Evaluation of Test-Driven Approaches for Embedded Software Development

Samuli Mononen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 30.07.2020

Supervisor

Prof. Ivan Vujaklija

Advisor

M.Sc. Eero Vaajoensuu

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Samuli Mononen

Title Evaluation of Test-Driven Approaches for Embedded Software Development

Degree programme Automation and Electrical Engineering

Major Translational Engineering **Code of major** ELEC3023

Supervisor Prof. Ivan Vujaklija

Advisor M.Sc. Eero Vaajoensuu

Date 30.07.2020 **Number of pages** 58+9 **Language** English

Abstract

Software testing is an essential part of software development and one of its goals is to reduce the amount of software bugs. The first stage of software testing, unit testing, is especially important as software bugs are the more expensive to fix the later they are detected. However, unit testing is neglected regrettably often. The usage of unit testing can be promoted by practising test-driven development (TDD), which is more commonly utilised in the traditional software development. Unfortunately, in the development of embedded software it is considered to be difficult to adopt, because embedded software is dependent on the specifics of the hardware it runs on. In order to ease the adoption, a guideline for TDD of embedded software was developed in this thesis. The guideline includes six testing tools and four mocking techniques. The techniques were tested in an embedded case project with one of the presented tools, Google Test. Each of the techniques were deemed to enable a real implementation to be replaced with a test double. The replacement allowed to overcome the challenges related to hardware dependencies and thus, enabled testing on host. The thesis also covers testing on the target hardware by showing how Google Test was able to be modified to run on target and how the required communication was set up between the target hardware and the development machine. In the future, it could be studied whether these testing tools and mocking techniques are sufficient for unit testing embedded software that utilises a real-time operating system.

Keywords test-driven development, embedded software, embedded systems, unit testing, google test

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Samuli Mononen

Työn nimi Testivetoisten sulautettujen ohjelmistojen kehitystapojen arviointi

Koulutusohjelma Automaatio ja sähkötekniikka

Pääaine Translationaalinen tekniikka **Pääaineen koodi** ELEC3023

Työn valvoja Prof. Ivan Vujaklija

Työn ohjaaja DI Eero Vaajoensuu

Päivämäärä 30.07.2020 **Sivumäärä** 58+9 **Kieli** Englanti

Tiivistelmä

Ohjelmistotestaus on tärkeä osa ohjelmistokehitystä ja yksi sen tavoitteista on vähentää ohjelmistovirheiden määrää. Ohjelmistotestauksen ensimmäinen vaihe, yksikkötestaus, on erityisen tärkeä, sillä ohjelmistovirheiden korjaaminen on sitä kalliimpaa, mitä myöhemmin ne havaitaan. Yksikkötestaus kuitenkin sivuutetaan valitettavan usein. Sen käyttöä voidaan edistää harjoittamalla tekniikkaa nimeltä testivetoinen kehitys (engl. *test-driven development, TDD*), jota käytetään yleisimmin tavallisessa ohjelmistokehityksessä. TDD:n käyttöönottoa pidetään kuitenkin vaikeana sulautettujen ohjelmistojen kehityksessä, koska sulautetut ohjelmistot ovat riippuvaisia laitteistosta, jolla ohjelmistoa suoritetaan. Tässä työssä TDD:n käyttöönoton helpottamiseksi kehitettiin ohjenuora, joka koostuu kuudesta testityökalusta ja neljästä riippuvuuksien korvaamistekniikasta (engl. *mocking techniques*). Tekniikoita testattiin sulatetussa projektissa yhdellä ohjenuoran testityökaluista, Google Testillä. Jokainen tekniikoista todettiin toimivaksi alkuperäisen toteutuksen korvaamisessa testisisijaisella (engl. *test double*). Tämä poisti laitteistoon liittyvät riippuvuudet ja siten mahdollisti testaamisen kehitysympäristössä. Työ kattaa myös testaamisen kohdelaitteistolla näyttämällä miten testityökalu saatiin muokattua toimimaan kohdelaitteistolla ja miten vaadittava kommunikointi pystytettiin kehitysympäristön ja laitteiston välille. Tulevaisuudessa olisi hyödyllistä tutkia, voidaanko näitä testityökaluja ja riippuvuuksien korvaamistekniikoita hyödyntää testattaessa sulautettuja ohjelmistoja, jotka käyttävät reaaliaikaista käyttöjärjestelmää.

Avainsanat testivetoinen kehitys, sulautetut ohjelmistot, sulautetut järjestelmät, yksikkötestaus, google test

Preface

The basis of this research stemmed from having experienced significant challenges in legacy projects for which executable tests had not been created. Instead, they had only been tested with ad hoc methods. Since there were no tests to be executed, it was difficult to ensure that new changes to the implementation would not cause unintended effects to existing features. Later, I came across the TDD technique and realised that if the projects had applied it, the continuation of the development of those projects later on could have been far easier. However, as explained in this thesis, the technique was not created for the embedded domain. Hence, I was interested in researching the applicability of TDD for embedded software, which then led to the development of the guideline presented in this thesis.

I would like to take this opportunity to thank a few people involved in this process. First and foremost, I am grateful to my supervisor Ivan Vujaklija especially for helping me to form the aim for this thesis but also for an excellent guidance overall. I would also like to thank my friend Aleksi Mäkinen for assisting with the writing procedure and colleague Teemu Mäntykallio for helping with the implementation. In addition, I thank my instructor Eero Vaajoensuu for proofreading. Lastly, I would like to thank my dearest, Nina Haatanen, for supporting me far from the very beginning until the very last end and taking care of my mental health during this entire process.

Otaniemi, 30.07.2020

Samuli Mononen

Contents

Abbreviations

AAA	Arrange-Act-Assert
ARM	Advanced RISC Machines
BSD	Berkeley Software Distribution
BSL	Boost Software License
C	C programming language
C++	C++ programming language
CI	Continuous Integration
FSM	Finite state machine
GPIO	General-purpose input/output
GPL	General Public License
GTEST	Google Test
GMOCK	Google Mock
LGPL	Lesser General Public License
HAL	Hardware abstraction layer
IDE	Integrated Development Environment
MIT	Massachusetts Institute of Technology
NIST	National Institute of Standards and Technology
OS	Operating System
SUT	Software under test
PTHREADS	Posix threads
RISC	Reduced instruction set computer
RTTI	Run-time type information
TDD	Test-Driven Development
UML	Unified Modeling Language
U.S	United States
USB	Universal Serial Bus
XP	Extreme Programming

1 Introduction

In recent years, due to the growing need to rely on machines and computer systems, the resulting financial losses exacted by software failures have grown to extreme measures. Software failures occur due to software bugs. Software bugs can be, for example, missing or incorrect pieces of code [1]. In a report released in 2018 [2], a software-testing company Tricentis analysed 606 software failures that occurred during 2017. The failures were estimated to result in financial losses worth 1.7 trillion dollars. Unfortunately, the failures caused by software bugs are not only financial but also life-threatening. The bugs have already compromised vital public services, including, but not limited to, healthcare, defence, government administration and energy production. Therefore, the need for improvements in software testing has significantly risen in response to these wider issues.

Software testing is a procedure, which aims to detect and prevent the existence of software bugs. It can consist of multiple verification levels, such as unit, integration, system and acceptance testing, stated from the lowest to highest levels [1], [3]–[7] Unit testing aims to test the execution of the smallest possible component in a system independently of the other components. Integration tests combine components to verify their aggregation. System testing ensures that the system delivers what has been specified. Finally, acceptance testing checks that the system delivers what was requested. While the detection of bugs in any testing level is desired, they should preferably be detected during unit testing. That is because the costs to fix the bugs are significantly higher if they are found during later testing levels [3]. However, the usage of those tests is often neglected [8], [9].

In order to ensure that each feature of a software will be unit tested, test-driven development (TDD) can be used. TDD is a technique to build software in increments. First, a failing test is written and only then is the desired feature programmed [10]. TDD has proven to result in several benefits [11]–[13] and is most commonly practised in the development of non-embedded software, such as desktop or mobile applications. However, the usage of TDD has been low in the embedded software domain [14], [15], which covers software that is built directly into a device or a piece of hardware, such as automotive, home and medical applications [16]–[19]. The usage has been low because TDD is more difficult to apply to embedded than to non-embedded software. TDD is difficult to apply in the embedded domain because embedded software is dependent on the particularities of the individual hardware it runs on. Nevertheless, studies have shown that TDD of embedded software is feasible [14], [20], [21] and that the benefits can outweigh its cost [20]. Only a few studies [21], [22] have so far concentrated on the tools and methods required for the large-scale adoption of TDD.

Therefore, the aim of this thesis is to develop a systematic guideline for TDD of embedded software. Through this guideline, the adoption of TDD is eased, which makes developers more likely to practise TDD and thereby reduce the amount of undiscovered software bugs and consequently the overall development costs. In order to form this guideline, the thesis collects and combines existing tools and methods, and evaluates them through a set of examples. The evaluation is to be performed on an embedded motor controller case project.

The focus is on statically typed object-oriented programming languages and the language of interest is selected to be C++ due to the recent increase in popularity among embedded software developers [19]. The thesis covers tools that are free to use to ensure that the guideline presented in this thesis can be utilised by anyone. Hence, commercial tools are excluded from the thesis. In addition, the thesis concentrates on reducing software bugs in the development phase instead of during the testing phase. Lastly, the intention is to study testing of software by executing it. Hence, the focus is on dynamic testing and static testing is therefore excluded from this thesis.

The thesis is organised as follows. Chapter 2 introduces the principles of software testing and test-driven development. Chapter 3 collects existing tools and methods from the literature and shows how to set up one of the tools, Google Test. The collected methods are experimented in an embedded case project and an example test case from the project is illustrated with all of the identified methods in Chapter 4. Chapter 5 evaluates Google test and the methods based on the example test case. Finally, Chapter 6 states the conclusions of this thesis.

2 Background

The losses due to software failures increased over 2000 % in the U.S from the year 2002 to 2017 [2], [23]. Software failures occur due to software bugs and errors, further elaborated in subsection 2.1.1. A study released by National Institute of Standards and Technology (NIST) [23] estimated the losses of software bugs to be worth of 59.5 billion dollars annually in the U.S in 2002. Later, a software-testing company Tricentis [2] estimated that software bugs resulted in the world-wide losses worth of 1.7 trillion dollars (with 75 % of those in the U.S) in 2017. If the growth of the losses is assumed constant, the observed growth was a terrifying 22.7 % every year from 2002 to 2017. The reason why software failures have become so common is simply due to the massive growth of utilisation of software, which started slowly in the sixties with the introduction of computing according to Mili and Tchier [24]. Later, it continued to grow exponentially after the emergence of the internet. Nowadays, software can be found in all aspects of our lives and it has a notable and continuously increasing impact to the global economy.

Software failures are often simple mistakes that can have devastating effects. In 1991 [5] a missile of the U.S missed the target and killed 28 of its own soldiers. The reason for the miss was due to a slight error in the clock of the system after 14 hours of operation. In 1996 [5], [25] an unmanned rocket, with an estimated development cost of 7 billion dollars, exploded only after 40 seconds of its launch. The rocket exploded due to overflow conditions that had occurred because the software tried to convert a large number (64-bit) into a memory space (16-bit) into which it could not fit. In 1999 [26] the control to the space probe Mars Climate Orbiter was lost and the probe was destroyed. The destruction was due to the mixed use of imperial and metric units. An example of a wider scale software issue occurred on the first of January in 2000 [5], [25], when a large portion of all software suffered from the famous Y2K bug. Back then, software had utilised only two digits to represent a year. However, since the century changed, four digits would have been required to represent the correct year.

Software bugs can be prevented by performing software testing, which is a systematic process that aims to detect software bugs [27]. The process of identifying and correcting the bugs is called debugging [4], [5]. Software testing can lead to great savings. As an example, NIST [23] estimated that a third of the fails that occurred in 2002 could have been avoided with proper software testing, which could have led to savings worth of 22.2 billion annually. Software testing and its concepts are described in section 2.1 and since this thesis aims to reduce the number of software bugs with a technique called TDD, the technique is presented and discussed in section 2.2.

2.1 Software testing

Software testing cannot detect all bugs and it is often performed to show that the software does not contain any bugs. However, according to Myers et al. [27], the intention of software testing is not to prove that a software is free of bugs but only to aim to discover them. In fact, according to them, it is not even possible for a test

program to detect all bugs of a software. Since this thesis deals with reduction of software bugs and resulting failures, the exact terms are defined in subsection 2.1.1 to avoid any misunderstandings. The reason why test programs cannot detect all bugs is explained in subsection 2.1.2. However, in order to understand the reason, software testing methods are first introduced in the same section. Software testing can be performed on different levels, which are described in subsection 2.1.3. Finally, as this thesis aims to develop a guideline for embedded software developers, different types of software and especially embedded software are presented in subsection 2.1.4.

2.1.1 Software bug, error and failure

The term software bug is a synonym for a software fault or software defect, but the terms software failure, software error and software bug each have their own distinct meanings. The meanings of the terms, however, are related, and the terms are often utilised interchangeably. Moreover, different sources do not completely agree on the definitions of these terms.

Bertolino and Marchetti [1] define software bugs as the initial reason for a software failure to occur. Software bugs may be, for example, missing or incorrect pieces of code. A software bug may cause a software error, which is an unstable state of a software. Finally, a software error may cause a software failure, which is an inability of the software to meet its specification. Chauhan and Naresh [3] disagree with Bertolino and Marchetti with the definitions for a software bug and software error. Their definition for a software bug combines the effect of a software bug and software error defined by Bertolino and Marchetti. Hence, they do not define software errors as unstable states of software. Instead, they define them as mistakes made by programmers that have caused the software bugs in the first place. IEEE Standard Glossary of Software Engineering Terminology [28] does not provide unambiguous definitions for the terms but multiple definitions for each of them. The different definitions given by Bertolino and Marchetti as well as Chauhan and Naresh can all be found from the IEEE standard.

This thesis follows the definitions provided by Bertolino and Marchetti. The definitions utilised can also be summed up as a possible “bug-error-failure“ chain, as shown in Figure 1. As an example of the chain, consider the following metaphor. A measurement system may have a type error in the software, such as the type error in the rocket presented in the beginning of this Chapter. The type error may cause inaccuracy in the measurement. Finally, the inaccurate measurement may result in an undesired action. In the metaphor, the type error is a software bug, the resulting inaccuracy is a software error and lastly, the undesired action is a software failure.

2.1.2 Software testing methods

Myers et al. [27] divide software testing into black-box and white-box testing methods. In the *black-box* testing method the tester views the software as a black box, inside which the tester cannot see. Hence, the internal behaviour of a software is not of interest in black-box testing. Therefore, the tester does not even have to possess an access to the internal implementation and the software is tested simply by

Figure 1: A software bug may cause a software error, which in turn may result in a software failure. In addition, one bug may eventually cause multiple failures and one failure may originally be caused by the combination of multiple bugs [3]

providing possible inputs to the software and verifying that the software functions according to the specification. Black-box testing is also referred to as data- and input/output-driven testing.

The black-box testing method cannot be utilised to find all software bugs [27]. If testing is performed according to the black-box testing method, all bugs could theoretically be found by performing *exhaustive input testing*. Exhaustive input testing is a technique in which test cases are created for every possible input condition. A test case is code that is utilised to test a certain functionality of software by establishing the desired preconditions and verifying that the expected post conditions are fulfilled [10]. However, Myers et al. [27] note that exhaustive input testing is not possible in practice due to the virtually infinite possible test cases and present an example. If a C++ compiler would be desired to be tested completely, there would have to be a test case for every possible combination of software code that the compiler can compile. Moreover, every invalid code would also require test cases to ensure that the compiler does not compile code that is syntactically incorrect.

In *white-box* testing the software is considered to be a white or transparent box, into which the tester can see [25]. Therefore, in white-box testing the tester is aware of the internal implementation and creates test cases that execute different paths of a software by considering the logic of the software. White-box testing is also known as clear, open, glass and transparent box testing, as well as logic-driven, code-based and structural testing.

As with black-box testing, the white-box testing method cannot be utilised to find all software bugs either [27]. If white-box testing is performed to find all the bugs, *exhaustive path testing* must be performed. Exhaustive path testing is a technique in which test cases are created for every possible path of the software. Hence, according to the technique, if the tests could cover each path, software could be tested completely. Unfortunately, exhaustive path testing is only feasible in theory due to three reasons. Firstly, Myers et al. [27] state that the number of possible paths of logic in the software may be enormous by presenting an example. If a non-terminating loop would have one if statement inside it and it would iterate twice, it would require only four unique paths to be tested completely. The paths can be computed using the formula $(a+1)^b$, where a is the number unique paths during one loop cycle and b is the number of loops to be executed. Four different test cases are simple to create but unfortunately, the number of required test cases increases rapidly. If there would be four nested if statements, and the loop would iterate 20 times, the number of required test cases would grow almost to 100 trillion. Moreover,

software in the real world is often significantly more complex and might require virtually an infinite number of test cases. Secondly, even if all the paths could be tested, which is possible in theory, it would not ensure that there would not be any bugs in the software. That is because the software could pass all the tests without even performing what it is really intended to do. Lastly, the execution of all the paths would still not ensure that the software works correctly with all possible input values. In conclusion, neither of the two methods can be utilised to completely test a software. Fortunately, a considerable amount of software bugs can be avoided without complete software testing.

2.1.3 Software testing levels

Software testing can be dived into static and dynamic testing [4], [5]. *Static testing* is testing in which the software is not executed. Examples of static testing are code reviews and the usage of static code analysis tools. *Dynamic testing*, on the other hand, is testing in which the software is executed. It is often dived into different levels. Multiple sources [1], [3]–[7] define four levels of software testing as: unit, integration, system and acceptance testing. Each of them aims to discover software bugs with a different set of tests. In addition, the levels may be varied depending on the system developed [4].

Unit testing is the first level of testing. It concentrates on ensuring that a unit fulfils its specification without the interaction with other units. Bertolino and Marchetti [1] define a unit as the smallest software component, which can be tested. It may consist of only a couple or up to hundreds of lines of code. Unit testing is usually performed by the developer, who has created the unit, because the developer understands how the unit functions.

Integration testing focuses on verifying that the communication between the units functions correctly. Hence, the aim is to expose bugs in the interfaces and interactions between units or systems. An interface may be internal or external. Internal interfaces are utilised by units within the software to communicate with one another. External interfaces cover the interfaces that are exposed to third party developers. Integration testing can be done either by the developers or by separate testers.

