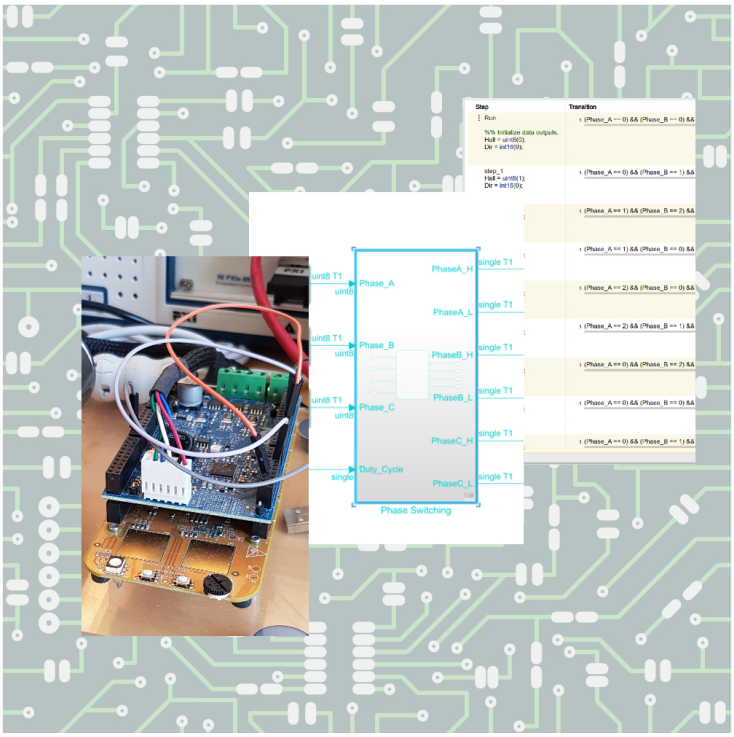
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Study on the Efficiency of Test-Driven Development for Automotive Systems using Model-Based Design

Master's thesis in Embedded Electronic System Design

VINCENT ADLER

Department of Embedded Electronic System Design
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Study on the Efficiency of Test-Driven Development for Automotive Systems using Model-Based Design

VINCENT ADLER

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Embedded Electronic System Design
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Study on the Efficiency of Test-Driven Development for Automotive Systems using Model-Based Design

Vincent Adler

Supervisor: Lena Peterson, Department of Computer Science and Engineering
Advisors: Viktor Nilsson & Tommy Jansson, Infotiv
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Embedded Electronic System Design
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: (From left to right) The board and power stage used during the study, a Simulink module block and the Simulink test harness environment.

Typeset in LaTeX
Gothenburg, Sweden 2020

Study on the Efficiency of Test-Driven Development for Automotive Systems using Model-Based Design

VINCENT ADLER
Department of Embedded Electronic System Design
Chalmers University of Technology and University of Gothenburg

Abstract

Since the automotive industry uses model-based design and strives towards more effective and qualitative development, this thesis explores the effectiveness of test-driven development with Simulink for development of an automotive system. To this end, a system for an automotive motor installation has been designed using a model-based design tool, Simulink, by alternating between two different development approaches; test-driven development (TDD) and a test-after (TA) approach. The quality of the code generated by these different approaches was evaluated using four metrics to cover different steps of model-based development (identification, semantics, relationships). The metrics used were: 1) Weighted blocks per module, 2) Coupling between modules, 3) Response for a module and 4) Lack of cohesion in blocks. To address the last step of model-based development (implementation), coverage of tests and time spent developing were used as metrics.

Trends could be observed showing that TDD improved the quality of larger modules, while TA improved the quality of smaller modules. The time spent on developing the modules, together with the above mentioned quality metrics, showed that the development was more efficient when using TDD. Simulink as a design tool does not restrict any of these approaches, however, the test environment of Simulink significantly hinders the use of TDD.

Keywords: Test-driven development, model-based design, automotive system.

Acknowledgements

I would like to thank my supervisors Viktor Nilsson and Tommy Jansson at Infotiv Technology Development, my supervisor Lena Peterson and my examiner Per Larsson-Edefors at Chalmers University of Technology.

Vincent Adler, Gothenburg, July 2020

Contents

Abbreviations

CAN	Controller Area Network
CBO	Coupling Between Objects
CBM	Coupling Between Modules
DIT	Depth of Inheritance Tree
HIL	Hardware-In-the-Loop
I/O	Input/Output
LCOB	Lack of COhesion in Blocks
LCOM	Lack of COhesion in Methods
MBD	Model-Based Design
NOC	Number Of Children
PXI	Peripheral component interconnect eXtensions for Instrumentation
RFC	Response For a Class
RFM	Response For a Module
TA	Test-After approach
TDD	Test-Driven Development
WBM	Weighted Blocks per Module
WMC	Weighted Methods per Class

1

Introduction

Due to increased complexity in today's software for embedded systems, in particular for automotive products, there is a need to reduce lead time and improve quality [1]. One way of reducing lead time is to implement test-driven development (TDD) [2]. This practise differs from the conventional method (the conventional method being implementation first, then tests) by developing the tests *before* the implementation. By designing the tests before the system is designed, the implementation is driven to fulfill already existing tests. This practise is the foundation of test-driven development.

In the industry of today, agile development is used extensively to improve efficiency. The agile workflow uses iterative development cycles to improve and finalize a product using increments of production. Testing must be in place for each increment of the product during the agile workflow. When TDD is used during development, all of the unit-tests are already in place to ensure functionality of each module. Also, being able to verify functionality instantly during development, by running continuous tests, serves as a strong tool to eliminate errors earlier in the process [3]. This functionality of TDD also lends itself well to continuous integration, using a common repository. One thing to be considered is that TDD verifies lower-level functionality. System level tests might still be needed.

Many companies in the automotive industry use model-based design, where certain software is used to graphically connect blocks to create a system, for developing embedded system software [4]. It provides higher level programming and efficiency when developing large systems. It is of interest to the automotive industry to further improve efficiency and TDD in combination with model-based design might achieve that. There are research studies where the efficiency of TDD has been studied [5][6][7]. However, these studies have not regarded model-based design, but rather code-based design. Neither have they been conducted with an automotive implementation in mind, but rather for software services and applications. A case study on a small-scale, automotive hardware system could showcase the strengths and weaknesses of this approach and give an example of how this workflow could be realized in a larger scale.

1.1 Scope and limitations

The study compares two different development methodologies. The comparison is based on their efficiency and on the quality of produced implementations and tests. The efficiency is defined as time consumption of development and testing while quality is defined by a set of metrics, explained in chapter 2.
The thesis is centered around different development methods, specifically for designing automotive systems. Since only one person has performed the study and the implementation in a limited, fixed time frame, the case used has been limited to a small motor system with a fixed set of requirements to fulfill. These requirements have been set by the company where this study has been performed.

Simulink is a tool used extensively in the automotive industry, and is provided by Chalmers University of Technology. Therefore, this study is limited to use Simulink and its libraries exclusively.

1.2 Aim and problem definition

The aim for this thesis is to develop an automotive system using model-based design with both test-driven development and a test-after approach and to compare these two methods of development and testing.

The first step of the thesis is to develop an automotive system with these approaches in a model-based manner. It is unclear if the implementation developed by the approaches would differ in quality. Thus, the first research question is:

> **Q1:** How does test-driven development compare to a more traditional approach of testing after implementing with respect to quality of code when using model-based design?

Besides the comparison between the two approaches regarding quality of code, it is of interest to evaluate if any of the two approaches are more efficient. If any approach is less time consuming, this could save costs for the industry. Thus, the second research question is:

> **Q2:** How does test-driven development compare to a more traditional approach of testing after implementing with respect to efficiency when using model-based design?

Since the design tool used is Simulink, it is of interest to determine any differences introduced by the tool when using either approach when developing. With respect to the tool used for model-based design, an additional question to be further researched is:

Q3: What strengths and weaknesses with Simulink can be observed when using it with either of the two development approaches?

2
Theory

This chapter explains the main concepts used in this study. The first section explains the workflow called test-driven development. The second section explains how the test-after approach is defined. The third section explains how the model-based design is utilized using Simulink. The fourth section presents Booch's steps for object-oriented design and explains how they are modified for this study. The last section presents how quality of code can be measured using a certain set of metrics.

