

# The Eagle

Matthijs de Groot  
Nando Kartoredjo  
Arjan Langerak  
Dmitry Malarev



# The Eagle

by

Matthijs de Groot  
Nando Kartoredjo  
Arjan Langerak  
Dmitry Malarev

in partial fulfillment of the requirements for the degree of Bachelor of Science at the Delft University of  
Technology,

Client:	Willem Zwetsloot,	Seastate5
Coach:	Marco Zuniga,	TU Delft
Bachelor Project Coordinator:	O.W. Visser,	TU Delft
	H. Wang,	TU Delft

*This thesis is confidential and cannot be made public until July 3, 2017*

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.



# Preface

At the end of the Bachelor program in TU Delft, a Bachelor Project takes place. This project is designed with the goal of *executing a task in an independent team*. This task can be done in the form of an internship or by an assignment given by a client.

This report contains the process and end result of the task, assigned by SeaState5. SeaState5 is a company located in Delft, that has as a goal to improve the future generation of wind parks by providing low-cost and high reliability products. One of their products is the *Grasshopper*, a system that allows for a light weight method to lift and upend monopiles, and the *Woodpecker*, a system that allows for improved noise absorption when working offshore.

Our task was to create a software framework for an unmanned vessel for seabed scanning. Our contact for SeaState5 is W. Zwetsloot and our TU Delft Coach throughout the project's duration is M. Zuniga, both of whom we want to thank for the time and effort spent in assisting and guiding us with this project.

*Matthijs de Groot  
Nando Kartoredjo  
Arjan Langerak  
Dmitry Malarev  
Delft, May 2017*

# Summary

SeaState5 wants to facilitate the monitoring process for seabed cables of offshore wind farms. The company wants to achieve this by creating an unmanned vessel, named the Eagle, [REDACTED], which can be analyzed from a remote location.

As students from TU Delft, we have been tasked to create a software framework that allows the Eagle to go to a location by receiving instructions from the user, and to send environmental data back to shore via a long range connection. In addition, telemetry should be possible using short-range communication.

The software framework for the Eagle has been build with the design goals of modularity and stability in mind. [REDACTED]

[REDACTED] The user is capable of configuring the Eagle by adjusting the configuration file, stored on the SBC. This way, the Eagle pursues a modular design; sensor can be attached and detached without changing the software, only an alteration of the configuration file is required.

The Eagle will be capable of collecting environmental data, and storing it into a local database. The Eagle also contains in-memory data storage for each sensor.

Periodically, the Eagle will send data to the cloud using long range communication. Due to bandwidth limitations only crucial data will be sent to the cloud. The Eagle also provides telemetry, which allows all sensor data to be accessed by a nearby user. The Eagle is also capable of retrieving instructions from the cloud. Those instructions allow the eagle to perform specific tasks.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Structure . . . . .	1
<b>2</b>	<b>Research</b>	<b>2</b>
2.1	Problem Description . . . . .	2
2.2	Problem Analysis . . . . .	2
2.2.1	Accomplish tasks without user interaction . . . . .	2
2.2.2	Read and capture data from the environment . . . . .	3
2.2.3	Send and store the captured data . . . . .	3
2.2.4	Transfer the data from the Eagle to the client . . . . .	3
2.2.5	Software . . . . .	3
2.3	Requirements . . . . .	3
2.3.1	Vessel Control . . . . .	3
2.3.2	Collection of Sensor Data . . . . .	4
2.3.3	Transmission of Sensor Data . . . . .	4
2.4	Design Goals . . . . .	5
2.4.1	Modularity . . . . .	5
2.4.2	Stability . . . . .	5
<b>3</b>	<b>Implementation</b>	<b>6</b>
3.1	Framework . . . . .	6
3.2	Configuration . . . . .	7
3.3	CAN Driver . . . . .	7
3.4	Sensor Server . . . . .	8
3.4.1	CAN messages . . . . .	8
3.4.2	Instructions . . . . .	9
3.5	Control . . . . .	9
3.5.1	PathFinding . . . . .	9
3.5.2	InstructionPrinter . . . . .	9
3.6	Database . . . . .	9
3.7	Cloudnet . . . . .	10
3.7.1	Filter . . . . .	10
3.7.2	Embedded database connector . . . . .	10
3.7.3	Connection/Driver . . . . .	11
3.7.4	Cloud connector . . . . .	11
3.8	Telemetry . . . . .	11
3.8.1	Server . . . . .	11
3.8.2	Interface . . . . .	12
<b>4</b>	<b>Reflection</b>	<b>14</b>
4.1	Ethical Implications . . . . .	14
4.2	Recommendations . . . . .	14
4.2.1	CAN driver . . . . .	14
4.2.2	Sensor Server . . . . .	14
4.2.3	Cloudnet . . . . .	15
4.2.4	Telemetry . . . . .	15

---

<b>5 Conclusion</b>	<b>16</b>
<b>A Development Process</b>	<b>17</b>
A.1 GitLab . . . . .	17
A.2 Gitlab CI . . . . .	17
A.3 Google Test . . . . .	17
A.4 Valgrind . . . . .	17
A.5 Vera++ . . . . .	18
A.6 CppCheck. . . . .	18
A.7 SonarQube . . . . .	18
A.8 Trello . . . . .	18
A.9 Doxygen . . . . .	18
<b>B SIG</b>	<b>20</b>
B.1 Evaluation . . . . .	20
B.2 Improvements . . . . .	20
B.2.1 Code complexity . . . . .	20
B.2.2 Test code volume . . . . .	21
<b>C Project Description</b>	<b>22</b>
C.1 Project Description . . . . .	22
C.2 Company Description. . . . .	22
C.3 Auxiliary Information . . . . .	22
C.4 Info Sheet . . . . .	23
<b>D Field Test</b>	<b>24</b>
D.1 Test Plan . . . . .	24
D.2 Results . . . . .	31
<b>Bibliography</b>	<b>32</b>



# Introduction

To deliver the generated electricity back to shore, offshore wind farms need to be connected by large cables that are placed on the seabed. The positioning of these cables is crucial, for they can be easily damaged by rocks or a ship's anchor if the cables are laid incorrectly.

In the current state, placing the cables and monitoring them is expensive, requiring specialized ships and trained crew members to complete these tasks. In addition, if it turns out that the cables are placed improperly, the operation of relocating the cable is extremely expensive and time consuming, for the monitoring process will be once again needed after the deed.

SeaState5 wants to prevent the high cost of these processes by making the monitoring autonomous, [REDACTED]. By doing this, the cost of using a crew would be nullified, thus making the monitoring process more efficient.

## 1.1. Structure

This report will consist of four chapters. Chapter 2 will discuss the research part of this project. To determine the requirements for this project, first, the problems should be described and analyzed. After the problems have been analyzed, the design goals and the requirements could be determined. At chapter 3, the details of the implementation can be found. The chapter contains the description of the framework, and the description of each module in the framework. Each module will have a detailed description about its functionalities, and about the relations and functions of the internal modules, which make those functionalities possible. At the end, we will reflect on the delivered product in chapter 4. To support the company in the future, we also included recommendations. These recommendations will consider the implementation, but also the overall product. We will also reflect on the ethical implications of the delivered product.

In addition, there are four appendices. To make the development process go as smooth as possible, tools and methods were used. At appendix A, those used tools and methods have been listed. In between our process, we were required to evaluate the source code by SIG. After the results from the evaluation came in, we solved the issues SIG gave us. By solving the issues, we were able to increase our software quality. Everything about SIG can be found at appendix B. Appendix C includes the original project description, made by the company, SeaState5, and published at BEPsys. Appendix D contains the field tests with a scale model of the Eagle.

At the end of this report, the bibliography can be found.

# 2

## Research

This chapter describes the problem issued by our client, SeaState5. The problem will be analyzed to derivate the requirements for the project. To determine the course of the development, the design goals were set.

### 2.1. Problem Description

As described in chapter 1, offshore wind farms require large cables that run back to shore to deliver all the generated electricity to the consumers. Before a submarine cable can be placed, the ocean floor is inspected to find the most cost effective route to shore and to minimize the risk of disturbing wildlife. Because laying submarine cables is very expensive, it is very important to correctly position the cables. After the cables are deployed they are inspected to ensure that they are laid correctly and undamaged. [REDACTED]

[REDACTED]

In the next section, the problem is broken down and analyzed for requirements that can be used to fulfill this goal.

### 2.2. Problem Analysis

In order to achieve the goal, the Eagle must be able to do the following tasks:

- Accomplish tasks without user interaction.
- Read and capture data from the environment.
- Send and store the captured data.
- Act upon captured data.
- Transfer the data from the Eagle to the client.

These subgoals can be achieved by the following methods:

#### 2.2.1. Accomplish tasks without user interaction

[REDACTED]



### 2.2.2. Read and capture data from the environment

The vessel will have several sensors on board, all connected to a CAN bus, to perform submarine surveys. These sensors include a sonar for depth measurements, GPS for location data, temperature sensors, etc.

The Eagle should be able to be reconfigured to use a different set of sensors without requiring a change in the software. The software should be configurable to use any sensor that is available with minimal effort.

### 2.2.3. Send and store the captured data

The SBC will be connected to the CAN bus and should be able to log all the information that is available on the CAN bus. A database will be used to store information on external memory, e.g. a SD card.

The SBC will not insert all information into the database to avoid overloading the database. The information from the CAN bus will be available in the program at the full frequency for use in high frequency control algorithms.

The database should automatically adapt when sensors are added or removed from the Eagle.

### 2.2.4. Transfer the data from the Eagle to the client

The data should also be sent to a cloud-based storage solution to be used onshore. The data will be sent to the cloud using a satellite connection so there will be bandwidth limitations. To overcome the bandwidth limitations only a selection of the available data will be sent to the cloud. If the vessel is close-by all the data will be available over a high bandwidth connection such as Wi-Fi.

### 2.2.5. Software

The goal of this project is to develop a modular software framework for the SBC to provide the functionality as described above. The software framework should be developed in such a way that it can be used for any sensor configuration and easily adapted for different tasks using new control modules.

A scale model of the vessel, including the computers, the sensors, and other hardware is available to test the software framework in a controlled environment.

## 2.3. Requirements

In this section we will translate the functionality that we described in section 2.2 into a set of requirements for the software framework. The requirements can be divided into three categories: Vessel control, Collection of sensor data and the transmission of sensor data. The requirements are listed below and can be found in subsections 2.3.1 through 2.3.3.

