EXAMENSARBETE

Modular MiniTest Tester

Mattias Kallsäby

Civilingenjör, Datateknik
2017

Luleå tekniska universitet
Institutionen för system- och rymdteknik

Abstract

RealTest is a company located in Västerås, Sweden, that works with embedded systems and test systems. RealTest had a need for a new test system for one of their products, the MiniTester (MT) mk2 that is used to test Drive Control Units (DCUs) used on trains. The problems that had to be solved were, finding out the most common faults of the product, make the test system modifiable and scalable, have a software running on a Windows PC with a working GUI and test logic and design a hardware component measurement unit to generate and measure signals.
The system described in this report is the software parts of the developed MiniTest tester that runs on a Windows PC as well as the hardware design. The programming of the measurement unit and the testing of that unit is not covered by this report.
The hardware consist of custom PCBs and Arduino boards. The Windows PC programs implemented a GUI and test logic according to the goals with a few exceptions left for further work. These programs have been evaluated by emulating the hardware. The system is shown to be modifiable in practice by implementation and scalable in theory.

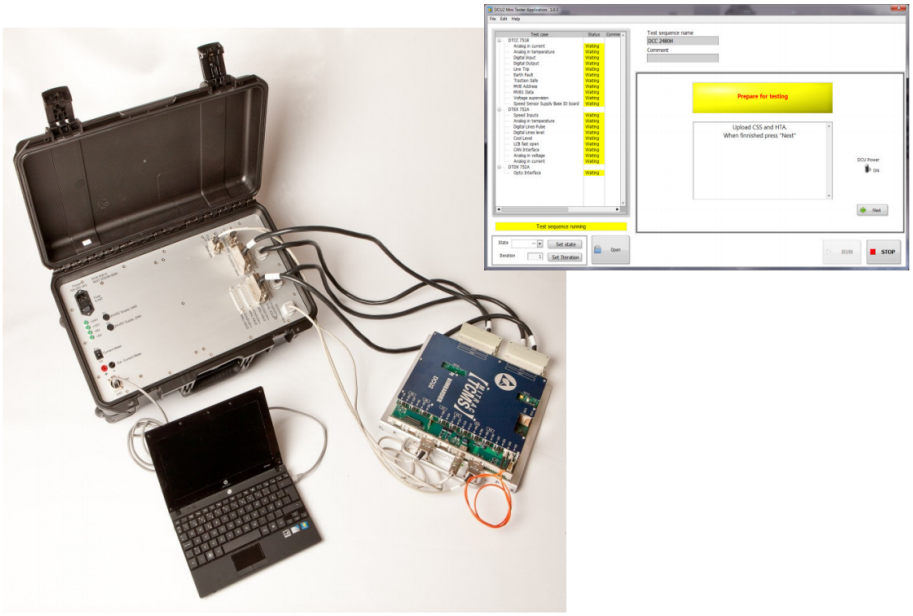
Contents

1 Introduction

1.1 Background

RealTest is a company located in Västerås, Sweden, that works with embedded systems and test systems. RealTest has developed and currently offers a portable test system, Minitester(MT) mk2. The Minitester is produced for Bombardier, a manufacturer for planes and trains. The purpose of the MT is to do function tests on Drive Control Units (DCUs). A DCU is a control board that is used on trains. The DCU comes in different models. The models are used for different purposes with different hardware interfaces. For example, one DCU may control things like the doors and lights or the engine. The DCU is made up of components and the difference between the different models is which setup of components is installed. The MT is developed to be able to test a multitude of different DCUs with multiple outputs for different layouts of components. MT can automatically test a DCU according to a specific test sequence fit for each DCU model.

Figure 1: Minitester connected to a PC and a DCU and snapshot from test GUI.



The MT is placed in a suitcase to allow for it to be brought out in the field. The purpose of automatically testing DCUs in the field is to limit the amount of time a train is at a standstill because of a computer fault. If the fault was thought to be because of a DCU it used to be reclaimed. When a DCU were

reclaimed, it was sent back to Bombardier in Sweden for examination. In many cases, they found that there was nothing wrong with the DCU and that the problem was located elsewhere in the train. This can now be detected on site using the MT without having to send the DCU back to Sweden.

Figure 2: Minitester suitcase.

The production of the MT hardware is not done by RealTest but is instead done by a subcontractor. When the hardware is built, it is sent to RealTest who then program and test the MT before shipping it to Bombardier.

1.2 Motivation

The production volume of the MT is low and therefore some of the manufacturing work is done by hand. Human interaction can often lead to production faults. This uncertainty forces RealTest to examine and test each unit thoroughly before it's sent to the customer. This additional work is costly and time consuming since it's done completely manually. A great amount of knowledge of the MT is required for these examinations.
The production volume of the MT is about to increase to meet an increasing demand. The need for a dedicated test machine only appears when the production volume reaches a threshold where it is more expensive to do the testing by hand. Therefore RealTest aims at developing a test system for the MT. The MT will be modified over time to account for new needs and functions of the DCU to be tested. Hence, the test system should be designed to easily adapt to such changes.

1.3 Problem Definition

1. What are the most common faults of the MT? RealTest want to research how a test system could be designed and developed that will find these common faults. This question was given to us by RealTest so that we could understand what to focus on when testing the MT.

2. The MT will change over time. RealTest wants to know if the system can be modifiable enough to keep up with the changes to the MT. How can these future modifications be added?

3. RealTest wants to know how a system like this could scale up in a factory environment, if the production is to increase even further in the future. How can the system scale?

   (a) We learned from visiting the factory, see section 3.1.1, that scaling the test system would increase the number of operator screens and hardware test equipment. To scale the system, it would be optimal to reduce the cost of those parts and make them as efficient as possible.

4. The system should contain a central software. The software must run on a windows PC.

   (a) RealTest wants to be able to run the software from their own computers and therefore need software to run on windows machines.

   (b) See section 1.3.1 for a more clear definition.

5. The system should support one or more measurement units.

   (a) RealTest wants to have hardware that can handle the signals that the MT use. One option given was to buy measurement instruments to accomplish this. We decided to build our own hardware so that

we could design with other goals in mind such as scalability and modifiability.

(b) See section 1.3.2 for a more clear definition.

1.3.1 Central software requirements

1. The central software should implement test logic. The test logic must consist of test execution instructions. RealTest wants to use the concept of test logic according to the demands below. They have previously built systems in this way and it has been proved useful for them.

   (a) The instructions must be modifiable without adding or editing code.

   (b) The instructions must include what type of test case will be executed and in what order.

   (c) The instructions must include parameters for test cases. For example, setting the voltage level of a voltage generator.

2. The Central software should communicate with a test operator. The communication with a test operator should be done with a graphical user interface (GUI). RealTest wants to have this GUI so that the subcontractors or they themselves can operate the system.

   (a) The GUI should be usable by the subcontractors that are producing the hardware of the MT. The user should not need any special education prior to using the GUI.

   (b) (Optional goal) The GUI should be able to display previous test results.

   (c) (Optional goal) The GUI should have a view for editing and creating test execution instructions.

3. The central software must be able to use the measurement unit(s) to execute tests.

   (a) (Optional goal) The central software should be able to use various test equipment, that are using an Ethernet interface, to execute tests. This is to incorporate test equipment built by RealTest to this system.

4. The central software must store test results in a database. It was important to store data of the testing so that the data can be viewed in the future. It can be used to determine if a device was tested before shipping and also to see if there was anything out of the ordinary with a specific device before shipping.

1.3.2 Measurement unit requirements

1. The measurement unit must be able to handle all the different signals in table 1.
2. The measurement unit must be accessible from a PC and the central software.
3. The measurement unit must be able to run independently of the central software. Other systems should be able to invoke the interface of the measurement unit.

Table 1: Minitester functionality

Function	**Amount**	**Specification**
Digital In	13	24V
Digital Out	16	24V
Analog Current Out	17	20mA, 100mA, 500mA, 800mA
Analog Voltage Out	2	24V
Resolver simulator	1	
Pulse simulator	3	9V
PWM Out	2	24V 30Hz-1kHz
Pt100 simulator	6	
CAN interface	1 channel	
MVB interface	2 interface with 2 channels	
Ethernet	1 port	

1.4 Delimitation

A resolver is an analog electrical device that is used for measuring degrees of rotation. The resolver simulator function listed in table 1 will not be tested. The functions are physically located on different cable connections on the MT and RealTest are satisfied with a proof of concept solution so we will limit ourselves to test at least one channel for each function but not all.

We are two students working in this project and will therefore divide this work between us. Therefore, in this project I will work with the design of the hardware of the measurement unit but not the implementation or evaluation of that design.