2.1 Test-driven development

The process of test-driven development (TDD) is applied on the basis that one follows the requirements for a system to write tests for this system, before any type of implementation is in place [8]. These tests are doomed to fail, since no implementation has been developed yet. When a test fails, code is developed to make this test pass. This way, an implementation grows from the fulfillment of tests of the requirements. In theory, all code is developed to make tests pass, and therefore full coverage is achieved for all tests. When the design is fully developed, all code has dedicated tests and is ready for verification.

To start a development process with the test-driven approach, a prerequisite for the workflow is that there exist requirements for the system. The developer uses the requirements to define test cases. In this stage, it falls on the developer to define inputs, outputs and method structure for the intended implementation. The test case will need this structure to communicate with the system, so it has to be defined in this stage. While the test case is designed, as soon as the test does not compile, or a run test fails, the developer switches attention to the implementation and rectifies it to let the test compile or pass, respectively. Already, parts of the implementation are being designed from creating compiling test cases. And this is the core philosophy of this workflow; when a test fails (or fails to compile due to the implementation not corresponding with the design of the test), the implementation is rectified in order to make the test pass. When a test passes, the developer enters the refactor stage. This is where code clean-up and commentary is done to make the system more readable. The test is run again after this stage to verify that the system still works as intended. When the test finally passes and the implementation is refactored, another test can be designed to continue the process.

Following this workflow, an implementation will grow from well-defined tests which

in turn are designed to fulfill the requirements. This workflow can be seen in Fig. 2.1. When all tests are designed and pass, the implementation should fulfill all requirements.

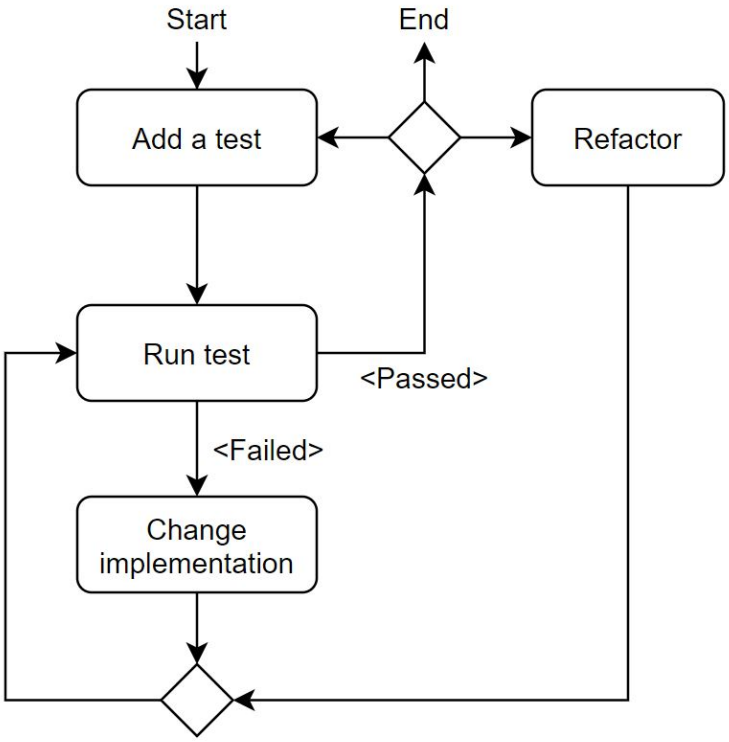


Figure 2.1: Test-driven development process. After adding and running a test, the outcome of the test decides if the implementation needs to be changed or if the code is ready for refactoring (clean-up). After refactoring and the test passing, the process can restart.

2.2 Test-after approach

Many developers today use a test-after (TA) workflow [6]. In contrast to TDD, when following this workflow, the developer starts by developing an implementation, which is supposed to fulfill the requirements for the system. During this stage, tests are carried out sporadically to verify functionality. When the design is considered done, a test is designed for verification. If this test fails, the implementation is altered to make it pass. When the test passes, either a new implementation is added to fulfill more requirements, or the design is refactored. In this study, the refactoring stage is included when using this approach, since it will yield a more fair comparison between the two approaches with respect to time spent developing. When this workflow is finished, there should be an implemented design which fulfills the tests. These tests in turn should verify the requirements for the system. A flow chart depicting this workflow can be seen in Fig. 2.2.

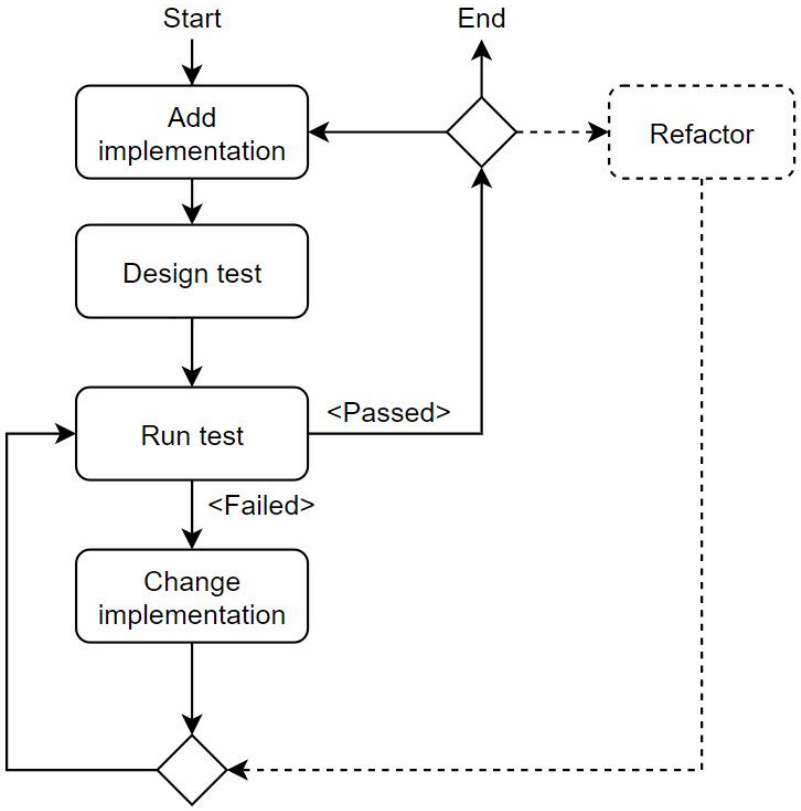


Figure 2.2: General depiction of the traditional approach to software system development. Tests are designed after the implementation is added, and the implementation is changed until it passes the test. Refactoring is depicted in dashed lines since it is not always a part of the process.

2.3 Model-based design in Simulink

Model-based design is primarily used to model systems. A common example of this is to model a plant and develop a controller for this plant [9]. These two parts can then be simulated and deployed on the physical plant to create a working system. Model-based design tools let the objects be visualized as blocks in a model. This representation makes the system easier to compare to a real car and gives the developer a high-level view when designing the system.

The model-based design tool used in this case has been Simulink [10]. Simulink is one of the most widely used tools in the automotive industry [11] and has therefore been of interest when studying the efficiency of test-driven development with model-based design.

Simulink differs significantly from the tools used for code-based design; one of the key differences being that code-based design uses lines of code to form a system, where Simulink uses a graphical interface where blocks are placed and routed using

signals. This introduces an important difference for this study: the hierarchy. In Simulink, the hierarchy is represented with blocks within blocks, instead of method calls which are used in code-based design.

Another difference is the test definition process, where test cases can be created using test harnesses. When doing code-based test design, a separate test file is created to test the design. In Simulink, a test harness is created which does the same thing, but is represented by blocks that run a test sequence and test assertions separately. These blocks can be seen in Figure 2.3, in this example used to test a voltage control unit.