- Vessel control
  1. Path Finding Algorithm.
  2. Communication with Hardware controller
- Collection of sensor data
  1. CAN bus communication driver.
  2. Retrieving the sensor data.
  3. Storing the sensor data.
- Transmission of sensor data
  1. Communication of data with a close-by user
  2. Communication of data with the cloud server
  3. Data filtering framework

### 2.3.1. Vessel Control

To autonomously accomplish tasks the vessel must be able to perform actions based on sensor data and the requested outcome. The Control module framework is a framework to easily create new control modules for different tasks. Control modules act upon instructions and sensor data to accomplish an instructed goal.



### **Communication with Hardware Controller**

The hardware controller has to execute the decisions of the control modules. To communicate with the hardware controller the SBC must be able to communicate with the hardware controller over the CAN bus. The control modules will create CAN messages which the CAN bus driver will transmit over the CAN bus.

### **2.3.2. Collection of Sensor Data**

To collect the sensor data, the software framework must be able to communicate with the CAN bus, decode the CAN frames and store the sensor data.

#### **CAN Bus Communication Driver**

The software framework must implement a CAN bus communication driver to be able to communicate with the sensors as well as with the hardware controller. The CAN bus communication driver will translate the CAN messages between the CAN frames used by the kernel and the structured objects used in the software framework.

#### **Retrieving the Sensor Data**

The sensors will broadcast their sensor data over the CAN bus. The software framework must retrieve the CAN frames from the CAN bus and decode the raw bytes of the CAN frames into a structured object for use in the software framework.

#### **Storing the Sensor Data**

The sensor data that is received will be stored into a in-memory database for use throughout the software framework. The in-memory database allows fast-access for other modules such as the telemetry module and the control modules. The sensor data will also be selectively stored in a lightweight persistent database to maintain historical data.

### **2.3.3. Transmission of Sensor Data**

The software framework must be able to transfer telemetry data over a high-bandwidth link such as Wi-Fi. The software framework must also be able to send data to the cloud using any available link.

#### **Communication of Data with a close-by user**

When the Eagle is close-by the user should be able to view as much information about the vessel as possible. When a high-bandwidth link is available, such as Wi-Fi, the user should be able to view all the data on the vessel. This includes data that is logged in the persistent database as well as high-frequency volatile data such as engine temperature.

The high-bandwidth link will be implemented over Wi-Fi because it provides high bandwidth and a relatively long range when used line-of-sight.

#### **Communication of Data with the cloud server**

When the Eagle is used offshore, it must be able to communicate with the cloud server at all times. The vessel must be able to obtain new instructions as well as report sensor data back to the cloud server. Since Wi-Fi is not always available, the framework needs to support long range communication of data. However, this suffers from low bandwidth and the communication must be adjusted accordingly. Only a limited selection of the data will be sent over the long range communication channel.

#### **Filtering Framework**

Which data that is sent to the database, which data is sent to the cloud and which data is available for telemetry should be configurable because they all have a different bandwidth.

## **2.4. Design Goals**

In this section, we will elaborate the design goals for this project. The design goals determine the general course of thought when implementing the final product, later discussed in chapter 3. The main design goals of this project will be modularity and stability.

### **2.4.1. Modularity**

The goal is to create a modular software platform that can be used to control the Eagle in order to complete the vessel's complex tasks. The reason for a modular platform is that it can be easily adapted to new capabilities of the vessel as well as a different sensor configuration.

### **2.4.2. Stability**

The vessel will be operating offshore, so a software failure will require a maintenance crew to navigate to the vessel in order to repair it, which is both time consuming and costly. The software platform must therefore be designed to be failsafe and be as stable as possible. To ensure the stability of the platform, it will be programmed according to the KISS (Keep It Simple, Stupid) principle. The individual parts of the software will be thoroughly tested using an automated test system. The complete software framework will be functionally tested using the available, on scale, prototype.

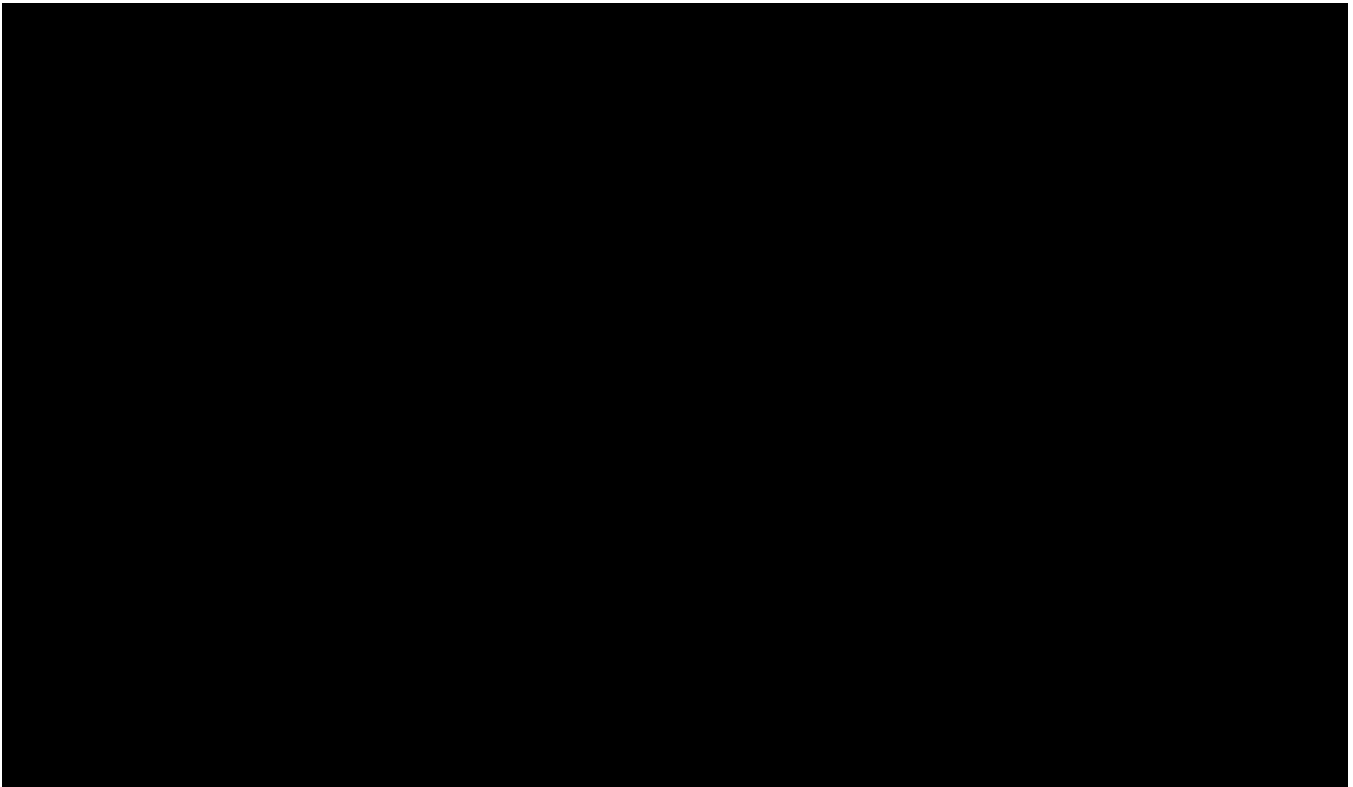
# 3

## Implementation

In this chapter, the final implementation of the Eagle's framework will be described as well as changes that were made compared to the initial implementation.

### 3.1. Framework

The framework of the Eagle is divided into modules to improve readability and maintainability of the software. However, the way the modules communicated was changed throughout the development process. The old overview can be found at figure 3.1. The new overview can be seen in figure 3.2.



When comparing the new overview with the old overview, the database module is no longer the central module, and is instead only used for the interaction with the cloud and the satellite communication. Now, the sensor server has replaced the database as the central module. The sensor server is the reincarnation of the older *sensor data logging* module. In addition to its predecessor, it contains buffers to store the CAN messages. More about the additional functionalities can be found at section 3.4.

The telemetry module corresponds with the *communication of data over Wi-Fi* (Shortened to *Wi-Fi*) module. The module was renamed for two reasons. The first reason is that the cloudnet module also uses Wi-Fi

within the project. Therefore it would become confusing which functionality the Wi-Fi module contains. The second reason is to stay consistent with the naming convention to name based on their functionality rather than the hardware they use. The telemetry module retrieves data directly from the sensor server instead of the database. This is done to reduce the time it takes to fetch the latest data, thus making the data available quicker. The control module also uses the sensor server instead of the database. The control module requires high frequency low latency access to the latest data thus making the in-memory data storage of the sensor server a better fit.

Due to high costs of satellite communication, the cloud communication was only implemented using a Wi-Fi connection. However, the satellite connection can be added as an additional module, without changing the existing modules. For reasons similar to the Wi-Fi module, the SatLink module was renamed to Cloudnet.

## 3.2. Configuration

One of the design goals for the software framework is modularity. In order to make the software platform modular, we tried to make all the modules as configurable as possible. For example, if a new sensor is added to the vessel it is added to the configuration file and does not required any change to the program. The configuration file is a JSON file that includes the configuration for every module, as well as every sensor. The configuration file is read at startup to determine which buffers to create, which tables to create, which CAN messages to expect, etc.

The software was designed to work dynamically based on the configuration. This creates some challenges because the configuration is not known at compile-time, thus requiring a multitude of configuration lookups throughout the program. We have implemented a separate configuration module that handles the parsing of the configuration file at startup, as well as provide classes with fast storage and lookup of configuration parameters that are required throughout the program.

The sensor server is one of the classes that is heavily dependent on the configuration, the configuration specifies which CAN messages to expect and how to decode them. Another class that relies heavily on the configuration is the database. The database creates tables for all the sensors and instructions that are specified in the configuration. Other uses of the configuration file include the file path of the database file, selecting which sensor data to send to the cloud storage and the interface to use the CAN messages.

## 3.3. CAN Driver

The communication with the CAN interface has been implemented in a separate module. The CAN driver is implemented using `SocketCAN`, which is a CAN network stack available in the Linux kernel. `SocketCAN` provides a way to communicate to a CAN bus over supported CAN transceivers using sockets. The CAN driver module provides several data structures to simplify the communication of CAN messages throughout the program and reduce the dependency on the underlying `SocketCAN` implementation. A CAN frame consists of several parts: an identifier, a field holding the amount of data bytes, and the actual data bytes. The identifier can be seen as the address of a CAN frame. The remaining two fields specify the number of data bytes, and the data bytes.

A CAN frame is converted to a `CANMessage` object consisting of a CAN identifier and a `CANData` object holding the byte array, which is zero-filled up to eight bytes. The `CANData` object has the functionality to interpret a set of bits within the byte array as one of the following data types: boolean, integer, unsigned integer, floating point, and double precision floating point.

