Bachelor Project - CAPTAIN

Appendix

Bachelor - Electronics Engineering
7. Semester project Fall 2017
Group: 17115

Aarhus University, School of Engineering
Supervisor : Michael Alrøe

19. December 2017

CAPTAIN

Nicolai K. Bonde
Studienr. 201405146
AU ID: au512366

Troels Ringbøl Brahe
Studienr. 20095221
AU ID: au283169

Abstract

This project details the development of an autonomous naval surveying vessel in the Autumn of 2017 named CAPTAIN. Several very different modules had to be developed for the system and work together simultaneously to fulfill the requirements.

The system has a website user interface with an interactive map to enable the user to give commands to sail to a certain coordinate or cover an area. The commands are sent to a server which passses them on to the controller unit.

The controller can use its navigation algorithms to calculate and traverse paths on autopilot with the help of a GPS, and give feedback to the user about progress.

The user is also able to view live diagnostics data, and even to modify some of the parameters used within the navigation algorithms and the autopilot in real-time.

The results of this project was a fully functioning prototype that was extensively tested and verified.

Resumé

Dette projekt beskriver udviklingsforløbet af en selv-sejlende båd der kan kortlægge havområder kaldet CAPTAIN. Flere meget forskellige moduler skulle udvikles til systemet, og skulle fungere samtidigt for at opfylde kravene.

Systemet har en brugergrænseflade på en hjemmeside med et interaktiv kort der giver brugeren mulighed for at give kommandoer til at sejle til et bestemt sted, eller at kortlægge et bestemt område. Kommandoerne sendes til en server som sender disse videre til en kontrol-enhed.

Kontrol-enheden anvender navigations-algoritmer til at beregne og følge en sejlrute på autopilot med hjælp fra en GPS, og give feedback til brugeren om hvor langt man er nået.

Brugeren kan også se diagnostiske øjebliks-data for båden, og endda ændre nogle af parametrene der bruges i navigationsalgoritmerne og autopiloten.

Resultatet af projektet er en fuldt ud funktionsdygtig prototype der klarede alle tests og opfylder formålet.

Contents

1 Preface **4**

2 Introduction **5**
2.1 Areas of responsibility 6

3 Requirements **8**
3.1 Use case 3 - Edit parameter profile 8
3.2 Use case 11 - Calculate coverage path 9
3.3 Use case 12 - Run coverage path 10
3.4 Use case 5 - Request diagnostics 10

4 Scope **11**

5 Method **12**
5.1 ASE model 12
5.2 V model 12
5.3 SysML 13

6 Analysis **15**

7 Architecture **17**
7.1 Hardware architecture 17
7.2 Software architecture 18

8 Design **23**
8.1 Protocols 23
8.1.1 Communication between server and client 23
8.1.2 NMEA 0183 v2 24
8.2 Hardware design 24
8.2.1 Boat 24
8.2.2 System overview 25
8.3 Software design 26

9 Implementation **29**

10 Test **30**
10.1 Unit tests 30
10.2 Integration tests 31
10.3 Acceptance test 31

11 Results **33**

12 Discussion **35**

13 Future work **37**

14 Conclusion **38**

1 Preface

This documents details the development process of the Boat Autopilot "CAPTAIN" bachelor project, developed at Aarhus University School of Engineering. The report was written by Troels Ringbøl Brahe and Nicolai Kjærsgaard Bonde. During development of the project, both were studying Electrical Engineering.

Along with this process report, a project report has been written, as well as documentation, meeting schedule, and additional items in the appendix.

Associate professor Michael Alrøe was the project supervisor.

The project will be handed in December 19th 2017, and will be evaluated at an oral exam on January 17th 2018.

Thanks goes to Michael Alrøe for valuable supervision and friends and family for giving feedback on this report before handin.

2 Introduction

In recent years, there has been an increased focus on autonomous vehicles, both by land, sea, and air. Self-driving cars, drones, and automatic survey vessels are examples of this, and the trend seems to continue as the technology matures and becomes accepted more broadly in society.

Autonomy ensures repeatability, reduces many categories of risk, and - by definition - eliminates the error-prone human element from the equation, often leading to much better results and favourable cost.

This project focuses on developing a system that can survey an area of the sea from inputs in a UI autonomously, thus removing the need for route planning by surveyors, enabling them to focus on data analysis rather than data acquisition.

An overview of the system is seen in figure 1.

As such, the system requires a number of components.

A UI which allows the user to set up parameters such as the width of the survey equipment, and specify an area on a map to cover in an intuitive way. The map will be updated continually if a network is available, otherwise an offline version of the most important quadrants is used.

A controller which takes care of navigation and autopiloting based on user commands, parameters, and GPS data. Relevant information such as position of the ship, planned route, and completed route are then shown in the UI.

A server which hosts the website and writes/reads data to/from the website and controller.

A prototype ship with a rudder and thruster and hull space for the controller unit, GPS, battery, motor controller(s).

A command computer to communicate with the controller and view the UI.

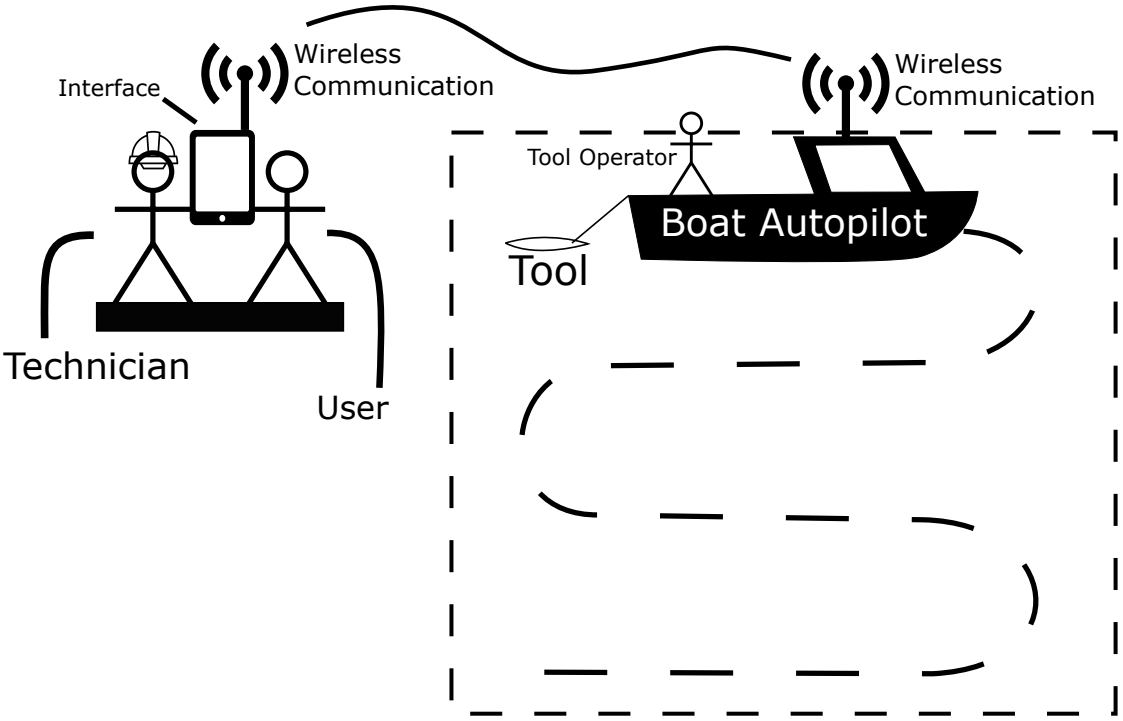

Figure 1: Rich image of system

2.1 Areas of responsibility

Table 1 shows the general outline of which group member had the responsibility for which area. As seen, each section was a shared effort, but generally not equally time-consuming for each developer.

Requirements specification, architecture, defining the acceptance test, and choosing system components was done as a collaborative effort, only when the project entered the design and implementation phase did tasks become individual.

Tables 2, 3, and 4 shows a more detailed view of these responsibilities.

Overview	
Area	**Developer**
Website	Troels and Nicolai
Controller	Troels and Nicolai
Hardware	Troels and Nicolai

Table 1: Overview of areas of responsibility

Website	
Task	**Developer**
Initial setup in bootstrap	Nicolai
Edit parameters	Troels and Nicolai
Coverage	Nicolai
Point to point	Troels
Status	Nicolai

Table 2: Areas of responsibility for the website

Controller	
Class	**Developer**
JSONReceiver	Nicolai
Navigation	Troels
Autopilot	Nicolai
DCMotor	Troels and Nicolai
Servo	Troels and Nicolai
JSONTransmitter	Troels
GPS	Nicolai
Unit tests	Troels and Nicolai
Integration tests	Troels

Table 3: Areas of responsibility for the controller

Hardware	
Area	**Developer**
Component analysis and acquisition	Troels and Nicolai
Design	Nicolai
Implementation	Nicolai

Table 4: Areas of responsibility for the hardware

3 Requirements

To get a sense of the system and what requirements the system should have, some mock-ups of a ui were created and can be found in the documentation in section 2.1 on page 7.