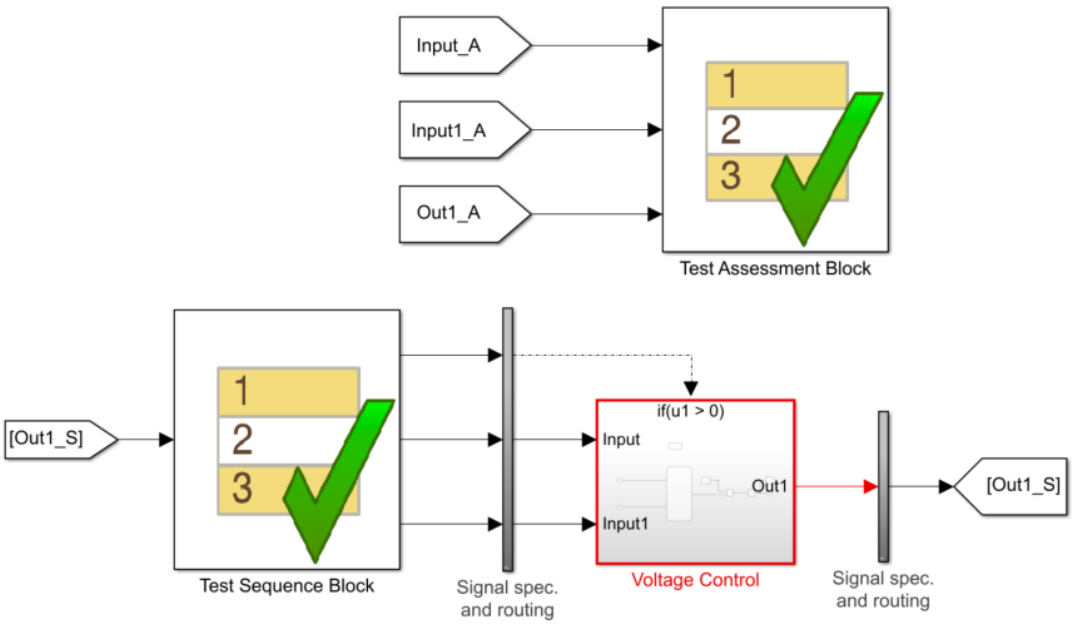


Figure 2.3: An example of a Simulink test harness. Separate blocks are used for defining a test sequence and test assessments. The red outlined box is the system under test.

This is one example of how the functionalities of Simulink are similar in function to code-based design, but can differ in application. One must consider the differences between design methods when evaluating how applicable a test-driven development approach can be with model-based design through Simulink.

In Simulink, methods are replaced by blocks with different functions. These blocks are connected with signals instead of variables, much like electronic components are connected with wires. These blocks form modules, which in code-based design would equal to programs. The terms *modules*, *blocks* and *signals* will be used extensively in this report. A visual example of the Simulink modules can be seen in Fig. 2.4 and a similar example of blocks connected with signals can be seen in Fig. 2.5.

Working with Simulink is made easier with toolboxes. These sets of functions and

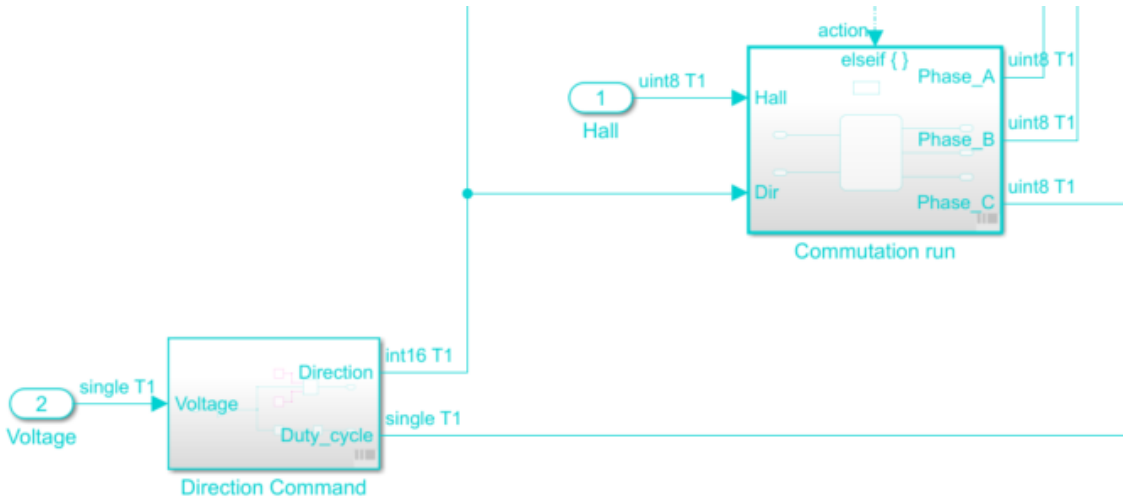


Figure 2.4: Example of modules in the Simulink development environment. The two modules *Direction Command* and *Commutation run* are different modules, where output from *Direction Command* is used by *Commutation run*. Each module also receives input from exclusive input ports, *Voltage* and *Hall* respectively.

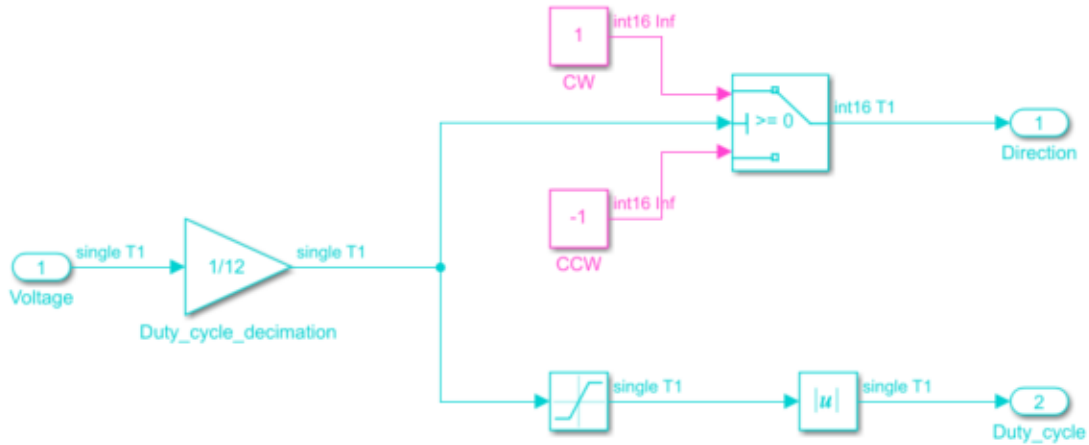


Figure 2.5: Example of blocks and signals in the Simulink development environment. This is the inner workings of the module *Direction Command* from Fig. 2.4. Here, each block corresponds to a method or function. The blocks are connected by signals, represented by arrows.

features can prove essential for an effective development process. Simulink has many toolboxes to help with testing and verifying, such as Simulink Test [12] (used for testing designs), Simulink Coverage [13] (used for monitoring test coverage) and Simulink Requirements [14] (used for setting requirements for the design).

When using TDD, the methodology includes creating tests before creating any implementation-code. This approach is hampered by the Simulink Test toolbox since it can only generate tests from already existing implementation blocks. But there is a workaround. Creating an empty block with input-output pins enables the generation of a test case, and from there a test can be designed before the rest of the implementation. This calls for a modification of the TDD workflow when working with Simulink, which can be seen in Fig. 2.6. The initial creation of a test has been divided into two parts. The first part being the creation of an implementation block with I/O-ports. These I/O ports do not have to connect to anything inside the block at the time being, but they need to exist for the test harness to be able to access the block from a higher block hierarchy. The second part of this stage is to generate the test harness for the newly created block, and define the test cases and assertions within this harness.

2.4 Booch's steps for object-oriented design

The evaluation of this study has striven to reflect the quality of the design. To achieve this goal, metrics have been defined and used for this purpose. To define these metrics, Booch's steps to design object-oriented software has been used as a baseline [15]. These steps sum up the process of designing object-oriented, and can be modified for use with model-based design. The original steps include:

- Identification of Classes (and Objects)
- Identification of the Semantics of Classes (and Objects)
- Identification of Relationships Between Classes (and Objects)
- Implementation of Classes (and Objects)

These steps can be modified for usage with model-based design by replacing classes and objects with their model-based counterparts: blocks and functions. After modifying them to fit for a model-based context, the new steps have been defined as the following:

- **Identification of Blocks (and Functions):** This step includes identifying what functions and blocks must be designed to fulfill the desired feature or requirement.
- **Identification of the Semantics of Blocks (and Functions):** This step includes defining the meaning and function of each block
- **Identification of Relationships Between Blocks:** This step includes identifying and designing the routing and communication between blocks.
- **Implementation of Blocks:** This step includes the construction of the