System testing verifies a complete system. Verification means ensuring that the software fulfils its requirements and that the software has been built as specified [19]. The requirements may consist of both, functional and non-functional requirements. Examples of non-functional requirements are performance, stress and robustness requirements. System testing is normally performed by separate testers.

Acceptance testing is similar to system testing as it also tests a complete system based on predefined requirements. However, instead of verification, acceptance testing validates the system. Validation means that the software performs what it is intended and that the right software has been built [19]. Acceptance testing should be performed by the customer.

Software testing is expensive. In fact, it may consume up to half or even more of the man-hours spent on a project [29], [30]. Due to the high cost of software testing,

it is not surprising that especially the first levels of testing are often dismissed [8], [9]. The first levels can be discarded, because if the product functions correctly in system and acceptance testing, there is no need for testing of the units or interfaces. However, while the dismissal of the first levels may seem as a way to reduce overall development costs, such an action may actually result in significant increase in development costs. The increase may occur, because the costs for debugging a bug are much higher if the bug is detected during higher than lower levels of testing [3].

The increase in debugging costs does not concern only software testing levels. Boehm [31] stated almost 40 years ago that the costs grow exponentially as the software moves from one development phase to another. All in all, the debugging costs are the lesser the earlier a bug can be detected. Therefore, if a software bug can be detected already at unit testing, the software development costs can be significantly lesser than if the bug would have been detected at a later software development stage. Even better, if unit tests would be created and executed during development, the debugging costs could be reduced to minimum. Hence, a technique utilising such an activity is presented in section 2.2.

2.1.4 Software types

The definition of the different software types is hardly unambiguous. Manzoor [32] defines software types into system, programming and application software. However, he also notes that the distinction is often imprecise. System software functions as a middleware between application software and the underlying hardware. It is utilised in operating systems and device drivers. Programming software consists of tools for the software development, such as compilers, linkers and debuggers. Application software allows the end users to complete an intended task and is nowadays utilised in almost every form of human activity. Hence, it is hardly feasible to provide a short but an extensive list of examples. Nevertheless, some examples of application software can be found from the fields of gaming, medical, industrial automation and communication.

Forward and Lethbridge [33], on the other hand, propose a different classification for defining the types of software, which consists of four types: data-dominant, systems, control-dominant and scientific software. Data-dominant software includes software for managing data and users of it are consumers, businesses and engineers. Systems software is utilised for achieving levels of abstraction in operating systems, device drivers and servers. Control-dominant software controls a particular system, which can range from small embedded systems to large control systems such as air traffic control. Finally, computation-dominant software includes software for solving conceptual problems and is utilised for research, artistic and scientific purposes.

Software can also be divided into non-embedded and embedded software [34]. Non-embedded software can be run on different devices and platforms and is nowadays found from computers and smart phones as an example. It is also utilised in web applications and in the operating systems on which the computer and mobile applications run on. In contrast, embedded software is specialised for the hardware on which it is executed and has time and memory related constraints [16]. More simply

stated, it is computer software that is utilised in embedded systems. An embedded system has multiple definitions. For example, both Ganssle [17] and Oshana et al. [18] define it to be a combination of hardware and software, which performs a dedicated function and is usually part of a larger system. Ganssle adds that in some cases an embedded system might include additional mechanical or other parts and Oshana et al. add that it is constrained in its application. Emilio [16] defines it as a computer-controlled device that performs dedicated real-time control tasks and notes that they are cheaper than non-embedded devices. Laplante and Ovaska [19] define it to be a system that contains one or more computers or processors, which are vital to the system, but the system is not explicitly called a computer.

Embedded software is a major part of all software systems. Even though Manzoor [32] does not mention embedded software and Forward and Lethbridge [33] only define it as a subtype of the control-dominant software, embedded software still accounts for a significant part of software systems. In fact, according to Oshana [18] over 95 % of all software systems are considered to be embedded.

2.2 Test-driven development

Test-driven development (TDD) is a software design and development technique that aims to build software incrementally [4], [10], [35]–[38] by guiding the programming with the three-phased *red-green-refactor* mantra, as illustrated in Figure 2. During the *red* phase, a failing automated test is written. Then, in the *green* phase, code is programmed until the test passes. Finally, in the *refactor* phase, duplication is eliminated while ensuring that the test passes. Even though it is presented in this thesis as a technique to ensure that unit testing is applied in the software development, TDD is as more of a design than a testing technique [39]. In addition, TDD is related to the test-first programming concepts of the Extreme Programming (XP), which is software development methodology [40] that aims to improve software quality and responsiveness to the changing requirements.

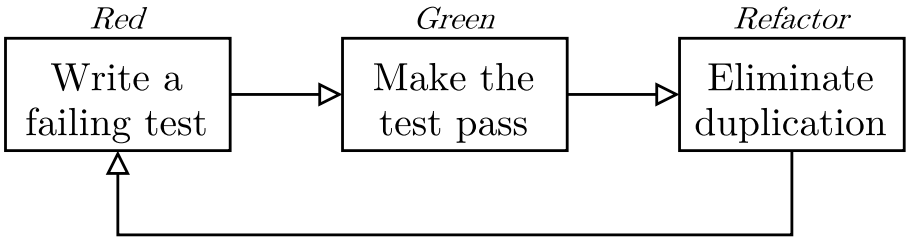


Figure 2: Red-green-refactor mantra of TDD

Even though TDD is claimed to have several benefits, which are presented in subsection 2.2.1, it is not widely utilised in the development of embedded software. Salo and Abrahamsson [15] studied the usage of XP in European embedded software organisations with a survey that included 35 individual projects in 13 industrial organisations. They found out that the organisations have been able to successfully

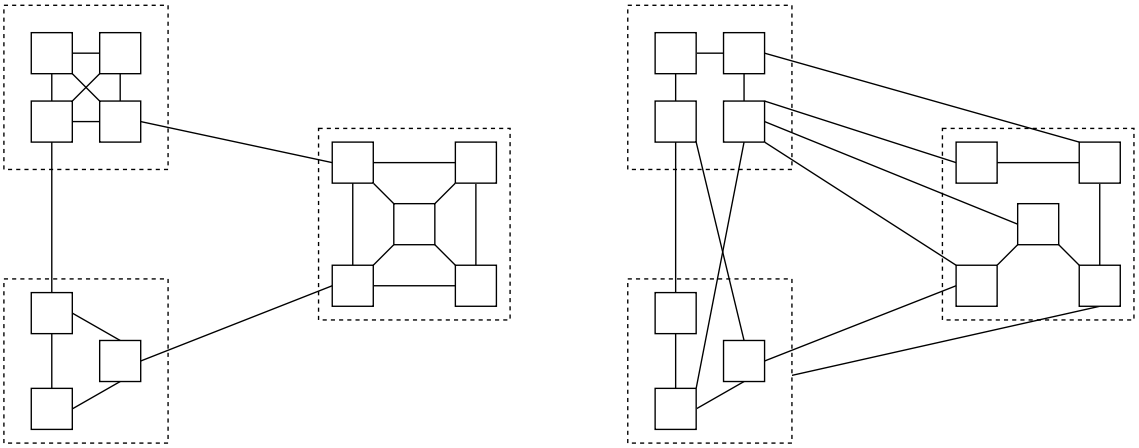
apply XP to embedded software. However, the study showed that TDD was the second least applied practice of XP as only 18 % of the respondents had applied it systematically or mostly and 56 % had never or only rarely applied it. In addition, it was also considered to be the second least useful practice of XP. On the other hand, they noted that it could not have been estimated whether the respondents had ever applied TDD. Since TDD is claimed to have multiple benefits but only few companies apply it, its cost efficiency is discussed in subsection 2.2.2. Finally, even though TDD is cost-efficient to apply to some extent, it is challenging, especially in the embedded domain. For example, Xie et al. [41] claim that developing embedded software is more challenging to develop than non-embedded software. The difficulties of applying TDD to the embedded domain are examined in subsection 2.2.3. Fortunately, TDD is feasible in the embedded domain by testing on host with the help of test doubles, which is discussed with additional suggestions for adoption TDD in subsection 2.2.4. Finally, the different types of test doubles are introduced in subsection 2.2.5.

2.2.1 Benefits

TDD has been claimed to have a number of benefits. Most importantly, it leads to fewer software bugs, according to Beck [35], Jeffries and Melnik [42], Grenning [14], and Munck and Madsen [13]. Beck [35] and Sommerville [37] also note that it assists programmers in understanding what a piece of code is responsible of. Hence, the software testing team can shift focus from reactive to proactive work. In addition, project managers can estimate the remaining tasks accurately enough so that they are able to involve real customers in daily development. Finally, software engineers can collaborate constantly and deliver software on daily basis. O'Regan [25] adds that TDD ensures that testability is emphasised during development, which simplifies the testing efforts.

According to Bosas [43], Shen and Yang [34], Canfora et al. [44] and Munck and Madsen [13], TDD also improves code quality. The quality is claimed to be improved because the design of the code must be modular. Modular code is code that has a high cohesion and low coupling. Cohesion in software is defined by Laplante and Ovaska [19] as connections inside a module, for example, connections between the methods in a class. By contrast, coupling in software denotes connections between modules, such as connections between classes. The difference is illustrated in Figure 3, in which dashed larger rectangles classes illustrate classes and smaller rectangles member functions of a class. Figure 3 (a) illustrates software architecture, which has high cohesion and low coupling whereas Figure 3 (b) illustrates an architecture, which has low cohesion and high coupling. By organising the architecture as in Figure 3 (a), modifications are only required to be executed in a single place if a module needs to be replaced, e.g. due to hardware change. If the architecture is organised as in Figure 3 (b), a change requires modifications to multiple locations.

One of the claimed benefits is also improved well-being of the developers, according to Grenning [10]. Developers have been reported to sleep better and have more free-time on weekends. It also rewards developers because after a test passes, the developer can be confident that the code performs its intended task. In addition,



(a) High cohesion and low coupling (b) Low cohesion and high coupling

Figure 3: An easily unit testable software design has a high cohesion and low coupling (a), whereas a design that has low cohesion and high coupling (b) is difficult to test.

TDD provides a progress monitor as tests show what has been developed and it can be used as a definition of done. Jeffries and Melnik [42] also claim that the programmers using TDD commonly report having reduced stress levels, lesser need for rework and better understanding of the requirements. In addition, developers working with legacy software often experience fear to change the working software as even a minor change could cause undesired side effects. TDD helps to reduce that fear as the tests created with it can be utilised to verify that new code has not broken existing functionalities. Hence, the possibility for unintended side effects can be reduced significantly.

In addition to reducing anxiety to modifying legacy code, Sommerville [37] and Shen and Yang [34] add that the costs of regression testing are also reduced with the utilisation of TDD. Regression testing is a type of software testing, which ensures that a recent modification has not broken existing features [45]. Grenning [14] agrees and states that manual testing is not practical to apply to each iteration of the software because it takes a considerable amount of time. However, dismissing regression testing leads to longer times between defect injection and detection making it more difficult to find a bug and fix it. With TDD, feedback is received rapidly, and problems can thus be found and fixed more easily and in a shorter time because bug reporting and repair overhead including debugging can be completely avoided. As an additional benefit, the tests will act as a system documentation by itself and they ease the understanding of the code.

Grenning [14] also adds that there are additional benefits when TDD is applied to the embedded domain. It allows verifying software correctness without hardware, which is beneficial when hardware does not exist yet or it is too expensive for each developer to possess. TDD also reduces the time spent on compiling, linking and uploading software for target as well as debugging by detecting faults on the development environment. Lastly, it isolates interaction issues between hardware and software.

2.2.2 Cost efficiency

Previous studies have evaluated the cost efficiency of TDD for both, non-embedded and embedded software. For example, Sanchez et al. [39] studied the practise of TDD for Java programming at IBM and conducted a survey for the employees to examine how TDD affects code quality and the impact on productivity. All respondents indicated that test creation made it easier to produce better quality code. As for the productivity, the extra time spent for writing tests was reported to be anything from none to more than 25 %. They noted that after starting to use TDD the quality of their work was improved, and the defect densities were below the standards of the industry. Also, the data gathered showed promising results of keeping the code complexity at a constant level even though the software had aged. However, some developers indicated that the productivity might decrease, but they noted that the decrease in productivity might be compensated with better product quality. In addition, they stated that the code written turned out to be more testable and developers and testers indicated that finding errors was easier with unit tests in place. Also, four of the seven testers responded that they felt that the quality had improved, two had no prior experience and one felt that the quality was the same as with other projects. As an additional observation, it was emphasised that a fix in the project in question did not break other functionalities, which was common in other projects, in which TDD was not practised. Many also responded that the utilisation of TDD eased over time and one even considered that TDD should be mandatory.

Nachiappan et al. [46] conducted four case studies to gather empirical evidence on TDD practise in an industrial context. The pre-release defects decreased between 40 % to 90 % in comparison to similar projects that did not practise TDD. The initial development time increased between 15 % to 35 % after the adoption of TDD.

Schooenderwoert and Morsicato [12] evaluated the suitability of XP for embedded software development based on a three-year long case project. They had very low bug rate, only around 50 bugs were noticed in integration or later testing levels. Unfortunately, they did not evaluate the cost-efficiency of the technique, but the minor number of bugs allowed them to spend almost all the development time to adding value instead of debugging. Even though the usage of a test-driven approach might have increased the initial development costs, the fact that they were able to spend close to 100 % of time on development is unheard of, as debugging often consumes up to half of all resources of a software project, as stated in subsection 2.1.3.

George and Williams [47] studied whether the usage of TDD leads to superior quality in the industry and whether the code is developed faster than programming without tests. The programming language in question was Java and they tested their hypotheses in three different companies with 24 developers. The developers were divided in pairs and half of the pairs were instructed to develop a bowling game by using TDD and half of them to control group, which used traditional method. It was concluded that TDD lead to superior external code quality, but it took 16 % more time for the developers. Nevertheless, the control group primarily did not write test cases even though they were instructed to do so, which made the comparison uneven. Also, a survey was conducted to the same developers, which resulted in the conclusions

that TDD led to simpler design and that the lack of initial design did not slow the process down. However, the transition process to TDD mindset was considered to be difficult by every other developer. TDD was also an effective approach according to 80 % of the developers and 78 % claimed it to improve productivity. The results were mentioned to be statistically significant.

Bissi et al. [48] conducted a literature review to analyse the effects of TDD on internal and external quality as well as on productivity. They analysed 27 papers ranging from 1999 to 2014 regarding TDD on both embedded and non-embedded domains, which consisted of academic and industrial environments. They reported 76 % of the studies to have noted an increase in the internal quality and 88 % of the studies to have identified an increase in the external quality. The productivity had increased in academic environments but decreased in industrial environments. Their final conclusion was that TDD increases both internal and external quality with the cost of lower developer productivity.

2.2.3 Difficulties

Shen and Yang [34] collected and compared previous studies on applying agile methods to the embedded software development, in which TDD was also evaluated. They identified five difficulties in adopting TDD for all types of software. There is often pressure of the schedule, developers have negative first impressions about TDD and the mindset for testing first is difficult to adopt. In addition, TDD is a skill that takes long to master and the development environments often do not meet the requirements of TDD, such as rapid execution of the tests. They also listed six characteristics why agile methods, including TDD, are difficult to apply to the embedded domain. Embedded software developers must possess not only software development skills, but also skills related to hardware development. There is also fierce competition in the industry, which creates competitive pressures. As for the hardware, it is often not available until late in the project and limited resources of embedded systems also create challenges related to the limited memory space, processing power and execution time. In addition, hardware changes are frequent, and they also have an impact to the software. Finally, embedded software often has additional performance requirements, such as real-time constraints and safety issues.

Xie et al. [41] analysed 52 reports to evaluate the usage of agile methods in embedded software development. They found out multiple characteristics of to be repeatedly mentioned to hinder the development, which overlap more or less with the difficulties that Shen and Yang [34] identified. Xie et al. noted that hardware dependency complicates development, especially since the hardware is often available only late in the development. Embedded development also requires the developer to possess a wide range of knowledge and the development tools for the embedded domain are limited. In addition, embedded software has often tight resource constraints, such as memory and battery capacity as well as response times. The development environment is also more dynamic than in the non-embedded domain, because requirements may change not only because of a customer but also due to changing hardware.

Ronkainen and Abrahamsson [36] explored the suitability of agile software development methods, including TDD, for the embedded domain. They gathered experiences for over a year from the development of a low-level telecommunications software. Based on the findings, they claim that there are multiple issues in the usage of TDD in embedded software development. For example, the test environment is different from the development environment and memory and performance constraints prevent the installation and execution of all tests simultaneously. Also, simulation tools may be shared with hardware developers preventing daily testing.

Srinivasan et al. [49] studied the usage of agile methods for embedded systems development and found out that the greatest challenge for an organisation attempting to apply TDD is to find the correct tools and techniques without reducing the efficiency of current development work.

2.2.4 Recommendations

Testing can be performed either on host or on target. Grenning [10] stresses the importance of testing on both and explains them. Testing on host is an activity, in which tests are run on the development environment, more specifically, on the same computer and operating system on which the code is programmed. Testing on target is testing in which the developed software is uploaded to a particular embedded hardware, which then executes the tests. The results of the tests are then transferred back to the host environment, in which the results are finally inspected. Boydens et al. [20] compared the two and found the following results. The advantages of testing on host were that existing tools can be utilised, and the feedback was rapid. The testing environment was also easy to set up. The disadvantages were that there was no guarantee that the mocked hardware calls functioned as intended as the code was compiled for the development environment. Hence, it was not certain that the code was run equivalently on the target hardware, which could have led to cross compiling issues. Advantages of testing on target were that the test results did not include the same uncertainties as the real compiler was be utilised. Cross compiling issues were also not present. Disadvantages consisted of considerably longer testing cycles and a significantly more difficult setup. The complete automation of testing on target was also considered to be more difficult than testing on host since it required the tests to be transferred to the target, starting the target and retrieving the results in addition to executing and inspecting them. Hence, the utilisation of a continuous integration (CI) server would have been significantly more difficult compared to testing on host, in which tests had to be only executed and inspected. More importantly, testing on host allowed the tests to be executed virtually on any environment, whereas testing on target would have required the target hardware to be somehow connected to the CI server. They also noted that integrated development environments (IDE) did not exist for embedded development and the tests in general required special care as their memory footprint were not able exceed the memory available on the target.