1.5 Thesis Structure

Section 1 provides the background to the thesis and the problems addressed are specified and motivated. Also delimitations to the thesis are described.

Section 2 discusses related work and solutions.

Section 3 describe our method to approach the problems and the contribution that I made to this project.

Section 4 describes related theories needed to solve the problems. Technologies, architectures and design pattern that were interesting are described and discussed.

Section 5 describes our implemented solution, architecture and components. It is described in UML diagram, class diagrams and text. It presents communication protocol and the framework for testing.

Section 6 discuss and evaluates how well our solution handles the problem. What it does and what it does not do well.

Section 7 discusses the solution to the problem addressed by the thesis, including an assessment of what could have been done differently for better results and what parts can be considered as good results.

Section 8 describe our conclusions and the outcome of this project. It outlines things that are left to do to reach a more complete solution and what could be done in the future.

2 Related Work

The MT itself being a test system for DCUs is designed to test in and out signals of the DCUs, and consequently the Modular MiniTest Tester (MMTT) will need to operate on the same kinds of signals. This suggests that the MMTT should be similar in its design as the MT. However, since the MMTT as a single system shall be capable of testing current and upcoming versions of the MT it need to be more modular. Another difference is that the MMTT will be used in a lab environment while the MT build for outdoor missions. Hence, although the MT and the MMTT are targeting tests of essentially the same signals, they do not share the same requirements on design and implementation.

A similar attempt at creating a test system was made where the objective was to find a testing framework for automotive systems [5]. In this attempt, they achieved their goals of designing an automatic testing framework. They also achieved the goal of implementing a tool chain that includes managing test requirements, review test cases, automatic testing processes and test report analysis and they also added functionalities to make the framework more intelligent. They developed a "Test Core" that automatically performs a testing process that includes selecting test cases, deciding sequence of test cases and

executing test suites [5]. They added functionality such as test loops and max failure to be able to run a test case over and over, this feature was added in a different way in our system. In section 5.3 we show a solution where a test case get to trigger test cases to execute them in succession. This allows us to trigger multiple test cases in a loop to get a similar effect. In our solution there is no max failure count so that our loop would be infinite until a successful result is achieved. A solution is presented in section 8 to deal with the infinite loop problem.
They implemented an analysis functionality to the system that we never did. It is a drawback to our design that we lack any analysis functionality and should have been a goal if there was more time for this project.
Nothing from their work was used in our system but it was useful for us to be able to compare our system to theirs to find the weakness of the infinite loop.

National Instruments (NI) is a company working with test, measurement and control solutions. NI has developed modular instrument systems for automated tests. These modular instruments inspired our design. In their CompactDAQ controller, *"Data Acquisition (DAQ)"*, the modules are connected on a bus and the modules all have a shared power supply, much like our solution. The Controller has an Ethernet interface to communicate with a PC running a Labview application. With a modular instrument the costs and size are lowered and has an extended lifetime [12]. The drawback of buying the modular instruments from NI is that it can have a high initial cost to buy the controller and the modules needed. We designed out hardware to run specific test cases while their design is more general. A drawback of having the general design is the redundant and unnecessary functionality. By designing our own modular system we can implement very specific hardware requirements.

3 Method

In this section we describe our method to approach the problems and the contribution that I made to this project.

3.1 Problem Approach

In this section our approach to solving the problems are described.

3.1.1 Investigation

We asked questions about RealTests subcontractors and the problems RealTest has experienced. Questions asked were, "What parts of the MT is assembled by hand?", "What are the most common faults?". We assumed that the largest cause of mistakes would be because of human error, which is a safe assumption considering humans are among the major contributors to breakdowns [3]. Therefore, we wanted to know what parts of the MT was assembled by hand.

These questions were asked so that we could try and get an understanding of what they thought were the most common faults and how we could achieve the goal of creating test logic for the MT. An alternative method for investigating this could have been to look at statistics about where the faults usually appears and then see if they were missed by the test operator or the test equipment. Visiting the subcontractors could also have been a good method to find out what was the most common faults from manufacturing the MT, this was however not done. By making such a visit we would have wanted to see an assembly of a MT or ask questions to figure out what the process was.

A visit was also made to a factory to see how the test instruments are used today in the industry. Here, we wanted to investigate how production works in a larger scale, so that a plan could be put in place for how to scale up the test system. This visit was a guided tour and we had not prepared any questions.

Research was conducted to figure out how to design the system in a modifiable way. We looked at different design patterns and communication protocols.

3.1.2 Design

After gathering knowledge of possible problems that have to be taken into consideration, architectural design patterns were picked for different parts of the system. Patterns had to be picked to achieve the goals to be modifiable, scalable and to implementing test logic. When a pattern was found that could provide the attributes required to accomplish the goals it was used in the design. The patterns that were chosen were not analyzed or compared to other patterns. The patterns were mostly familiar to us and we had worked with them before and therefore we had the confidence to use them without the analyze. Communication protocols and Application Programming Interfaces (API) were formulated in the design. By discussion between me and the other student different communication protocols were analyzed.

Unified Modeling Language (UML), Sequence Diagrams and text documents were used in designing the system. Sequence Diagram is useful for designing how the system behaves and communicates between components. UML was used to design the structure of the code. UML have a benefit of graphically showing a design which allowed to better communicate design ideas. Text documents was used for describing APIs and the file structures.

3.1.3 Implementation

We used an agile work process during the implementation phase which helped in dealing with new requirements that got updated. There were three sprints planed for the implementation. During these sprints some parts of the design and some parts of the goals did change. To begin with the goals were 1, 4 and 5 from chapter 1.3. Point 2 and 3 became clear after interviewing RealTest

and most of the sub points were adopted during the work when we got a better understanding of what RealTest wanted to achieve with this project. We also added goals of our own when we had ideas for how the system should work.

3.2 Work Contribution

We were two students working in this project. We worked together in creating the overall system design. We set up rules for how the different components would work together and how they could communicate.

My part was the server and GUI. I designed all of the architecture of these two systems and the interface used between them. It was my responsibility to solve the test sequence problem. I helped design and build the hardware but had no part in the design or implementation of the software running on it. I also had no part in evaluating the hardware once it was built.

4 Theory

This section describes related theories needed to solve the problems. Technologies, architectures and design pattern that were interesting are described and discussed.

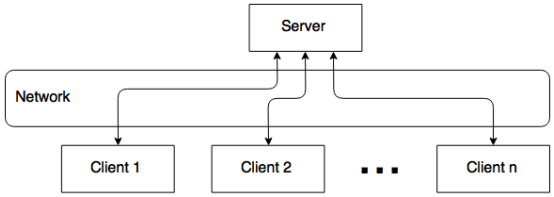
4.1 Scalability

Scalability is the ability of a system to work well when it or its context increase in size or volume. Scaling a system is done by adding extra hardware or upgrading existing hardware without changing the application [15]. Scaling the amount of monitors and measurement units in this system will use one of two architectural designs. It will have a client to server architecture or a peer to peer architecture.

4.1.1 Client Server

Client Server type architecture is a network architecture that has a centralized administrative system. Clients connects to the server as seen in figure 3. A client requests activity from a Server to accomplish tasks [13]. The client is often seen as the front-end part of a system and the server as the back-end. A user would interact with the front-end part of the system with some interface. The front-end will use the back-end part for shared resources but can have some private resources as well, resource meaning some calculation or data. Benefits of scaling client to Server architecture is that it is simple, upgrading the server immediately allows for more clients. A drawback to this is that it can cause network congestion.

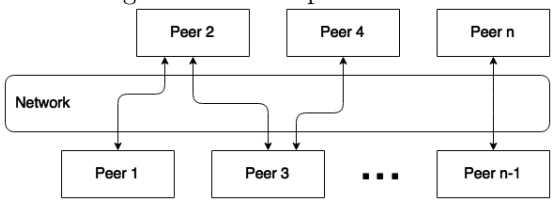
Figure 3: Client server Network.



4.1.2 Peer to peer

Peer to peer (P2P) is a network architecture where nodes ("peers") share resources amongst each other without the use of a centralized administrative system. The peers connect directly to each over the network, seen in figure 4. A challenge in peer to peer programming is determining how peers find each other on a network [14]. A common method of peer discovery in .NET applications is to use a central discovery server, which will provide a list of peers that are currently online [14]. There has been much interest in emerging Peer-to-Peer (P2P) network overlays because they provide a good substrate for creating large-scale data sharing, content distribution, and application-level multicast applications [6].

Figure 4: Peer to peer network.



4.2 Communication Protocol

4.2.1 Hypertext Transfer Protocol

Hypertext Transfer Protocol(HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems [16]. HTTP communication is done with requests and responses. A request is sent from a client by first setting up a TCP connecting with a three-way handshake, sending the request header and content. The client is then waiting to receiving a response from the Server. Finally, when the response has been received the TCP connection is closed. The connection is always closed after the interaction is complete.