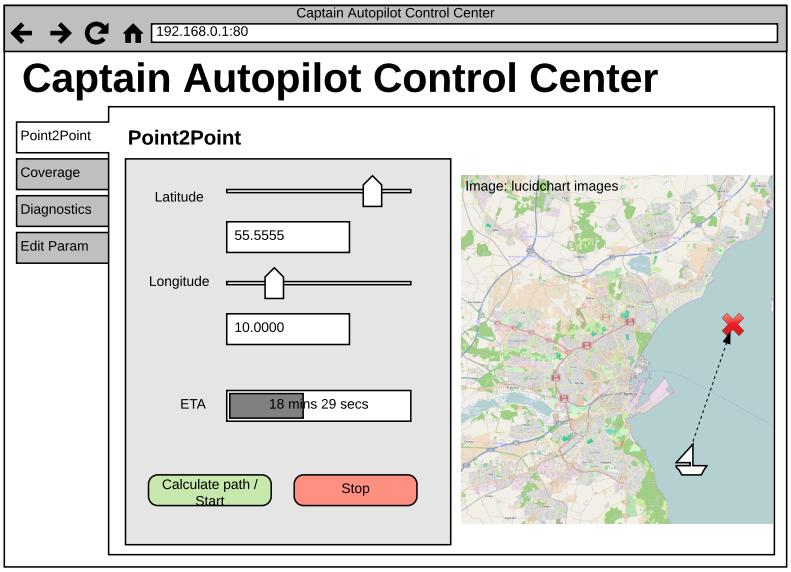

Figure 2: Mockup for the Point to point menu

An example of one of these mock ups is for describing a way point to navigate towards, can be seen in figure 2. It describes how the a user should interact with the system, and how the system should communicate to the user.

To describe the functionallity in further detail, a user case driven approach has been used. First of all the actors of the system has to be identified. In figure 2 is the use case diagram for the system, on the right are the actors that initiate a use case. On the left the other actors are.

Initiating actors or primary actors of the CAPTAIN system, are a technician, and a user. The technician is an actor who setups the system, and has a more in depth knowledge of the system then the user. The user could be anyone, since all of the complicated work should be handled by the system or the technician.

The usecase diagram on figure 3 also list 13 different use cases. A use case describe a way to use the system, in this system they mainly describe a button of function that can be started by the user. In this section a few use cases will be looked at, but for all the fully dressed use cases, one can have a look at the documentation section 2.3 on page 13.

3.1 Use case 3 - Edit parameter profile

One of the first things that was realized the system needed, was a way to describe parameters of the system. Parameters could be PID-loop terms, or the size of the boat, i

Figure 3: Use case diagram

fact anything that could be of use to the system. So to be able to save these parameters, the concepts of a parameter profile came to be. A parameters profile is essentially a list of any kind of parameters. Use case 3 - "Edit parameter profile", is the use case used to change the values of the of an already created parameter profile. It's important to note that it is the technician who initiates this use case, since know what exactly the values should be could require some deeper knowledge of the system.

3.2 Use case 11 - Calculate coverage path

There are a tonne of ways to navigate way points, first of one needs to decide on how to describe a way points. So for this system way points can be describe in two ways, either as a single point, or as a rectangle. For the rectangle the system should calculate a path that covers a the rectangle with lines that have a predefined distance between them. This use case should be initialized by the user, as a simple press on the user interface. In return the user interface should display the calculated path, so the user can tell if the

path is what they wanted.

3.3 Use case 12 - Run coverage path

With a calculated path, ei. list of way points to follow, the boat should be able to follow these points. When the user presses a button label "Run" Use case 12 - "Run coverage path" is initiated, and it should not finish until the boat has reached the last way point of the list of way points. The boat should get though the way points using a control loop. While the boat is running a the estimated time en-route should be displayed along with the current position of the boat.

3.4 Use case 5 - Request diagnostics

At any point in time, it might be convenient for the user or the technician, to know how the boat is doing. In other words, getting the diagnostics data from the boat. diagnostics data might include GPS information, what position the rudder is set to and so forth. This use case can, as mentioned be initiated by either the user or the technician, by the press of a button in the user interface.

4 Scope

The scope of this project has been analyzed with the use of the MoSCoW method. This method is used to prioritize what should be worked on in the project. The method is separated into 4 levels of priority; **Must**, **Should**, **Could**, and **Won't**.

The following priorities have been chosen for this project:

Must
- Navigate way points from user input
- Be compatible with NMEA protocol GPS input
- Use GPS for localization
- Implement a PID control loop

Should
- Control thrusters in two-thruster catamaran
- Use a graphical user interface for user interaction
- Be able to change the PID parameters

Could
- Control wheel in outboard motor on boat
- Be generic enough to use with other engine types

Won't
- Use polygon-coverage for a specified area
- Avoid obstacles

5 Method

In the following sections the methods used to design and implement the system are described.

5.1 ASE model

The ASE development model structure is shown in figure 4. This has been used to decide what and when to produce in the development process, and has been used in projects in earlier semesters.

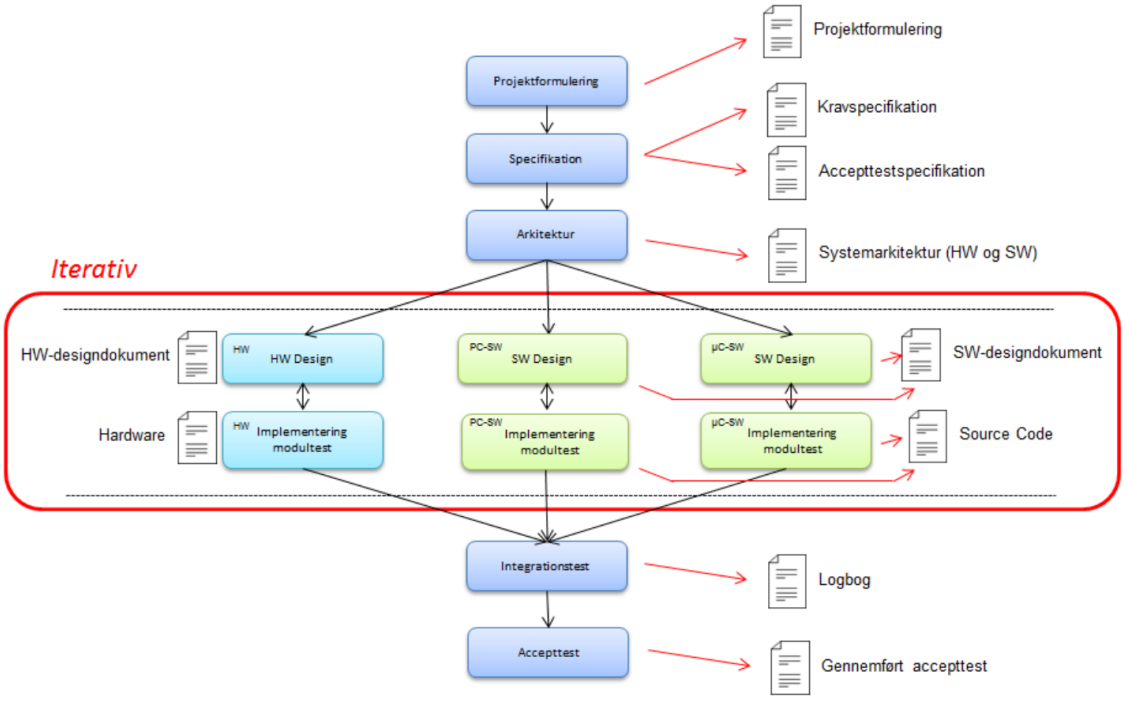

Figure 4: ASE development model[**ASE_model**]

The model is similar to the waterfall model, where all requirements are specified at the beginning of the project, architecture is developed, and the project enters the design phase. After iterating through design and verification in module tests until all design has been completed, the project undergoes several integration tests, and finally the acceptance test. The test section describes this process in detail.

5.2 V model

The V model in figure 5 is used for hardware as well as software development, and it describes each development phase and their corresponding test artifacts.

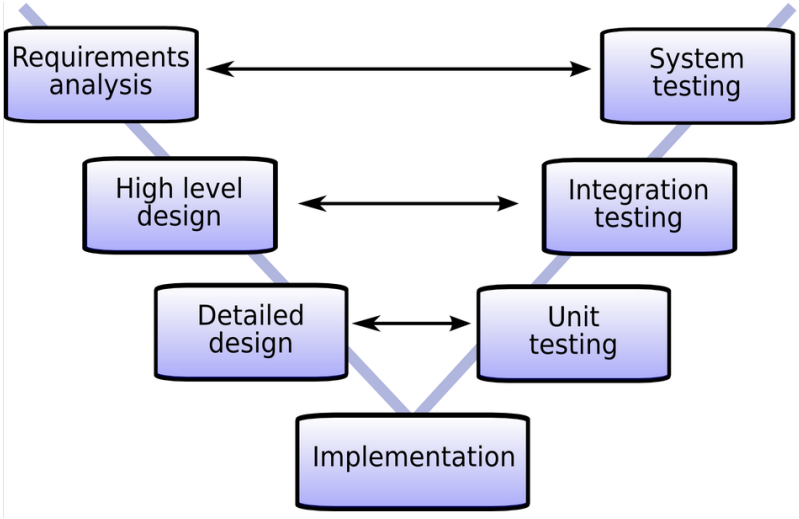

Figure 5: The V model

The model shows that an acceptance test is specified from the requirements specification, several integration tests are defined from the design, and that the functionality of each module in the system is verified with unit tests. This model is very effective in breaking down the project into several phases and getting an overview of which test phase corresponds to which development phase. The group made extensive use of this approach during development, more about tests can be found in the test section of this document.