According to Grenning [10], the memory and performance limitations of embedded software can mostly be overcome by running tests on host, because the resources of a development machine are usually multiple magnitudes greater than the resources

of a target device. However, he notes that in order to allow testing on target, the hardware dependencies have to be removed, which can be done with the help of mocking. Mocking is an activity in which a real implementation is replaced with a test double, which is an object or component that replaces a real one for testing purposes. Multiple types of test doubles exist, and they are presented in the next subsection 2.2.5. Mocking can also reduce the execution time of tests [50] and can be required to be able to test scenarios that are difficult to reproduce. In addition, since mocking allows testing on host, target hardware is not required for testing, which allows all developers to perform testing even if the target devices are scarce or do not even exist yet.

Grenning [10] lists multiple techniques to overcome memory limitations if testing is desired to be performed on target, which could be beneficial to overcome the cross compilation issues mentioned by Boydens et al. [20]. Firstly, one can try to find a small enough test harness or modify the target system for testing purposes to have plenty of memory to be able to hold all the tests. Tests can also be run in subsets, which allows running significant amount of tests using little memory. Also, naturally by paying attention to the memory consumed by the tests can be enough to fit tests into the build.

As for ensuring that tests add as little overhead as possible, Bosas [43] underlines the importance of automated tests. He argues that manual testing is expensive and that automated tools can transform labour-intensive testing to be performed in seconds. Meszaros [51] and Beck [35] also emphasise that tests should be able to be executed rapidly. That is because developers practising TDD often run the at least a subset of them every few minutes. Meszaros also states that running tests often should be encouraged as it speeds up the detection of software bugs, which in turn usually reduces the costs of fixing them. In addition, if tests require manual intervention, developers tend to run them less frequently, which often increases the costs of fixing the bugs. Beck [35] also states that tests should be isolated as they must not affect one another. That is because if one test breaks early, the system might be in an unpredictable state for the next test. In addition, isolated tests are independent of the order, which allow only a subset of them to be run and encourages to write modular code. Langr [38] also emphasises that tests should be easy for the readers to understand. One technique to ensure that tests are simple is to use a mnemonic *arrange-act-assert* (AAA). *Arrange* stands for setting up proper conditions. *Act* is for executing the part of code to be tested. Finally, *assert* is for verifying that the expectations have been met.

Additional suggestions for teams considering the transition to TDD are provided by Sanchez et al. [39]. Firstly, there should be one person who has the passion to spend extra time on learning the practise and the required tools to be able to convince and help all team members. It should also be understood that not everything can be automated, for example, one might have to visually inspect a physical movement of a product. Nevertheless, everything should be tried to be automated because execution of manual test is not as frequent as for the automated ones. Thirdly, measurable objectives should be set, for example, desired code coverage could be 80 %. Additionally, all tests should be run prior to uploading to a master code

base and there should be a build process running all tests once a day and a person responsible for the process, such as a CI server. Finally, a test should also be written for every detected bug and tests should be refactored to minimise their duplication. In addition, Shen and Yang [34] state that developers should be convinced that TDD is useful to try and it should be made simple for developers to start with. In addition, training should be provided and TDD should be adopted already at the start of a process.

2.2.5 Test doubles

Test doubles can be divided into five types: dummy objects, test stubs, test spies, mock objects, and fake objects [51]. One noteworthy point to state is that according to Langr [38], test doubles are often called mocks. Since developers mostly utilise mock objects, the minor difference in terminology is not harmful. For the purposes of this thesis, the term *mocking* is used to describe the utilisation any of the test doubles and their functionalities.

Dummy object is a placeholder object that is never utilised by the SUT but must nevertheless be provided. A dummy object removes the need from constructing a real object when an object must be provided, but it is not to be used by the SUT. Often a null object is sufficient; however, sometimes a real object must be defined. In this regard, care must be taken that the type matches to the expected type. In addition, dummy objects and null objects are not synonyms. Dummy objects are designed not to be utilised, whereas null objects are utilised but designed to do nothing. The difference here is subtle yet an important distinction.

Test stub replaces a real object to allow the SUT to be controlled. The return values of test stub methods can be defined as desired for a particular test, which allows the SUT to be forced down to paths that would otherwise be impossible to be covered in tests. To better evaluate the potential results, two types of test stubs have been developed – responders and saboteurs. A responder is used to test the functionality under normal conditions by returning the desired data from its methods. A saboteur is used to test the functionality under abnormal situations by raising exceptions or errors.

Test spy expands the test stub functionalities to allow the test to verify indirect outputs after executing the SUT. It has the same properties as a test stub but adds the ability to save the calls performed to its methods. The saved calls can then be examined to verify proper execution in a test.

Mock object expands the functionalities of a test stub or spy even further. In addition to controlling the return values and saving function calls, it compares whether the calls performed to it match predefined expectations. For example, the expected parameters and the amount of expected function calls can be defined. Mock objects can be defined strictly to verify that they have been called in the same order as expected or leniently to tolerate varying orders of calls or even missed calls. A leniently defined mock could only verify that the calls have one or more correct arguments.

Fake object differs significantly from the other types. It is not controlled nor

observed by the test. Instead, it replaces the functionality of a real object by providing a sub-implementation or the similar implementation in a simpler way. Fake objects are often utilised when the real dependency is too slow or not available for testing purposes.

Langr [38] sums up the types of test doubles: a dummy object is not used by the software; a test stub returns hard-coded values; a test spy saves the information for later inspection; a mock object verifies itself based on expectations set upon it; and a fake object provides a sub-implementation of the real implementation.

3 Implementation

Since TDD is a technique that could potentially reduce the increasing costs of software failures as presented in section 2.2, it should be able to be adopted to the embedded domain. However, as explained in subsection 2.2.3, embedded software has challenges such as hardware dependencies as well as memory and performance restrictions, which make the adoption challenging. Fortunately, the challenges can be overcome by testing on host, which can be achieved with the help of mocking as introduced in subsection 2.2.4. In order to simplify testing and to allow the creation of automated test cases, testing tools can be utilised. Hence, the section 3.1 presents unit testing frameworks for C++. Even though unit testing frameworks for C could be utilised to some extent also with C++, they are omitted as this thesis was chosen to focus on C++, because C++ has recently gained more popularity in the embedded domain [19]. As for mocking, multiple techniques exist to overcome the hardware dependencies to allow testing on host, which are introduced in section 3.2.

One of the presented testing tools, Google Test, and all of the introduced mocking techniques were tested in a case project at Etteplan as consulting work. The customer had developed their own hardware and gave requirements for the functionalities of the software. The project was suitable for evaluating Google Test and the methods presented as it required the usage of the STM32 library functions, which lead to the hardware dependency issues state earlier. In the project, a motor controller was implemented, however the exact details of the project are left out of this thesis to preserve trade secrets. The setup of the Google Test for testing on target is shown in section 3.3. In addition, testing on target was also experimented to allow the comparison between testing on target and testing on host with each of the mocking techniques. Testing on target required Google Test to be modified to run on the target hardware. Hence, the required changes to Google Test and the setup of the communication between the target hardware and the development environment are demonstrated in section 3.4.

3.1 Tools

Six unit testing frameworks for C++ were identified. They were UnitTest++, CppUTest, Google Test, Boost.Test, Catch2 and CxxTest. Boydens et al. [20] mention UnitTest++, CppUTest and Google Test to be suitable for embedded development and all of them to function multiple operating systems, such as on Linux, Windows or Mac. In addition, Boost.Test is claimed to function on multiple operating systems and since Catch2 and CxxTest are supplied as headers, they should also perform on multiple platforms as well.

UnitTest++ [52] consists mostly of the standard C++ and makes minimal use of advanced library and language features. Hence, it should be easily portable for embedded development. It is licensed under MIT license, thus allowing both private and commercial usage. However, it is a one-man project and the latest version (2.0.0) of it has been released in 2017. Hence, future support is not to be expected. Also, according to Boydens et al. [20] it requires some modifications to adapt the

framework to specific needs. It is also noteworthy to mention that another developer has continued the work and the new project [53] seems to be active but with only a single active developer the framework might suddenly lose support.

Google Test (gtest) [54] is a C++ testing framework, which can also be used for testing C code. It is based on xUnit architecture and developed by Google. The term xUnit refers to test frameworks that automate the execution of tests created by programmers [51]. Google Test contains also a mocking framework called Google Mock (gmock), which can be used with or without Google Test itself. It is licensed under BSD 3-Clause, which means that its usage is permitted both in private and commercial use. According to Cordemans et al. [22], Google Test is the most complete C++ unit testing framework but the downside of it is that it is not directed towards embedded software.

CppUTest [55] is a unit testing framework for C and C++. It also based on xUnit architecture and written in C++. It is continuation for the CppUnit [56] C++ testing framework. According to the home page of CppUTest, its usage in embedded systems is frequent and it is claimed to work for any C or C++ project and to be simple to use. It also includes a mocking framework, called CppUMock. It is licensed under the same BSD 3-Clause as Google Test. The framework is praised by Grennings [10]. However, he is one of the authors of the framework there can be said to be a conflict of interest as he is one of the developers of CppUTest.

Boost.Test [57] is a C++ testing library and part of peer-reviewed and portable Boost C++ libraries. It is licensed under BSL-1.0, which permits private and commercial usage. It is actively developed by the Boost community.

Catch2 [58] a multi-paradigm unit testing framework for C++ and Objective-C. It is mostly distributed as a header file but some of the features may require additional headers to be included. It has the same BSL-1.0 license as Boost.

CxxTest [59] is a simple C++ testing framework. It is utilised simply by including it as a header. In addition, CxxTest seems promising for embedded development as it does not require advanced features of C++, which could make it difficult to compile and execute on an embedded processor. An example of such advanced features is run-time type information (RTTI). CxxTest is notably different compared to the other frameworks as it utilises a Python script to create the test executable from header files. It is licensed under LGPL license.

In conclusion, all of the presented testing frameworks could be utilised for creating unit tests for C++. They all seem to work on multiple operating systems and they are all free to use even in commercial projects. However, only CppUTest and Google Test include a mocking framework, which is essential in allowing testing on host.

3.2 Methods

Cordemans et al. [22] have presented four techniques for mocking: macro preprocessed-, link-time-, interface- and inheritance-based mocking. Macro preprocessed- and link-time-based mocking techniques can be utilised for both, object-oriented and procedural code. Interface- and inheritance-based mocking, on the other hand, are object-oriented mocking techniques and cannot be used with procedural code.

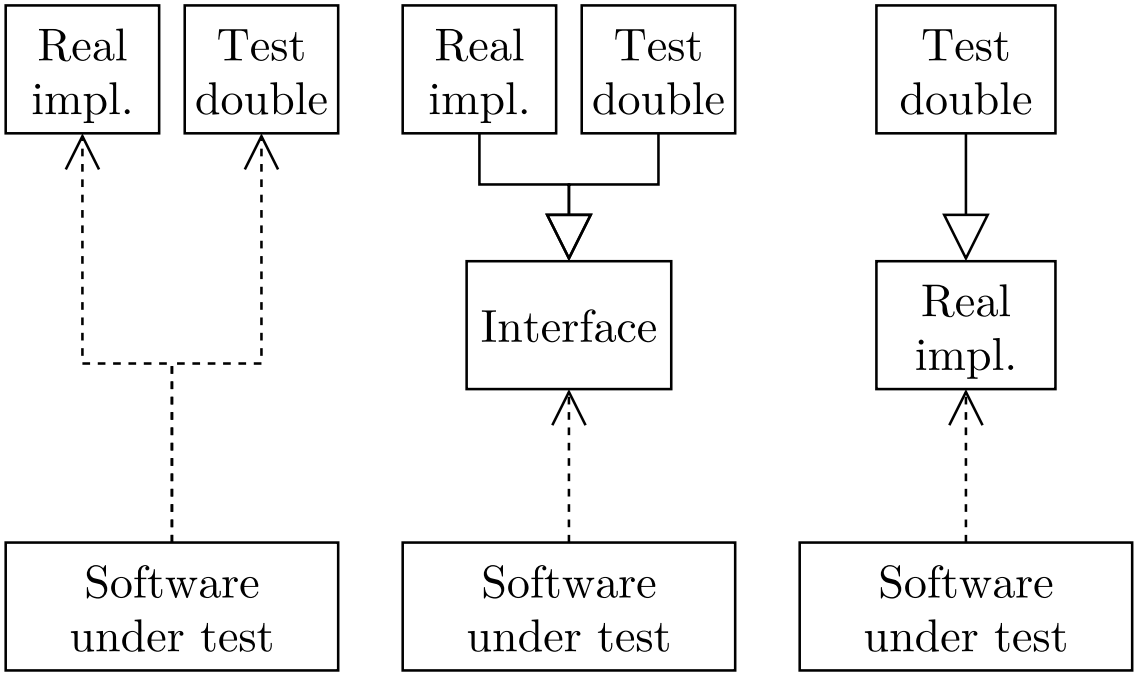
The macro preprocessed-based mock replacement technique utilises conditional compilation to choose a real implementation for production and a test double for testing. Conditional compilation [60] is a process, which allows for compilation of only the desired lines of the source code. Conditional compilation is achieved by using the conditional compilation directives, *#ifdef*, *#ifndef*, *#else*, and *#endif*, for the preprocessor. A preprocessor modifies the source code according to the directives before compilation. Firstly, the *#ifdef* directive determines whether the macro definition provided after exists. If the macro definition does not exist, the following lines of code after the directive are excluded from the compilation. Secondly, the *#ifndef* directive functions reversely. If the macro definition provided for it exists, the following lines of code are excluded from the compilation. Thirdly, the *#else* directive functions similarly to the else statement in the C++ language by reversing the effect of the previous conditional compilation directive. Fourthly, the inclusion or exclusion of lines of code after any of the directives ends when the *#endif* directive has been reached. Finally, the choice of the desired implementation can be achieved by placing a real implementation into one file and a test double into another file. Then, only the desired implementation can be included by utilising the directives based on whether the argument has been defined. The argument can be defined by providing it as a compiler flag.

The link-time-based mock replacement technique utilises a linking script or integrated development environment (IDE) to choose the desired implementation of a class during linking. Similar to macro preprocessed-based mock replacement, link-time-based mock replacement requires the implementations to be in different files. The files are then placed into different folders. In addition, the common code under test is placed into a third folder. During a test, the linker can then choose to link the common code under test with either real implementations or test doubles. The choice is achieved by providing only the desired folder for the linker. The linker also ensures the existence of the files and implementations and warns if they are missing. The macro preprocessed and link-time mock replacement techniques are illustrated in Figure 4 (a).

The interface-based mock replacement technique utilises abstract classes. An abstract class in the C++ is a class that has at least one pure virtual function. A pure virtual function is a function that has been declared virtual and does not have a definition. The definitions are provided by the real implementation and test double, which must derive from the same abstract class. That allows the desired implementation to be chosen during run-time. [60] The desired implementation can then be used simply by declaring it as any other object because any calls to it are directed through the common interface. Inheritance-based mock replacement also allows using both implementations simultaneously because both are compiled and linked to the executable. In comparison to the previous mock replacement techniques, the interface-based mock replacement does not require to be controlled during compilation or linking. The technique is visualised in Figure 4 (b)

The inheritance-based mock replacement technique is based on creating test doubles with inheritance and method overriding. Test doubles inherit from real classes and override the methods that are desired to be tested. As with the interface-based

mock replacement technique, the inheritance-based technique allows the desired implementation to be chosen during run-time. In order to allow for the desired methods to be overridden, the corresponding methods in a real class must be declared virtual. While the inheritance-based mock replacement technique enables the usage of both implementations, Cordemans et al. [22] criticise it. According to them, it does not lead to testable design but rather enables it. The technique is visualised in Figure 4 (c)



(a) Macro preprocessed- and link-time-based mocking techniques

(b) Interface-based mocking technique

(c) Inheritance-based mocking technique

Figure 4: UML class diagram examples of the different mocking techniques. The first two techniques (a) rely on choosing the correct implementation during compilation or linking. The third technique (b) is based on a common interface class. Finally, the fourth technique (c) utilises inheritance and method overriding.

Even though all the presented techniques are called replacement techniques, only the macro preprocessed- and link-time-based techniques replace a real implementation with a test double. The interface- and inheritance-based techniques do not replace a real implementation but enables a test double to be used in place of a real implementation. Therefore, the interface- and inheritance-based techniques require the desired implementation to be provided. In order to provide the desired implementation using the interface- and inheritance-based techniques, dependency injection must be used. Dependency injection is a technique in which an object delivers the dependencies to another object. In practise, the dependencies are provided for classes instead of classes creating dependencies for themselves. Dependency injection consists of at least three types of techniques: parameter, constructor and

setter injection. The names imply how the types are used. The parameter injection makes use of method arguments to provide the dependency. [51] In the constructor injection the dependencies are supplied in the constructor of a class. The setter injection utilises public methods to require the user to provide the dependencies. [22]

3.3 Testing on host

Almost all functionalities implemented in the case project were tested, but only the hardware calls that utilised the STM32 library functions are covered in this thesis, since they were the functionalities that required the usage of test doubles to allow testing on host. As for the unit testing framework, it was chosen to be gtest due to the client request. The version of it was chosen to be the newest version 1.10.0 to ensure that all the documented functionalities were available. Unfortunately, the documentation provided little information for setting up gtest. Hence, secondary sources of information, such as discussions on the Stack Overflow website [61], had to be relied on in the setup. In addition, most of the instructions found were written for Linux environments and thus, as explained in subsection 3.3.2, it was difficult to find out that Windows did not support POSIX threads (pthreads).

The client was using a Windows-based development environment, which required that the tools had to support Windows-based operating systems (OS). Hence, a 64-bit Windows 10 (version 1903) was used as an OS as it was the Windows version provided by Etteplan. However, most of the instructions for the chosen tools were written for Linux-based OSs. Thus, the tools were first tried on Linux before installing them to Windows. The Linux was run in a virtual machine on top of the chosen Windows version. The virtual machine was chosen to be Oracle VM VirtualBox [62], which is a free and open-source hypervisor software. VirtualBox was chosen as it was familiar to the author and proven to be suitable for virtualisation. The Linux version was chosen to be a 64-bit Ubuntu. The version 18.04.03 of Ubuntu was selected because it was the latest version available that had long term support available. The installation of gtest required multiple tools, which are introduced in subsection 3.3.1. The installation process of it is shown in detail for Windows and Linux in subsection 3.3.2.

