Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di II livello in
Informatica

An Empirical Assessment on the Effectiveness of Test-Driven Development Techniques for Embedded Systems

Supervisors
Giuseppe Scanniello
Simone Romano

Candidate
Michelangelo Esposito

Academic Year 2021-2022

Abstract

In this thesis, we define two experiments (*i.e.*,, a baseline experiment and its replication) and analyze their results. The goal of these experiments is to compare Test-Driven Development (TDD), an incremental approach to software development where tests are written before production code, with a traditional way of coding where production code is written before tests (*i.e.*,, NO-TDD). TDD has been the subject of numerous studies over the years, with the purpose of determining whether applying this technique would result in an improved development. In this study, TDD and NO-TDD have been contrasted in the context of the implementation of embedded systems (ESs). The goals of the replication study are to validate the results of the baseline experiment and to generalize them to a different and more real setting. The most remarkable differences concern the implementation task and the experimental procedure: as for the task, we asked the participants to implement a larger and more complex ES in the replicated experiment; as for the procedure, the implementation task was not accomplished in the replicated experiment under controlled conditions and the developed ES (*i.e.*,, the result of such task) was then deployed and tested on a real software/hardware environment. The Participants in the experiments were final year students in Computer Science enrolled to the *Embedded Systems* course at the University of Salerno, in Italy. Before taking part in the experiment, they were trained on concepts spanning from unit testing and its guidelines, to the introduction of TDD. The results obtained by an aggregated analysis of data gathered from both the baseline and the replicated experiments suggest that there is not a significant difference between TDD and NO-TDD on: productivity, number of written tests, a... . On the other hand we observed a significant on: ... When analyzing experiments individually, we observed that ...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A computer hardware and software combination created for a particular purpose is an Embedded System (ES). In many cases, ESs operate as part of a bigger system (*e.g.,* agricultural and processing sector equipment, automobiles, medical equipment, airplanes, and so on) and within tight resource constrains (*e.g.,* small battery capacity, limited memory and CPU speed, and so on). The global ESs market is expected to witness notable growth. A recent report evaluated the global ESs market 89.1 billion dollars in 2021, and this market is projected to reach 163.2 billion dollars by 2031; with a compound annual growth rate of 6.5% [1]. This growth is mostly related to an increase in the demand for advanced driver-assistance system (in electric and hybrid vehicles) and in the number of ESs-related research and development projects.

Test-Driven Development (TDD) is an incremental approach to software development in which a developer repeats a short development cycle made up of three phases: *red*, *green*, and *blue/refactor* [2]. During the red phase, the developer writes a unit test for a small chunk of a functionality not yet implemented and watches the test fail. In the green phase, the developer implements the chunk of functionalities as quickly as possible and watches all the unit tests pass. During the refactoring phase, the developer changes the internal structure of the code while paying attention to keep the functionality intact—accordingly, all unit tests should pass. TDD has been conceived to develop "regular" software, and it is claimed to improve software quality as well as developers' productivity [3]. ESs have all the same challenges of non-embedded systems (NoESs), such as poor quality, but add challenges of their own [6]. For example, one of the most cited differences between embedded and NoESs is that embedded code depends on the hardware. While in

principle, there is no difference between a dependency on a hardware device and one on a NoESs [4], dealing with hardware introduces a whole new set of variables to consider during development. Furthermore, the limited resources on which an ES usually operates may make it extremely difficult to properly test the system in its entirety. Over the years, a huge amount of empirical investigations has been conducted to study the claimed effects of TDD on the development of NoESs (*e.g.,*, [5]). So far no investigations have been conducted to assess possible benefits concerned the application of TDD on the development of ESs although some authors, like Greening [4], believes that embedded developers can benefit from the application of TDD in the development of ESs.

In this work of thesis, we investigate the following primary research question (RQ):

RQ. To what extent does the use of TDD affect the external quality and productivity of the developed ES?

To answer this RQ, we present the results of an exploratory empirical assessment constituted of two experiment, a controlled experiment that acts as the baseline for the study, and its replication to study the application of TDD on the implementation of ESs. The participants were final year Master's degree students in Computer Science enrolled to an ES course at the University of Salerno, in Italy. The goal of these experiments is to increase the body of knowledge on the benefit (if any) related to the application of TDD in the context of the development of ESs. To that end, we compare TDD with respect to a more traditional, test-last, development practice, where test cases are written after the production code. From here onwards, we refer to this traditional way of coding as NO-TDD. Whichever the used approach (TDD and NO-TDD), the participants in the implementation of an ES where asked to use a mock to model and confirm the interactions between a device driver and the hardware. The mocked implementation would intercept commands to and from the device simulating a given usage scenario. In the replication experiment, the participants had to implement an additional ES with the end goal of replacing the mocks with real hardware components (a number of sensors and actuators) before deploying the ES on the actual hardware platform they had mocked up to that point. The logic of the ES was deployed on a Raspberry Pi model 4 board, and the developed test cases were executed in the real environment. Variations in the replicated experiments (task and the experimental procedure) were introduced to validate the results of the baseline experiment and to generalize these results to a

more real setting. The results obtained by an aggregated analysis of the data from both the experiments suggest that there are no significant differences between TDD and NO-TDD on: productivity, number of written tests, a..., respectively. On the other hand, we observed a significant difference on: ... When analyzing experiments individually, we observed that ...

As for the following thesis structure, it is made up of five chapters: The first two act as the foundational knowledge concepts that provide a general overview on the main topics concerning the TDD methodology and ES technologies: chapter one contains an overview on the software testing process, analyzing the main approaches, with a focus on TDD. Chapter two will provide information on the general ESs concepts, enabling technologies and implementation challenges, before discussing the techniques for testing such systems. In chapter three, we conduct a review of the literature by examining the most relevant previous empirical studies on the application of TDD and the current testing methodologies for ES. Chapter four will contain the detail explanation of our approach for the definition of the two experimental studies, the analysis of the results, and our answers to the research questions. Finally, chapter five will provide the conclusions to the thesis, along with a discussion on the possible ramification the research could manifest when moving forward in the analysis of TDD for ES development.

Chapter 2

Test Driven Development

2.1 Overview on software testing

Software testing is an essential part of the development process and lifecycle of a system, as it helps to verify that an implemented solution is reliable and performs as intended in most situations. Testing can be defined as the process of finding differences between the expected behavior specified by the system's requirements and models, and the observed behavior of the implemented software; unfortunately, it is impossible to completely test a non-trivial system. First, testing is not decidable; second, testing must be performed under time and budget constraints [6], therefore testing every possible configuration of the parameters of a system is unfeasible. Today, developers often compromise on testing activities by identifying only a critical subset of features to be tested.

There are many approaches to software testing, including unit testing, integration testing, system testing, as well as performance, penetration and acceptance testing. Each of these approaches has its own specific goals and methods, as well as a different suite of tools built to support them and ensure that the testing process is always consistent, its execution is easily automated, and the test outcomes are always clear; furthermore, they are often used in combination with each other to ensure that a software product is thoroughly tested and conform to its specification. More in detail, the main testing techniques are:

- **Unit Testing** is a method of testing individual units or components of a software product in isolation; its goal is to verify that each unit of code is working correctly and meets the specified requirements. These kinds of tests are usually written by the developers who also wrote

the corresponding production code, and they are run automatically as part of the build process. Techniques also exist to generate input configuration for unit test automatically, by searching amongst the input space for the program.

- **Integration Testing** is a method of testing how different units or components of a software product work together. The goal of integration testing is to ensure that the different parts of the system are integrated correctly and that they function as expected when combined. Integration tests are typically more complex than unit tests, as they involve multiple units of code working together.

- **System Testing** is a method of testing a complete software product in a simulated or real-world environment. The goal of system testing is to ensure that the software meets the specified requirements and performs as expected when running in a real-world environment. System tests may involve testing the software on different hardware or operating systems, or with different data inputs and configurations.

- **Acceptance Testing** is a method of testing a software product to ensure that it meets the needs and expectations of the end user. The goal of acceptance testing is to verify that the software is fit for its intended purpose and that it meets the requirements of the user. Acceptance tests are often written by the end user or a representative of the end user, and they may involve testing the software in a real-world environment

- **Performance Testing** is the process of evaluating a system's performance in terms of responsiveness and stability under a particular workload; it is usually done to determine how a system behaves in terms of various inputs and how it responds to different levels of traffic. ... Used to test availability, reliability, and other parameters.

- **Penetration Testing** is the practice of testing a system, network, or application with the objective of identifying vulnerabilities that an attacker could exploit. This security evaluation of the system happens by simulating an attack and identifying any weaknesses that could be exploited by a malicious party. Penetration testing can be conducted by both internal and external security teams and is often used as a means to identify and remediate any potential security risks.

2.1.1 Software Development Lifecycle

Before discussing how testing activities are performed in more detail, it is essential to introduced how testing is integrated in the development process. The term Software Development Lifecycle (SDLC) refers to the entire process of developing and maintaining software systems, from the initial concept, to its end-of-life period.

One of the first SDLC models introduced in software engineering is the Waterfall model; it is a linear, sequential approach to software development in which there is a strict, marked division between the different phases, such as requirements gathering, design, implementation, testing, and maintenance. The main weakness of this model is that it does not allow for much iteration or flexibility: once a phase is completed, it is difficult to go back and make changes to earlier phases; this can lead to a very long feedback cycle between requirements specification and system testing, resulting in wasted time and resources in case a design flaw is not discovered until later in the development process. Additionally, the Waterfall model assumes that all requirements can be fully gathered and understood at the beginning of the project; such a simplification is often not applicable in modern software development, where ever-evolving requirements and functionalities are the norm. Finally, the model does not account for the fact that testing and deployment are ongoing processes, not a single event at the end of the project. Figure 2.1 highlights the main phases of the Waterfall model.

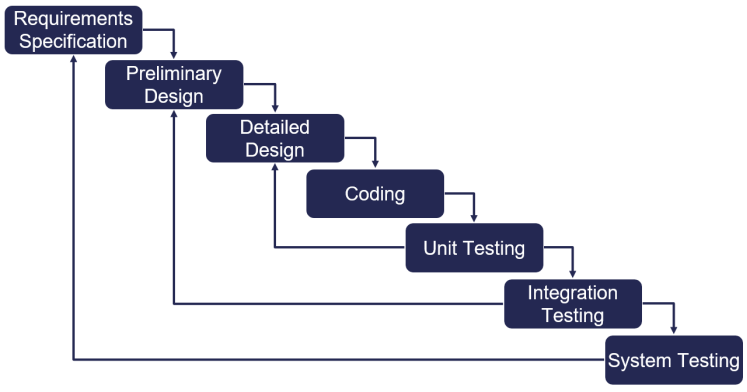


Figure 2.1: The Waterfall model

Modern software development strays away from non-incremental models, as today's applications are continuously evolving and adapting; instead, iterative approaches are preferred, where the sequential chain of the Waterfall

model is replaced by a cyclical process during which the development team goes through multiple iterations or cycles of planning, designing, building, testing, and evaluating the product.