5.3 SysML

In developing the architecture and design of the system, SysML has been used as in previous projects. SysML is a way of modeling systems with software and hardware components, where UML, another modeling language, primarily describes software systems. Figure 6 below gives an overview of the different diagram types in SysML used to describe functionality and design.

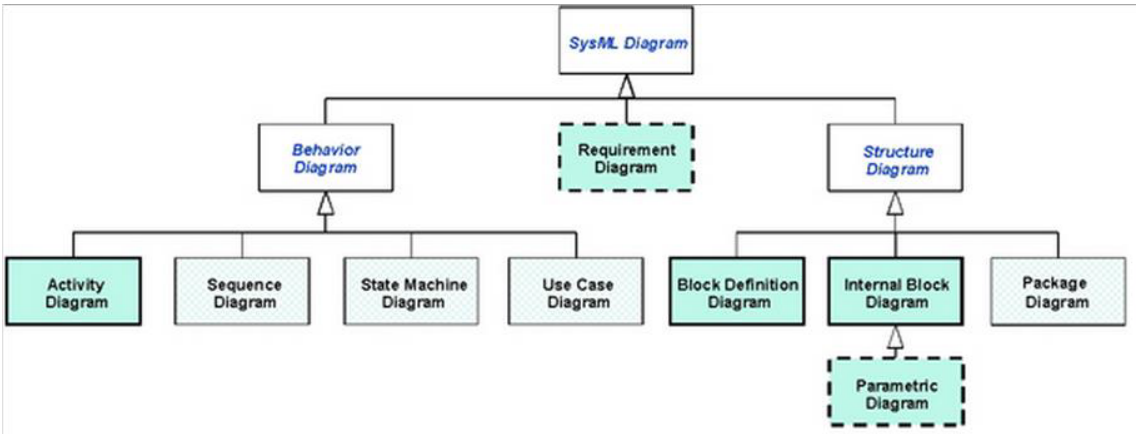

Figure 6: SysML overview[**SysML**]

SysML is widely used in the engineering industry, and enables other developers to quickly gain insight into the system without prior knowledge.

It is effective in bringing the system from idea to requirements specification, to architecture, and in bridging the gap between architecture and implementation, since SysML describes each module of the system and the logical relation between modules precisely. This also gives clues as to how to to implement the system, particularly in software, since class diagrams can be translated almost directly to internally consistent class hierarchies, if specified in enough detail.

The two structure diagrams used in this system is the Internal Block Diagram (IBD) and the Block Definition Diagram (BDD). The BDD is used to break down the system in its logical blocks, describe the relations between them, and the components of each block. The IBD is used to describe the relationship between the modules primarily in hardware. The IBD typically has a signal list that lists all the signals in the system, their types in the real world, and their values if applicable.

The behavioural diagrams used in this system are the Sequence Diagram (SD) and the Use Case diagram (UC). Use case diagrams describe the actors that interact with the system, and which use cases the system contains. For the software design, an Application Model and Domain Model have been used to describe the behaviour of the software and the overall structure. This is further detailed in the documentation, with class diagrams and function descriptions for each module.

The hardware has been designed from what's called a black-box approach, where the individual components in the IBD are treated as self-contained units. Each component is described in the hardware design section of the documentation.

6 Analysis

From the introduction, several components have to be chosen from a wide variety of viable components, both software and hardware.

The UI is a website made using Bootstrap and AngularJS. These were chosen because they are popular and intuitive libraries for implementing websites, and has a lot to offer in terms of design, styling and ease of use. An alternative was React, but the group felt like this would require more work, and was less documented than the others.

Setting up the map was done with Leaflet, or rather a version of Leaflet called Leaflet-directive. There were many map options, but since it was required that the system could work offline, this eliminated several candidates. With Leaflet-directive, the markers and lines were easy to set up and change, and the library had some built-in control schemes that lived up to the expectations of the group.

The website hosted on a NodeJS server, this was chosen because NodeJS is incredibly easy to set up and host locally for testing purposes, and requires very few packages installed to get working. These and other factors make it a very fast and responsive server solution. It is also written in JavaScript, which was a good fit when the group was working with JS on the client side as well.

The controller is a Raspberry Pi 3b, and this was chosen for a variety of reasons. The RPi 3b was one among 16 candidates that were considered for the system controller. A detailed analysis of this can be found in the hardware design section of the documentation, but ultimately the choice came down primarily to price and available documentation, and the latter would turn out to be relevant later on.

The GPS is a Ublox Neo 7-M. This unit offers reasonable performance for a non-RTK GPS, which is the only reasonable type to use in the system. Only bigger projects with a lot more funding would invest in an RTK GPS, whose price can easily exceed thousands of USD. The availability of this unit in the workshop contributed to its selection, but other similar GPS modules didn't offer considerably different performance, so the choice ultimately fell on the Ublox.

GeographicLib was used for geographic calculations, specifically for calculating rhumb lines and distances. Early in the development of the navigation unit, these functions were programmed manually, but it turned out that the algorithms offered by GeographicLib were much faster and more precise than our own. It was then decied to use GeographicLib, but exclusively for these two functions.

The MiniPID library was used to implement the Autopilot PID loops, since it is widely used when implementing PIDs in microcontrollers.

PiGpio was chosen for controlling the Raspberry Pi pin outputs. Initially it seemed like using Python scripts would be the way to control the outputs, but this C++ library was recommend by many developers, and contained the required functionality for the motor classes to access the corresponding hardware.

The NMEA protocol was used for communication between the GPS, Navigation, and

Autopilot components, as specified in the original problem description given by EIVA. Several write-ups of the original NMEA-0183 protocol were found, and the necessary command message structures were identified and implemented.

A C++ JSON library, nlohmann created by Niels Lohmann, was used to parse to and from the JSON format used in shared data files.

Boost test and FakeIt were chosen as the test frameworks, for more information on this, see the Test section of this document.

7 Architecture

In this section the hardware and software architecture will be discussed. The architecture is the ground work that enables the design of the system. Taking the requirements and dividing it in to blocks, modules, etc.

For a more in depth explanations of the different subjects discussed in this section, one can have a look at the System architecture, section 5, on page 40 in the documentation.

7.1 Hardware architecture

To get a better understanding of the components what are need for this system. A block definition diagram or BDD was devised, as seen on figure 7.

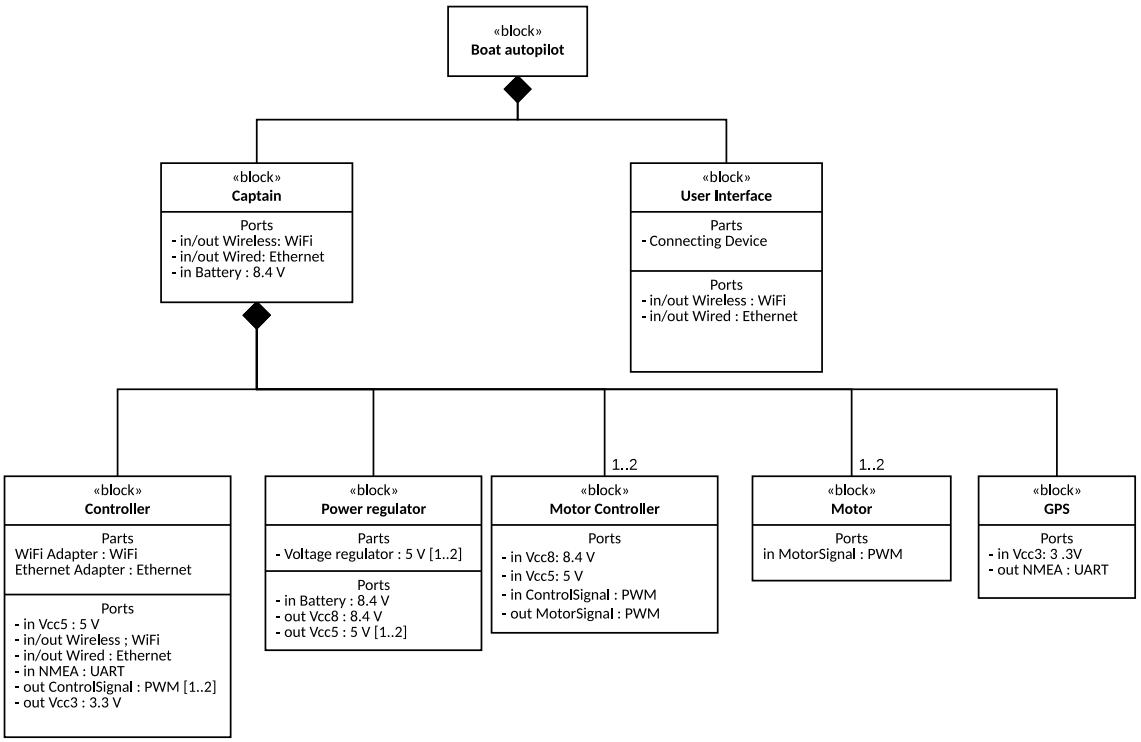

Figure 7: General block definition diagram (BDD)

The system can be broken down into two separate parts. One is the user interface, this is a clients personal computer. This personal computer needs to have either WiFi or an Ethernet port, this is so it can connect to the system. The system is the other part, and it has been named Captain.