3.3.1 Installation tools

The setup of gtest required the usage of multiple tools, which consisted of CMake, g++ and Make. CMake is a cross-platform and open-source software tool for maintaining the compilation of software compiler-independently [63]. G++ [64] is a compiler and part of the GNU compiler collection (GCC). Make [65] is a tool to manage the process of compiling programs from source files. There are multiple versions of Make available, of which GNU Make was chosen for this thesis as it was familiar to the author. Both, g++ and GNU Make have been licensed under third version of the GNU General Public License (GPL) and can be used freely.

In order to use CMake, g++ and Make tools on Windows, either Cygwin or MinGW was required to be utilised. Both are command line tools, which allow the

execution of a subset of Linux programs on Windows. Cygwin [66] aims to mimic Linux functionalities by enabling the usage of multiple GNU and open source tools on Windows that are native to Linux. MinGW (minimalist GNU for Windows) [67] on the other hand, is simply a port of GNU compiler tools, which provides a development environment for Windows. The most notable difference of the two is that MinGW does not have an emulation layer and Cygwin does. Hence, MinGW applications are targeted for Windows and Cygwin applications for Linux. Both are free software as they are licensed under the same GPL license as g++ and GNU Make. Cygwin and MinGW were both utilised. Cygwin was installed with CMake and g++ packages. The installer of Cygwin enabled the installation of the packages simply by checking them to be installed within the installation process. MinGW was installed similarly as it also allowed the installation of desired packages simply by marking them for installation while installing MinGW itself.

3.3.2 Test framework setup

Gtest was required to be built from its source code as the precompiled libraries were not provided within the gtest framework. Thus, the libraries were compiled for Windows and Linux separately. Fortunately, gtest was delivered with CMake files, which allowed the compilation of the libraries on different platforms.

On Windows, two modifications were required. Firstly, since Windows did not support pthreads, they had to be disabled to allow proper execution of gtest. It was achieved by changing the default argument OFF to ON for the *gtest_disable_pthreads* option in the *CMakeLists.txt* file located at *googletest/googletest* subfolder. The modification is illustrated in Listing 1, where the default setting is shown on the line 1 and the fixed setting on the line 2.

Listing 1: The *pthreads* option had to be disabled on Windows

```
1 option(gtest_disable_pthreads "Disable uses of pthreads in gtest." OFF)
2 option(gtest_disable_pthreads "Disable uses of pthreads in gtest." ON)
```

Secondly, CMake generator was switched to generate the correct Makefiles in the corresponding terminal. For Cygwin the generator was selected to Unix Makefiles by running the following command:

```
$ cmake -G "Unix Makefiles"
```

For MinGW the generator was selected to MSYS Makefiles by running the following command:

```
$ cmake -G "MSYS Makefiles"
```

The libraries were then able to be built on Windows. On Linux the previous steps were not needed. Building the libraries started by first cloning gtest from its Git repository [54]. The cloning was performed in the corresponding terminal, Cygwin, MinGW or Linux. Then, Makefiles were generated with CMake by running the following command:

```
$ cmake .
```

Finally, static libraries of gtest were compiled with g++ by running the following command:

```
$ make
```

The result consisted of four static library files: *libgtest.a*, *libgtest_main.a*, *libgmock.a* and *libgmock_main.a*. They were created into a new *googletest/lib* folder by the *make* command. The *libgtest.a* library consisted of the functionalities of gtest and the *libgmock.a* library of the functionalities of gmock. The *libgtest_main.a* and *libgmock_main.a* libaries provided implementations for main() function. However, the second one, *libgmock_main.a*, was left out as it provided the same functionalities as the *libgtest_main.a*. It would only have been required if gmock would have been utilised with some other unit testing framework than gtest.

The framework was integrated into an IDE. However, considering the multitude of varying IDEs, general build commands for compiling and linking and the used file structure are presented instead of explanation of the integration to the IDE. With the general build commands and file structure, the framework can be utilised regardless of the environment choice. The file structure consisted of three subfolders within the *tests* folder: *include*, *libraries* and *source*. The header files of gtest were located in the *gtest* folder under the *googletest/googletest/include* folder and header files of gmock in the *gmock* folder under the *googletest/googlemock/include* folder. Both folders were copied to the *include* folder. Previously built static libraries, excluding *libgmock_main.a*, were copied from the *googletest/lib* folder to the *libraries* folder. Tests were placed into the *source* folder. The folder structure and their contents are shown in Figure 5.

The tests for each class were placed into their respective compile units (.cpp files). Here, a single, the *tests.cpp* file is shown with the skeleton required to implement a test. The file is shown in Listing 2, where gtest and gmock were included on the lines 1 and 2 and a test without any conditions was defined on the lines 4 to 6. Even though the contents of the test were left empty, gtest still generated a functional executable, which allowed to test that gtest was correctly set up. The test was defined using the *TEST* macro, followed by arguments for the names of the desired test suite and test name. For the test shown, inclusion of gmock headers was not required, but it was included in the example as it was required for other tests that used the mocking functionalities of gmock.

Listing 2: tests.cpp - Test skeleton and includes required for an executable test

```
1 #include <gtest/gtest.h>
2 #include <gmock/gmock.h>
3
4 TEST(TestSuiteName, TestName) {
5
6 }
```

In order to build and run the tests, the compiler had to be able to find the gtest and gmock header files. Thus, the *include* folder had to be added to the list of folders to be searched from. The addition was achieved by providing the *include* folder for the compiler with the *-I* option. In addition, the linker had to be aware of the *gtest*, *gtest_main* and *gmock* libraries and they were supplied using the *-l* option. The inclusion of the *gtest_main* library was not necessary. However, if it had been left out, a function main would have been required to be defined in the tests. An example of a function main that could have been utilised instead of including the library *gtest_main* is shown in Listing 3. The *libraries* directory was supplied with

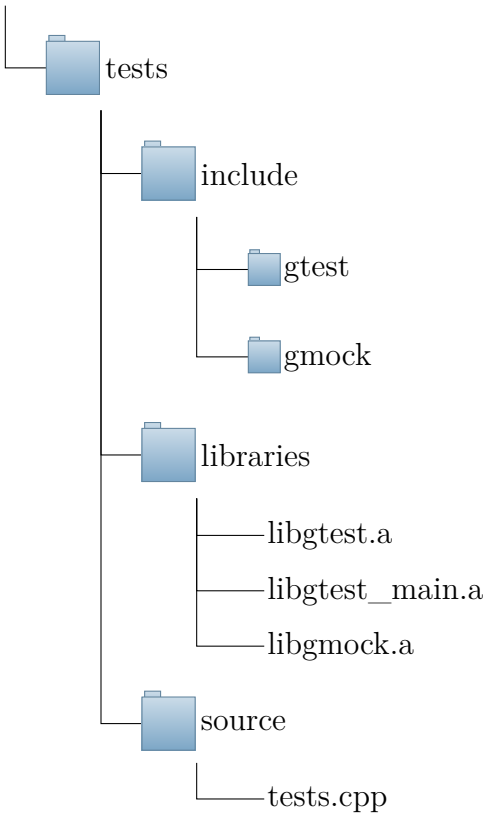


Figure 5: Folder structure for using gtest

the *-L* option for the linker. Lastly, on Linux, the linker needed the *-pthread* option to be included as gtest supported multithreading on Linux.

The commands for building the example test on Linux are shown in Listing 4. In the test, the object file was generated on the line 1 and the object file was linked to the executable named *tests* with the *-o* option on the line 2. The build commands were executed in the *tests* folder. The same commands were also utilised on Windows, excluding the *-pthread* option. Finally, the *tests* were able to be executed. The output of the execution is shown in Listing 5.

Listing 3: Main function to be included if the *gmock_main* library is not included for the linker

```
1 int main(int argc, char **argv) {
2     ::testing::InitGoogleTest(&argc, argv);
3     return RUN_ALL_TESTS();
4 }
```

Listing 4: Example commands required to build unit tests using gtest

```
$ g++ -c source/tests.cpp -Iinclude
$ g++ tests.o -o tests -Llibraries -lgtest -lgtest_main
   -lgmock -pthread
```

Listing 5: Example gtest output

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from TestSuiteName
[ RUN      ] TestSuiteName.TestName
[       OK ] TestSuiteName.TestName (0 ms)
[----------] 1 test from TestSuiteName (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

3.4 Testing on target

Running tests on target device was not utilised in the case project because it was not desired by the customer. Nevertheless, the tests concerning the hardware calls with the STM32 library functions were experimented to allow the comparison of running

tests on host versus on target. The tests were run on a common Nucleo-F767ZI [68] development board, as shown in Figure 6, and the reasoning for the chosen development board is explained in the next subsection 3.4.1. The communication between the host device and the development board was achieved with serial communication. It was selected since Karlesky et al. [69] had reported success in customising Unity, a unit testing framework for C, by printing results through serial port. The tools and setup required by the serial communication are explained in subsection 3.4.2. Finally, the installation of gtest and the changes required to it are shown in detail in subsection 3.4.3.

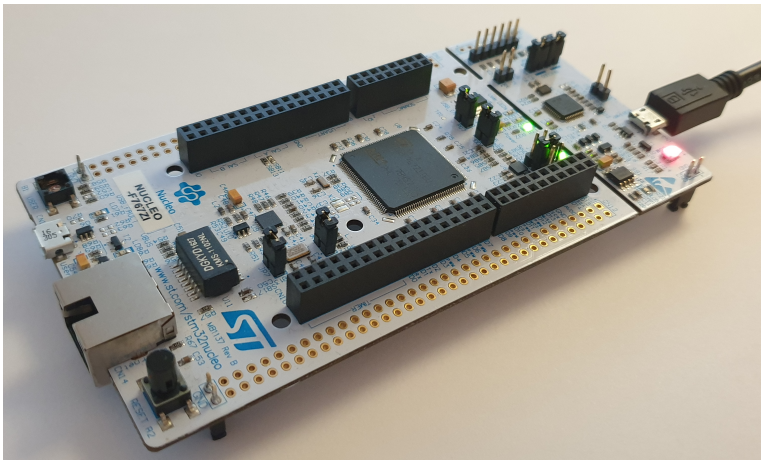


Figure 6: Nucleo-F767ZI development board was chosen for the demonstration of running tests on target. The microcontroller on the board was able to be programmed and powered simply by connecting the board to a computer with a USB-cable

3.4.1 Target device

Since the tests were not run on the case project, the target device was able to be chosen freely. It was chosen to be a common Nucleo-F767ZI [68] development board. due to five main reasons. Firstly, the board was built around the same microcontroller family as the one utilised in the case project and it was a potential candidate for other products as well. That is because the development board included the STM32F767ZI microcontroller, which is a 32-bit microcontroller based on the ARM Cortex-M7 processor. ARM processors have been widely popular among embedded systems as according to Langbridge [70], 75 % of 32-bit embedded systems and 90 % of embedded reduced instruction set computer (RISC) systems had utilised ARM processors in the early 2020s. Cortex-M processor product line was also designed for the most energy-efficient embedded devices [71]. Additionally, at the time of writing this thesis, M7 processors were the highest performance Cortex-M processors of the Cortex-M product line [72].

Secondly, the board was simple to program because it included an integrated ST-LINK/V2 [73] debugger and programmer. It meant that an external programmer was not required to program the processor on the board. Instead, the board was able

to be programmed simply by connection it to a computer with a universal serial bus (USB) cable. The same USB-cable provided also power for the board. Therefore, the demonstration only required a USB-cable and the board as hardware.

Thirdly, the board was easy to expand. It included the ST Zio connector, which is an extended version of the Arduino Uno V3 connector. Arduino Uno is an exceptionally popular development board based on the ATmega328 processor. Due to the vast popularity of Arduino Uno development boards, there is a multitude of extension boards available for the Arduino Uno. Since the Nucleo-F767ZI included an Arduino Uno-based connector, the extension boards manufactured for Arduino Unos could also be utilised with the Nucleo-F767ZI.

Additionally, the target board was able to be configured with STM32CubeIDE, which is a C and C++ development platform and based on a popular IDE called Eclipse. In addition to the integrated toolchain, it allows automatic code generation for peripheral initialisation, and it was utilised to initialise, for example, UART and Leds.

Finally, the development board was affordable, costing approximately 20 euros at the time of writing this thesis. The affordability ensured that people desiring to repeat the steps shown in this thesis could easily acquire the board. In addition, Etteplan, for which this thesis was made, also possessed a great number of these boards. Hence, the board was also a desirable choice for Etteplan as its employees were easily able to follow the guideline shown in this thesis.

3.4.2 Serial communication

PuTTY [74] was utilised to view the test results on a serial port. It is a tool, which emulate terminal and act as a serial console or network file transfer application. PuTTY supports many network protocols and it can also connect to a serial port. It is a free and open-source tool.

On Windows, PuTTY did not need to be installed because an executable file (.exe) was provided that executed simply without an installation. On Linux, however, the installation of PuTTY was required. Prior to the installation, the package list of Linux was first updated by running the following command:

```
$ sudo apt-get update
```

Updating the package list ensured that the latest version of PuTTY was able to be installed. Then, PuTTY was installed simply by running the following command:

```
$ sudo apt-get install -y putty
```

Finally, PuTTY was ready to be utilised on Linux. However, on Linux PuTTY required root privileges. Otherwise it would not have opened the serial connection.

Hence, PuTTY was opened from the command line with root privileges by running the following command:

```
$ sudo putty
```

Then, the correct serial port had to be retrieved. On Windows the serial port appeared as a *COM* port, followed with a number. In this case the port was *COM1*. The serial port was retrieved with the Device Manager, as shown in Figure 7. Device Manager was a tool to view and control hardware attached to a Windows computer. It was part of the Windows OS and does not require a separate installation. In order to monitor the serial port on Linux, the USB had to be enabled in the VirtualBox, as shown in Figure 8. Then, the serial port was able to be retrieved on Linux by running the following command:

```
$ dmesg | grep tty*
```

The command printed multiple lines of text, from which it was determined that the port was *ttyACM0*. However, on Linux the ports can be seen as paths to a folder or an executable and they are located in the *dev* folder. Hence, the full name of the port was */dev/ttyACM0*. The output of the previous command was the following:

```
[  109.408003] cdc_acm 2-2:1.2: ttyACM0: USB ACM device
```

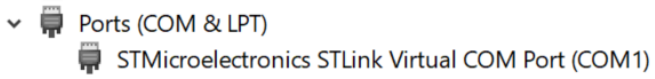


Figure 7: Serial port was retrieved with Device Manager on Windows

After the aforementioned steps, the connection was able to set with PuTTY by setting the connection type to *serial* and the speed to *115200* on both, Windows and Linux. Also, the serial line had to be set, which was the only difference between the two OSs. On Windows it was set to *COM1* and on Linux to */dev/ttyACM0*. Finally, the *Implicit CR in every LF* option was enabled under Terminal options to ensure that the strings were printed with line breaks. The configuration on Windows is shown in Figure 9 (a) and on Linux in Figure 9 (b).

In order to be able to see the results on a serial monitor, the writes to the stream buffer had to be directed to the serial port. Examples of functions that write to the stream buffer are *std::cout* and *printf* functions. Directing the messages was achieved by utilising the USART communication. The processor on the target board provided three different USART interfaces but the user manual [68] of the Nucleo-F767ZI target board showed that the USART3 was the intended one on this particular model.

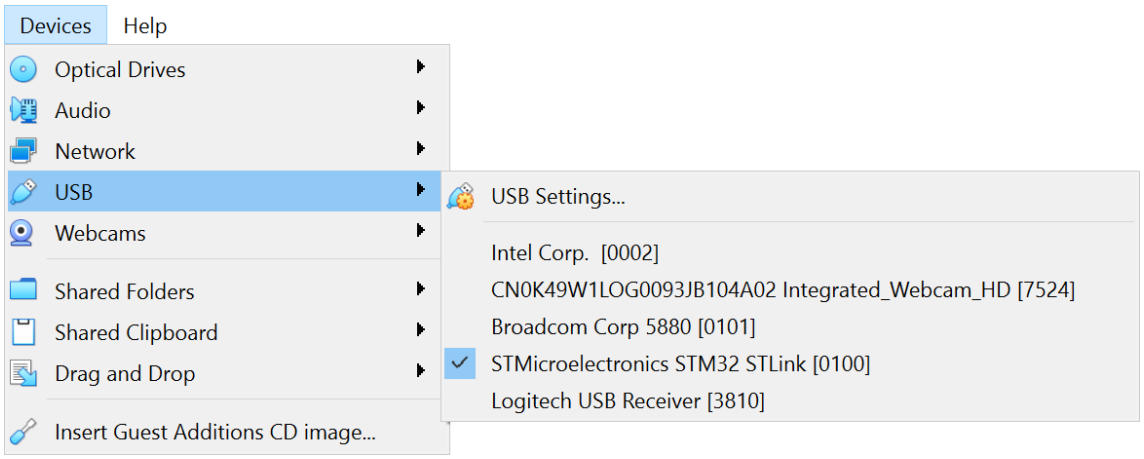
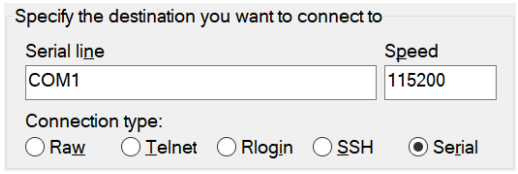
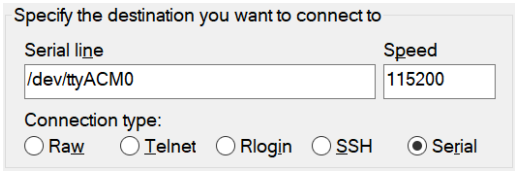


Figure 8: USB had to be enabled in VirtualBox



(a) PuTTY configuration on Windows



(b) PuTTY configuration on Linux

Figure 9: The configurations serial lines in the configuration of PuTTY were different on different operating systems. On Windows (a) the serial line was set to *COM1* and on Linux (b) to */dev/ttyACM0*

The initialisation of USART3 was generated by STM32CubeIDE and is shown in Listing 6. The baud rate set for PuTTY can also be seen on the line 3.

