Experiences from Test-Driven Development for Prototyping Software in Commercial Vehicles Industry

by

Mihael Ursic

LIU-IDA/LITH-EX-A--14/068--SE

2014-12-26
Final Thesis

Experiences from Test-Driven Development for Prototyping Software in Commercial Vehicles Industry

by

Mihael Ursic

LIU-IDA/LITH-EX-A--14/068--SE

2014-12-26

Supervisor: Anders Fröberg
Examiner: Erik Berglund
Abstract

This master’s thesis, carried out at MAN Truck & Bus AG, presents a self-observational case study of the software development methodology Test-driven development (TDD) in the context of developing a framework for human-machine interface concepts in commercial vehicles. Software developers are constantly looking for ways to improve productivity and the quality of their code. TDD has been said to do precisely this, but not many experience reports from new practitioners can be found, making it difficult to know what to expect when using it for the first time. This thesis focuses on the experiences of a beginner to the TDD practice and follows the development of the framework and the changes made to the design over time. The framework, consisting of a C++ server application and an Android client, was developed using TDD. Decisions, obstacles and general experiences from the development process are documented in this report with the aim of finding out how TDD works in practice for a beginner and how well the practice is suited for this particular kind of project.

It was concluded that TDD seems to have both benefits and drawbacks, as it appears to facilitate lower coupling of the code and a more structured design, but also complicates the changing of public interfaces since the changes often also affect the test code. Subjectively perceived effects of the practice included that developer focus improved, that testing actually occurred and that the continuous passing of tests gave confidence and a sense of accomplishment to the developer. Furthermore it was concluded that experienced developers may reap more benefits from TDD whereas developers with little experience might have a harder time adjusting to the practice and may not see any notable improvement on the design. The observed developer in this study also found that TDD was difficult to get used to and that it would have been helpful to initially pair up with an experienced TDD practitioner to be properly guided through the first steps and to form good habits.

Some parts of the framework developed were left out of the TDD because of complexity and time reasons, leading the suitability of the practice for similar frameworks to be judged as moderate. The areas that induced problematic situations were multithreading, networking and graphical user interfaces which were all considered difficult to handle with TDD.
I would like to thank Andreas Froschermeier for introducing me to the EPASS department of MAN Truck & Bus AG, as well as Thomas Megerle and Stefan Schleufe for giving me the opportunity to write my master’s thesis on an interesting topic under their supervision. I also thank Stefan Schleufe for the support and help given to me during the thesis work.

Furthermore I thank my examiner at Linköping University, Erik Berglund, for helping me to define the thesis topic and for the feedback I received on ideas and on this report during the thesis work.

I also extend my gratitude to my fellow thesis writers and interns at MAN; Mouli Karanam, Hugo Pascual and Julia Voss, for making the time at MAN very enjoyable.

Lastly I thank my family and friends who have supported me throughout my studies.
Contents

Chapter 1 Introduction ................................................................................................................... 1
  1.1 Motivation .......................................................................................................................... 1
  1.1 Context............................................................................................................................... 1
  1.2 Aim ..................................................................................................................................... 2
  1.3 Scope and Limitations ......................................................................................................... 2

Chapter 2 Theoretical Background ............................................................................................... 3
  2.1 Case Study ........................................................................................................................ 3
  2.2 Data Collection .................................................................................................................. 3
  2.3 Software Testing ............................................................................................................... 4
    2.3.1 Test Doubles ................................................................................................................ 4
  2.4 Test-Driven Development .................................................................................................. 5
    2.4.1 Test-Driven Development Principles ......................................................................... 7
  2.5 Common Mistakes Made When Using Test-Driven Development ..................................... 8
  2.6 A More Nuanced View of the Principles and Concepts of Test-Driven Development ............. 9
    2.6.1 Results of Study .......................................................................................................... 9
  2.7 Effects of Test-Driven Development .................................................................................. 10
    2.7.1 Summary ................................................................................................................... 12

Chapter 3 Method ......................................................................................................................... 15
  3.1 Case Study ........................................................................................................................ 15
  3.2 Literature Research ........................................................................................................... 15

Chapter 4 Development ............................................................................................................... 17
  4.1 Requirements Specification ............................................................................................... 17
  4.2 Server Application ............................................................................................................ 17
    4.2.1 Choice of Development Environment for Server-Side Development ......................... 18
    4.2.2 Implementation .......................................................................................................... 18
## Contents

4.3 Client Application............................................................................................................. 26  
4.3.1 Choice of Development Environment for Client-Side Development.............................................. 26  
4.3.2 Implementation ........................................................................................................... 26  
4.4 General Observations and Lessons Learned ........................................................................... 29  
4.4.1 Use of Version Control – An Important Factor........................................................................ 29  
4.4.2 Forgetting to Refactor .................................................................................................. 30  
4.4.3 Test-Driven Development and Focus .............................................................................. 30  

### Chapter 5 Results ............................................................................................................ 31  
5.1 Server Application .......................................................................................................... 31  
5.1.1 Final Server Application Design .................................................................................... 32  
5.2 Client Application ........................................................................................................... 32  

### Chapter 6 Evaluation and Discussion ............................................................................... 35  
6.1 Development .................................................................................................................. 35  
6.1.1 Effects of Test-Driven Development .............................................................................. 35  
6.1.2 Test-Driven Development as a Practice ......................................................................... 37  
6.1.3 Mistakes When Performing Test-Driven Development ......................................................... 38  
6.1.4 Test-Driven Development of an HMI Prototyping Framework ............................................ 39  
6.1.5 Tools and Programs Used ............................................................................................. 40  
6.2 Validity ............................................................................................................................ 41  
6.3 Future Work ..................................................................................................................... 41  

### Chapter 7 Conclusion ..................................................................................................... 43  
7.1 Recommendations ........................................................................................................... 43  

References ............................................................................................................................... 45
List of Figures

Figure 1-1: System overview diagram where the framework is the central part........................................2
Figure 2-1: The TDD process. .....................................................................................................................6
Figure 4-1: Upfront design of the server application ................................................................................18
Figure 4-2: Early architectural design of server application with central Server class. ...........................19
Figure 4-3: ConnectionHandler to connect to the network. .................................................................22
Figure 4-4: Design including the RuntimeInfoFileLogger .......................................................................23
Figure 4-5: ConnectionHandler replaced with NetworkConnector and NetworkProtocolHandler ........24
Figure 4-6: Server as Mediator. .................................................................................................................25
Figure 4-7: SocketFactory introduced to the client application. ...............................................................27
Figure 4-8: Design with a Client class. ........................................................................................................28
Figure 4-9: ClientView interface introduced. ............................................................................................28
Figure 4-10: Example use of the client application framework. ...............................................................29
Figure 5-1: Final architectural design of the server application. .............................................................31
Figure 5-2: Screenshot of the running example client application (from emulator). ...............................33
Chapter 1 Introduction

This chapter intends to give the reader an introduction to the topic of the thesis. It will motivate the study performed and describe the context in which it has been performed, as well as present the aim and limitations of the study.

1.1 Motivation

Test-driven development is a software development practice that has been widely used and that has received a lot of attention, partly in the context of the software development methodology eXtreme Programming [1]. Its effects have been studied, but results vary and no definite conclusions can therefore be drawn. Existing research in the form of comparative case studies and experiments has mainly been made using professional programmers, mostly at least somewhat familiar with TDD, although there are studies with students not previously familiar with the practice.

This thesis aims to look at TDD in practice from the perspective of a non-professional developer completely new to the practice. Another interesting difference between this study and many others is that it will look at TDD performed by one single developer, as opposed to a team of developers. This study, most importantly, focuses mostly on the experiences of the developer, rather than on quantitatively measurable effects. The resulting experiences and reflections can be interesting for other first-time practitioners, developers thinking about using the TDD for the first time and for companies considering introducing employees to the practice.

1.1 Context

MAN Truck & Bus AG (henceforth MAN) is one of the leading truck manufacturers in the world. Within MAN, the division Engineering Powertrain Alternative Drivetrain (EPA) is responsible for innovative and fuel efficient drivetrain solutions, such as hybrid powered commercial vehicles. Currently, modern human-machine interface (HMI) concepts are finding their way into commercial vehicles. These concepts comprise mainly the visualization of operational data, such as energy consumption or energy flow. Future concepts, called economy trainers, are planned to be developed in order to support the driver in the reduction of fuel consumption.

Previously, the EPA department has been bound to develop, present and discuss the needed HMI functions based only on drawings and presentations in Microsoft PowerPoint and similar. Because of the limitations of these static methods, the need for an HMI-prototyping environment, which would enable the department to rapidly demonstrate dynamic HMI concepts and test them in prototype vehicles, has been identified.

This study will look into TDD in the context of developing a framework meant to be used as a base when creating such concepts.
1.2 Aim

The aim of this thesis is to collect experiences and reflections from a first-time practitioner of the software development practice Test-driven development in a real-life software development project. In particular, the goal is to answer the following questions.

- How does TDD work in practice for someone with no prior experience of the technique?
  - What possible problems or obstacles can occur? How can these be avoided or solved?
- What are the perceived advantages, disadvantages and effects of using TDD?
- How well is TDD suited for developing a system such as the one described under Context above?
- How well is TDD suited for developing network applications?

1.3 Scope and Limitations

MAN’s vision of the system gives some initial requirements and limitations to the implementation part of the project. For the product to fill the needs of the company, it should consist of a server that simulates vehicle data, such as speed and fuel consumption, as well as a separate remote device that should request and receive this information over an Ethernet connection and display this on the screen. Furthermore, the core server code had to be platform independent and written in C++, to accommodate the flexible needs of the company. The remote device will, in the scope of this thesis, be limited to an Android emulator. The initial visual description of the framework and surrounding parts is presented in Figure 1-1.

![Figure 1-1: System overview diagram where the framework is the central part.](image)

The thesis focuses on the use of TDD in the base case and will not explore more advanced uses or variations of the practice to any greater extent (e.g. UITDD). Availability of tools and IDEs (Integrated Development Environments) was limited to the selection provided by MAN. Thus, the choice of tools used in the scope of this thesis was made within this selection.
Chapter 2 Theoretical Background

This section covers relevant background for research methodology, a brief description of some important software testing concepts and terminology, as well as a description of Test-driven development, along with a presentation of previous studies on TDD and its effects.

2.1 Case Study

The purposes of different research methods differ [2]. Four general types of purposes have been identified; exploratory, descriptive, explanatory and improving. The exploratory purpose means to create hypotheses by finding out what is happening, the descriptive to create a picture of phenomenon’s or situations’ current state, the explanatory to explain a problem or situation, and the improving to improve something about the phenomenon under study.

Furthermore, a study can be either deductive, meaning that a hypothesis exist from the start and the study aims to confirm or reject it; or inductive, where observations are made in the study, and hypotheses are formed from the results and related to existing studies [2]. Here connections between exploratory and inductive, and explanatory and deductive research can be made, where exploratory research is more inductive and explanatory more deductive.

According to [2], the following is a good definition of case study, specific for the context of software engineering.

Case study in software engineering is an empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified [2]

In other words, a case study investigates a real-life situation, where context and phenomenon are not easily separable, using multiple sources of data.