Captain is the hardware platform for this project, and it also needs WiFi and an Ethernet port, so it can be connected to. Further more it needs an external battery to power it. Captain is also built up of subcomponents, or parts. these parts are; a GPS, 1 to 2

motors, 1 to 2 motor controllers, a power regulator, and a controller. The controller is the brain of the operation, it communicates with the user interface, and dictate what the motors should do, and it reads from the GPS receiver. The power regulator is used to regulate the battery voltage, so the controller, the motor, and the motor controller can use it. The motor controller is used to take the control signals from the controller, and drive the motor with them.

With the block now defined, an IBD or internal block diagram can be created, and seen in figure 8. This diagram describe how the different blocks of the BDD connect to each other, via the signals that are defined in the BDD as well. A full signal list and description can be found in the documentation in section 5.1.1.2 on page 42.

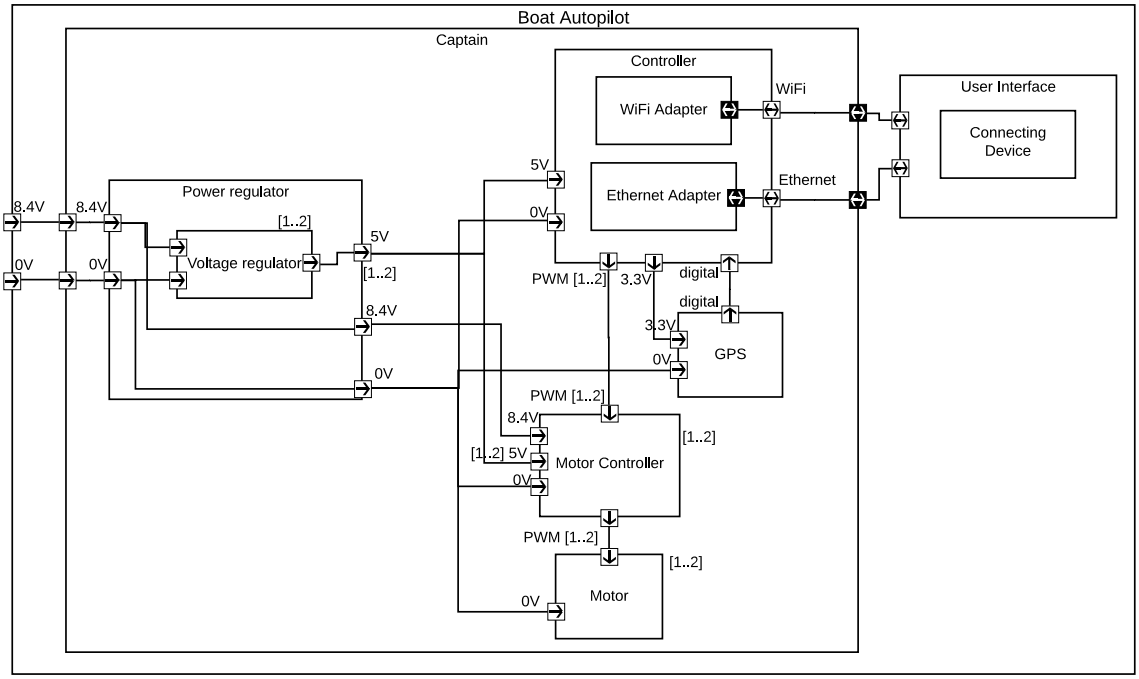

Figure 8: General internal block diagram (IBD)

7.2 Software architecture

The software architecture is described with a domain model, an application model and several sequence diagrams.

Lets start out by having a look at the domain model, it can be seen on figure 9. The domain model is used to describe the system should act to an actor interacting with it. In the domain model it can be seen how the user or technician can interact with the web interface, and how it then communicates to the controller. The controller acts on the motors, which in turn change the boats position. This that affects, what the GPS receiver reads, and this information is then passed on to the controller. The information

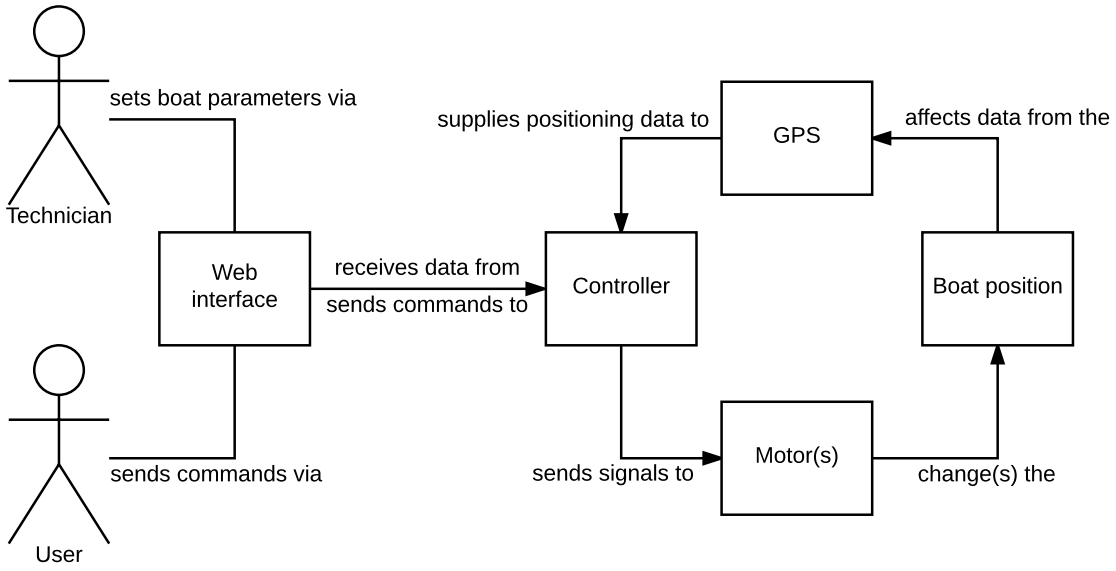

Figure 9: Domain model

With a domain model and use cases, an application model can be created, as seen in figure 10. It describes the functionality of the block from the domain model, it also classifies the blocks as either boundary, control or entity. A boundary block is something that interacts with the real world, a control block is the block that mediates functionality of boundary blocks and entity blocks. The entity block is a representation of information.

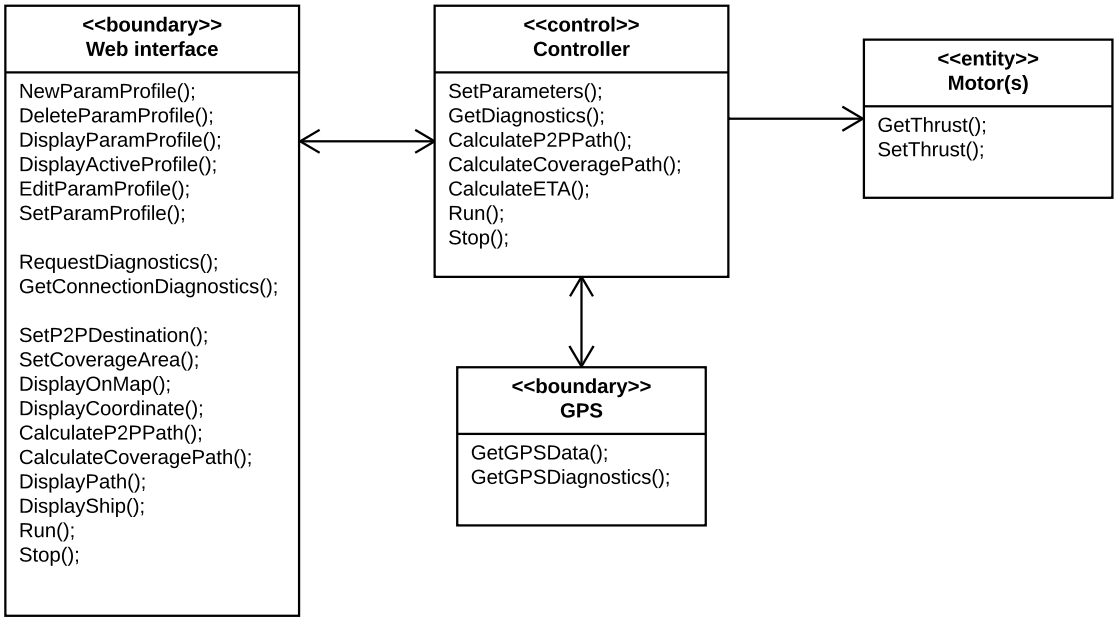

Figure 10: Application model

With the functionality of the blocks figured out in the application model, its time to look at sequence diagrams. There are a lot of sequence diagrams, so only a few interesting ones will be discussed in this report. For all the sequence diagrams, have a look in the documentation in section 5.2 on page 43.

The sequence diagrams follow the use cases, therefore lets have a look at the once that correspond to the once discussed in the requirements section, that is; use case 3 - "Edit parameter profile", use case 11 - "Calculate coverage path", use case 12 - "Run coverage path", and use case 5 - "Request diagnostics".

Figure 11 explains how a technician edits a parameter. First the parameter profile is displayed, then it is edited by the technician.