Directing those messages required overriding the *_write* function that writes to the stream buffer. It was located in the *syscalls.c* file that was generated by STM32CubeIDE. The overriding was possible because the *_write* function was defined weak. In the new overridden function, the messages were directed to the *HAL_USART_Transmit* function. The *HAL_USART_Transmit* was a function that was utilised to transmit data through USART. It was provided with the STM32 libraries and located within the *stm32fxx_hal_usart.c* file [75]. It required four parameters: the intended USART to be utilised, the pointer to the data buffer, the amount of the data to be sent, and the desired timeout duration.

The implementation of the overridden *_write* function is shown in Listing 7. On line 1, the *extern "C"* was included as the function to be overridden was C code instead of C++. On line 4, the usage of the *HAL_USART_Transmit* function is shown. The first parameter of it was the variable that was defined to represent the USART3. The *ptr* was the pointer to the message and the *len* variable for the message length. However, the *ptr* had to be casted to the *uint8_t* type as that was the required type by the function. Lastly, a number 1000 was provided as the fourth argument for the function to ensure that the timeout would not be too short. Finally, on the line 5, the length of the message sent was returned as the return type of the

function was an integer and the original implementation had the same functionality. The implementation was placed simply to the file containing the *main* function.

Listing 6: Initialisation of USART3 was handled by STM32CubeIDE development platform

```
1  USART_HandleTypeDef husart3;
2
3  static void MX_USART3_Init(void)
4  {
5      husart3.Instance = USART3;
6      husart3.Init.BaudRate = 115200;
7      husart3.Init.WordLength = USART_WORDLENGTH_8B;
8      husart3.Init.StopBits = USART_STOPBITS_1;
9      husart3.Init.Parity = USART_PARITY_NONE;
10     husart3.Init.Mode = USART_MODE_TX_RX;
11     husart3.Init.CLKPolarity = USART_POLARITY_LOW;
12     husart3.Init.CLKPhase = USART_PHASE_1EDGE;
13     husart3.Init.CLKLastBit = USART_LASTBIT_DISABLE;
14     if (HAL_USART_Init(&husart3) != HAL_OK)
15     {
16         Error_Handler();
17     }
18 }
```

Listing 7: Overriding the *_write* function was required to be able to see the results of the tests on serial monitor

```
1  extern "C"
2  int _write(int file, char *ptr, int len)
3  {
4     HAL_USART_Transmit(&husart3, (uint8_t*)ptr, len, 1000);
5     return len;
6  }
```

3.4.3 Test framework setup

Google Test had to be compiled with the same compiler that was utilised to compile the target executable. The correct C and C++ compilers were required to be provided for CMake. The compilers were provided with the *-DCMAKE_C_COMPILER* and *-DCMAKE_CXX_COMPILER* options respectively. The C compiler provided was *arm-none-eabi-gcc* and the C++ compiler was *arm-none-eabi-g++*. Also, since the target did not run Linux or any other operating system for that matter, it did not

have the POSIX threads. Hence, the *pthread* option had to be disabled similarly to when compiling gtest for Windows. The disabling of POSIX threads is illustrated in subsection 3.3.2 in Listing 1. In addition, since the aim was to build static libraries of gtest, the *-DCMAKE_TRY_COMPILE_TARGET_TYPE* option was included with *"STATIC_LIBRARY"* as its parameter. Also, gmock was not required as there was no need for mocking while running tests on target. Hence, building of it was prevented similarly to disabling the pthreads. It is shown in Listing 8, where the original version is on the line 1 and the modified version on the line 2.

Listing 8: File CMakeLists.txt - Preventing the building of gmock

```
1 option(BUILD_GMOCK "Builds the googlemock subproject" ON)
2 option(BUILD_GMOCK "Builds the googlemock subproject" OFF)
```

The build process resulted in a vast amount of template errors, which was fixed by removing the *SYSTEM* from the directories to be included. The fix was performed to the *googletest/CMakeLists.txt* file. The change is illustrated in Listing 9, where lines 1 and 2 illustrate the original version and lines 4 and 5 illustrate the fixed version. The commands are long and therefore they have been cut to ensure that the change is visible. Hence, the three dots represent the options that are excluded from the example.

Listing 9: File googletest/CMakeLists.txt - Fixing template errors

```
1 target_include_directories(gtest SYSTEM INTERFACE ... )
2 target_include_directories(gtest_main SYSTEM INTERFACE ... )
3
4 target_include_directories(gtest INTERFACE ... )
5 target_include_directories(gtest_main INTERFACE ... )
```

The steps presented allowed the compilation of the static libraries. However, when the static libraries were included into the build process of the target board, the compilation resulted in multiple *undefined reference to* errors due to multiple missing functions: *___sync_synchronize*, *regfree*, *regexec*, *regcomp*, *dup*, *dup2*, *getcwd*, and *mkdir*. The first six errors, *___sync_synchronize*, *regfree*, *regexec*, *regcomp*, *dup*, and *dup2* were fixed by compiling the libraries with the same compiler options as the target build. The compiler options were provided for CMake with the *–DCMAKE_CXX_FLAGS* option. The provided options were *-mcpu=cortex-m7*, *-std=gnu++14*, *-g3*, *-O0*, *-ffunction-sections*, *-fdata-sections*, *-fno-exceptions*, *-fno-rtti*, *-fno-threadsafe-statics*, *-fno-use-cxa-atexit*, *-Wall*, *-fstack-usage*, *-mfpu=fpv5-d16*, *-mfloat-abi=hard*, and *-mthumb*.

Since the remaining two errors, *getcwd* and *mkdir*, were not able to be fixed with the addition of the compilation options, the functionalities that caused those errors had to be disabled. Firstly, the *getcwd* error was overcome by modifying the

FilePath::GetCurrentDir function in the *googletest/src/gtest-filepath.cc* file. The *getcwd* is normally a functionality to return the current working directory. Since such a functionality was not present on the target, it had to be faked. Fortunately, other users of gtest had encountered the same issue and faked the *FilePath::GetCurrentDir* function to return the hard coded *./* string, which represent the current directory on Linux systems. The modification is illustrated in Listing 10 on the lines 1 to 7. In order to utilise the functionality, the additional *GTEST_OS_NONE* macro definition was added to the line 4. The modified version of the line 4 is shown on the line 9. In addition, the macro definition was provided for the compiler with the *–DCMAKE_CXX_FLAGS* option.

Listing 10: File googletest/src/gtest-filepath.cc - Fix *getcwd* error

```
1 const char kCurrentDirectoryString[] = "./";
2 ...
3 FilePath FilePath::GetCurrentDir() {
4 #if GTEST_OS_WINDOWS_MOBILE || GTEST_OS_WINDOWS_PHONE ||
      GTEST_OS_WINDOWS_RT || GTEST_OS_ESP8266 ||
      GTEST_OS_ESP32
5     return FilePath(kCurrentDirectoryString);
6     ...
7 }
8
9 #if GTEST_OS_WINDOWS_MOBILE || GTEST_OS_WINDOWS_PHONE ||
      GTEST_OS_WINDOWS_RT || GTEST_OS_ESP8266 ||
      GTEST_OS_ESP32 || GTEST_OS_NONE
```

The last error, *mkdir*, was due to the functionality that was utilised to create a folder. Since that would not have made sense on an embedded platform without an OS, it was also safe to be removed. The removal was achieved by modifying the *FilePath::CreateFolder* function in the *googletest/src/gtest-filepath.cc* file. The modification ensured that the folder creation was disabled when compiling with the *GTEST_OS_NONE* macro definition. The macro definition was the same that was provided for fixing the *getcwd* error. The modification is shown in Listing 11 on the lines 3 and 4 in which the execution of the *mkdir* command shown in line 6 was prevented. Also, the *result* variable was simply defined to be 0 to ensure proper compilation.

The final build command for CMake is shown in Listing 12. As with compiling gtest for the host environment, the command had a dot (.) at the end of the command as an argument for setting the current directory as the source folder.

Finally, gtest was ready to be utilised for testing on the target. It had to be included for the linker with the *-lgtest* option. Since the target required the initialisation of the peripherals, such as *USART*, the main function provided by the *gtest_main* library was not able to be utilised. Hence, gtest had to be initialised by the user as explained in subsection 3.3.2. The initialisation utilised is shown in Listing 13,

where line 1 differs from the example shown in Listing 3 because command line arguments cannot be provided for an embedded platform. The following Chapter 4 shows how each of the mock replacement techniques were utilised to test an Led controller on host. In addition, testing the same Led controller on target was also experimented and is presented as well.

Listing 11: File googletest/src/gtest-filepath.cc - Fix *mkdir* error

```
1  bool FilePath::CreateFolder() const {
2  #if ...
3  #elif GTEST_OS_NONE
4      int result = 0;
5  #else
6      int result = mkdir(pathname_.c_str(), 0777);
7  #endif
8
9      if (result == -1) {
10         return this->DirectoryExists();
11     }
12     return true;
13 }
```

Listing 12: Final command to build the static googletest library files for unit testing on Nucleo

```
$ cmake -DCMAKE_TRY_COMPILE_TARGET_TYPE="STATIC_LIBRARY"
   -DCMAKE_C_COMPILER=arm-none-eabi-gcc
   -DCMAKE_CXX_COMPILER=arm-none-eabi-g++
   -DCMAKE_CXX_FLAGS="-mcpu=cortex-m7 -std=gnu++14 -g3
   -O0 -ffunction-sections -fdata-sections
   -fno-exceptions -fno-rtti -fno-threadsafe-statics
   -fno-use-cxa-atexit -Wall -fstack-usage
   -mfpu=fpv5-d16 -mfloat-abi=hard -mthumb
   -DGTEST_OS_NONE" .
```

Listing 13: The main function provided with gtest was not able to be utilised on the target as it had to initialise peripherals. Therefore, the initialisation of gtest was placed into the main function

```
1    ::testing::InitGoogleTest();
2    return RUN_ALL_TESTS();
```

4 Results

Testing of each of the mock replacement techniques presented in section 3.2 was necessary to allow the suitability of each of them to overcome the challenges of embedded software to be evaluated. Testing was performed in a case project at Etteplan, in which a motor controller was implemented. The techniques were utilised with one of the presented unit testing tools, Google Test. In addition, testing on target was also experimented, though not in the case project, but on a common development board. The development board was chosen, because testing on target was not desired by the customer and it allowed this thesis to provide examples with hardware that anyone could acquire, hence making the guideline applicable to a wider audience.

In order to present a simple and instructive guidance on how the techniques were able to be utilised, mocking of a simplified version of the digital input/output controller from the project was chosen as an example. The controller utilised HAL functions provided within the STM32 library. HAL is a part of code that allows the software to interact with the hardware at an abstract level instead of at a detailed hardware level. The controller was named to be the *Led* class and it consisted of two methods, *turnOn* and *checkState*. The *turnOn* method turned the defined Led on, by setting the defined general-purpose input/output (GPIO) pin to high with the *HAL_GPIO_WritePin* function. The *checkState* checked whether the defined Led was turned on and returned a Boolean value accordingly. As with the *turnOn* method, the *checkState* method called the *HAL_GPIO_ReadPin* function, which returned the result. The desired Led was provided with two arguments in the constructor of the class: the GPIO peripheral and the number of the pin. The *Led* class is shown in Listing 14. Even though the presented *Led* class can be considered to be almost a dummy class as it does not have any additional logic that would require testing, it illustrates how all of the *HAL* functions provided within the STM32 library were able to be mocked. The header inclusions and const keywords are omitted from the code to simplify the examples but the complete versions can be found from the provided Git repository [76]. Additional observations were collected into Appendix A, which may be useful for practising TDD or unit testing with C++.

Each of the mock replacement techniques allowed a real implementation to be replaced with a test double. The usage of the test double depended on the technique. For example, the macro preprocessed- and link-time-based techniques replaced the existing implementation completely and did not require the user to provide the test double. However, it meant that there was not a direct access to the test double and further modifications were required to allow the control of the test double in the tests. On the other hand, the inheritance- and interface-based replacement techniques did not completely replace the existing implementation but only allowed a test double to be utilised instead of the real implementation. Hence, it had to be provided with dependency injection. The positive aspect was that dependency injection ensured that additional modifications were not required to gain access to the test double in the tests. The replacement of a real implementation with a test double with each of the techniques is demonstrated in section 4.1. Then, the tests were able to be

created and the test double was able to be controlled in the tests, as explained in section 4.2. It also explains how the same *Led* class was able to be tested on target.

Listing 14: File Led.h - Original implementation of the Led class, which was able to be tested on host with each of the mocking techniques. In addition, testing of it was also exemplified on on target

```
1  class Led {
2  public:
3      Led(GPIO_TypeDef* io_port, uint16_t io_pin) :
4          port(io_port),
5          pin(io_pin)
6          {};
7
8      void turnOn() {
9          HAL_GPIO_WritePin(port, pin, GPIO_PIN_SET);
10     }
11
12     bool checkState() {
13         return HAL_GPIO_ReadPin(port, pin);
14     }
15
16 private:
17     GPIO_TypeDef* port;
18     uint16_t pin;
19 };
```

4.1 Test double replacement

In order to allow the replacement of the STM32 HAL library functions with test doubles, the HAL functions were not able to be directly called in the *Led* class. Hence, they were moved from the *Led* class to the new *HAL* class. The *HAL* class was then either declared within the class or provided for it depending on the technique, further elaborated in the following section 4.2. The methods were called from the *Led* class similarly to the original implementation and are shown in Listing 15.

As with the original *Led* class, the *HAL* class consisted of two methods that called the same HAL functions. The names of the methods were the same as the names of the HAL functions but without the *HAL_* prefix. Hence, the names were *GPIO_WritePin* and *GPIO_ReadPin*. They also had the same return types as the HAL functions themselves. The *HAL* class for the macro preprocessed- and link-time-based mock replacement techniques is shown in Listing 16.

Listing 15: File Led.h - Mocking of the STM32 HAL library functions required them to be moved from the *Led* class to the new *HAL* class. The *HAL* class was then either declared within the *Led* class or provided for it, depending on the mock replacement technique

```
1 ...
2     void turnOn() {
3         hal.GPIO_WritePin(port, pin, GPIO_PIN_SET);
4     };
5
6     bool checkState() {
7         return hal.GPIO_ReadPin(port, pin);
8     };
9 ...
```

Listing 16: File HAL.h - Original HAL class simply collected the STM32 HAL library functions together to allow replacing them with a test double

```
1  class HAL {
2  public:
3          void GPIO_WritePin(GPIO_TypeDef* io_port, uint16_t
               io_pin, GPIO_PinState pin_state) {
4                  HAL_GPIO_WritePin(io_port, io_pin,
                       pin_state);
5          }
6
7          GPIO_PinState GPIO_ReadPin(GPIO_TypeDef* io_port,
               uint16_t io_pin) {
8                  return HAL_GPIO_ReadPin(io_port, io_pin);
9          }
10 };
```

The macro preprocessed-based mock replacement technique relied on the *#ifdef*, *#else*, and *#endif* preprocessor directives and the *UNITTEST* macro definition, as shown in Listing 17. The macro definition controlled the inclusion of the desired header. By compiling the code with the macro definition, a test double was included and without it a real implementation was included. Hence, for testing, the code was compiled with the macro definition and for production without it. The macro definition was provided for the compiler with the *-D* option.

The implementation of the test double was also named to be *HAL* but it was placed into the new *HALMock.h* file. It had the same methods as the original implementation, but they were implemented with the *MOCK_METHOD* macro of gmock. The *HAL* class is shown in Listing 18. Furthermore, not all methods of the original implementation would have been needed to be mocked. For example, if the

HAL class had had other methods, only the ones that were called in the tests would have been needed to be mocked.

Listing 17: The macro preprocessed-based mocking relied on the *#ifdef*, *#else*, and *#endif* preprocessor directives and the *UNITTEST* macro definition

```
1 #ifdef UNITTEST
2     #include "HALMock.h"
3 #else
4     #include "HAL.h"
5 #endif
```

Listing 18: File HAL.h - Since the macro preprocessed- and link-time-based techniques replaced the real implementation completely, the test double was also named to be *HAL*

```
1 class HAL {
2 public:
3     MOCK_METHOD(GPIO_PinState, GPIO_ReadPin, (GPIO_TypeDef
          * GPIOx, uint16_t GPIO_Pin));
4     MOCK_METHOD(void, GPIO_WritePin, (GPIO_TypeDef* GPIOx,
          uint16_t GPIO_Pin, GPIO_PinState PinState));
5 };
```

The link-time-based mock replacement technique utilised the same test double as the macro preprocessed-based technique, but instead of relying on preprocessor directives, the header files of the real implementation and the test double were placed into different folders. The header of the real implementation and production specific files, such as the file containing the main function, were placed into the *production* folder. The headers containing test doubles and the tests itself were placed into the *testing* folder. Common headers and compile units were placed into the upper source folder. The folder structure is shown in Figure 10. Since the specific files for production and testing were separated from each other, the desired executable was able to be built simply by building the compile units from the desired folder, *production* or *testing*, and combining them with the compile units from the common *source* folder. The commands for building the executable for testing, excluding the options for gtest and gmock, are shown in Listing 19. The executable for production was able to be built by switching the *testing* folder to the *production* folder in the first command. The commands were run in the *source* folder.

The presented approach did not make use of a common interface header as suggested in section 3.2. A common interface header was not needed because the implementation of the test double was created solely with headers by utilising the *MOCK_METHOD* macro of gmock. However, the common interface header would have been useful if a fake object would have been used instead of a test double.

Hence, it was also experimented. The common interface header required the headers in the *production* and *testing* folders to be replaced with compile units. The compile units included a common header file, which was located in the upper source folder. Otherwise the folder structure was the same as shown in Figure 10. The same commands as in Listing 19 were utilised, but the common header was included from the upper source folder by utilising the *-I.* option in the compilation. The *I* option added its argument to the list folders to be searched for headers for the compiler. Since the headers were located in the current folder, the required argument was a dot (.) In other words, the current folder had to be added to the list of folders to be searched from.

