Data analysis application for the ORCHID

Stefan van der Heijden
Nils Hullegien
Stas Mironov
Maaike Visser

TU Delft

Preface

This report concludes the TI3806 Bachelorproject (BEP) of Q2 in the academic year of 2018/2019 at the Delft University of Technology. For this BEP we, Nils Hullegien, Stefan van der Heijden, Stas Mironov, and Maaike Visser, created a data visualization application for the ORCHID, a unique experimental power plant designed for fundamental investigations. This project caught our eye from the get-go because it combines a number of different software development aspects, as well as because of its implications for the development of renewable energy sources.

The application was created over the span of 10 weeks, 2 of which were dedicated to research and 1 of which to the final presentation. This report contains all the information on the design, implementation, and testing of the final application. As this was an academic project, we focus in particular on the challenges we faced in drafting and implementing the main design. Also included are our recommendations for future improvements.

We would like to extend our special thanks to our client, Adam J. Head, for his close involvement and indispensable contributions to this project.

Summary

The Organic Rankine Cycle Hybrid Integrated Device (ORCHID) is a small scale power plant that is used to study the fundamental gas dynamic behavior of dense organic fluids, as well as the behavior of turbomachinery. In order to draw accurate conclusions about the raw sensor data generated by the ORCHID one has to know when the system is in steady-state. Currently, determining the steady state over historical data is cumbersome, and difficult to do in real time.

Our application aims to solve the problems with the current information workflow by consolidating the functionality that is currently spread across multiple applications into one main application, as well by offering steady state detection over real-time data. Aside from the lack of steady state detection capabilities, our client indicated that the applications currently in use often lag or crash. Therefore we defined three design goals: Performance, Reliability, and Ease of Use.

The main challenge we encountered during this phase was finding a way to properly connect the different external applications needed to properly process the ORCHID's data. The design goals were continuously referenced during the implementation phase to ensure the quality of our application. Additionally, we used unit, integration, and manual testing. The last category also comprised user tests conducted with our client to ensure that the final product would meet his requirements.

With our final application, we solve the client's main problem: it is now possible to detect whether or not a system is in steady state while an experiment is being conducted. This greatly reduces both the amount of time the client has to invest, as well as the amount of energy needed to conduct a successful experiment.

Contents

Chapter 1: Introduction

The Organic Rankine Cycle Hybrid Integrated Device (ORCHID) is a small scale power plant — designed by our client — that is used to study the fundamental gas dynamic behavior of dense organic fluids, as well as the performance of turbomachinery. The ORCHID models the Rankine cycle [3], an ideal power cycle commonly adopted when building steam power plants, but using an organic fluid instead of water as the working fluid medium.

Organic Rankine Cycle (ORC) power systems are gaining recognition because of their potential for recovering waste heat in mobile and stationary applications, for example in heavy-duty truck engines or geothermal wells. However, at this time the cycle efficiency of these systems is still too low to make them attractive/-competitive as a renewable energy source. More design iterations and detailed engineering is still needed to make the technology mature enough for the vast majority of applications and user cases. Furthermore, the software codes used to design these systems are not yet validated because of a lack of experimental data. By providing this much-needed experimental data, the ORCHID aims to aid the research into this immature technology [9].

The experimental data from the ORCHID is currently acquired and processed through a number of applications written in LabVIEW[29], a graphical programming language created by National Instruments (NI)[30]. In order to draw valid conclusions from the data it is important to be able to detect whether or not the system is in steady state, meaning that a number of relevant variables describing the internal processes are constant over time [2]. If a system is not in steady state, it is impossible to say whether fluctuations in variable values are due to a fundamental process, or because of disturbances in the machinery.

With the current data acquisition and processing setup it the steady state is difficult to identify in historical data sets, and impossible to identify in real time. A workaround is to let the ORCHID run for multiple hours at a particular operational point and to assume that the system eventually reached steady state. However, this process is wasteful both in terms of energy — running the ORCHID for 8 hours can consume as much energy as a typical household does in a year — as in terms of the client's time, as instead of knowing during their experiment when steady state has been reached, they can only conclusively say so afterwards.

Our aim for this Bachelor End Project (BEP) was to build an application that can remedy the aforementioned problems. This application facilitates fast identification of the steady state of the ORCHID, as well as a way to easily explore data sets in a visual application. Most importantly, steady state can be detected both in real time as in historical data sets. As such, our application allows the client to make better use of the unique data sets, *in situ*, generated by the ORCHID.

This report is structured as follows. In Chapter 2 we analyze the workflow of the ORCHID's data acquisition and processing process and define the exact problem we aim to solve. The problems we defined in this chapter formed the foundation for the final design and design goals which can be found in Chapter 3. Once the design had been established we could use it as template for our implementations. Some highlights of the implementations, as well as the reasoning behind the choices for certain frameworks and libraries can be found in Chapter 4. The was in which we assured the quality of our implementation through testing is found in Chapter 5. The final product and process are evaluated in Chapter 6 and 7 are respectively. Here we also give some recommendations for future work. We wrap up the report and this project in the conclusion in Chapter 8.

It should be noted that we have integrated the research report we wrote during the first two weeks of this BEP into relevant sections of this final report such as to create a more cohesive reading experience. The full research report can be found in Appendix A.

Chapter 2: Problem Statement

When creating a solution to any problem, one of the first steps is to determine exactly what the problem *is*. As such, an important aspect of the two-week research period at the beginning of this BEP was to define and analyze the problem more clearly. The results from this analysis are found in this chapter.

For context, we start in Section 2.1 by describing the current data acquisition and processing workflow. Then, a clear problem definition and analysis is given in Section 2.2. Our proposed software solution, based on the analysis, is found in Section 2.3.

2.1. Current Workflow

This section describes the way in which data is currently acquired from the ORCHID, as well as the way in which the data is being processed.

2.1.1. Data Acquisition

The ORCHID gathers two kinds of data: raw sensor data, such as temperatures, pressures, and mass flow rates of the fluid inside the ORCHID, and visual data in the form of images of the gas flow. The sensor data is gathered automatically - it is this data that is visualized by our application.

The raw sensor data from the ORCHID is gathered by more than 50 sensors that measure temperature, pressure, and mass flow rate of the fluid in each of its components, e.g., heat exchangers, buffer vessels ect. Every second, the data is sent to a Field-Programmable Gate Array, more specifically, a compact RIO or cRIO[25], which is configured to act as a controller for the ORCHID. The controller sends the data to a collection of LabView VI's (LabVIEW programs) that work together to control and monitor the ORCHID, as well as to store gathered data [1]. Each measurement variable is stored in the VI as a so-called shared variable[27]. These variables can be be accessed by other VI's, or be broadcast over for example a Transmission Control Protocol/Internet Protocol (TCP/IP) connection. As such, the shared variables can be used to communicate information about the ORCHID to the outside world.

From the VI collection, the data is sent to two more locations: a visualization application, and a database for permanent storage. The visualization application is an implementation of the LabVIEW Datalogging and Supervisory Control Module (DSC module) [28] which can display raw data in a number of graphs in real time. The permanent storage of the data is done by a Citadel 5 database [24]. A more in-depth description of this database can be found in Section 3.3.2 and 4.1.3. Data in the Citadel 5 database can be viewed through another NI application, the NI Measurement & Automation Explorer (NI MAX)[22].

The ORCHID's current automatic data acquisition flow is represented graphically in Figure 2.1.

2.1.2. Data Processing

There are two ways in which a user can interact with the acquired data. As mentioned in the previous section, the sensor data generated by the ORCHID is visualized primarily by the DSC module. The DSC module plots the data it receives in real time, allowing a user to closely monitor the state of the ORCHID. However, no further data processing is done at this stage; only raw (unsmoothed/unfiltered) data is shown.

In the NI MAX, the user can perform a limited number of statistical operations on the raw data, and display the result. However, the only way to perform more complex operations on the data is to export the data to a CSV file, clean up the CSV file, import the data into MATLAB [36], and then call MATLAB operations on the imported data.

[1] The original control, monitor, and storage VI's were developed by Carya Automatisering.

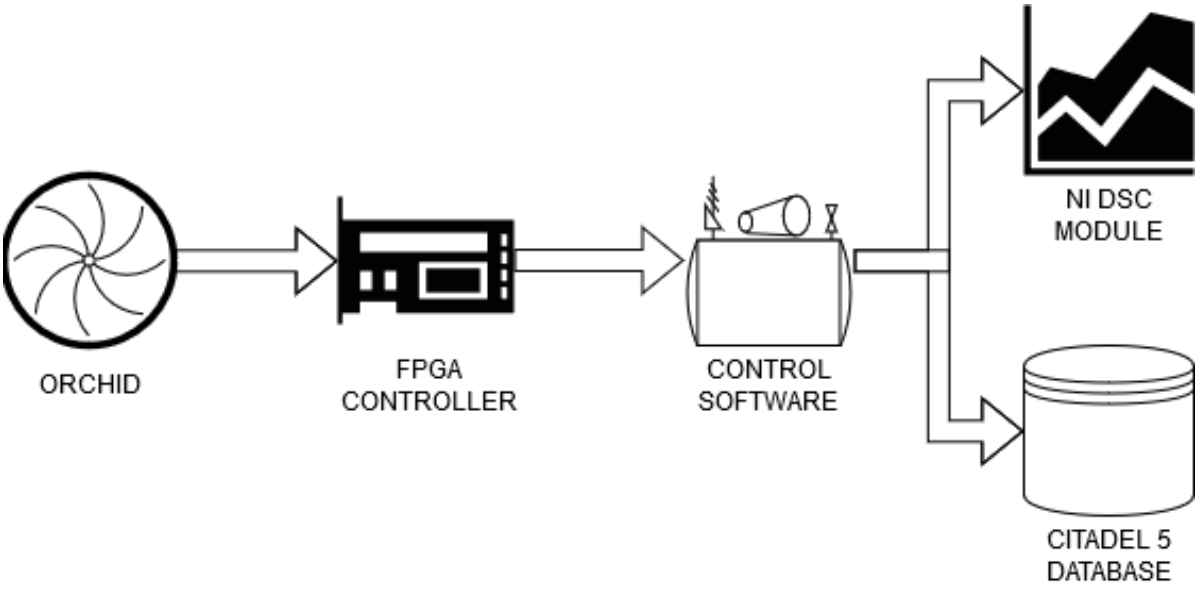

Figure 2.1: The current flow of data from the ORCHID to the database and visualisation module.

2.2. Problem Definition and Analysis

The current method of processing data from the ORCHID is less than optimal for a number of reasons. Firstly, in order to visualize the data and extract it multiple applications (at least one for the real-time data and one for historical data) and manual actions are necessary. Real time data being visualized in the DSC module cannot be filtered or smoothed and cannot be extracted directly. Historical data loaded in the NI MAX can be filtered and smoothed, but it is still not possible to see which parts of the data were gathered while the system was in steady state. Instead, the data has to be extracted to a CSV file, cleaned up, imported into MATLAB, and then analyzed.

The reason that several different applications are used is related to the second issue: in the client's experience the applications currently being used are prone to lagging and even crashing unexpectedly, especially when displaying large amounts of data. This is especially undesirable for software controlling the operation of the ORCHID, making it necessary to split functionality across multiple programs.

The frequent lagging and crashing may be due to the fact that the client only has access to older versions of both the DSC module, the Citadel 5 database, and the NI MAX. For reference: the client uses version 16.0.0 (released in 2016) of the DSC module, and 17.0 (released in 2017) of the NI MAX. Upgrading to newer versions of the software is not possible because this would require altering the control software of the ORCHID. A side effect of the current applications being less modern is that their GUI's are outdated and at times even counter-intuitive.

Lastly, the current approach for determining steady state for real time data is wasteful. Since it is not possible to know whether the system is in steady state while an experiment is being conducted, the system is instead run for a number of hours, after which it is assumed that steady state has been reached. At its maximum power consumption, the ORCHID consumes 400 Kw per hour. Since a typical test run takes 8 hours, one experiment can consume up to

$$400[kW] * 8 * 60 * 60[s] = 3200[kWh]$$

. This amounts to the electricity used in an average house in one year [4].

Aside from the DSC module and the NI MAX there are a number of other software packages that offer data management and post-processing capabilities, such as FlexPro[43] and ibadatmanager[10]. However, these packages are not tailored specifically to power plant monitoring, nor do they offer options for monitoring data in real-time. Software explicitly built to monitor power plants is not commercially available, because it is normally built for a specific plant. In short, no existing software solutions to our client's problem exist.

2.3. Proposed Solution

The aim for our BEP was to develop an application that solves the aforementioned problems. This application, specific for the ORCHID, visualizes the data, supports the application of smoothing/filtering algorithms, and allows the client to explore the data visually so the data can be interpreted more easily. Most importantly, the application allows the client to identify the steady state of the ORCHID both in real time as over historical data.