A key example is the Agile SLDC model, which values flexibility and collaboration, and prioritizes customer satisfaction and working software over strict plans and documentation. One of the main principles of Agile development is the use of small, cross-functional teams that work together to deliver working software in short sprints or iterations: this allows for frequent feedback and adjustments to be made throughout the development process between the clients and the development teams, rather than waiting until the end of a project to make changes. Figure 2.2 highlights the phases of the Agile process:

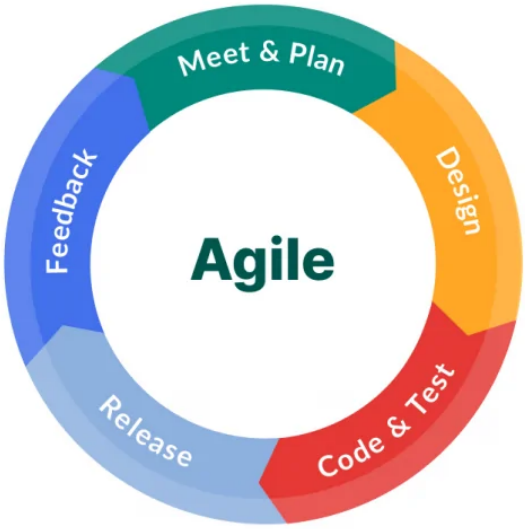


Figure 2.2: The Agile model

The main steps performed during each of these phases are summarized below:

- **Design**: in this phase, the team designs the architecture, user interface and overall functionality of the software; the design process is iterative and collaborative, with the team working closely with customers, as well as with the stakeholders, with the objective to ensure that the software meets the agreed-upon needs.
- **Code & Test**: in this phase, the team writes the code for the software

and performs testing to ensure that it is functioning correctly; agile development places a strong emphasis on automated testing, which allows for quick feedback on the quality of the delivered code modules.

- **Release**: the team releases the software to customers and stakeholders for feedback; this allows the team to gather feedback on the software and make any necessary adjustments before the final release.
- **Feedback**: the team reviews the feedback received from customers and stakeholders and makes any necessary changes to the software. Feedback is incorporated into the development process in an iterative manner, allowing the software to continuously improve over time.
- **Meet & Plan**: the team meets to plan the next iteration of development, reviews the progress made in the previous iteration, sets goals for the next iteration, and assigns tasks to team members. The team also reviews and adjusts the development plan as needed to ensure that the software is on track to meet the customers' needs.

2.2 Test-Driven Development

2.2.1 Overview

Unit testing is arguably the most used testing technique since by itself it can already provide a general assessment of the quality and reliability of a software solution. TDD is a software development approach that builds on top of the concept of unit testing; it was firstly introduced in 2003 by Kent Back in the book "Test-Driven Development By Example" [2]: while there is no formal definition of the process, as the author states, the goal is to "write clean code that works". Compared to traditional SDL processes, TDD is an extremely short, incremental, and repetitive process, and is related to **test-first programming** concepts in agile development and extreme programming; this advocates for frequent updates/releases for the software, in short cycles, while encouraging code reviews, unit testing and incremental addition of features.

At its core, TDD is made up of three iterative phases: "Red", "Green" and "Blue" (or "Refactor"):

- In the "**Red**" phase, a test case is written for the chunk of functionality to be implemented; since the corresponding logic does not exist yet, the test will obviously fail, often not even compiling.

- In the "**Green**" phase, only the code that is strictly required to make the test pass is written.
- Finally, in the "**Blue**" phase, the implemented code, as well as the respective test cases, is refactored and improved. It is important to perform regression testing after the refactoring to ensure that the changes didn't result in any unexpected behaviors in other components.

Each new unit of code requires a repetition of this cycle [7].

The figure below provides a representation of the TDD cycle:

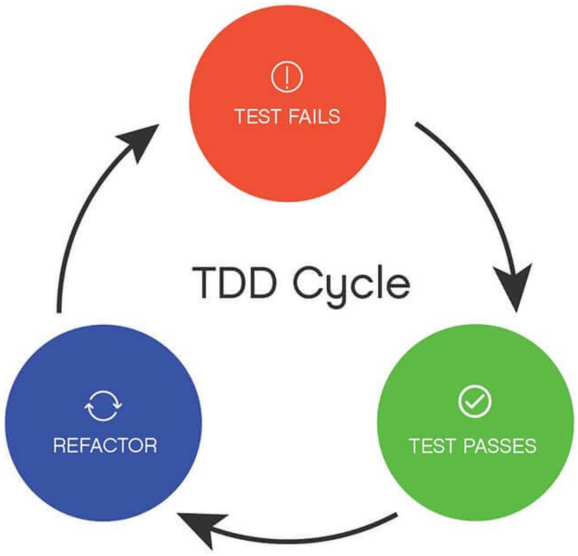


Figure 2.3: The Test Driven Development cycle

As previously stated, each TDD iteration should be extremely short, usually spanning from 10 to 15 minutes at most; this is possible thanks to a meticulous decomposition of the system's requirements into a set of **User Stories**, each detailing a small chunk of a functionality specified in the requirements. These stories can then be prioritized and implemented iteratively.

User stories can vary in granularity: when using a fine-grained structure when describing the task, this can be broken up into a set of sub-tasks, each corresponding to a small feature; on the other hand, with coarser-grained tasks, this division is less pronounced [8]. Even when the same task is considered, the outcome of the TDD process will change depending on the

level of granularity employed when describing it; there is no overall right or wrong approach, rather it is something that comes from the experience of the developer to break tasks into small work items [8].

The general mantra of TDD revolves around the "Make it green, then make it clean" motto

2.2.2 TDD advantages

The employment of TDD can result in a series of benefits during the development process, such as:

- **Regression testing**: by incrementally building a test suite as the different iterations of TDD are performed, we ensure that the system
- **Very high code coverage**: coverage is a metric used to determine how much of the code is being tested; it can be expressed according to different criteria such as statement coverage, i.e., how many statements in the code are reached by the test cases, branch coverage, i.e., how many conditional branches are executed during testing, or function coverage, i.e., how many functions are executed when running the test suite. While different coverage criteria result in different benefits, by employing TDD we ensure that any segment of code written has at least one associated test case.
- **Improved code quality**: as we are specifically writing code to pass the tests in place, and refactoring it after the "Green" phase, we ensure that the code is cleaner and overall more optimized, without any extra pieces of functionalities that may not be needed.
- **Improved code readability and documentation**: test act as documentation...
- **Simplified debugging and early fault detection**: Whenever a test fails it becomes obvious which component has caused the fault: by adopting this incremental approach and performing regression testing, if a test fail we will be certain that the newly written code will be responsible. For this reason, faults are detected extremely early during the testing process, rather than potentially remaining hidden until the whole test suite has been built and executed.

Chapter 3

Embedded Systems

3.1 Overview

ESs can be defined as a combination of hardware components and software systems that seamlessly work together to achieve a specific purpose; they can be dynamically programmed or have a fixed functionality set, and are often engineered to achieve their goal within a larger system. They are commonly found inside devices that we use on a daily basis, such as cell phones, traffic lights, and appliances; here, these systems are responsible for controlling the functions of the device, and they are required to work continuously without the need for human intervention, besides the occasional battery replacement/recharge. This requirement implies that, in most circumstances, providing maintenance to ESs is challenging or straight up unfeasible; therefore, the design process of a system of this kind must account for a series of additional challenges and constraints, that are typically not considered as much in NoESs, in order to ensure their ability to operate in a stand-alone manner and in a wide range of conditions. Many ESs, in fact, are deployed into physical environments that do not have access to a network or are not covered by an internet connection, or are even subject to harsh and adverse weather conditions. Furthermore, many ESs are used in applications that require a high degree of safety and security, such as in the aerospace or medical industries, meaning that the system must be able to operate without fail and without compromising safety.

Despite the many challenges, in recent years, ESs have seen a steep surge in popularity, and have driven innovation forward in their respective areas of interest: everywhere, spanning from the agricultural field, to the medical and energy ones, ESs of various size and complexity are employed, especially

in areas where human intervention is impractical or straight up impossible. As the demand for more advanced and sophisticated devices continues to increase, the role of ESs will only become more prominent.

This requires the use of embedded software, which is specifically designed to run on the limited hardware of the system.

There are many types of ESs, including microcontrollers, digital signal processors (DSPs), and field-programmable gate arrays (FPGAs). Each of these types of systems has its own unique characteristics and is suited to different types of applications.

3.2 Enabling technologies

3.2.1 Microcontrollers

One of the key enabling technologies for ESs is their microcontroller, which is a small single-chip computer that is used to manage the functions of the larger system, in a power-efficient way and ideally at a low cost. While some applications require their own custom-made microcontroller and other custom hardware components built ad-hoc for them, there is a wide variety of general-purpose microcontroller boards, sensors, and actuators that are far easier to program and can also be customized for a high range of applications, which makes them highly versatile.

One type of general-purpose microcontroller that is commonly used for ESs development, as well as for many other IoT purposes, is the Arduino; it is an open-source platform that is based on the Atmel AVR microcontroller; it is widely used in hobbyist and educational projects because of its simplicity and low cost. Many Arduino versions exist, each with a different form factors, amounts of resources on board and I/O pins; as a result they used for applications of increased complexity and needs. Some examples include the Arduino Nano, with the smallest form factor among the Arduino boards and very limited, the Uno, and the Mega.

Another popular general-purpose microcontroller platform is the Raspberry Pi, a small single-board computer that is based on the ARM architecture, which is also what many mobile processors are based on. It comes in different variants, from an Arduino Nano-sized Raspberry Pi Zero and Zero W, but equipped with much more resources, to the Pi 3 and 4 models, which are effectively capable of running a more complex OS, supporting even up to 8GB of RAM.

In addition to these general-purpose microcontrollers, there are also many proprietary devices that are designed for specific applications: given this

high specialization, these microcontrollers often offer limited customization capabilities, and may not be easily programmable, if at all, by the user. Some examples of proprietary microcontrollers include the Microchip PIC and the Texas Instruments MSP430; these chips are often used in industrial and commercial applications, where a high level of performance and reliability is required. They may also be used in applications where security is a concern, as their design may be kept confidential to protect against tampering or reverse engineering attempts.

Overall, the choice of microcontroller for an ES will depend on the specific requirements of the user. General-purpose microcontrollers such as Arduino and Raspberry Pi may be suitable for hobbyist or educational projects, while proprietary microcontrollers may be better suited for industrial or commercial applications where performance and reliability are critical.

3.2.2 Embedded software and communication protocols

Software-wise, more complex ESs can be equipped with their own **Embedded Operating System** (EOS), specifically designed to run on embedded devices, as they have a smaller footprint and fewer features compared to general-purpose OSs like Windows or Linux, and are optimized for low power consumption. Popular EOSs include TinyOS and Embedded Linux. Furthermore, the real-time requirements of some ESs calls for their own **Real-Time Operating System** (RTOS); these are specialized OSs that are designed to provide a predictable response time to events, even when there are many tasks running concurrently. RTOS are essential for ES that require fast and reliable performance, such as in aircraft control systems or medical devices. A notable and open-source example is FreeRTOS.

At the lower level, communication between different components is crucial for the proper functioning of the system; there are various communication protocols that allow transmission and exchange of data, with the most notable ones being UART, I2C, and SPI:

- **Universal Asynchronous Receiver/Transmitter (UART)**: one of the simplest protocols used for serial communication between devices; it is full-duplex, which means that data can be transmitted and received at the same time. UART is widely supported by many microcontrollers and microprocessor devices, however, it is typically slower than other communication protocols and can be less reliable over longer distances.

- **Inter-Integrated Circuit (I2C)**: a two-wire communication protocol that is used for communication between devices on the same circuit

board; it is a half-duplex protocol, so data can only be transmitted or received at a time. I2C is commonly used to connect devices such as sensors, displays, and memory to a microcontroller. It is relatively fast and can support multiple devices on the same bus; however, it still has a limited data rate.

- **Serial Peripheral Interface (SPI)**: a two-wire communication protocol used for communication between devices; it is a synchronous protocol, which means that data is transmitted and received in a coordinated manner. SPI is relatively fast and can support multiple devices on the same bus, however, it requires more wires and pins compared to I2C and as a reseult can be more complex to implement.

Besides communication between different components sharing the same circuit, multiple devices are often deployed as part of a larger system and must be able to efficiently communicate between each other to achieve their purpose; therefore, it is essential for ESs to be equipped with a robust suite of wireless communication protocols. As always, determining which protocol to employ depends on the constraints the system is dealing with, such as being limited to a low power consumption or being required to maintain a low-latency communication.