2.2 Data Collection

Empirical studies can be performed collecting either qualitative or quantitative data, or both [2]. The quantitative data consists of numbers and classes – in other words statistical data. Qualitative data, on the other hand, consists of descriptions made with words and pictures, etc. In case studies, mostly qualitative data is collected, as it is considered to give descriptions of more rich and deep character.

Especially with qualitative data, the use of triangulation gives an important increase of the precision of the study [2]. This means using more than one data source, more than one observer, more than one method and/or more than one theory/viewpoint to create a better understanding of the studied situation.
2.3 Software Testing

Software testing is the “act of exercising software with test cases” with the aim to show that the execution is correct and to find failures [3]. A test case consists of – apart from the actual test – definitions of the inputs given to the code under test and the outputs received from it. Test cases contain the preconditions (the circumstances before the test is executed), the actual input (what the test actually does), the expected output (what the direct results of the input are supposed to be), and the postconditions (the circumstances that should be present after the test execution).

There are several approaches to testing in software [3]. One of these is black box testing, or functional testing, in which the software’s specification is used to create test cases. In other words, the specification gives all needed information to specify pre- and postconditions, inputs and outputs. Another approach to testing is white box testing, or structural testing. Here the implementation of the software under test is known, and also used to create test cases. This way it is possible to test the actual implementation and more specific parts of the software “under the hood”.

Furthermore, software testing can be divided into different levels. Jorgensen [3] mentions unit testing, integration testing, and system testing and also makes connections to their corresponding waterfall design levels; detailed design, preliminary design and requirements specification, respectively. Here unit testing means testing the implementation – the smallest possible testable part [1], integration testing means testing subsystems, or combinations of units, and system testing means testing the complete system [4].

Test automation means using a software tool to automate the testing process [5]. This is a very important part of software testing as it allows for many fast test runs without any effort from the developer/tester (except specifying and coding test cases). With automated tests, it is also possible to ensure that all tests get run the same way every time. Furthermore it enables regression testing; testing to make sure that the software did not break after a change, which would be too expensive to perform manually. The automated tests can be created using some dedicated testing software that makes the testing process simpler and often provides some additional information and statistics related to the testing.

One important thing to keep in mind regarding software testing is that it can only show the existence of failures, and never prove that the software does not contain any errors [4]. Because of the many different combinations of inputs in most systems, exhaustive testing, in which all combinations are tested, is not practically viable.

In order to get an idea of how thoroughly the software has been tested, the measurement test coverage can be used [6]. Test coverage gives a quantitative quality measure of the testing and provides a mean for determining when enough testing has been done. There are a few more specific variations of code coverage, but basically it shows how many lines of production code that have been executed by the tests.

2.3.1 Test Doubles

In the context of unit testing, the terms stub, fake, dummy and mock are often mentioned. They are used when the code under test makes calls to other units [7]. The fakes, stubs, dummies or mocks are then used to represent these other units, in order to reduce the complexity of the test cases. It also removes the possibility that the test case fails because of some other piece of code (another unit) that is not under test. A generic term for these types is test double. Martin Fowler [8] gives the following summarized explanation of the differences between the different types of test doubles⁴.

---

⁴ A more elaborate description about this terminology and the use of the different types of test doubles is provided by G. Meszarios, xUnit Test Patterns: Refactoring Test Code, Westford, MA: Pearson Education, Inc., 2007.
Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.

Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).

Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.

Mocks are objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

(My emphases)

2.4 Test-Driven Development

Test-driven development (TDD) is a practice in software development, where automated software tests are written before the functional code. [9] [1] As a practice, different forms of TDD have been used by developers for several decades [1]. Before getting its name, the practice as such was being used in software development, although more informally. The formal TDD, however, takes the idea of testing first a bit further. Here tests are always written first, the tests are kept as small as possible, and the functional code is just enough to pass the tests. TDD is also an important part of the software development methodology eXtreme Programming (XP), where it is said to be necessary for analysis, design and testing. After the introduction of XP, TDD has been subject to more attention, not only as a part of XP, but also as an independent practice. Different names have been used for the practice, including Test-driven design, Test-first design and Test-first programming.

The TDD process consists mainly of three steps, repeated in an iterative manner as a cycle [10]. First, the developer comes up with a way to test some functionality that is to be added, and thereafter a corresponding automated test is written. Now, as there is actually no production code corresponding to this test, the test might at first not even compile, but this is all according to the TDD practice. The next step is then to make the test compile by adding just enough code for that to happen. Now the test will (or should) fail when run. This is the time to write the actual production code; but only just enough to make the test case pass and nothing more. When the new test and all other tests pass it is time for the next step; refactoring. Important to note is that all tests should pass before moving on. This is to make sure that the newly added code does not break any other piece of functionality. Furthermore it should be noted that a newly added test should fail at least once before passing, to make sure that the test does not always pass.

The third and last step of the TDD cycle is refactoring [10], [1]. In order for TDD to become more than just a testing technique, and let it be part of the design and analysis part of development, this step is necessary. Refactoring means rewriting code to remove duplication and change its structure while still maintaining the same external behavior of it. In the context of TDD, the idea is to stop after a test passes to make improvements to the code if possible. By running all tests again after the code changes, it can be asserted that the change was actually a refactoring since passing tests mean that the same external behavior persists.

The three steps of TDD are thus to write a test that will fail, write just enough code to make it pass and finally to refactor the code. In a shorter form this is often referred to as red-green-refactor\textsuperscript{2}, which has also been called the TDD mantra [10]. A visual representation of the TDD cycle is presented in Figure 2-1.

\textsuperscript{2} This refers to the use of testing software which usually presents the developer with a red bar if a test does not pass and a green bar if it does.
Though TDD means that the developer now has to maintain both test and production code as the program changes, it also provides the developer with quick feedback through the tests [1]. Because of this feedback, the developers are said to be more comfortable making changes to the code. If the changes break something, the tests will tell them – fast. TDD is also often said to produce “clean code that works”, meaning that the automatic tests demonstrate (to some extent) that the software works and that the constant refactoring makes it clean [11].

To take small steps in TDD is something that is also often considered important. When searching the Internet for introductory examples of TDD in practice, or looking at Kent Beck’s examples in [10], one can see how small these steps actually are. It can be as small as letting a method return a hardcoded value if that will make the test pass. This is quite counterintuitive, especially since the developer already knows that the implementation will not be sufficient. If the implementation passes all tests, however, it is all according to the process of TDD. By adding new tests, the hardcoded solution will fail, and the code has to be changed. The reason behind this – seemingly somewhat ridiculous – principle is to help the developer to stay focused on the task at hand. When trying to take bigger, more complex steps at once, it is easy to get distracted by changes and refactoring needed in the surrounding code, possibly leading the developer to forget what the task was in the first place. TDD lets - or makes - the programmer focus on one little step, even if it means that the code will only be temporary. This gives the programmer confidence, as some functionality has been added successfully, and because it gives the programmer a point to return to in case something does not work in the next step. Christensen [11] summarizes this into two principles; keeping focus and taking small steps.
2.4.1 Test-Driven Development Principles

Christensen [11] describes a number of principles of the TDD process, also described by Beck [10]. These are presented below.

**Test first** – This is the core of TDD. The tests should be written before the production code. This removes the possibility to forget (or “choose to forget”) to write tests, and lets the developer pinpoint the location of a possible future failure.

**Automated test** – When using TDD all the tests are executed very often, since the developer has to make sure that a code change did not break any other code. Manual testing would be practically impossible when not dealing with a very small product. Therefore automated testing is an important part of TDD.

**Test list** – Answers the question of what should be tested. Christensen [11] recommends the developer to write a list of all tests that he/she knows will have to be written. When a new idea appears or a discovery is made, new tests are added to this list. The following quote comparing the test list with a grocery shopping list makes a good description of the test list. “It keeps me focused, I don’t forget a lot of things, but it does not exclude me from putting some extra stuff in the shopping bag” [11]. The list does not have to be perfect and too much time should not be spent trying to get it to be. Instead the developer should try to move forward fast; the purpose of the list is to help to keep focus and to help remembering. Beck [10] does not limit the list to contain only tests, but simply uses it as a to-do list, where tests are mixed with necessary refactoring, etc.

**One step test** – The programmer should pick a test that will teach him/her something and which he/she is confident can be implemented. In other words, the programmer should not pick a test that is too simple, because there is no learning possibility there. Nor should a too complex test, for which there are no good implementation ideas, be picked.

**Fake it (’til you make it)** – This relates to writing the simplest possible production code that will pass the tests. Returning a hardcoded value is perfectly fine, even though the programmer is aware that the implementation is far from complete. The idea is to get the test to pass and then change the code step by step. A way to look at this somewhat absurd principle is that the aim of a particular step/iteration is not to provide a complete implementation of the functionality tested, but to handle a specific test case.

**Triangulation** – In the context of TDD, triangulation means that at least two examples are needed in order to generalize and “drive an abstraction [...] into existence” [10]. In other words, when “fake-it”-code is written, it has to be triangulated in one of the following steps/iterations. To make sure this is not forgotten, the test list is updated with added triangulation tests.

**Isolated test** – The amount of testing performed in one test case should be limited, and isolated tests should be in isolated test cases. This makes it easier to pinpoint failures and to identify what actually went wrong if failures should occur.

**Evident data** – The tests should be written in a readable way, so that also other developers can easily understand what is being tested and how. Therefore calculations of expected values, etc. can be explicitly written out in the test case code, instead of only the values. Calculations or descriptions can also be written in comments for clarification purposes.
Representative data – The data elements used in the tests should each represent a unique “conceptual aspect or a special computational processing”. This means that not all possible values need to be tested, but simply one of every important kind. As an example, consider a function isNegative(int value) returning true if the provided integer is negative. To test this method it is perhaps not necessary to test more than three values; one negative, zero, and a larger positive value.

Assert first – This principle tells the developer to write the assertions for a test case first, and focus on the setup and design of the test case later. This helps the developer to see what is actually needed for the test case, before starting to write it. Just like the test-first principle is a bit backwards from traditional development, this principle reverses the test design by starting at the end.

Obvious implementation – This principle says that simple operations should just be implemented in one go, instead of using ridiculously small steps.

Evident tests – The test code should be kept simple and readable. It should not contain unnecessary complexity that might improve performance a little bit, or that simply looks better. This is because complexity increases the possibility for defects. Loops, recursion, etc. should be avoided as much as possible. In a perfect TDD world, test cases should only contain assignments, method calls, and assertions.

Break – Christensen presents this principle for TDD, which probably many would agree covers all programming and many other demanding processes as well. It simply states that you should take a break when you are tired or if you are feeling stuck.

Some of these principles might contradict the strictly theoretical view of TDD to a small degree. However, both Christensen and Beck lead the readers though the process of TDD with examples in [11] and [10], respectively, thereby aiming these principles at TDD in practice. The following sections will give some complementary views and opinions on what is right and wrong in TDD.

2.5 Common Mistakes Made When Using Test-Driven Development

A survey looking into common mistakes made when using TDD is presented by Aniche and Gerosa in [12]. The survey was posted online and 218 software developers volunteered to answer it. The questions asked were specific to TDD and not software development in general. 75% of the respondents had used TDD for a maximum of three years, and 90% used it in industry, 50% in open source projects and 20% in academy.