Fast identification of the steady state allows the client to tweak operation parameters of the ORCHID in such a way that steady state is obtained more quickly, thus reducing the operation cost of the ORCHID, and allows the client to more quickly interpret their data sets.

Chapter 3: Design

After defining and analyzing the problem in Chapter 2 we had a global idea of what our solution should look like. However, before we could start implementing our ideas we had to draft a clear design of our final product. We first defined a number of design goals (see Section 3.1) to adhere to during the development process to make sure that our product would align with the client's needs. Because of the complex nature of our problem, in drafting our design we met with some interesting challenges which we describe in Section 3.2. Our final design can be found in Section 3.3.

3.1. Design Goals

Our problem analysis left us with a clear view of the specific issues our software should solve. Working together with the client, we defined a number of design goals to target the existing issues as effectively as possible. The main design goals are: Reliability, Ease of Use, and Performance, each of which will be explained briefly below.

3.1.1. Reliability

One of the main downsides of the current software and the way the data is processed is the reliability. As mentioned before, some components of the software, particularly the DSC module, are outdated. This module frequently lags or even crashes when handling large amounts of data.

There is also no way in which to reliably detect the steady state of the ORCHID when an experiment is being conducted, even though knowing whether or not the system is in steady state at a given point in time is needed to guarantee the validity of the measurements and conclusions drawn based on them.

3.1.2. Ease of Use

Another way in which we wanted to improve the current workflow was by making the process of acquiring, processing, and visualizing data easier. As mentioned in Section 2.2, the current process takes multiple manual actions and is spread across three different applications. By consolidating the different kinds of desired functionality into one application and reducing the number of steps needed to acquire and process data we intended to make it easier for the user to use their data sets.

3.1.3. Performance

Each time an experiment is conducted on the ORCHID, a large amount of data is generated — as mentioned earlier, the ORCHID has more than 50 sensors that all collect data once every second for multiple hours. However, loading large data sets in the DSC module or the NI MAX takes a long time. Even when the data is loaded, the applications often lag and are liable to crash completely. To improve upon this experience a lot of our focus was on the Performance of the application.

3.2. Design Challenges

One of the main challenges of this project was to find a way to unite a process that was previously spread across a number of actions and applications into one application. At first, our aim was to write our entire application in Java for consistency's sake. However, there were certain kinds of functionality that were not possible to implement in Java alone.

Firstly, the ORCHID control software itself could not be significantly altered. As such, the measurement data had to be acquired either from the Citadel 5 database in the case of historical data, or through a VI in the case of real time data. However, neither the Citadel 5 database nor LabView's VI's have functionality to communicate directly with a Java program, meaning we had to find a solution for this communication

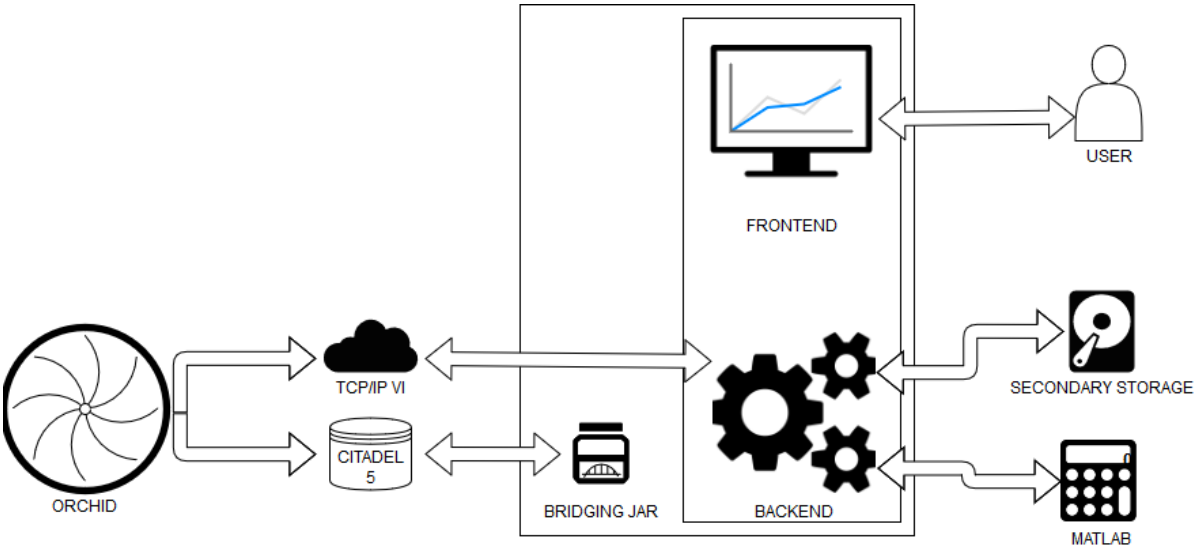

Figure 3.1: A global overview of the final design of our application. The ORCHID's data is stored by the Citadel 5 database and broadcast by our TCP/IP VI. It is then read by our application and visualized and processed. The user can choose to save the processed data to their local secondary storage.

ourselves.

Secondly, we could not use Java statistics libraries for the data processing. Instead, we used a number of different MATLAB files (see Appendix F). The reason for this was threefold. Firstly, our client already had a number of MATLAB files with the required functionality prepared. Secondly, the client wanted to be able to quickly change or add operations later, which would be much easier for them to do in MATLAB than in Java. Thirdly, and most importantly, the client uses FluidProp [1], a thermodynamics library, for their calculations (see also Section 3.3.1. This library does not have an interface with Java, but *does* have an interface with MATLAB.

Our main challenge then was to find a way to combine the different components while still facilitating Ease of Use, Performance, and Reliability.

3.3. Final Design

After going through a number of iterations the final design was determined. While it was clear that not all desired functionality would be able to be implemented in pure Java, we did not want to sacrifice too much on Ease of Use. Therefore we decided to create one main application that would act as the point of interaction for the client. The main application is divided in a front-end in the form of a Graphical User Interface (GUI) that the client interacts with, and a back-end that contains the logic needed to interact with the external components, each of which is described in more detail below. The final design can be found in Figure 3.1.

3.3.1. MATLAB

The different kinds of filtering, smoothing, and steady state detection algorithms that our client uses are implemented in MATLAB. It is possible to interact with MATLAB from a Java program by using the MATLAB Engine API for Java [35]. Through this API, we can call MATLAB functions with the ORCHID data as input, and retrieve the output.

FluidProp

A simple way to determine whether or not a system is in steady state is to calculate the energy balances over different components of the ORCHID. If the amount of thermal power going into a component is equal to the amount coming out this indicates steady state behaviour.

In addition to standard MATLAB functions, we use an interface called FluidProp [1] to calculate the energy balances. It allows the user to access different thermodynamics libraries that contain equations of state for different fluids. The libraries also contain thermodynamical and fluid transport models — exactly the kind of model the ORCHID's data is meant to help validate.

3.3.2. Citadel 5

The historical data of the ORCHID is stored in a 32-bit Citadel 5 database. This database is an integral component of many of the software applications offered by National Instruments. It is designed to quickly store and retrieve data and to allow users to share data across networks. As mentioned in Section 2.1.1, data from a Citadel 5 database can be visualized with the NI Measurement & Automation Explorer.

The data in a Citadel 5 database is organized into traces, which cover a certain time period. Each trace is divided into subtraces each corresponding to a specific measurement. The database is stored on disk as a group of related files placed in one folder. The files are saved in a compressed format, and as such are only accessible through the NI MAX [24].

There is no standard way to query a Citadel 5 database from a Java program, since no Java Database Connectivity (JDBC) [12] drivers exist for it. However, it is possible to configure a Citadel 5 database as an Open Database Connectivity (ODBC) [5] data source [21]. This data source can then be queried by a Java application by using the Java JDBC-ODBC bridge package. The JDBC-ODBC bridge creates a connection between the Java application and the database after which the database can be queried using basic SQL queries. Not every database accepts all queries — the Citadel 5 database only supports queries for one specific keyword [23].

3.3.3. TCP/IP VI

The control software of the ORCHID is fully implemented in LabVIEW. As we discovered in our research phase (see Section 8.1 of Appendix A), there is no standard way of connecting a VI to a Java program that is running on a different machine. However, it *is* possible to send shared variable data from a VI over a network using the User Datagram Protocol or TCP protocols [26, 27]. Java for its part offers functionality for listening to network ports. Sharing the variable data then becomes a matter of sending to and listening to the correct ports. As such, we decided our design needed to include a VI that could send data packets with information about the shared variables to our application across a TCP/IP connection.

Currently, the client's data acquisition and data processing machine are connected by a simple local network. However, the TCP/IP protocol works for more complex networks as well. As such, using this method allows the client to expand the software and share the sensor data over more complex network setups than the current local setup later on.

3.3.4. Bridging JAR

At the end of the research phase we had a reasonably clear idea of what our final design would look like. All of the components from the previous sections had already been included. The only major change to the design was the addition of what we have dubbed the "Bridging JAR".

As will be explained in more detail in Section 4.1.3, as we were implementing our design we realized that the functionality for querying the database would have to be split off from the main application to make sure we could use an up-to-date version of Java as our main development language. We chose to gather this functionality in a separate JAR file and package it with our main application. As such, it is not truly an external component, but also not a regular Java library.

Chapter 4: Implementation

The design was the foundation for the implementation. This chapter describes the specific ways in which we implemented each aspect of the design. We start with a description of the programming languages and frameworks we used in Section 4.1. In Section 4.1.3 we talk about some of the challenges we faced that were unique for this project. The chapter concludes with a description of our final implementation with a focus on some of the more interesting aspects in Section 4.2.

4.1. Languages and Frameworks

In this section we motivate the decisions we made about the programming languages and frameworks we used to develop our application.

4.1.1. Java

When considering which language to use for the development of our main application, the choice quickly fell on Java [31]. There are a number of reasons for this choice.

Firstly, all members of the development team had used Java before and are proficient in it. This was important to us because we only had 7 weeks to develop the software product. Using a language we were all familiar with allowed us to use our time more efficiently.

Another advantage of Java's "fame" is that there are many off- and online resources available that could help us surmount any roadblocks we might encounter. The plethora of resources is also useful for our client in the event that they would like to develop the application further after the completion of our BEP.

Thirdly, because Java is so widely used a lot of extended functionality exists in the form of libraries. This was particularly useful for us since we needed to communicate with a number of external applications in different ways — with MATLAB directly through its Java API, with a VI over a TCP/IP connection, with the database through a JDBC-ODBC bridge, and internally through an eventbus. Java has either native support or offers the functionality through a library for all of these interactions.

Version

The specific version of Java we used for the main application is Java 8 64-bit [19]. While this version of Java is outdated (for context: the latest version is Java 11 [18]), it is the most recent version to properly support the MATLAB engine.

It is important to note that while our main application is written in Java 8 the Bridging JAR mentioned earlier in Section 3.3.4 was written in Java 7 32-bit. The specific implementation as well as the necessity for using a different Java version for the JAR are explained further in Section 4.2.1.

4.1.2. JavaFX

There are a number of GUI development frameworks and libraries available for Java, some more appropriate for our project than others.

One of the requirements for our application was that it had to be able to visualize large amounts of data in a graph. During our research phase we actively looked for similar projects to see if we could follow a similar approach. In particular we found a project by [33]. They represent data in real time by using MATLAB GUIDE [11], a drag-and drop style UI builder. However, since developing this application was part of attaining a bachelor's degree in Computer Science, we felt that a framework using more a classic programming approach was more suitable.

Another framework we considered is libGDX [34]. Its main advantage is that it is optimized for fast data processing and overall performance in terms of speed. At first this made it seem like a good fit for our project, however, only one of our team members had previous experience with libGDX. Upon further exploration it

became that libGDX has such a large number of features that figuring out which ones to use and *how* to use them would require a large time investment to get the entire team up to speed. Coupled with the fact that many of graphical components have to built from the ground up instead of being a standard part of the library means that the cost of the initial time put into learning framework would be too large for the payoff.

In the end, we found two options that met our requirements for a Java GUI framework that both takes a classical programming approach and did not have too steep of a learning curve for our team: JavaFX [38] and Swing [17], both created by Oracle[20]. Between these our choice fell on JavaFX both because it was originally created as a replacement for Swing [15], and because JavaFX natively includes a lot of the functionality that Swing needs external libraries for.

4.1.3. Implementation Challenges

Much like was the case with the design, the main implementation challenge came from the fact that there were several external components that had to work together properly. Aside from challenging our software development skills by forcing us to come up with creative solutions, some of which will be highlighted in Section 4.2, the main influence was on the version of Java we ended up using.

Interaction with Citadel 5 database

As mentioned in Section 3.3.2, the Citadel 5 database storing the historical data from the ORCHID cannot be queried directly from a Java program because no Citadel 5 JDBC[12] driver — a driver enabling the communication between a Java program and a specific database — exists. Instead, in order to access the data from a Java program, the database first has to be configured as an ODBC data source [21]. Once this has been done, the database can be queried by using Java's JDBC-ODBC bridge package.