Zigbee is a wireless communication protocol specifically designed and built for low-power, low-data-rate applications [9]; it is often used in sensors and other devices that need to communicate over short distances, such as in home automation systems or industrial control systems. Its very low power consumption makes it so ZigBee is one of the most well-suited protocols for use in devices that need to operate for long periods of time without access to a power source (*i.e.,* a quite substantial subset of ESs). Bluetooth is another wireless protocol that is commonly used in ES which was designed for medium-range communication and is most commonly used to connect devices such as phones, tablets, and laptops to other devices, such as speakers, keyboards, or headphones. Bluetooth is a widely supported standard and is often used in applications where compatibility with a range of different devices is important; furthermore, with its low-energy variant, Bluetooth can help further optimize power consumption in devices that require it. LoraWAN and SigFox on the other hand, are two Low-Power, Wide-Area (LPWA) communication protocol designed for IoT and machine-to-machine (M2M) applications; a typical use case is the transmission of small amounts of data over long distances, making them well-suited for use in remote monitoring systems or other applications where conventional communication methods are not practical. For network communications, IP, and it's low

energy version 6LoWPAN, are widely used protocols; they are exercised for transmitting data between devices at the network level and are a key component of the Internet. Finally, at the application level, lightweight messaging protocols such as the Message Queue Telemetry Protocol (MQTT) are a common choice.

3.3 Design and challenges

From a design and development standpoint, working with ESs can be complex, as it involves a wide range of skills and disciplines, including computer science, electrical engineering, and mechanical engineering. It is often necessary to work closely with other team members, including hardware and software designers, to ensure that the system meets all of its requirements, functional and especially non-functional.

Going through multiple hardware revisions in order to meet the requirements can be extremely expensive.

Failures in ESs should always be evident and identifiable quickly (a heart monitor should not fail quietly) [10]. Given the high criticality of such systems, ensuring their dependability over the course of their lifespan is essential; ESs can be deployed in extreme conditions (*i.e.,* weather monitoring in extreme locations of the planet, satellite managements systems, devices inside the human body, and so on), where maintenance operations cannot be performed regularly, and high availability is expected.

The dependability of an ES can be expressed in terms of:

- **Maintainability**: the extent to which a system can be adapted/modified to accommodate new change requests. As ESs becomes more complex and feature-rich, it is becoming increasingly important to design them with maintainability in mind. This includes designing systems that are easy to update and repair, as well as ensuring that they can be easily replaced if necessary.

- **Reliability**: the extent to which a system is reliable with respect to the expected behavior.

- **Availability**: the extent to which a system remains available for its users.

- **Security**: the extent to which a system can keep data of its users safe and ensures the safety of their users.

These dependability attributes cannot be considered individually, as there are strongly interconnected; for instance, safe system operations depend on the system being available and operating reliably in its lifespan. Furthermore, an ES can be unreliable due to its data being corrupted by an external attack or due to poor implementation. As a result of particular care should be applied in the design of these systems, especially ...

An important, and often critical, aspect of ES, which defines the greatest challenges when engineering them, is the limited quantity resources available: these systems often have very small amounts of memory and processing power, so it is important to carefully design the software to make the most efficient use of them. This can involve using specialized programming languages and techniques, such as RTOSs and low-level hardware access. Furthermore, many such systems may be powered by using a battery, and thus the hardware they are equipped with, often purpose built, must be highly efficient in its operations. Finally, from the software point-of-view, it is essential that the system operates deterministically and with real-time constrains.

Another notable challenge is the requirement of some systems to perform real-time processing tasks. This means that they must be able to process data and provide a response within a specific time frame. For example, an ES in a car might be responsible for controlling the engine and transmission; it must be able to process data from sensors and make decisions about how to control the engine and transmission in real-time, as the car is being driven.

In addition to the technical challenges, there are also many non-technical factors that must be considered when developing ESs. For example, the system must be able to operate within the physical constraints of the device it is being used in, and it must be able to withstand the environmental conditions in which it will be used.

In conclusion, ESs are specialized computer systems that are designed to perform a specific task within a larger system; they are able to perform real-time processing and operate in a stand-alone manner, but they also face the challenge of limited resources and power. Despite this, the use of ESs has grown significantly, and they are now found in a wide variety of applications.

3.4 Testing Embedded Systems

Testing ESs poses a series of additional challenges compared to traditional systems: first, in the case of ESs that are highly integrated with a physical environment (such as with Cyber Physical Systems, CPSs), replicating the exact conditions in which the hardware will be deployed may be difficult;

additionally field-testing of these systems can be unfeasible to dangerous or impractical environmental conditions (*e.g.,* a nuclear power plant, a deep-ocean station, or the human body). Secondly, given the absence of a user interface in many cases, the lack of immediate feedback makes the outcomes of the tests less observable. Moreover, resource constraints may not allow developers to fully deploy the testing infrastructure on the target hardware and thus slowing down further the process by impeding or delaying automation and regression testing. The hardware and software heterogeneity proper of ESs can also make it difficult to test them in a consistent and repeatable way. Finally, the testing of time-critical systems has to validate the correct timing behavior which means that testing the functional behavior alone is not sufficient.

3.4.1 X-in-the-loop

To mitigate some of these issues and approach ES testing in an incremental manner which allows engineers to only focus on one aspect at a time, the general testing process of ESs follows the X-in-the-loop paradigm [11], according to which the system goes through a series of steps that simulate its behavior with an increased level of detail, before being effectively deployed on the bare hardware; subcategories in this area include Model-in-the-Loop, Software-in-the-Loop, Processor-in-the-Loop, and Hardware-in-the-Loop:

- With **Model-in-the-Loop (MIL)** or **Model-Based Testing** an initial model of the hardware system is built in a simulated environment; this coarse model captures the most important features of the hardware system by using mathematical models [12]. As the next step, the controller module is created, and it is verified that the controller can manage the model, as per the requirements. Commonly, after the testers establish the correct behavior of the controller, its inputs and outputs are recorder, in order to be verified in the later stages of testing.

- With **Software-in-the-Loop (SIL)**, the algorithms that define the controller behavior are implemented in detail, and used to replace the previous controller model; the simulation is then executed with this new implementation. This step will determine whether the control logic, *i.e.,* the controller model can be actually converted to code and, perhaps more importantly, if it is hardware implementable. Here, the inputs and outputs should be logged and matched with those obtained in the previous phase; in case of any substantial differences, it may be necessary to backtrack to the MIL phase and make the necessary

changes, before repeating the SIL step. On the other hand, if the performance is acceptable and falls within the acceptance threshold, we can move to the next phase.

- The next step is **Processor-in-the-Loop (PIL)**; here, an embedded processor, the one with which the microcontroller on the target hardware is equipped, will be simulated in detail and used to run the controller code in a closed-loop simulation. This can help determine if the chosen processor is suitable for the controller and can handle the code with its memory and computing constraints. At this point, developers have a general idea about how the embedded software will run on the hardware.

- Finally, **Hardware-in-the-loop (HIL)** is the step performed before deploying the ES to the actual target hardware. Here, we can run the simulated system on a real-time environment, such as SpeedGoat [13]. The real-time system performs deterministic simulations and has physical connections to the embedded processor, *i.e.,* analog inputs and outputs, and communication interfaces, such as CAN and UDP: this can help identify issues related to the communication channels and I/O interface. HIL can be very expensive to perform and in practice it is used mostly for safety-critical applications. However, it is required by automotive and aerospace validation standards.

After all these steps, the system can finally be deployed on the real hardware. A common environment for performing the simulation steps discussed above is Simulink [14]; it is a graphical modeling and simulation environment for dynamic systems based on blocks to represent different parts of a system: a block can represent a physical component, a function, or even a small system. Some notable features include: scopes and data visualizations for viewing simulation results, legacy code tool to import C and C++ code into templates and building block libraries for modeling continuous and discrete-time systems.

Figure 3.1 ...

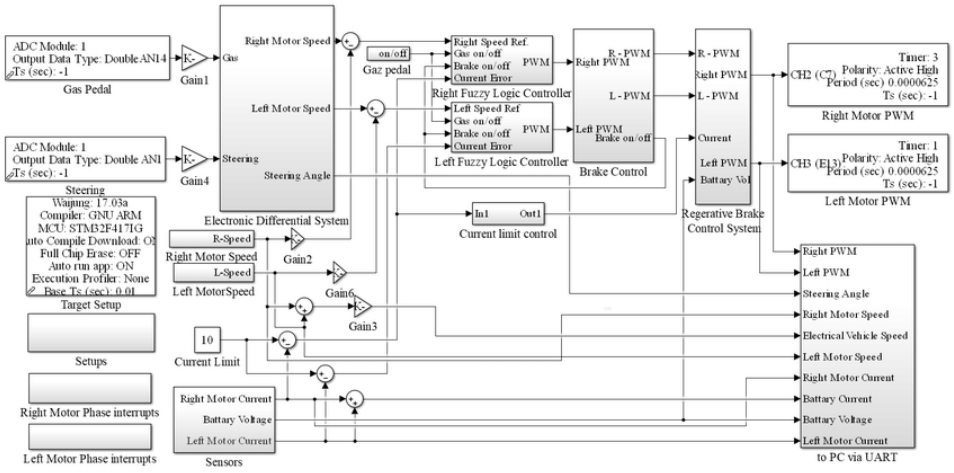


Figure 3.1: Some components of a Simulink model for an autonomous driving system

3.4.2 A TDD pipeline for Embedded Systems

Often, quality in embedded software is generally tied to platform-specific testing tools geared towards debugging [15]. ...

Applying TDD practices to ESs could potentially result in a series of benefits:

- ...
- Reduce risk by verifying production code, independent of hardware, before hardware is ready or when hardware is expensive and scarce.
- Reduce the number of long target compile, link, and upload cycles that are executed by removing bugs on the development system.
- Reduce debug time on the target hardware where problems are more difficult to find and fix.
- Isolate hardware/software interaction issues by modeling hardware interactions in the tests.
- Improve software design through the decoupling of modules from each other and the hardware. Testable code is by necessity, modular.

In [4], the author proposes the "Embedded TDD Cycle", as a pipeline made of the following steps:

1. **TDD micro-cycle**: this first stage is the one run most frequently, usually every few minutes. During this stage, a bulk of code is written in TDD fashion, and compiled to run on the host development system: doing so gives the developer fast feedback, not encumbered by the constraints of hardware reliability and/or availability, since there are no target compilers or lengthy upload processes. Furthermore, the development system should be a proven and stable execution environment, and usually has a richer debugging environment compared to the target platform. Running the code on the development system, when it is eventually going to run in a foreign environment can be risky, so it's best to confront that risk regularly.

2. **Compiler Compatibility Check**: periodically compile for the target environment, using the cross-compiler expected to be used for production compilations; this stage can be seen as an early warning system for any compiler incompatibilities, since it warns the developer of any porting issue, such as unavailable header files, incompatible language support, and missing language features. As a result, the written code only uses facilities available in both development environments. A potential issue at this stage is that in early ES development, the tool chain may not yet be decided, and this compatibility check cannot be performed: in this case, developers should take their best guess on the tool chain and compile against that compiler. Finally, this stage should not run with every code change; instead, a target cross-compile should take place whenever a new language feature is used, a new header file is included or a new library call is performed.

3. **Run unit tests in an evaluation board**: compiled code could potentially run differently in the host development system and the target embedded processor. In order to mitigate this risk, developers can run the unit tests on an evaluation board; with this, any behavior differences between environments would emerge, ad since runtime libraries are usually prone to bugs [4], the risk is real. If it's late in the development cycle, and a reliable target hardware is available, this stage may appear unnecessary.