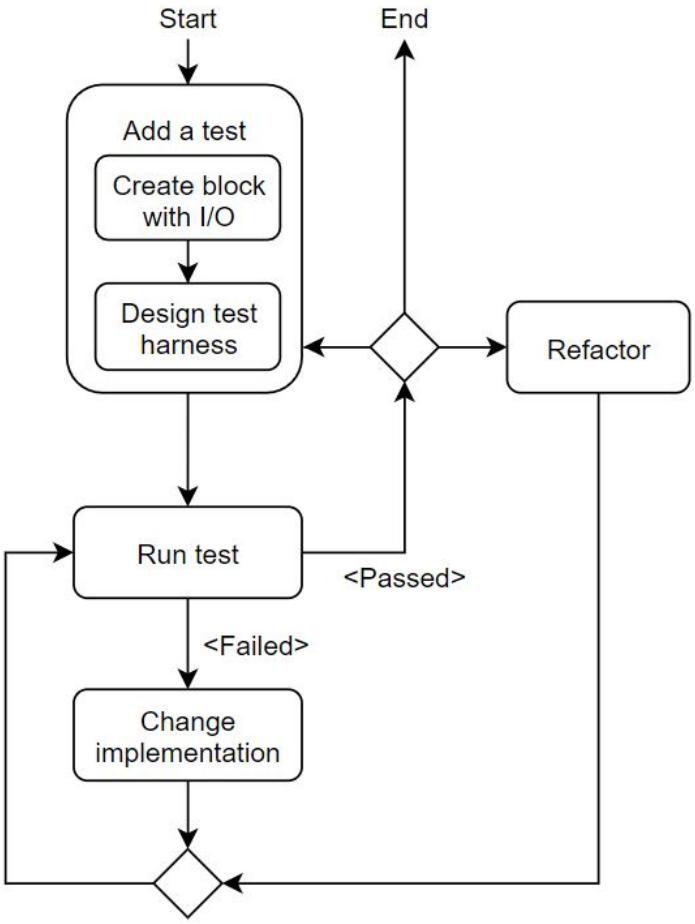


Figure 2.6: Test-driven development process, altered for a Simulink design process. The initial state of the TDD workflow has been altered for test-harness generation to be possible. After adding and running a test, the outcome of the test decides if the implementation needs to be changed or if the code is ready for refactoring (cleaning up). After refactoring and the test passing, the process can restart.

blocks and their inner workings. This step also includes the tests that accompany the implementation.

These redefined steps have been used as a foundation for the idea of model-based design. When defining metrics to measure quality of code, these steps must all be covered by the metrics to ensure that the whole process of design is evaluated.

2.5 Quality of code

The quality of code in object-oriented software can, according to Chidamber and Kemerer [16], be measured using the following metrics:

- **Weighted methods per class (WMC):** The total count of methods within

the class.
- **Depth of inheritance tree of a class (DIT):** The longest path of inheritance which ends at the class.
- **Coupling between objects (CBO):** The number of other classes that use any methods or variables of this module and vice versa.
- **Response for a class (RFC):** The number of methods which can be executed in response to a message from another class.
- **Number of children (NOC):** The number of classes that inherit this class directly.
- **Lack of cohesion in methods (LCOM):** The difference between cohesive methods and non-cohesive methods. A cohesive method uses similar instance variables, where a non-cohesive method consists of only unique instance variables or no instance variables at all.

All the above metrics describe the complexity of a method, which can serve as an indication of quality, where more complex is often less effective and harder to interpret or debug. This relation between the quality and complexity of the code has been the presumption assumed in this study.

These metrics can be modified to be used with model-based design and since inheritance has not been used with Simulink in this study, the DIT and NOC metrics can be ignored. After removing redundant metrics and modifying the ones remaining, the following model-based design quality metrics are formed:

- **Weighted blocks per module (WBM):** The total count of blocks within the module.
- **Coupling between modules (CBM):** The number of other modules that uses any blocks or functions of this module and vice versa.
- **Response for a module (RFM):** The number of blocks which can be executed in response to a signal from another module.
- **Lack of cohesion in blocks (LCOB):** The difference between cohesive blocks and non-cohesive blocks. A cohesive block uses similar signals and constants, where a non-cohesive block consists of only unique signals and constants.

These metrics have been used for measuring the quality of modules developed, where a higher value indicates lower quality, and such, a lower value indicates higher quality.

3
Methodology

This chapter explains the procedure followed when performing this study. The case and workflow used for the study is defined and the evaluation of the study is described and explained.

3.1 Case of study

To answer the first and second research questions from section 1.2, **Q1** and **Q2**, a comparison must be made between test-driven development (TDD) and test-after approach (TA). These two approaches have been defined in chapter 2. To compare the two approaches, a case will be used.

The case of study is from an automotive domain, where an electric motor which could represent a windshield wiper or a cooling fan on a car has been used when developing the system. This case has been defined to represent a common system in the automotive industry and at the same time be usable as a visual example of test-driven development.

3.1.1 Hardware

In cars, many smaller electric motors are present. Air conditioning, actuators, window lift controls and windshield wipers are just a few examples of these. To represent these systems, the hardware used in this study has been a small brushless direct current (BLDC) motor [17], driven by a 3-phase power stage [18]. The board used to load and run the implementation has been a S32K144 evaluation board [19] from NXP Semiconductors [20].

3.1.2 Requirements

The system developed for the aforementioned hardware has the following requirements specified.

1. The speed and direction of the motor should be controllable and settable between 0 and 2400 rpm with steps of 1 rpm.
2. The speed of the motor should be measured and accessible by a Controller Area Network (CAN) bus.
3. The speed and acceleration of the motor should have an upper limit within the recommended values of the motor for safety reasons.

4. The angular position of the motor should be settable with a resolution of 30° steps.
5. The angular position of the motor should be measured and accessible by a Controller Area Network (CAN) bus.
6. The number of rotations since start should be recorded and accessible by a Controller Area Network (CAN) bus.
7. An emergency brake safety feature should be in place to stop the motor.
8. A colored light on the board should visualize the current speed of the motor.

These requirements originate from the company where this study has been performed and are pre-set for this study as previously stated in section 1.1.

3.1.3 Test environment

Due to research questions **Q1** and **Q2** comprising both development of the implementation and the tests, testing is a central part of this thesis. Both unit tests in simulation and hardware-in-the-loop (HIL) tests have been used to verify the implementation. These two methods are necessary when developing hardware-dependent systems to test both software (unit tests) and functionality (HIL tests).

For unit testing in simulation, Simulink Test have been used. This software for the Simulink environment generates test harnesses for unit testing and manages tests in suites for running and debugging several tests at once. Additionally, Simulink Coverage has been used to generate coverage reports in these tests.

An HIL test is used to test a hardware system from an objective viewpoint. This is done by stimulating the system with signals that are expected to generate a certain outcome or output signals according to the requirements of the system. If the expected outcome or output is generated by the system, it passes the HIL test. The signals fed to the system simulate the intended environment for the system under test, which in this case is a car.

Contrary to the unit tests, which are developed using Simulink, the system tests used in the HIL testing are developed using Python. When tests have been designed they are implemented into the test environment which starts with the implementation being translated into C-code and built onto the hardware while simultaneously being uploaded to a continuous-integration tool-chain, called Git [21]. Git provides a service to upload implementation designs for continuous testing. When code is committed to Git, another service called Jenkins [22] initiates acceptance tests defined with yet another service called Robot Framework[23]. The system tests, which are developed using Python, are run with NI VeriStand [24] using a PXI [25]. The flowchart of the setup can be seen in Fig. 3.1. This test environment ensures that each implementation that is uploaded to Jenkins is thoroughly tested on the hardware with HIL tests.

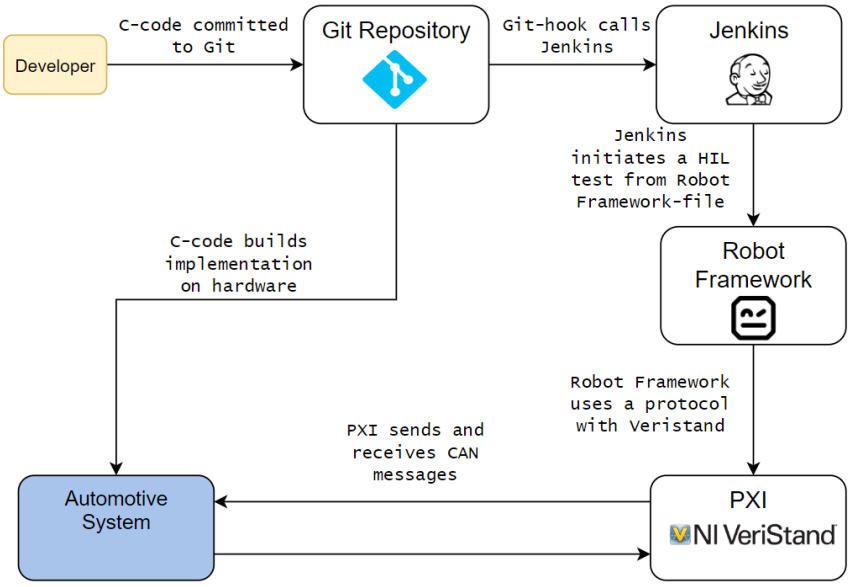


Figure 3.1: Test environment used for unit and HIL tests. The implementation code is built onto the system board and committed to Git. From the Git repository, Jenkins is called using a hook-script. Jenkins sets up and starts autonomous tests using Robot Framework. The HIL tests issued by Robot Framework uses the PXI-system with NI VeriStand to send input and receive output from the system through a CAN-protocol. This test environment enables autonomous testing during development and can be executed remotely.