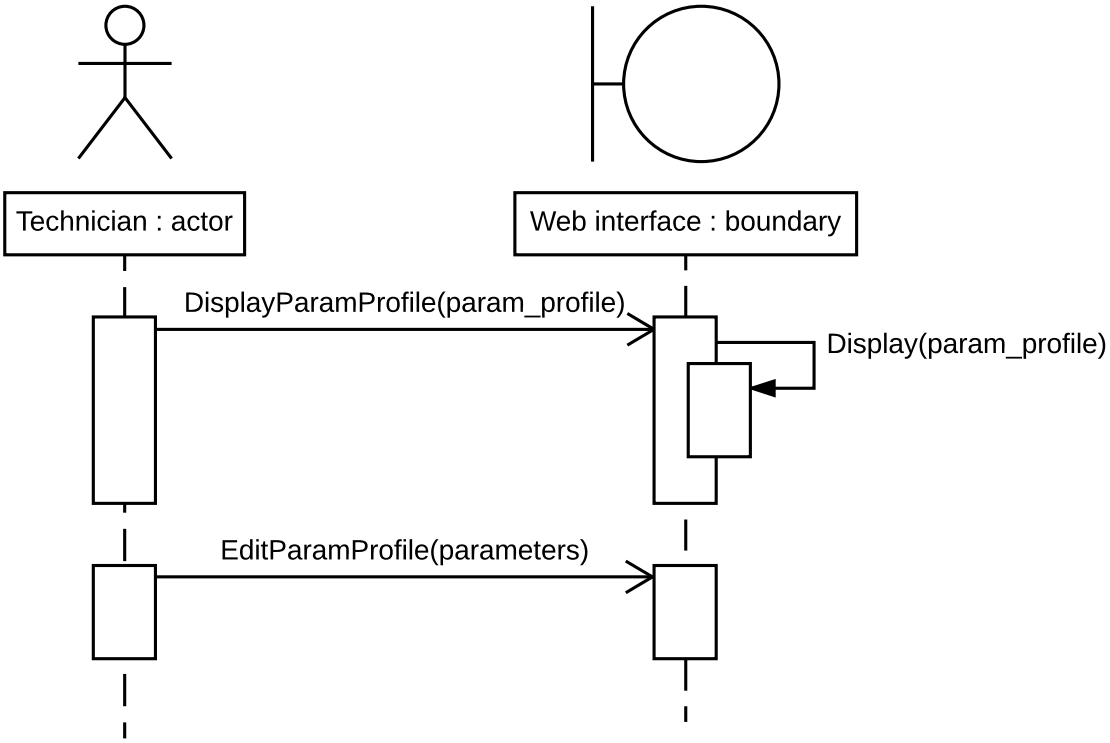

Figure 11: Sequence diagram for use case 3 - "Edit parameter profile"

Figure 12 is showing how a user tells the system how to calculate a coverage path. This is done thought he user interface, which tells the controller to calculate a path. The controller uses GPS data to calculate the path and then returns the path to web interface. Then the path is displayed.

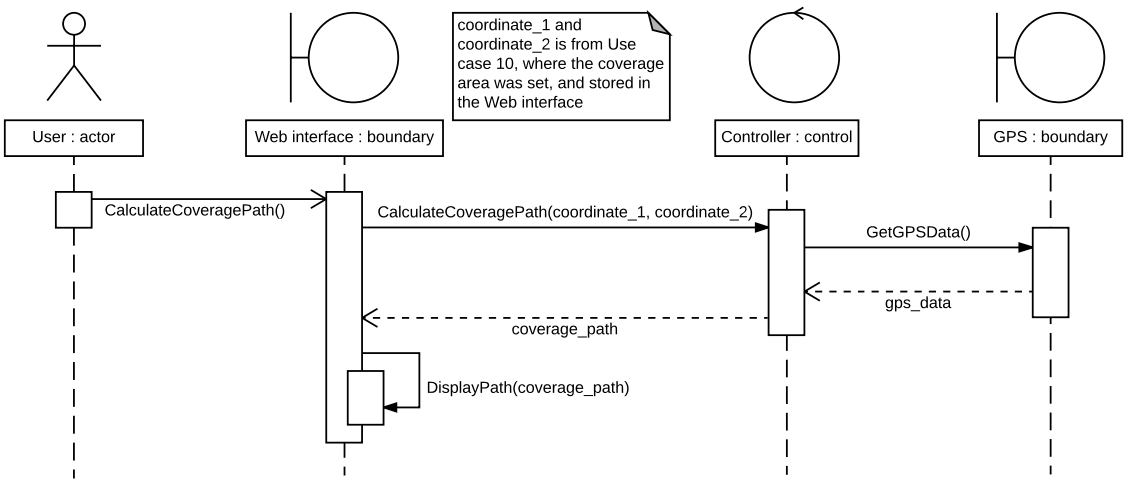

Figure 12: Sequence diagram for use case 11 - "Calculate coverage path"

Figure 13 illustrates what happens in use case 12. When the user tells the web interface to run it relays the message to the controller which in turn starts a loop. This loop gets GPS data and tells the motors what to do. It tells the web interface to display the boat position. The controller also calculates the estimated time en-route and the web interface displays it.

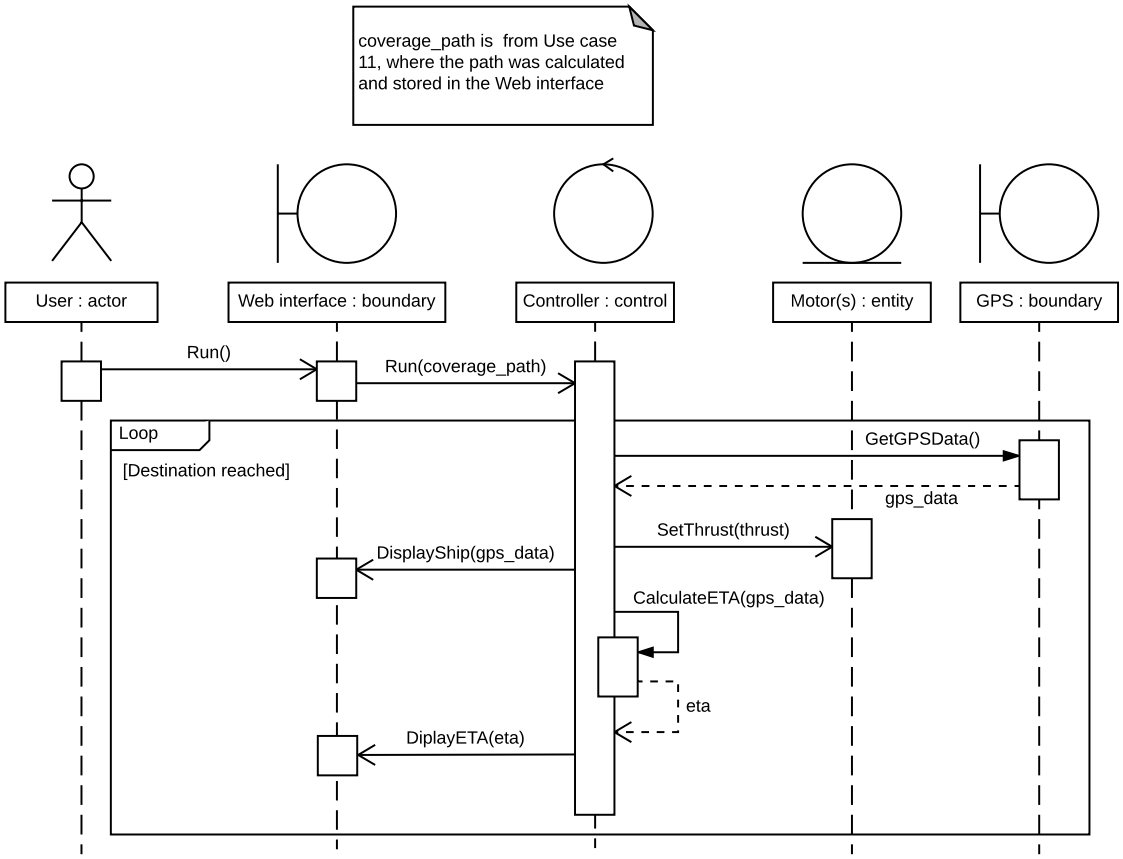

Figure 13: Sequence digram for use case 12 - "Run coverage path"

Lastly, figure 14 show how the system get and displays diagnostics data. When the

user requests diagnostics data, the web interface asks the controller for it. The controller responds by getting the current thrust of the motor and the GPS data along with the GPS diagnostics data. The controller also get connection diagnostics from the web interface, all of this is then displays on the web interface.