4.2.2 Web-socket protocol

Web-socket is a communication protocol that provides full-duplex communication. It uses a TCP channel to send message back and fourth. To setup the channel a web-socket handshake is initiated by the client. When the handshake is done the TCP channel is open and usable until either the server or the client closes that TCP connection. The web-socket allows the server to initiate actions and to push information to the client as long as the client first initiate a connection [8]. Pushing information means that it is the server that takes the initial action to start the interaction of transferring information to a client. Pull is when the Client has to request the interaction from a server before the server can provide the information.

4.2.3 Constrained Application Protocol

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained networks in the Internet of Things [4]. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation [4]. It supports a built-in discovery of services and resources and it also incorporates key concepts of the Web such as URIs and Internet media types [17]. The built-in discovery service is useful for when using the peer-to-peer pattern so that peers can discover services and resources without a central registry.
CoAP is similar to Hypertext Transfer Protocol (HTTP) in its use of URI and it is possible to implement Representational state transfer (REST) API with CoAP. CoAP mainly uses UDP while HTTP uses TCP, however, CoAP can be made to use TCP if needed.
Web-socket has two-way communication and HTTP does not but the CoAP has something between the two. CoAP has two-way communication with the use of its observer functionality. The observer functionality works like a regular request except that the server can respond multiple times. The observer functionality introduces a way for the server to continuously send messages to the client without the need for the client to request the interaction. It is however limited to so that the client first has to request to be a observer and the client cannot respond on the same channel back to the server but instead has to make new requests.
Drawbacks of the CoAP are that the server cannot push information to the client. The client has to use the observe functionality which is indicating to the server that the client wants to get updates about a resource. This way will allow the server to send data to the client but it was not meant to be used to set up a full-duplex communication [4].

4.2.4 Message Queue Telemetry Transport

Message Queue Telemetry Transport (MQTT) is a transport protocol. "MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport

protocol. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium [19]."

MQTT uses a subscription based communication method. In MQTT there are clients and a server or a broker. Sometime it is called server and sometime it is called broker [20][21]. The clients either publish or subscribe or does both. The server or broker receives all messages from clients and filters them, decide who is interested in it and then sends the message to all subscribed clients [21]. This is similar to how observation in CoAP works, however, in MQTT a client can both be a subscriber and a publisher while in CoAP the server publishes and a client subscribes. CoAP act more according to a server client based approach while MQTT is more of a M2M protocol. The difference that a client can be both publisher and subscriber makes it possible for MQTT to be used with two way communication if two clients subscribe to topics from each other.

4.3 Service Oriented Architecture

Service Oriented Architecture (SOA) is an architecture that encapsulates and abstracts functionality in services that are published with an interface [1]. The Services of SOA are loosely coupled with each other and the consumer. Loosely coupled means that the components have low or no dependencies and low or no interaction with other separate components. So that if one of the components were to change it would not affect other components much or at all. This is useful because it makes the system robust to changes. In the case of SOA that means that if a service changes it would not effect how well the consumer works other than the use of that one specific service. It also means that in the case of changes at the consumer side, the services would still work in the same way they did before the change to the consumer occurred.
SOA can be implemented with three parts, provider, consumer and registry. The provider implements a service and provides an interface to access its functions. The consumer consumes the functions that services provide, using the specified interfaces of the provider. The registry is an optional part that solves the problem of finding services. The registry is a lookup table for available services. Providers register their services to the registry. Consumers can then use the registry to find services amongst the providers.
A system based on SOA can implement late binding, that is the ability to discover and make use of resources at runtime rather than having been the designed to use the resource at design time [18]. New services become available to the end-user when a consumer discovers a new provider that provides those new services.
One important benefit with SOA is that it can be evolved based on existing system investments rather than requiring a full-scale system rewrite [2]. Services can be developed and added independently, without having to alter the existing system, even after the system has been deployed.
The loose coupling also make the system more maintainable and makes it more easy to incorporate subsystems.

SOA is predominantly being applied in the form of web-services. It has proven useful in the web but it's benefits can also be used elsewhere [2]. The web-services are not only an implementation of SOA it is a collection of multiple technologies such as, XML, simple object access protocol (SOAP), representational state transfer (REST).

4.4 Adapter Pattern

Adapter Pattern is used when two components need to cooperate and when there interfaces does not match [7]. The adapter pattern is being used to incorporate functionality from components that are not compatible with the current system. It does so without the need for changing the component but rather wrapping it with a new interface. The adapter pattern can be implemented as seen in

Figure 5: Adapter pattern.

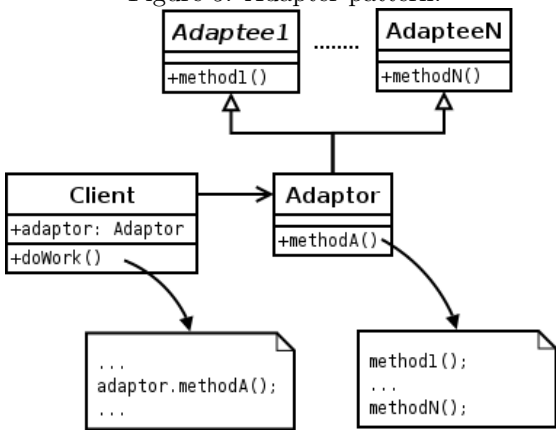


figure 5. The Adaptee is a class that encapsulates a component that has the functionality. A client can then call for the method from an Adaptor which in turn will call for the method from the Adaptee.
Adapter pattern will allow for a modular implementation in software where each subsystem has an adaptee that plugs into the adapter of the core system. Placing functionality behind an adapter pattern will decouple the functionality from the interface. It makes it easier to replace components that are behind the adapter.

4.5 Database

Databases are systems that are being used to handle data. A database typically has search and sort functionality to allow for fast and efficient ways of handling datasets.
A relational database presents information in tables. The data in the database can have relations, linking the content of rows with each other [10]. There are

many relational databases to choose between. Some of the most common are Oracle DB, MySQL, Microsoft SQL Server and PostgreSQL [11]. All of these databases use structured query language (SQL). SQL is used for managing data held in a relational database. With the SQL and one of the databases names above it is possible to retrieve data which are related to each other in an efficient way.
Relational databases has a wide range of uses including storing customer information and providing a searching functionality in a large dataset.

5 Implementation

This section describes our implemented solution, architecture and components. It is described in UML diagram, class diagrams and text. It presents communication protocol and the framework for testing.

5.1 MMTT Design

Table 2 shows a list of concepts that are used in the following sections. The MMTT system consists of three parts as can be seen in figure 6. Orchestrator, Administrator and GUI Application. This is to separate different responsibilities, which will allow for independent development of different parts. The three components are supposed to be working independently from each other using defined interfaces to communicate with each other.

To make the system modifiable a SOA design was implemented. The Test Cases were our services so that modifying or adding new Test Cases/services would not require a rewrite of the rest of the system. The modules will be connected to the orchestrator via a plug and play connection over CAN-bus. The Services that correspond to a Test Case will then be registered on the Orchestrator from the modules that implement the tests. This makes it possible to add new modules in the future when new kind of Test Cases might be needed. It is these modular units together with the SOA that make the hardware part of the MMTT system highly modifiable.

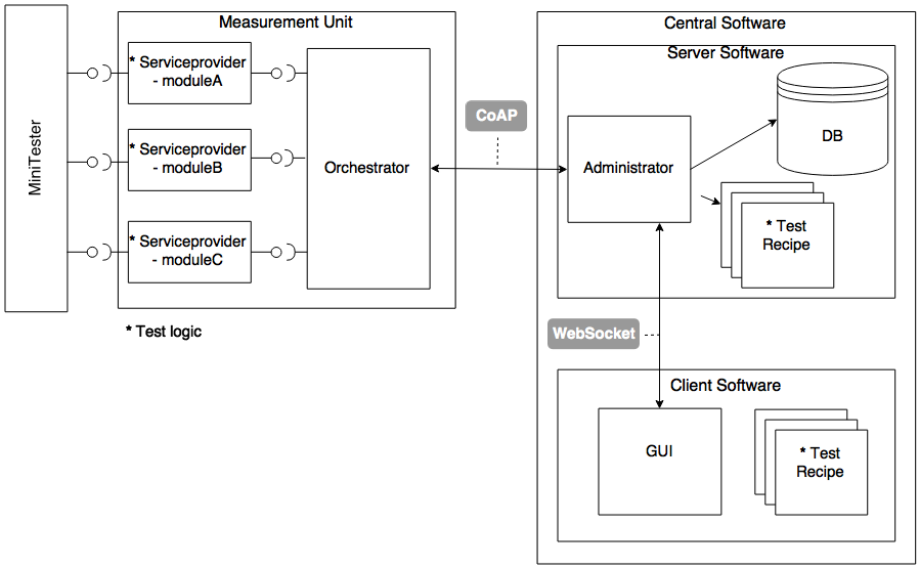
We decided to make both the Orchestrators and the Module low cost, using Arduino boards to run them on. Arduinos are inexpensive and there are many shield boards that can expand their hardware capabilities [9]. We also built custom boards with our own electronics on them to be able to cover all of the functionality of the MT seen in table 1.