A difficulty arose because of the fact that the JDBC-ODBC bridge was removed from Java 8 onward [16]. As such we initially decided to use Java 7 to develop our application. Specifically, we wanted to use the 64-bit version to make sure our heap space was as large as possible. However, the Citadel 5 database used by the client is the 32-bit version. Since the JDBC-ODBC package only enables communication between Java and ODBC sources of the same bittage, and upgrading the database was not an option lest our client run the risk of losing their data, we had to downgrade to the 32-bit version of Java 7 as well.

Interaction with JavaFX

While using Java 7 32-bit allowed us to query the database, it also imposed some limitations. The first issue we ran into is that the most recent version of JavaFX to work well with Java 7 is JavaFX 2.2 [13]. However, JavaFX 2.2 is outdated: the most recent version is JavaFX 11.

As we were implementing the different GUI features of our application, it became clear that JavaFX 2.2 simply did not include all the functionality we needed. In particular, in JavaFX 2.2 it is not possible to create an interactive chart with multiple data sets and y-axes, something our client *did* require. Therefore, the decision was made to upgrade our project to Java 11 64-bit. Since JavaFX is not automatically included with Java 11 this change meant setting up the new environment was slightly more time consuming. On the other hand, upgrade to a 64-bit Java version meant that we had access to a lot more heap space, making it easier to keep our product aligned with our design goal of Performance.

Of course, using Java 11 meant that we no longer had access to the JDBC-ODBC bridge mentioned in the previous section. Therefore we decided to split off this functionality into a separate JAR, to compile this JAR with Java 7 32-bit, and then call it from our main application as a sub-process. This means that aside from the main Java version, our client also needs to have at least a run-time environment of Java 7 32-bit installed on their machine.

Interaction with MATLAB

The final change in the Java version was driven by the need to interact with the MATLAB engine. While the documentation suggests otherwise, the engine only works well with Java 8. Therefore, we had to downgrade again from Java 11 64-bit to Java 8 64-bit. This turned out to be the easiest change; not many alterations to the code base were needed. As an added bonus, Java 8 includes JavaFX as a package, eliminating the need for importing it separately.

4.2. Final Implementation

In the end the structure of our application closely mirrors the design we made. We have a main application, divided in a front-end and a back-end, as well as a ODBC-JDBC bridging JAR, a data sending VI, several

MATLAB files, and an external thermodynamics lirary. In the following sections the implementation of each component is described more in depth, highlighting areas we found particularly interesting or challenging.

4.2.1. Main application

As mentioned earlier, our main application consists of a front-end and a back-end. These two components communicate with each other through an *eventbus* which is explained further below.

Front-end

The front-end of our application is almost completely written in JavaFX. The application's starting point is the `Main` class, which loads the first screen through the `ScreenLoader`. This `ScreenLoader` is responsible for loading all other screens of the application, potentially in a new window. Since we wanted the same `ScreenLoader` instances to be accessible from all other front-end classes we decided to make it a singleton [7]. Figures G.1, G.2, and G.3 show the final look of the main menu screens.

While most of our GUI is implemented using standard JavaFX components, there were a few custom components we had to implement to meet the requirements of the client.

OrchidChart

One of the ways in which we wanted to facilitate the Ease of Use was by giving the client the option to interactively explore the data. In order to do so, we needed a chart on which we could plot the variables which the client could navigate by for example clicking and dragging, or by using buttons. While standard JavaFX *does* include a `LineChart` class [14] class which can be altered to be interactive, there were two ways in which it was too limited for our purposes.

Firstly, it is not possible to *smoothly* interact with the chart — it cannot be moved in small increments, only in bigger jumps. Secondly, it is not possible with a regular `LineChart` to plot dates on an axis in an attractive format - they must either be plotted as `Strings`, which inhibits proper data sorting, or as `Longs`, which is less readable. This was undesirable for us, seeing as the purpose of the chart was to plot the values of parameters versus time.

In order to add this functionality, we extended JavaFX's the `LineChart` and its supplementary `ValueAxis` class with our own `ScrollableLineChart` and `ScrollableNumberAxis` classes and added the necessary scrolling and plotting functionality.

Another feature our client required was the option to display different variables on different y-axes, for example temperature on one and pressure on the other. JavaFX does not allow a single chart to have two axes, however, it *is* possible to plot two charts on top of each other and make it *look* like they are one. We did just this, uniting the two charts in another class (`OrchidChart`) which both makes sure that the two "subcharts" remain synched up and handles the data that has to be displayed.

A simplified UML diagram of the final inheritance structure can be found in Figure G.4. Some images of an `OrchidChart` displaying data screen can be seen in Figure H.1.

TableView

Another challenging aspect was the table we wanted to use to display our variables (see the table on the right of Figure G.4). The standard way to create such a table in JavaFX is by using `TableView`. The check boxes in the table were meant to give the user the option to toggle certain variables, filters, or steady state detection on and off. However, `TableView` does not out-of-the box come with the ability to affect other parts of the application. We circumvented this restriction by creation our own `Event` types that the TableView could fire on its parent, the `OrchidChart}. The \verb`OrchidChart| then called the relevant methods on each of its sub charts. A sequence diagram showing this process can be found in Figure H.2.

Caching

The `LineChart` we used as a basis for our own chart implementations has a built-in data structure to keep track of the data being rendered on screen. Initially it seemed as if this data structure was efficient enough to render thousands of data points simultaneously without causing any significant delays. However, when we ran a performance test where we loaded an entire experiment's worth of data (upwards of 2.5 million datapoints) we decidedly hit the limit of what the GUI could handle.

It turned out that even when the data being displayed on the actual screen is limited to several thousands of data points, the application acts as if it is rendering every single data point it knows about. To work around this problem we implemented a very simple caching system. Instead of adding every data point to the `LineChart` straight away, we instead divided the full data set into "chunks" — lists of data points spanning a certain time. These chunks were put into a map where each key corresponded to the earliest data point in a chunk. Then, by observing the time the GUI *should* display, we could make sure that the only points the GUI would try to render were the points corresponding to that timespan.

Back-end

The back-end of our main application contains the logic needed to retrieve the data from the ORCHID and send it to the front-end to be displayed. It also contains functionality to communicate with MATLAB and to load/save data from/to a file. Below, we describe some particularly interesting components of the back-end.

ServiceLocator

When we started implementing our design we were not sure yet exactly what components we would end up needing in the end. Therefore, we thought it prudent to make our code base as extensible as possible.

In order to do this we made use of the service locator pattern. First, we divided as much as possible of the back-end functionality into separate modules. Each concrete module subclasses a module type. A central central class, the service locator, registers each specific module. A sequence diagram of the process of registering a module can be found in Figure H.3.

The main advantage of using the service locator pattern is that, because the service locator only relies on the overarching module interface, the specific implementations of each module can be changed without requiring extensive refactoring of the existing code base.

Eventbus

Early on in the implementation process we realized that we needed a way in which the front-end and back-end of our application could communicate. We could not simply use the main JavaFX thread to process all the incoming data, because this would make the application unresponsive while it was loading data. Therefore we decided that a multithreaded, asynchronous way of communicating would be more appropriate.

We knew that we only needed to communicate one kind of data — numerical data from the ORCHID — but the communication had to be reliable enough not to lose information. We decided to choose for an eventbus, a solution that is robust but not overly complex.

Briefly said, an eventbus works as follows: Objects can register to the eventbus as a subscriber to listen for certain kinds of events, which are posted by other objects at relevant times. The main advantage of this approach over a more standard Observer pattern[41] is that there is no coupling between Subjects and Observers. Instead of an observing class needing a reference to the object it is observing, it can simply observe the eventbus for the kind of Event it needs to react to. This way, the event can come from any source, and that source is free to change in the future with no refactoring needed on the Observer's part.

Another advantage of using an eventbus is that an event can have a payload, and as such can be used to send data. In this way, whenever new data was received from the ORCHID, it could be sent *with* the event signalling its existence.

Using an eventbus has some drawbacks as well. A common complaint is that having too many kinds of events can make the eventbus too crowded and the code difficult to understand. The impact of this disadvantage on our project was limited since we only had two kinds of data and subscribers — Real Time and Historical. Another disadvantage is that event-based code is more difficult to debug, since there is no way to control or know in which order events are sent or retrieved. While this meant part of our back-end could not be unit tested, it is a problem we would have had with other asynchronous methods of communication as well.

There are a number of libraries that implement an eventbus. We decided to use Green Robot's eventbus [39] because one of our group members had worked with it previously and had good experiences with it in terms of robustness and simplicity.

TCP client and VI

As mentioned before, during our research phase we decided that the best way to communicate real-time data from the ORCHID to our main application would be to create a VI that could send the data packets over a TCP connection.

The receiving TCP client was written in Java. Implementing this client was rather straightforward — we simply created a `Socket` which connects to the VI and reads and parses the data it receives on a separate thread. The data is then sent to the front-end through an eventbus event.

The more challenging aspect of the connection was creating the VI sending the data packets, as none of us had worked with LabView before. Luckily, a number of training courses are available from NI. Through the client, we also came into contact with the original developer of the control/monitoring/storage software who could then help us to get started. Additionally, LabView comes with a number of sample VI's that can be loaded as a template and then altered. One of these examples illustrated how to set up a TCP connection from a VI, significantly reducing the time it took us to do so ourselves.

Armed with our new knowledge we could create a VI which could run side by side with the original ORCHID software. It accesses the global variables from the control VI once every second and assigns them a timestamp. The timestamp and data value are then stored in an array and type cast to a "cluster" — a collection of data of different types. By doing so we could then add the variable names to each data point. The generated clusters were converted to JSON strings which could then be sent over a TCP connection.

Database client

Since we moved the actual database querying functionality to a separate JAR, we needed a way to run this JAR from our main application and retrieve its output. The way in which we do this is by starting the separate JAR as a process within our application and then pushing arguments to its input stream to make it query the database. The results from the query are then directed to the JAR's standard output. The retrieval of the data is done in a separate thread such as not to make the GUI unresponsive. When all the data has been retrieved, it is posted on the eventbus.

The bridging application itself is a very simple Java program. It only consists of a main method that opens a connection with the database, queries the database, goes through the result line by line, and writes each line to its standard output. The main complication in properly querying the database laid in figuring out the exact way in which the query should be made. While a guide is available [23], using the queries as they are described led to a variety of different exceptions. In the end, it was through trial and error that we determined a format that allows us to query a specific period in the Citadel 5 database.

The main disadvantage of using a separate JAR to query the database is that the process is significantly more difficult to debug. It is cumbersome to inspect any errors that might happen inside the JAR from the main application. However, as mentioned before, this was the only way in which we could integrate the functionality into our application without having to use Java 7 32-bit.

MATLAB

Once we retrieve the data from the ORCHID it is further processed by applying several MATLAB functions to it. The files containing these functions can be found in Appendix F. The functions were chosen and implemented in close communication with our client. Our reasoning for implementing the filters and steady state detection algorithm we did can be found in Chapter 7 of Appendix A.

The way in which the functions are called on the data is reasonably straightforward. When the application is started, so is the MATLAB engine. The engine has access to all files in a designated MATLAB folder. Calling a function is then done by using the name of the necessary file and the data to be evaluated.

One complication was that Matlab and Java use completely different data types. This mostly led to less elegant Java code, as we had to apply a number of operations for the data to adhere to the format required by MATLAB.

The way a user can apply the MATLAB functions from our application is simple: they navigate to either the real-time menu screen (see Figure G.3) or the historical menu screen (see Figure G.2). There they can select filters and steady state detection techniques from the drop-down menus. In the real-time menu screen they also have the option to click "Energy Balances" which displays a Piping & Instrumentation Diagram with the energy balances of relevant components.

Chapter 5: Testing

Testing is imperative for developing a solid application. It is a good way to ensure the code quality and to catch bugs before they cause problems. For our project, we relied on three types of tests: unit tests for the back-end and business logic of the application, integration tests for the interaction between the GUI and the back-end, and manual testing for the usability of the GUI. We describe our unit- and integration tests in Section 5.1, and our user tests in Section 5.2. We also discuss our reasons for leaving certain parts of our application *un*tested.

When working on a software product, especially one of this size, adding new code can at times have unexpected side effects for the existing code base. To make sure that new additions would not create bugs in the pre-existing code base we added the unit tests and integration tests to our CI build. As such, they were ran any time new code was pushed to the GitLab repository. In this way we could make sure that new code did not cause bugs in established parts of the software.

Our goal was to reach at least 80% test coverage over the entire application, excluding parts of the GUI. However, We did not achieve this goal rather a 76% test coverage was attained. This was mostly due to the usage of the ServiceLocator and the ModuleWrapper, which made our code very difficult to test. We did manage to test some modules that are abstracted away by the ServiceLocator through reflection.