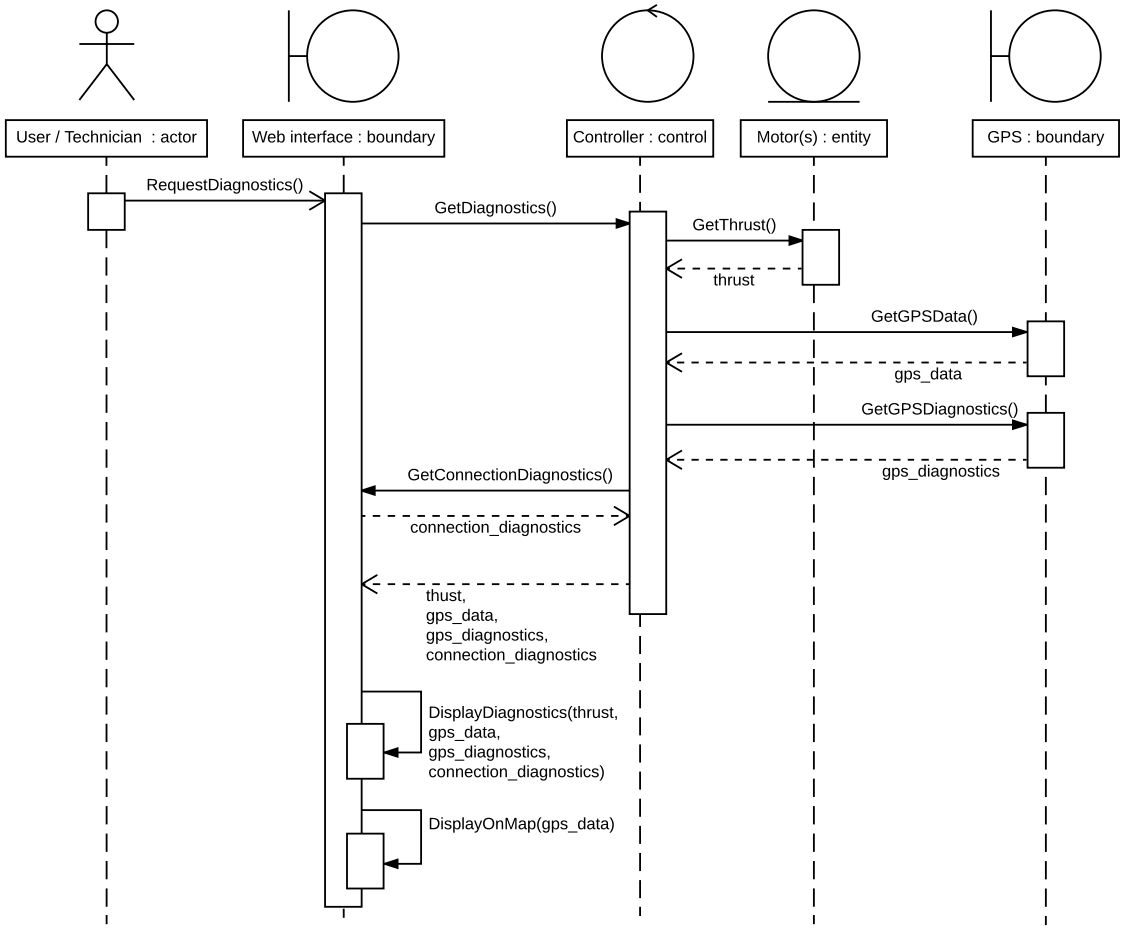

Figure 14: Sequence diagram for use case 5 - "Request diagnostics"

8 Design

8.1 Protocols

This section describe the protocols used in the system. Communication between the the controller, server and client takes place using several .json files, JavaScript Object Notation. Communication between the GPS, Navigation, and Autopilot modules in the controller uses the NMEA-0183 standard commonly used in marine navigation systems.

8.1.1 Communication between server and client

The system uses three .json files to send data between the controller, server, and client modules; fromNav.json, toNav.json, and activeParam.json. The latter contains the parameters the technician has specified for the system in the UI, and an example can be seen in listing 8.1.

```
{
        "name_": "Saint Princess",
        "parameter_names_": ["P", "I", "D", "Tool Width"],
        "parameters_": [10,5,1, 10],
        "creation_timestamp_": 1507201741743
}
```

Listing 8.1: Example of activeParam.JSON

This protocol could easily be extended to include more parameters that the technician could specify. A receiver class in the controller reads this file and sets the parameters in the autopilot.

Secondly, the toNav.json file contains the commands to the controller. An example is shown in listing 6.1

```
{
        "func_": "calcP2P",
        "target_position_": {
                "latitude_":56.187317092640775,
                "longitude_":10.18372267484665
        }
}
```

Listing 8.2: Example of a calcP2P call in the toNav.JSON

As seen, this command is to calculate a path to a certain coordinate given in latitude and longitude.

The fromNav.json file contains all the user-relevant information for what's happening in the system. This includes the current path to traverse, how much of this path has

already been completed, estimated time enroute (ETE), diagnostics data, such as connectivity data, motor status and telemetry, and timestamps in order to differentiate the file from previous versions.

Examples of these components of the fromNav.json file, and more examples of the toNav.json and activeParam.json files, can be found in the protocols section of the design, implementation, and test chapter in the documentation.

8.1.2 NMEA 0183 v2

The NMEA standard is a naval standard developed by the National Marine Electronics Association. The protocol is comprised of ASCII characters, and are transferred over a serial connection.

The messages used in the system are the GGA (GPS information) message from the GPS to the navigation unit, an APB (Autopilot message with direction to steer, cross track error, and other information) message from the navigation to the autopilot, and a VTG (Vector Track and speed over the Ground) message also from the navigation to the autopilot. By using these standard messages, it is possible to remove a component from the system and insert a replacement that also follows this protocol, making the system much more modular.

The contents of these messages can be seen in the protocols section of the design, implementation, and test chapter in the documentation.

8.2 Hardware design

Despite mainly being a software project, a considerable amount of hardware is required. The system requires a boat model, a controller unit, two motors, a battery, and a way to decouple the three modules from each other to avoid overcurrent and noise on the controller supply.

8.2.1 Boat

The boat was acquired at Navitas, one group having used it for a previous project already. The boat hull type is a deep v with a d-drive inboard motor configuration. A picture of the boat is seen below in figure 15

Figure 15: Saint Princess, a remote controlled yacht

8.2.2 System overview

Figure 16 shows the overall hardware design.

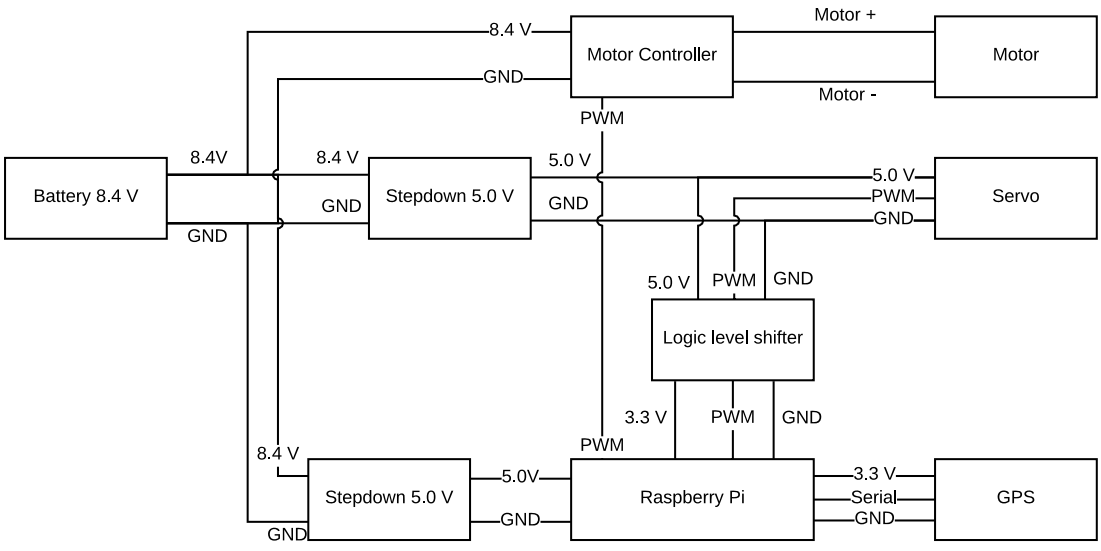

Figure 16: Hardware design

All components are supplied by a battery, which is stepped down for the controller and servo, and a logic level shifter is used between the controller and servo.

The controller chosen for the system is the Raspberry Pi 3b, and the GPS a Ublox Neo 7-M. The motor controller is a Cytron MD30C rev. 2, and the stepdown circuits used for

the servo and controller are LM2596's which were available at the engineering school's Embedded Stock. The logic level shifter was likewise found in Embedded Stock.

A detailed explanation of why these components were chosen can be found in the hardware design section of the design, implementation, and test chapter in the documentation.

8.3 Software design

The project contains code in many different languages, totaling over 30000 lines. For brevity, this section only the design of the navigation and autopilot classes and their dependencies is included here. The full design can be seen in the software design section of the design, implementation, and test chapter in the documentation.

Figure 17 shows the overview of the navigation class and its dependencies. It is seen that the navigation uses a number of helper classes.

Coordinate which contains a latitude and longitude, CoverageRectangle which contains two Coordinates which define a rectangle on the map, TargetPosition containing one Coordinate, an enumerator containing the names of the commands the navigation module can execute.

NavigationData contains information about the path to complete, the estimated time enroute, and how much of the path has already been traversed. Pose contains a Coordinate for the current position, and an orientation.

Along with the helper classes, Navigation has references to several interfaces: The Autopilot, GPS, Receiver, and Transmitter interfaces. The Navigation module's own interface is seen at the top of the figure.