The interpretation of the data in a `CANMessage` is done by the sensor server when it retrieves the `CANMessage` object from the `CANDriver` and converts it to a `BufferData` object. The `BufferData` object is inserted into the Sensor Server's buffers. The data bytes are decoded according to the parameters specified in the configuration file.

The `CANData` class also has methods to create a byte array from on a set of fields. These methods are used to construct a `CANMessage` from a `BufferData` object in order to transmit it using the CAN driver.

The CAN driver module provides methods to receive CAN frames from the CAN bus as well as send CAN frames to the CAN bus. However, it does not interpret or use the data. The CAN driver is deliberately kept as small as possible to separate the device-specific, platform-specific code from the rest of the program.

### 3.4. Sensor Server

The sensor server module is the central module of the program. The sensor server's task consists of storing sensor data, storing instructions and providing access to the stored data as well as transmit and receive data to and from the CAN interface.

#### 3.4.1. CAN messages

The sensor server's worker periodically retrieves new CAN messages from the CAN driver module. Every CAN frame that is received from the CAN driver module is decoded according to the configuration parameters that exist for every CAN ID. If a CAN frame is received with an unknown CAN ID it is discarded. The configuration of a CAN ID includes which fields exist within the eight byte byte-array of the CAN message and what data types the fields contain. A CAN message has been visualized in figure 3.3.



Figure 3.3: A CAN Message. Each block represents a byte.

The configuration file contains the information on how to decode a CAN message. The fields do not have to conform to their actual size, for example we can have a 10 bit unsigned integer. The fields can start on any bit within the byte array, so we can have a byte array with 2 booleans and 2 unsigned integers, one of 10 bits and one of 20 bits in the first 4 bytes of CAN message. A schematic representation is given at figure 3.4.

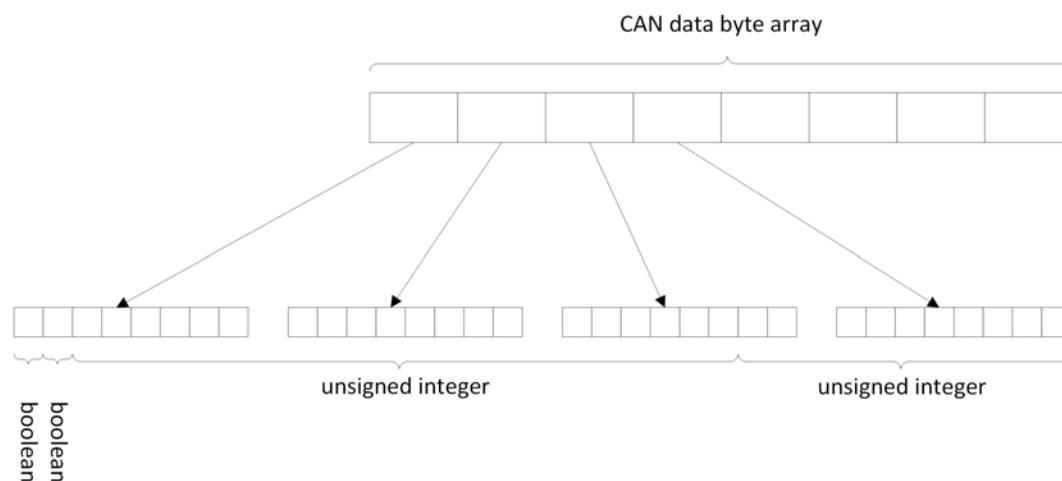


Figure 3.4: Data within a CAN message. Each block represents a byte. The first four bytes are split into their corresponding bits.

The sensor server has two types of sensor data buffers. One buffer is holding the last 40 CAN messages that are received, regardless of their CAN ID. The second type of buffer is a separate buffer for each CAN ID and holds the last 40 CAN messages for each CAN ID.

After a CAN message is parsed into a `BufferData` object it is inserted into the global buffer and into the corresponding separate buffer. The buffers are implemented using doubly linked lists so when the buffer deletes the oldest entry it does not require moving the remaining elements.

The sensor data buffers are used to supply the rest of the modules with the sensor data that was received over the CAN bus.

### 3.4.2. Instructions

Similarly to the sensor data buffers there is a global instruction buffer, and for every instruction type there is a separate buffer. The global instruction buffer is used to maintain the strict ordering of the instructions. The separate buffers are used by the control module to obtain the latest instruction of a specific instruction type.

The instructions are inserted into the buffers by the `DatabaseLogger`, which periodically searches the database for new instructions and inserts the new instructions into the buffers of the sensor server.

The instructions are kept for up to two hours, after which they are deemed outdated and are thus removed from the buffers. The messages are removed from both the global buffer and the separate buffers simultaneously. The instructions are available to the control algorithms without accessing the database, greatly reducing the load on the database as well as minimize latency.

## 3.5. Control

The control module is a framework that can be used to easily implement control algorithms. The functionality, such as the creation of a thread and starting and stopping the threads, are provided by the interface class `IControl`. To verify that the framework is functional and that it succeeds in creating an easy to use framework for creating control algorithms, we implemented a basic `PathFinding` control algorithm.

### 3.5.1. PathFinding

The `PathFinding` algorithm fetches the `goto` instructions from the sensor server to fill a list of set points to which the vessel must travel. The `PathFinding` algorithm calculates the correct heading and speed for the designated path and sends the heading and speed to [REDACTED] will adjust the heading and speed of the vessel accordingly.

### 3.5.2. InstructionPrinter

To test the communication of instructions using the cloud server we implemented the `InstructionPrinter`. The `InstructionPrinter` was used to visualize the received instructions by periodically checking the current instructions in the sensor server's buffer and logging them to `stdout`. While this is obviously not a control algorithm, the `IControl` framework made this debugging feature very easy to implement and thus proves that the framework succeeds in its goal to provide an easy-to-use framework for implementing control algorithms.

## 3.6. Database

The database module is used to log data to persistent storage. [REDACTED]

[REDACTED] The database module provides the program with all the methods necessary to use and access the database.

Every sensor that is configured to be logged to persistent storage is inserted into the database by the `DatabaseLogger`. The interval at which sensor data is logged to the database can be configured in the configuration file.

The `Cloudnet` module requests new instructions from the cloud server and inserts the new instructions into the database. The `DatabaseLogger` will periodically read the instruction tables to determine if there are new instructions. If there are new instructions the `DatabaseLogger` will insert the instructions into the `Sensor Server's` buffers.

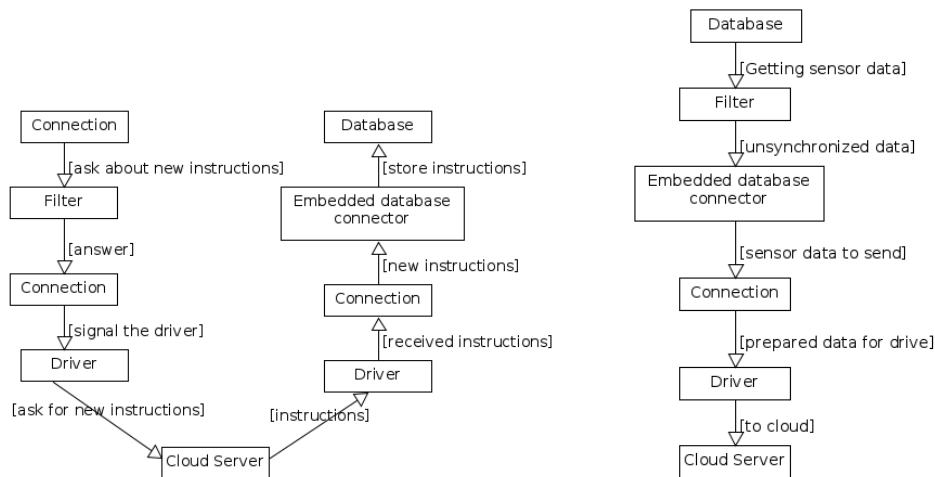
The location of the database file is configurable in the configuration file, it is therefore possible to put the database on external storage such as an SD card. The database is automatically created when it does not exist so the SD card can be safely removed. When the SD card is removed the database file can be examined and

the data can be analyzed using programs such as Matlab [12].

The configuration of the database is completely dynamic, the sensors and instructions that are in use by the system are read from the configuration file and the tables are created accordingly. The database automatically creates tables that do not exist. A new sensor can be added to the configuration file and the database will automatically create the tables to accommodate the new sensor without requiring manual intervention.

### 3.7. Cloudnet

The Cloudnet module is responsible for the communication between the external cloud database and the embedded database. The module is split up into different submodules in order to keep a clear separation between the different functionalities. This makes sure that the cloudnet module modular. It also makes it possible to test each functionality independently from each other. The submodules are the filter, the embedded database connector, the connection/driver and the cloud connector.



(a) Diagram of the interaction between modules for getting instructions from the cloud into the embedded database.

(b) Diagram the interaction between modules for sending sensor data from the embedded database into the cloud.

Figures 3.5a and 3.5b show how the different submodules interact with each other. A description of each submodule is given in the subsequent subsections.

#### 3.7.1. Filter

The filter keeps track of which values of the embedded database are synchronized with the cloud server. It keeps track of the time when the values were either put into the database or when the values were send to the cloud server. This information can then be used by other submodules to figure out what data still needs to be synchronized.

#### 3.7.2. Embedded database connector

The embedded database connector is responsible for getting the sensor data out of the database that needs to be send to the cloud server. However, not everything that is stored in the database is needed in the cloud server. Therefore, this submodule can be configured to select only a specific set of sensors. This can be specified further by configuring what kind of data from those sensors needs to be send. There is also a configuration for specifying how much time there at least need to be between subsequent sensor data. If this time is too short, then this data is not retrieved from the database.

The second responsibility of this submodule is to put instructions into the embedded database. This submodule inserts the data of the instruction into its own table. It also inserts a reference in a global instruction table which can be used to find the new instructions and their corresponding values.



### 3.7.3. Connection/Driver

This submodule consists of a Connection and a Driver module. Those work together in order send and receive data from the cloud server. This is split up in order to adhere to a modular framework. The controlling of the transmission hardware and the logic behind sending and receiving data is therefore split up into these two modules.

The Driver module is responsible for controlling the transmission hardware. This module can communicate to the cloud server with a HTTP REST service. Multiple implementations of the Driver module can exist where each implementation corresponds to different transmission hardware. This module should have an implementation for satellite connection. However, satellite hardware is too expensive for the prototype. Instead, only a driver which uses Wi-Fi is implemented. [REDACTED]

[REDACTED] receiving data from the cloud server. It contains procedures to make sure that data from the embedded database can be synchronized with the cloud server. It can for instance create new tables in the cloud server. Or it can ask the cloud server for instructions that are not yet synchronized. These procedures also contains methods to be able to recover from situations such as sending too much data to the cloud server and with temporarily network failures.