5.1. Unit and Integration Testing

For our unit testing, we used JUnit 5 [42]. For the visualization of our test coverage, we used Jacoco [32]. Our final code coverage from Unit Tests was 76%, as can also be seen in Figure 5.1

We decided not to unit test the following packages of our application: - Our GUI package, as this is filled with user interface code that has to be executed on a JavaFx thread - The ScreenLoader class, for the same reason we did not test the GUI package. - The Main class, as all of this code will be tested when you actually start the application.

5.1.1. Integration Testing

While unit tests are well suited for testing the business logic of an application, they are less useful for testing how different parts of an application work together. Therefore, in order to test this aspect of the software we used integration tests with Mockito [37] instead. The Mockito tests brought our total test coverage to [INS] and even allowed us to test some parts of our front-end.

5.2. Manual Testing

The main method of testing the usability of our GUI was through manual testing. This approach had two goals: test how intuitive our application is to use, and how well it responds when displaying large amounts of data.

bep

bep

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
app		89%		100%	4	31	6	58	4	30	0	4
modules.backend.client		71%		61%	7	32	31	106	2	19	1	3
modules.backend.events		55%		n/a	2	5	5	11	2	5	1	3
Total	142 of 601	76%	10 of 28	64%	13	68	42	175	8	54	2	10

Figure 5.1: Our test coverage

The intuitiveness of our application was tested in close cooperation with our client. Our approach was to let the client use the application as often as possible and let them provide feedback on the look and feel. Since the application was being built specifically for our client to use, their preferences were the driving factor behind most of visual changes.

The other aim of our manual tests was track our program's performance. We wanted to test whether our application would still respond well if we added a large amount of data. We found that while our application can easily handle having over 2500000 data points in memory, rendering this many points in the GUI decreased the responsiveness drastically. This led us to make the changes described in Section 4.2.1, after which there was no significant dip in performance of our program, even with large datasets.

While manual testing was very useful to improve the quality of our application, it was also very time-consuming because loading in a large amount of data takes a long time. It is also not that useful for catching actual bugs, so it did not necessarily increase Reliability. Even so, it was useful for Ease of Use and Performance.

Chapter 6: Product evaluation

In this chapter we evaluate our final product. First, in Section 6.1, we evaluate which of our requirements we did or did not meet, and the reasons why. We evaluate how our application compares to the design goals we set, and discuss the feedback we received from the Software Improvement Group (SIG) in Section 6.2 and 6.3 respectively. We conclude the chapter with a short discussion and recommendations for future work in Section 6.4.

6.1. Evaluation of requirements

In this section we evaluate the requirements that were determined during the research phase (see Chapter 5 of Appendix A). The requirements that were *not* met can be found in Table 6.1, together with a brief explanation of why there were not met.

Looking at the table, there are two main reasons for why a certain requirement was not met: either we lacked the time to implement the feature, or we decided that a different approach was more appropriate. Even so, none of the requirements that were not met were classified as "Must-Haves" [8]. As such, even though we did not manage to implement every single feature, we feel like we have met the requirements we had set out to fulfill. In particular, we implemented all requirements related to the identification of the steady state — our client's top priority.

6.2. Evaluation of design goals

Here we evaluate whether our final product adheres to the design goals as outlined in Section 3.1.

Reliability Part of why our software was necessary was because the software the client was using would often crash when they tried to load in large data sets. When testing our application with a data set of over 2.5 million total points, no lags or crashes were recorded. As such, we feel that our goal of reliability has been met.

Ease of Use Our interpretation of this design goal was that the application should be easy and intuitive to use, and should especially facilitate exploration of the data. We feel like through close communication with the client we were able to tailor the software specifically to their requirements and as such meet this goal.

Performance Similar to the goal of Reliability, our confidence in the fact that we have achieved this goal comes from the fact that even when testing the application with a large data set, no significant lag was introduced. Additionally, loading in large data sets from the data base takes at most ten minutes, significantly shorter than the old applications the client was using.

6.3. SIG

The first evaluation from SIG can be found in Appendix E. The main criticisms were that there were no tests yet, and that some methods were too large, making them less maintainable. Overall our code still scored 3.5 out of 5 points. We tried to incorporate the feedback to the best of our abilities by splitting up large methods and creating test classes.

In the second evaluation from SIG, we can see that even though we added more code, the score has remained the same. We also managed to add some testing classes. On the other hand, while we removed some of the overly complex classes, other complex methods were added. Looking back, we could have made a stronger endeavor to remedy the issues raised in SIG's first evaluation.

6.4. Discussion and Recommendations

If we look at the final design from Section 3.3 and the implementations in Section 4.2, not many differences can be found. The only major change is that we sectioned off the database querying functionality and put it in

Requirement General Requirements	Explanation
The application must be written in Java 7, 32-bit version.	In the end our application was written mainly in Java 8 64-bit with a small part being written in Java 7 32-bit. See Section 4.1.3 for more details.
Raw sensor data - Data acquisition	
The application could support saving data to a .mat file.	Reading and writing to a file turned out to be more complicated than initially thought. Therefore, we focused on allowing the application to read and write to CSV files only.
The application could support saving processed data to other file formats than .csv.	Similar to the previous point, time did not permit us to implement this feature.
The user could be able to indicate their preferred file format.	Similar to the previous point, time did not permit us to implement this feature.
GUI	
The user should be able to select a part of the graph to zoom in into that specific subsection.	After conferring with the client it was decided that interacting with a graph in this manner is not the most intuitive way.
The user should be able to add more filters by entering the necessary MATLAB code in a text box.	Instead of entering code into a text box, the user can create more .mat files and add these to the proper folder.
The user should be able to pause a graph, meaning the graph will not be updated anymore until the graph is started again by the user.	It was decided that this feature did not add enough functionality compared to the time it would cost to implement it.
The data that is received while a graph is paused should be buffered. When the graph is unpaused, the buffered data must be rendered.	Similar to the previous point, we decided not to implement this feature.
The user could be able to choose the color of each variable in the graph.	It was decided that implementing this feature would be too time-consuming. Instead, each variable is assigned a random but consistent color.
Image data	
No requirements were met.	Time constraints did not permit us to process image data in addition to raw sensor data.

Table 6.1: This table contains all the requirements that were *not* met, as well as the reasons why they were not met.

a separated JAR. This illustrates the importance of a serious research phase. If we had not written the research report in Appendix A prior to starting development, it is likely that we would have run into significantly more trouble while implementing.

Although our design proved to be quite strong, our final implementation turned out more complex than strictly necessary. We attempted to make our software general and extensible by introducing for example a service locator and an eventbus, but did not make use of these aspects in every part of the software. Especially in the GUI some events are handled by the JavaFX event handling functionality that would have been better suited to the event bus.

From the fact that not all of the requirements we set for ourselves were it is clear that there is still quite some room for improvement for our application. Aside from processing the visual data, our application could mainly be improved by adding more Easy of Use related features. For example, the process of adding additional filtering algorithms could be simplified. The code base itself could also be improved further, for example by using the existing design patterns more consistently.

Chapter 7: Process Evaluation

In this chapter we evaluate our development process. We describe our development methodology in Section 7.1, followed by challenges we encountered in Section 7.2.

7.1. Development Methodology

Software development is a complex process, so a certain plan of action is needed to steer it in the right direction. As we only had seven weeks available for developing the actual software, we initially we planned to use the SCRUM method[40]. In this method you work on a project in rapid iterations and you meet often. Our aim was to have sprints of one week each, and decide each week what we would do in each sprint. Also wanted analyse at the end of each sprint to see what we could improve.

While earlier in the project we adhered quite strictly to the structure of a typical SCRUM project, later on we let go a bit of this structure. We found that people worked better if everyone picked a task to work on, communicated their choice to the other team members, and then accomplished that task. By dividing the labor in this way we increased individual accountability.

One way in which our development process was different from a typical BEP was that our client was very involved with the development of the application. Members of the development team met with the client multiple times per week. This gave the client to closely monitor the application as it was being developed, as well as provide useful insights and input.

7.1.1. Version management

During this project we used GitLab [6] to manage our source code and track our issues. We also set up Continuous Integration (CI) which would make sure our builds would fail if the code quality was not up to standard or if a test would fail. We used Checkstyle and PMD for static code analysis.

We also used GitLab for code reviewing. Each merge request that was made had to be approved by at least two other team members. In this way, we both made sure that everyone was up to date with the code base and ensured code quality.

7.2. Challenges

Quite early on we found that, while good in theory, the SCRUM method of software development did not really work for us during this particular project. Especially since people had other obligations, such as exams or jobs, as well, meeting every day proved to be realistic. Additionally, we felt that making a sprint review and sprint plan every week took time that would be better spent developing.

Sadly, the fact that we did not keep up with our initial plan also meant that the intra-team communication suffered, causing people to not know what other people were doing at times. This also decreased the amount of accountability — at times people neglected tasks until reminded. Meeting less often face-to-face also decreased the number of opportunities for brainstorming and pair-programming to solve programming issues.

Even though our efficiency and process as a whole suffered because of the lack of structure, we *had* agreed on a certain division of tasks beforehand. Because everyone took their own task seriously, the most important parts of our software were continuously being worked on, even when communication faltered. Especially during the last few weeks of the project we were able to pick back up where we had left off. In this time we created a prioritized list of goals that still *had* to be achieved, and in the end we managed.

7.2.1. Recommendations

We feel as if there are ways in which we could improve our process in future projects. For example, we would try to adhere more strictly to the SCRUM methodology. While the documentation surrounding the method can be time consuming, in hindsight we believe that making the documents would have saved us time in the long run because we would have been able to work more efficiently. Meeting regularly also increases the opportunity for creative brainstorming and keeps people accountable.

Something we already did, but would definitely recommend for the future is consulting the client as much as possible, especially if they are as involved as our client was. Since the client is the person who has to use the software in the end their opinion is invaluable.

Chapter 8: Conclusion

The goal of this project was to develop a software application that could help our client to make better use of the unique data sets generated by the ORCHID, an experimental power plant used to study among other things the Organic Rankine Cycle. The main issue the client had with the applications they were using was that they would often lag and crash, and that they did not offer a convenient way to detect the steady state of the ORCHID in historical data, and no way *at all* in real-time data.

The main challenge in developing our application was that it had to interact with many different components: it had to retrieve historical data from a Citadel 5 database, retrieve real-time data over a TCP/IP connection, and process the data by interacting with MATLAB. Additionally, it had to manage this interaction while being reliable, easy to use, and fast.

After seven weeks of development our application is able to visualize both large historical data sets as real time data. More importantly, we are able to detect the steady state of the system, both in real-time as for historical data. Since this was the main requirement set by our client, the project was successful.

Appendix A: Research Report

Appendix B: Introduction

The Organic Rankine Cycle Hybrid Integrated Device (ORCHID) is a small scale power plant, designed by our client and used to study the fundamental gas dynamic behavior of dense organic fluids, as well as the behavior of turbomachinery. It currently uses a number of applications written in LabVIEW[1], a graphical programming language created by National Instruments (NI)[2], for data acquisition and analysis. In order to draw conclusions about the ORCHID's measurement data, it is important that the system's steady state can be identified. The steady state of the system is reached when relevant variables describing the internal processes are constant over time [2], and identifying whether or not the ORCHID has reached this state is currently not possible to do in real-time. Additionally, the current applications are outdated and are no longer maintained, leading to frequent crashes.

In our bachelor end project, we will build an application from scratch that acquires the data from the ORCHID and filters, analyses and visualizes it. It will also be able to identify the steady state real-time.

This report is structured as follows. In chapter 2, we discuss the ORCHID in more detail and describe the data acquisition and data processing as it is currently done. In chapter 3, we state our problem definition and discuss some of the ways in which our application will improve the current setup. In chapter 4, we talk about the different design goals and how we aim to achieve them. In chapter 5, we give our requirements in the MoSCoW style [8] for the general application, GUI, and post-processing methods. In chapter 6, we explain what development tools we will use during our project. In chapter 7, we describe how we will filter and smooth the ORCHID's data, as well as how we will identify the steady state of the system, as well as the reasons for choosing each specific method. Chapter 8 explains how our application will acquire data from the ORCHID, both from a database as well as in real-time. In chapter 9, we discuss the different languages and frameworks we will use for our project. The chapter considers the main programming language, the framework for the GUI, the language used for the data post-processing functions, and the database in which the ORCHID stores its data. Our conclusion will be given in chapter 10.

[1] http://www.ni.com/nl-nl/shop/labview.html
[2] http://www.ni.com/

Appendix C: Infosheet

General Information

Title of the project: Data acquisition and processing application for the ORCHID **Name of the client organization:** Propulsion & Power Aerospace Engineering, Delft University of Technology **Date of the final presentation:** 1-02-2019

The final report for this project can be found at: http://repository.tudelft.nl.

Description

The ORCHID is an experimental power plant that models the Organic Rankine Cycle. In order to draw conclusions about the data gathered by the ORCHID, it is important to know when the system is in steady state. In order to make efficient use of both the client's time, as well as of the energy required to operate the ORCHID, the steady state has to be detected while an experiment is being conducted. However, this is currently not possible.