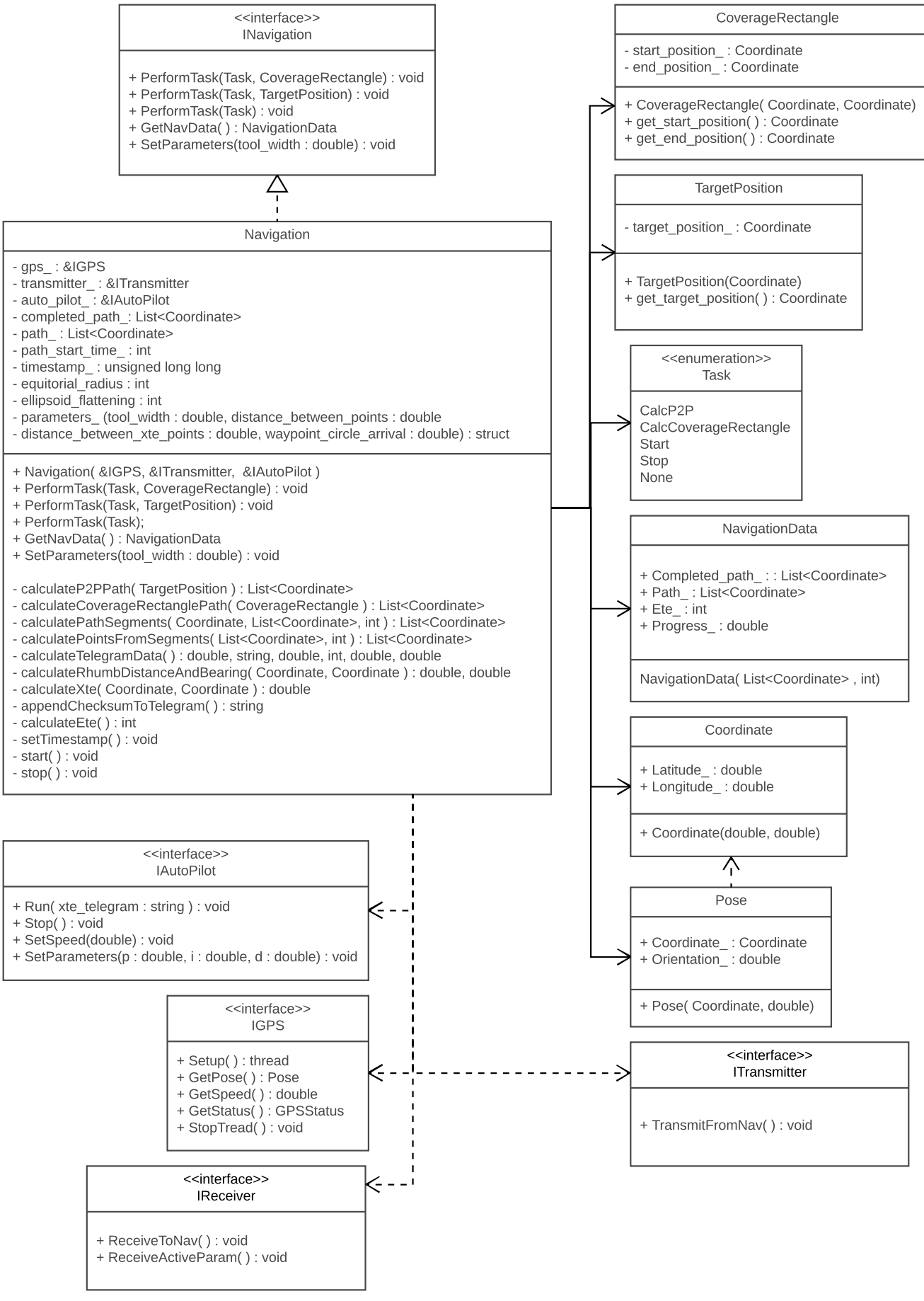

Figure 17: Navigation class diagram

Figure 18 shows the overview of the autopilot class and its dependencies. This module has an interface for the Navigation and Receiver to access, and has references to two interfaces; a rudder and a thruster.

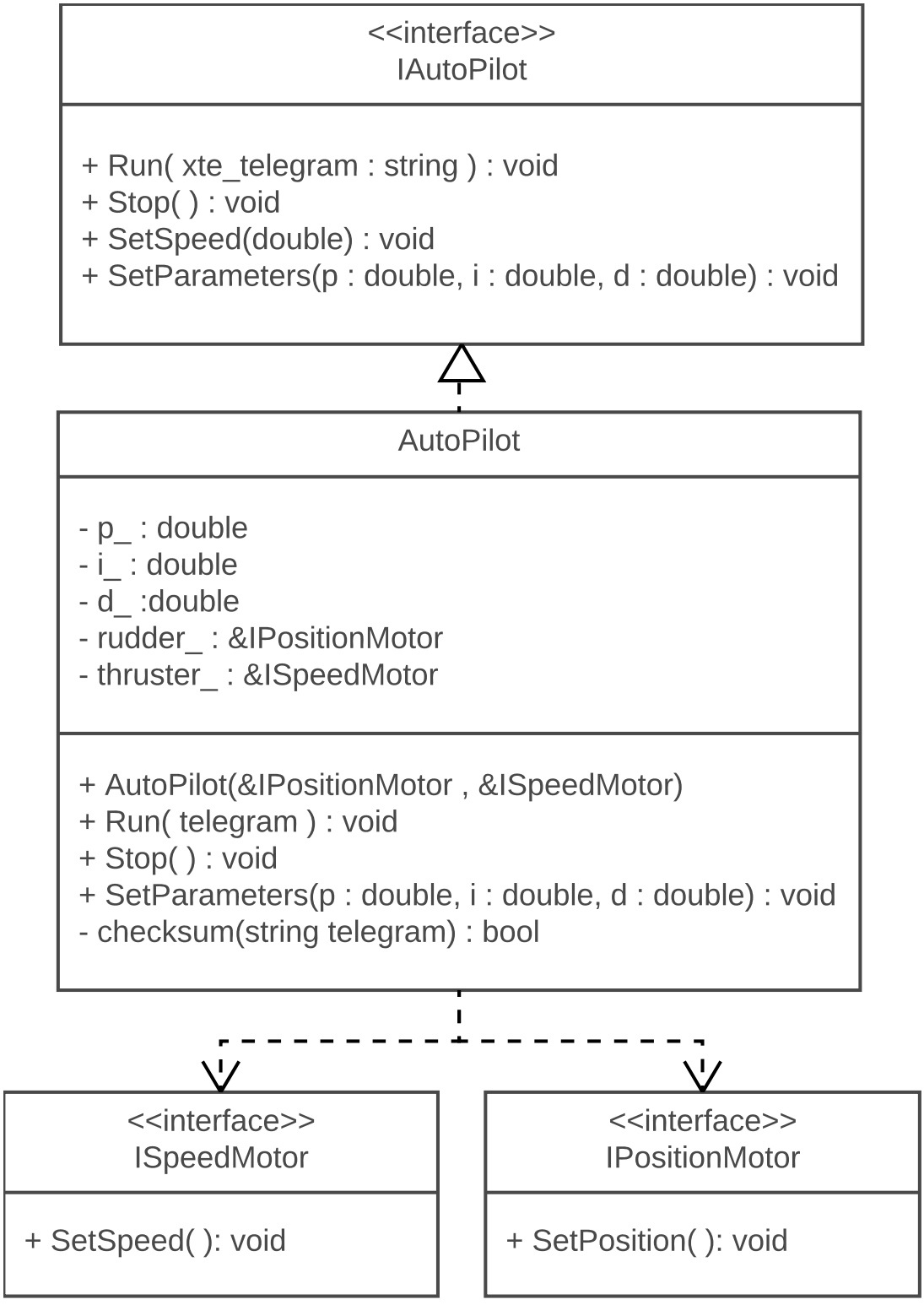

Figure 18: Autopilot class diagram

9 Implementation

10 Test

Testing was not initially something both team members had long experience with, but it soon became clear that verification would become an important part of the development process, particularly software.

One group member (Nicolai) had taken software testing the previous semester, and presented the general ideas and concepts before any controller code was written.

The team decided that unit testing would be mandatory for all C++ code that was possible to test, since it offered great benefits, and potentially a whole new way of thinking about software development. This also means that all classes that requires testing need an interface, to be able to make mock and fake classes.

After debating and attempting different testing frameworks, it was decided that the Boost unit testing framework and the FakeIt isolation framework would be used.

This combination proved to be very effective, but since both members of the team had no experience with either framework, there were a few issues, and a lot of learning, before the frameworks were used to their full potential.

At the end of development, over 170 tests totaling more than 8000 lines of code were written.

10.1 Unit tests

Listing 10.1 shows an example of a unit test of the Servo class.

```
1  BOOST_AUTO_TEST_CASE(Servo_test_SetPosition_50_returns_1500)
2  {
3          fakeit::Mock<IGPIO> gpioMock;
4          fakeit::Fake(Method(gpioMock, GpioServo));
5          fakeit::Fake(Method(gpioMock, GpioSetMode));
6
7          IGPIO & gpio = gpioMock.get();
8
9          Servo uut(gpio);
10
11         uut.SetPosition(50);
12
13         fakeit::Verify(Method(gpioMock, GpioServo).Matching([](auto gpio,
               auto pulsewidth)
14         {
15                 //Ignore the gpio, since it could change and is not
                       important
16                 return pulsewidth == 1500;
```

```
17          })).Exactly(2); //It should call it once in the constructor and
                once again when we set it.
18  }
```

Listing 10.1: Servo test of SetPosition

Unit testing has three phases, Arrange, Act, and Assert.

In the example, mocks and fakes are first set up, or Arranged, to isolate the test from its external dependencies; other classes and functions. This way, an error in a function will only lead to errors when testing that function, and not when testing other functions which depend on it.

The unit under test (uut) is then constructed, and a function is called; this is the Act step.

Finally, the Assert step verifies that the function did exactly what was intended, in this case calling another function with a specific argument matching a specific value. This is done using a lambda function.

Most unit tests in the project are much more complicated than this simple example, but the general structure and procedure is the same.

10.2 Integration tests

For some of the integration tests, the previous unit tests were repeated without the mocks and fakes, to verify that everything still behaves as desired when the real classes are constructed instead of fakes and mocks from the interfaces.

