Software Document

Project: LIBERTY

Task: SOFTWARE ARCHITECTURE

Document Version Number: 2.0
Date: 2017/11/28
Author: Bill Zhang
Editors: Andi-Camille Bakti, Gabriel Negash
Edit History: https://github.com/Gabetn/DPM_01_Project_Documentation

TABLE OF CONTENTS

1.0 Software Architectural Diagram

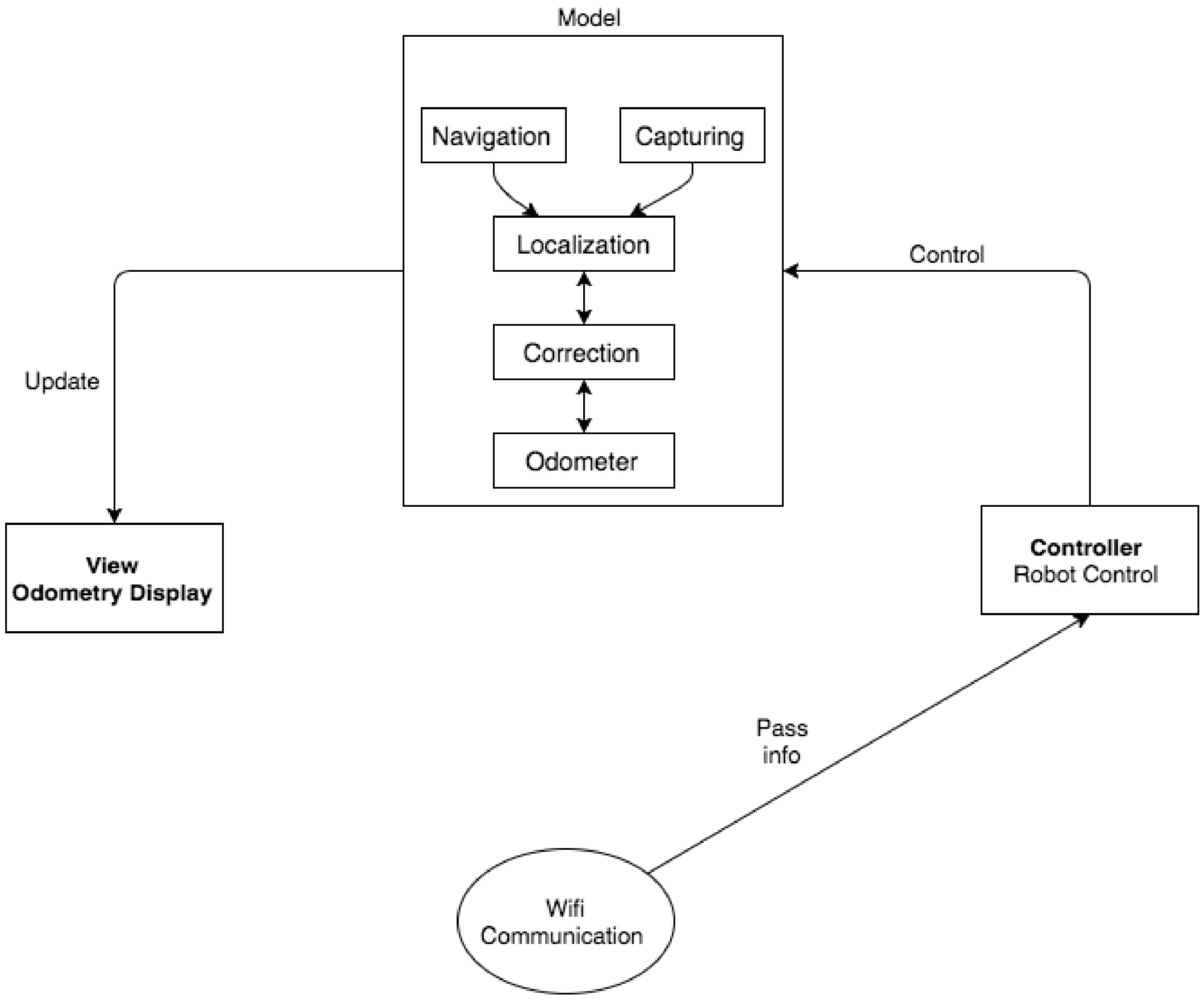

Figure 1: Diagram of the system’s software architecture.

2.0 Architectural Description & Rationale

Our design utilized aspects of two software architectural design patterns, Model-View-Controller (MVC) and Layered patterns. This hybridized architectural pattern is visualized in Figure 1.

The MVC architectural pattern allows software to separate and make swappable modules. Although there is no requirement to represent data and therefore our view in multiple ways, MVC was selected as the overall architectural style because it allows for swappable controllers which will allow for easier transitions between testing controllers and development controllers. Therefore, MVC helps to make the overall structure of the code very clean, and it allows us to update the functional classes without changing the controller.While it is well understood that there is no need to represent any data during the competition, having necessary information such as odometry data or way point marks displayed on the screen is very helpful for debugging purposes as well as testing purposes. Therefore, a minor reason to select MVC, on top of swappable controllers, is that it offers a decent way to display the data that is required for debugging purposes.