4. **Run unit tests in the target hardware**: the objective here is again to run the test suite, however this time doing so while exercising the real hardware. One additional aspect to this stage is that developers could also run target hardware-specific tests. These tests allow developers to characterize or learn how the target hardware behaves. An additional

challenge in this stage is limited memory in the target. The entire unit test suite may not fit into the target. In that case, the tests can be reorganized into separate test suites, where each suite fits in memory. This, however, does result in more complicated build automation process.

5. **Run acceptance tests in the target hardware**: Finally, in order to make sure that the product features work, automated and manual acceptance tests are run in the target environment. Here developers have to make sure that any of the hardware-dependent code that can't be fully tested automatically is tested manually.

Chapter 4

Literature Review

4.1 Related Work

In the past, the Empirical Software Engineering community has taken interest into the investigation of the effects of TDD on several outcomes [16] [17] [18], including the ones of interest for this study—*i.e.,*, software/functional quality and productivity. These studies are summarized in Systematic Literature Reviews (SLRs) and meta-analyses (*e.g.,* [19, 20, 21, 22]). The SLR by Turhan *et al.* [22] includes 32 primary studies (2000-2009). The gathered evidence shows a moderate effect in favor of TDD on quality, while results on productivity is inconclusive, namely there is no decisive advantage on productivity for employing TDD. Bissi *et al.* [19] conducted an SLR that includes 27 primary studies (1999-2014). Similarly to Turhan *et al.* [22], the authors observed an improvement of functional quality due to TDD, while results are inconclusive for productivity. Rafique and Misic [21] conducted a meta-analysis of 25 controlled experiments (2000-2011). The authors observed a small effect in favor of TDD on functional quality, while again for productivity the results are inconclusive. Finally, Munir et al. [20] in their SLR classifies 41 primary studies (2000-2011) according to the combination of their rigor and relevance. The authors found different conclusions for both functional quality and productivity in each classification.

An example of long-term investigation is the one by Marchenko et al. [23]. Theauthors conducted a three-year-long case study about the use of TDD at Nokia-Siemens Network. They observed and interviewed eight participants (one Scrum master, one product owner, and six developers) and then ran qualitative data analyses. The participants perceived TDD as important for the improvement of their code from a structural and functional

perspective. Moreover, productivity increased due to the team improved confidence with the code base. The results show that TDD was not suitable for bug fixing, especially when bugs are difficult to reproduce (e.g., when a specific environment setup is needed) or for quick experimentation due to the extra effort required for testing. The authors also reported some concerns regarding the lack of a solid architecture when applying TDD.

Beller *et al.* [24] executed a long-term study in-the-wild covering 594 open-source projects over the course of 2.5 years. They found that only 16 developers use TDD more than 20% of the time when making changes to their source code. Moreover, TDD was used in only 12% of the projects claiming to do so, and for the majority by experienced developers.

Borle *et al.* [25] conducted a retrospective analysis of (Java) projects, hosted on GitHub, that adopted TDD to some extent. The authors built sets of TDD projects that differed one another based on the extent to which TDD was adopted within these projects. The sets of TDD projects were then compared to control sets so as to determine whether TDD had a significant impact on the following characteristics: average commit velocity, number of bug-fixing commits, number of issues, usage of continuous integration, and number of pull requests. The results did not suggest any significant impact of TDD on the above-mentioned characteristics.

Latorre [26] studied the capability of 30 professional software developers of different seniority levels (junior, intermediate, and expert) to develop a complex 4 software system by using TDD. The study targeted the learnability of TDD since the participants did not know that technique before participating in the study. The longitudinal one-month study started after giving the developers, proficient in Java and unit testing, a tutorial on TDD. After only a short practice session, the participants were able to correctly apply TDD (e.g., following the prescribed steps). They followed the TDD cycle between 80% and 90% of the time, but initially, their performance depended on experience. The seniors needed only few iterations, whereas intermediates and juniors needed more time to reach a high level of conformance to TDD. Experience had an impact on performance—when using TDD, only the experts were able to be as productive as they were when applying a traditional development methodology (measured during the initial development of the system). According to the junior participants, refactoring and design decision hindered their performance. Finally, experience did not have an impact on long-term functional quality. The results show that all participants delivered functionally correct software regardless of their seniority. Latorre [15] also provides initial evidence on the retainment of TDD. Six months after the study investigating the learnability of TDD, three developers, among those

who had previously participated in that study, were asked to implement a new functionality. The results from this preliminary investigation suggest that developers retain TDD in terms of developers' performance and conformance to TDD. Although the above-mentioned studies [15, 16, 3, 5] have taken a longitudinal perspective when studying TDD, none of them has mainly focused on the effect of TDD applied to the development of ESs.

4.2 Testing Embedded Systems

Garousi *et al.* [27] provided a systematic literature mapping for ES testing by reviewing 312 papers, focusing on types of testing activity, types of test artifacts generated, and the types of industries in which studies have focused. As the word cloud presented in the survey suggests, topics such as model-based and automated/automatic (testing), (test-case) generation, and control systems are among the most popular ones. Most of the review papers (137 of 312, around 43.9%) present solution proposals without rigorous empirical studies; 98 (31.4%) papers are weak empirical studies (validation re- search). 36 (11.5%) are experience papers. 34 are strong empirical studies (evaluation research). 2 and 5 papers, respectively, are philosophical and opinion papers. In terms of level of testing considered in the papers, most of them (233 papers) considered system testing. 89 and 36 papers, respectively, focused on unit and integration testing. By focusing on unit testing, two of the sources ([28], [29]) applied TDD to embedded software. Several practical examples of automated unit test code were provided. As for the type of test activities, there is a good mix of papers proposing techniques and tools for each of the test activities, with a major focus on test execution, automation and criteria-based test case design (*e.g.,* based on code coverage).

[11]

Chapter 5

Experimental Approach

5.1 Overview

Software engineering is a rapidly growing field that has seen significant advances in recent years. As software systems have become increasingly complex and critical to the functioning of modern society, the need for effective and efficient methods for designing, building, and maintaining software has become more pressing than ever. One of the key ways in which researchers in the field of empirical software engineering work to improve the state of the art is through experimental studies.

In this chapter we will present in detail the planning and approach we followed to establish the studies and analyze their results. A controlled study, the baseline, followed by a replication, was conducted with the participation of 9 undergraduate Master's degree and third-year Bachelor's degree students enrolled in the *Embedded Systems* course at the University of Salerno in Italy. Participation in the studies was agreed upon by the students and was voluntary, with the outcome not directly affecting the final mark of the students for the exam.

Before taking part in the first study, a set of lectures and training sessions was held with the objective of training the participants on the topics tackled by the studies, namely unit testing, test scaffolding and TDD.

Overall, this experimental study aims to contribute to the body of knowledge in the field of empirical software engineering by providing new insights and understanding into the application of TDD for ES development.

5.2 Research Questions

As for the baseline experiment, and following the main research question presented in the introduction section of this thesis, we defined the main goal of this study by applying the Goal Question Metrics (GQM) template [30]. According to the GQM, for an organization to measure purposefully it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals.

The main goal for the baseline experiment is therefore defined as follows:

Analyze the use of TDD **for the purpose of** evaluating its effects in the development of ESs **with respect to** the quality of produced source code and its complexity and the developers' productivity **from the point of** view of the researcher and lecturer **in the context of** an ES course involving second year Master students in Computer Science.

According to this objective, three research questions were defined:

- **RQ1.** Does the use of TDD affect the quality of the solution to a programming task? *Aim*: We defined this 'to understand if (and possibly to what extent) the use of TDD with respect to YW makes a difference with respect to quality of the written source code. An answer to RQ1 would have practical implications within the Agile community. For example, in case the use of TDD positively affects source code quality, lecturers can teach TDD in ES courses so spreading Agile in the academic context and then facilitating the adoption of this approach in the industrial context. In other words, if the newcomers of the working market are familiar with TDD, and it is shown that it produces better quality source code for ESs, the software industry could be encouraged to migrate their development from NO-TDD to TDD.

- **RQ2.** Does the use of TDD affect the complexity of the solution to a programming task? *Aim*: With this RQ, we intend to understand if (and possibly to what extent) the complexity of the source code—how difficult to understand it is from the developer's perspective—of an ES is affected by the development approach (TDD vs. NO-TDD). The answer to this question helps us to better understand TDD applied to the development of ESs from a different perspective. For example,

the researcher could be interested to study the extent with which the quality of the source code of an ES affects software maintenance and evolution. A positive answer to RQ2 could justify this research direction.

- **RQ3.** Does the use of TDD increase developers' productivity? Aim: We defined this RQ to understand if (and possibly to what extent) there is an effect of TDD on the developers' productivity. The answer to RQ3 helps us to better understand TDD in the development of ESs companies operating in the context of the development of ESs could be encouraged to use TDD in case: (*i*) there is an evidence that this approach improves productivity and (*ii*) developers are familiar with this approach before being hired (e.g., TDD has been learned at university).

5.3 Experimental Units

The participants for the two experimental studies were students at the University of Salerno, in Italy; they were a mix of second-year master's degree students in Salerno, and students visiting the university by means of the Erasmus program; this last group was further made up of Master's degree students and third-year Bachelor's degree students. Both groups were enrolled in the *Embedded Systems* course at the University of Salerno. As for the Master's students, not all of them had a computer science background (Bachelor's degree).

The *Embedded Systems* course covered the following topics: modeling and design of an ES, state machines, sensors and actuators, embedded processors, memory architectures, embedded security and privacy concepts, embedded operating systems and scheduling, and ES testing. No TDD concepts were taught directly in the course; the topic was in turn covered by the lectures and training sessions preceding the studies.

Participating in the studies was voluntary; the students were informed that any gathered data would be treated anonymously and shared for research purposes only; furthermore, participation would not directly affect their final mark for the *Embedded Systems* course; however, in order to encourage student participation, those who took part in the studies were rewarded with a bonus in their final mark. Among the students taking the course, 9 participated.

The participants for the tasks were later asked to carry out their task by either using TDD or NO-TDD (i.e., any approach they preferred, except for

TDD) depending on the group they were partitioned in. and on the period the task took place.

5.4 Experimental tasks design

The experimental objects for the studies were three code katas, programming exercises aimed at practicing a technique or a programming language. All three were designed according to the target platform in mind, a Raspberry Pi model 4, even in the two tasks of the baseline study have not been effectively deployed on the real hardware; this was done in order to make the mocked implementation of the participants resemble as close as possible the embedded implementation. The main mocked component utilized was a facade for the Raspberry Pi's General Purpose Input/Output library, available as open source software [31]. For the replication study, another factor that influenced the design of the kata was the availability of the hardware, including sensors and actuators, as well as how hard these would have been to test in real time.

5.4.1 IntelligentOffice

The first task for the baseline study; the goal is to develop a system which allows the user to manage the light, window blinds, and air quality level inside an office.

5.4.2 CleaningRobot

The second task for the baseline study; the goal is to develop a system to control a cleaning robot which moves in a room and cleans the dust on the floor along the way.

5.4.3 SmartHome

This last task was The goal of this task is to develop an intelligent system to manage various aspects of room inside a house.

Further details on the experimental objects, including user stories, hardware used, and other information, are provided as an appendix to this thesis.

5.5 Variables and hypotheses

The participants were asked to carry out each task by using either TDD or the approach they preferred (i.e., NO-TDD), therefore one of the independent variables considered is **Condition**, a nominal variable assuming two values, TDD and NO-TDD. The data was collected over two periods for the controlled study, and over an additional period for the non-controlled study, so a second independent variable is **Period**, assuming the values $P1, P2, and P3$. During the three periods both treatments (TDD or NO-TDD) were applied. Finally, since the participants were split into two groups, the last independemt variable is **Group**, which can assume the values G1 and G2.