### 3.7.4. Cloud connector

The cloud connector is responsible for automatically switching between Wi-Fi and satellite connection. The design is that it can test which connection is the most reliable and then use that connection. However, since the prototype only has Wi-Fi, this module could not be fully implemented. But it is left in because when satellite hardware is added, this module can then be used to switch between Wi-Fi and the satellite hardware.

## 3.8. Telemetry

[REDACTED].

The telemetry server allows the client to access all sensor data provided by the sensor server. It is up to the client to decide which table it would like to access. The server is only capable of responding with the most recently added entry in each buffer. The client is thus not able to retrieve older data. This choice is made because losing telemetry data, for example due to a slow connection, is not considered a critical failure. The telemetry data functions as information for the user to interpret and evaluate. The user should still be capable of interpreting and evaluating the data even if some entries are missing.

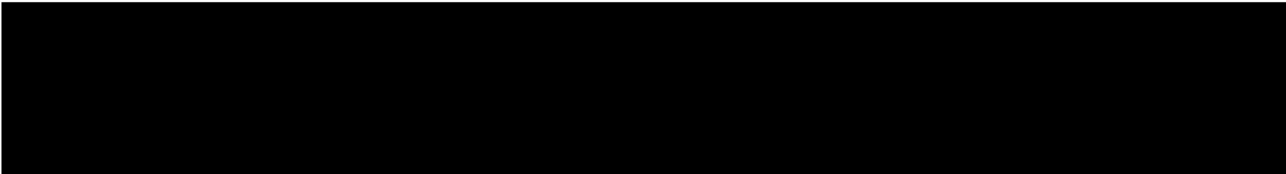
The telemetry does, in comparison with the old implementation, not make use of the filter network as seen in figure 3.1.

The client side of the telemetry module has been considered out of scope for this project. It is up to the company to develop one should it be required.

### 3.8.1. Server

[REDACTED]

The server is able to respond to requests with the most recently added entries in each sensor server buffer. Which entry from the buffer the server will send depends on the client's request. The client can request the most recent entry from a buffer (also known as table, to stay consistent with the database terminology) by providing the table name in the HTTP URI path. The server will check if the table exists, and if so, will respond with the most recent entry. The client is also able to request specific columns from those entries. Those columns can be requested by adding the column names in the URI query, in addition with the required query parameter. If the client does not state any columns, the server will respond with all columns from the [REDACTED]



If the short path is used without appending a query, the server will respond with the configuration file, allowing the client to synchronize the configuration settings between the client and the Eagle.

If a wrong URI is used to access the server, it will respond with a custom HTML error response. Each HTML error page will contain an error code, with the detailed explanation of the error code, and a hint allowing the client to fix its mistake. When a correct URI is used, the server will respond with a JSON data structure containing the requested data. The response will consist out of one JSON object, each containing:

-   
  
  


The complete decision tree for the telemetry server can be found at figure 3.6.

Internally, the telemetry server makes use of one request factory, and multiple request handlers. The factory determines which response should be send back. This is done by parsing the URI, and checking the elements within it. If a condition, stated in figure 3.6, is met, it will let the corresponding request handler generate the response. Thus, there exist a handler for the long path, the short path, and for each HTTP status. The factory does not need to access the sensor server buffers, and from all the request handlers, only the long path request handler needs access to it. This model follows the designs of separate responsibility, and modularity.

### 3.8.2. Interface

To access the sensor server, an interface has been implemented between the sensor server and the telemetry module. This decision has been made to follow the desired design goals of modularity, but also for practical reasons. Functions within the interface will retrieve the data from the sensor server, and will parse it to the desired result for the telemetry. The interface allows the telemetry module to interact with the sensor server, without accessing the sensor server itself. As long as the interface has been initialized alongside the sensor server, the telemetry will be able to use functions within the interface.

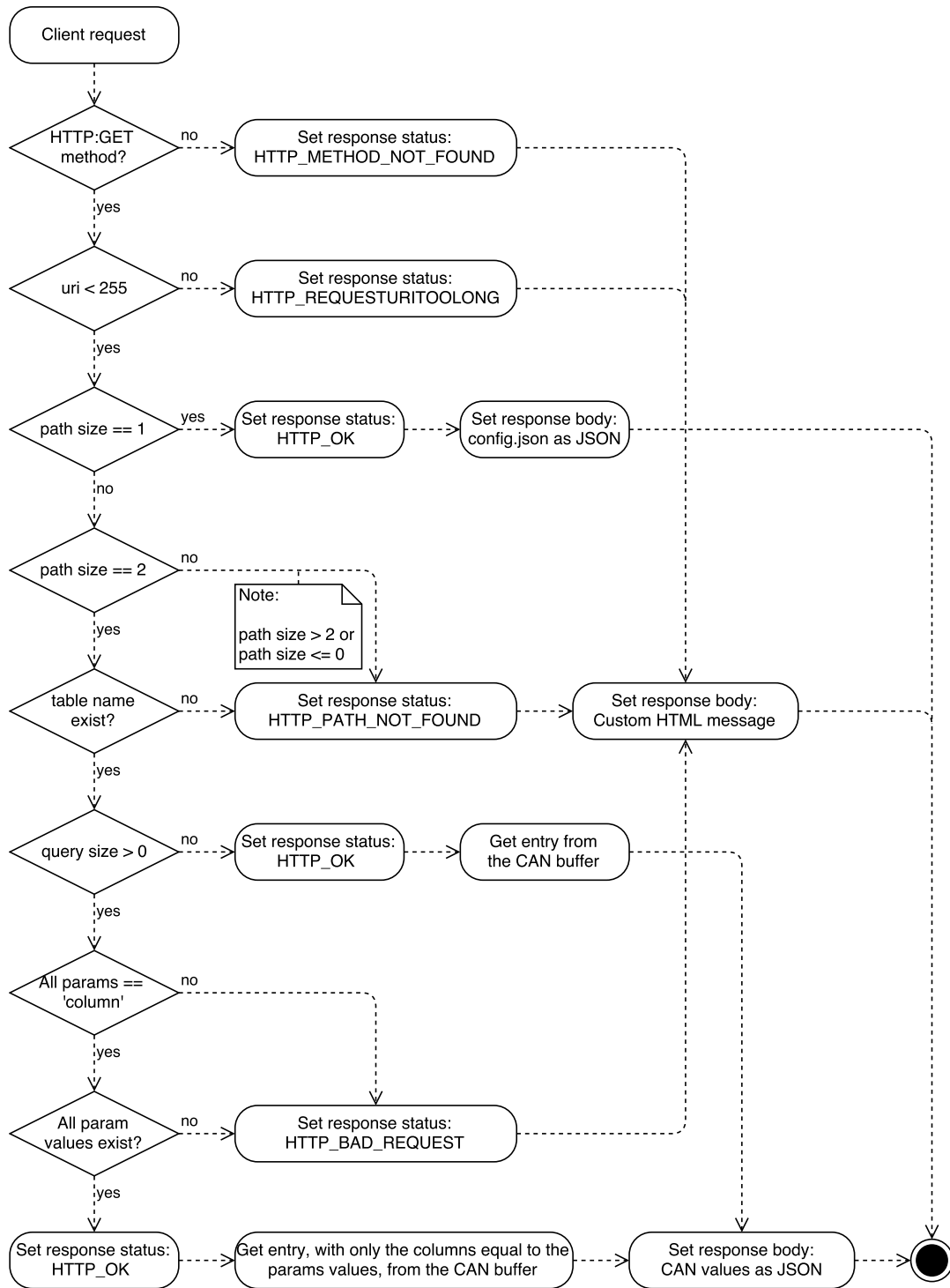


Figure 3.6: The decision tree for the HTTP telemetry server. The end state corresponds with the server sending the response back to the client.

# 4

## Reflection

In this chapter, we will reflect on the final product. First, we will discuss the ethical implications, as a result of our product. Second, we will make recommendations based on our experience with the development, to support the company in future developments.

### 4.1. Ethical Implications

The intent was to build a very modular and extensible platform that can be used for any sensor and any use case by changing the configuration file and implementing new control algorithms. The extensibility of the platform also means that it can be used for different purposes than it was originally designed for. While it is possible that the platform is used for nefarious purposes, it is not intended to be used for such causes. Because the purpose of the vessel is mostly determined by the control algorithms and not influenced by the capabilities of the framework we conclude that it is up to programmer of control algorithms to determine whether the control algorithm provides functionality that is usable for either good or evil.

### 4.2. Recommendations

To support the company, SeaState5, with future projects and developments with the Eagle, we would like to add some recommendations based on our experience with the development of the product. Those recommendations will contain possible improvements considering the current implementation for our part of the product, and suggestions to improve the overall product.

#### 4.2.1. CAN driver

The CAN driver module is an essential part of the program, it provides the program with the functionality to communicate over the CAN bus. In the future, it might seem more feasible to communicate over different technologies, such as PROFIBUS [13] or Modbus [6]. The CAN driver module should be rewritten to implement a common Bus Driver class to make the implementation of bus drivers in the future easier.

The CAN ID is often used as the unique id identifying a sensor or instruction type. To make the framework more universal every sensor should be given an unique ID that is not dependent on the technology that is used.

#### 4.2.2. Sensor Server

During the development process the design of the modules changed, where the database as a central module was replaced by the sensor server due to performance requirements. The in-memory approach of the Sensor Server's buffers was preferred over the on-disk approach of the database. This choice also meant that not all data had to be stored in the database, but only a fraction of the sensor data had to be written to the database.

The functionality that was added to the Sensor Server to implement this change in design made the Sensor Server a very large and complex module. Several parts of the Sensor Server can be viewed as autonomous components, and as a result, those parts could be removed from the Sensor Server class and instead moved to their own class. Splitting up the sensor server would contribute to a module that is easier to extend, test and maintain.

### 4.2.3. Cloudnet

The cloudnet module was initially designed for using satellite communication by an unknown software method because these methods differ by satellite hardware. Because of this uncertainty, a special object named DataTable was made to hold information received from the hardware. Then, the driver implementation of this hardware only needed to convert from its own data type to a DataTable. However, since it is currently designed to use with a HTTP cloud server this object could be replaced by a JSON object. This will reduce some code in both cloudnet but also the database module since that needed to construct a DataTable for the cloudnet module. This will also create some more consistency because all the other modules use JSON objects for communication with the embedded database. The framework already includes a specific JSON parser named "JSON for modern C++".

Another recommendation is improve the cloudconnector. At the moment, it is designed to be able to switch between Wi-Fi and satellite communication automatically. But this submodule cloud connector is missing this functionality. Also it is very complex to keep track of which connection is online, which one is stable etc. A better way would be to switch from a satellite connection to Wi-Fi by using a configuration option. This way you could still test with Wi-Fi but also use satellite communication without having the recompile the code.