During the research phase it became clear that it was not possible to implement all the necessary data processing features in pure Java. Therefore, the main challenge of this project was to find a way properly connect the different applications needed to process the data while keeping sight of our main design goals: Reliability, Performance, and Ease of Use.

Since we only had seven weeks available to develop the application — not counting the two weeks of preliminary research — we decided to use the SCRUM methodology. During the development process the communication between team members became a bit less frequent, which was detrimental to the SCRUM process. During the last three weeks of the project we made a concerted effort to improve our communication and stick to our goals.

The final product is a central application written in Java with an intuitive GUI. The client can either query a database or connect directly to the ORCHID to retrieve and analyze their data — all the interaction with external components is handled in the back-end. The application was tested through unit tests, integration tests, and manual tests, the latter in close cooperation with the client.

The final product answers the client's main need: the ability to identify the steady state of the ORCHID during an experiment. The pro However, there are some ways in which the Ease of Use of the product could be improved, for example by allowing the user to add more kinds of filters and steady state detection algorithms, as well as by processing more kinds of data.

Members of the project team

Name: Stefan van der Heijden *Interests: Contribution and role:* Developer, steady state and filtering research, MATLAB functions

Name: Stas Mironov *Interests: Contribution and role:* Developer, code modularity, LabView interaction, TCP/IP VI

Name: Nils Hullegien *Interests:* Chess, swimming, beer brewing *Contribution and role:* Developer, GUI,

Name: Maaike Visser *Interests:* Weightlifting, woodworking, jewellery making *Contribution and role:* Developer, GUI, database interaction, final report

All members of the development team contributed to the research report.

Client

Name: Adam J. Head *Affiliation:* Propulsion & Power Aerospace Engineering, Delft University of Technology

Supervisor

Name: Jan Rellermeyer *Affiliation:* Distributed Systems, Delft University of Technology

Contact person

Maaike Visser, mebpvisser@gmail.com

Appendix D: Project description

Appendix E: SIG evaluations

E.1. First evaluation

[Analyse]

De code van het systeem scoort 3,5 sterren in ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Unit Size en Unit Complexity.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes.

In jullie project is ClientDB.query() een goed voorbeeld. Dat is een heel proces dat nu binnen één method wordt geïmplementeerd. Zo'n aanpak wordt vanuit het oogpunt van onderhoudbaarheid problematisch op het moment dat de functionalteit gaat toenemen. Voor ClientTCP.connect() geldt hetzelfde, en daar valt het op dat de implementatie vrij low-level is. Sinds Java 7 is de code die jullie nu in het finally-blok hebben staan niet meer nodig en kan worden vervangen door een try-with-resource blok. Als laatste voorbeeld kun je FileSystemAdapter.saveChartDataDefault op twee manieren aanpassen: of je splitst de functionaliteit op in aparte sub-methodes, of je maakt een nieuwe datastructuur-class die de huidige TreeMap<String, TreeMap<String, Number» vervangt.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een descriptieve naam kan elk van de onderdelen apart getest worden en wordt de overall flow van de methode makkelijker te begrijpen.

De methodes die hier naar voren komen lijken sterk op die bij Unit Size, en je zou door een verbeter verdeling van functionaliteit over de methodes dus beide problemen tegelijk kunnen verhelpen.

Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen.

Over het algemeen scoort de code dus bovengemiddeld, maar zijn er ook nog wel mogelijkheden tot verbetering. Hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

E.2. Second evaluation

In de tweede upload zien we dat het codevolume is gestegen, terwijl de score voor onderhoudbaarheid met 3,5 sterren ongeveer gelijk is gebleven.

Bij Unit Size en Unit Complexity, de gebieden die in de feedback op de eerste upload als verbeterpunt aangemerkt werden, zien we weinig structurele verbetering. De voorbeelden die eerder werden genoemd zijn weliswaar niet meer in de code aanwezig, maar omdat jullie in de nieuwe code ook weer nieuwe en complexe methodes hebben toegevoegd zijn die verbeteringen grotendeels weer teniet gedaan. Een voorbeeld daarvan is OrchidChart.addCachingListeners(), waar een hele beslisboom in één methode gescheiden door commentaar is geïmplementeerd.

Wel is het positief dat jullie naast nieuwe productiecode ook nieuwe testcode hebben geschreven. Het is ook gelukt om een deel van de achterstand in testcode voor de "oude" code weg te werken. De hoeveelheid testcode blijft nog wel wat achter, maar als je kijkt naar de achterstand die jullie hadden viel dat ook te verwachten. Voor de nieuwe code wordt er duidelijk meer "test-driven" gewerkt dan eerst, waardoor de voordelen van de tests al tijdens de ontwikkeling kunnen worden benut.

Uit deze observaties kunnen we concluderen dat de aanbevelingen deels zijn meegenomen tijdens het ontwikkeltraject.

Appendix F: MATLAB files

```
function done = PhD_Startup(~)

clc;
clear all;

addpath(genpath(cd)) %genpath: generate path string p = genpath(folderName)
                     %addpath: adds a path string to the top of the search path
                     %addpath('C:\Users\LocalAdmin\IdeaProjects\orchid')


% This script must be run to initialize all fluidprop instances of the interested fluids. It
% aslo allows to set different thermodynamic libraries.

Init_FluidProp;

% Set the libraries. These correspond to the various equations. Each
% library corresponds to an equation.

Set_Refprop; %More accurate
% Set_StanMix; %Faster

done = 'done';

end
```

[caption=The function that is called when starting the MATLAB engine. Initializes FluidProp.]

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     Init_FluidProp       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc
% For RefProp Mixtures
global MM_RP Water;
% For StanMix Mixtures
global MM_SM;


Water = actxserver ('FluidProp.FluidProp');

MM_RP = actxserver ('FluidProp.FluidProp');

MM_SM = actxserver ('FluidProp.FluidProp');
```

[caption=The function initializing FluidProp.]

```
%function output = EnergyBalance(sc, dp, da, db, c, diffp, m_dotevp, m_dotbp, avg1, vp12, lfdk,
```

```
function output = EnergyBalance(a)


% This function calculates the energy balances accross all the hardware in
% the ORCHID facility.

global MM_RP MM_SM Water;

% Select the fluid of interest
Fluid = {MM_RP Water};

%Measurement Data
%% HTT Unit


        %%Differential Pressure Venturi. ALWAYS FIXED SAME
        %dpv = 0.33; % MUI - Manual User Input
        %Suction Pressure [barg]
        %sc = 4;      % MUI - Manual User Input
        sc = a(1,1);
        %Discharge Pressure [barg]
        %dp = 7;      % MUI - Manual User Input
        dp = a(1,2);
        %Pipe diameter
        %da = 3; %[inch] % MUI - Manual User Input
        da = a(1,3);
        %Venturi neck diameter
        %db = 1.43; %[inch] % MUI - Manual User Input
        db = a(1,4);
        %Discharge coefficient
        %c = 0.98;  % MUI - Manual User Input
        c = a(1,5);
        %%Controller Percentage [%]
        %cp = 32;   % MUI - Manual User Input
        %Differential Pressure. DELTA P
        %diffp = dp - sc;   % MUI - Manual User Input
        diffp = a(1,6);
        %HTT Heater Temperature PV [C]
        tthttpv = a(1,14); % SI - Sensor Input (HTT - PV)

        %HTT Therminol VP1 Oil density [kg/m3]
        rhohtt = -0.9097*tthttpv+0.00078116*tthttpv^2-(2.367*10^-6)*tthttpv^3+1083.25; % OT - Output t

        %Head [m]
        h = diffp*10^5/9.81/rhohtt; % OT

        %%%%%%%%%%%%%%%%%Changes between files%%%%%%%%%%%%%%%%%%%%%%%%Volumetric flow [m3/hr]
        vfhtt = c*sqrt((2*diffp/rhohtt)*(pi*(0.5*da)^2))/(sqrt(((pi*(0.5*da)^2)/(pi*(0.5*db)^2))^2-1))

        %HTT Mass flow rate [kg/s]. Same as Qmass in https://www.efunda.com/formulae/fluids/venturi_flo
        m_dothtt = vfhtt*rhohtt/3600;
        %TT003 [C]
        tt003 = a(1,15); % SI - Sensor Input
        %TT004 [C]
        tt004 = a(1,16); % SI
        %TT002 [C]
```

```
tt002 = a(1,17); % SI
%Mass flow evaporator on the heating loop [kg/s]
%m_dotevp = (tt003-tt002)/(tt003-tt004)*m_dothtt; % MUI
m_dotevp = a(1,7);
%Mass flow by-pass [kg/s]
%m_dotbp = m_dothtt - m_dotevp; % MUI
m_dotbp = a(1,8);
%Cp Therminol VP1 oil [kJ/kg-K]
cpt = 0.002414*tt003+5.9591*10^-6*tt003^2-2.9879*10^-8*tt003^3+4.4172*10^-11*tt003^4+1.4
% OUTPUT: Thermal Power by-pass [kW]
w_dotbp = m_dotbp*cpt*(tt003-tt002); % OP - output picture TF0
% OUTPUT: Thermal Power given to the evaporator [kW]
w_dotevp = cpt*(tt003-tt004)*m_dotevp; % OP - output picture TF1
%Average Current [A] L1
%avg1 = 278.9;                    % MUI
avg1 = a(1,9);
%Average Voltage [V] L1
%vp12 = 400;                      % MUI
vp12 = a(1,10);

% OUTPUT: Electric Power [kW] L1
w_dotel = avg1*vp12*sqrt(3)/1000;  % OP TF-1 put it under the label of the heater
%%MOV006a [%]
%mova = 46.2;        %SI
%%MOV006b [%]
%movb = 46.6;        %SI
%OUTPUT: Thermal / Electric Power difference (Perhaps think about
%putting this value--since no attributed physical location, and
%those of the T, P, m in a table aside to the P&ID.
tepd = (w_dotel-w_dotevp)/w_dotevp; %The difference in electric power
% to the thermal power. These should be equal in steady state and
% thus 0. %OT



%% Working fluid (MUI: Indicate by user. SI. Sensor Input)
%MUI. Local Fluid Density Krohne [kg/m3]
%lfdk = 718.3;
lfdk = a(1,11);
%%MUI. Local Fluid Temperature Krohne [C]
%lftk = 63.9;
%SI. Mass flow rate Kronhe [kg/h]
mfrkh = a(1,18);
%OT. Mass flow rate Kronhe [kg/s]
mfrks = mfrkh/3600;
%MUI. Max RPMs Diaphragm Pump [rpm]
%rpmdp = 1050;
rpmdp = a(1,12);
%MUI. Max Volumetric flow diaphragm pump [L/min]
%mvfdp = 136;
mvfdp = a(1,13);
%SI. DAQ RPMs VSD [rpm] look in list for tag name...
rpmvsd = a(1,19);
%%SI. Control Frequency VSD
%cfvsd = 0.247;
%SI. TT001 [C]
```

```
        tt001 = a(1,20);
        %%SI. PT008 [barg]
        %pt008 = 9.4;
        %SI. PTA003 [barg]
        pta003 = a(1,21);
        % Predicted fluid density [kg/m3]
        [sp_vol_001,ErrorMsg] = invoke(Fluid{1},'SpecVolume','PT',pta003+1,tt001);
        ErrorCheck(ErrorMsg);
        rho_001 = 1/sp_vol_001; % OT - rho001
        %OT. Krohne Density Measured Vs. Predicted
                kdmvsp =(rho_001-lfdk)/lfdk;
        % Volumetric flow main pump [L/min]
        vfmpl = rpmvsd/rpmdp*mvfdp;
        %OT Volumetric flow main pump [m3/min]
        vfmpm3 = vfmpl/1000;
        % OT Mass flow pump [kg/min]
        mfpmin = vfmpm3*rho_001;
        % Mass flow pump [kg/hr]
        mfphr = mfpmin*60;
        % OT Mass flow pump [kg/s]
        mfps = mfpmin/60;
%Balances
        %OT. Mass Flow Rate Krohne and Pump Difference
        mfrkpd = (mfphr-mfrkh)/mfrkh;
        %SI TT005
        tt005 = a(1,22);
        %SI PZA003 [barg]
        pza003 = a(1,23);
        %SI TT006
        tt006 = a(1,24);
        %SI PT002
        pt002 = a(1,25);

        %%Tsat. Saturation Temperaure
        %[tsat,ErrorMsg] = invoke(Fluid{1},'Temperature','Pq',pt002+1,1);
        %ErrorCheck(ErrorMsg);

        %h_in evap. enthalpy in the evaporator [J/kg]
        [hinevp,ErrorMsg] = invoke(Fluid{1},'Enthalpy','PT',pt002+1,tt006); % Enthalpy
        ErrorCheck(ErrorMsg);