As for the dependent variables considered in the studies, these are: **QLTY**, **PROD**, **TEST**, **CYC**, **COG**, **LOC**. The variables QLTY, PROD and TEST have been used in previous empirical studies [17], [32], [33]. As for the others, ...

QLTY quantifies the external quality of the solution a participant implemented. It is formally defined as follows:

$$QLTY = \frac{\sum_{i=1}^{\#TUS} QLTY_i}{\#TUS} * 100$$

where $\#TUS$ is the number of user stories a participant tackled, while $QLTY_i$ is the external quality of the $i-th$ user story; to determine whether a user story was tackled or not, the asserts in the test suite corresponding to the story were checked: if at least one assert in the test suite for the story passed, than the story was considered as tackled. $\#TUS$ is formally defined as follows:

$$\#TUS = \sum_{i=1}^{n} \begin{cases} 1 & \#ASSERT_i(PASS) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Finally, the quality of the $i-th$ user story (i.e., $QLTY_i$) is defined as the ratio of asserts passed for the acceptance suite of the $i-th$ user story over the total number of asserts in the acceptance suite for the same story. More formally:

$$QLTY_i = \frac{\#ASSERT_i(PASS)}{\#ASSERT_i(ALL)}$$

As a result, the $QLTY$ measure deliberately excludes unattempted tasks and tasks with zero success; therefore, it represents a local measure of external quality calculated over the subset of user stories that the subject attempted. $QLTY$ is a ratio measure in the range $[0, 100]$.

PROD estimates the productivity of a participant. It is computed as follows:

$$PROD = \frac{\#ASSERT(PASS)}{\#ASSERT(ALL)} * 100$$

where $ASSERT(PASS)$ is the total number of asserts that have passed, by considering all acceptance test suites, while $ASSERT(ALL)$ refers to the total number of asserts in the acceptance suites. The $PROD$ variable can assume values between 0 and 100, where a value close to 0 indicates low productivity in the implemented solution, while a value close to 1 refers to high productivity.

The TEST variable quantifies the number of unit tests a participant wrote. It is defined as the number of assert statements in the test suite written by the participant; this variable ranges from 0 to ∞.

As for the additional three dependent variables regarding the general code quality in the submitted projects, these are defined as follows: (CYC) refers to the Cyclomatic Complexity of the implemented solution; it is a value used to determine the stability and level of confidence in a program, and it measures the number of linearly-independent paths inside a program module. A program with a lower Cyclomatic Complexity is generally easier to understand and less "risky" to modify; it can be used as an estimate on how difficult the code will be to cover/test.

(COG) is the Cognitive Complexity of the solution; it is a measurement of how difficult a program module is to intuitively understand. A method's Cognitive Complexity is based on a few rules [34]:

1. Code is not considered more complex when it uses shorthand syntax that the language provides for collapsing multiple statements into one.
2. Code is considered more complex for each "break in the linear flow of the code".
3. Code is considered more complex when "flow breaking structures are nested"

Finally, LOC is the total number of lines of code written by the participant; it's defined as the sum of the individual lines of code in both the production code source file and the test code source file.

5.6 Study design

For the controlled baseline experiment, the participants were randomly split into two groups, G1 and G2, having 4 and 5 members respectively. For the

first period P1, the group G1 was assigned the TDD version of the first task, *IntelligentOffice*, while the group G2 was assigned the NO-TDD version; on the other hand, during period P2, the group G1 was assigned the NO-TDD version of the second task, *CleaningRobot*, while the group G2 was assigned the TDD version. Therefore, the design of our study can be classified as a repeated-measures, within-subjects design. In each period, the participants in G1 and G2 dealt with different experimental objects, therefore, at the end of the study, every participant had tackled each experimental object only once.

As for the replication study, the group structure remained the same, however each participant was randomly assigned the TDD or NO-TDD version of the third and final experimental task, *SmartHome*.

5.7 Procedure

Before our study took place, we collected some demographic information on the participants. To this end, the participants were asked to fill out an on-line pre-questionnaire (created by means of Google Forms).

The *Embedded Systems* course, during which the study was conducted, started in September 2022. The first task, P1, took place on Tuesday, December 6th 2022, while the second, P2, took place on Tuesday, December 13th 2022.

Between the start of the course and P1, most participants had never dealt with TDD, while they were somewhat familiar with unit testing and iterative test-last development. In the weeks prior to the beginning of the experiment, all the participants attended a series of frontal lectures and training/homework sessions during which they were trained on the main topics they would encounter during the future experimental tasks.

The schedule for the experiments went as follows:

- Day 1 - Frontal lecture - Introduction on unit testing and its guidelines. Followed by a
- Day 2 - Frontal lecture - Test scaffolding and Raspberry Pi GPIO library-
- Day 3 - Frontal lecture - Test-Driven Development -
- Day 4 - Training task - TDD exercise -
- Day 6 - First experimental task - *IntelligentOffice*

- Day 7 - Second experimental task - *CleaningRobot*

...

After each period of the controlled baseline study, participants were asked to fill out another on-line questionnaire, with the purpose of describing their general experience with the implementation of the task, focusing on their testing approach. The structure of the post-questionnaires was made up of three interval scale questions, and a variable number of open-ended questions, two for the TDD group and three for the NO-TDD, with the latter having an additional question, as the first open-ended question, asking to provide information about the chosen approach for testing. Furthermore, the post-questionnaire presented at the end of period P2 contained an additional open question: here, participants had to provide their feelings towards both testing practices, TDD and NO-TDD.

More specifically, the interval scale questions were:

- **Q1.** Regarding the comprehensibility of the provided user stories, I have found them: (Very unclear | Unclear | Neither clear nor unclear | Clear | Very clear).
- **Q2.** I have found the development task: (Very difficult | Difficult | Neither easy nor difficult | Easy | Very easy).
- **Q3.** Applying (*i.e.,* TDD or NO-TDD) to accomplish the development task has been: (Very difficult | Difficult | Neither easy nor difficult | Easy | Very easy).

As for the open-ended questions:

- (**NO-TDD only**) Describe the no-TDD approach you have followed to accomplish the development task.
- Provide your feelings (both positive and negative) about (*i.e.,* TDD or NO-TDD).
- Provide your feelings (both positive and negative) about the development task.
- (**Task 2 only**) After applying (*i.e.,* TDD or NO-TDD) in the last exercise, do you have any thoughts on the differences between the two approaches and your preference for using one over the other?

No formal questionnaire was provided for the replication study; however, after the hardware deployment step, each participant was individually interviewed about their overall experience with the studies:

1. Provide your feelings (both positive and negative) about the final development project, (*e.g.,* development pipeline, used technologies).

2. Provide your feelings (both positive and negative) about the development approach (*i.e.,* TDD or NO-TDD) used to accomplish the final development project:

   - TDD: did you perform any refactoring?
   - NO-TDD: did you test the code at all? If so, which approach did you use?

3. Provide your feelings about the overall training experience (seminars, exercises, and homework on TDD and NO-TDD, experiments, and final task):

   - Positive and negative points and challenges encountered when applying TDD
   - What can be done to improve the application of TDD in the development of ESs
   - Please provide a discussion on TDD vs. NO-TDD in the development of ESs

5.8 Analysis Methods

1. **Individual analysis**: as a first way to summarize the distribution of the dependent variables,

2. **Aggregate analysis**:

Meta-analysis is a research process used to systematically synthesize or merge the findings of single, independent studies, using statistical methods to calculate an overall or "absolute" effect. This kind of analysis does not simply pool data from smaller studies to achieve a larger sample size: analysts use well recognized, systematic methods to account for differences in sample size, variability (heterogeneity) in study approach and findings (treatment effects) and test how sensitive their results are to their own systematic review protocol (study selection and statistical analysis).

5.9 Results

5.9.1 Individual analysis

In this section we will report the values observed for each dependent variable during their individual analysis. For each experimental task, we will provide a table summarizing the minimum and maximum values, mean, median, and standard deviation of each dependent variable, for the two separate conditions (*i.e.,* TDD and NO-TDD), before considering the values of the variable for the first two tasks simultaneously, in order to provide an overview of the differences between the two testing approaches.

Besides the tables, box plot charts will be used to visualize the values assumed by the dependent variables. A box plot chart is a type of chart often used in explanatory data analysis; it visually shows the distribution of numerical data and skewness through displaying the data quartiles (or percentiles) and averages.

The information that can be extracted form a box plot chart includes:

- **Minimum value**: the lowest value, excluding outliers (shown at the end of the lower whisker).
- **Maximum value**: the highest score, excluding outliers (shown at the end of the upper whisker).
- **Median**: marks the mid-point of the data and is shown by the line that divides the box into two parts. Half the values are greater than or equal to this value and half are less than this value.
- **Inter-quartile range**: the middle "box" represents the middle 50% of values for the group. The range of values from lower to upper quartile is referred to as the inter-quartile range. The middle 50% of scores fall within the inter-quartile range.
- **Upper quartile**: 75% of the scores fall below the upper quartile.
- **Lower quartile**: 25% of scores fall below the lower quartile.
- **Whiskers**: the upper and lower "whiskers" represent scores outside the middle 50% (i.e. the lower 25% of scores and the upper 25% of scores).
- **Outliers**: observations that are numerically distant from the rest of the data. They are defined as data points that are located outside the whiskers of the box plot, and are represented by a dot.

To provide an example of the kind of information that a box plot chart can provide, please consider the following figure displaying four student groups' opinions on a subject:

...

Some observations that can be made include:

- The box plot is comparatively short (see box plot (2)). This suggests that overall the student have a high level of agreement and therefore the values are very similar between each other.
- The box plot is comparatively tall (see box plots (1) and (3)). This, on the other hand, suggests students hold quite different opinions about this aspect or sub-aspect.
- Obvious differences between box plots (see box plots (1) and (2), (1) and (3), or (2) and (4)). Any obvious difference between box plots for comparative groups is worthy of further investigation.
- The 4 sections of the box plot are uneven in size (see box plot (1)). This reveals that many students share a similar view at certain parts of the scale, but in other parts of the scale students are more variable in their views. A long upper whisker in the means that students views are varied among the most positive quartile group, and very similar for the least positive quartile group.
- Same median, different distribution (see box plots (1), (2), and (3)). The medians, which generally tend to be close to the average, are at the same level. However, the box plots in these examples show very different distributions of views. It's always important to consider the pattern of the whole distribution of responses in a box plot.

Task 1: The following tables display the values for the variables measured for the first experimental task, IntelligentOffice, for the two groups, by applying each of the two conditions.

Task 1 - TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	73	96	81.12	77.77	10.73
PROD	72	96	83	82	10
TEST	8	10	9.5	10	1
CYC	21	28	24.75	25	2.87
COG	14	25	19	18.5	4.69
LOC	154	195	167	159.5	18.95

Task 1 - NO-TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	65.55	82	75.35	78.77	7.43
PROD	56	84	76	80	11.66
TEST	0	12	3.8	0	5.49
CYC	12	18	15.6	16	2.19
COG	9	17	14	15	3
LOC	74	157	111.6	100	33.69

(a) QLTY (b) PROD

(c) TEST (d) CYC

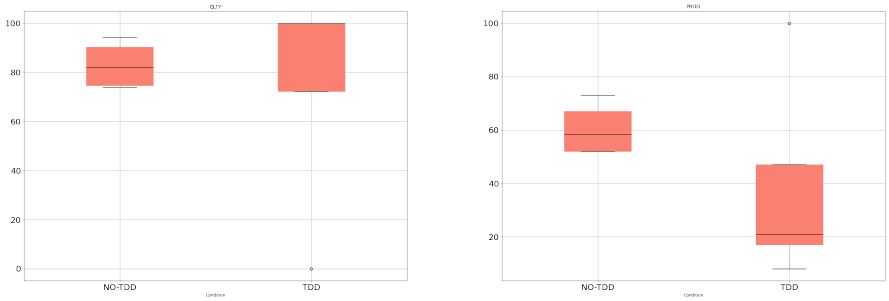
(e) COG (f) LOC

Figure 5.1: Box plots for task 1, *IntelligentOffice*

Task 2: The following tables display the values for the variables measured for the second experimental task, CleaningRobot, for the two groups, by

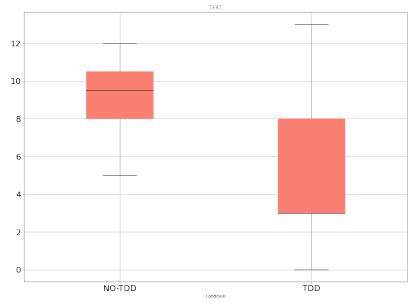
applying each of the two conditions.