A layered architecture organizes a system into layers with related functionality associated with each layer, in order to facilitate incremental development on an existing system. As such a layered platform was chosen to facilitate the development of the functional classes found in the Model section of our MVC. By organizing the subsystems in the Model component with the bottom layer being the odometer and each layer sharing functionality with the layer beneath it each layer is not directly dependent on every other layer and does not expand the amount of dependencies in the codebase. The order of the layers we determined based off of dependencies as navigation and capturing depend on it location which is itself determined and corrected by localization and correction. The localization and correction are based on the odometer, therefore, the relation of their dependency determines that the odometer is in the bottom and functionality that is dependent is at the top. The selection of the layered architecture allows us to add and update the functional classes without changing the bottom layer. For example, both navigation and capturing are new classes written exclusively for the project, not having much if any basis in previous labs. As such they need to be updated and changed constantly during development. With the layered architecture, it is possible to update and change the navigation and capturing without changing the layers beneath them. The rework required for refactoring is minimized.

3.0 Architectural Components

The controller component in the architectural design corresponds to the robotControl class in the system. The model component corresponds to all the classes in the system except *robotControl* and odometry display. The view component corresponds to the odometry display class in the system. This class might be removed after debugging to reduce the size of the system.

4.0 Layered Block Diagram

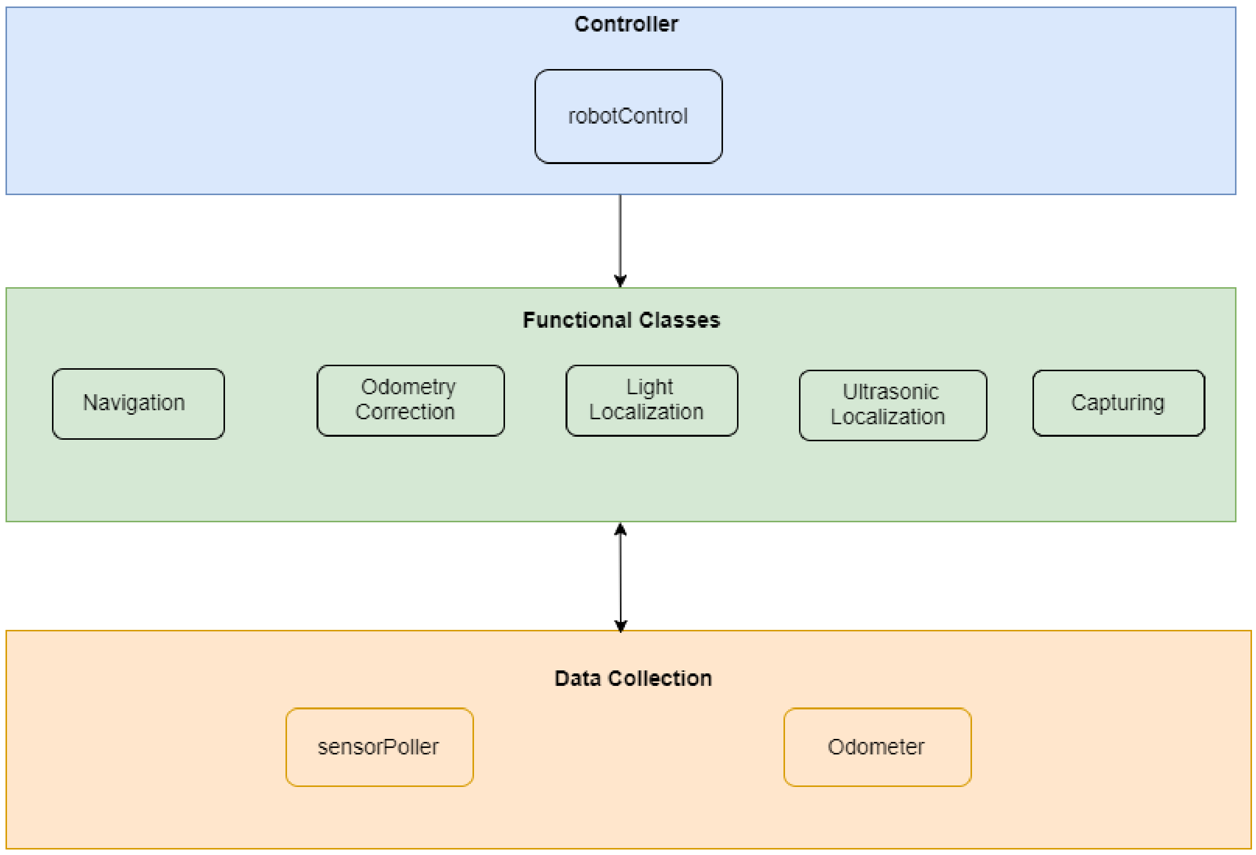

Figure 2: Layered Architectural Diagram

In addition to the MVC architecture, a layered structure is implemented as is mentioned in section 2. The entire system is divided into 3 layers named after their properties. Starting from the bottom, the data collection layer is composed of odometer and the sensor poller whose responsibilities are to track location and collect data from sensors, respectively. The common point is that they both provide data about the status of the robot for the functional classes to perform certain tasks.

The layer above the data collection layer is the functional class layer, as the name suggests, this layer is composed of functional classes that are designed to perform individual task such as localization, navigation, capturing and etc. These classes utilize the data collected from the bottom layer. Also, as the data collection layer does not depend on the functional layer, and since the functional layer is much more likely to be updated during development, it is placed above the data collection layer to reflect that a change in the function classes will not require any change in the data collection class.

The top layer is the controller layer. It is the simplest class in term of the size of the class because its sole task is to manage sequential and concurrent processes. However, multi-threading management takes place in this layer and it can be challenging to ensure more than one threads running perfectly. Also, the integration phrase will mainly take place in this class. Hence, the frequency of updating and changing for this class is the highest among all classes. It is important to place this layer on the top to reduce the required changes that are caused by changing the controller class.