Listing 19: Commands required to build tests with the link-time-based mock replacement mock replacement technique

```
$ g++ -c testing/*.cpp
$ g++ *.o
```

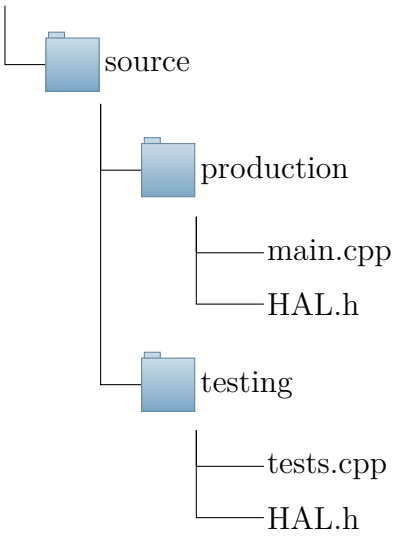


Figure 10: The link-time-based mock replacement allowed the usage of test doubles by separating the real implementations and the test doubles to different folders

The interface-based mock replacement technique required the creation of an abstract interface class that declared the methods as the original *HAL* class but the methods were defined to be pure virtual. The new interface class was named to be *HALInterface* and it is shown in Listing 20. The real implementation and the test double then inherited from the interface class. They also provided implementations for the methods of the interface. The change to the real implementation is shown in Listing 21. As for the test double, it also inherited from the interface class but also an argument *(override)* was provided as for the *MOCK_METHOD* macros it

was suggested in documentation of gmock [54] when mocking pure virtual methods. The required changes for the *HALMock* test double compared to the original implementation of the *HAL* class are shown in Listing 22.

Listing 20: File HALInterface.h - The interface-based mock replacement technique required the creation of an abstract interface class from which the real implementation and the test double inherited

```
1 class HALInterface {
2 public:
3     virtual void GPIO_WritePin(GPIO_TypeDef* io_port,
          uint16_t io_pin, GPIO_PinState pin_state) = 0;
4     virtual GPIO_PinState GPIO_ReadPin(GPIO_TypeDef*
          io_port, uint16_t io_pin) = 0;
5 };
```

Listing 21: File HAL.h - Modifications required by the interface-based mock replacement

```
1 class HAL : public HALInterface {
2 ...
```

Listing 22: File HALMock.h - The interface-based mock replacement technique the test double to inherit from the HALInterface class and the addition of (override) parameter to each method

```
1 class HALMock : public HALInterface {
2 public:
3     MOCK_METHOD(GPIO_PinState, GPIO_ReadPin, (GPIO_TypeDef
          * GPIOx, uint16_t GPIO_Pin), (override));
4     MOCK_METHOD(void, GPIO_WritePin, (GPIO_TypeDef* GPIOx,
           uint16_t GPIO_Pin, GPIO_PinState PinState), (
          override));
5 };
```

The inheritance-based mock replacement technique was utilised by inheriting from the real implementation *HAL* and overriding the desired methods. In this example, both the *GPIO_WritePin* and *GPIO_ReadPin* methods were overridden. However, the methods to be overridden had to be defined as virtual in the original implementation. Hence, the methods of the *HAL* class were defined as virtual. The modifications are shown in Listing 23. The implementation of the test double was similar in comparison with the macro preprocessed- and link-time-based mock

replacement techniques. The only difference was the required inheritance from the real implementation, as shown in Listing 24. Since the inheritance-based mock replacement technique inherited from the original implementation, the *Led* class did not need to be modified as with the interface-based technique.

Listing 23: File HAL.h - The inheritance-based mock replacement technique required the original methods of the *HAL* class to be defined as virtual

```
1 class HAL {
2 public:
3     virtual void GPIO_WritePin(GPIO_TypeDef* io_port,
          uint16_t io_pin, GPIO_PinState pin_state) {
4     ...
5
6     virtual GPIO_PinState GPIO_ReadPin(GPIO_TypeDef*
          io_port, uint16_t io_pin) {
7     ...
```

Listing 24: File HALMock.h - The inheritance-based mock replacement technique only required to inherit from the original *HAL* class. Otherwise the implementation was the same in comparison with the macro preprocessed- and link-time-based techniques

```
1 class HALMock : public HAL {
2 ...
```

4.2 Test double usage

The usage of the test doubles varied on the techniques. The macro preprocessed- and link-time-based techniques replaced the real implementation completely with a test double. Therefore, the desired implementation did not have to be provided for the class to be tested. Hence, the *HAL* class was able to be utilised simply by declaring it as a member within the class as shown in Listing 25. However, if the test double was declared as private, it was not able to be accessed outside of the class. This meant that it was not able to be controlled in the tests. The control was achieved with three different modification approaches. The first approach utilised the principle of the macro preprocessed-based technique. It changed the *HAL* member from private to public in the tests. The change was achieved in the tests by adding wrapping a public keyword inside preprocessor directives before the declaration of the *HAL* class. Hence, similarly to the mock replacement technique, the *HAL* class was defined to be public in the tests and private while compiling for production by having the *UNITTEST* macro definition present only while compiling the code for tests. The modification is shown in Listing 26.

Listing 25: File Led.h - The *HAL* class was able to be declared within the *Led* class with the macro preprocessed-based and link-time-based techniques

```
1  ...
2  private:
3      GPIO_TypeDef* port;
4      uint16_t pin;
5      HAL hal;
6  };
```

Listing 26: File Led.h - Preprocessor directives allowed the *HAL* class to be defined as public for testing and private for production

```
1  ...
2  private:
3      GPIO_TypeDef* port;
4      uint16_t pin;
5  #ifdef UNITTEST
6  public:
7  #endif
8      HAL hal;
9  };
```

The tests were then able to be created. The test double was controlled with the *EXPECT_CALL* and *EXPECT_EQ* macros of gmock. The *EXPECT_CALL* macro enabled controlling the return value of the intended method of a particular object and verifying that it had been called the right amount of times with the right parameters. The desired method was chosen by providing two arguments for the macro. The first argument was the name of the desired object and the second the name of the desired method. The control of the test double was provided with additional clauses for the macro. For example, to perform a desired action once, a clause *WillOnce* was placed after the macro with a dot (.) in between the macro and the clause. The clause required an action to be provided as an argument. In order for the action to return an intended value, the function *Return* of the namespace *testing* was placed as an argument for the clause *WillOnce*. Finally, the intended return value was provided as an argument for the function *Return*. The *EXPECT_EQ* macro verified that the two values were equal. For example, in this instance it was utilised to verify that a return value of a particular method matched a predefined value. It required two parameters, the method to be verified and the predefined value. The order of the parameters did not matter. It only affected the error message, which was understandable with both orders. In the project it was chosen that the testable value would be the first parameter.

The tests are shown in Listing 27. In order to ensure that they are as simple

as possible to understand, they were created in the form of arrange-act-assert, as suggested by Langr [38] in subsection 2.2.4. The first *CheckLedState* test first declared the *Led* class on the line 2. Then, two expectations were set for calling the *GPIO_ReadPin* method of the *HAL* class on the lines 3 to 5. Also, the expectations were also set for the parameters, with which the method had to be called. On the first call the method would return the value *GPIO_PIN_RESET* and on the second call it would return the value *GPIO_PIN_SET*. The first value was a struct definition for 0 and the second for 1. Hence, they represented Boolean values false and true respectively. Then, the *checkState* method of the *Led* class was called twice and the return values were verified on the lines 6 and 7. This test ensured that the *checkState* method was able to return both Boolean values and that the struct values matched the intended Boolean values. The second *TurnLedOn* test also declared the *Led* class on the line 11. Then it set an expectation for the *GPIO_WritePin* method of the *Led* class on the line 12. As with the first test, the parameters with which the method was to be called, were set. Finally, the test called the *turnOn* method of the *Led* class on the line 13. The return value was not verified as the method did not return anything.

Listing 27: File tests.cpp - Tests shown with the macro preprocessed- and link-time-based mock replacement techniques

```
1  TEST(LedTest, CheckLedState) {
2      Led led(LED1_GPIO_Port, LED1_Pin);
3      EXPECT_CALL(led.hal, GPIO_ReadPin(LED1_GPIO_Port,
           LED1_Pin))
4          .WillOnce(testing::Return(GPIO_PIN_RESET))
5          .WillOnce(testing::Return(GPIO_PIN_SET));
6      EXPECT_EQ(led.checkState(), false);
7      EXPECT_EQ(led.checkState(), true);
8  }
9
10 TEST(LedTest, TurnLedOn) {
11     Led led(LED1_GPIO_Port, LED1_Pin);
12     EXPECT_CALL(led.hal, GPIO_WritePin(LED1_GPIO_Port,
           LED1_Pin, GPIO_PIN_SET));
13     led.turnOn();
14 }
```

The second approach utilised the principle of the link-time-based mock replacement technique. It provided two definitions of the *Led* class. The definitions were placed into different files and as with the mock replacement technique, the desired file was chosen with the linker. The first definition was for production and had the access to the test double defined as public. The second definition was for the tests and had the access defined as private. Hence, it allowed creating the tests in the same way as with the first approach. The tests with this approach were shown in Listing 27.

The third approach relied on exposing the test double with a new public method. Since the new method was intended to be private in the production, it was implemented into a new class, which inherited from the original class. However, in order to allow the new method to expose the test double, its access had to be set to protected instead of private in the original class. The new method was defined into the new *LedExposed* class, as shown in Listing 28. The *LedExposed* class inherited from the original *Led* class on the line 1. The inheritance had to be public to ensure that the methods of the original *Led* class were able to be accessed from the *LedExposed* class. The method to expose the *HAL* class was defined to be public and named *getHAL*. The *getHAL* method returned a reference to the object *hal*, as shown on the line 5. The *LedExposed* class was declared and tested instead of the original class in the tests, as shown in Listing 29 on the lines 2 and 7. Finally, the test double was able to be accessed with the *getHAL* method, as shown on the lines 3 and 8.

Listing 28: File LedExposed.h - The *HAL* member was able to be exposed with a new method. The method was implemented into a new class that inherited from the original class, ensuring that the member was not exposed in the original class

```
1 class LedExposed : public Led {
2 public:
3     LedExposed(GPIO_TypeDef* io_port, uint16_t io_pin) :
          Led(io_port, io_pin) {}
4
5     HAL& getHAL() {
6         return hal;
7     };
8 };
```

Listing 29: File tests.cpp - With the help of the new class the *HAL* member was able to be accessed in the tests

```
1 TEST(LedTest, CheckLedState) {
2     LedExposed led(LED1_GPIO_Port, LED1_Pin);
3     EXPECT_CALL(led.getHAL(), GPIO_ReadPin(LED1_GPIO_Port,
          LED1_Pin))
4 ...
5
6 TEST(LedTest, TurnLedOn) {
7     LedExposed led(LED1_GPIO_Port, LED1_Pin);
8     EXPECT_CALL(led.getHAL(), GPIO_WritePin(LED1_GPIO_Port
          , LED1_Pin, GPIO_PIN_SET));
9 ...
```

As for the interface- and inheritance-based mock replacement techniques, the declaration was not able to be utilised because those techniques required the implementation to be chosen. Thus, dependency injection had to be introduced. All three types of dependency injection techniques presented in section 3.2 were able to be utilised with both the interface- and the inheritance-based techniques. In addition, dependency injection was also tested to work with the macro preprocessed- and link-time-based techniques. However, since the usage was similar with all the mock replacement techniques, the dependency injection techniques are only shown with the inheritance-based technique. The usage was almost the same with the macro preprocessed- and link-time-based techniques. The only difference required was that the *HALMock* had to be changed to *HAL* in the tests. Similarly, with the interface-based technique each instance of the *HAL* class had to be replaced with the *HALInterface* class.

The parameter injection required the methods of a class to receive the dependency as an argument. The modifications to the *Led* class are shown in Listing 30 on the lines 2 and 6. Hence, in the tests the *HAL* class was first declared and then provided as an argument for the methods of the *Led* class, as shown in in Listing 31.

Listing 30: File Led.h - With parameter injection the test double was provided as an argument for the methods

```
1 ...
2     void turnOn(HAL &hal) {
3         hal.GPIO_WritePin(port, pin, GPIO_PIN_SET);
4     };
5
6     bool checkState(HAL &hal) {
7         return hal.GPIO_ReadPin(port, pin);
8     };
9 ...
```

Listing 31: File tests.cpp - With the parameter injection the *HALMock* class was provided for the *Led* class while calling its methods

```
1 TEST(LedTest, CheckLedState) {
2     HALMock hal;
3     Led led(LED1_GPIO_Port, LED1_Pin);
4     EXPECT_CALL(hal, GPIO_ReadPin(LED1_GPIO_Port, LED1_Pin
          ))
5         .WillOnce(testing::Return(GPIO_PIN_RESET))
6         .WillOnce(testing::Return(GPIO_PIN_SET));
7     EXPECT_EQ(led.checkState(hal), false);
8     EXPECT_EQ(led.checkState(hal), true);
9 }
```

```
10
11 TEST(LedTest, TurnLedOn) {
12     HALMock hal;
13     Led led(LED1_GPIO_Port, LED1_Pin);
14     EXPECT_CALL(hal, GPIO_WritePin(LED1_GPIO_Port,
           LED1_Pin, GPIO_PIN_SET));
15     led.turnOn(hal);
16 }
```

The constructor injection required the dependency to be provided within the constructor of the class as a reference. Hence, the reference member of the *HAL* class was also required to be declared in the *Led* class. The modifications are shown in Listing 32 on the lines 3, 6 and 12. The *HAL* class was provided with an initialiser list as it resulted in the fewest lines of code, but naturally a normal constructor method could also have been used instead. The constructor injection was utilised in the tests by supplying the *HAL* class in the constructor of the *Led* class, as shown in Listing 33 on the lines 3 and 13.

Listing 32: File Led.h - The test double was provided as a reference for the *Led* class with the constructor injection

```
1  class Led {
2  public:
3      Led(GPIO_TypeDef* io_port, uint16_t io_pin, HAL &hal)
           :
4          port(io_port),
5          pin(io_pin),
6          hal(hal)
7          {};
8  ...
9  private:
10     GPIO_TypeDef* port;
11     uint16_t pin;
12     HAL &hal;
13 };
```

The setter injection relied on providing the dependency with a public method after initialising the object, as shown in Listing 34 on the lines 8 to 10. However, in contrast to the parameter and constructor injections, the setter injection required the dependency to be a pointer instead of a reference, as shown on the line 23. A pointer was required because a reference was not able to be left uninitialised, but the same was possible with a pointer. It also required the *turnOn* and *checkState* methods to call the methods of the *HAL* class with the arrow operator instead of the dot operator. The usage of the setter injection in the tests is shown in Listing 35

in which the *HAL* class was provided for the was provided for the *Led* class after the declaration, as shown on the lines 4 and 9.

Listing 33: File tests.cpp - With the constructor injection the *HALMock* class was provided for the *Led* class in its constructor

```
1  TEST(LedTest, CheckLedState) {
2      HALMock hal;
3      Led led(LED1_GPIO_Port, LED1_Pin, hal);
4      EXPECT_CALL(hal, GPIO_ReadPin(LED1_GPIO_Port, LED1_Pin
           ))
5          .WillOnce(testing::Return(GPIO_PIN_RESET))
6          .WillOnce(testing::Return(GPIO_PIN_SET));
7      EXPECT_EQ(led.checkState(), false);
8      EXPECT_EQ(led.checkState(), true);
9  }
10
11 TEST(LedTest, TurnLedOn) {
12     HALMock hal;
13     Led led(LED1_GPIO_Port, LED1_Pin, hal);
14     EXPECT_CALL(hal, GPIO_WritePin(LED1_GPIO_Port,
           LED1_Pin, GPIO_PIN_SET));
15     led.turnOn();
16 }
```

Listing 34: File Led.h - The setter injection required the test double to be a pointer instead of a reference

```
1  class Led {
2  public:
3      Led(GPIO_TypeDef* io_port, uint16_t io_pin) :
4          port(io_port),
5          pin(io_pin)
6          {};
7
8      void setHAL(HAL *h) {
9          hal = h;
10     }
11
12     void turnOn() {
13         hal->GPIO_WritePin(port, pin, GPIO_PIN_SET);
14     };
15
16     bool checkState() {
```

```
17            return hal->GPIO_ReadPin(port, pin);
18        };
19
20    private:
21        GPIO_TypeDef* port;
22        uint16_t pin;
23        HAL *hal;
24    };
```

Listing 35: File tests.cpp - With the setter injection the *HALMock* class was provided for the *Led* class with the *setHAL* method after the declaration

```
1  TEST(LedTest, CheckLedState) {
2      HALMock hal;
3      Led led(LED1_GPIO_Port, LED1_Pin);
4      led.setHAL(&hal);
5  ...
6  TEST(LedTest, TurnLedOn) {
7      HALMock hal;
8      Led led(LED1_GPIO_Port, LED1_Pin);
9      led.setHAL(&hal);
10 ...
```

Finally, testing on target was simpler as it did not require any modifications to be made to the original *Led* class presented in Listing 14. The tests were also able to be formed in more meaningful manner. Instead of testing both return values of the *checkState* method, the first test verified that the Led was off initially. The second test then turned the Led on and verified that its state was on afterwards. The tests are shown in Listing 36. The following Chapter 5 evaluates the mock replacement techniques as well as testing on target based on the results shown in this Chapter.

Listing 36: File tests.cpp - Tests on target did not required the usage of test doubles

```
1  TEST(LedTest, LedIsOffInitially) {
2      Led led(LED1_GPIO_Port, LED1_Pin);
3      EXPECT_EQ(led.checkState(), false);
4  }
5
6  TEST(LedTest, LedTurnsOn) {
7      Led led(LED1_GPIO_Port, LED1_Pin);
8      led.turnOn();
9      EXPECT_EQ(led.checkState(), true);
10 }
```

5 Evaluation

The results presented in the previous Chapter 4 showed that testing on host can be achieved with any of the four mock replacement techniques presented in section 3.2. Hence, all of them are suitable for overcoming the difficulties related to TDD in the embedded domain that were presented in subsection 2.2.3. In addition, since the techniques allow tests to be executed without hardware, the execution of the tests can be automated to run on any CI server. The downside of the techniques is that they require the original code to be modified. The results also showed that testing on target can also be considered as an option. Though, with a greater number of tests, memory or performance issues may start to hinder the practice of TDD if testing on target is practised. The different techniques, as well as testing on target, are evaluated in this Chapter.

Since each of the mock replacement techniques allowed a real implementation to be replaced with a test double, the differences consisted of the changes required to the source code as well of the usage in the tests. Naturally, the best technique would have been one that requires the fewest amount of modifications to the original code and is simplest to practise, which was emphasised by Shen and Yang [34].

The macro preprocessed-based technique led to a bit messier code but the modification was acceptable as it only modified the inclusion of headers and not, for example, class implementations. The link-time-based technique, on the other hand, did not require the code to be changed at all, which is a clear advantage over the macro preprocessed-based technique. However, the requirement to modify the linking process and have the real implementation and test double in particular folders may result in a bit confusing arrangement of the source code. Hence, neither of the techniques can be recommended over the other.