        %h_out evap. enthalpy out the evaporator [J/kg]
        [houtevp,ErrorMsg] = invoke(Fluid{1},'Enthalpy','PT',pza003+1,tt005); % Enthalpy
        ErrorCheck(ErrorMsg);

        %OP TF2 - Evaporator Power MM side [kW]
        pinevp = ((houtevp-hinevp)*mfrks)/1000;
        %OT - Power difference over evaporator [%] (Power in - Power out)/Power in
        pipo = ((w_dotel-pinevp)/w_dotel)*100;
        %h_in cold side regenerator [J/kg]
        [hicsr,ErrorMsg] = invoke(Fluid{1},'Enthalpy','PT',pta003+1,tt001); % Enthalpy
        ErrorCheck(ErrorMsg);
        %%Check with local temp and quality
        %[cltq1,ErrorMsg] = invoke(Fluid{1},'Enthalpy','Tq',tt001,0); % Enthalpy
        %ErrorCheck(ErrorMsg);
        %h_out cold side regenerator [J/kg]
```

```
[hocsr,ErrorMsg] = invoke(Fluid{1},'Enthalpy','PT',pt002+1,tt006); % Enthalpy
ErrorCheck(ErrorMsg);
%%Check with local temp and quality
%[cltq2,ErrorMsg] = invoke(Fluid{1},'Enthalpy','Tq',tt006,0); % Enthalpy
%ErrorCheck(ErrorMsg);
%OP TF3 Thermal load [kW]
tl1 = ((hocsr-hicsr)*mfrks)/1000;
%%Tsat
%[tsat2,ErrorMsg] = invoke(Fluid{1},'Temperature','Pq',pta003+1,1);
%ErrorCheck(ErrorMsg);
%SI TT007
tt007 = a(1,26);
%SI PZA007
pza007 = a(1,27);
%SI TT008 [C]
tt008 = a(1,28);
%SI PT005 [barg]
pt005 = a(1,29);
%h_in hot side regenerator [J/kg]
[hihsr,ErrorMsg] = invoke(Fluid{1},'Enthalpy','PT',pza007+1,tt007); % Enthalpy
ErrorCheck(ErrorMsg);
%h_out hot side regenerator [J/kg]
[hohsr,ErrorMsg] = invoke(Fluid{1},'Enthalpy','PT',pt005+1,tt008); % Enthalpy
ErrorCheck(ErrorMsg);
%%Tsat
%[tsat3,ErrorMsg] = invoke(Fluid{1},'Temperature','Pq',pt005+1,1);
%ErrorCheck(ErrorMsg);
%OP TF4 Thermal load [kW]
tl2 = ((hohsr-hihsr)*mfrks)/1000;
%OT Power difference over regenerator [%] (Cold + Hot)/Cold
ch = ((tl1+tl2)/tl1)*100;
%SI TT009 [C]
tt009 = a(1,30);
%SI PT006 [barg]
pt006 = a(1,31);
%h_in hot side condensor [J/kg]
hihsc = hohsr;
%h_out hot side condensor [J/kg]
[hohsc,ErrorMsg] = invoke(Fluid{1},'Enthalpy','PT',pt006+1,tt009); % Enthalpy
ErrorCheck(ErrorMsg);
%%h_out cold side condensor [J/kg] using quality
%[hocscq1,ErrorMsg] = invoke(Fluid{1},'Enthalpy','Tq',tt009,0); % Enthalpy
%ErrorCheck(ErrorMsg);
%%Tsat
%[tsat4,ErrorMsg] = invoke(Fluid{1},'Temperature','Pq',pt006+1,1);
%ErrorCheck(ErrorMsg);
%OP TF5 Thermal load [kW]
tl3 = ((hohsc-hihsc)*mfrks)/1000;
%OT System Power Balance (Evaporator Power in + Condensor Power out) /Power in
epicpo = ((pinevp+tl3)/pinevp)*100;


%% Cooling loop

%SI TT011 [C]
tt011 = a(1,32);
```

```
        %SI PTA002 [barg]
        pta002 = a(1,33);
        %SI TTA010 [C]
        tta010 = a(1,34);
        %h_in cold side condensor [J/kg]
        [hicsc,ErrorMsg] = invoke(Fluid{1},'Enthalpy','PT',pta002+1,tt011); % Enthalpy
        ErrorCheck(ErrorMsg);
        %%h_in cold side condensor [J/kg] using quality
        %[hicscq,ErrorMsg] = invoke(Fluid{1},'Enthalpy','Tq',tt011,0); % Enthalpy
        %ErrorCheck(ErrorMsg);
        %h_out cold side condensor [J/kg] using quality
        [hocscq2,ErrorMsg] = invoke(Fluid{1},'Enthalpy','Tq',tta010,0); % Enthalpy
        ErrorCheck(ErrorMsg);
        %SI Volume flow rate [L/min]
        vfr = a(1,35);
        %OT Volumetric flow coolant pump [m3/min]
        vfmp = vfr/1000;
        % Eurol -27 Density [kg/m3]
%         ed = 1080;
        %Predicted density water
        [pvw,ErrorMsg] = invoke(Fluid{2},'SpecVolume','Tq',tt011,0);
        ErrorCheck(ErrorMsg);
        pdw = 1/pvw; % OT
        % Mass flow pump [kg/min]
        mfpclmin = vfr*vfmp;
        %OT Mass flow pump [kg/s]
        mfpcls = mfpclmin/60;
        %OP TF6 Thermal load [kW]
        tl4 = ((hocscq2-hicsc)*mfpcls)/1000;
        %OT Power difference over condensor [%]
        cond = ((tl3+tl4)/tl3)*100;
%         %PI02
%         pi02 = 9999;
%         %PI03
%         pi03 = 9999;
%         %TI04
%         ti04 = 9999;
outputTable = [rhohtt, h, cpt, tepd, mfrks, rho_001, kdmvsp, vfmpm3, mfpmin, mfps, mfrkpd, pipo, ch, e
outputPicture = [w_dotel, w_dotbp, w_dotevp, pinevp, tl1, tl2, tl3, tl4];
output = [outputTable, outputPicture];
end
```

[caption=The function used to calculate the energy balances over components of the ORCHID.]

```
%function [currentValue, v2fi, xfi, d2fi, R] = SteadyState2(currentValue, previousValue, v2fi, xfi, d2
function output = SteadyState2(a)

    currentValue = a(1,:);
    previousValue = a(2,:);
    v2fi = a(3,:);
    xfi = a(4,:);
    d2fi = a(5,:);
    lambda1 = a(6,:);
    lambda2 = a(7,:);
    lambda3 = a(8,:);

    %Method 1 to calculate the variance using xfi which is an average of
```

```
    %past datapoints.
    v2fi = lambda2.*(currentValue - xfi).^2 + (1-lambda2).*v2fi; %Equation 9

    %Exponentially weighted moving average.
    xfi = lambda1.*currentValue + (1-lambda1).*xfi; %Equation 2

    %Method 2 to calculate the variance, using only the previous datapoint.
    d2fi = lambda3.*(currentValue - previousValue).^2 + (1-lambda3).*d2fi; %Equation 13

    %Compare the two variances in such a way that when the sensordata
    %doesnt change much in time the R is close to 1. When the sensordata
    %starts changing
    R = ((2-lambda1).*v2fi)./d2fi; %Equation 15