On average, developers evaluated their first time using TDD low, indicating that it is difficult to learn or get accustomed to. The most common mistakes in TDD were identified as (from most to least common):

1. Writing complex test scenarios, instead of refactoring to simplify.
2. Forgetting to refactor code after a test passes.
3. Refactoring another part of the code, when still working on a test.
4. Not beginning with the simplest possible test.
5. Not implementing the simplest code to pass the test.
6. Using bad names for the tests.
7. Not making sure that a new test fails before implementing the corresponding production code.

It can also be noted that Kent Beck is one of the inventors of TDD. His description of the process and principles of TDD should therefore be representative of the “official” TDD technique. For more details and examples relating to the principles of TDD, see [11] and [10].
8. Not refactoring test code.
9. Not running the whole suite of tests, but only the current test case.

Validity in this study is subject to for example the following considerations. The results are based on programmers’ own opinions and self-evaluation which might not give accurate results. The questions asked were based on observed mistakes in other studies. It could very well be the case that there are other common mistakes\(^4\) that were not investigated in this study, thereby resulting in a somewhat distorted view of the reality.

### 2.6 A More Nuanced View of the Principles and Concepts of Test-Driven Development

A qualitative analysis of TDD among beginners to the practice was conducted in the form of a discussion session by Aniche et al. [13]. Most of the subjects had 0-3 years of experience with TDD and more than three years of total software development experience. Notes were taken during the discussion to document opinions and feelings, and the discussion was recorded and transcribed to provide further data.

#### 2.6.1 Results of Study

On the question about whether TDD is a design method or a testing method, most participants in [13] agreed it is a design method. It was for instance said that the developers have to decouple the code in order to be able to write unit tests, and that the fast feedback given by the automated tests enables the developers to find design flaws early, and consequently fix them. Furthermore it was said that TDD makes the developers write simpler code, since they themselves have to use it in the tests. Only one participant (out of 10) considered TDD a testing technique as opposed to a development practice.

Regarding refactoring, most participants found that TDD provided them with greater confidence to make code changes, since the tests would warn them if something went wrong.

Since the practice of TDD differs from traditional coding it might be hard to get used to. This was expressed by the participants, as none of them actually believed in TDD when introduced to it, and it was mentioned several times in the discussion that learning TDD is not easy. Almost all of them reported not feeling very productive in the beginning of using TDD, but that it is better in medium terms, as bug fixing is easier. They agreed that the benefits of TDD are not immediate, but appear after a while of using it.

According to the discussion, TDD in itself is not a solution for all design problems, but it helps to create a good design faster than with a traditional technique.

A mix of opinions was discovered when discussing the question of the so called baby steps in TDD - that is making as little as possible to pass a test. Some believed that this should be followed strictly, as the developer might otherwise forget to test some corner cases, resulting in later problems when refactoring. Others were of the opinion that the steps could be adjusted according to developer experience. They said that always doing baby steps would not be productive. The authors discuss this in the article, saying that the concept of baby steps is often misunderstood. According to them, it exists to promote a simple solution in favor of a complex one when both solve the problem. It is not meant to make sure corner cases get tested, which would be more related to testing than design.

Also in this study, important validity concerns are raised by the authors. Many participants of the study showed good knowledge about TDD – both practical and theoretical – but whether what they said in the discussion actually corresponds to their normal use of the practice could not be determined. Some of the

---

participants also mentioned practices that differ from the theoretical TDD, and some had no experience of using TDD in professional context. The selection of the ten participants, as well as the small number in itself, sets limitations on the generalization of the findings. All of these factors should be taken into consideration when drawing conclusions from the results.

### 2.7 Effects of Test-Driven Development

It is not hard to simply speculate about the possible effects of using TDD. One could for example assume that, since many tests need to be written, the development process will take more time, and that the many tests will make the code better, or at least in some way “safer”. These assumptions could be based purely on speculation or on anecdotal data, but in order to get more reliable data about the effects of TDD, studies that thoroughly evaluate TDD and its effects need to be looked at. Fortunately, such evaluating studies and case studies involving TDD have been made. In this chapter a few of these studies and their findings are presented.

In 2004 George and Williams [9] published an article titled “A structured experiment of test-driven development” which presents some observed effects of TDD. The study was focused around two hypotheses. The first one was that code written with TDD would have better quality than code written in a more traditional manner (design-develop-test), and the second one was that code would be written faster with TDD than with the more traditional method. The code quality was measured by the number of passed functional black-box tests. To measure the speed of code writing, the total time of development was used. Empirical data was obtained from three experiments, where the test groups consisted of professional programmers. All groups were asked to develop an application based on a set of requirements.

The results presented by George and Williams [9] show that the code written in the style of TDD passed around 18% more of the black-box test cases than the code written in the more traditional manner, confirming the first hypothesis. The second hypothesis was however not supported by the results, as TDD developers needed around 16% more time to finish the application. Further analysis of the results suggests that the higher code quality of the TDD developers might be a direct effect of the practice itself, but could also be a result of the additional time used in developing. Also the code coverage of the tests written by the TDD developers was analyzed, and it was found that it - on average - exceeded industry standard.

George and Williams [9] also presented the results of a qualitative survey conducted with the developers in the experiment. They found that the general experience among the developers was that TDD improves productivity and that it is an effective way to produce code with good quality. It was also noted that developers found that TDD aids less complicated design, and that not having a design beforehand was not a problem. Some developers did however also express that it was hard “getting into [the] TDD mindset”.

The validity of the study [9] is somewhat limited, as will be described, but the results can nevertheless provide some indication of the effects of TDD. Possibly affecting the results of the study are these factors brought up by the authors: the small size of the study, a modification of the experiment after the first trial, the fact that pair programming was used in all groups, that the application was relatively small, and that the programmers had varying experience of TDD and pair programming. Furthermore, only one control group wrote any meaningful tests (although they were all specifically asked to do so), which affects the result regarding programmer productivity.

In “Evaluating advantages of test driven development: a controlled experiment with professionals”, Canfora et al. [14] present a study similar to that of George and Williams presented in [9]. Similarly to George and Williams, the study was conducted with a focus around two questions. The first one aimed to find out whether TDD is more productive than a more traditional test-after approach and the second one whether TDD improves unit testing quality.

The experiment was carried out by letting the 28 participants, all knowledgeable in software development and introduced to TDD with training sessions, implement and test a specified system. During development
the subjects were required to fill out forms to provide more relevant data to the study. Each subject had two assignments, carried out in two different runs. Every subject completed one of the assignments using the traditional approach, and the other using TDD, although the assignments and development approaches were mixed among the subjects so that both assignments were performed using both approaches.

The study [14] concluded that TDD is slower than the traditional approach. The authors do, however, believe that the extra time was used to improve code quality, though they were not able to do more than speculate about this. A second conclusion of the study was that it could not give evidence to support the hypothesis that development with TDD gives more precise and accurate test cases. It was observed that test cases from the traditional approach were grouped together to form larger test cases, which, according to the authors, affects the precision of the assertions. The authors mean that developers are more likely to miss assertions because they don’t divide the problem into smaller parts, but only look at it as a whole. This, they claim, affects the quality of the unit testing. It was also noted that TDD had lower standard deviation in all the metrics, which means that development using this practice is more predictable, which in turn is good when planning projects. The authors present a possible explanation for this, saying that with TDD all developers have to write roughly the same amount of tests with approximately the same precision, while the traditional approach leaves it up to the individual developer to decide how much testing will be performed, leading to more unpredictable time usage.

Validity concerns expressed by the authors for the results of the study [14] include the following. It is possible that the time window given to the subjects was too small, leading them to hurry and thereby not perform all actions that they would wish. This is a concern, since TDD takes a lot of time. Also the scope of the tasks was limited to be relatively small, which might affect the ability to generalize from the results.

Another study, conducted by Siniaalto and Abrahamsson, is described in the article “A comparative case study on the impact of test-driven development on program design and test coverage” [15]. Here, three case projects were studied, all with the aim to develop a real software product to a real customer. The teams worked for nine weeks, doing six iterations following an agile development method. Two teams used an iterative test-last development process, while the third one used TDD. As the teams were developing software for real customers, they did not all develop the same application. The team members were all undergraduates with 5-6 years of studies behind them. The members of the two test-last teams had previously worked in the industry, while only one in the TDD team had industrial experience. The TDD team did, however, consist solely of Software Production majors. All teams were encouraged to write tests.

The metrics used by Siniaalto and Abrahamsson [15] indicated that somewhat less coupled code was produced using TDD, but the differences were small between the projects and the coupling was low also in the other projects. Therefore it is the authors’ opinion that conclusions cannot be drawn from this result. The test coverage in the TDD project was significantly better than in the other projects. Since all teams were encouraged to write tests, this indicates that TDD produces more thorough tests. The authors also note that all projects finished on time, meaning that the use of TDD did not affect productivity. Further results indicated that TDD may require some professional expertise and that the effects of TDD do not have to be good, when performed by less experienced programmers.

Validity concerns expressed in the article [15] include the following. The experience level of the team members varied. They point out the possibility that the programmers in the TDD team might have been less experienced as they were not professionals, but also bring up that a professional programmer is not necessarily more skilled than a non-professional one. Furthermore, as previously mentioned, the teams developed different products. Measures were taken to make the comparison fair, but this may still have affected the results. The size of the projects may pose a limitation to the generalization of the results, as they each amounted to less than 10 000 lines of code and were written in approximately 1 000 man hours.

In the article “Assessing test-driven development at IBM”, Maximilien and Williams [16] present results and lessons learned from a TDD project at IBM. It had been observed that no tests, or only last-minute tests, were usually written at IBM Retail Store Solutions. Therefore it was decided to try TDD when developing a
non-trivial system. Many developers were new to the practice and also to the used programming language, Java, and therefore had to learn during development.

A 50% improvement in the defect rate was observed and attributed to the fact that when using TDD, testing actually occurred [16]. No great effect on productivity could be detected, although it was slightly lower than without TDD. The authors also point out that this result might be a bit off, since a new project management software was used, which was believed to have improved productivity. A perceived benefit to the design was observed, as the authors believed it to be easier to make late changes. Furthermore, the use of daily integration and regression tests was described as reducing late integration problems. Another observation was that the developers liked the TDD practice and have continued to use it after the project.

Maximilien and Williams [16] also share a number of recommendations based on their experiences in the project. Two of these are presented below.

- Start using TDD from the start of the project and make sure the developers know that the productivity will not be affected to any greater extent, as they might initially be frustrated with the time it takes to write unit tests.
- Add new tests when a problem is found. It does not matter when this occurs, but at least one new test should validate both the existence and removal of the defect.

Presented in [17] is an experiment conducted by Erdogmus et al. also aimed to evaluate the effects of TDD. More specifically this experiment was focused on TDD’s effects on external quality and programmer productivity. The experiment was carried out in an undergraduate programming course. Data was collected from 24 third-year students in this course. Eleven students were part of the test-first group and 13 in the test-last (control) group. To investigate the possibility that TDD is effective simply because it encourages testing, the number of tests written was measured as an intermediate variable. The task was to provide a solution to the Bowling Score Keeper by Robert Martin, previously used in another TDD study. The control group developed this using an iterative test-last approach, in which tests were written after a user story was completed, while the TDD group divided the user stories into smaller pieces and writing the tests before the production code. The development was performed in an incremental manner, with the requirements of the program changing, by adding new features as the old ones were completed.