The final recommendation is to take advantage of the fact that sending/retrieving from the cloud server and sending/retrieving from the embedded database is done in separate modules. Those modules could be run in different threads so that data could be pre-fetched from the embedded database in one thread, and sending to the cloud server in another thread. This can reduce the time of getting the data into the cloud server in case the satellite connection is unstable.

### 4.2.4. Telemetry

It is recommended for the company to develop their own client to interact with the HTTP telemetry server. As stated in section 3.8, the development for the client has not been included within the scope of this project. To support the company with the development, there are two decisions made within the project. First, the server requires the HTTP protocol. Because of the wide use of the protocol, the company would be able to develop their own client by using a variety of languages. Thus, it is recommended that the company develops a client using a language which is best known within the company, allowing easy and fast development. Second, there is a client example to support this development. It is also recommended to create a graphical user interface for the client. This would improve usability for those who are unfamiliar with the program, or using command lines in general.

It is recommended to recreate the custom HTTP error messages for the HTTP telemetry server, as the current implementation is rather basic. This allows the client to read the information, and by reading it, to recover from its mistake. From a designing standpoint, changing the current error messages by using the corporate identity allows the client to recognize the error message as part of a SeaState5 product line.

Considering the current implementation.

The telemetry server only uses a small portion of the .NET framework. Additional libraries increases the complete size and complexity of the program, which as consequence increases the building time substantially. It is feasible to develop a HTTP server for the telemetry server without any dependency from external libraries. In addition, removing the dependency provides complete control over the HTTP server implementation. The complete control allows flexibility when it comes to interacting with the other modules, failures to be fixed without waiting for support from a third party, etc.

# 5

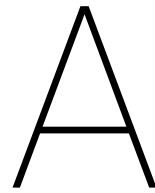
## Conclusion

SeaState5 wanted a modular software framework for the Eagle so that new functionalities, as well as hardware changes, are possible without major changes to the software. There is no existing software that meets all the requirements of SeaState5 because the Eagle is newly designed product. The goal of the project was to develop a modular software framework for the Eagle that can be extended and adapted by SeaState5 in the future.

The project was executed within the team of SeaState5. The whole process of development, including working with the other members of SeaState5, turned out to be very educational. SeaState5 has stated clear requirements for the software framework, however, the implementation for the framework was discussable during the development. This approach is different from what we have encountered so far and lead to a new vision on responsibility for the software that we create.

During the project, a fully functional software framework was developed for the Eagle. The software framework was successfully tested (See appendix D) multiple times using the available, on scale, prototype of the vessel. The tests proved that the software framework provided all the required functionalities (See section 2.3). The software framework is extendable to provide the vessel with new capabilities, as well as improve existing capabilities.

The successful tests demonstrate that the software framework provides the functionalities that are necessary for the vessel to operate, as well as providing a possibility to easily extend the functionality by adding new control modules. The project was deemed a success by SeaState5 and intends to use the framework for the Eagle in the future. The project was also deemed successful by the team members, since it was able to comply to all the requirements and design goals set for this project.



# Development Process

In this appendix, we will discuss the different aspects of the development process, and the tools to support and improve it. Each of those aspects will be discussed within the section, named after the used corresponding tool.

## A.1. GitLab

In team-based software development, it is very important to keep the code centralized and to implement a version control system. We have chosen to use git for version control of our source code. Because the code is proprietary, we used a private instance of GitLab [8] to provide us with an easy to use interface for our git repository.

## A.2. Gitlab CI

Another important aspect of developing in a team is to avoid differences in the build environment that can lead to a different outcome of code between developers. Building and running all the code quality tools is also a very time-consuming task that can easily be automated. In this project, we used GitLab CI [9] because it is a nicely integrated Continuous Integration tool that comes bundled with GitLab. GitLab CI works by attaching GitLab CI Runners to the GitLab instance and including a `.gitlab-ci.yml` configuration file to the repository.

When a commit is pushed to the repository, GitLab CI automatically executes all the commands that are defined in the configuration file. In our case, this included building, testing, static analysis of the source code, as well as running memory check tools such as Valgrind. We used a docker image to run and test our code, which made sure that the build environment was clean and consistent. Several private servers were used as GitLab CI Runner so that every push to every branch of the source code can be built and tested in a timely fashion. Because GitLab CI is an integrated component of GitLab, GitLab also provides a nice interface that automatically shows whether a commit has failed or passed and provides with a detailed log file for every build.

The use of GitLab CI has proven itself very valuable because it helped finding configuration issues of local build environments numerous times. It also made sure that all code that was merged into the master branch was of good quality and prevented the breaking of the master branch.

## A.3. Google Test

The project used C++ as the programming language. Unfortunately, the support for code quality tools of C++ is lower than other languages, such as Java. We've searched for a good unit testing framework for C++ and ended up with Google Test [10]. After some adjustment, the Google Test Framework provided us with a valuable way to test our code. We have aimed to have at least 100% line coverage and 80% branch coverage of our code.

## A.4. Valgrind

Valgrind [4] is a memory checking tool that spots memory allocation and de-allocation errors, such as memory leaks or the use of non-initialized values in C and C++ programs. Because we are programming with

limited hardware on an embedded platform, it is crucial that the program is as efficient as possible, which includes the use of pointers and other shared-memory constructs. Such memory constructs improve the efficiency but also increase the complexity of the memory management. To prevent memory leaks and other memory management related bugs, we used Valgrind to spot these errors. Valgrind was automatically run by the GitLab CI Runners on every commit.

### A.5. Vera++

Vera++ is a static analysis tool that can be used to check for style violations in the source code [18]. In this project we used Vera++ to check for code style violations. Vera++ was automatically run on every commit by the GitLab CI Runners.

We have chosen to adopt the Stroustrup code style [16]. Stroustrup is an indent style developed by the creator of C++, Bjarne Stroustrup. Stroustrup allows, among other things, the opening braces for blocks within functions to be at the same line, and short functions to be written within one line. Stroustrup creates clear visible code, without demanding many additional lines.

### A.6. CppCheck

CppCheck is a static analysis tool that can be used to check for common bugs in the source code [11]. CppCheck does not check for syntax issues or code style issues. Instead, it checks for common mistakes such as out of bound checking, memory leak checking, possible nullpointer references, etc. CppCheck was also run on every commit by the GitLab CI Runners.

### A.7. SonarQube

The GitLab CI system was used to automatically test every commit that was pushed to the GitLab repository using the GitLab CI Runners. The GitLab CI Runners would automatically run numerous tools that check and test the code quality. The tools were all configured to generate XML files with the results of their analysis. The resulting XML files were fed into SonarQube [15] to generate a global overview.

SonarQube is an open source platform for continuous inspection of code quality. SonarQube is capable of visualizing the analysis results into a web based overview. The overview contains all sorts of metrics regarding the quality of our code. The results of all the inspections that were run by GitLab CI Runners were available to us in a clear and uncluttered way.

### A.8. Trello

Trello [3] is a web-based project management application. The tool allows so called boards to be made, where lists can be added to. Within a list, a card can be created which contains information like a description, the ownership, a deadline, etc. While originally created to visualize Kanban, an agile software development method, it supports all kinds of management methods which require the visualization Trello offers.

We used Trello to visualize our own software development. For our development, we chose SCRUM as the method, which is another agile software development method. The choice has been made, besides its popularity within current developments, because the practices of SCRUM are familiar with us and the employees from SeaState5.

To visualize SCRUM [1] within Trello, the Trello board has been transformed into a SCRUM board. This can be easily done by using the lists to represent the columns TODO, in progress, and DONE. Each card will then represent a sprint. The cards are able to switch between lists, thereby representing the switching from columns in SCRUM.

### A.9. Doxygen

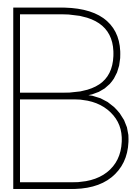
Documentation is an important part of developing software. Doxygen creates a clear vision of the functionalities of the created functions, and the relation between those, without the viewer having to read the source code. The additional visibility allows outsiders, newcomers to the project, or even current developers, to quickly understand the project. The importance of documentations will become noticeable as the project will increase in complexity during the development.

Documentation will be done in this project using Doxygen [17]. Doxygen is a documentation tool designed to be used on C++ programs, but can also accept other languages such as Java and Python. The benefit



---

of Doxygen is that the documentation is maintained in source and is thus always up to date with the current implementation of the code. The Doxygen tool creates external documents based on the source documentation for easy navigation and access through the documentation. Doxygen is also capable of generating the relation between functions and classes, by generating amongst other things an UML of the project.



SIG

In this appendix, we will discuss SIG. SIG (short for Software Improvement Group) is a company that specializes in controlling code to give insight and uncover potential flaws into a project. Using testing metrics, SIG creates a report with explanation on what may be wrong with a project's code and how those issues could be avoided in the future.

As part of the project, our code had to be sent to SIG for evaluation. In the next session, SIG's initial feedback can be read. After the first evaluation, we were given approximately 3 weeks to edit the code according to the feedback, which had to be re-sent to SIG for a second time afterwards. The evaluation, found below in section B.1, is translated from Dutch to English.

## B.1. Evaluation

The system's code scores 3.5 stars on our maintenance model, which means that the code is above-average in maintainability. The highest score hasn't been achieved due to a lower score for Unit Complexity.

For Unit Complexity, we monitor the percentage of methods in the code that are above-averagely complex. Splitting such methods into smaller parts leads to an easier understanding, easier testing and easier maintenance of these methods.

In your project, for example, the method `SensorLogger.parse_can_msg` is unnecessarily complex. The bit `to_uint32((uint) d->get_bit_start(), (uint) d->get_bit_end())` appears in every case of the switch-statement, so that it can be then converted to the right type. When it comes to readability, it is better to execute that bit of code once, to avoid remaining with a gigantic switch-statement.

In `SensorData.cpp`, something similar to the aforementioned can be seen. In that class, it's also advisable to separate the shared code, which not only will result in increased readability, but also will make it easier to edit that code in the future (that way, you only have to edit the code once, rather than edit it six times).

At least, the presence of test-code is promising. Hopefully, the volume of test-code will increase as more functionalities are added to the project.

Overall, the code scores above-average. Hopefully, this score can be maintained throughout the remainder of the development phase.

## B.2. Improvements

The suggestions of SIG were incorporated into the existing code base and were also applied to new code. The main point of the SIG evaluation was code complexity. The second point of the SIG evaluation was test code volume. We will address the points of the SIG evaluation below.

### B.2.1. Code complexity

The `SensorLogger`, which is currently called the `SensorServer`, was rewritten to create a better solution for decoding and encoding of CAN messages. In the rewritten code the complexity of the method is drastically decreased by delegating (complex) tasks to other methods.