MySQL will be used for the database. Time and date of test execution needs to be stored in the database, as well as information about the success or failure of test cases. The parameters from the test execution also need to be stored. A relational database will allow for this data to link together. Other relational databases could have been used such as Oracle Database or Microsoft SQL server. Since there is no need for performance or high storage capacity any

Table 2: List of concepts

Concept	**Description**
Modular MiniTest Tester(MMTT)	MMTT is the name for our overall system. The GUI, the central software and the measurement units all combined.
Orchestrator	The Orchestrator is a service registry unit that is part of the measuring unit. It will have direct communication with the central software.
Module	A Module is a low level device that can measure and generate signals on Input/Outputs (IOs) pins.
Administrator	The Administrator is the central software that will handle the sequence instructions and handle communication with both the GUI application as well as the measurement unit (Orchestrator).
GUI application	The GUI application is, as the name suggests, an application with a graphical user interface that allows an operator to both edit and create test sequence instructions and execute them on the central software (Administrator).
Test Case	A Test Case is either a measurement/signal generation on some test equipment or measuring device, or it can be a instruction to a test operator on what to do.
Test Recipe	A Test Recipe is an instruction of sequences of Test Cases. It describes the Test Cases, in which order they are executed and how they should be executed.

Figure 6: Overview of the system design.



of these databases could work. MySQL was the choice because it is a database that I have worked with before so it was easy to setup.

5.2 Communication Framework

5.2.1 Administrator to Orchestrator

The CoAP protocol will allow the Administrator and Orchestrator to communicate with each other. There are methods of delaying a request with CoAP, which is useful for this kind of an application. For example, when the administrators request the execution of a Test Case that would take a long time to finish, the response doesn't have to be returned immediately and no timeout would occur.
There is also benefits like the Observation functionality that would allow the Orchestrator to send data back to the Administrator multiple times. This could be useful for Test Cases where some data has to be displayed in real time in the GUI, so the administrator would just observer the progress of a Test Case and give this real time feedback to the operator.

The MQTT protocol could also have been used here and might have worked just as well or may even do better. It is unclear since we did not try the MQTT but instead focused on getting other parts of the system to work.

The communication could also be done with a basic TCP or even UDP con-

Message Type	Description
GET	Request to get data
UPL	Request together with data to be stored or used by the receiver
DEL	Request to delete or remove something from the receiver
RUN	Request to execute a process
STP	Request to stop or halt the execution of a process
RSP	Response to a request

Table 3: Message type descriptions

nection. This would require more time to implement our own protocol for communication which would require a longer development time.

5.2.2 Administrator to GUI Application

The administrator implements a Web-socket server interface. The Web-socket server allowed us to have both the server and client push message to each other. Once a connection was made from the client, there was a TCP channel open that both parties could use freely to send messages.
The interface has two resources that can be connected to. One resource is located on the path */testengine* and it controls the test engine for the MMTT system. One resource is located on the path */recipe* which allows uploading, updating and downloading of Test Recipes.
With the HTTP it was too difficult to get a two-way communication working. By using HTTP alone it did not work, instead we had to try and find other technologies such as JavaScript based libraries to make use of as well which made it too time consuming and out of scope for this project. Using a CoAP server did work for the most part. The CoAP failed when the server was going to push requests to the GUI application. The observer function of CoAP did only work when the client first could expect when the server would want to push requests. If the client has to know when the server wants to communicate it is no longer a true push functionality.

To differentiate from different kinds of messages being sent to a resource a new layer of communication was added on top of the Web-socket protocol. This protocol has six different kinds of messages and an array of data. The protocol has three characters to identify a message type. The different types can be seen in table 3. Following the message type identifier is an array of data where each element is surrounded by the '{}' parenthesis.

Table 4 shows what commands can be sent between the Administrator and the GUI Application. Sequence diagrams of three different scenarios can be seen in figure 7, 8 and 9.

In figure 7 the GUI will request to run a TestRecipe containing only one TestCase that will be in turn executed on the Orchestrator.

Figure 7: Administrator UML.

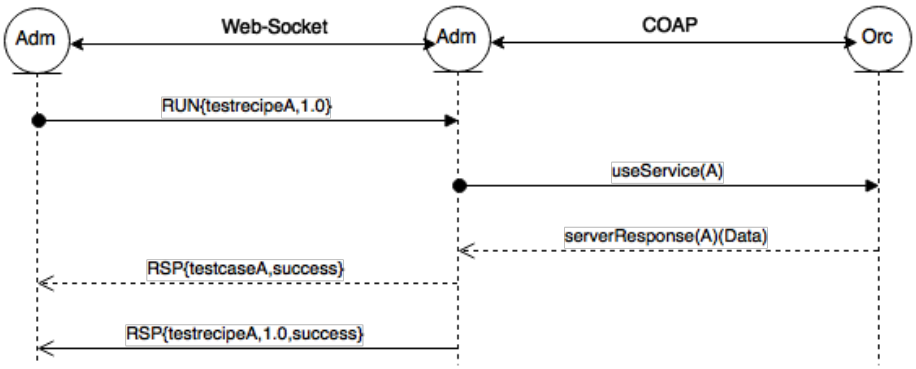
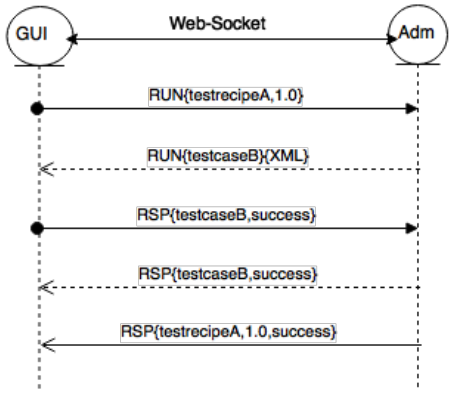


Figure 8: Administrator UML.



In figure 8 a TestRecipe is executed that only contains one TestCase that will be executed on the GUI as a manual TestCase.

In figure 9 a TestRecipe is requested from the Administrator and then edited before being sent back to the Administrator.

Communication with the Orchestrator is according to an API that is designed by my colleague and will therefore not be brought up in this report.

5.3 Test Recipe and Test Cases

Most of the services from the orchestrator are operations that perform a test on the MT hardware, from now those services will be called Test Cases. To allow for more general use of the Test Case, we will also use this term to describe any kind of operation that performs a test or test related task. The Test Case can therefore also be an instruction to the operator and a question to the operator

Figure 9: Administrator UML.

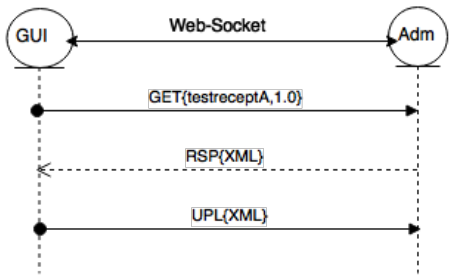


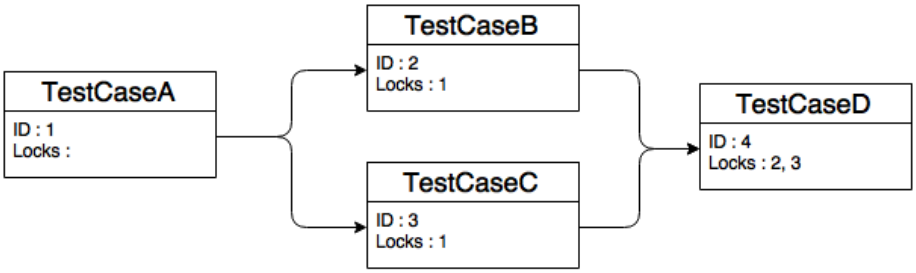
Table 4: Administrator GUI API

Server / **client**	**Description**
UPL{XML}	uploads a XML file with all the information about the test recipe
RSP{XML} / GET{}	Returns XML file with the recipes stored on the administrator and some basic information about the recipe
RSP{XML} / GET{NAME,VERSION}	Return XML file describing the recipe
RUN{NAME,VERSION}	Runs the TestRecipe
STP{NAME,VERSION}	Stops the TestRecipe
GET{NAME,VERSION}	subscribe ongoing test recipe
DEL{NAME,VERSION}	Unsubscribe ongoing test recipe
RSP{NAME,ID,SUCCESS}	notify gui that TestCase is complete
RUN{NAME,ID}{XML}	Request gui to run manual test
RSP{NAME,VERSION} {NAME,ID,SUCCESS}	tells admin that TestCase has completed
RSP{NAME,VERSION, STATUS}	Tells GUI that the test recipe has finished running.