During this phase, a few design errors were found, including a dependency problem between the transmitter and navigation units. Without the unit test base, it is doubtful that this and other issues would have been resolved as quickly as they did, since they required some restructuring of those classes.

The integration tests were thus instrumental in sanitizing the code base and the dependency trees between classes.

10.3 Acceptance test

Finally, two special tests were made, verifying the entire code base including the UI. In these tests, the GPS inputs and motor outputs were mocked, but the rest of the classes and other elements were constructed. The UI was then started, and commands sent through the .json files. The first test calculates and traverses a point to point path, and shows the progress and finish in the UI. The second test calculates a coverage path and shows it in the UI.

These two special tests were useful when verifying the final product, and enabled the development team to easily build a main program to run the acceptance test, since the entire code base's functionality had been verified.

During the acceptance test, the boat was "piloted" by one of the team members (by carrying it), while the other sent commands using the UI on the website. The reason for

doing the acceptance test out of water was to be sure that nothing would go wrong with the waterproofing, and the system wouldn't be lost.

11 Results

This section presents the results of the acceptance test, and a more detailed write-up can be found in section 8 of the Documentation. Table 5 shows the overall results of the acceptance tests.

Use cases 1-4 deal with the creation, deletion, editing, and activition of user profiles in the UI, both main scenarios and an extension. The verification of the Edit parameters menu rested on the result of these tests. New profiles were created as detailed in the test specification, and the deletion function worked as described, even in the edge cases where an active profile was deleted, or the last profile in the list was deleted. Editing and activation worked as described as well, and thus all four tests passed.

Use case 5 have to do with the diagnostics window of the UI, and whether data is available for the motors, GPS, and connectivity, as well as a map showing the location and orientation of the boat. This use case passed as well, since all relevant data was updated at the expected interval in the expected format.

Use cases 6-9 cover the functionality of the point to point interface. The system must be able to set markers either by clicking on the map, or setting a latitude/longitude pair, which creates a marker in the specified position. It must also calculate, start traversing, and stop a point to point path. All these four use cases passed the tests.

Finally, use cases 10-13 describe the coverage rectangle interface. Like the previous four, coordinates must be put onto the map by clicking or typing them in, and a path must then be calculated, traversed, and stopped. All four of these use cases have strong similarity to the point to point use cases, and it was thus expected that the result would be the same for these similar tests. This assumption turned out to be true, and all four tests passed in a similar manner to the point to point tests.

Use case	Status
UC1 - New parameter profile	Passed
UC2 - Delete parameter profile	Passed
UC2 - Delete Parameter profile extension	Passed
UC3 - Edit parameter profile	Passed
UC3 - Edit parameter profile extension	Passed
UC4 - Set active parameter profil	Passed
UC5 - Request diagnostics	Passed
UC6 - Set point to point destination	Passed
UC6 - Set point to point destination alternate flow	Passed
UC7 - Calculate point to point path	Passed
UC8 - Run point to point path	Passed
UC9 - Stop point to point path	Passed
UC10 - Set coverage area	Passed
UC10 - Set coverage area alternate flow	Passed
UC7 - Calculate coverage path	Passed
UC8 - Run coverage path	Passed
UC9 - Stop coverage path	Passed

Table 5: Overview of acceptance test results

It was very satisfying for the development team to see the system pass all the acceptance tests, especially given that a lot of time was spent verifying the code base.

12 Discussion

This section is a discussion of the results from section 11.

All functionality of the system has been verified in the acceptance test. However, as mentioned earlier, the project scope was so big that many hard choices had to be made underway to achieve this result.

A depth sensor was never attached to the final product, since it would require a lot of time to evalute another hardware component to purchase, order, and integrate in the system. It was decided early that this module would not be the focus, but that the system, and specifically the protocols used, should be so generic and robust that additions like this one could easily be made by a future team. The development team felt that there was great value in having a system that could easily be expanded upon or changed, and much less value in introducing additional hardware when time was limited because of the many mandatory components that had to be developed.

In the same vein, polygon coverage was dismissed very early on due to its potentially immense complexity. Many projects taking considerably more time than ours and with much larger development teams are dedicated solely to developing efficient and robust polygon coverage algorithms. Developing an area coverage algorithm turned out to take considerable time by itself due to edge cases and making sure the algorithm wasn't too slow for the system to work, so rejecting polygon coverage from the outset seems to have been a very reasonable decision.

The question of whether to use files to hold the data exchanged between the controller, server, and website or finding another solution was debated in the group when coding the website early in development. It was decided that this simple solution would suffice since most of the system has no time-critical components (excluding the navigation algorithms themselves), since it relies upon user input, and a GPS which updates only once per second; a very long time relative to the speed of the processor.

It was also possible to develop a function to save a path object to a file, which could then be read at a later time to repeat a coverage rectangle, very useful for a survey team studying a certain region of the sea floor over time. This functionality was, similar to others dicussed in this section, deemed unnecessary for the final product, but would have been added had there been more time.

On the navigation side of things, it was possible to have used great circle lines instead of rhumb lines. The decision to use rhumb lines was made strictly because is gives the user much better coherence between what she sees in the UI, and the path the system actually calculates. Rhumb lines become straight lines on a mercator projection, which is what the map uses to transform coordinates from an oblate spheroid and a rectangular map. Besides, it's rare that a survey is conducted across great distances, and so the advantages of using great circle lines are negligible.

The development team is overall pleased with the final product, and with having learned a great deal about tests, component choices, software design, and taking respon-

sibility of the full development cycle as a small team. Making a wide variety of modules in several different languages all work together as intended was very satisfying, since it indicates that our decision to value generality and extendability over a full production-level implementation of the system (which we wouldn't have been able to finish in time) paid off.

13 Future work

There are a lot of things that could be better in the Captain project, and there are also some additions that would just make for a better product.

One thing that was put under won't in the MoSCoW is polygon coverage, this was simply do to the fact that polygon coverage is a very large subject. Usually when one wants to cover an are it is not a square. Even then generally its not square with the longitude and latitude lines, like the system is limited to at this time.

Another thing that would be a nice to have is a way to navigate a none straight point to point path, if maybe a user wants to cover a river, this would be very impractical with the current point to point system.

The idea of saving an already calculated path, so it can be run in the exact same way at a later time. This would make it possible to compare data collected on separate occasions and eliminate the position differences as a factor.

A thing that might not be optimal in the system at the moment is the communication between the website and the controller. It is handled by reading and writing to files. This was done for simplicity, but it might be a better idea to make the controller host a http server so it can handle GET and POST requests it self.

With the current implementation the controller only supports one boat type. But the software design is built up of rather generic interface so it should be possible in the future to include a variety of boat types.

One thing that we were intending to get done for the project was an echo sounding device that would measure the depth of the sea bed below the boat. But we didn't end up getting around to it.

The system could be classified as an mobile autonomous robot, but at this stage it does not have obstacle avoidance. This could be used to make sure that the boat is not going to run ashore, hit a bouy or even another boat.

If the system was ever to become anything other then a scale test bed, then it would both need a internet connection vis GSM. It would also probably need an RTK GPS receiver. An real time kinematic GPS receiver is sub centimeter precise and some can have interpolated position every 1/10 of a second, which could make the autopilot much more stable.

Lastly on the topic of sensors, it would probably be a good idea to have some sort of battery monitoring system, if the system is going to be battery powered.

14 Conclusion

The goal of the project was to develop an autonomous boat for surveying the sea floor. Commands were to be sent through a UI, and the system should then internally figure out how to execute the given task.

The components and libraries chosen for the project turned out to generally be well-suited to their purposes.

Bootstrap enabled the team to quickly set up a good-looking intuitive UI, and later integration with Leaflet and hosting with NodeJS turned a potentially daunting website and server design task into something much more manageable.

PiGpio and MiniPID saved a lot of time during the Autopilot development, and the function calls on the DCMotor and Servo classes, but some issues had to be solved in setting these up.

The JSON parser created by Niels Lohmann was effective in parsing the JSON format, despite some functionality that was not implemented by the author of the library.

GeographicLib made the navigation code slightly faster than it would have been with the team's own rhumb line algorithm, which is the only functionality that was imported from this library.

C++11 functions were used throughout the controller system, and the team members gained more insight about the new developments in the C++ standard this way.

The system is able to navigate between waypoints defined by which method the user chooses in the UI. The user can see the diagnostics in the corresponding page, and the technician or user can modify the parameters as well. The point to point page allows the user to quickly calculate and start traversing a route to the destination, as well as stop the system in case of emergency, or simply wishing to go elsewhere. Finally, the rectangle coverage functionality allows the user to travse a selected area on the map with a toolwidth chosen in the parameter menu. The system thus fulfills all requirements for the UI.

The NodeJS server hosts the website, and handles http requests from the website, reading or modifying the data files as needed. This is

The system is also compatible with, and uses, the NMEA standard GPS and Autopilot messages, and uses the GPS for localization. It has a PID loop controlling each motor, and the parameters for the rudder PID can be changed by the user in the UI. The system fulfills all requirements for the controller unit.

The logic level shifter and step-down circuits used to power the Raspberry Pi, DC motor, and Servo work as intended, and decouple the different components as desired. The high currents seen when testing the DC motor initially were managed properly with the driver circuit, and none of the other components were negatively impacted. Finally, the Raspberry Pi 3b provided the desired functionality at an affordable price, and had enough documentation (and available libraries) to make setup and verification of the

PWM outputs relatively painless.