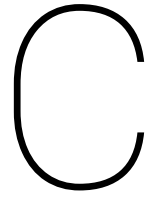
Other complex classes throughout the program were also refactored to create a code base that was easier to test, due to smaller functions, and easier to extend because the functionality can be reused when it is needed elsewhere.

The public API of several classes have been altered to handle complex code within the class and provide simple functions to access the complex functionalities. The retrieval of fields from a `CANData` object is one of the examples.

### **B.2.2. Test code volume**

The second point of SIG was that the presence of test code was promising and that SIG would like to see the volume of test code increase as new functionality is added. The volume of test code has increased linearly with the amount of production code, as the (relative) test coverage remained the same. We aimed to achieve 100% line coverage and 100% branch coverage. During the project we found that the available tools for C++ code coverage all suffered from erroneous branch reporting due to branches added by the compiler. The exact branch coverage was therefore difficult to determine but we have succeeded in 100% line coverage in most of the code.

As described in chapter A, we have embedded the use of several code quality tools into our development process. All the tools reported their findings to our SonarQube server which created an aggregated report. During the rest of the development process, we continuously checked the SonarQube reports to improve our code.



# Project Description

In this appendix, we will give the project description from BEPSys, stated by the company, SeaState5. This is an direct copy from the original text, only altered to fit the current format.

## **C.1. Project Description**

Will you join our team of pioneers to build a revolutionary unmanned vessel? And contribute by developing a framework for the control algorithms of this vessel?

Together with SeaState5; an innovative Yes!Delft company, you will be part of the development of the core tech for our autonomous vessel, called the Eagle. Our vessel will be used in applications such as hydrographic survey, environmental monitoring, search and rescue (SAR) and maritime safety.

Together with experienced control system-, mechanical-, and maritime- engineers you will develop an application which will run on-board the vessel. The application will be a platform with two purposes; it will be able to implement algorithms for executing core tasks; such as navigation, path and heading control, object detection. Depending on the complexity the team is free to choose a specific task and implement it in the developed framework. And secondly serve as a server from which users can remotely view and log data from the vessel through a user interface. A split will be made between the vessel systems and the payload systems.

Currently a preliminary design had been made for the hardware components, which will be finalized in collaboration with the skills from the participating team before the project starts.

## **C.2. Company Description**

SeaState5 has been founded in 2015 and is developing robotic systems for the maritime sector. Among new installation tools and noise mitigation systems for the offshore wind sector, we continuously develop new ideas into potential products. SeaState5 has a partnership with the TU Delft's Robotic Institute, to accelerate the development of its robotic solution within the offshore renewable sector.

## **C.3. Auxiliary Information**

Are you interested in applying your skills in a fast moving organisation, and test your implemented software design on our testing vessel, then join our team!

We offer a part-time on-site working environment with free beers on friday ;), and many activities with the team during the project.

## C.4. Info Sheet

**WANT TO PROGRAM AN UNMANNED VESSEL?**

**CS BEP project:** Design and testing of an autonomous vessel's software framework

**OBJECT DETECTION**

**NAVIGATION**

**PATH & HEADING CONTROL**

**GOAL:** develop the core tech for autonomous vessels, used for hydrographic surveying, environmental monitoring, search and rescue (SAR), and maritime safety.

**YOU ARE:** a computer science student, with knowledge of application design.

**@ SeaState5:** Our core business is developing robotic maritime systems. We are located at Yes!Delft.

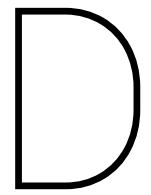
Interested? Join our team!  
More info: [wz@seastate5.com](mailto:wz@seastate5.com) or BEP0ys

**TU Delft** Delft University of Technology

**YES!Delft**

**SeaState5**

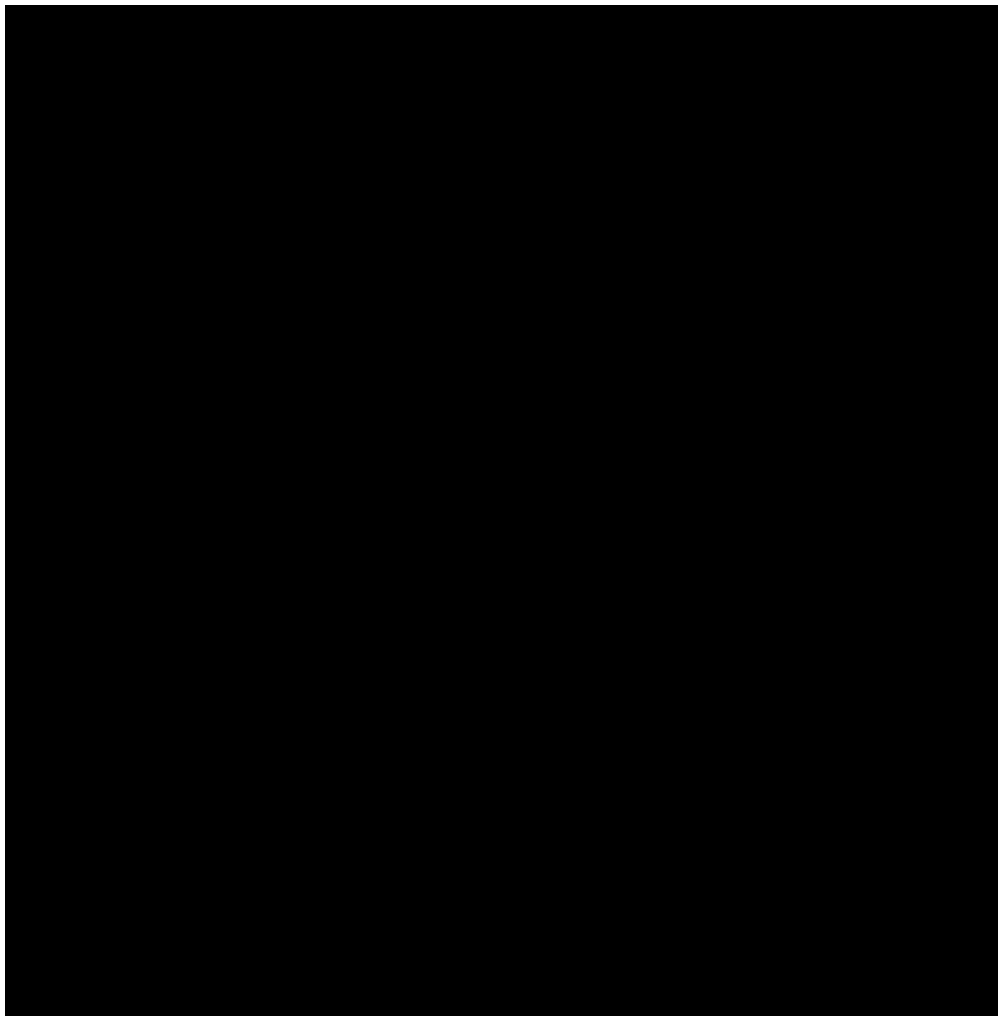
Figure C.1: The original info sheet, given in addition to the project description.