3.2 Workflow

When conducting this study, the starting approach was TDD for the first module implemented. The approach was then switched to TA for the second module. Then back to TDD for the third module. This alternation between the two development approaches was continued for every module of the system. When working with each module, the following workflow was used.

3.2.1 Development and testing

Each development cycle started with an inspection of the requirements. To follow the requirements is key to every project, since this is what the customer is expecting from the implementation. The requirements have been used as a focal point for the functionality of the implementation when applying either of the two approaches.

One of the two approaches has been used to develop a module for the system. When the implementation has been fully developed and passed the tests, the module has been considered complete. After a finished module, a logbook is filled in with findings from the development. After this phase, the next module is developed using the other of the two approaches, and the cycle continues. When all modules had been developed, the system was considered complete, and evaluation ensued.

3.2.2 Data collection

To be able to answer any of the research questions and reduce bias during the evaluation of the study, an experiment logbook has been filled in after each implemented module to keep information intact. This logbook was used to record information about the modules designed and reflections about the process. The data stored in this logbook has later been used when evaluating the study and has been created following guidelines for studies found in the works of J.M. Morse et al., describing how to collect useful data for a qualitative study [26]. The following information has been noted in the logbook:

- Date of completion
- Size of module
- Time spent
- Used design approach
- Test coverage
- Functionality of module
- What was easy to accomplish?
- What was difficult to accomplish?
- General experience
- New findings regarding the workflow used
- Other important discoveries/limitations/external causes

3.2.3 Evaluation

To finally be able to answer research questions **Q1** and **Q2**, the process must be evaluated. Thus, this workflow step is the most important. As stated before, this was done after all modules for the system had been designed.

When evaluating each completed module, the metrics used were *code quality*, *code coverage of test cases* and *total time spent developing and testing a feature*. The quality of the code was measured using the model-based quality metrics from section 2.5. First, the `WBM` for each module is the number of blocks it consists of. The `CBM` has been found by counting the number of other modules that have signals going to, or coming from the module in question. To find the `RFM`, the number of blocks which are executed in response to a signal originating from the module in question have been summed up. When calculating `LCOB`, Eq. 3.1 has been used, where P is the number of blocks in the module which do not share any signals with other blocks and Q is the number of blocks in the module that do share signals with other blocks.

$$\texttt{LCOB} = \begin{cases} P - Q, & P > Q \\ 0 & P \leq Q \end{cases} \tag{3.1}$$

To verify software, every line, module and feature of a system should be covered by a test case. The coverage metric of a system shows this as a percentage of the code covered. This percentage should ideally be 100% for a system to be fully covered by its tests. This metric will determine the quality of the tests and ensure that the tests fulfil their purpose. The Simulink Coverage software has been used to measure execution-, decision- and condition coverage. Execution coverage determines whether parts of the system are executed during tests. Decision coverage determines whether decisions in flow charts and state machines are taken or not during tests. Finally, condition coverage determines if the possible outcomes for the conditions of the system are tested. This means that each condition is `true` and `false` at least one time each during test execution.

When developing features, the time spent developing and testing has been used as a metric of efficiency, with the assumption that the features compared in this regard can be considered somewhat equally complex and time-consuming. For this comparison to be more justifiable, a classification of features has been used. The classification used when defining feature size and complexity was the following:

- Extra Small: A module with no subsystems and 1-5 individual blocks.
- Small: A module with no subsystems and 6-10 individual blocks.
- Medium: A module with 1-2 subsystems of subsequent small size and 11-15 individual blocks.
- Large: 3-5 subsystems of subsequent medium size and more than 16 individual blocks.

This classification has been used for the comparison of time spent developing, where modules of the same size category have been compared with each other.

All the metrics used for evaluation combined cover the modified Booch's steps from section according to Table 3.1. As such, the metrics have been considered viable for evaluating the design process using model-based design.

Table 3.1: Covered steps of model-based design by metrics.

Metric	Identification	Semantics	Relationships	Implementation
WBM	X	X		
CBM			X	
RFM		X	X	
LCOB		X		
Coverage of tests				X
Time spent developing	X	X	X	X

4
Results

The results yielded regarding the two different practises (TDD and TA) and the found results from the experiment logbook are displayed here.

4.1 System

The system developed includes the modules depicted in Fig. 4.1. Looking back at the requirements from section 3.1.2, requirement **1**, **2**, **3**, **5** and **7** are fulfilled by the designed system, while requirement **4**, **6** and **8** remain unresolved due to time constraints on the study. The revisited requirements and their fulfillment status are visualized in Table 4.1.

Table 4.1: Revisited requirements and their fulfillment status.

		Fulfilled
1	The speed and direction of the motor should be controllable and be able to be set between 0-2400 rpm.	X
2	The speed of the motor should be measured and accessible.	X
3	The speed and acceleration of the motor should have a upper limit within the recommended values of the motor for safety reasons.	X
4	The angular position of the motor should be able to be set with a resolution of 30° steps.	
5	The angular position of the motor should be measured and accessible.	X
6	The number of rotations since start should be recorded and accessible.	
7	An emergency brake safety feature should be in place to stop the motor.	X
8	A colored light on the board should visualize the current speed of the motor.	

4.2 Quality of code

The quality of the code has been measured using modified metrics from Chidamber and Kemerer [16], as explained in chapter 2. `WMB` has been measured using Simulink

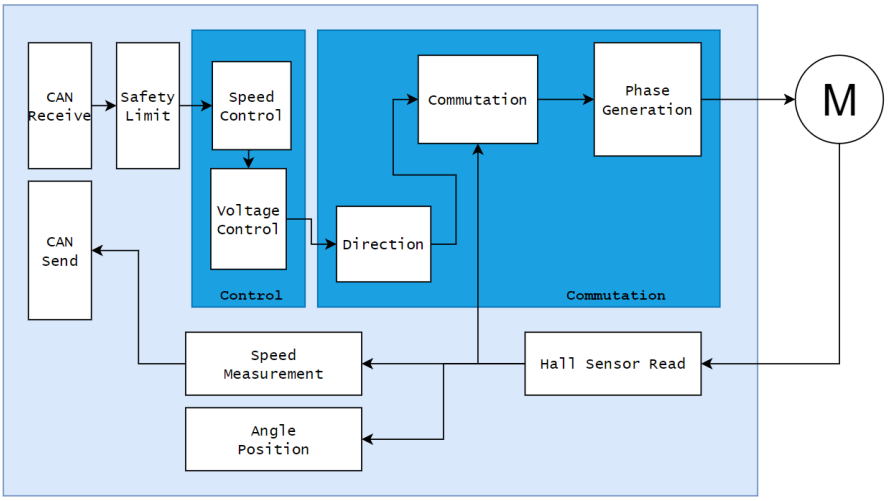


Figure 4.1: The system developed for a motor device used in the automotive industry. All modules have passed their respective tests. The CAN Send- and Receive-modules are connected to an external system with a CAN-bus (not shown in this figure since it is not part of the designed system) and the rightmost circular symbol marked with the letter 'M' is a BLDC motor.

Check, which provides a block count for each module. `CBM` and `RFM` has been measured by manual inspection of the Simulink model. A similar manner of manual inspection has been used to find the values P and Q which have been used to calculate `LCOB`.

The quality of the test cases were measured using test coverage, which has been evaluated using the Simulink Coverage software and has been measured to 100% for each module regarding execution, decision and condition coverage. The evaluated metrics for the modules can be seen in Table 4.2. The modules developed allow for three comparisons: small, medium and large modules. These comparisons can be seen in Fig. 4.2, Fig. 4.3 and 4.4.