The TDD subjects in [17] produced a considerably high amount of tests, compared to the control group (the median was 100% higher and the mean 52% higher, however ignoring two subjects of the control group that did not write any tests, results in a TDD mean 28% higher than the control group). The code quality in the groups was very similar, while the productivity of the TDD group was higher (21% higher median, 28% higher mean).

Concluded by the authors was that test-first programmers tend to write more tests per unit of programming effort (in this context a user story), and that a higher number of test lead to a better productivity [17]. Therefore it cannot be concluded that TDD directly increases productivity. The results do however indicate that the end effect of TDD is an (indirect) increase in productivity. Also this study suffers some validity concerns. As only one small, unique program was developed, it can be argued that the results are not completely generalizable. Furthermore the participants in the study were all students, and the results might not be representable to other groups.

2.7.1 Summary

As the selection of studies on the effects of TDD shows, it is not really clear what effects can be expected from using TDD. The results often contradict each other, and many of the results are subject to at least some validity concerns, making it even more difficult to draw any general conclusions. This is true, not only for this selection of studies, but in general for studies about the effects of TDD, as concluded by Kollanus in “Discipline and Practices of TDD (Test Driven Development)” [18]. Kollanus performed a literature review,
looking at 40 empirical studies of TDD, and found that the results often were contradicting. It was concluded that the 40 studies show weak support for improved external quality, very little evidence of improved internal quality, and some evidence of decreased productivity when using TDD. As the studies’ results contradicted each other, it is however difficult to say if these observed effects are all actually effects of TDD.

It can also be concluded that some differences in the actual practice of TDD exist, something that is not really brought up as impacting factors on the results in many studies. Considering that Aniche, Ferreira & Gerosa [13] in their discussion session noticed different opinions among the participants concerning basic concepts of TDD it is possible that different forms of TDD were used in some studies. This could of course affect the results and make it even more difficult to draw any general conclusions on the effects of TDD.
Chapter 3 Method

In this chapter the used methods for the study will be presented.

3.1 Case Study

Since this thesis aims to investigate one real-life situation, where it would be difficult to separate the context from the phenomenon, the case study approach was an appropriate choice of research method. Therefore, an exploratory and inductive case study was used in this thesis in an attempt to fulfil its aim.

As the researcher (observer) and the observed in this study were the same individual, several observers could not be used. The practical collection of data was also limited in the number of available methods (it would be impossible for a researcher to perform an interview with him-/herself and would not make any sense to use a questionnaire for example). Instead, the data collected had to come directly from the subject/researcher. Most of this data consisted of observations, thoughts and reflections made during and after the development. In order to capture these in a good way, and make sure that as little as possible was forgotten, a logbook was maintained. The intent of the logbook was to provide a basis for evaluation after development, especially to be able to determine if and how feelings about TDD, the approach to TDD and the design changed over the course of the development.

3.2 Literature Research

The literature used for the Chapter 2 Theoretical Background was collected using Internet search engines, primarily Google Scholar. Relatively many sources could be found for studies on the topic of TDD. Not all of these were used, as this would be unnecessarily time consuming in the context of this thesis. This, since the literature study primarily is meant to give an idea of what kind of research that has previously been done in the area, and provide some typical results. Furthermore, others have already made substantial literature reviews on the topic of TDD and its effects, with results also presented in this report. Since similar conclusions could be drawn from the chosen selection of studies as was drawn by Kollanus [18] in a literature study of considerable size, covering 40 articles, this selection was considered sufficient as background research on the topic of TDD’s effects.
Attempts were made to find sources covering first-time use of TDD, that is to say studies or experience reports made with absolute beginners in TDD. Only a very small selection of blog posts and similar could be found covering this type of TDD use, of which even fewer provided any substantial amount of information about experiences made, lessons learned, recommendations for other beginners, and similar, which would have been interesting in this project\(^5\). However, sources introducing TDD to beginners could be found, of which Beck [10] was the most useful. When searching the Internet, there are many other introductory sources with practical focus, such as tutorials. These can be found in several forms, including blog posts and videos.

Chapter 4 Development

This chapter will describe the development of the prototyping framework for MAN. First, the framework, its surroundings and intended use will be presented along with a more specific look at the requirements of the system. Following this, the actual development will be described in chronological order, focusing on the use of TDD in practice and the experiences and observations made, as well as obstacles faced during this process. Also presented in this section are important decisions; mainly those related to the TDD practice and principles, but also other decisions regarding the environment and setting for the development. Lastly, some general experiences and observations made during the development will be presented.

4.1 Requirements Specification

Before starting to implement the server application, it was necessary to specify the requirements of it. Up to this point only a conceptual description of the system had been available. The requirements came from MAN and were specified and documented through a discussion with the thesis supervisor.

In short, the requirements specified that the system should consist of a server and a number of Android clients. The idea was that the server would handle incoming connection requests from the clients and decide if they should be able to connect or not. Connected clients would then be able to request subscriptions to a number of data feeds on the server, and the server would send updates about changes to the subscribed clients. The data used should come from the signals of a CAN (Controller Area Network) bus, through a CAN interface. Furthermore it was specified that the server should be able to generate a few different kinds of data, so that the framework itself could be tested and demonstrated without actually connecting it to a CAN interface. It was important that this signal generation provided the same interface to the server application as the real CAN interface, so that they could easily be exchanged.

It was specified that the server should be running on a Windows computer, but that the core functionality had to be platform independent. This core functionality consisted of the most central logic of the server application, such as handling incoming messages, keeping track of registered clients and updating the registered clients with the information they requested. However, the actual network connection was not considered part of the core functionality, since that part would have to be specific for different platforms. Also excluded from the core functionality was the generation of data. The platform independence of the core functionality was wanted because the server application was later to be run on different platforms. It would then be advantageous if the core files could be directly reused and only smaller parts of the application rewritten.

This requirements specification gave an idea of what MAN wanted and also gave some specific design restrictions as described above. Since TDD should be started with only little or no upfront design, this imposed a slight concern for me. It was decided to try to keep the upfront design to a minimum, but in order for the system to actually be useful for MAN some design decisions had to be made beforehand.

4.2 Server Application

The server application was considered the most important and central part of development. Hence it was decided to start the development with this part of the system. It was also decided that the development of the
server and client applications would not be performed in parallel – at least not initially. The reason behind this was to be able to focus more on one part at a time.

### 4.2.1 Choice of Development Environment for Server-Side Development

The choice of IDE (integrated development environment) for developing the server-side of the system was Microsoft Visual Studio Professional 2010. This choice was motivated by the fact that it was the newest version of Microsoft Visual Studio available at MAN and standard for C++ development at the division. Furthermore, unit testing support in C++ is integrated in the IDE, which was considered suitable for TDD. No so called mocking framework that simplifies the creation and use of test doubles was used, so all test doubles were created manually.

### 4.2.2 Implementation

In this section the implementation process and development of the server-side framework and application will be described. First, the initial design will be presented and then changes and additions to it, as well as observations made during development will be presented.

#### 4.2.2.1 Upfront Design of the Server Application

The upfront design was created directly from studying the requirements of the system. There was already a rough overview of the system as a whole (see Figure 1-1), which gave some guidance for the architecture of the system, but a more detailed internal architecture for the server application is shown in Figure 4-1.

![Figure 4-1: Upfront design of the server application.](image)

As can be seen not much of a design existed from the start but rather a list of components; how they would interact was not yet known. That the server application would have clients was a given from the very start, as was the assumption that these somehow would need to be represented locally in the server application. It therefore seemed like an obvious choice to let clients be represented by instances of a Client class. Furthermore, the requirements specified that a configuration file should be provided for the server application, in which a list of available signals, as well as the desired port number of the server and the desired signal source should be specified. The server application would therefore have to be able to read and interpret this file, which led to the decision to create a ConfigFileReader class. Yet another requirement was that the server application should be able to generate signal data. As this was also a quite separate part of the
server application, that was also not considered to be core functionality, it was decided to let the signal generation be handled by another class, MessageGenerator. This class and the real CAN bus interface were also supposed to be interchangeable, leading to the quite obvious decision to let them both implement the same interface, SignalSource. In order for all these components to be able to communicate and to hold the state of the server application it was also soon in the development realized that a central class, Server, was necessary (see Figure 4-2).

![Diagram of server application](image)

**Figure 4-2:** Early architectural design of server application with central Server class.

### 4.2.2.2 Where to Start

One of the most critical parts of the system was the communication between server and clients. It was therefore tempting to start the development with this part to make sure that I could get it to work. As it was, however, also a part considered quite complicated I chose to leave it for later. As the TDD principle *one step test states*, a test that is not too complicated, but that you can still learn something from should be picked. As not beginning with the simplest test was also concluded to be one of the most common mistakes made in TDD by Aniche and Gerosa [12], it was considered a good and well supported plan to leave the complicated issue of communication for later, and focus on something simpler to start with.

Before starting the actual development, the IDE, Visual Studio, was attained and installed. Internet tutorials and forum posts helped with information about how to set up a C++ project with accompanying unit test project. Thereafter a few simple unit tests, such as a trivial function adding two integers, were written to see how the unit testing and TDD would work in practice. These tests themselves were very simple, but it took some time and effort to setup the test project and the production code project correctly before everything would work together.
4.2.2.3 Creating a MessageGenerator

The server application was supposed to be able to generate/simulate data. This part of the application was deemed an appropriate starting point, as it did not seem too difficult and it could still teach me something about the application. As soon as I had created the test class, however, and was going to write the first test, I realized that there was no real simple test to write. There was no method as simple as those found in most examples of TDD in textbooks and on the Internet. I therefore had to start with something more complicated, namely the generation of signal data.

This data was supposed to be generated as follows and updated to the clients every second. The value was supposed to increase linearly from 0 to 100 in 10 seconds and then immediately drop back down to 0 and start over. I tried to divide the test into several parts. First, the value should not be negative and not exceed 100, which was a pretty straight-forward test – simply get the speed data and fail it if it is below 0 or over 100. After getting this to work, a second test was introduced to assert that the value changed over time. Development continued with a few more small tests. Duplication of code was detected, and both test and production code was refactored. It now felt like the design was actually driven a bit by the tests, although the refactoring made was easily predictable. These tests did, however, not test the functionality completely, but only some small aspects of it. It was decided to try “the big test” and test the functionality that I actually wanted to test. Therefore a test for the periodic ramp was written and the corresponding functionality was implemented. Here the red-green-refactor rhythm of the development was not completed in the matter of a few minutes as Beck [10] recommends. The reason for this was simply that I knew what I wanted to do, but did not know exactly how to do it. That meant spending some time doing Internet research, thus lengthening the time between red and green. As the generation of data was supposed to continue over time, it had to be done in a separate thread, and the test had to wait for the thread to be able to determine if the generation worked correctly. This meant that the test actually took more than ten seconds to run.

As the signal generation was now working, the MessageGenerator was left in favor for other functionality. It was deemed that adding generation of another signal would be very similar to the generation already present and would therefore not teach me much about the application.