    %Compare d2fi * Rcrit / (2- lambda1)*v2fi (prevents possibility of
    %divide by 0)
    %R_new = (d2fi .* R_thresh) / ((2- lambda1).*v2fi); %Prevents divide by zero
    output = [currentValue; v2fi; xfi; d2fi; R];
end
```

[caption=The function containing the steady state detection algorithm.]

Appendix G: Screenshots

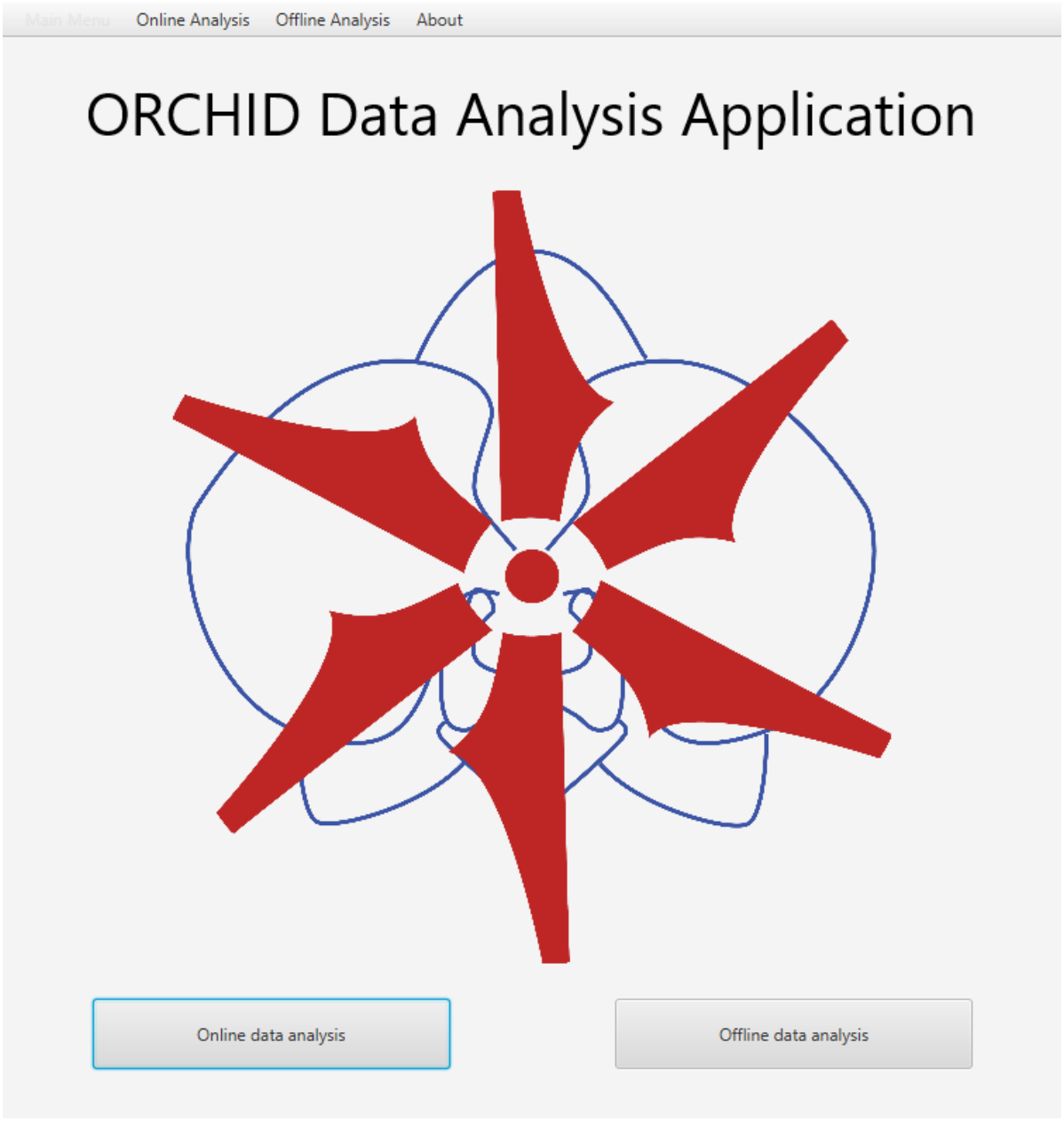

Figure G.1: The application's main menu.

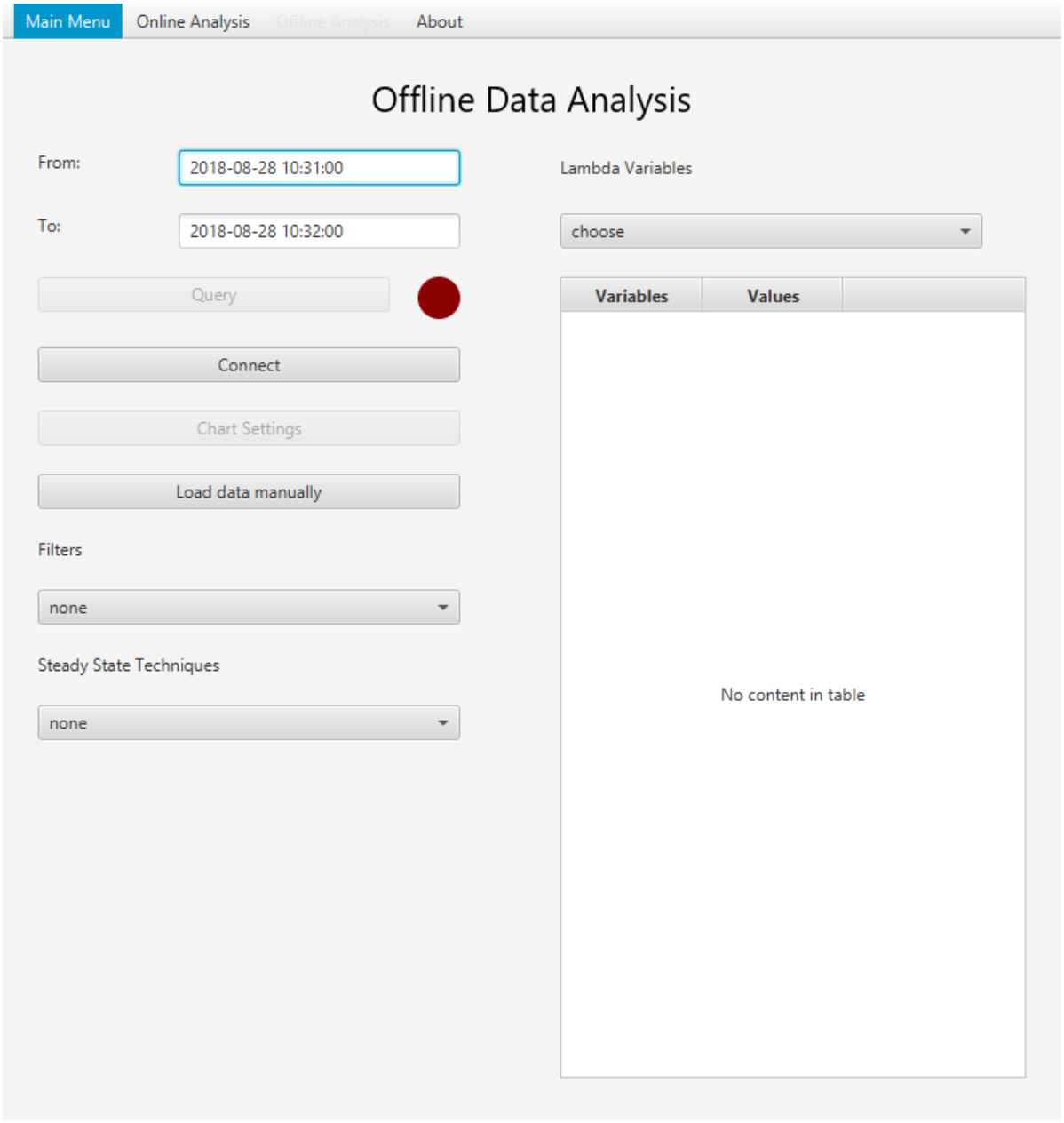

Figure G.2: The application's historical data menu. From here the database can be queried.

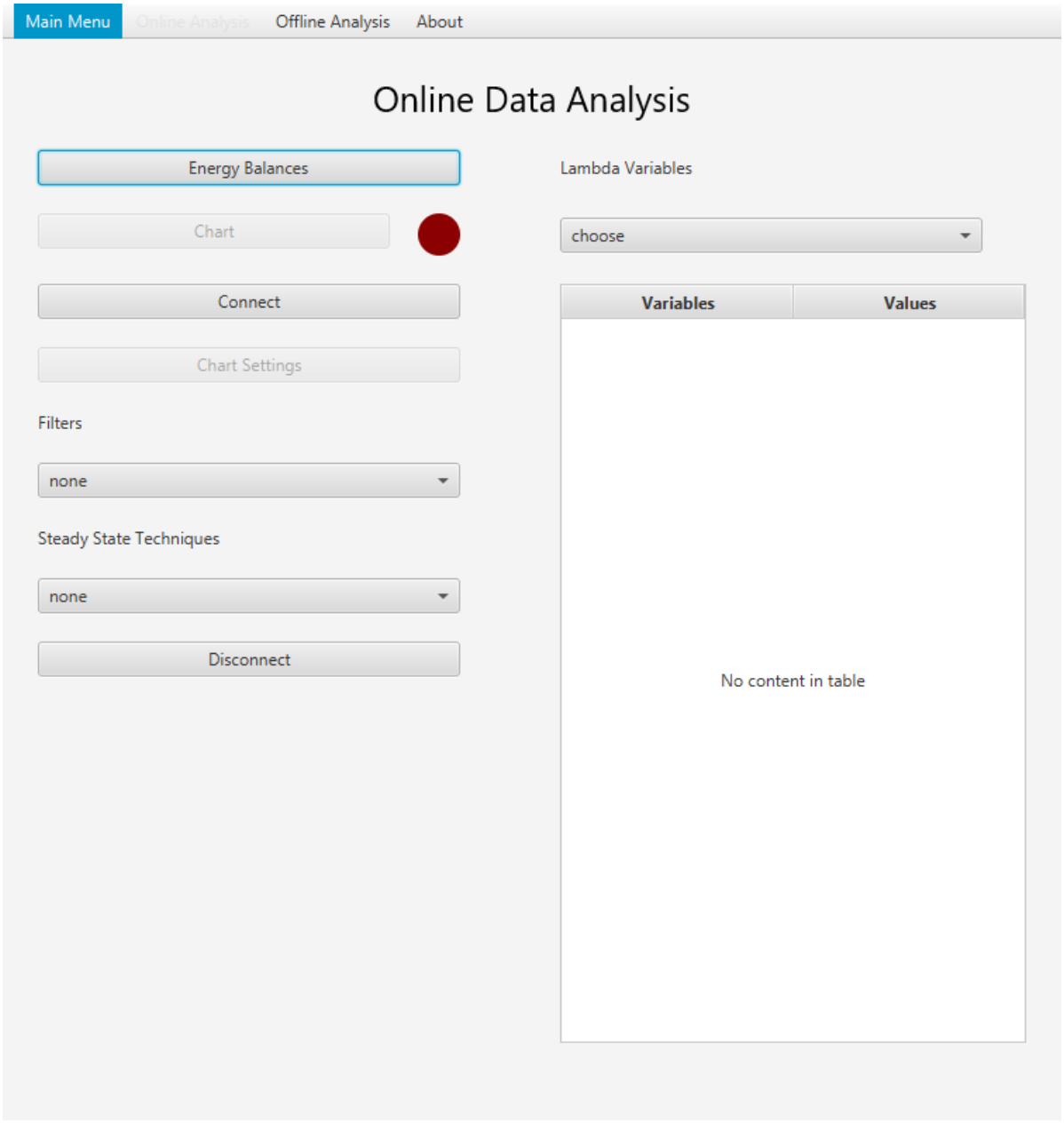

Figure G.3: The application's real-time data menu. From here a connection with the VI sending the real-time data can be made.

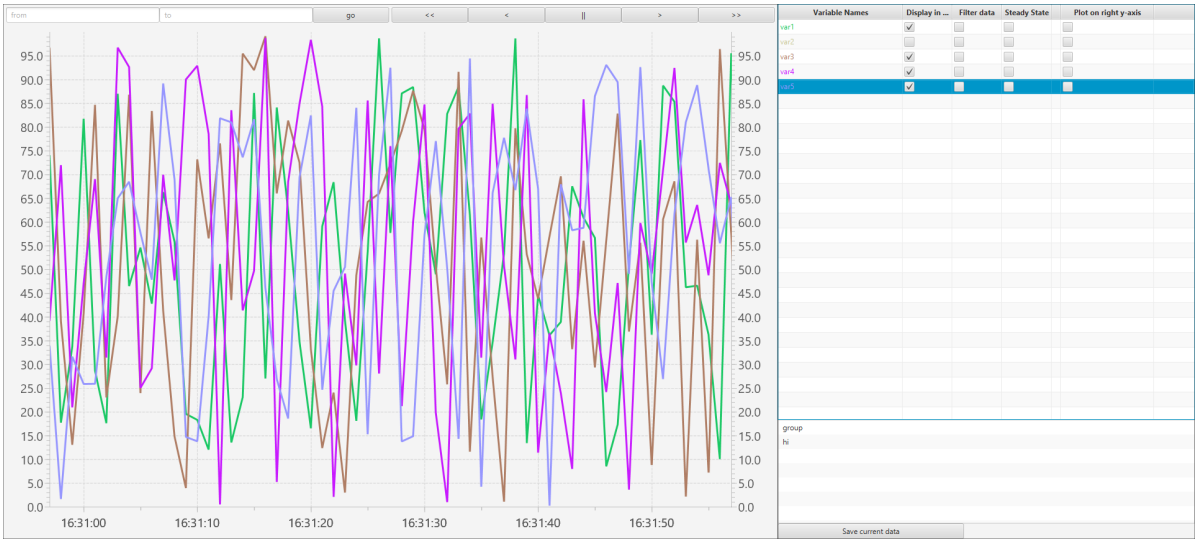

Figure G.4: The application's main chart screen, here filled with dummy data. On the right the user can select which variable they would like to display.

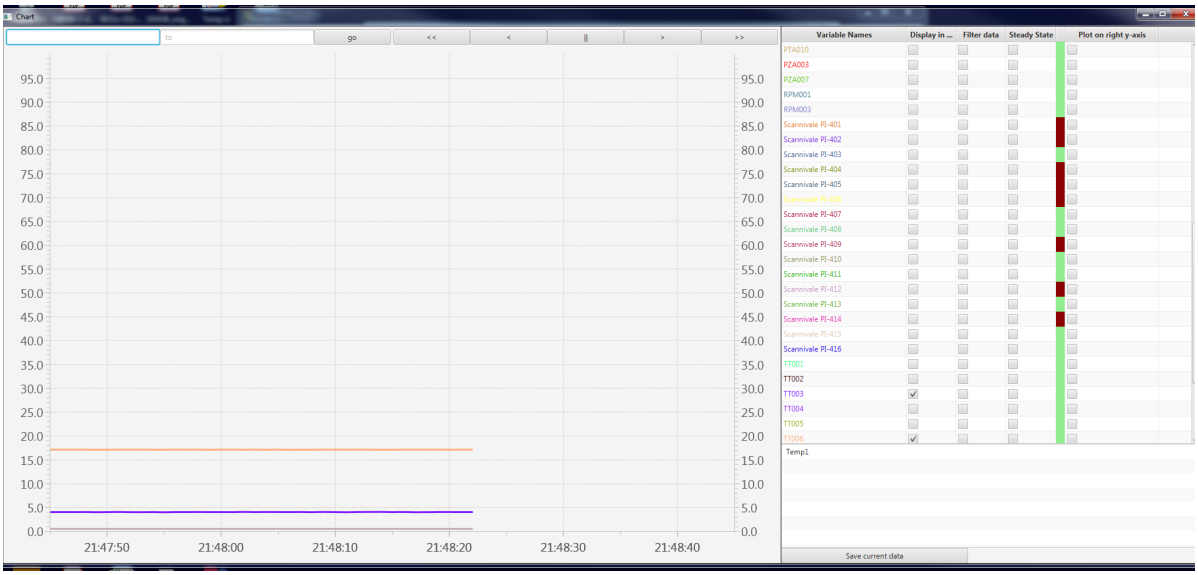

Figure G.5: The real-time chart showing a green light for a variable that is in steady state and a red light for a variable that is not.

Appendix H: UML and sequence diagrams

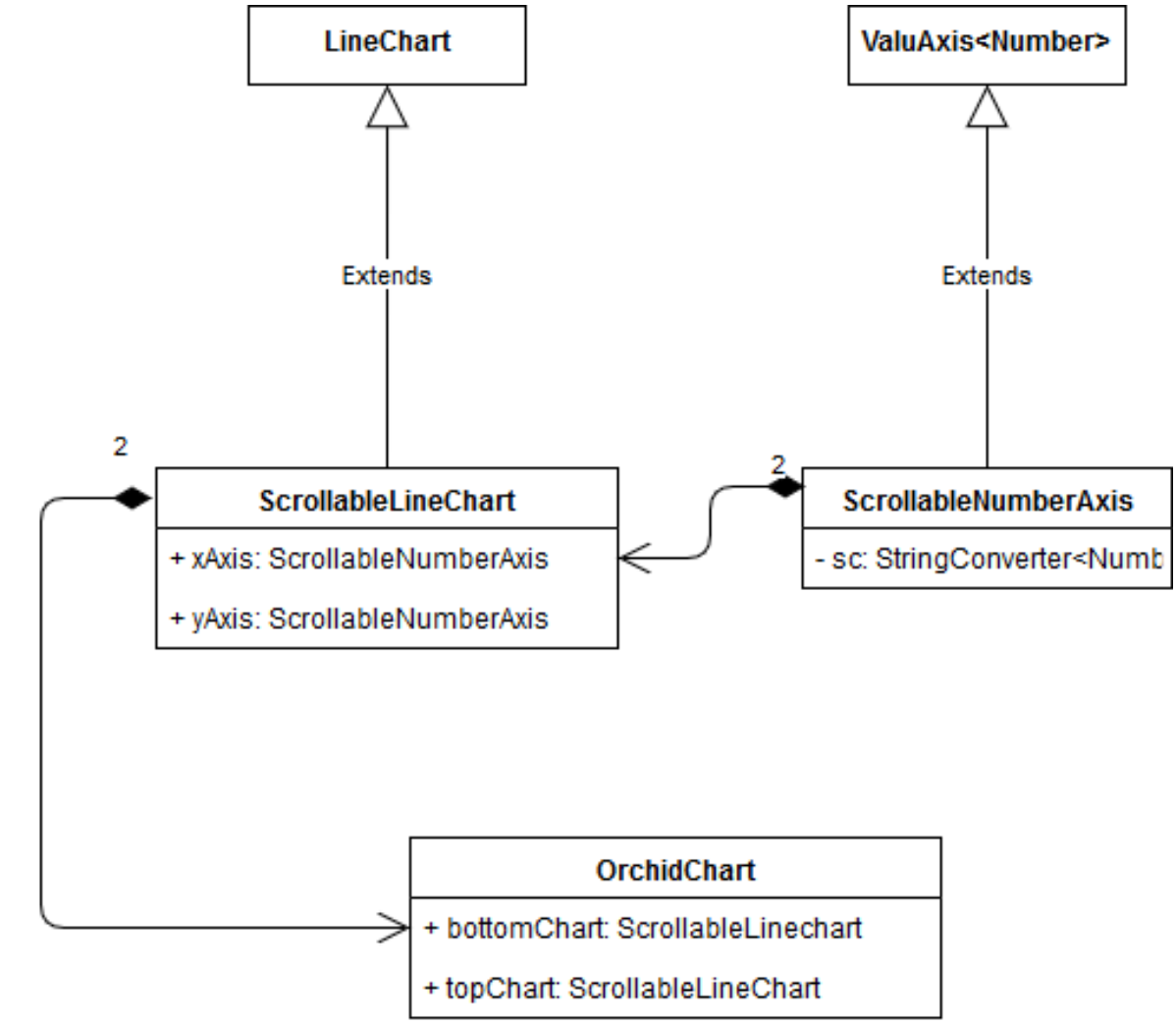

Figure H.1: A simplified UML diagram of the inheritance relation between JavaFX's LineChart and ValueAxis classes, and our own classes adding more functionality.

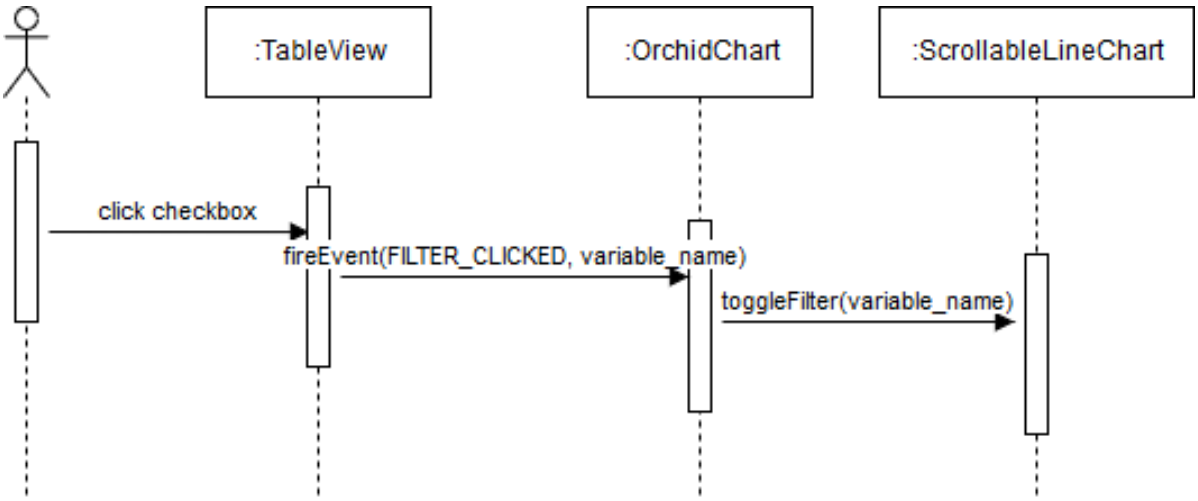

Figure H.2: A sequence diagram showing the cascade of method calls resulting from a checkbox being clicked in the TableView showing all the variables in the chart.

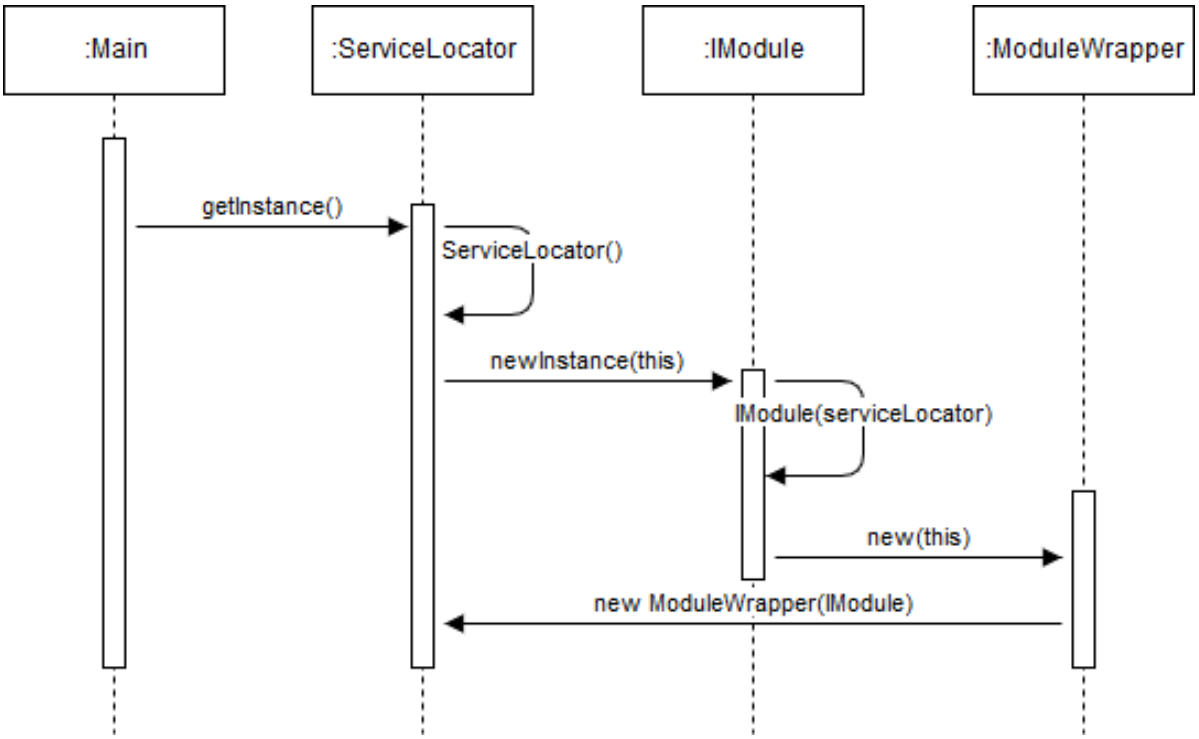

Figure H.3: A simplified sequence diagram showing the way in which the service locator initializes a module.

Bibliography

[1] Asimptote. Fluidprop, 2019. URL http://www.asimptote.nl/software/fluidprop.

[2] Songling Cao and R.Russell Rhinehart. An efficient method for on-line identification of steady state. *Journal of Process Control*, 5(6):363 – 374, 1995. ISSN 0959-1524. doi: https://doi.org/10.1016/0959-1524(95)00009-F. URL http://www.sciencedirect.com/science/article/pii/095915249500009F.

[3] Y.A. Cengel and M.A. Boles. *Thermodynamics: An Engineering Approach*. McGraw-Hill series in mechanical engineering. McGraw-Hill Education, 2018. ISBN 9781260092684. URL https://books.google.nl/books?id=lN9JswEACAAJ.

[4] Enerdata. Global energy statistical yearbook 2018, 2019. URL -BNurldate={2018-01-25}.

[5] Carl Rabeler Gene Milener, Craig Guyer. What is odbc?, 2017. URL https://docs.microsoft.com/en-us/sql/odbc/reference/what-is-odbc?view=sql-server-2017.

[6] GitLab. A full devops toolchain.*, 2019. URL https://about.gitlab.com/.

[7] Lokesh Gupta. Java singleton pattern explained, 2012. URL https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/.

[8] Duncan Haughey. Moscow method, 2014. URL https://www.projectsmart.co.uk/moscow-method.php.

[9] Adam Joseph Head, Carlo De Servi, Emiliano Casati, Matteo Pini, and Piero Colonna. Preliminary design of the orchid: A facility for studying non-ideal compressible fluid dynamics and testing orc expanders. In *Proceedings of the ASME Turbo Expo 2016: Turbomachinery Technical Conference and Exposition GT2016*, June 2016.

[10] iba. ibadatmanager, 2019. URL https://www.iba-ag.com/en/products/ibadatmanager.

[11] Mathworks Inc. Create a simple app using guide, 2018. URL https://nl.mathworks.com/help/matlab/creating_guis/about-the-simple-guide-gui-example.html.

[12] Oracle Inc. Jdbc overview. URL https://www.oracle.com/technetwork/java/overview-141217.html.

[13] Oracle Inc. Javafx 2.2, 2014. URL docs.oracle.com/javafx/2/api/index.html.

[14] Oracle Inc. Class linechart<x,y>, 2015. URL docs.oracle.com/javase/8/javafx/api/javafx/scene/chart/LineChart.html.