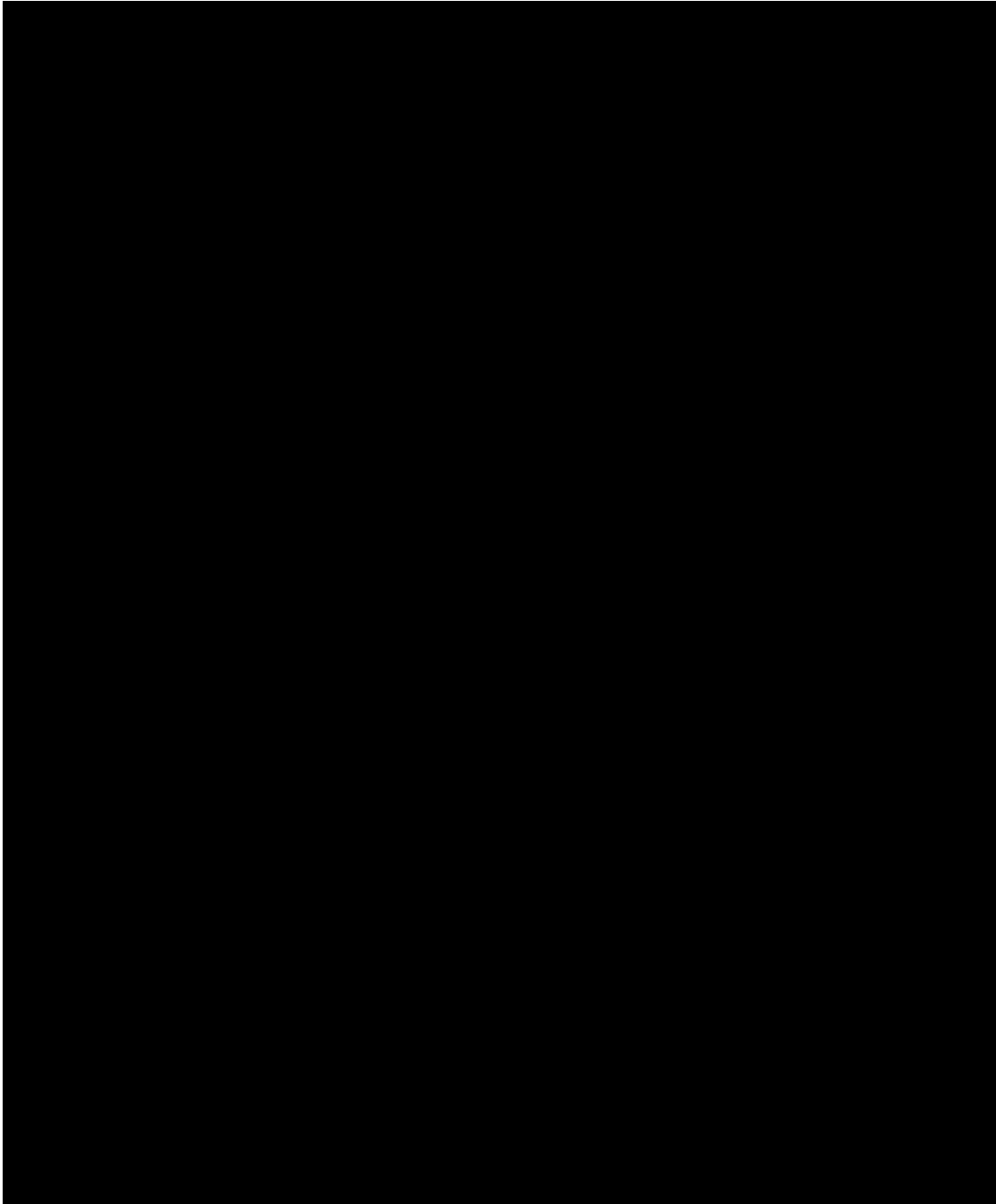
## Field Test

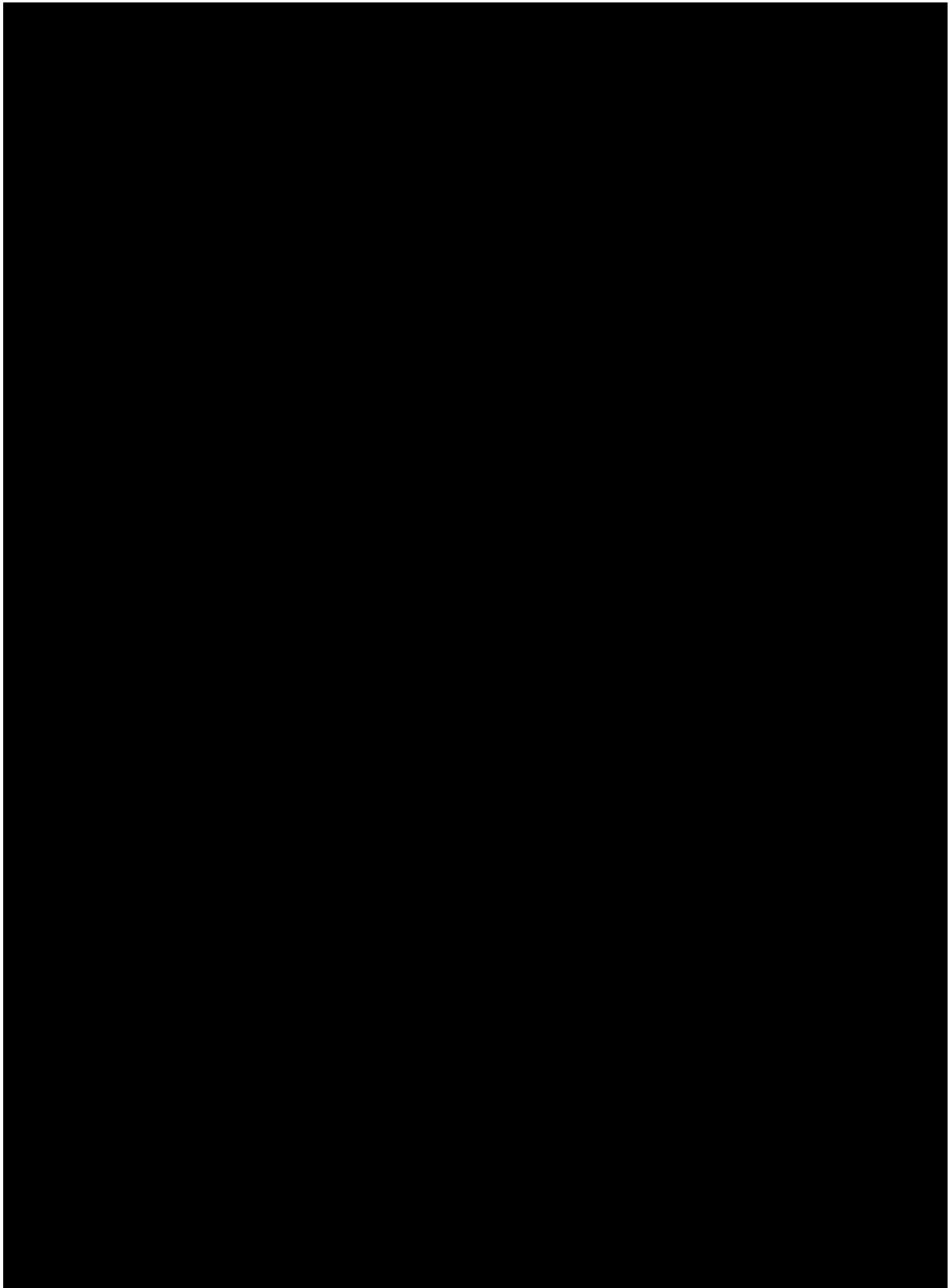
Multiple field tests have been conducted with a scale model prototype of the Eagle. The field tests extends over two days. The second test day has been used to improve failures found in the first day. The second day has been predetermined to be the final test day. In this appendix, we will give the direct copy of the test plan for the field test, and the results from both days.

### **D.1. Test Plan**

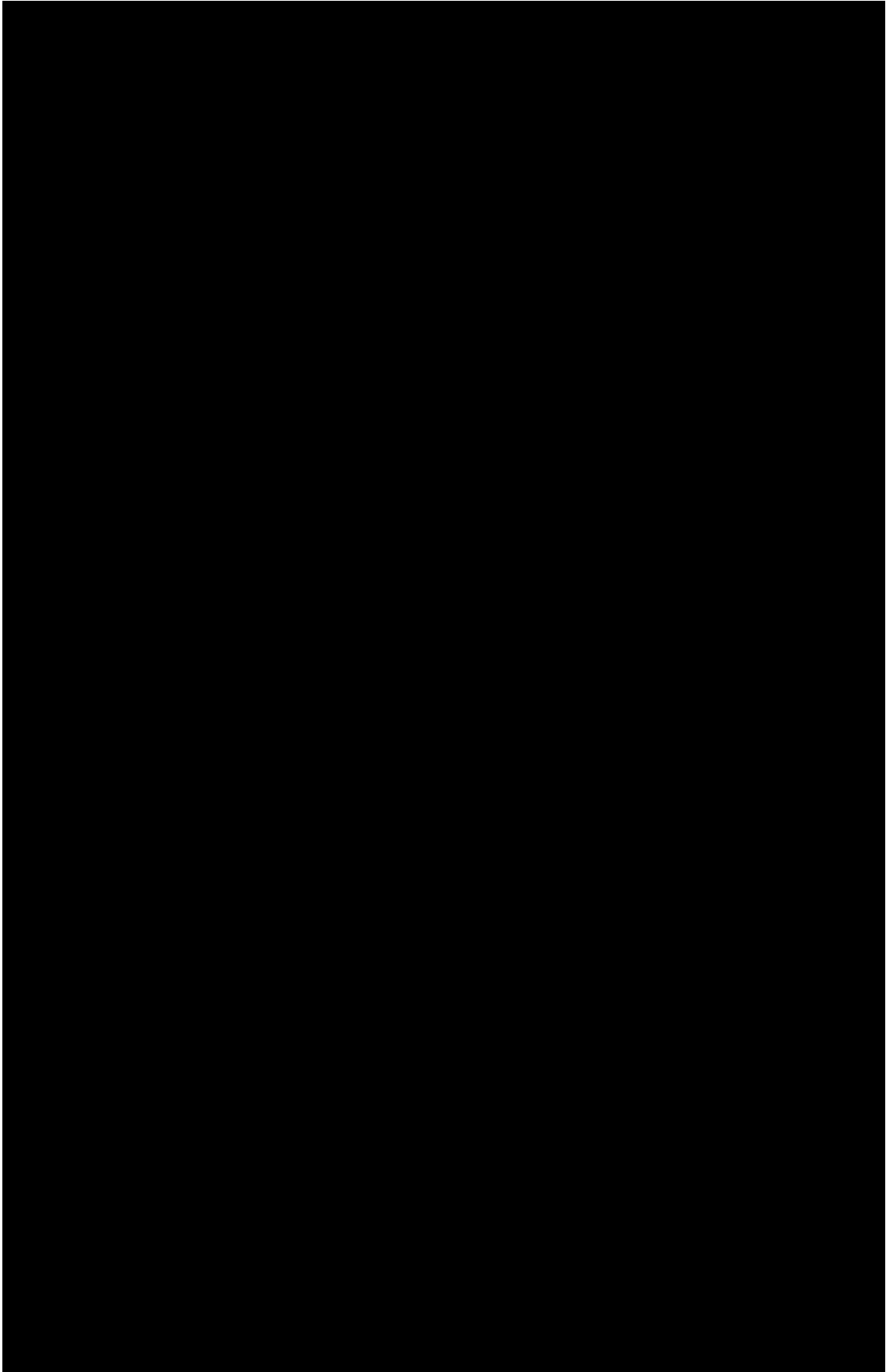


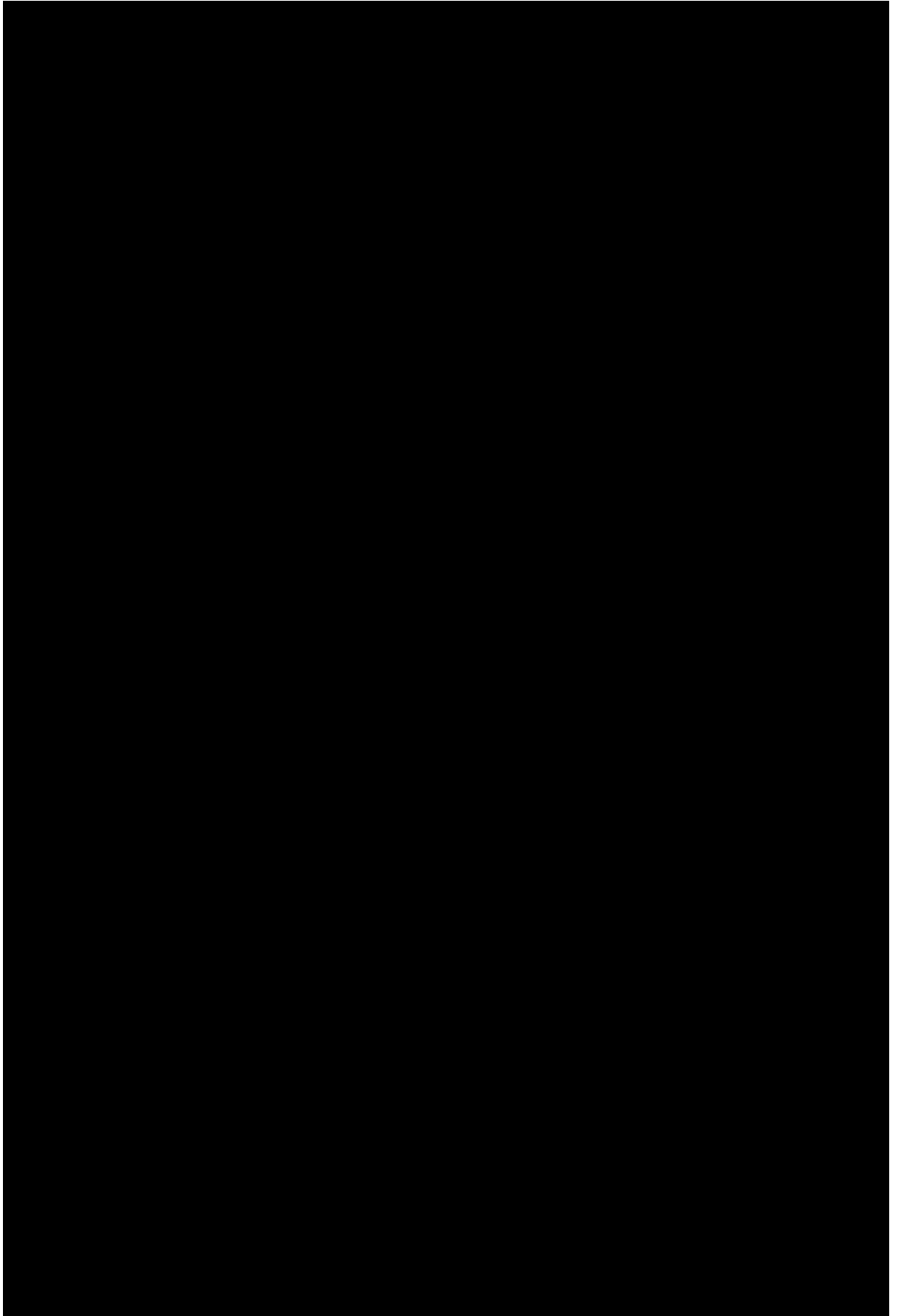
Test plan control systems 'Eagle' – rev 1.0