4.2.2.4 Test-Driven Development Prohibiting Refactoring

As weird as it might sound, TDD actually stopped me from refactoring the code at one point. I was looking through and commenting the code in the MessageGenerator when I realized I could make it look better by using the Observer design pattern, in which a subscription for a data feed would be done to the MessageGenerator which would in turn send updates to its subscribers. The idea was really tempting and I was close to implementing it when I realized that I was not paying attention to what my tests were telling me. I looked for any indications in the tests that could point towards the Observer pattern, but saw that my tests actually told me not to go for the Observer pattern solution. In the current implementation the test actively called the MessageGenerator to get the latest data for a signal. This could of course result in many requests and responses between the components (which is probably why the idea of the Observer pattern came to mind). The signal generation did however update the data much more frequently than what was needed by my test. With the Observer pattern this would mean that the server core application would receive a lot more updates than wanted. With the current solution however, the refresh rate of the signal source was irrelevant to the server core, and the server core could decide for itself when an update was needed. This led me to stay with the current solution – at least until a test would actually tell me to do something different.
4.2.2.5 Creating a ConfigFileReader

The server application was to be provided with a configuration file, from which it would retrieve information such as which signals that were available for subscriptions, which port the server should use on the network and which signal source should be used by the server (MessageGenerator or a real CAN bus interface). I decided to start with testing the reading of the port number. Tests were written, first to see if the production code could produce the same port number as was to be found in the configuration file, then to see if exceptions were thrown when the file did not exist or was badly formatted.

The descriptions by Kent Beck [10] did not seem to correspond very well to my experiences so far. He suggests deleting your code if it takes more than a few minutes to find a working solution – all to keep the red-green-refactor cycle going rhythmically. For me, who did not have any substantial experience in C++ programming, some solutions, however, had to be allowed some time to be built up, as they included doing Internet research, finding documentation and examples and figuring out how to apply them in my specific scenario. Often I found myself knowing what I wanted the code to do, but the challenge was in finding corresponding functionality – or parts of it - in the standard library.

Up to this point it was noted that the tests had not been driving the design to any notable extent and that my tests perhaps were too big. It was however difficult to make the tests any smaller. An attempt at this was made when testing the signal generation. Then, smaller tests were written first, which only tested part of the functionality and never could handle the full functionality. Therefore a bigger test, covering the full functionality was needed. Implementing this test effectually made the previous smaller tests redundant, resulting in a feeling of unnecessary work.

The ConfigFileReader tests introduced some difficulties because of the use of file streams. The file streams could throw exceptions for a number of reasons and it would probably have been a good idea to mock it out to separate it from the unit under test. However, as I at this point was still just getting started with the TDD practice I did not want to spend too much time figuring out how to mock the file stream. By using many different configuration files, however, many scenarios could still be tested, where the behavior of the file stream could be predicted. Thus only one unit was effectually tested.

During the development of the ConfigFileReader, the requirements of the framework were further specified, and the format of the configuration file had to be changed. This meant that I had to make some changes in the production code and in my tests. It turned out that the only change needed for the tests was in the configuration files used by the tests. The production code reading the files required more change, but as I had the tests to guide me I felt confident changing the code as they would tell me when I was done. During all of this, new ideas for tests were continuously added to my TDD to-do list, to be handled later.

Once again the time between red and green was quite long. It did however seem like this could not be avoided. Most of the time was spent debugging and trying to understand why the tests failed. The case was usually that a seemingly simple piece of code would contain an error that would probably have been caught faster by a more experienced C++ developer. To try to achieve a faster cycle, I decided to try to focus on the TDD principle baby steps in order to try to minimize the room for errors in each test.

4.2.2.6 Creating the Server Class

When trying to write a test to check that all clients with a subscription to a signal got periodical updates, a realization was made. I did not know where the update would come from. It was decided to let the Server class be responsible for this. I did not want it to connect directly to the network, but let it call a ConnectionHandler class to handle that part to be able to keep the core code separate (see Figure 4-3). I started writing a test containing these new classes, but quickly realized that something was wrong. I was using methods in the Server class that did not yet exist to test another method that did not exist. I decided to note all the other methods that were needed and test-drive them first. In other words I first thought of a scenario where some functionality was needed in the Server class, and as I got a better view of what was
needed, I also got a better understanding of which tests were needed for this class. One of these tests was simply checking that a Client could be added to the Server. Here I decided to make a test double of the Server, the class under test, by directly extending it in order to check if the Client was actually added to it. Mocking the class under test was something I did several times during the development of the server application. Even though this might seem like a strange decision it was done because I wanted to test the inner state of a class in a simple way.

When developing the Server class it became clear that multithreading would be needed in order to update the clients periodically. This imposed a problem, since C++ did not provide any platform independent threads in the standard library (before C++11 [19]). This meant that I had to find a free and portable third party library, available for commercial use, for handling the threads. I ended up using TinyThread++[^6]. The library itself was quite straightforward, but the multithreading in general made the development more complex and imposed some limitations on the design.

### 4.2.2.7 Writing Runtime Information to a Log File

One requirement specified that the server application should provide some runtime information to the server user by writing it to a file. This file should contain information about clients connecting and disconnecting, as well as the subscription changes of the clients. Since the states of the Client and Server instances contained this information the thought to use the Observer design pattern to observe the Server and Clients soon occured to me. Tests were written to make sure that changes to a Client's subscriptions would be written to the file and, similarly, tests were written to make sure that changes to the Server's client list were documented. This all resulted in two observer interfaces; the ClientListener and the ServerListener. These were both implemented by a RuntimeInfoFileLogger class used to write the information to a file. The architectural view of the application with these classes included can be seen in Figure 4-4.

4.2.2.8 Connecting to the Network

The first idea to handle the actual connection with the network was to use a class called ConnectionHandler that would be used by the rest of the application to send and receive data over the network, and generally handle everything that concerned communication with clients (see Figure 4-3). As I wanted to start test-driving this ConnectionHandler, though, I realized that it would be very difficult to try to fake the network connection. Therefore, the idea to separate the “protocol” from the actual network connection appeared in order to be able to test at least some of the functionality. The idea was to create two classes instead of a ConnectionHandler; one to handle the network connection (using TCP sockets) and one to act as a translator between the network connection and the rest of the application (see Figure 4-5).
The class acting as a translator (NetworkProtocolHandler) was quite trivial to test drive, while the class handling the actual network connection (NetworkConnector) was not. After some consideration, it was therefore decided not to test-drive the NetworkConnector. The motivation for this decision consists of several points. First, the NetworkConnector would have to be implemented using several threads to let multiple clients communicate with the server application at the same time. This would make the unit testing of the class quite complex. Furthermore when testing a multithreaded application, you will, to my knowledge, need to freeze one or several threads while letting one or several others run, in order to control the execution. This means that the tests take long time to execute, thereby severely slowing down the red-green-refactor rhythm. There are also very many different ways in which the execution can take place when multiple threads are involved, leading to many slow tests. As I already experienced a problem while test-driving the generation of signals in the MessageGenerator, where the tests took several seconds to run, I felt like it would be the best course of action to not test-drive the NetworkConnector. Second, I did not have any concrete ideas on how to mock out the external dependencies of the class and did not find sufficient guidance on the Internet. Third, I had already separated the NetworkProtocolHandler from the NetworkConnector, giving me the opportunity to test-drive some of the functionality regarding the communication with the clients.

Thus, the NetworkProtocolHandler was test-driven, and the NetworkConnector developed without unit tests. The NetworkConnector was initially developed in a separate project using TCP sockets, taking inspiration from a number of blog and forum posts on the Internet, and using open source examples to help write code that suited the needs of this application. Slightly modified open source chat applications were used to test the socket connections and the sending and receiving of data in the NetworkConnector. When considered finished, the NetworkConnector was integrated with the rest of the server application. Since everything else had been tested with unit tests, and the NetworkConnector to some extent had been manually tested, this integration worked without problems.
4.2.2.9 Refactoring to Use the Mediator Pattern

During the development of the NetworkProtocolHandler and NetworkConnector, it was recognized that the Server class was starting to look like a Mediator (design pattern), but that there were still some questions about how all the classes would communicate with each other. I decided to investigate if the Mediator pattern could not be implemented to help to structure the program, since it was starting to get a bit difficult to work with because of unclear and unstructured dependencies. After looking at the code and drawing up a simple sketch of a class diagram I decided to refactor the Server into a mediator (see Figure 4-6). The NetworkProtocolHandler and the NetworkConnector would both register to the Server, which would use the NetworkProtocolHandler as some kind of translator to figure out the meaning of received messages and to format outgoing messages. With this setup, it would be possible to change the protocol used if needed; you would only have to create a new NetworkProtocolHandler. Similarly, and more importantly, it would be possible to create and use a new NetworkConnector, which would be necessary to be able to use the application on another platform.

After making the change to use the Server as a mediator, the Server class had become quite big and difficult to get a good overview over. It was simply not clear what the responsibility of the class was. Therefore I created a new class, Mediator, to handle all the mediating and let the Server class only deal with its own responsibilities. The resulting architecture of the server-side can be seen in section 5.1.1.

![Figure 4-6: Server as Mediator.](image)

4.2.2.10 Making the Application Thread-Safe

Since the Mediator gets multiple simultaneous calls from different threads and has to forward these calls to other classes, there existed a need to make sure that the application could handle this. As stated earlier, testing a multithreaded application is far from trivial, so it was not seen as a viable (especially in terms of time) option to try to test-drive the application to a thread-safe state. Therefore it was decided to add mutex locks whenever a shared resource was used. As this did not change the interface or the functionality, all tests still passed. There was however no guarantee that the application would run completely problem-free when completed. It was decided that the best way to handle this was to simply wait for the Client application to be finished and then test the two together manually.
4.3 Client Application

A decision made before starting the implementation of the client application was that the GUI would not be developed in a test-driven manner. The requirements of the client application were clear on that the underlying framework was in focus and that the example application and GUI would simply be used to demonstrate that the framework worked. In other words it was not essential that it worked perfectly. Furthermore, test-driving of GUIs (UITDD) is not a simple task and the most common approach is to decouple the GUI from the logic and test-drive the layer just beneath the GUI [20]. Hellman et al. [20] bring up a couple of other approaches, but for a simple demonstration application as this one they were considered somewhat excessive. Therefore it was decided to try to decouple the GUI from the logic as much as possible and test-drive everything but the GUI.

4.3.1 Choice of Development Environment for Client-Side Development

For the development of the client application the Eclipse IDE with the ADT (Android Developer Tools) plugin was used. This was acquired from the Android developers’ website by downloading the so called ADT Bundle, including the essential tools and components to develop Android applications. One important part of the package is that the Android Virtual Device Manager was included; making it possible to use the Android Emulator to simulate the execution of an application. The package also included the unit testing framework JUnit, including an Android extension, integrated with the IDE. This was used to perform the test-driven development.

4.3.2 Implementation

In this section the implementation and development of the client-side framework and application will be described. Starting with an initial design, the section will present changes and observations made before reaching the end result.