The significant downside of the macro preprocessed- and link-time-based techniques was that even though they allowed a real implementation to be replaced with a test double, the test double was difficult to be accessed in the tests. Naturally, the access would have not been an issue if the test double would have been intended to be utilised as a dummy object or test stub as then the test double would not have been needed to be controlled. In addition, if the real implementation would have been defined to be public, there would not have been an issue with the access. However, neither of these scenarios were frequent in the case project. Instead, the test double was almost always defined to be private and required to be controlled in almost in every test. The first and second approaches to allow the test double to be accessed had the same characteristics as the macro preprocessed- and link-time-based techniques. The first approach with the preprocessor directives caused the code to be less readable but did not modify the existing implementation. The second approach did modify the code at all but required a certain arrangement of the source code. The third approach did not have an influence on the readability of the code nor on the organisation of the files, but it modified the existing implementation by requiring the real implementation to be defined to be protected. The third approach was better than the first and second ones in the sense that the code remained more readable. However, the modification could have been problematic if other classes

would have inherited from the class in question. With the real implementation defined as protected, the inheriting classes could have changed the access to the implementation to public, which could have caused a security vulnerability. Hence, neither the macro preprocessed- nor the link-time-based technique was perfect as each of the three possible approaches to expose the test double had undesired effects.

The advantage of the interface-based technique was that it allowed the creation of test doubles even before implementing the real implementations. Hence, wider adoption of the technique could increase the modularity of the code significantly and allow each developer to program and test their module of code without having to depend on other developers. In addition, it did not require the real implementation to be modified. However, the disadvantage was the requirement for the interface class. That is because if the creation of test cases is started in the middle or late in the development a substantial amount of rework is required with the interface-base technique, which was noted in the case project. Hence, the early adoption of TDD must be emphasised even more with this technique than with the other techniques, which confirms the suggestion provided by Shen and Yang [34] to adopt TDD already in the beginning of a project as it would eliminate this disadvantage. On the other hand, the interface-based technique may create a bit of confusion since the developers must utilise the name of the interface and not the name of the real implementation. For example, in the results the instances of the *HAL* class would have been needed to be replaced with the *HALInterface* class. Though, if inheritance would be utilised anyway in the code, the usage of an interface class could be recommended over the other techniques as then it would not require any additional modifications.

The inheritance-based technique was the most utilised mocking technique in the project. The reason for that was the easy usage after most implementations were already developed. In addition, as with the macro preprocessed- and link-time-based techniques, there were not any possibilities for confusions as the instances of the *HAL* class did not have to be modified as was the case with the interface-based technique. The main downside was that the methods of the real implementation to be mocked had to be defined to be virtual. Now, if other classes would inherit from the real implementation, they could redefine the method, which might not be intended and could cause a security vulnerability. In addition, it may cause a slight performance reduction. Another issue with the technique was that it required the real implementations to be compiled with the tests.

A common factor for the macro preprocessed- and link-time-based techniques in comparison with the interface- and inheritance-based techniques was that only the latter allowed the usage of both real implementations and test doubles simultaneously in tests. It was not required in the project but could have been useful in a certain scenario.

The interface- and inheritance-based techniques required the usage of dependency injection, though it has to be noted that dependency injection could have been utilised with all the techniques. Firstly, the parameter injection resulted in the simplest declaration of a class and it ensured that the dependency was not forgotten to be provided as otherwise the code would have not compiled. However, it would have been tedious to utilise, if the methods requiring the dependency would have

been required to be called in multiple places. Hence, it could be recommended only if those methods are called in few locations. The constructor injection had an advantage that it resulted in the simplest form as the dependency was provided only once and within the constructor. However, it was noted in the project that it may lead to long parameter lists, which was not desired. On the other hand, long parameter lists were able to be shortened by combining multiple parameters into a struct, which ensured that the constructor injection was able to be utilised even when a significant amount of dependencies had to be supplied. Lastly, the setter injection did not have the same issue but since the dependencies had to be provided after calling the constructor, there was a risk of forgetting to perform the injection. Naturally, if multiple dependencies would have been required to be provided, then the declaration of a class would have become tedious as multiple lines of code would always have been required.

Testing on target did not require additional interface layers such as the *HAL* class. Therefore, the original implementation did not require to be modified. In addition, the tests were able to verify actual functions and not rely on test doubles. Hence, testing on target seemed superior compared to testing on host but it was actually hardly the case. Firstly, the tests in question happened to be able to be safely tested. The tests could have, for example, verified the functionalities of a motor. Consequently, if the tests would have verified abnormal conditions, they could have potentially broken the motor. In addition, with testing on target one has to be more careful that the tests do not affect one another, which was also emphasised by Beck [35] in subsection 2.2.4. For example, the tests presented would not have passed if their order would have been reversed. That is because the second test turned the Led on and it did not turn it off. Hence, the first test would have then checked that the Led is on and not off and the test would have failed.

Additionally, with testing on target one also has to consider the setup and execution of tests. Firstly, setting up Google Test for testing on target was significantly more difficult in comparison with setting it up for testing on host. That, however, was not surprising as it was already mentioned by Cordemans et al. [22]. In addition, the serial communication was also required to be set up, which meant that if different target devices would be intended to be tested, the serial communication would have to be set up for each device. Naturally, this does not prove that other testing tools are as difficult to be utilised for testing on target, but it is probable if they are not aimed for embedded development. Nevertheless, Catch2 [58] and CxxTest [59] seem to be a simpler alternatives as they only require a header that must be included in the compilation. However, they do not include a mocking framework, which means that if at least some of the tests would be desired to be performed on host, another testing framework would also have to be utilised. That is a disadvantage, as the usage of two frameworks would probably increase the initial costs since developers would have to learn to create tests with two different frameworks. As for automating the tests, tests executed on target are significantly more difficult to be set to be executed in a CI server, as was explained by Boydens et al. [20]. Also, as emphasised by Meszaros [51] and Beck [35], testing cycles may be too long as the developer must wait not only for a test executable to compile but also for it to be flashed to a target

hardware before tests can be executed and results can be seen. Finally, as Ronkainen and Abrahamsson [36] explained, each developer must possess the hardware and they must ensure that tests do not exceed the memory and performance limitations.

It has to also be mentioned that additional challenges were noted in the project that were not able to be overcome with gtest and the mock replacement techniques. Though, they were not specific for the embedded domain but could also have been faced in a non-embedded domain. Fortunately, other means were discovered that eliminated the challenges. For example, as explained in Appendix A, a missing return statement was not able to be detected in the tests, but it was overcome with a compiler option. As a second example, overloaded operators were not able to be directly mocked with gmock, but it was also achieved with an additional method, as presented in Appendix A. Consequently, even though this thesis shows how to overcome the hardware dependencies and memory and performance related issues, other challenges may arise that require additional unit testing knowledge that is not presented in this thesis.

Additionally, as emphasised by Meszaros [51] and Beck [35], tests should be able to be run often, which requires that they should be able to be compiled rapidly. As explained in Appendix A, it was noted in the case project that the build times were significantly longer on Windows than on Linux. The difference was most likely due to the pthreads option, which was able to be utilised only on Linux. The exact reason, however, was not studied in more detail. In addition, the build times were able to be reduced significantly by extracting the constructors and destructors of the test doubles to a separate compile unit. Therefore, if the build times seem to prevent the practise of TDD, building the tests on a different operating system can shorten the build times. Also, if the change to another OS is not possible, additional modifications, such as the one presented, may speed up the build times to be fast enough to allow TDD to be practised.

In conclusion, each of the mocking techniques as well as testing on target have their pros and cons and none of them can be said to be better than the rest. Nevertheless, it can be recommended that the interface-based technique should only be applied if a project is new or only very little of it has been developed. In addition, one or more techniques can be utilised in a single project. As for testing on target, it seems superior at first glance but since it does not remove the challenges of the embedded software, it may be unusable for certain types of tests. Also, a notion from the case project was that the preferred techniques were the macro preprocessed- and inheritance-based techniques. The macro preprocessed-based technique was utilised over the link-time-based technique as the preprocessor directives were more familiar to the developers than the linking configurations. The inheritance-based technique was preferred over the interface-based technique as the tests had to be able to be created with minimal effort. Since the case project had already been started before implementing the tests, the interface-based technique would have been more difficult to apply than the inheritance-based technique. As for accessing the test double, if either the macro preprocessed- or the link-time-based technique is utilised, it is better to provide the test double with the dependency injection techniques than to expose it with any of the three approaches presented in section 4.2. That is because the

dependency injection techniques do not harm the readability nor create a security vulnerability. As for the injection techniques, the choice does not matter significantly. One could say that the parameter injection could be practised if the method calls utilising it are scarce, the constructor injection could be preferred with only few dependencies and the setter injection could be utilised if the implementation must not always have to be provided. However, these statements of the injection techniques are merely opinions and open to debate.

6 Conclusions

This thesis aimed to develop a guideline for embedded software developers to assist in the introduction of the TDD technique. The guideline encompasses a set of testing tools and methods. The identified tools were UnitTest++, Google Test, CppUTest, Boost.Test, Catch2, and CxxTest, though only Google Test and CppUTest contained a mocking framework. The implementation was exemplified with Google Test. Four mock replacement techniques were identified and they were evaluated in an embedded case project, in which a motor controller was implemented. The evaluation was based on testing functionalities on host that relied on the STM32 library functions. In addition, the techniques were compared with testing on target, which was chosen to be the Nucleo-F767ZI development board.

The developed guideline was successfully utilised while writing the unit tests in the case project even though it was applied only after a significant portion of the functionalities were already developed. As for the evaluation, all four mocking techniques allowed the replacement of a real implementation with a test double, though none of them was deemed to be superior. The usage of test doubles allowed testing on host, which ensured that TDD could be practised as it removed the dependencies to the hardware as well as memory and performance limitations of the target. The chosen testing tool, Google Test, was considered to be suitable for TDD of embedded software.

This thesis illustrates how TDD can be practised in the embedded domain. It includes detailed steps for the installation of Google Test for Windows and Linux. In addition, the modifications required to run it on an embedded device are provided. Furthermore, the usage of the different mocking techniques is explained with examples from the real world, which assist in the adoption of the techniques. Lastly, this thesis is not only useful for the introduction of TDD but also for developing unit tests after development since the tools and methods required are the same as with TDD.

In summary, although embedded software has hardware dependencies, this evaluation of this guideline shows that the existing tools and methods are sufficient for the adoption of TDD in the embedded domain. Google Test is an example of a tool that can be recommended for testing embedded software since it can be utilised for testing on both the host and the target. The presented mocking techniques allow testing on host and thereby any of them can be chosen to allow TDD of embedded software.

References

[1] A. Bertolino and E. Marchetti, "A brief essay on software testing", *Software Engineering, 3rd edition. Development process*, vol. 1, pp. 393–411, 2005.

[2] *Software fail watch: 5th edition*, https://www.tricentis.com/software-fail-watch/, Accessed: 2020-02-18.

[3] N. Chauhan, *Software Testing - Principles and Practices*. Oxford University Press, 2010.

[4] B. Hambling, P. Morgan, S. Angelina, G. Thompson, and P. Williams, *Software testing: an ISTQB-ISEB foundation guide*. British Informatics Society Limited, 2010.

[5] Y. Singh, *Software testing*. New York: Cambridge University Press, 2012.

[6] S. Koirala and S. Sheikh, *Software testing*. Jones & Bartlett Learning, 2009.

[7] S. Desikan and G. Ramesh, *Software Testing: Principles and Practices*. Pearson India, 2007.

[8] D. Harrison, K. Lively, *et al.*, "Achieving devops", *Springer Books*, 2019.

[9] B. P. Douglass, *Design patterns for embedded systems in C: an embedded software engineering toolkit*. Elsevier, 2010.

[10] J. W. Grenning, *Test-Driven Development for Embedded C*. Pragmatic bookshelf, 2011.

[11] M. J. Karlesky, W. I. Bereza, and C. B. Erickson, "Effective test driven development for embedded software", in *2006 IEEE International Conference on Electro/Information Technology*, IEEE, 2006, pp. 382–387.

[12] N. Van Schooenderwoert and R. Morsicato, "Taming the embedded tiger - agile test techniques for embedded software", in *Agile Development Conference*, 2004, pp. 120–126. DOI: 10.1109/ADEVC.2004.21.

[13] A. Munck and J. Madsen, "Test-driven modeling of embedded systems", in *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC)*, 2015, pp. 1–4.

[14] J. Grenning, "Applying test driven development to embedded software", *IEEE Instrumentation and Measurement Magazine*, vol. 10, no. 6, pp. 20–25, 2007. DOI: 10.1109/MIM.2007.4428578.

[15] O. Salo and P. Abrahamsson, "Agile methods in european embedded software development organisations: A survey on the actual use and usefulness of extreme programming and scrum", *IET software*, vol. 2, no. 1, pp. 58–64, 2008.

[16] M. Emilio and P. Di, *Embedded Systems Design for High-speed Data Acquisition and Control*. Springer, 2015.

[17] J. Ganssle, *Embedded systems dictionary*. CRC Press, 2003.

[18] R. Oshana and M. Kraeling, *Software engineering for embedded systems: Methods, Practical techniques, and Applications*, 2nd edition. Newnes, 2019.

[19] P. A. Laplante and S. J. Ovaska, *Real-time systems design and analysis: tools for the practitioner*, 4th edition. John Wiley and Sons, 2011.

[20] J. Boydens, P. Cordemans, and E. Steegmans, "Test-driven development of embedded software", Mar. 2010.

[21] M. Smith, A. Kwan, A. Martin, and J. Miller, "E-tdd - embedded test driven development a tool for hardware-software co-design projects", 2005. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-26444461294&partnerID=40&md5=9d0f62bc9db1ce6c5021ad13a98bfabe.

[22] P. Cordemans, S. Van Landschoot, J. Boydens, and E. Steegmans, "Test-driven development as a reliable embedded software engineering practice", *Studies in Computational Intelligence*, vol. 520, pp. 91–130, 2014. DOI: 10.1007/978-3-642-40888-5_4. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84958523962&doi=10.1007%2f978-3-642-40888-5_4&partnerID=40&md5=b8c8f425a498e1f59b592f8f4dc085e9.

[23] G. Tassey, "The economic impacts of inadequate infrastructure for software testing", *National Institute of Standards and Technology*, 2002.

[24] A. Mili and F. Tchier, *Software testing: Concepts and operations*. John Wiley & Sons, 2015.

[25] G. O'Regan, *Concise Guide to Software Testing*. Springer, 2019.

[26] M. I. Board, *Mars climate orbiter mishap investigation board phase i report november 10, 1999*, https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf, Accessed: 2020-07-07, 1999.

[27] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*, 3rd edition. John Wiley & Sons, 2012.

[28] "Ieee standard glossary of software engineering terminology", *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[29] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing", in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, IEEE, 2008, pp. 162–168.

[30] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost", in *Proceedings of the 2006 international workshop on Automation of software test*, 2006, pp. 85–91.

[31] B. W. Boehm, *Software Engineering Economics*. Prentice-Hall, 1981.

[32] A. Manzoor, *Information Technology in Business*. CreateSpace Independent Publishing Platform, 2012.

[33] A. Forward and T. C. Lethbridge, "A taxonomy of software types to facilitate search and evidence-based software engineering", in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, 2008, pp. 179–191.

[34] M. Shen, W. Yang, G. Rong, and D. Shao, "Applying agile methods to embedded software development: A systematic review", in *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)*, 2012, pp. 30–36. DOI: 10.1109/SEES.2012.6225488.

[35] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[36] J. Ronkainen and P. Abrahamsson, "Software development under stringent hardware constraints: Do agile methods have a chance?", in *Extreme Programming and Agile Processes in Software Engineering*, M. Marchesi and G. Succi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 73–79, ISBN: 978-3-540-44870-9.

[37] S. Ian, *Software engineering*, Tenth edition. Pearson Education Limited, 2011.

[38] J. Langr, *Modern C++ Programming with Test-Driven Development*. The Pragmatic Programmers, 2013.

[39] J. C. Sanchez, L. Williams, and E. M. Maximilien, "On the sustained use of a test-driven development practice at ibm", in *Agile 2007 (AGILE 2007)*, Aug. 2007, pp. 5–14. DOI: 10.1109/AGILE.2007.43.

[40] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[41] M. Xie, M. Shen, G. Rong, and D. Shao, "Empirical studies of embedded software development using agile methods: A systematic review", in *Proceedings of the 2nd international workshop on Evidential assessment of software technologies*, 2012, pp. 21–26.

[42] R. Jeffries and G. Melnik, "Guest editor's introduction: Tdd - the art of fearless programming", *IEEE Software*, vol. 24, no. 3, pp. 24–30, 2007. DOI: 10.1109/MS.2007.75. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-34248356409&doi=10.1109%2fMS.2007.75&partnerID=40&md5=76185d1814158491802392b770fca2c4.

[43] J. Bosas, "Automated testing importance and impact", in *2018 IEEE AUTOTESTCON*, 2018, pp. 1–4.

[44] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio, "Evaluating advantages of test driven development: A controlled experiment with professionals", vol. 2006, Jan. 2006, pp. 364–371. DOI: 10.1145/1159733.1159788.

[45] A. Geras, M. Smith, and J. Miller, "A prototype empirical evaluation of test driven development", in *10th International Symposium on Software Metrics, 2004. Proceedings.*, IEEE, 2004, pp. 405–416.

[46] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: Results and experiences of four industrial teams", *Empirical Software Engineering*, vol. 13, pp. 289–302, Jun. 2008. DOI: 10.1007/s10664-008-9062-z.

[47] B. George and L. Williams, "An initial investigation of test driven development in industry.", Jan. 2003, pp. 1135–1139. DOI: 10.1145/952532.952753.

[48] W. Bissi, A. G. S. S. Neto, and M. C. F. P. Emer, "The effects of test driven development on internal quality, external quality and productivity: A systematic review", *Information and Software Technology*, vol. 74, pp. 45–54, 2016.

[49] J. Srinivasan, R. Dobrin, and K. Lundqvist, "'state of the art' in using agile methods for embedded systems development", in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, IEEE, vol. 2, 2009, pp. 522–527.

[50] A. S. Mazuco and E. D. Canedo, "Reports with tdd and mock objects: An improvement in unit tests", *ICSEA 2016*, p. 85, 2016.