asking for parameter values.
A Test Case can have parameters associated with them. There are two kinds of parameters, arguments and results. Arguments is a parameter that is given before the execution of the Test Case. The argument can then be used in the Test Case execution. The result is a parameter that is expected to be returned from the Test Case when it's done or during execution.
Parameters have a flag called RunTime that describes whether the parameter can be used by other Test Cases. If a Test Case needs to take information from a previously executed Test Case, they will both need to have a parameter with the same name and where both parameters have the RunTime flag set to true. The first Test Case will have to be a result parameter when using the RunTime flag. The Test Case that will use the data from that result will have an argument parameter flagged with the RunTime flag. Once a runtime parameter has been set it can be used multiple times and it can also be changed by a later result parameter.

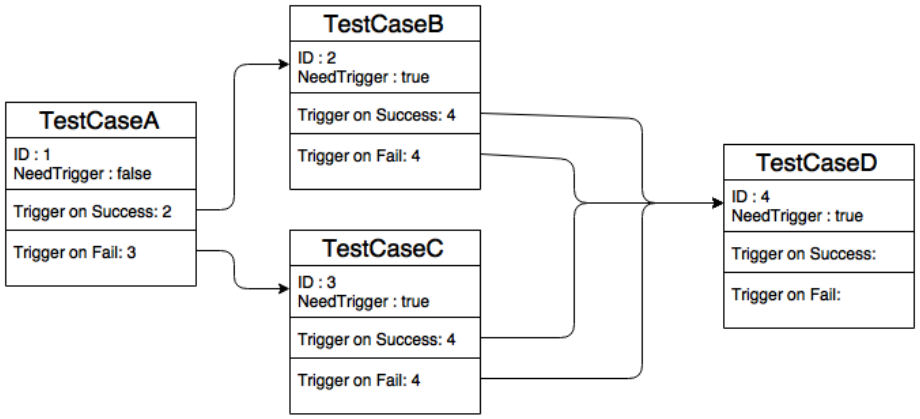
A Test Recipe is a collection of Test Cases combined with instructions about when to execute the Test Cases and with what parameters. To instruct in what order to execute the Test Cases we first introduce locks. A lock is an attribute that a Test Case can have that won't allow it to execute until another Test Case has finished. This behavior can be seen in figure 10. Here TestCaseA starts executing and after it has finished TestCaseB and TestCaseC will execute. Finally, when all other Test Cases have finished TestCaseD will be executed.

Figure 10: Test Case locks interaction



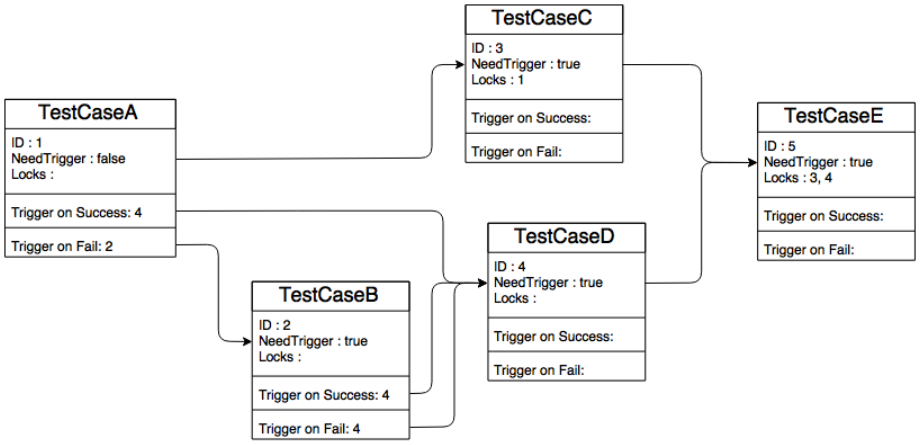
A Test Case can have triggers and a trigger requirement. If a Test Case needs to be triggered it won't execute until some other Test Case triggers it to execute. A Test Case can have the attribute to trigger on success or on failure depending on the outcome of the Test Case. This is illustrated in figure 11. First TestCaseA executes and when it is finished it has either succeeded or failed. If it was successful TestCaseB will execute and if it failed TestCaseC will execute. Finally, when either TestCaseB or TestCaseC are done TestCaseD will execute.

Figure 11: Test Case trigger interaction



The need for both triggers and locks will be apparent when there is a need for an execution plan shown in figure 12. Here TestCaseC will always be executed after TestCaseA but TestCaseB will only execute if TestCaseA fails. TestCaseE should not execute until TestCaseC is finished and if TestCaseA fails it will also have to wait for TestCaseB. Lets also introduce the concept of an empty Test Case. An empty Test Case does not do anything but when executed in the program will immediately finish successfully. In this example, TestCaseD is an empty Test Case.

Figure 12: Test Case locks and triggers interaction



The Test Recipe is described in text-based XML language.

5.4 GUI

The GUI was made in Windows forms. Windows forms comes with a drag and drop GUI design software. It made it easy to create a quick GUI and did not require any graphical design tools. The GUI application has functionality such as Recipe Editor, Test Recipe Executor, Test Execution Results and an options menu.
The Recipe Editor allows the user to create or edit existing test recipes. The view is made with a table container, when a Test Case is added it is placed in a new row. The Test Case has input text boxes for its name, version and handler class. It has two check boxes called unlockOnSuccess and unlockOnFail. These options determine if the Test Case should unlock other Test Cases when it succeeds or fails. There is also a check box called NeedTrigger that determine if the Test Case requires a trigger to run. The setting for each test case can be seen in figure 13 and the settings for arguments and results that belong to a test case can be seen in figure 14.

Figure 13: Recipe Editor Test Case.

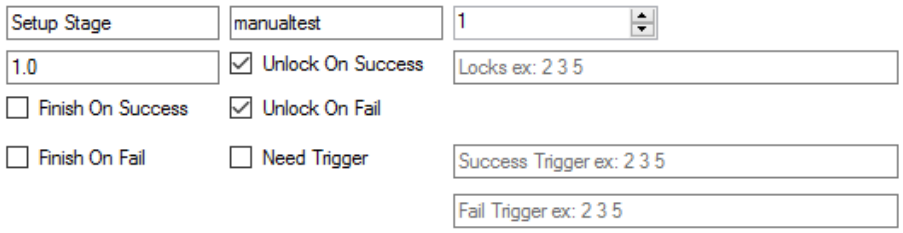
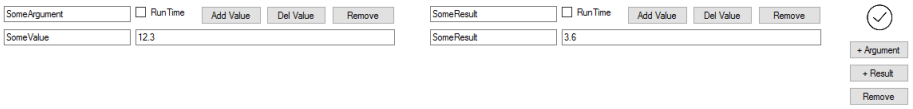


Figure 14: Recipe Editor arguments and results.



The Test Recipe Executor will fetch Test Recipes from the Administrator and display them in a drop down menu. The recipes can then be selected, which will bring up all the Test Cases on to the screen. There is a play button that will initiate the execution. There is also a stop button to stop the execution. When the GUI application receives a request to run a manual test it will open up a new window where the instructions for the manual test is displayed and input boxes for the result parameters. Once the operator has followed the instructions and input all the result parameter, he can press one of the two buttons on the pop up window, one for success and one for fail. The implementation is limited

in that it can only display one manual test at a time and while a manual test is being displayed the rest of the program freezes due to threading not being handled properly.

Figure 15: Recipe Executor view. See figure 16, 17 and 18 for an enlarged image of the three different parts in red in this image.