4.3.2.1 Initial Client Application Design

After evaluating the development of the server application, I believed the lack of initial design had made it more difficult to find design flaws and possible good refactoring. I therefore decided to try to look a bit closer at the requirements of the client application for the upfront design. From the requirements it was clear that the client application would be a lot simpler than the server application. It was principally only supposed to do the following:

- Read a configuration file.
- Send messages to the server requesting to be connected.
- Subscribe to a number of signal feeds on the server application.
- Receive updates from the server and display these in a simple table.

Therefore, the first design consisted only of two components; the main Android activity, which in a sense would also be the view (GUI) and a ServerConnector class representing the server application locally.

4.3.2.2 Creating the ServerConnector class

It was decided to start with the test-driving of the ServerConnector class. The thought was that the rest of the client application would be able to use this class as if it was the real server application, to send and receive messages, etc. Therefore the ServerConnector class had to connect to the server application via a socket
connection. Although I did not manage to do it when developing the server application, I decided to try to mock the socket, to be able to simulate different network scenarios. I found that it was indeed possible without too much trouble but that I had to change the design a bit to make it work. I added a SocketFactory interface, specifying a method for creating a Java socket and let the ServerConnector class take an instance of the SocketFactory in the constructor. Then the ServerConnector class could use this instead of directly creating a socket. By implementing the SocketFactory with a mock and giving it to the ServerConnector, it was then no problem to substitute the real socket for a mock when performing the tests. To be able to run the actual production code I also had to make an implementation of the SocketFactory for real use. This meant that the design now looked as in Figure 4-7.

![Figure 4-7](image)

**Figure 4-7: SocketFactory introduced to the client application.**

### 4.3.2.3 Introducing a Client Class

After some test-driving of the ServerConnector class it was possible to set up a socket connection and send messages. A question arose, however, when test-driving the receiving of messages; namely how the rest of the application would know about the message. In order to separate the GUI from the rest of the application, the need for a new class was identified. This class would be a Client that could then be used by an Android activity. The Client would have a reference to a ServerConnector instance and would register as an observer to the ServerConnector to receive updates from it. To separate the part of the ServerConnector notifying its observers from the rest of the ServerConnector, a ServerObservable class was also introduced in a refactoring, leading to the design shown in Figure 4-8.

The ServerConnector class could now be test-driven to a relatively complete state, but when starting to test-drive the Client class new considerations had to be made to decide how an Android activity would actually communicate with the Client. It was clear that the activity would create a Client instance, but how the activity would be able to be updated was not. After some consideration a ClientView interface was introduced and referenced by the Client. This ClientView would then receive instructions on what to display by the Client and each implementation of the ClientView would be able to decide how to display it (see Figure 4-9). This is important, since the framework is to be used in creating different concepts in the future, with different GUIs.
4.3.2.4 A Configuration File Reader for the Client Application

As the client application was written in Java for Android, there were a couple of more or less simple ways to handle the reading of the client configuration file, by using the Android resources. It was decided to use an external XML file for the client configuration file, which would be stored in an application specific folder on the Android device’s external storage. For standard resources such as strings it is as easy as one method call from an Activity to retrieve the wanted value, but for external XML resources, an XML reader has to be used. The test-driving of such an XML reader was left out, because of a couple of reasons. Firstly, the test-driving of the ConfigFileReader for the server application made the tests quite inflexible, which would not want to be repeated for the client application. Secondly, if trying to make the tests more flexible by letting the tests create the files used, the tests and the production code would in the end be doing basically the same thing.

4.3.2.5 Testing the Client Application Together with the Server Application

As most of the requirements of the client side framework at this point were fulfilled, a simple example Android Activity was created to be able to see if the two applications could actually work together in reality (outside of the world of unit tests). The Activity implemented the ClientView interface, instantiated a Client, provided the Server with a SocketFactory and used the ConfigFileReader to retrieve configuration information from the configuration file. The Activity also contained the logic for controlling the framework, for instance letting the user decide (to some degree) when to connect or disconnect from the server application and what actions to take if a problem occurred. It should be noted that the Activity only initiated...
the Client and Server objects and thereafter used the Client to perform actions on the framework, depending on the information given through the ClientView interface methods, or from the user. The Activity itself was in other words only used to display information and to relay user commands to the framework. Figure 4-10 shows the full client design including the example Activity (some user interface parts specific to this example implementation have been left out).

Figure 4-10: Example use of the client application framework.

4.4 General Observations and Lessons Learned

Presented below are a few general observations that were made during the development of the prototyping framework, but which did not appear in the context of any particular part of the framework.

4.4.1 Use of Version Control – An Important Factor

During the first time of development I did not have access to version control. I experienced how development can really be difficult without it, when a new test and corresponding production code made several other tests fail. It was not obvious what was wrong and after some time of trying to find the source of the problem, I would have been very happy if I could have unmade the changes I had made, or if it would
have been possible to compare the old version with the new one to see what I actually had changed. Following this experience, I installed the version control tool TortoiseSVN to aid me, and tried to commit my changes often in order to easily be able to revert to a previous version and to keep track of the changes.

4.4.2 Forgetting to Refactor

A realization made after a couple of weeks of using TDD was that I had not been doing much refactoring. I had often forgotten to stop after a test passed to see if there was any duplication that could be removed, or if there were other code improvements that could be made through refactoring. Every now and then I had stopped to get an overview of the code and discovered that some design decisions could be changed to improve the code. As I tried to force myself to look for possible refactoring opportunities after each test passed, I noticed that some tests were either a bit too simple to refactor (a typical example is when the principle obvious implementation is used), or too difficult for me to find a better solution for. I did, however, look for design flaws to the best of my ability and tried to remove code duplication by extracting methods and rearranging code.

4.4.3 Test-Driven Development and Focus

Apart from indicating when a piece of functionality was done, as well as giving me a sense of accomplishment and satisfaction, TDD also helped me stay more focused. When working on passing a test, I often came across some piece of code that I instantly saw needed a change – either a refactoring or change of the interface. Normally, without TDD, I would very often jump right into changing such a piece of code, thinking it necessary to change it instantly. This can sometimes work out fine, if the change is small, but sometimes the change cascades across the code and other necessary changes are discovered along the way. In the worst of these cases, so much time would be spent changing other things in the code the original task would be forgotten. With TDD, however, my behavior was different. If I came across a piece of code that I wanted to change, I made a note of it in my TDD to-do list and handled it later. When I discovered that a piece of code needed to change immediately, before the current test would be able to pass, I stopped and went back. In other words, I either removed the test at hand or commented it out, making a note to myself to finish it as soon as possible. Then I either refactored the piece of code that was bothering me, or changed/added some tests to fit the new need that had arisen.
Chapter 5 Results

In this chapter the resulting prototyping framework will be briefly presented, beginning with the server application and then moving on to the client application.

5.1 Server Application

The previous chapter presented the most central parts of the process of developing the prototyping framework, along with observations made and lessons learned during the process. Also design decisions and architectural diagrams showing the design at different stages of the development were presented. Here only the final result will be presented in the form of an architectural diagram and a brief description of the parts and features of the application.

![Diagram of server application architecture](image.png)

Figure 5-1: Final architectural design of the server application.
5.1.1 Final Server Application Design

The final architectural design of the server application is presented in Figure 5-1. As can be seen, the central class is the Mediator, which holds one reference to an instance of each of the classes INetworkProtocolHandler, SignalSource, NetworkConnector and Server. The AbstractMediator is an abstract class which the concrete Mediator extends, and was only introduced to simplify testing by allowing the creation of test doubles of the Mediator class. The classes connected to the Mediator do not communicate directly with one and other in any way, but all communication between the classes goes through the Mediator, which thereby decides what actions are to be performed upon receiving an update from one of the connected classes.

As previously stated, the NetworkConnector is responsible for setting up socket connections to listen for incoming connections and to communicate with connected clients. The subclass, WindowsNetworkConnector, is a platform specific implementation of the NetworkConnector. When the application needs to be ported to another platform the idea is that a new implementation will be created for that platform, to handle network communications.

The INetworkProtocolHandler specifies an interface for translating incoming messages and creating outgoing messages based on data given by the Mediator. Derived from this class is the concrete NetworkProtocolHandler which specifies one way in which the protocol used can look like. If for some reason this protocol would need to be changed, a new implementation of the interface could easily be added and used interchangeably with the existing one.

SignalSource specifies the interface that is implemented by the MessageGenerator, and in the future also a CAN bus interface, making it possible to use the two interchangeably to receive signal data.

Client is the class holding information about a registered client. This includes the IP address, port number and a list of subscription identifiers. ClientListeners can be registered with the Client to receive updates about changes in the Client’s subscriptions.

The Server class holds the state of the server application, with references to local representations of physical clients in the form of Client instances. It decides which clients will be added and which will be rejected a connection with the server application and is also responsible for notifying the registered clients with updates. Furthermore it holds references to ServerListener instances which are updated with information regarding added or removed Clients. The reference to ClientListener might seem out of place, but provides a way to make sure that one or more ClientListeners get registered with all Clients added to the Server.

RuntimeInfoFileLogger implements both the ClientListener and ServerListener interfaces and writes runtime information to a log file on disk, which was a further requirement of the application.

Finally, the ConfigFileReader parses a configuration file for data. It is not connected with the rest of the application, but instead the data obtained from using the class’s methods can be used by the rest of the framework by passing it in method and constructor parameters.

5.2 Client Application

The final client application design can be seen in Figure 4-10, which also includes an example Android Activity to show one simple possible way to use the client framework. The Activity is used to control the framework through a Client instance. The Client in turn uses the ServerConnector to communicate with the server application – in other words the ServerConnector class locally acts as the server application. The ServerConnector informs registered ServerListeners, such as the Client, about received messages, and the Client can decide which methods to call on the ServerConnector or ClientView classes. As the Activity in this case is the ClientView, it implements methods such as showUpdates() and decides how to display the
updates when the method is called from the Client. The ConfigFileReader can be used to retrieve configuration data from the client configuration file.

In Figure 5-2 a screenshot of the running example application is shown. The client application has here retrieved three signal IDs from the configuration file and requested a subscription to these signals from the running server application. The received data is displayed in a simple table layout. Signal number one and two are updated with new values approximately every second, while signal number three was not supported by the server application as indicated by the status field.

![ClientApplication](image)

<table>
<thead>
<tr>
<th>Id</th>
<th>Signal name</th>
<th>Value</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Test Signal no. 1</td>
<td>35</td>
<td>OK</td>
</tr>
<tr>
<td>2</td>
<td>Test Signal no. 2</td>
<td>118</td>
<td>OK</td>
</tr>
<tr>
<td>3</td>
<td>Test Signal no. 3</td>
<td></td>
<td>ID_NOT_FOUND</td>
</tr>
</tbody>
</table>

**Figure 5-2: Screenshot of the running example client application (from emulator).**
Chapter 6 Evaluation and Discussion

This chapter will evaluate and discuss the development of the HMI prototyping framework, the Test-driven development practice - both in general and related to this study’s specific context – and discuss validity concerns and possible future work.

6.1 Development

In this section the development of the prototyping framework will be discussed and evaluated. My experiences will be related to the literature presented in Chapter 2 and discussed in regards to the effects of TDD, the TDD practice itself and the suitability of TDD for the project carried out in this thesis. Also tools and programs used will be briefly discussed with regards to their suitability for TDD in the context of this thesis.