[51] G. Meszaros, *xUnit test patterns: Refactoring test code*. Addison-Wesley, 2007.

[52] *Unittest++*, https://github.com/unittest-cpp/, version 2.0.0, Github repository. Accessed: 2020-07-10.

[53] *Unittest++continued*, https://github.com/neacsum/utpp, Github repository. Accessed: 2020-07-10.

[54] *Google test*, https://github.com/google/googletest, version 1.10.0, Github repository. Accessed: 2020-01-20.

[55] *Cpputest - cpputest unit testing and mocking framework for c/c++*, https://cpputest.github.io/, version 3.8, Accessed: 2020-03-12.

[56] *Cppunit test framework*, https://www.freedesktop.org/wiki/Software/cppunit/, Accessed: 2020-07-10.

[57] *Boost c++ libraries*, https://www.boost.org/, Accessed: 2020-07-02.

[58] *Catch2*, https://github.com/catchorg/Catch2, Github repository. Accessed: 2020-06-17.

[59] *Cxxtest*, https://http://cxxtest.com, Accessed: 2020-05-21.

[60] H. Schildt, *C++: The complete reference*, 3rd edition. McGraw-Hill/Osborne, 1998.

[61] *Stack overflow*, https://stackoverflow.com/, Accessed: 2020-02-03.

[62] *Virtualbox*, https://www.virtualbox.org/, Accessed: 2020-01-16.

[63] R. Bast and R. Di Remigio, *CMake Cookbook*, 1st edition. Packt Publishing, 2018.

[64] *Gcc, the gnu compiler collection*, https://gcc.gnu.org/, Accessed: 2020-04-13.

[65] *Gnu operating system - gnu make*, https://www.gnu.org/software/make/, Accessed: 2020-04-13.

[66] *Cygwin - this is the home of the cygwin project*, https://www.cygwin.com/, Accessed: 2020-04-13.

[67] *Mingw - minimalist gnu for windows*, http://www.mingw.org/, Accessed: 2020-04-13.

[68] *Um1974 stm nucleo-144 boards*, 028599, Rev 7, STMicroelectronics, 2017. [Online]. Available: https://www.st.com/resource/en/user_manual/dm00244518-stm32-nucleo144-boards-stmicroelectronics.pdf.

[69] M. Karlesky, G. Williams, W. Bereza, and M. Fletcher, "Mocking the embedded world: Test-driven development, continuous integration, and design patterns", in *Proc. Emb. Systems Conf, CA, USA*, 2007, pp. 1518–1532.

[70] J. Langbridge, *Professional Embedded ARM Development*. John Wiley & Sons, 2014.

[71] H. Tschofenig and M. Pegourie-Gonnard, "Performance of state-of-the-art cryptography on arm-based microprocessors", in *NIST Lightweight Cryptography Workshop*, 2015.

[72] *Cortex-m7 - arm*, https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m7, Accessed: 2020-05-22.

[73] *Tn1235 overview of st-link derivatives*, Rev 3, STMicroelectronics, 2019. [Online]. Available: https://www.st.com/resource/en/technical_note/dm00290229-overview-of-stlink-derivatives-stmicroelectronics.pdf.

[74] *Putty*, https://www.putty.org, Accessed: 2020-05-22.

[75] *Um1905 description of stm32f7 hal and low-layer drivers*, 027932, Rev 3, STMicroelectronics, 2017. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/user_manual/45/27/9c/32/76/57/48/b9/DM00189702.pdf/files/DM00189702.pdf/jcr:content/translations/en.DM00189702.pdf.

[76] *Example codes of the thesis*, https://github.com/SamuliMononen/Thesis, Github repository. Accessed: 2020-07-11.

[77] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, Third. Geneva, Switzerland: International Organization for Standardization, Sep. 2011, p. 1338. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

A Best practices

This appendix presents challenges that were noted in the case project. Explanations are provided for overcoming the challenges along with example code snippets.

Speeding up the build process

Building the test executable took considerable amount of time, as according to the documentation of gmock [54], the majority of the time spent on compiling a test double is spent on generating its constructor and destructor. Hence, the compilation was able to be sped up by extracting the constructors and destructors of the test doubles to a separate compilation unit. Originally, the constructor and destructor were defined in the corresponding header file and if nothing was required to be initialised in the constructor nor deleted in the destructor, they did not have to be defined at all, as shown in Listing 37. However, in order to extract them to a compilation unit, they had to be declared in the header file and the destructor had to be declared to be virtual, as shown in Listing 38. Finally, the constructor and destructor were able to be declared in another compilation unit, as shown in Listing 39.

Listing 37: File HALMock.h - Original version of a test double, which lead to slow compilation

```
1 class HALMock : public HAL {
2     ...
3 }
```

Listing 38: File HALMock.h - The constructor and destructor had to be declared in the test double to allow them to be defined in a new compilation unit

```
1 class HALMock : public HAL {
2     HALMock();
3     virtual ~HALMock();
4     ...
5 }
```

Listing 39: File HALMock.cpp - The constructor and destructor were moved to a new compilation unit to speed up the compilation

```
1 #include "HALMock.h"
2
3 HALMock::HALMock() = default;
4 HALMock::~HALMock() = default;
```

In addition, the build process took longer on Windows than Linux. On Windows it took around one to one and a half minutes whereas on Linux it took approximately 20 seconds. After these modifications the build process took only 30 seconds on Windows and 5 seconds on Linux. The difference in the build times on Windows with MinGW and Cygwin was not significant as it was couple seconds faster on MinGW. On the other hand, the difference between the operating systems was all the more astonishing as Linux was utilised on the same computer in a virtual machine, which should only decrease the performance. Apart from having different compilers, the only difference in the build process was the pthreads option that was only able to be utilised on Linux.

Missing return statement

It was found out that the *EXPECT_TRUE* macro of gtest was not able to be utilised to detect whether a return statement was missing. The scenario shown in Listing 40 was that two functions were chained; the *foo* function called the *bar* function. The *foo* function did not have a return statement, but the *bar* function did have. Both functions were supposed to return a Boolean value. According to C++11 standard [77], such a behaviour is undefined, which means that the implementation can vary from one compiler to another. While that is alarming by itself, it most certainly was not acceptable in the project, in which the test executable was compiled with a different compiler than the executable for the target microcontroller. The issue was overcome by adding a compiler flag *-Werror=return-type*. It treats return-type warnings as errors and prevents the compilation of the software if return statements are missing from functions of which return types are non-void.

Listing 40: Missing return statement

```
1 bool bar() {
2     return true;
3 }
4
5 bool foo () {
6     bar();
7 }
```

Test double as a dummy object or test stub

If test doubles were intended to be utilised as dummy objects or test stubs, expectations were not set for the calls to the methods of the particular test doubles. However, if calls were nonetheless made to those methods, gmock would result in printing warning messages. The issue was overcome with the help of *Nicemock*. The original test double was provided as an argument for the *NiceMock* inside the angle brackets

as shown in Listing 41 on the line 2. Then, it was provided for the class to be tested on the line 3 and tested that it returned true on the line 4.

Listing 41: If a test double was utilised as a dummy object or test stub, it had to be defined as *NiceMock* to prevent printing of warning messages

```
1 TEST(ExampleTest, WithNiceMock) {
2       testing::NiceMock<FooMock> fooMock;
3       foo(fooMock);
4       EXPECT_EQ(foo.bar(), true);
5 }
```

Static methods

Static inline methods were able to be replaced only with the macro preprocessed- and link-time-based mock replacement techniques.

Fixing compilation errors

A programming mistake with either gtest or gmock may lead to a vast amount of error messages, from which the mistake is difficult to determine. An example of a common mistake in the case project was the utilisation of the *TEST* macro instead of the *TEST_F* macro. The simplest approach to avoid these mistakes was to write code similarly to the principles of TDD; create and execute tests one by one, thereby restricting the source of an error to a single location. As for the reason for the vast amount of error messages, they are probably caused by the heavy usage of templates within gtest and gmock.

Mocking methods containing const arguments

Methods containing const variables were required the const argument to be left out from the *MOCK_METHOD* macros. An example is shown in Listing 42 where the original method is shown on the line 1 and the mocked version on the line 2.

Listing 42: Mocking methods containing const arguments

```
1 virtual void foo (const bool bar);
2 MOCK_METHOD(void, foo, (bool bar), ());
```

Mocking const methods

Const methods were also required to be able to be mocked, which was fortunately provided within gmock. For example, if a method *foo* of the class *Foobar* was intended

to be mocked, only the additional fourth *(const)* argument had to be added for the *MOCK_METHOD* macro. The class *Foobar* is shown in Listing 43 on the lines 1 to 3 and the mocked version of it on the lines 5 to 7.

Listing 43: Mocking const methods

```
1 class Foobar {
2     virtual const bool foo();
3 }
4
5 class MockFoobar : public Foobar {
6     MOCK_METHOD(bool, foo, (), (const));
7 }
```

Mocking overloaded operators

Overloaded operators turned up to be impossible to mock as gtest did not support such functionality. Fortunately, there was a workaround. One can delegate operator call to another method. Such scenario is shown in Listing 44, where the overloaded method () is delegated to call method foo, which can then be mocked as usual.

Listing 44: Mocking overloaded operators

```
1  class Foobar {
2      bool operator ()() {
3          foo();
4      }
5
6      virtual bool foo() {
7          return true;
8      }
9  }
10
11 class MockFoobar : public Foobar {
12     MOCK_METHOD(bool, foo, ());
13 }
```

Global variables

The usage of global variables prevented testing altogether. Hence, they had to be removed. An example is shown in Listing 45, where a switch case inside the *foo* function was to be tested. However, since the *state* variable was declared as a global variable, its state was not able to read within the tests. However, by providing the

variable as an argument for the *foo* function, as shown in Listing 46, its value was simple to be checked in the test with the *EXPECT_EQ* macro.

Listing 45: Global variable used in function

```
1 TYPE_STATE state;
2
3 void foo() {
4     switch (state):
5         case FIRST_STATE:
6             state = SECOND_STATE;
7 }
```

Listing 46: Global variable modified with parameter as reference

```
1 void foo(TYPE_OF_STATE &state) {
2     switch (state):
3         case FIRST_STATE:
4             state = SECOND_STATE;
5     }
6 }
```

Hidden warnings with Eclipse

The project utilised Eclipse as an IDE. Eclipse integrated gtest easily but it did not show warnings created by gtest. Instead, the warnings were only able to be seen if the test executable was ran directly from command line. Warnings occurred when the calls to the test doubles were made without defining them to be expected with the *EXPECT_CALL* macros. The reason might have been a setting in Eclipse that was not able to be found.

Ordered function calls

Some tests required to verify that mocked functions or methods were called in a particular order. For example, one test had to check that motor was first enabled, before it was tried to be started. The order of the calls is not automatically checked by gmock but it can be set to verify them. The intended order was able to be set in multiple ways. The simplest approach was to set the test to ensure that each call was performed in a defined order. Such a scenario is shown in Listing 47, where the order was set on the line 3 with the *InSequence seq* declaration. The declaration required the calls then to be executed in the order specified in the test.

Listing 47: Ensuring the correct order of the execution of mocked methods

```
1 TEST_F(FooTest, StartMotor)
2 {
3     InSequence seq;
4     EXPECT_CALL(foo, enable);
5     EXPECT_CALL(foo, start);
6     foo.advance();
7 }
```

In addition, only certain calls were able to be set to be required to be called in a particular order. It was achieved by first declaring an expectation variable as shown in Listing 48 on the line 3 for the *enable* method of the *foo* class. Then, the expectation for the second call was set on the line 4, followed by the *After* clause. The expectation variable was given for the *After* clause, which set the *start* method to be required to be called after the *enable* method has been called. The Listing 47 and Listing 48 were essentially different implementations for the same functionality. The difference was that if there would be a third method call, it would have required to be called in the given order in the first example but not in the second example.

Listing 48: Ensuring the correct order of the execution of mocked methods

```
1 TEST_F(FooTest, StartMotor)
2 {
3     testing::Expectation exp = EXPECT_CALL(foo, enable);
4     EXPECT_CALL(foo, start)
5         .After(exp);
6     foo.advance();
7 }
```

Test code duplication

In order to reduce code duplication, parameterised tests were used. It was also noted that previously created test fixtures should be needed as the parameterised tests also required the use of initialised parameters and classes.

Parameterised tests helped to reduce code duplication. When a state machine was tested it had two tests, both representing tests for different states but essentially for the same functionality. Example of one of them is shown in Listing 49. In both tests the state two digital inputs were read and if either input was detected to be high, the state ought to advance to the next. The difference of the tests was that the verification of start and end states were different. The tests required three scenarios for both; one for each input to be detected and one for both of them to be detected at the same time. As one can see, many of the code lines seemed to be duplicated, both inside the tests as when compared to each other.

Duplication was reduced with the parameterised tests of gtest. The test macro was changed from *TEST_F* to *TEST_P* and hard coded zeros and ones were were replaced with parameters. The parameters were received with the function *GetParam* of gtest. In addition, a struct was created to combine the two parameters as only one parameter was able to be passed for the parameterised test. Also, a new class was written, which inherited from the same test fixture used in the first version and from the *WithParamInterface* object of gmock, that took the created struct as parameter. Lastly, the tests needed to be instantiated with the particular parameter values with the *INSTANTIATE_TEST_SUITE_P* macro. An example of the implementation is shown in Listing 50.

While additional setup was required, the final result led to less duplication in the tests and enabled an easy generation for the second test as only a new test case with *TEST_P* was required to be created. It was also considered that the start and end state could also have been parameterised to let only one parameterised test handle all three scenarios of both previous tests. It was not deemed to be worthwhile as otherwise the naming of the parameterised tests would not have expressed true functionality of both test cases leading to less informative tests for regression testing.

Listing 49: Code duplication in test cases

```
1  TEST_F(ExampleFixture, InputIsDetected) {
2      foo.state = foo.FIRST_STATE;
3      EXPECT_CALL(mockDigital1, read)
4          .WillOnce(testing::Return(0));
5      EXPECT_CALL(mockDigital2, read)
6          .WillOnce(testing::Return(1));
7      foo.advance();
8      EXPECT_EQ(foo.state, foo.SECOND_STATE);
9
10     foo.state = foo.FIRST_STATE;
11     EXPECT_CALL(mockDigital1, read)
12         .WillOnce(testing::Return(1));
13     EXPECT_CALL(mockDigital2, read)
14         .WillOnce(testing::Return(0));
15     foo.advance();
16     EXPECT_EQ(foo.state, foo.SECOND_STATE);
17
18     foo.state = foo.FIRST_STATE;
19     EXPECT_CALL(mockDigital1, read)
20         .WillOnce(testing::Return(1));
21     EXPECT_CALL(mockDigital2, read)
22         .WillOnce(testing::Return(1));
23     foo.advance();
24     EXPECT_EQ(foo.state, foo.SECOND_STATE);
25 }
```

Listing 50: Code duplication reduced using parameterised tests

```
1  struct Inputs {
2      Inputs(bool in1, bool in2) :
3          input1(input1), input2(input2) {}
4      bool input1, input2;
5  };
6
7  class ParameterisedExample : public ExampleFixture,
8      public ::testing::WithParamInterface<Inputs> {};
9
10 TEST_P(ParameterisedExample, InputIsDetected) {
11     auto inputs = GetParam();
12     foo.state = modes.FIRST_STATE;
13     EXPECT_CALL(mockDigital1, read)
14         .WillOnce(testing::Return(inputs.input1));
15     EXPECT_CALL(mockDigital2, read)
16         .WillOnce(testing::Return(inputs.input2));
17     foo.advance();
18     EXPECT_EQ(foo.state, foo.SECOND_STATE);
19 }
20
21 INSTANTIATE_TEST_SUITE_P(ExampleFixture,
22         ParameterisedExample, ::testing::Values(
23     Inputs(0, 1),
24     Inputs(1, 0),
25     Inputs(1, 1)
26 ));
```

Naming of parameterised tests was achieved by providing a fourth argument for the *INSTANTIATE_TEST_SUITE_P* macro. As instructed in the documentation [54], it was created as lambda function. An important aspect to note is that gtest required the test names to only contain alphanumeric ASCII characters. In the case of gtest, alphanumeric referred to lower and uppercase letters and numbers. The usage of other characters led to errors that are difficult, from which the cause was not trivial to figure out.

Testing array contents

As a more advanced example, testing of data transmission via SPI is shown in Listing 51. Firstly, a global *sentData* array had to be declared as shown on the line 1. Then, a self-created test function named *saveContents* was created on the lines 3 to 8. It had four parameters, though only the second of them was utilised and the rest of the parameters only required their type to be defined without the argument.

The function saved the contents of the *pData* pointer to the global *sentData* array. The test declared first the *Hal* class on the line 12 and provided it as an argument for the *Foo* class on the line 13. Then, the data to be transmitted was declared on the line 14. The expectation for the *SPI_Transmit* method was set on the lines 16 to 21. Firstly, only the third argument, which represented the length of the message, was checked. Secondly, the *DoAll* macro was utilised with the *Invoke* macro to allow calling of the *saveContents* function with the same arguments as the *SPI_Transmit* method was called. The reason, why the *saveContents* function only had to define types without arguments for the first, third and fourth parameters was that the second argument contained the information to be transmitted and the rest were not needed. Then, the *transmit* method of the class *Foo* was called on the line 22 and finally, the contents of the global *sentData* array were checked by comparing the contents against the *ElementsAre* macro with the *EXPECT_THAT* macro on the line 23.

Listing 51: Testing contents of an array that is in the scope of the function

```
1  uint8_t sentData[3] = { 0 };
2
3  void saveContents(SPI_HandleTypeDef*, uint8_t *pData,
       uint16_t, uint32_t) {
4      for (uint8_t i = 0; i < sizeof(sentData); i++) {
5          sentData[i] = *pData;
6          pData++;
7      }
8  }
9
10 TEST(ArrayTest, TransmitViaSPI)
11 {
12     HAL hal;
13     Foo foo(hal);
14     uint32_t data = 0x112233;
15
16     EXPECT_CALL(hal, SPI_Transmit(_,_,3,_)
17         .WillOnce(
18             DoAll(
19                 Invoke(saveContents)
20             )
21         );
22     foo.transmit(data);
23     EXPECT_THAT(sentData, ElementsAre(0x11, 0x22, 0x33));
24 }
```