Table 4.2: Values derived from evaluation of the modules.

Module (TDD)	Size	Coverage	WMB	CBM	RFM	LCOB
Speed Measurement	L	100%	18	2	3	0
Angle Position	M	100%	15	1	2	0
Phase Generation	M	100%	11	5	1	1
Direction	S	100%	9	4	3	4
Safety Limit	XS	100%	3	1	0	1
Module (TA)						
CAN Send/Receive	L	100%	39	7	2	0
Speed Control	L	100%	18	0	2	4
Hall Sensor Read	M	100%	15	3	1	0
Voltage Control	S	100%	9	4	2	3
Commutation	S	100%	7	3	3	1

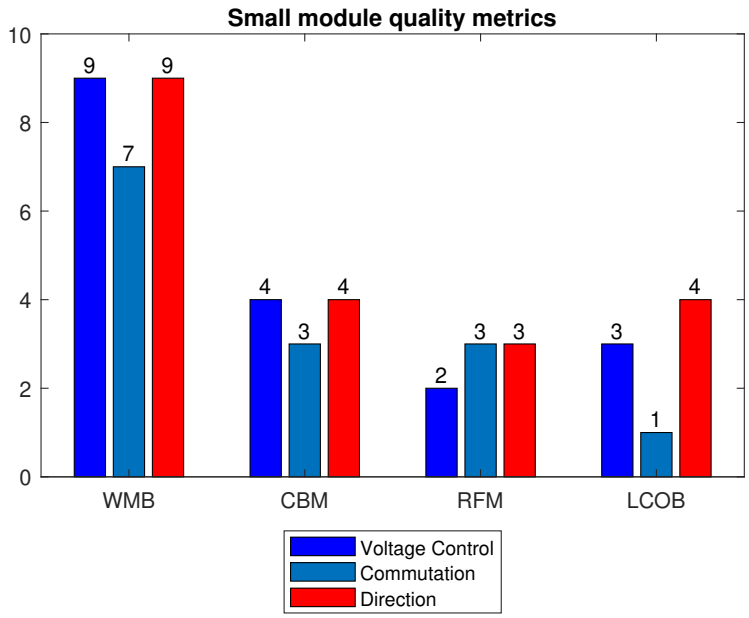


Figure 4.2: Comparison of quality metrics between TDD and TA. Higher values indicate inferior quality and subsequently lower values indicate better quality. The blue bars represent the modules developed using TA and the red bars represent the module developed using TDD.

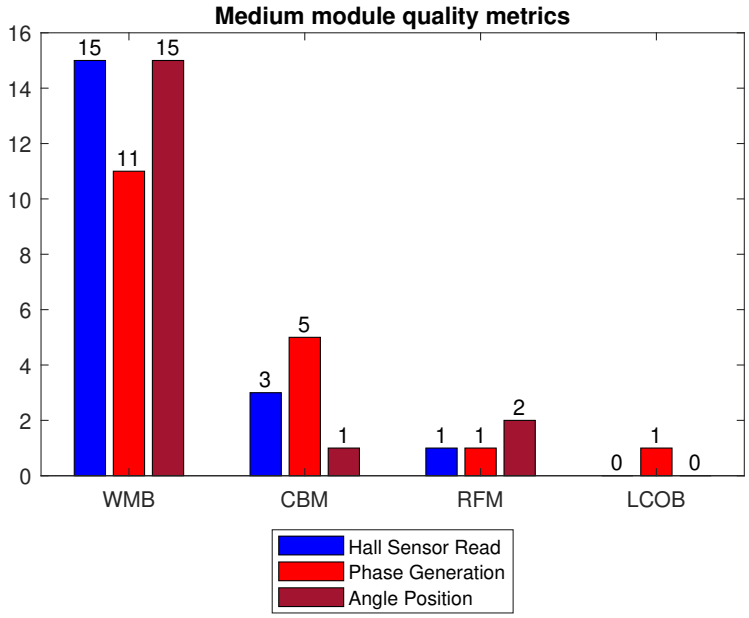


Figure 4.3: Comparison of quality metrics between TDD and TA. Higher values indicate inferior quality and subsequently lower values indicate better quality. The blue bars represent the module developed using TA and the red bars represent the modules developed using TDD.

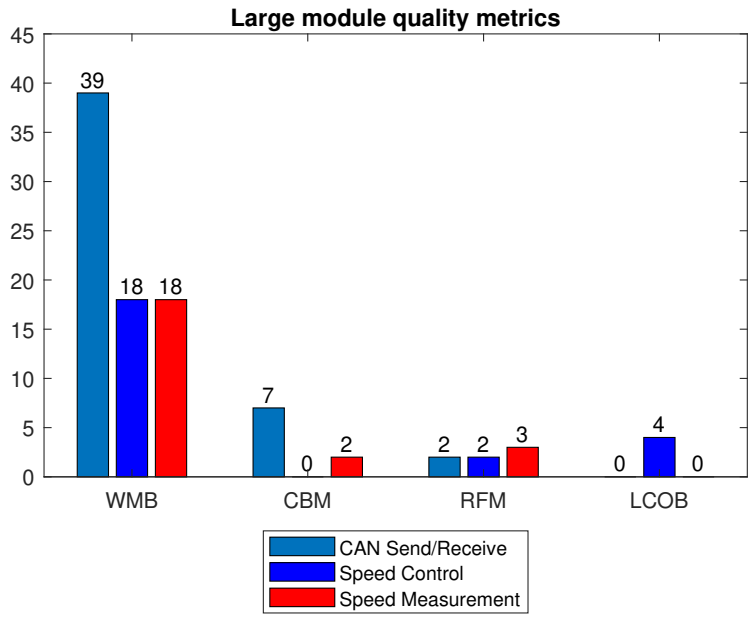


Figure 4.4: Comparison of quality metrics between TDD and TA. Higher values indicate inferior quality and subsequently lower values indicate better quality. Three large sized modules are compared. The blue bars represent the modules developed using TA and the red bars represent the module developed using TDD.

4.3 Development time

The time spent developing each module can be seen in Table 4.3. This has been measured in whole hours and includes the process of both developing implementation and designing tests for the corresponding implementation.

Table 4.3: Modules and their respective size, method and development time. It can be observed that the development time does not directly correlate to the size of the module.

Module	Size	Method	Development time
Safety Limit	XS	TDD	5h
Voltage Control	S	TA	26h
Commutation	S	TA	17h
Direction	S	TDD	13h
Hall Sensor Read	M	TA	16h
Phase Generation	M	TDD	16h
Angle Position	M	TDD	23h
CAN Send/Receive	L	TA	40h
Speed Control	L	TA	29h
Speed Measurement	L	TDD	16h

4.4 Overview comparison

To compare both quality metrics and time spent between modules as an overview of the overall performance difference between modules and approaches, radar diagrams have been used. Since the coverage of all tests have been 100% for all modules, no distinct result has been derived from the coverage metric. The coverage metric has thus not been included in the radar diagrams.

The radar diagrams in Fig. 4.5, Fig. 4.6 and Fig. 4.7 use the information from the previous sections in this chapter and are divided in a similar manner with separate comparisons of small, medium and large modules.

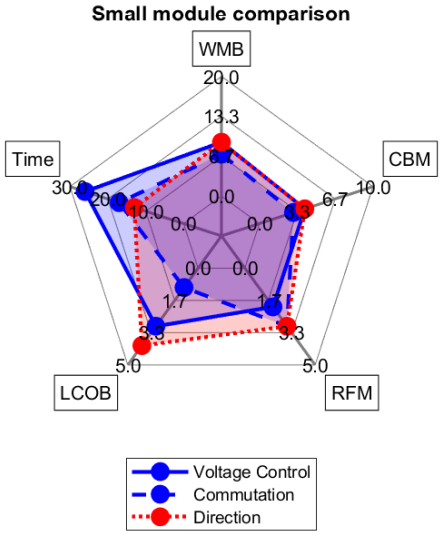


Figure 4.5: Radar diagram depicting the quality and development time of three small modules. Higher values indicate inferior quality (or efficiency regarding `Time`) and subsequently lower values indicate better quality (or efficiency regarding `Time`). The blue and red areas depict modules developed using TA and TDD respectively.