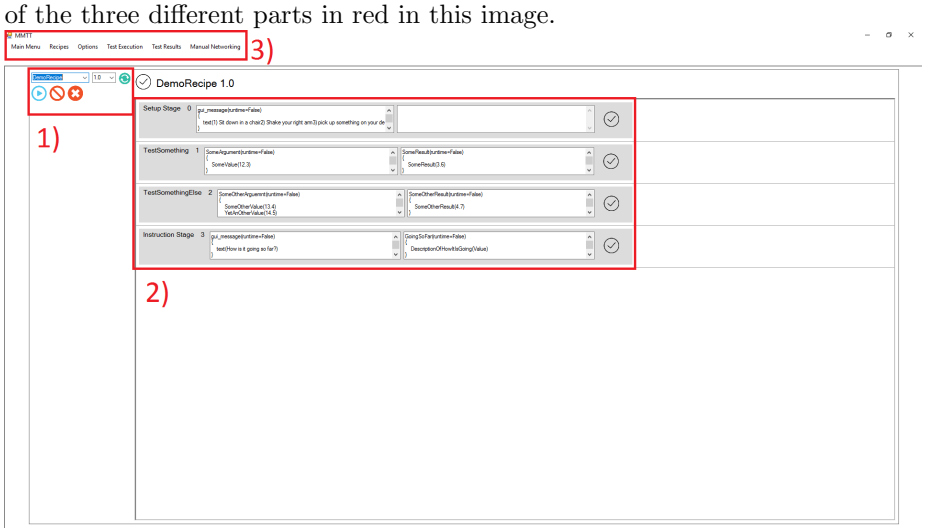


Figure 16: Recipe Executor test case selection. An enlarged image of the red square named 1) in figure 15.

Figure 17: Recipe Executor view. An enlarged image of the red square named 2) in figure 15.

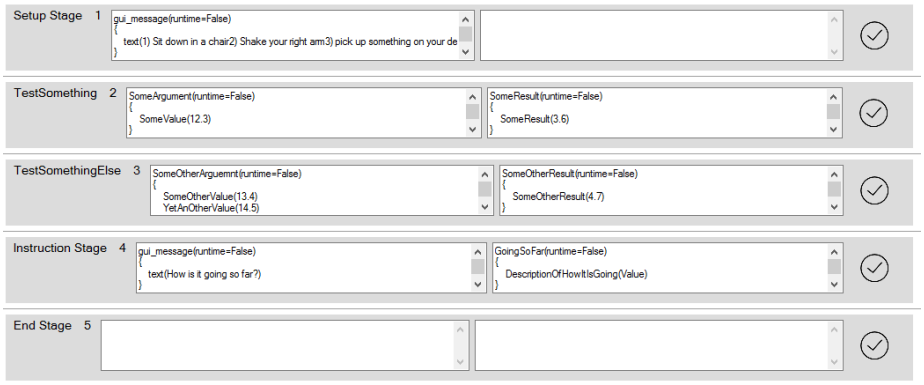
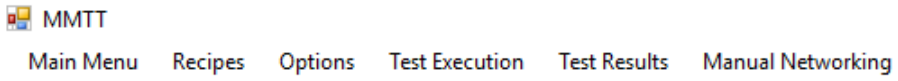


Figure 18: Recipe Executor view. An enlarged image of the red square named 3) in figure 15.



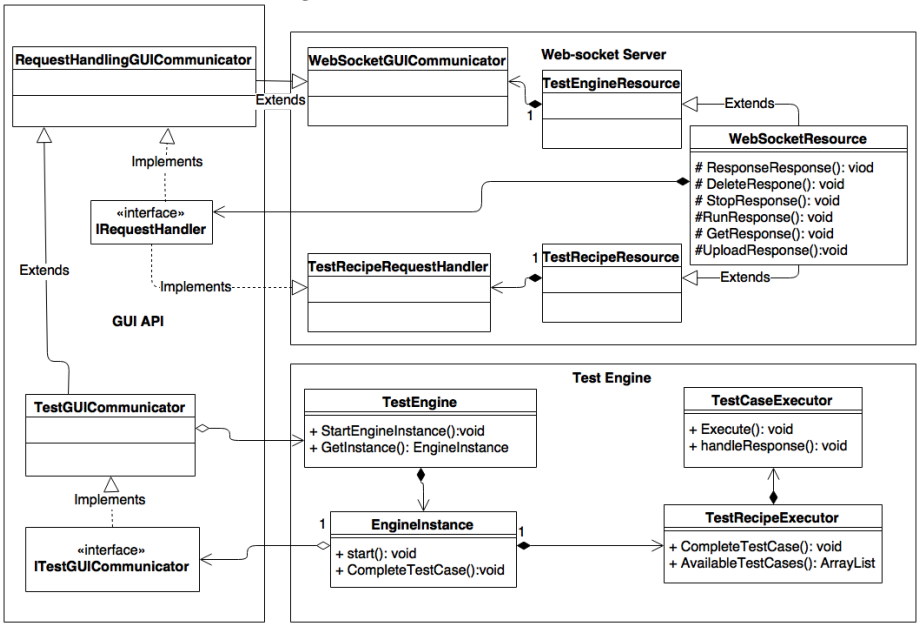
Test Execution Results is a view that searches the database for previously executed tests and displays them.
In the options menu there are settings for IP and port for the orchestrator and also a connection string that is used to connect to the MySQL database.

5.5 Administrator Design

The administrator has mainly three parts to it as seen in figure 19. It has a test engine, web server, and an interface between the two. The test engine and web server run independently of each other and use the interface in between to communicate.

The test engine consists of a static class called TestEngine that has static methods that will control the tests. When a new test starts to run the TestEngine will create a new object of the class EngineInstance and then call the Start Method of the EngineInstance. Multiple engine instances can run at the same time so that multiple tests can be performed at the same time.
Each TestInstance have an object instance of the class TestRecipeExecutor. The TestRecipeExecutor can read a Test Recipe and execute each Test Case in the order specified in the recipe. To execute a Test Case, an object of a class that

Figure 19: Administrator UML.



implements the abstract class TestCaseExecutor will be instantiated. This is an implementation of the adapter pattern where the adaptor is the TestCaseExecutor and its implementations are the adaptees.

The GUI API part defines interfaces for the GUI. It's purpose is to allow the Test Engine to communicate with a GUI and for the GUI to communicate back to the Test Engine. It also defines the interface IRequestHandler that is for handling the different Message Types described in section 5.2. Both the Test Engine and the Web-socket server is coupled with the GUI API but not with each other. This means that the Web-socket server could be exchanged for another type of server to handle the communication with any GUI applications. The purpose of decouple the GUI API from the web server was to make it more convenient in the future to exchange the web server for an alternative way to communicate with the GUI application.

The Web-socket Server is built with the websocket-sharp library. The websocket-sharp has sessions for connected clients and these sessions in turn have a specific behavior. An implementation of the IRequestHadler is done in a layer on top of the websocket-sharp behaviors. A class called WebSocketResource is extending the websocket-sharp behavior class. The websocket-sharp behavior has abstract methods for OnOpen, OnClose, OnError and OnMessage. These methods are called when the resource is connected to, disconnected from, on error and when

a message is received. Each WebSocketResource will instantiate an instance of the IRequestHandler class and use it when a message is received. The IRequestHandler will then handle all of the different type of messages. This way the IRequestHandler is completely Decoupled from the Websocket part, but the websocket is more strongly dependent on the IRequestHandler interface.

6 Evaluation

This section discuss and evaluates how well our solution handles the problem. What it does and what it does not do well.

6.1 Test Sequence

The test sequence implemented with Test Recipes can be tested without actual measurement units. An empty TestCaseExecutor was created that only put the thread to sleep for a few seconds, outputs the Test Case information in a console and then finishing the Test Case successfully. By creating a Test Recipe that only contains these empty Test Cases and Test Cases that were handled by the GUI application, the application could run. Using this method, multiple tests were executed to test different Test Recipes. It confirmed that this implementation actually does have sequence instructions that also include the description needed to execute the Test Cases.

The TestEngine is still a bit unstable with some minor bugs in it, but overall the system can interpret Test Recipes that I build and execute them.
When creating a test sequence, it was discovered that the test parameters had some problems. Some errors that can occur when a user inputs faulty values to the parameter were not being handled correctly. Those errors cause the GUI application to crash. There were also some problems with the sequence triggers where in some cases they seem to break the system, leaving the Administrator waiting for something to happen and not executing the proper Test Case.

6.2 Modifiability

During the development, The web server from CoAP had to be changed to a web-socket server. During this process, I got to experience the difficulty level of the supposedly modifiability of the GUI communication interface talked about in section 5.5. It was possible without rewriting any parts from the GUI API (seen in figure 19). It was also possible to make the underlying system work with the new Web-socket server. The system felt modifiable and I am confident I could replace the web-socket server with another duplex server. There were no other analysis of the modifiability. There are methods for architecture-level modifiability analysis [22]. One approach for analysis is Architecture-level modifiability analysis (ALMA) described in the report of the same name [22]. There were no firm quantitative data that allowed them to determine the accuracy

of the prediction, nor the coverage of risk of the analysis method. No other approach to analyze modifiability was considered.