Task 2 - TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	0	100	74.44	100	43.31
PROD	8	100	38.6	21	37.35
TEST	0	13	5.4	3	5.12
CYC	9	19	12	10	4.06
COG	2	40	12.8	4.0	15.91
LOC	80	203	128	115	45.61
Task 2 - NO-TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	74	94.43	83.07	81.94	10.17
PROD	52	73	60.5	58.5	10.24
TEST	5	12	9	9.5	2.94
CYC	16	36	23.5	21	9
COG	11	49	29	28	15.57
LOC	178	260	207.5	196	37.11

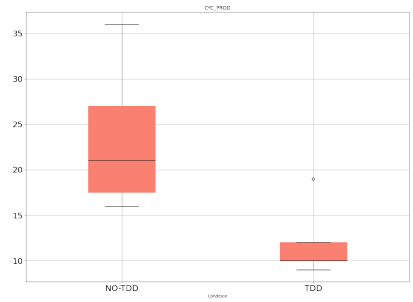


(a) QLTY

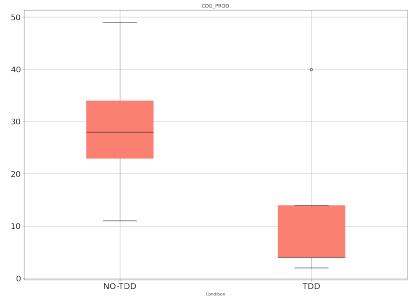
(b) PROD



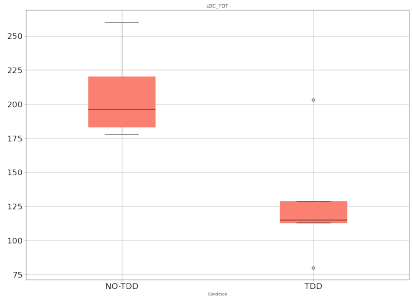
(c) TEST



(d) CYC



(e) COG



(f) LOC

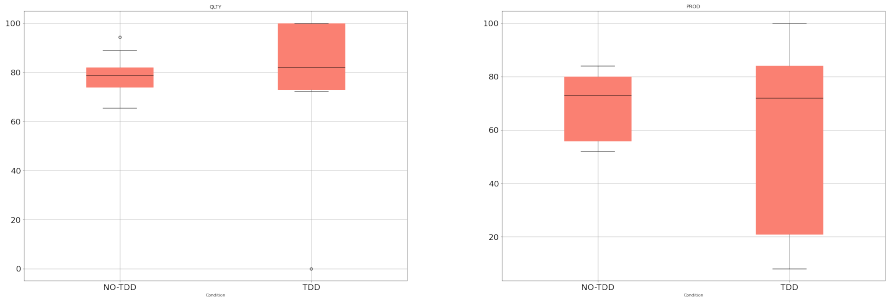
Figure 5.2: Box plots for task 2, *CleaningRobot*

Tasks 1 & 2: the following tables display the values for the variables measured for the first two experimental tasks, i.e., the controlled study, for

the two groups, by applying each of the two conditions.

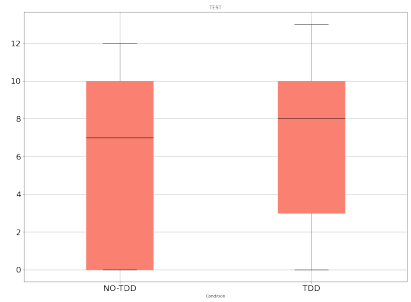
Task 1 & 2 - TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	0	100	77.41	82	31.52
PROD	8	100	58.33	72	35.76
TEST	0	13	7.22	8	4.26
CYC	9	28	17.66	19	7.51
COG	2	40	15.55	14	12.06
LOC	80	203	145.33	154	39.97

Task 1 & 2 - NO-TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	65.55	94.43	78.78	78.77	9.11
PROD	52	84	69.11	73	13.22
TEST	0	12	6.11	7.0	5.08
CYC	12	36	19.11	16	7.07
COG	9	49	20.66	15	12.56
LOC	74	260	154.22	157	60.32

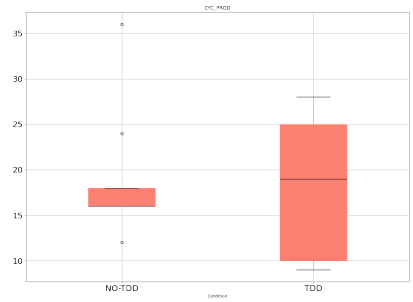


(a) QLTY

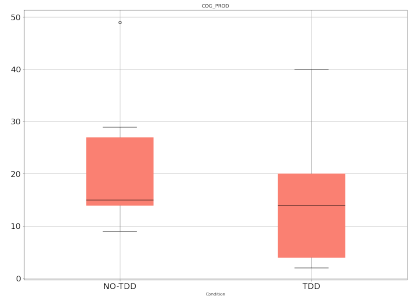
(b) PROD



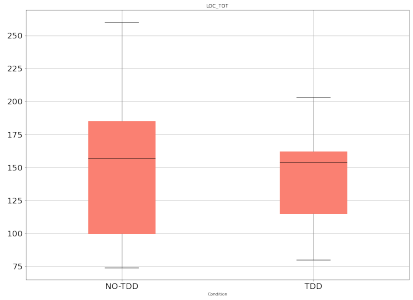
(c) TEST



(d) CYC



(e) COG



(f) LOC

Figure 5.3: Aggregated box plots for tasks 1 and 2

Tasks 3: finally, these last tables display the values for the variables measured for the third experimental task (non-controlled study), SmartHome,

for the two groups, by applying each of the two conditions.

Task 3 - TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	85	100	95	100	7.07
PROD	83.33	100	93.33	100	9.13
TEST	7	18	11.6	12	4.15
CYC	15	30	22.6	20	6.58
COG	18	34	25.8	25	7.08
LOC	150	232	187	175	36.15

Task 3 - NO-TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	80	100	86.25	82.5	9.46
PROD	75	100	83.33	79.16	11.78
TEST	0	11	4.5	3.5	5.44
CYC	16	23	19.5	19.5	2.88
COG	19	25	21	20	2.70
LOC	93	164	125.5	122.5	36.82

(a) QLTY

(b) PROD

(c) TEST

(d) CYC

(e) COG

(f) LOC

Figure 5.4: Box plots for task 3, *SmartHome*

5.9.2 Meta-analysis

The following figures display the results of the aggregate analysis on the variables of the baseline and replication studies through means of forest plot charts. The Standard Mean Difference (SMD) was chosen as the effect measure, as suggested

Forest plots are ...

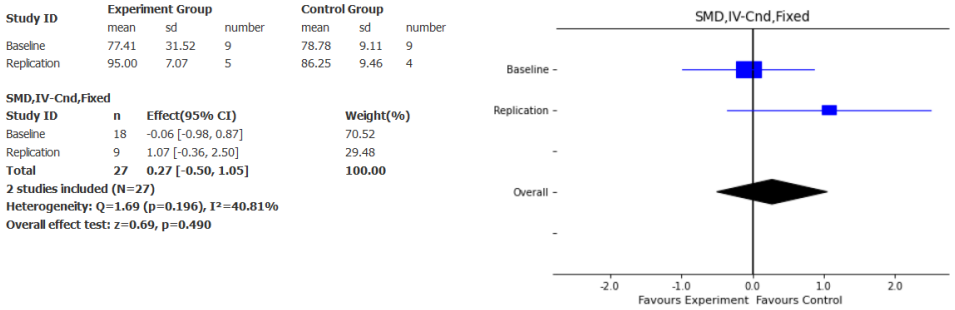


Figure 5.5: Forest plot chart for the QLTY dependent variable

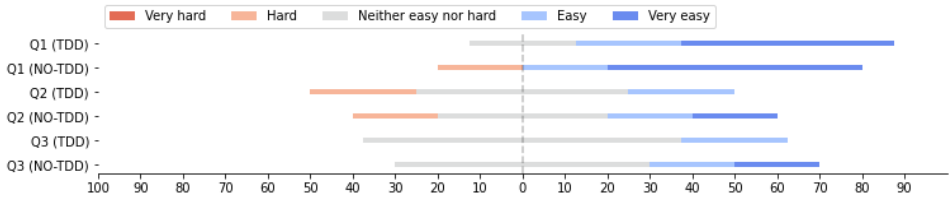
5.9.3 Post-questionnaire Analysis

Fig 5.6 summarizes the answers provided by the participants in the post-experiment questionnaires, comparing the responses by the employed condition (TDD or NO-TDD). Regarding user story comprehensibility, in the first experimental task, *IntelligentOffice*, the majority of participants have a similar agreement on the matter, with percentages of agreement of 88% (TDD) and 80% (NO-TDD). As for the second task, *CleaningRobot*, there is a more substantial difference, with only 40% of the TDD group having a positive perception of the user stories' comprehensibility, compared to the 87% of the NO-TDD group.

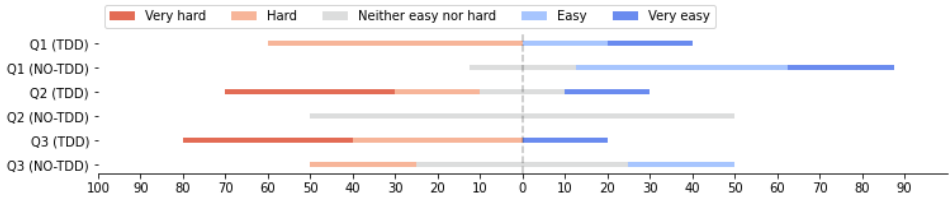
A similar trend can be noted for the second question, regarding the general task feasibility; here, in the first experimental task, the participants somewhat agree, with 50% of the TDD group and 40% of the NO-TDD group feeling neutral about the difficulty of the task. In the second task, the answers are quite mixed, with 40% of the participants in the TDD finding the task at hand very difficult to understand, and the other 60% equally split between finding the task hard, feeling neutral about it or considering it very easy. On the other hand, all of the participants in the NO-TDD group agreed on feeling neutral about their ease in developing it.

Finally, question three was focused on the difficulty of the participants in

applying their reference condition (*i.e.,* TDD or NO-TDD): for task 1, the TDD group did not have a strong opinion on applying this technique, with 76% feeling neutral about it, and the other 24% finding the application of TDD easy but not too easy; The NO-TDD group had also the majority of opinions feeling neutral about their approach (60%), however this time 20% of the participants felt very good about applying NO-TDD; this is probably due to their seniority with the approach.



(a) First experimental task, *IntelligentOffice*



(b) Second experimental task, *CleaningRobot*

Figure 5.6: Diverging stacked bar charts for the post-questionnaires

Thematic analysis is widely used in qualitative psychology research

Template Analysis is a form of thematic analysis which emphasizes the use of hierarchical coding but balances a relatively high degree of structure in the process of analyzing textual data with the flexibility to adapt it to the needs of a particular study [35]. In Template Analysis, it is permissible (though not obligatory) to start with some a priori themes, identified in advance as likely to be helpful and relevant to the analysis. These are always tentative, and may be redefined or removed if they do not prove to be useful for the analysis at hand.