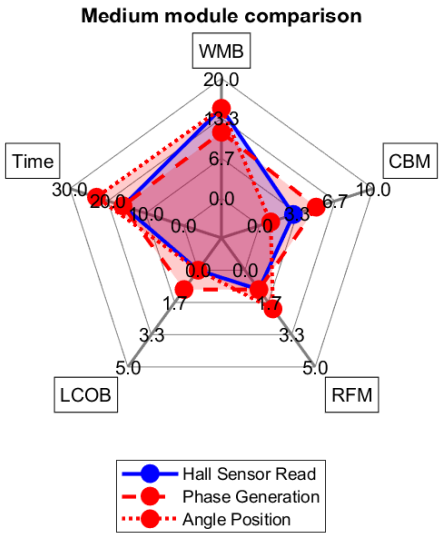


Figure 4.6: Radar diagram depicting the quality and development time of three medium modules. Higher values indicate inferior quality (or efficiency regarding `Time`) and subsequently lower values indicate better quality (or efficiency regarding `Time`). The blue and red areas depict modules developed using TA and TDD respectively.

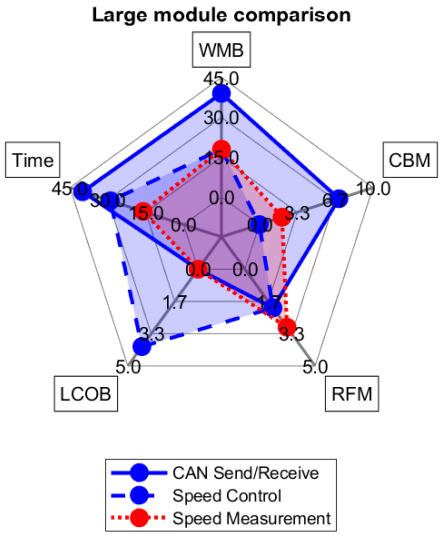


Figure 4.7: Radar diagram depicting the quality and development time of three large modules. Higher values indicate inferior quality (or efficiency regarding `Time`) and subsequently lower values indicate better quality (or efficiency regarding `Time`). The blue and red areas depict modules developed using TA and TDD respectively. **Note:** The axes for `WMB` and `Time` have been adjusted in this diagram unlike Fig. 4.5 and Fig. 4.6.

To view the trends of quality and efficiency for the two approaches, the diagram

in Fig. 4.8 has been produced. A magnified version of the diagram can be seen in Fig. 4.9. This has been produced to be able to view the changes of the values for `CBM`, `RFM` and `LCOB` more clearly. In the cases of multiple modules within the same size classification of same design approach (as two instances of TA modules in the 'Small' category, two instances of TDD modules in the 'Medium' category and two instances of TA modules in the 'Large' category) a mean value has been used in the trend diagrams.

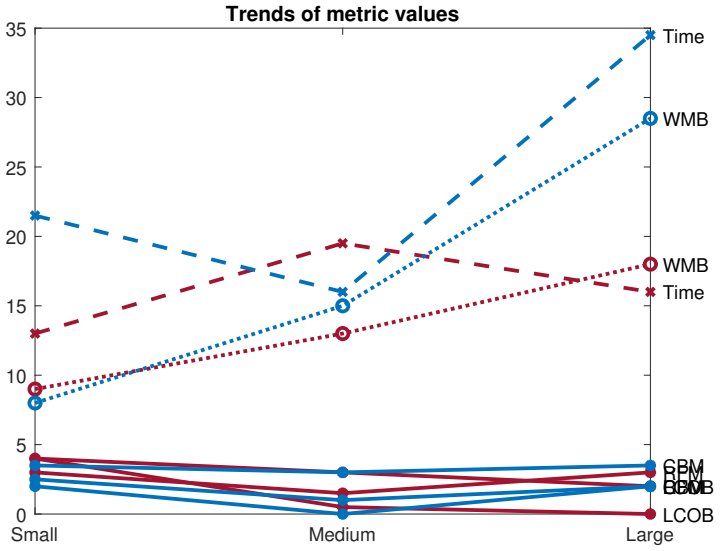


Figure 4.8: Trends of metric values for TA and TDD. Blue lines indicate that TA was used, while red indicates that TDD was used.

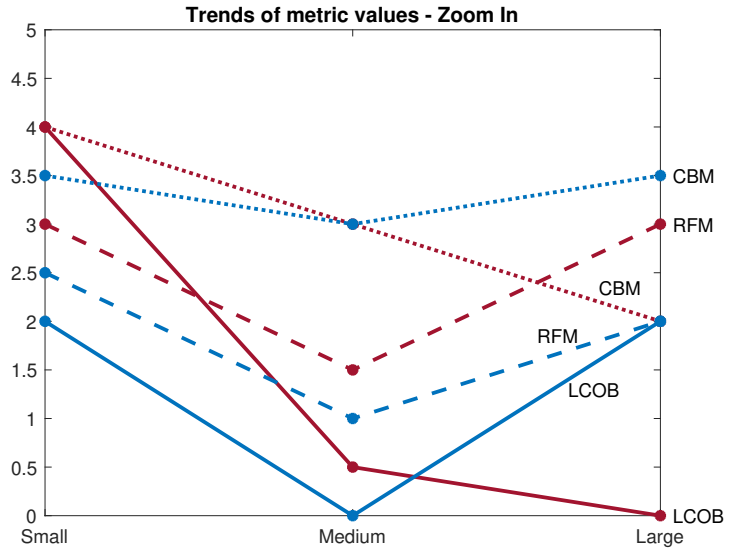


Figure 4.9: Magnified version of the trends of metric values for TA and TDD. Blue lines indicate that TA was used, while red indicates that TDD was used. `CBM` and `RFM` use dashed and dotted lines respectively in this version for a clearer view.

4.5 Findings from the logbook

The experiment logbook that has been filled out during the case study contains entries which reveal results regarding the two workflow approaches from a developer's point of view. These results regarding the two workflows have been condensed into positive and negative effects found from using the two approaches for this case study. The positive and negative effects found for TA and TDD can be found in Table 4.4 and Table 4.5 respectively. Besides these effects, some effects were found regarding how Simulink as a tool has hindered the usage of TDD. These negative effects caused by the model-based design tool can be found in Table 4.6.

Table 4.4: Logbook findings for TA.

TA	
Positive effects	Negative effects
Using the Simulink environment was easier. Less restrictions when experimenting. Developing HIL tests was simpler.	Difficult to get full coverage when designing tests. Difficult to define tests for the current implementation.

Table 4.5: Logbook findings for TDD.

TDD	
Positive effects	Negative effects
Defined tests help with understanding how the implementation could be designed. Defining and creating state machines was easier. Good understanding of the module function.	Difficult to define initial tests for a model-based implementation. Arithmetic loop problems due to the test growing by small increments.

Table 4.6: Logbook findings for Simulink.

Simulink
Negative effects on TDD
Generation of test harnesses requires an implementation block. Editing test's I/O-ports is difficult in test harnesses. Test suite needs to be manually updated with architectural changes.

5
Discussion and Conclusion

The study of test-driven development with model-based design in an automotive context has resulted in the following insights and reflections.

5.1 Discussion

The results found in chapter 4 lay the foundation for this discussion. Here, the research questions, referred to as **Q1**, **Q2** and **Q3**, are addressed. These questions have been stated in section 1.2.

5.1.1 Quality metrics and development time

To answer **Q1** and **Q2**, the results regarding the quality metrics and development time can be observed. Observing the diagrams in Fig. 4.2, Fig. 4.3 and Fig. 4.4, it is unclear whether any conclusive patterns or strong behaviors regarding any of the isolated metrics can be identified. There is a slight trend of `RFM` being lower for TA, which would imply, according to Booch's modified design steps, that TA is slightly better during the *Semantics* and *Relationships* steps of design. However, this is hard to determine if this is the case due to the small margin and small sample size that this statement is based on.

As for the other metrics, more definite conclusions can be drawn from the radar diagrams. Looking at Fig. 4.5, it can be observed that for smaller and possibly simpler modules, TA has yielded lower or similar values for both `WMB`, `CBM`, `RFM` and `LCOB`. Regarding development time however, TDD has yielded a lower value. This points to the conclusion that, for small modules, TA is the approach that generates better quality models while TDD is more time efficient.

When observing Fig. 4.6 a different result can be observed. In this diagram, the module designed using TA has yielded values for `WMB`, `CBM`, `RFM` and `LCOB` equal to or somewhere in between the values yielded by modules developed using TDD. This makes it difficult to determine if either of the approaches is preferable regarding quality or efficiency.