To be able to use other test equipment, the Test Engine implements the Adapter pattern. The TestRecipeExecutor will instance different tyes of TestCaseExuctors. These TestCaseExecutors are adaptees to other measurement units and test equipments that can be used to perform a Test Case. There has not been an attempt yet to implement any adaptees other than the Orchestrator executor, the GUI executor and the empty executor. A plan to implement an executor for a system that another person at RealTest has built has been put in place, however that will have to be outside the scope of this report.

6.3 GUI Application

To verify the usability of the system I set up a demo for employees at RealTest. The Demo consisted of a built Test Recipe and a short introduction about how Test Recipes work. At first the mechanics behind the Test Recipe was confusing but once I showed the images that are in section 5.3 it became clear how it was supposed to work. The GUI Application had to be usable in a work floor environment so they evaluated it after its simplicity. The Test Recipe Editor and Test Results view were not simple enough to use without a greater knowledge about how the system works and the GUI was not simple enough to be put in the work floor environment. This however does not matter too much since those features could be restricted to employees with proper training. The Test Execution view, that only consists of a few buttons and pop up windows for incoming manual Test Cases, did get positive feedback and was thought to be simple enough for most people to be able to use. We could have made a better evaluation of the usability by doing a larger survey. This however did not feel necessary since at first hand RealTest will be the only users of this system.

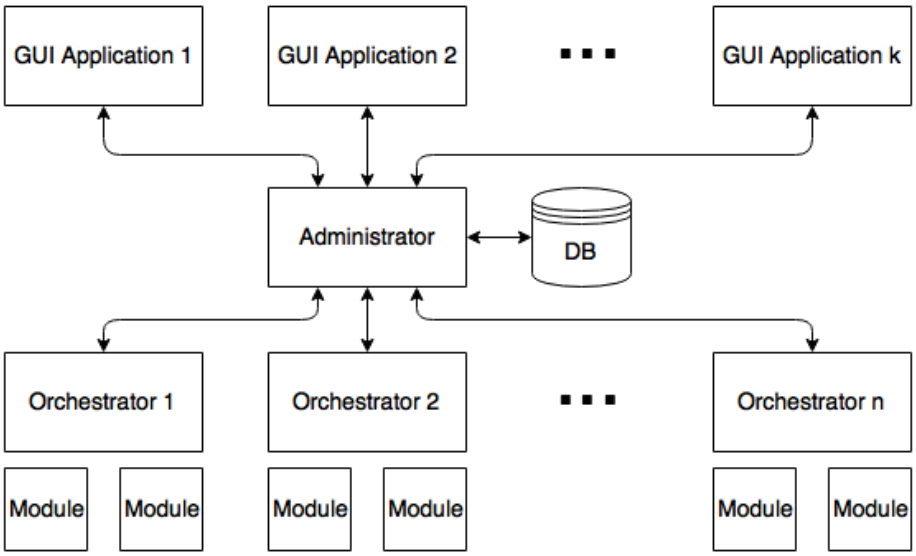
7 Discussion

This section discusses the solution to the problem addressed by the thesis, including an assessment of what could have been done differently for better results and what parts can be considered as good results.

7.1 Test Sequence and Test Case

Adding new or changing test sequences was done by creating a structure for the Test Recipes. It made it possible to create or update the test sequence in a regular text editor or by using the built in recipe editor in the GUI application. It did however get a bit too complicated with all the different ways of instructing execution order. It had been enough to just have the instruction be sequential and only execute one test after another. That would have saved a lot of time on the implementation. The overly complicated sequence instructions would only

Figure 20: Scalability solution 1



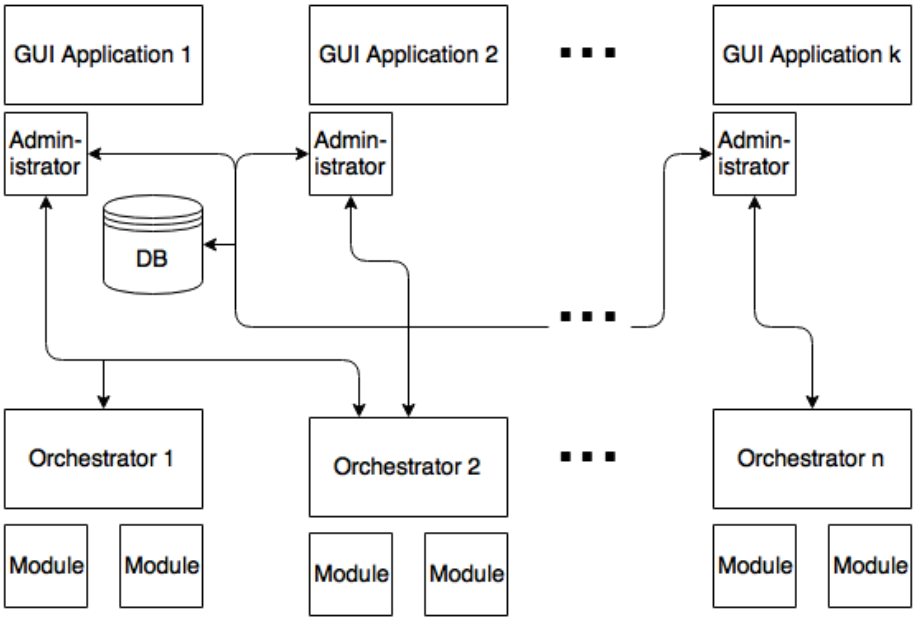
be useful if this system was to be used for more than just the testing of the MT. It was not possible to test or evaluate the TestRecipes without executing the TestCases on the Orchestrator, therefore, empty TestCases were created to simulate the Orchestrator. When executing the empty TestCases they wait for five seconds and then pretend to have received a successful return message from a Orchestrator and finished with a successful state. The delay before returning made it possible to test the execution of multiple TestCases at the same time. The empty TestCase made it possible to test and evaluate the TestRecipes.

7.2 Scalability

There will be two ways to scale our system. The first is by using the administrator as a central server. This will give the opportunity for lightweight GUI applications since it won't have to run all the logic. A model of this scaling can be seen in figure 20. The First solution requires a more light weight computer to run the GUI Application and thus reducing the cost for each monitor running the GUI. This was found to be an important part of the scaling.

The other way to scale the system would be to have no centralized server. It would require the implementation of a discovery function that would allow the administrator to scan a network to find Orchestrators. The only central system needed is the database so that the tests aren't recorded locally. A model of this approach can be seen in figure 21.

Figure 21: Scalability solution 2



7.3 Modifiability

Our design allows us to replace modules as we wish, giving us the ability to completely redo some parts of the tester without it disturbing the rest of the system. Although no analysis of modifiability was made, an attempt at directly modify the system was made. No ways of analyzing the modifiability in a precise way was found so the evaluation was based on the attempt at modifying the system directly.

There was however an idea for how the system is modifiable. The modifiability of the system can be divided into five Levels as seen in figure 22. The highest level is not meant to be modified. The modifiable system level are the parts of the system that could be modified without running to much risk of ruining the core system. The modifiable system level will be discussed more in chapter 8. Where the real modifiability comes to play is the External Systems level. As talked about in chapter 5.5 the Administrator API from the modifiable system level is easily replaceable and therefore the GUI Application can also be replaced by a new application as well. It would of course require the development of that new application but integrating it with the core system would be possible.

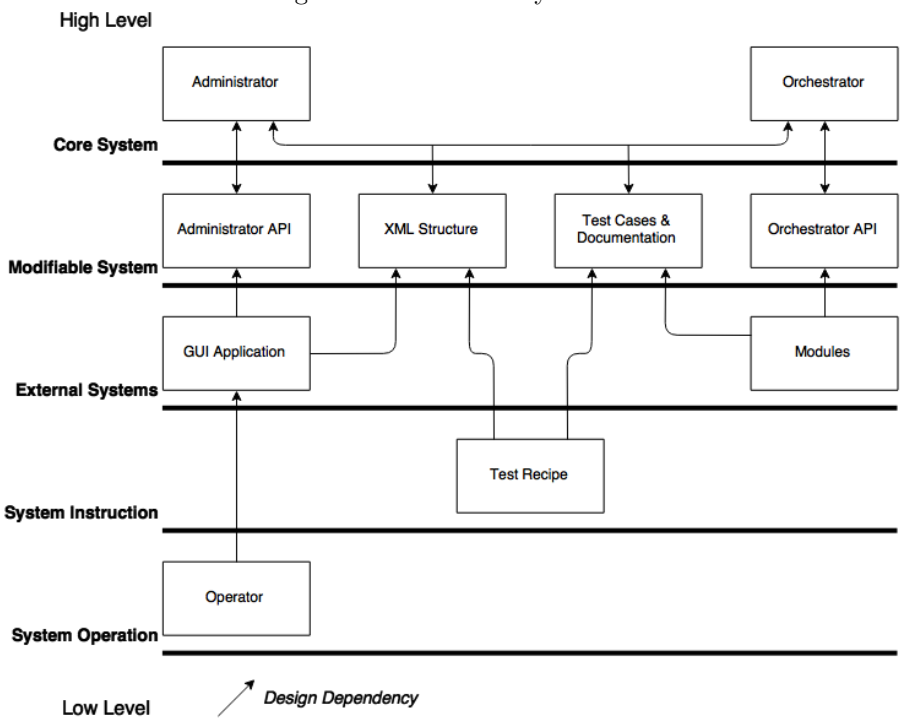
The modules from the external system level is not only replaceable but there is also the possibility to just add an additional module. This implementation is, however, not verified here.