5.10 Discussion

In this section we will discuss the obtained results and the implications of this study, as well as provide an answer to the RQs and analyze potential

threats to its validity.

5.10.1 Answers to Research Questions

5.10.2 Implications

5.10.3 Threats to Validity

In order to determine the potential threats to validity that may affect our studies, we referenced Wohlin *et al.*'s guidelines [36].

Completely avoiding/mitigating threats is often unfeasible, given the dependency between some threats: avoiding/mitigating a kind of threat (*i.e.,* internal validity) might intensify or even introduce another kind of threat [36].

Generally, the kinds of threats to validity that can occur in a study can be classified in the following categories:

1. **Threats to internal validity**. Internal validity the extent to which we can be confident that a cause-and-effect relationship established in a study cannot be explained by other factors; it makes the conclusions of a causal relationship credible and trustworthy. Without high internal validity, an experiment cannot demonstrate a causal link between two variables. There are three necessary conditions for internal validity and all three must occur to experimentally establish causality between an independent variable A (treatment variable) and dependent variable B (response variable):

   - The treatment and response variables change together.
   - The treatment precedes changes in the response variables
   - No confounding or extraneous factors can explain the results of the study.

   Threats to internal validity include:

   - **Selection bias**: groups are not comparable at the beginning of the study. For example, low-scorers were placed in Group A, while high-scorers were placed in Group B; Because there are already systematic differences between the groups at the baseline, any improvements in group scores may be due to reasons other than the treatment.
   - **Regression to the mean**: there is a statistical tendency for people who score extremely low or high on a test to score closer

to the middle the next time. Because participants are placed into groups based on their initial scores, it's hard to say whether the outcomes would be due to the treatment or statistical norms.

- **Social interaction and social desirability**: participants from different groups may compare notes and either figure out the aim of the study or feel resentful of others or pressured to act/react a certain way. For example, Groups B and C may resent Group A for some reason and, as such, they could be demoralized and perform poorly.
- **Attrition bias**: dropout from participants. For example, if 20% of participants provided unusable data, and almost all of them were from Group C, it will be hard to compare the two treatment groups to a control group.

2. **Threats to external validity**: the extent to which we can generalize the findings of a study to other measures, settings or groups. In other words, can we apply the findings of your study to a broader context? Threats to external validity include:

   - **Sampling bias**: the sampling considered for the study is not representative of the average population. If the sample includes only people that participated in the study voluntarily, they could have characteristics that may make them very different from other populations, like people randomly chosen.
   - **History**: an unrelated event influences the outcomes.
   - **Observer bias**: the characteristics or behaviors of the experimenter(s) unintentionally influence the outcomes, leading to bias and other demand characteristics. For example, a trainer of the experimental sessions unintentionally reveals that the outcome of the study will influence their final mark. As a result, participants will work extra hard from that point on to ensure their result are the best possible.
   - **Hawthorne effect**: the tendency for participants to change their behaviors simply because they know they are being studied. As an example, let's consider a set of participants that actively alters their behavior for the period of the study because they are conscious of their participation in the research.
   - **Testing effect**: the administration of a pre- or post-test, or the repetition of similar tasks affects the outcomes.

- **Aptitude-treatment**: interactions between characteristics of the group and individual variables together influence the dependent variable.
- **Situation effect**: factors like the setting, time of day, location, researchers' characteristics, etc. limit generalizability of the findings

3. **Threats to construct validity**

4. **Threats to conclusion validity**

As mentioned above, there are inherent trade-offs between validities: with internal and external validities for example, the more we control extraneous factors in your study, the less we can generalize our findings to a broader context.

As for our controlled and replication studies, let's consider the different threats individually:

Threats to internal validity. The main threat is perhaps related to the monitoring of the participants during the replication study; since they accomplished the implementation of the task at home, before deploying it on hardware under our supervision, we cannot be sure of the means by the participants to accomplish the task; we can however assume that, given the fact that the final score of the *Embedded Systems* course was not influenced in any way by the outcome of the task, the participants would have no reason to ... Besides this, a *selection* threat might have affected the overall results of the experiments, given that volunteers are generally more motivated and engaged compared to the average population [36]. Finally, another potential threat is *resentful demoralization*, which arises in participants when they receive a less desirable treatment; this causes them to not behave as they normally would. This last threat holds in all three experimental tasks, since it is related to the nature of the adopted experimental design.

Threats to external validity. The main external validity threat could be the one of *interaction of selection and treatment*: since both Bachelor's and Master's student were involved in the study, some of the latter with no prior testing experience or even without a strong Computer Science background, the results could potentially not be applicable to professional developers.

Threats to construct validity. Although we did not disclose the purpose of our study to the participants during the experimental tasks, they might have tried to guess it, and adapted their behavior accordingly, arising a threat of *hypotheses guessing*. Besides this, the threat of **evaluation apprehension** should have been fairly mitigated, since the participants knew that they would

be awarded the bonus score for the course regardless of their performance in the study.

Threats to conclusion validity In order to mitigate any potential threat of *random heterogeneity of participants*, before staring the experimental tasks, we trained the participants with a series of frontal lectures and exercise, in order to uniform their knowledge on the techniques and technologies that they would have later used and make the two groups as homogeneous as possible. A threat of *reliability of treatment implementation* might have occurred in tasks 1 and 2. For example, some participants might have followed TDD more strictly than others; however, this should equally affect both experimental groups.

Chapter 6

Conclusions

In this work of thesis we presented two experiments, a baseline experiment and its replication, with the objective of investigating how TDD can tackle ES development and how it compares to traditional, test-last, approaches to software development. We found that ...

At this point, there are a few directions for the research in the topic. First, the study could be replicated

Finally, a similar study could be replicated with the participants being professional software developers with years of experience in the ES field

Bibliography

[1] *Embedded Systems Market by Component (Hardware, Software), by Application (Automotive, Consumer Electronics, Industrial, Aerospace and Defense, Others): Global Opportunity Analysis and Industry Forecast, 2021-2031*. Tech. rep. Allied Market Research, Nov. 2022.

[2] Beck K. *Test-Driven Development by Example*. Addison Wesley, 2002.

[3] Hakan Erdogmus, Grigori Melnik, and Ron Jeffries. "Test-Driven Development". In: *Encyclopedia of Software Engineering*. Ed. by Phillip A. Laplante. Taylor & Francis, 2010, pp. 1211–1229.

[4] James W. G. *Object-Oriented Software Engineering Using UML, Patterns, and Java, Third Edition*. The Pragmatic Programmers, 2011.

[5] Itir Karac and Burak Turhan. "What Do We (Really) Know about Test-Driven Development?" In: *IEEE Softw.* 35.4 (2018), pp. 81–85.

[6] Bernd B. and Allen H. D. *Object-Oriented Software Engineering Using UML, Patterns, and Java, Third Edition*. Pearson, 2010.

[7] *Guidelines for Test-Driven Development*. URL: `https://learn.microsoft.com/en-us/previous-versions/aa730844(v=vs.80)?redirectedfrom=MSDN`.

[8] Itir Karac, Burak Turhan, and Natalia Juristo. "A Controlled Experiment with Novice Developers on the Impact of Task Description Granularity on Software Quality in Test-Driven Development". In: *IEEE Trans. Software Eng.* 47.7 (2021), pp. 1315–1330.

[9] *Zigbee*. URL: `https://csa-iot.org/all-solutions/zigbee/`.

[10] White E. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly, 2011.

[11] Vahid Garousi et al. "What We Know about Testing Embedded Software". In: *IEEE Softw.* 35.4 (2018), pp. 62–69.

[12] *What are MIL, SIL, PIL, and HIL, and how do they integrate with the Model-Based Design approach?* URL: https://www.mathworks.com/matlabcentral/answers/440277-what-are-mil-sil-pil-and-hil-and-how-do-they-integrate-with-the-model-based-design-approach.

[13] *SpeedGoat*. URL: https://www.speedgoat.com/.

[14] *Simulink*. URL: https://it.mathworks.com/products/simulink.html.

[15] Carl B. E. Michael J. K. William I. B. "Effective Test Driven Development for Embedded Software". In: (2006).

[16] Davide Fucci et al. "An External Replication on the Effects of Test-driven Development Using a Multi-site Blind Analysis Approach". In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*. ACM, 2016, 3:1–3:10.

[17] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. "On the Effectiveness of the Test-First Approach to Programming". In: *IEEE Trans. Software Eng.* 31.3 (2005), pp. 226–237.

[18] Lech Madeyski. "The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment". In: *Inf. Softw. Technol.* 52.2 (2010), pp. 169–184.

[19] Wilson Bissi, Adolfo Gustavo Serra Seca Neto, and Maria Cláudia Figueiredo Pereira Emer. "The effects of test driven development on internal quality, external quality and productivity: A systematic review". In: *Inf. Softw. Technol.* 74 (2016), pp. 45–54.

[20] Hussan Munir, Misagh Moayyed, and Kai Petersen. "Considering rigor and relevance when evaluating test driven development: A systematic review". In: *Inf. Softw. Technol.* 56.4 (2014), pp. 375–394.

[21] Yahya Rafique and Vojislav B. Misic. "The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis". In: *IEEE Trans. Software Eng.* 39.6 (2013), pp. 835–856.

[22] Burak Turhan et al. "How Effective is Test Driven Development". In: Oct. 2010.

[23] Artem Marchenko, Pekka Abrahamsson, and Tuomas Ihme. "Long-Term Effects of Test-Driven Development A Case Study". In: *Agile Processes in Software Engineering and Extreme Programming, 10th International Conference, XP 2009, Pula, Sardinia, Italy, May 25-29, 2009. Proceedings*. Ed. by Pekka Abrahamsson, Michele Marchesi, and Frank Maurer. Vol. 31. Lecture Notes in Business Information Processing. Springer, 2009, pp. 13–22. DOI: 10.1007/978-3-642-01853-4_4. URL: https://doi.org/10.1007/978-3-642-01853-4%5C_4.

[24] Moritz Beller et al. "Developer Testing in the IDE: Patterns, Beliefs, and Behavior". In: *IEEE Trans. Software Eng.* 45.3 (2019), pp. 261–284. DOI: 10.1109/TSE.2017.2776152. URL: https://doi.org/10.1109/TSE.2017.2776152.

[25] Neil C. Borle et al. "Analyzing the effects of test driven development in GitHub". In: *Empir. Softw. Eng.* 23.4 (2018), pp. 1931–1958. DOI: 10.1007/s10664-017-9576-3. URL: https://doi.org/10.1007/s10664-017-9576-3.

[26] Roberto Latorre. "Effects of Developer Experience on Learningand Applying Unit Test-Driven Development". In: *IEEE Trans. Software Eng.* 40.4 (2014), pp. 381–395. DOI: 10.1109/TSE.2013.2295827. URL: https://doi.org/10.1109/TSE.2013.2295827.

[27] Vahid Garousi et al. "Testing embedded software: A survey of the literature". In: *Inf. Softw. Technol.* 104 (2018), pp. 14–45. DOI: 10.1016/j.infsof.2018.06.016. URL: https://doi.org/10.1016/j.infsof.2018.06.016.

[28] James Grenning. "Applying test driven development to embedded software". In: *IEEE Instrumentation and Measurement Magazine* 10.6 (2007), pp. 20–25. DOI: 10.1109/MIM.2007.4428578.