6.1.1 Effects of Test-Driven Development

The first observed effect of TDD presented itself even before the actual development started. As the communication between server and clients was a very critical part of the system, my first instinct was to start the development with this part. Normally, I would probably have started experimenting with different approaches, simultaneously on client and server, until something seemed to work. The rest of the system would then have been built around this. With TDD, however, I remembered to start with something simple, adhering to the one-step-test principle, and got a slow start of the development where I felt that the task at hand was under control.

In accordance with subjects of George and Williams’ study [9], I found that TDD aids a less complicated design. I doubt that the design would be as simple and decoupled if I would have been developing the server application in my usual test-less manner. Besides, for instance the decision to use the Mediator design pattern in the server application was only made because it was getting difficult to write tests for the application. Without testing, it is quite likely that the Mediator pattern would never have been used, but that all classes would instead have communicated crisscross with each other. This would then have meant that changing something in the application, like introducing a new class would have been far more complicated.

The test cases written for the server application were at first quite big and unspecific, similar to what is described of the traditional tests (test-last) in the study conducted by Canfora et al. [14]. However, my tests tended to get smaller and more specific over time, as I got more used to TDD, getting closer to what in the same study was described of the tests written in a TDD manner. Perhaps could this then be an effect of TDD – that the tests written become small and specific – which would definitely be a desirable one. The fact that my tests initially were big and unspecific was perhaps a consequence of my relatively limited experience of TDD and of unit testing in general. It could also be an effect of the fact that the requirements and general upfront design initially were quite unspecific, making it difficult to translate the requirements into actual unit tests. According to Canfora et al. small, specific tests mean that fewer assertions are missed since the
problem is divided into smaller parts instead of only being seen as one big problem, so it is important to try and keep the test size to a minimum.

Siniaalto and Abrahamsson’s study [15] indicates that some professional expertise may be required for TDD and that the effects of TDD do not have to be positive if this expertise does not exist. I believe this to be a correct conclusion. In my opinion, my expertise was just about enough for me to get any benefits from using TDD. A developer less experienced would probably not have been able to make appropriate refactoring when needed and would perhaps also perceive difficulties with writing good tests and test doubles. A more experienced developer, on the other hand, would probably encounter fewer difficulties and reap more benefits of TDD. I also had some difficulties with writing good tests and test doubles in the beginning of the project, and I do believe that without my previous experiences with software testing, the difficulty level of using TDD would have been significantly higher. Therefore, if a developer trying out TDD for the first time does not have very much experience, I believe it to be a good approach to have an accustomed TDD practitioner as a guide until a basic level of understanding and confidence in the practice is acquired.

Maximilien and Williams [16] concluded that TDD produced code with fewer defects and attributed this to the fact that testing actually occurred. I believe this to be one of the most important parts of the TDD practice – that testing occurs. Unit testing is something that I have never really used as part of developing before, but only as a dedicated practice on existing code in a university lab course. My experience from developing this prototyping framework using TDD was, however, that the unit testing is very useful. First of all, it is difficult to test code that is not decoupled, so when writing tests first, one tries to make it easy for oneself by decoupling the code. Since decoupled code usually means that it is easy to make changes and add or exchange parts of the program, this is a good thing. Second, unit testing makes sure that the code does what it is supposed to do, or more specifically, it helps to find defects in the code. Of course the extent of this depends on how good and exhaustive the unit tests are, but I believe it is safe to say that some testing is better than no testing. Third, passing unit tests gives confidence, which I experienced many times during development. It gave me both confidence and a feeling of accomplishment and satisfaction to see my tests pass one by one and I really did feel more comfortable performing more substantial refactoring with the tests telling me if something went wrong.

Unfortunately, not much can be said about the productivity when using TDD as compared to a test-last approach, since no comparison is at hand. Nor can I provide any personally perceived views on the productivity, since my previous development experiences have been performed completely without unit tests. I will refrain from making any purely speculative comments regarding the productivity as comparative studies have already measured the productivity differences between using TDD and a test-last approach (although with varying results).

Of course the main motivation of TDD is not only productivity, but quality. As previously stated, some testing should be better than no testing. In the case of TDD, no untested code should be added, so it can be expected that the code will have better (external) quality. This is also supported by George and Williams [9], and Siniaalto and Abrahamsson [15], who observed better quality code when using TDD than when using a test-last development method. Kollanus [18] also gave support (although weak) for improved external quality when using TDD. Relating this to my own experience, I am quite certain that the quality of the code is better than if it would have been written without any unit testing, although it should be noted that this is my personal perception. It is furthermore difficult to speculate about whether writing the tests first actually mattered in the context of external quality, or if it was the use of unit testing in general. What I can say, is that I firmly believe that testing first helped to write “testable” code (code that is easy to test and to create test doubles for), which I believe to be an important base for efficient unit testing.

Somewhat connected to the productivity is the focus and structure I perceived when using TDD. In some sense my discipline increased because of the simple structure of a test list combined with the TDD cycle, which allowed me to get a good overview of the development progress. In general this gave me a more relaxed experience with a sense of having things under better control than with my usual test-less approach.
Many good things have been said about TDD, but one of the most important things to know about TDD is that it will not magically generate good design. This is something that is also in some way expressed by subjects in the study by Aniche, Ferreira and Gerosa [13], where they stated that TDD itself is not a solution for all design problems. Instead, I am convinced that developer experience plays a big part in how well designed the program will be. Perhaps TDD has a greater effect on more experienced programmers, as it seems to make it easier to see mistakes and flaws in the design at an early stage - if you are able to recognize these. But in order to see for example when a design pattern could be helpful to use, some knowledge of these design patterns need to exist, and preferably also experience of using them in practice. The same goes for general software design principles. The developer needs to know about these in order to follow them, because TDD will not in some unexplainable way lead the developer to unconsciously adhere to them. In short you have to know what good design is in order to create it. Therefore I would not recommend a novice programmer to try TDD, but would tell a relatively experienced programmer to try it out. As for me, personally, I believe I could have benefited a lot from a bit more prior programming and designing experience, but could still perceive some effects of TDD.

An unexpected effect of TDD that I experienced during development of the server-side framework was that it stopped me from refactoring. As refactoring is an integral part of the TDD cycle and really what is used to keep the design good, this is an interesting situation. In my case I saw that it was possible to use a design pattern and my first instinct was to implement it, as design patterns are supposed to be good design. However, when looking closer at the particular situation at hand it did not really fit in. In other words, after analyzing the design and the situation I came to the conclusion that refactoring would not improve the design. Therefore it is clear to me that the refactoring step of the TDD cycle should not be interpreted as “perform some refactoring, no matter what”, but instead as an encouragement to analyze the code and to actively look for design flaws. Then, the refactoring can take place if it can actually improve the design. Perhaps it would have been better to describe the TDD cycle as red-green-evaluate instead of red-green-refactor. This would, in my opinion, have clarified what is actually going on.

6.1.2 Test-Driven Development as a Practice

The TDD process was described in section 2.4 along with a number of TDD principles. After using TDD myself, I will below discuss the process of TDD, especially on the basis of these TDD principles.

One of the most central parts of TDD is that the tests are written first. For me, the most important difference between using TDD and not using it was that I wrote any tests at all. In my opinion, writing the tests first serves mainly two purposes. Firstly, it ensures that the tests get written at all, and secondly it forces the developer to think about the design. If not writing tests first, it is easy to forget to write them later. Even if tests are written later, some cases might be forgotten, leading to worse coverage and possibly fewer caught defects. When writing a test for some part of the program that does not yet exist, one is forced to think about how it should work and what would be the easiest way to test or use it. Furthermore, one has to think about how the creating of test doubles could be simplified, which can also impact the design.

The principle Fake it ’til you make it was one that I had some problems with. It felt strange to write code that I knew would have to be replaced when the next test was written. I tried to adhere to this principle but often found myself implementing functionality immediately, instead leaning more to the principle obvious implementation. Since both these principles coexist, I suppose one always has to decide what is appropriate for the specific situation.

The related Triangulation principle states that two or more tests are needed when faking an implementation. Beck [10] discusses this briefly, and states that he only uses triangulation when he is really unsure about how to perform a refactoring. When he already knows how to remove duplicated code and generalize the implementation, he means that there is no real need for the triangulation. After using TDD myself, I do agree with this. Many times the triangulation can feel quite silly, if it is already obvious that a
generalization is needed. I found that I used the *obvious implementation* principle more often than I used *Triangulation*. Perhaps this was because I was not used to TDD and was staying closer to my usual programming process.

The question of the so called baby steps is also discussed in the article by Aniche et al. [13], where the subjects had different opinions on how strictly the principle should be followed. Therefore I think it is perfectly fine for each TDD developer to make own judgments about how closely to follow this principle. Personally I agree with the subjects of the study that said that always doing baby steps would not be productive.

Isolating tests for better readability and pinpointing of failures is a principle which benefits can easily be understood. However, when writing the tests, it is quite easy to put an extra assertion in a test case, thereby actually testing several things. I tried to keep my test cases as isolated as possible, but some test cases did span more functionality than perhaps would have been ideal. Often this happened when two things that were very related had to be tested and where the setup of the test case was not trivial. In these cases I found that the readability would have been worse if the tests had been divided. Furthermore, messages were added to all asserts, meaning that if a test case failed, the tester could easily understand what went wrong, even if the name of the test case alone would not suffice.

The *Assert first* principle states that the developer should first make assertions, and then write the test. This principle was used by me when I did not know how to start writing a test. If I knew what functionality I wanted to test, but not really which test doubles would be needed, etc. it was good to go directly to the end and make the assertions. It then became clear what was needed to setup the test case. This method can in other words be quite useful when one gets stuck and does not know where to begin, but in my opinion it does not have to be used all the time. Mostly, I already knew what was needed for a test case and simply wrote it from top to bottom. It was, however, good to know that there was another method that could help if I needed it.

One useful tool when performing TDD was the Test list, or the TDD to-do list. Here I entered all ideas I got for new test cases, refactoring and everything else that needed to be dealt with regarding the development. If working on a test, I did not stray from it even if I found something that really needed to be done. I simply added this to the list and continued with what I was doing. This might not sound like much, but it proved to be a very good way for me to stay focused and to work in a more structured manner than usual.

### 6.1.3 Mistakes When Performing Test-Driven Development

In section 2.5 a number of common mistakes made in TDD were presented. Although I knew about these mistakes beforehand, I still committed a couple of the most common ones myself. Most important is probably that I often forgot to refactor the code. I often experienced a sense of accomplishment when a test passed, thinking that I had finished a part of some functionality. Of course, the real TDD loop was not finished in these cases, but happy about having completed one test, I soon started with a new one, completely forgetting the refactoring step. This improved during the course of the project which indicates that it might take some time before getting into the TDD way. As also found in the study by Aniche et al. [13], TDD is not easy and it does take time to get used to it.