As for the final radar diagram, Fig. 4.7, it can be seen that TDD has generated equal or lower values than TA for all the quality metrics except `RFM`. This higher value of `RFM` together with the value 2 for `CBM` (which places it in between the two

different TA modules' values of CBM) indicates that TDD has proven more problematic during the *Relationships* design step of the development process. As with previous statements like this in this section, it is hard to determine this behavior due to the small sample size. The development time has been lower for the module developed with TDD, which indicates that TDD has been more efficient.

Finally, looking at the trends for the two approaches found in Fig. 4.8, a shift from lower to higher time- and WMB-values can be observed for TA, and the inverse for TDD. One could speculate that this indicates that TDD would prove more efficient for larger modules, and possibly for larger systems, while TA has the same effect for smaller modules. The trends of the other values tells us that smaller modules often are generally more complex, which is interesting. This could be due to the nature of Simulink (or model-based design in general) where there are more supporting blocks for intricate functions which is more abundant in larger modules. This leaves the smaller modules with many, simpler blocks. This, in turn, raises the complexity of the smaller modules.

With the aforementioned reasoning in mind, the answer for **Q1** is thus:

- When using the model-based design tool Simulink, test-driven development generates a better quality of implementation when designing large systems with more than 16 blocks and multiple sub-systems. The test-after approach generates a better quality of implementation for smaller systems, with fewer blocks and no subsystems.

5.1.2 Practical comparison of TDD and TA

To thoroughly answer **Q2**, the practical differences of the two approaches can be observed. From the experimental logbook, results regarding the practical positive and negative effects of the two approaches can be derived.

As can be seen in Table 4.4, the positive effects of TA include the words *easier*, *less restricted* and *more simple*. The workflow is more trivial in the sense that you are not as controlled by guidelines as when using TDD. On the other hand, this freedom of TA when creating the implementation leads to the negative effects of hardships when defining and designing tests for full coverage, which can be seen in the same table.

Using TDD has proved more efficient during certain parts of implementation, according to the positive effects found in Table 4.5. When creating state-machines and when designing module functions, a good understanding of the functionality of the module as a whole is important. The case could be that TDD helps with this overhead understanding of the module, since the action of defining tests (which are derived from the requirements to test these functions) gives the developer that deeper understanding. The two negative effects found are of a practical nature and have direct ties to the tool used. An arithmetic loop occurs when a continuous signal

loops infinitely and thus fills up the computer's memory. A test harness is a sort of loop, where arithmetic loops can occur. During initial design of the tests, these loops easily happen since no blocks exist to discretise the signal.

With the aforementioned reasoning of section 5.1.1 and this section in mind, the answer for **Q2** is thus:

- When using the model-based design tool Simulink, test-driven development is more efficient when developing an automotive system of smaller scale.

5.1.3 Simulink's impact on the results

To answer **Q3**, the findings from the logbook regarding Simulink as a tool is observed. Looking at Table 4.6, it is clear that the testing has been more difficult to conduct with TDD than with TA. All the found problems with Simulink and TDD applies to the Simulink Test environment, not Simulink in general. This leads to the conclusion that Simulink Test is not designed for TDD, which indirectly hinders development with Simulink since testing is a crucial part of the development process.

With the aforementioned reasoning in mind, the answer for **Q3** is thus:

- When using the model-based design tool Simulink, neither test-driven development nor the test-after approach is hindered by Simulink as a design tool. However, the Simulink Test software hinders test-driven development and is clearly created for the test-after approach.

5.2 Future work

A larger study with more developers, thus a larger sample, would be an interesting next step to get closer to a definitive answer regarding the effectiveness of test-driven development. The development of larger systems with more modules would give a clearer picture of how the produced quality is affected by the design methodology. A comparison between using model-based design and code-based design would also be an interesting inclusion to this research question.

5.3 Conclusion

Test-driven development, where tests are designed before the implementation, has been compared to the contrariety test-after approach, where tests are designed after the implementation. The comparison has been done with model-based design as the method of development and through a case study in the automotive domain. The quality and efficiency of the two approaches have been studied. By alternating between the two approaches during development of the automotive system, an estimation of the two approache's quality and efficiency could be done with a set of metrics. These metrics together with insights logged throughout the process lead to

the conclusion that test-driven development produces higher quality modules when used to design larger systems with sub-systems. The test-after approach produces higher quality modules when used to design smaller modules. Test-driven development showed to be more efficient than the test-after approach. However, the model-based design tool used for this study (Simulink) showed to favor the test-after approach during testing.

In conclusion, this study provides an inclination that test-driven development is favorable when designing larger automotive systems.

Bibliography

[1] U. Eklund, H. H. Olsson, and N. J. Strøm, "Industrial challenges of scaling agile in mass-produced embedded systems," in *International Conference on Agile Software Development*. Springer, 2014, pp. 30–42.

[2] G. Nolan, "Test driven development," in *Agile Swift*. Springer, 2017, pp. 131–167.

[3] A. Cline, *Agile development in the real world*. Springer, 2015.

[4] J. Friedman, "Matlab/simulink for automotive systems design," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1. IEEE, 2006, pp. 1–2.

[5] A. Gupta and P. Jalote, "An experimental evaluation of the effectiveness and efficiency of the test driven development," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 285–294.

[6] L. Madeyski, *Test-driven development: An empirical evaluation of agile practice*. Springer Science & Business Media, 2009.

[7] S. Romano, D. Fucci, G. Scanniello, B. Turhan, and N. Juristo, "Findings from a multi-method study on test-driven development," *Information and Software Technology*, vol. 89, pp. 64–77, 2017.

[8] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[9] G. Nicolescu and P. J. Mosterman, *Model-based design for embedded systems*. Crc Press, 2018.

[10] Mathworks, "Simulink - Simulation and Model-Based Design," 2020. [Online]. Available: https://www.mathworks.com/products/simulink.html

[11] H. Altinger, "State-of-the-art tools and methods used in the automotive industry," in *Automotive Systems and Software Engineering*. Springer, 2019, pp. 59–73.

[12] Mathworks, "Simulink Test," 2020. [Online]. Available: https://www.mathworks.com/products/simulink-test.html

[13] ——, "Simulink Coverage," 2020. [Online]. Available: https://www.mathworks.com/products/simulink-coverage.html

[14] ——, "Simulink Requirements," 2020. [Online]. Available: https://www.mathworks.com/products/simulink-requirements.html

[15] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston, "Object-oriented analysis and design with applications," *ACM SIGSOFT software engineering notes*, vol. 33, no. 5, pp. 29–29, 2008.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[17] NXP Semiconductors, "FRDM-MC-LVMTR," 2020. [Online]. Available: https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/low-voltage-3-phase-motor-kit-for-frdm-platform:FRDM-MC-LVMTR

[18] ——, "Low-Cost Motor Control Solution for DEVKIT Platform," 2020. [Online]. Available: https://www.nxp.com/design/development-boards/automotive-development-platforms/hardware-tools-accessories/low-cost-motor-control-solution-for-devkit-platform:DEVKIT-MOTORGD

[19] ——, "S32K144 Evaluation Board," 2020. [Online]. Available: https://www.nxp.com/design/development-boards/automotive-development-platforms/s32k-mcu-platforms/s32k144-evaluation-board:S32K144EVB

[20] ——, "Development Kit for sensorless BLDC | NXP," 2020, [Online; accessed 19-February-2020]. [Online]. Available: https://www.nxp.com/design/development-boards/automotive-motor-control-development-solutions/arm-based-solutions-/s32k144-development-kit-for-sensorless-bldc:MTRDEVKSBNK144

[21] git, "Git," 2020. [Online]. Available: https://git-scm.com/

[22] The Jenkins Project. (2020) Jenkins. [Online]. Available: https://www.jenkins.io/

[23] Robot Framework Foundation. (2020) Robot Framework. [Online]. Available: https://www.robotframework.org/

[24] National Instruments, "What Is VeriStand?" 2020. [Online]. Available: https://www.ni.com/sv-se/shop/data-acquisition-and-control/application-software-for-data-acquisition-and-control-category/what-is-veristand.html

[25] ——, "PXIe-1071," 2020. [Online]. Available: https://www.ni.com/sv-se/support/model.pxie-1071.html

[26] J. M. Morse, M. Barrett, M. Mayan, K. Olson, and J. Spiers, "Verification strategies for establishing reliability and validity in qualitative research," *International journal of qualitative methods*, vol. 1, no. 2, pp. 13–22, 2002.