdometerExceptions   
 \*/***public void** travelToDistance(**double** xDist, **double** yDist, **boolean** avoid) **throws** OdometerExceptions  
   
*/\*  
 \* Stops all motors  
 \*/***public void** stopAll() **throws** OdometerExceptions

*/\*\*  
 \* This method returns true if another thread has called travelTo() or   
 \* turnTo() and the method has yet to return; false otherwise  
 \*   
 \** ***@return*** *\* returns true if the robots motors are moving, false otherwise  
 \*/*

**boolean** isNavigating()

*/\* Convert distance in cm to motor rotation \*/***public static int** convertDistance(**double** distance

*/\*  
 \* Convert and angle  
 \*/***public static int** convertAngle(**double** angle

## Threading and Running Drive Code:

*// Create Drive Thread*

driveManager.setDriveThread(**new** DriveThread() {  
 @Override  
 **public void** run() **throws** InterruptedException, OdometerExceptions {  
 *// Run code goes here*  
 completion();  
 }  
  
 @Override  
 **public void** completion() **throws** OdometerExceptions {   
 *// Any code that should be run after the main drive method completes*

*// This will be run Async.*

}  
});

## Starting the Drive Thread

*// Start Drive Thread Async*driveManager.start();