Another mistake made was to start developing the server application without any real upfront design. Literature suggests that TDD is supposed to start with very little upfront design or none at all, but I disagree with this – at least in the case of less experienced developers. In my opinion it is good to have a rough idea of how the program will look before starting. It does not have to be very elaborate, but just some initial rough plan of how to make the program work. Perhaps some will claim that this would hinder the tests from driving the design, but I believe that the tests will be able to highlight flaws in the design, which could lead the developer to change it. However, if no upfront design exists, it is difficult to picture what one is actually
working on. As the focus goes from one test to another, only covering some small piece of functionality at a time, it is difficult to at the same time try to imagine how everything works together. This also means that it can be harder to discover flaws in the design and necessary/beneficial refactoring. I attempted to start with more upfront design when developing the client-side framework, although it was still very simple and rough (perhaps mostly because of the simplicity of the client application overall). To further assist in keeping the big picture in mind at all times, I regularly sketched some class diagrams representing the code, to easier detect bad design. This was useful as a help, especially during the refactoring steps and also when writing new tests.

A situation that appeared several times when starting to work on a new requirement was that I did not know where to start. One of these times I realized that I had tried to start a bit too far ahead. That is, it became clear that I needed functionality that did not yet exist before the test would make sense. After analyzing the situation, and my entire approach, I saw that by starting too far ahead I had effectively taken a closer look at the requirement and how the new functionality would be used. I had created a scenario of its use and gotten a much clearer understanding of the requirement. The use of so called use cases and user stories could therefore probably be helpful to give the developer a better understanding of the requirements. Somewhat related is an approach I once saw suggested on an Internet forum, used when faced with a difficult or unfamiliar problem. The developer first wrote production code for the functionality needed, probably using pieces of code found online as help. When this production code worked, it was deleted and the whole functionality test-driven from memory. This way the tests are still given a chance to drive the design, while at the same time giving the developer a chance to get acquainted with the functionality. The use of this approach could perhaps have been useful during my development of the server-side NetworkConnector and is one that could be interesting to investigate further.

The time to complete one TDD cycle (red-green-refactor) many times took longer time than what is described as desirable by for instance Beck [10]. This can be seen as a mistake on my part for not making sure that the steps taken were small enough, or because I seldom removed code that took too long to pass a test. About halfway through the development, however, I started to get comfortable with, and used to, TDD and the frequency of this mistake was reduced.

Not using version control from the start of the development was a further mistake made. By committing the changes after adding some new functionality I had a safe point to return to if something would go wrong with the next added functionality. It also made it practically possible to follow the advice of Kent Beck [10] of throwing away the code and start over when feeling lost or when a solution got too complicated.

It was, furthermore, probably a mistake not to use a so called mocking framework during the development. Unfortunately it was not available for me which meant that all test doubles were created manually. This was both a tedious and time consuming task, which could probably have been greatly simplified had a mocking framework been available. The quantity and importance of test doubles was to some extent underestimated when starting the development and if done again greater effort to acquire a mocking framework would have been made.

6.1.4 Test-Driven Development of an HMI Prototyping Framework

One question asked in the introduction of this report was regarding the suitability of TDD in the specific context of this thesis. Based on my experiences in developing this framework I will make some comments about it below.

First, this system does include a simple graphical user interface for the client application. As described in [20], test-driving of GUIs is not a simple and straight-forward task. Therefore this was left out and created separately from the TDD process. It is my opinion that this approach is a good one. The possible benefits of test-driving the GUI would (in this case) probably not have been greater than the effort put in to make it work. I do however think that the user interface test-driven development (UITDD) is an interesting topic that
could potentially be beneficial, but it was decided that the area was too substantial to be covered in the scope of this thesis. It would perhaps be useful, however, to look into another UITDD approach when developing the real concept applications, which will surely be more complex than the simple example application developed as part of this thesis. One possible approach, as described in [20], would be to use a combination of a low-fidelity prototyping tool and a capture-replay testing tool. Then a prototype of the GUI could be created and evaluated from a usability viewpoint before actions would be performed and recorded on the prototype to later be automatically reproduced on the real GUI as tests.

Second, the framework created also includes network connections. As described in section 4.2 this imposed some challenges in how to use the TDD process because of the difficulty in mocking the network and because of multithreading. It was decided to act in a similar manner as with the GUI, and test-drive the layer just above the network connection, but leave the actual connection “layer” be written without TDD. Another way to see it is that the class connecting to the network was treated as a third party component. Since these should normally not be tested (as they are assumed to already have been tested), all that is tested is that calls are made to the component correctly. Perhaps the challenge in mocking out the network on the server side would not appear so difficult to a more experienced TDD practitioner, and perhaps it would not be as challenging with the help of a mocking framework. If more time had been available for this thesis project it would have been interesting to investigate the possibilities of test-driving the connection of the server application too. The used approach seemed to work pretty well, however, as the integration of the test-driven and not test-driven code worked problem free. Regarding the threading of the application, it would have been difficult to test if this was working correctly without setting up quite complicated test scenarios. As the test-driving of threaded applications was not the focus of the thesis, this was also added separate from the TDD process. All the real functionality was however still test-driven, and only the mutex locks preventing several threads from entering critical sections at once were added separately.

Third, the development of the MessageGenerator did not work well with TDD because multiple threads were used, where the test had to wait for the code under test to execute in a separate thread and make assertions during this execution. If done again, the threaded parts of the MessageGenerator would probably somehow have been tested separately from the rest of the application, so that the time consuming tests did not run every time all tests were run, in order to speed up the red-green-refactor rhythm.

Apart from what is mentioned above, TDD was well suited for developing the application, and no significant problems were met.

### 6.1.5 Tools and Programs Used

Below some of the different tools and IDEs used during the course of the project will be discussed. Only my personal experience and subjective views will be discussed and it is possible that some of the problems encountered do not normally occur. However, I believe it to be useful for other developers to know what may be experienced.

#### 6.1.5.1 Microsoft Visual Studio Professional 2010

The IDE used for the server application, Microsoft Visual Studio Professional 2010, worked well for test-driven development, but this particular version did unfortunately not include the IntelliSense intelligent code completion and refactoring tools for C++/CLI projects. This imposed an inconvenience which was towards the end of development partly removed by - in parallel with Visual Studio - using the Eclipse IDE for C/C++ Developers which provided such tools. Visual Studio also included a unit testing framework that worked well for TDD purposes, which the Eclipse IDE did not.

If done again, I would have made more inquiries about the possibility to acquire a newer version of Visual Studio in which IntelliSense could be used, or further investigated the possibility to use another IDE such as Eclipse with a third-party unit testing framework.
6.1.5.2 Eclipse IDE with Android Development Kit Plugin

This IDE worked very well for developing the client side framework and application. One downside was that the Android device emulator was a bit slow, meaning that the running of tests on the emulator took longer than would have been desired. For this relatively small project it was, however, in my opinion fast enough for performing TDD and the use of the emulator was a very practical way to perform tests without an actual Android device. It is also possible to run pure Java tests with normal JUnit (as opposed to the Android specific one which executes tests on the emulator), which is much faster. This could be useful in a larger project, where pure Java code is separated from Android specific code. Thus the Java code could be tested separately, making the total test run faster.

6.2 Validity

The internal validity of this study is heavily threatened by the fact that the researcher and the observed developer was the same person. This means that the effects of TDD could only be investigated from a subjective point of view, where the developer’s experience with TDD was compared to previous programming experiences. It is quite possible that some of the perceived effects of TDD are not actually stemming from the use of TDD, but from other factors, such as the setting, the choice of IDE, the programming languages used, and the specifics of the developed framework.

The reliability of the study is generally relatively low, since the framework could have been developed in different ways, leading to some different experiences than were present in this study. Furthermore the programming and design experience of the developer may play a big role in how obstacles are perceived and handled, and in how the principles of TDD are perceived and adhered to. In the context of beginners of the practice with experience corresponding to a fifth year software engineering student, the study is more reliable, as the observations and reflections made can be assumed to depend on the experience (both of TDD and general software engineering experience) of the researcher/developer.

External validity of the study can be considered moderate. In the context of beginners to the practice of TDD with moderate software development skills, the observations and reflections made regarding the use of TDD could be expected to be similar for other similar settings. Furthermore this study can provide a good insight into the use of TDD for beginners, as it brings up possible problems and gives a view of what can be expected when attempting TDD for the first time in a similar setting. Regarding the effects of TDD, these were not measured in this study but simply discussed from a subjective point of view. Therefore any speculations or conclusions drawn about these cannot be considered to have any substantial external validity.

6.3 Future Work

As problems arose in this study when introducing multithreading and socket connections between server and clients and these problems were not handled but worked around, it would be interesting to see investigations into how these problems could be solved. Questions like “How can multithreaded applications be test-driven?” and “How can a server-client application be test-driven?” could be provided with interesting answers. Furthermore, it would be interesting to see more studies and experience reports on the topic of user interface test-driven development (UITDD) and its possible effects, as this could potentially prove to be a useful technique.

More experience reports from first-time TDD practitioners and studies reporting subjective perceptions of TDD beginners would also be interesting, as it is not only important what measurable effects a development practice has on programming (such as productivity and test coverage), but also what the programmers think about it. A highly efficient development practice that nobody likes to use might in the long run not be better than a moderately efficient practice which everybody loves.
Chapter 7 Conclusion

In conclusion Test-driven development is a practice that seems to hold several benefits, but also some drawbacks. While the practice seems to facilitate lower coupling and more structured design, thereby simplifying the adding of new functionality and generally increasing the flexibility of the code, it also hinders the changing of interfaces since changes also have to be made to the test code. The most important effects of the day-to-day coding, as perceived by me, were that TDD helped me to stay focused on the task at hand, that testing actually occurred, that the code was written in a “testable” manner and that the tests provided confidence and a sense of accomplishment.

It was further concluded that experience may play a part in the success of TDD. A more experienced developer could be expected to reap more benefits from the practice than a beginner, who does not recognize where design improvements could be made. Furthermore, some unit testing experience would be recommended before attempting to use TDD for the first time. As a beginner to the TDD practice, the goal should be to learn TDD and not unit testing. If no prior experience to either exists, the start of TDD will likely be quite difficult and it is probable that TDD will therefore not be given a fair chance. However, even with prior unit testing experience, TDD can be difficult to learn. In this study the developer met several obstacles and had a hard time adjusting to the TDD way of thinking, which is something that has also been seen in other studies and could be expected by other TDD beginners.

The suitability of TDD for the particular framework that was developed in the scope of this study can be considered moderate. While a couple of parts of the framework were left out of the TDD process, most of it could be developed using the practice. It is also possible that an experienced TDD practitioner would deem the suitability to be higher, as he or she might have ideas on how to include the excluded parts in the TDD process.

7.1 Recommendations

For other developers interested in TDD, I recommend to give it a try. If possible, I would also recommend any beginners to keep an experienced TDD practitioner close by – at least at the very start – to help and answer conceptual questions that might not arise until theory is set in practice.

Furthermore I recommend using version control as it, in my opinion, is an essential tool to enable the developer to refactor aggressively and to give a safe point to return to where all tests pass. Also recommended is to use IDEs with automatic code completion and refactoring tools which can greatly assist the developer during development, since TDD includes a significant amount of refactoring. These tools can also greatly simplify the step from a test case that does not compile to one that does by automatically generating classes and method stubs.

Other than that I recommend anyone interested in TDD to read Kent Beck’s book “Test-Driven Development by Example” which, in my opinion, provides a good insight in how the practice works.
References


[26] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA, USA: Addison-Wesley, 1995.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att ingaextra-
ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Mihael Ursic