[15] Oracle Inc. Javafx frequently asked questions, question 6, 2016. URL https://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6.

[16] Oracle Inc. Jdbc-odbc bridge, 2018. URL docs.oracle.com/javase/7/docs/technotes/guides/jdbc/bridge.html.

[17] Oracle Inc. Package javax.swing, 2018. URL docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html.

[18] Oracle Inc. Java se development kit 11 downloads, 2019. URL https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html.

[19] Oracle Inc. Java se runtime environment 8 downloads, 2019. URL https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html.

[20] Oracle Inc. Oracle, 2019. URL www.oracle.com/index.html.

[21] National Instruments. Setting up a citadel database as an odbc data source for labview dsc, 2011. URL http://www.ni.com/white-paper/4853/en/.

[22] National Instruments. Retrieving data from a citadel database, 2013. URL http://www.ni.com/white-paper/12606/en/.

[23] National Instruments. Archived: Accessing citadel 5 data from other software, 2016. URL http://www.ni.com/tutorial/6668/en/.

[24] National Instruments. Logging data with national instruments citadel, 2018. URL http://www.ni.com/white-paper/6579/en/.

[25] National Instruments. Compactrio systems, 2018. URL http://www.ni.com/nl-nl/shop/compactrio.html.

[26] National Instruments. Basic tcp/ip communication in labview. Technical report, National Instruments, 11 2018. URL http://www.ni.com/white-paper/2710/en/.

[27] National Instruments. Using the labview shared variable. Technical report, National Instruments, 11 2018. URL http://www.ni.com/white-paper/4679/en/.

[28] National Instruments. Labview datalogging and supervisory control module, 2019. URL http://www.ni.com/nl-nl/shop/select/labview-datalogging-and-supervisory-control-module.

[29] National Instruments. What is labview?, 2019. URL http://www.ni.com/nl-nl/shop/labview.html.

[30] National Instruments. A global leader in automated test and automated measurement systems, 2019. URL http://www.ni.com/.

[31] Java. Java + you, download today, 2019. URL www.java.com/en/.

[32] Mountainminds GmbH & Co. KG and Contributors. Jacoco java code coverage library, 2017. URL https://www.eclemma.org/jacoco/.

[33] A. Kumar, M. Chandra, and S. Agarwal. Gui implementation of real time feedback system to learn singing. In *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pages 1051–1054, May 2016. doi: 10.1109/RTEICT.2016.7807991.

[34] Badlogicgames libGDX. The libgdx website, 2013. URL https://libgdx.badlogicgames.com/index.html.

[35] MathWorks. Matlab engine api for java, 2019. URL www.mathworks.com/help/matlab/matlab_external/get-started-with-matlab-engine-api-for-java.html.

[36] MathWorks. Math. graphics. programming., 2019. URL https://nl.mathworks.com/products/matlab.html.

[37] Mockito. Tasty mocking framework for unit tests in java, 2019. URL https://site.mockito.org/.

[38] OpenJFX. Javafx 11, 2019. URL openjfx.io/index.html.

[39] Green Robot. Eventbus: Events for android, 2019. URL http://greenrobot.org/eventbus/.

[40] Scrum.org. What is scrum?, 2018. URL https://www.scrum.org/resources/what-is-scrum.

[41] SourceMaking.com. Observer design pattern, 2019. URL https://sourcemaking.com/design_patterns/observer.

[42] Johannes Link Matthias Merdes Marc Philipp Christian Stein Stefan Bechtold, Sam Brannen. Junit 5 user guide, 2018. URL https://junit.org/junit5/docs/current/user-guide/.

[43] SAGE Technologies. Flexpro processing and reporting software, 2019. URL https://sagetechnologies.com/whats-new/5-flexpro-processing-reporting-software.