[29] Jing Guan, Jeff Offutt, and Paul Ammann. "An industrial case study of structural testing applied to safety-critical embedded software". In: *2006 International Symposium on Empirical Software Engineering (ISESE 2006), September 21-22, 2006, Rio de Janeiro, Brazil*. Ed. by Guilherme Horta Travassos, José Carlos Maldonado, and Claes Wohlin. ACM, 2006, pp. 272–277. DOI: 10.1145/1159733.1159774. URL: https://doi.org/10.1145/1159733.1159774.

[30] Rini Solingen et al. "Goal Question Metric (GQM) Approach". In: Jan. 2002. ISBN: 9780471028956. DOI: 10.1002/0471028959.sof142.

[31] *Raspberry Pi GPIO library mock*. URL: https://github.com/codenio/Mock.GPIO.

[32] Davide Fucci et al. "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?" In: *IEEE Trans. Software Eng.* 43.7 (2017), pp. 597–614.

[33] Ayse Tosun et al. "An industry experiment on the effects of test-driven development on external quality and productivity". In: *Empir. Softw. Eng.* 22.6 (2017), pp. 2763–2805.

[34] *Cognitive Complexity*. URL: https://docs.codeclimate.com/docs/cognitive-complexity.

[35] Nigel King. "Using Templates in the Thematic Analysis of Text". In: Jan. 2004, pp. 257–270. ISBN: 9780761948889. DOI: 10.4135/9781446280119.n21.

[36] Claes Wohlin et al. *Experimentation in Software Engineering - An Introduction*. Vol. 6. The Kluwer International Series in Software Engineering. Kluwer, 2000.

Appendices

Intelligent Office - TDD Group

Goal

The goal of this task is to develop an intelligent office system, which allows the user to manage the light and air quality level inside the office.

The office is square in shape and it is divided into four quadrants of equal dimension; on the ceiling of each quadrant lies an **infrared distance sensor** to detect the presence of a worker in that quadrant.

The office has a wide window on one side of the upper left quadrant, equipped with a **servo motor** to open/close the blinds.

Based on a **Real Time Clock** (**RTC**), the intelligent office system opens/closes the blinds each working day.

A **photoresistor**, used to measure the light level inside the office, is placed on the ceiling. Based on the measured light level, the intelligent office system turns on/off a (ceiling-mounted) **smart light bulb**.

Finally, the intelligent office system also monitors the air quality in the office through a **carbon dioxide** (**CO2**) **sensor** and then regulates the air quality by controlling the **switch** of an exhaust fan.

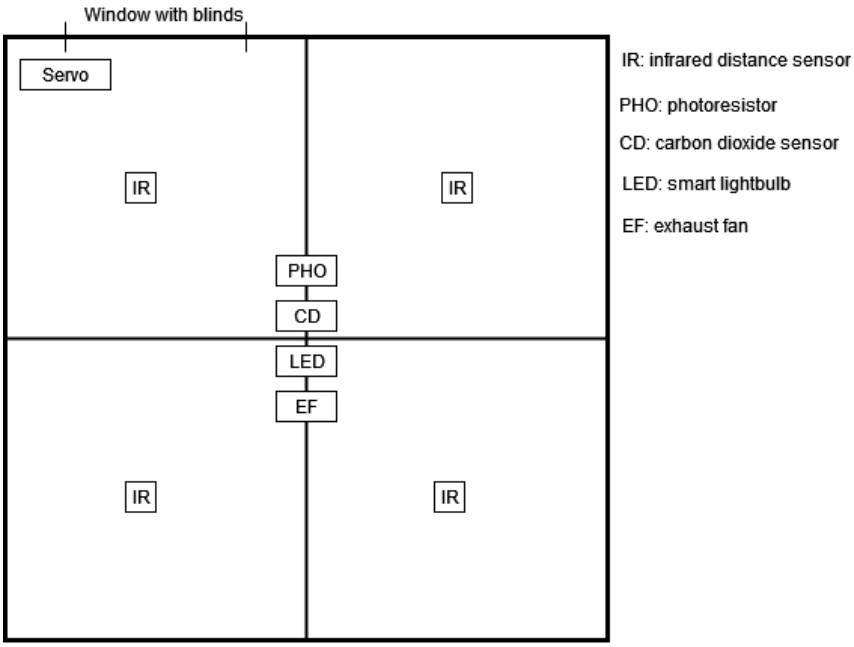
To recap, the following sensors and actuators are present:

- Four infrared distance sensors, one in each quadrant of the office.
- An RTC to handle time operations.
- A servo motor to open/close the blinds of the office window.
- A photoresistor sensor to measure the light level inside the office.
- A smart light bulb.
- A carbon dioxide sensor, used to measure the CO2 levels inside the office.
- A switch to control an exhaust fan mounted on the ceiling.

The communication between the main board and the other components happens with GPIO pins; GPIO communication is configured in BOARD mode. For further details on how to use the GPIO library refer to the **mock.GPIO** class in the source code.

Handle any error situation that you may encounter by throwing the **IntelligentOfficeError** exception.

The image below recaps the layout of the sensors and actuators in the office.



For now, you don't need to know more; further details will be provided in the User Stories below.

Instructions

Depending on your preference, either clone the repository at https://github.com/Espher5/intelligent_office or download the source files as a ZIP archive; afterwards, import the project into PyCharm.

Take a look at the provided project, which contains the following classes:

- **`IntelligengOffice`**: you will implement your methods here.
- **`IntelligengOfficeError`**: exception that you will raise to handle errors.
- **`IntelligengOfficeTest`**: you will write your tests here.
- **`mock.GPIO`**: contains the mocked methods for GPIO functionalities.
- **`mock.RTC`**: contains the mocked methods for RTC functionalities.

Remember, you are **NOT ALLOWED** to modify the provided API in any way (i.e., class names, method names, parameter types, return types). You can

however add fields, methods, or even classes (including other test classes), as long as you comply with the provided API.

Use **TDD** to implement this software system.

The requirements of the software system to be implemented are divided into a set of **USER STORIES**, which serve as a to-do list; you should be able to incrementally develop the software system, without an upfront comprehension of all the requirements. **DO NOT** read ahead and handle the requirements (i.e., specified in the user stories) one at a time in the order provided.

When a story is **IMPLEMENTED**, move on to the **NEXT** one. A story is implemented when you are confident that your software system correctly implements all the functionality stipulated by the story's requirement. This implies that all your **TESTS** for that story and all the tests for the previous stories **PASS**. You may need to review your software system as you progress towards more advanced requirements.

At the end of the task, fill out the post-questionnaire where, among other things, you will be asked to share your project (either as a link to a GitHub repository or as a sharing link to a ZIP file). The post-questionnaire is available at: https://forms.gle/H4eNDDR6CjLjUofY6.

User Stories

Remember to use TDD to implement the following user stories.

1. Office worker detection

Four infrared distance sensors, one in each office quadrant, are used to determine whether someone is currently in that quadrant.

Each sensor has a data pin connected to the board, used by the system in order to receive the measurements; more specifically, the four sensors are connected to pin 11, 12, 13, and 15, respectively (BOARD mode).

The communication with the sensors happens via the GPIO input function. The pins have already been set up in the constructor of the `IntelligentOffice` class.

The output of the infrared sensor is an analog signal which changes intensity according to the distance between the sensor and the object (i.e., 2.5V when an object is 50 cm away and ~0V when the object is out of the max range of the sensor). For this exercise, let's assume the input can be classified into just these two categories:

- Non-zero value: it indicates that an object is present in front of the sensor (i.e., a worker).
- Zero value: nothing is detected in front of the sensor.

Requirement:

- Implement `IntelligentOffice.check_quadrant_occupancy(pin: int) -> bool` to verify whether a specific quadrant has someone inside of it.

2. Open/close blinds based on time

Regardless of the presence of workers in the office, the intelligent office system fully opens the blinds at 8:00 and fully closes them at 20:00 each day except for Saturday and Sunday.

The system gets the current time and day from the RTC module connected on pin 16 (BOARD mode) which has already been set up in the constructor of the `IntelligentOffice` class. Use the instance variable `self.rtc` and the methods of `mock.RTC` to retrieve these values.

To open/close the blinds, the system commands a servo motor, which is a type of DC (Direct Current) motor that, upon receiving a signal, can rotate itself to any angle from 0 to 180 degrees. We can control it by sending a PWM (Pulse-Width Modulation) signal to its signal pin; this means sending a HIGH signal for a certain period of time (called duty cycle), followed by a LOW signal period. The duty cycle determines the angle the servo motor will rotate to.

The servo is connected on pin 18 (BOARD mode), and operates at 50hz frequency. Furthermore, let's assume the blinds can be in the following states:

- **Fully closed**, corresponding to a 0 degrees rotation of the servo motor.
- **Fully open**, corresponding to a 180 degrees rotation of the servo motor.

In order to calculate the duty cycle corresponding to a certain angle, refer to the following formula:

duty cycle = (angle / 18) + 2

The servo motor has already been configured and can be controlled by passing the duty cycle (see the formula above) corresponding to the desired angle to the `change_servo_angle(duty_cycle: float) -> None` method in the `IntelligentOffice` class.

Finally, since at this stage of development there is no way to determine the state of the physical servo motor, use the `self.blinds_open` instance variable to keep track of its state.

Requirement:

- Implement `IntelligentOffice.manage_blinds_based_on_time() -> None` to control the behavior of the blinds.

3. Light level management

The intelligent office system allows setting a minimum and a maximum target light level in the office. The former is set to 500 lux, the latter to 550 lux.

To meet the above-mentioned target light levels, the system turns on/off a smart light bulb. In particular, if the actual light level is lower than 500 lux, the system turns on the smart light bulb. On the other hand, if the actual light level is greater than 550 lux, the system turns off the smart light bulb.

The actual light level is measured by the (ceiling-mounted) photoresistor connected on pin 22 (BOARD mode). The communication with the sensor happens via the GPIO input function. For this sensor, the value returned by the GPIO input function is assumed to be in lux. The pin has already been set up in the constructor of the `IntelligentOffice` class.

The smart light bulb is represented by a LED, connected to the main board via pin 29 (BOARD mode). Communication with the LED happens via the GPIO output function. The pin has already been set up in the constructor of the `IntelligentOffice` class.

Finally, since at this stage of development there is no way to determine the state of the physical light bulb, use the boolean instance variable `self.light_on`, defined in the constructor of the `IntelligentOffice` class, to keep track of the state of the light bulb.

Requirement:

- Implement `IntelligentOffice.manage_light_level() -> None` to control the behavior of the smart light bulb.

4. Manage smart light bulb based on occupancy

When the last worker leaves the office (i.e., the office is now vacant), the intelligent office system stops regulating the light level in the office and then turns off the smart light bulb.

On the other hand, the intelligent office system resumes regulating the light level when the first worker goes back into the office.

Requirement:

- Implement `IntelligentOffice.manage_light_level() -> None` to control the behavior of the smart light bulb.

5. Monitor air quality level

A carbon dioxide sensor is used to monitor the CO2 levels inside the office. If the amount of detected CO2 is greater than or equal to 800 PPM, the system turns on the switch of the exhaust fan until the amount of CO2 is lower than 500 PPM.

The carbon dioxide sensor is connected on pin 31 (BOARD mode). The communication with the sensor happens via the GPIO input function. For this sensor, the value returned by the GPIO input function is assumed to be in PPM.

The switch to the exhaust fan is connected on pin 32 (BOARD mode). The communication with the sensor happens via the GPIO output function.

Both the pin for the CO2 sensor and the one for the exhaust fan have already been set up in the constructor of the `IntelligentOffice` class.

Finally, since at this stage of development there is no way to determine the state of the physical exhaust fan switch, use the boolean instance variable `self.fan_switch_on`, defined in the constructor of the `IntelligentOffice` class, to keep track of the state of the fan switch.

Requirement:

- Implement `IntelligentOffice.monitor_air_quality() -> None` to control the behavior of CO2 sensor and the exhaust fan switch.