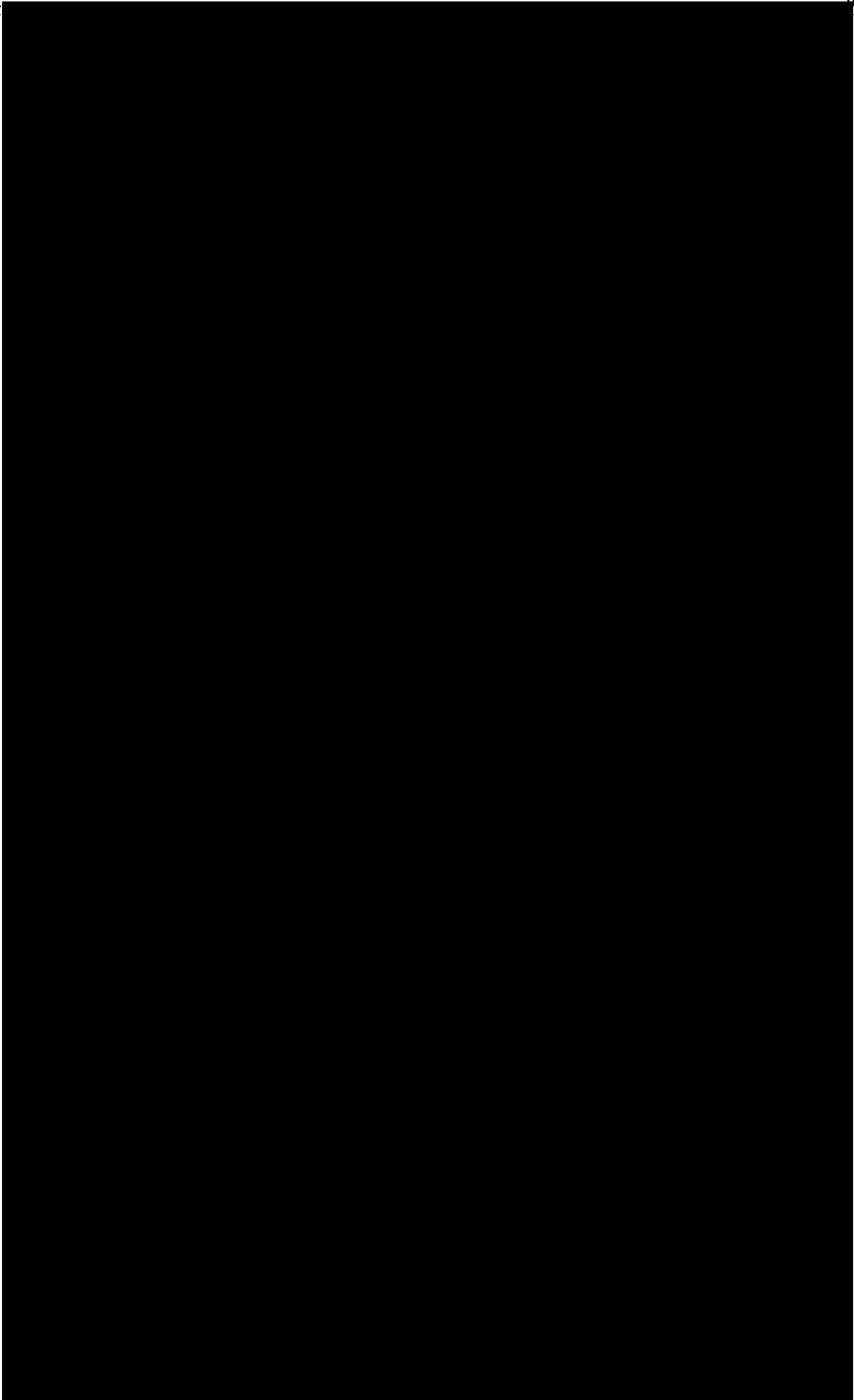


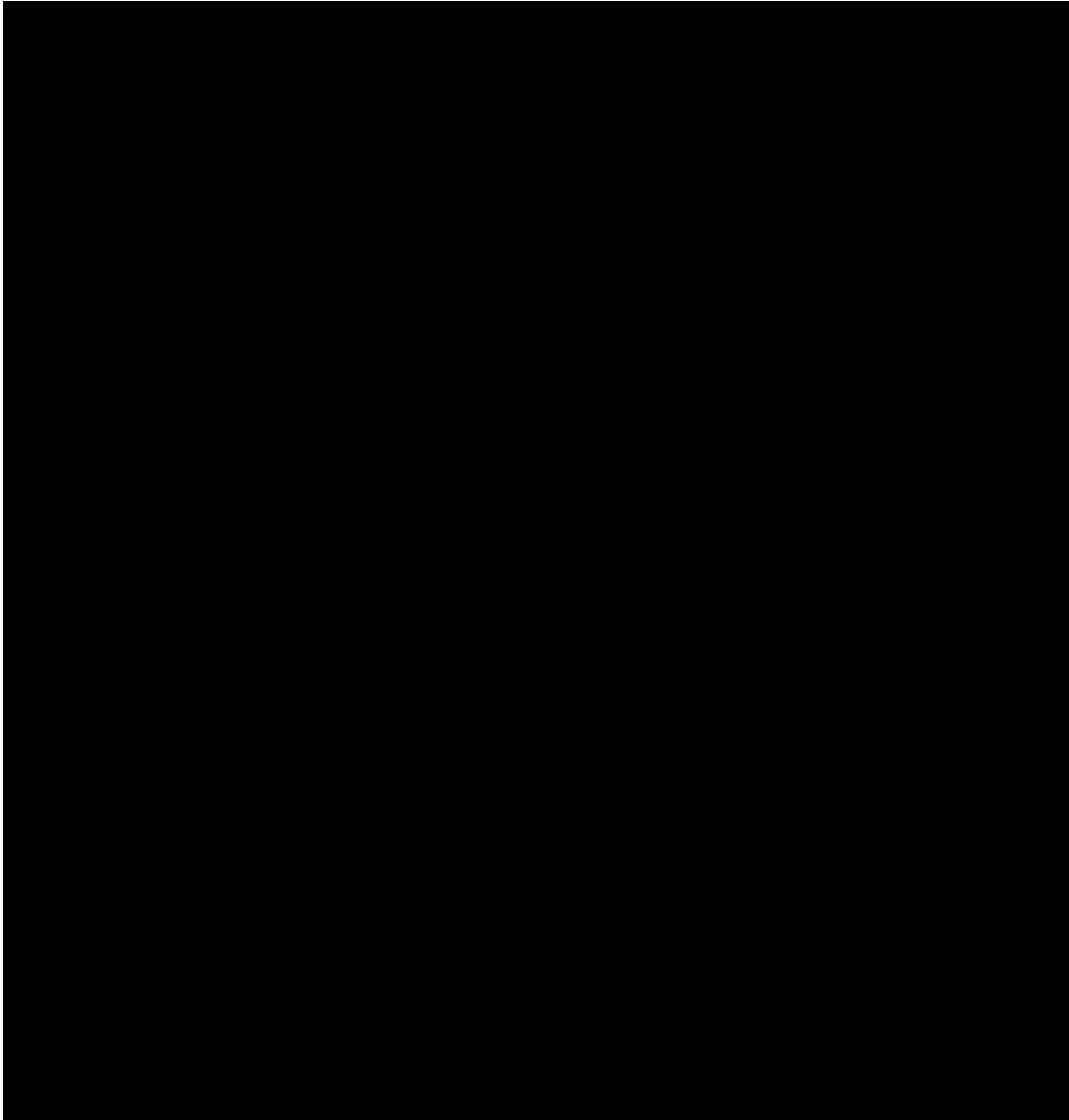












## D.2. Results

[REDACTED]

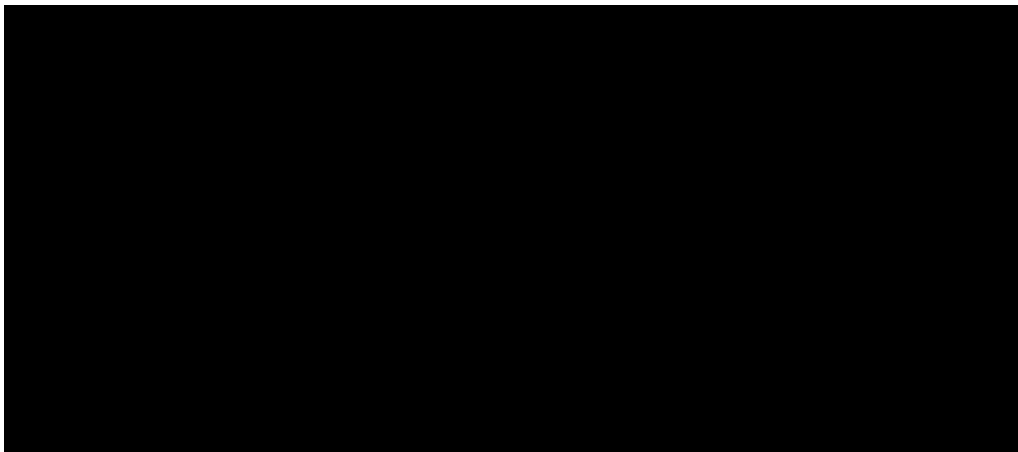


Figure D.1: The GPS traces of the Eagle, autonomously traveling a course retrieved from the cloud.