The last two levels does not provide modifiability but more flexibility in how the system is used.

7.4 GUI

More time should have been spent on the GUI. I found it difficult to design a layout for the interface that would feel more intuitive. For the operator that is going to use the GUI there has to be an intuitive way to navigate the different views and a way to control the different functionality. If I could do it again I would have put more time during the research stage to look in to how to make simple and user friendly interfaces.
Although the GUI for test execution got approved by RealTest there could still be some improvements to accomplish the goal of having a usable interface in a better way. For example, from the Test Execution view there should be no direct links to other parts of the program since other views should only be used by people with proper knowledge. The Recipe Editor could also work better with a more graphical interface for sequence tuning. I would like to build an interface that would look something like the graphs in section 5.3. There you would be able to drag and drop connections between Test Cases which I think would provide a better user experience.

Figure 22: Modifiability levels



7.5 MiniTester

During this project, the solution I was working on became more and more general. The central software does not deal with the Minitester itself but is instead a general test engine. It offers test sequencing, test logging and a user interface. Looking back at the system overview design it makes sense that the test engine would be a more general solution and the Orchestrator would be a more specific solution to the MT.
An issue that was apparent at the start of the project was that if we are building a test machine to test a test machine, why would there not be another test machine for our test machine. This could result in an infinite loop of building test machines. However since the test machine I'm building wont be produced more than one time it can be tested by hand once. The need for a dedicated test machine only appears when the production volume reaches a threshold where it is more expensive to do the testing by hand.

7.6 Security

Security was not considered in this report. One problem right now is that anyone with access to the same network that the Orchestrator and Administrator are connected to can access and control both devices, without any authentication. If however these devices are connected to a Local Area Network within a production facility or at RealTests office there would be no problem. The security of the Database is handles by MYSQL that has some authentication built in. To make it more secure the Administrator and Orchestrator would need authentication and maybe even encryption to make the communication secure.

8 Conclusions and Future Work

The outcome of this project was a test engine system with high modularity and a GUI application that could be used together with the test engine. The combined execution of both measuring units and central software could not be tested but was emulated. The emulated results show that the system would work. The goals for modifiability and scalability were evaluated to have been successfully achieved. The functionality of using the measurement units were not confirmed with the actual hardware but instead got confirmed using the emulation.
Sequence instructions were achieved by creating a Test Recipe that could be interpreted by the system and executed in a proper order. Some problems still exist in the forms of bugs and glitches but the concept has been proved to work. This implementation fulfilled the requirements for test logic described in section 1.3.1.
A GUI was successfully built and integrated with the central software to fulfill the second requirement in section 1.3.1. The GUI was deemed by RealTest to be simple enough to be used by the subcontractors. The data from execution was successfully stored in a database as requirement 4 in section 1.3.1 states.

The optional goals of showing previous test results and the ability to edit and create test recipes was implemented.
There are still some glitches in the core system that need to be fixed and the GUI needs more polish. The problems with the Test sequence that are talked about in section 6.1 need to be fixed as well. It is possible at the moment to loop a TestCase over and over again if it sets its failing trigger to trigger itself. The loop would continuously loop forever until the TestCase is successful. If the test is never going to succeed that is probably because there is something wrong the the unit under test, therefore it is appropriate that the test system would move on and report this problem to the operator. This could be avoided by having a counter as a RunTime-parameter that would count to a certain amount of times then either skip the TestCase or terminate the TestRecipe. Other solutions might exists and it could be worth looking in to.

There is a plan in place for how the system would be scaled up if it would ever be required. It is important to note that this has not in any way been implemented or tested and has only been thought about during the design.
The system has also proven to be modifiable and when the MT does change in the future this system will still be useful.

Next step for this system is to implement more TestCaseExecutors so that it can get a broader use. A cable tester is being developed by RealTest that will test cables used by the MT so an executor for that system is suggested.
There is a lot of room for improvement for the GUI and for improving the menus and work flow, which would increase the user experience. The GUI application could also use some patches to fix bugs and glitches.
To be able to scale the system there would have to be more implementations on how the Administrator recognizes different Orchestrators. Currently the Administrator has a static IP address where it expects to find an Orchestrator. This IP address is then used by the Orchestrator Executor to send messages to it. There could be a need for handling multiple Orchestrators at the same time. If for example one Orchestrator does not provide enough services then two Orchestrators could be used to cover all of the services needed. There could also be a better system for general use for other types of measurement and signal generating units to keep track on how to communicate with them.

References

[1] R. Perrey, M. Lycett, *"Service-Oriented Architecture"*, Dep. of Information Systems and Computing, Brunel University, Uxbridge UB8 3PH, 2003.

[2] Kishore Channabasavaiah and Kerrie Holley, and Edward M. Tuggle, JR. *"Migrating to a service-oriented architecture"*, IBM Corporation Software Group United States of America, April 2004.

[3] Sidney W. A. Decker *"Ten Questions About Human Error"*, Lund University, 2005.

[4] Carsten Bormann, *www.CoAP.technology*, retrieved 17-03-07.

[5] Dingyan Zheng, Shuyue Zhang *"Test Automation for Automotive Embedded Systems"*, Department of Computer Science and Engineering Chalmers University of Technology and University of Gothenburg, 2017.

[6] Eng Keong Lua, Jon Crowcroft, and Marcelo Pias, *"A Survey and Comparison of Peer-to-peer Overlay Network Schemes"*, University of Cambridge Ravi Sharma, Nanyang Technolohical Uuniversity Steven Lim, Microsoft Asia, (Volume 7, No. 2), q2 2005.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *"Elements of Reusable Object-Oriented Software"*, 1994.

[8] A. Melnikov, Google, Inc, *"Web-Socket"*, `https://tools.ietf.org/html/rfc6455`, retrieved 17-05-10.

[9] *"Arduino"*, `https://www.arduino.cc/en/Guide/Introduction`, retrieved 17-05-10.

[10] *"A Relational Database Overview"*, `https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html`, retrieved 17-05-10.

[11] *"DB-Engines Ranking"*, `https://db-engines.com/en/ranking`, retrieved 17-05-10.

[12] *"Understanding a Modular Instrumentation System for Automated Test"*, `http://www.ni.com/white-paper/4426/en/`, 2013-04-02.

[13] Gilbert Held *"Best Practicies Series, Server Management"*, 1999.

[14] Macdonald, Matthew. *"Peer-to-peer architecture."* Peer-to-Peer with VB. NET. Apress, (page 23-31.), 2003.

[15] *"What is Scalability?"* `https://msdn.microsoft.com/en-us/library/aa578023.aspx`, retrieved 2017-05-17.

[16] R. Fielding, UC Irvine, J. Gettys, Compaq/W3C, J. Mogul, Compaq, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, T. Berners-Lee, W3C/MIT, *"Hypertext Transfer Protocol"*, `https://tools.ietf.org/html/rfc2616`, retrieved 17-05-17.

[17] Z. Shelby, ARM, K. Hartke, C. Bormann, *"The Constrained Application Protocol"*, `https://tools.ietf.org/html/rfc7252`, retrieved 17-05-17.

[18] *"REST Service Contracts and Late Binding"* `http://whatisrest.com/rest_service_contracts/rest_service_contracts_and_late_binding`, retrieved 2017-05-18.

[19] *"MQTT"*, `http://mqtt.org/2`, retrieved 17-06-05.

[20] *"Oasis MQTT v3.1.1"*, `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718095`, retrieved 17-06-15

[21] *"MQTT Essentials Part 3: Client, Broker and Connection Establishment"*, `http://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment`, retrieved 17-06-15

[22] PerOlof Bengtsson, Nico Lassing, Jan Bosh, Hans van Vliet, *"Architecture-level modifiability analysis (ALMA)"*, `www.ElsevierComputerScience.com`, The Journal of Systems and Software, 2004.