UNIVERSITY OF SALERNO

DEPARTMENT OF COMPUTER SCIENCE

Master Thesis in Computer Science

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

Supervisors

Prof. Giuseppe Scanniello
Prof. Simone Romano

Candidate

Michelangelo Esposito
05225 00982

ACADEMIC YEAR 2021/2022

Acknowledgements

I would like to express my heartfelt gratitude to my family, including my parents and my sister, for their unwavering love and support throughout my academic journey and my life. Your encouragement and belief in me have been the spark that ignited my motivation and fueled my drive to succeed.

I want to thank my two nieces for reminding me to always find joy and laughter in life. Your infectious energy and positive spirit have brought so much light into my world, and I am grateful to have you both in it.

Lastly, I would like to extend my deep appreciation to my two supervisors for their invaluable guidance and support throughout the process of writing this thesis. Your expertise, insights, and mentorship have been instrumental in helping me complete this project successfully.

Thank you to each and every one of you for making this journey an unforgettable and fulfilling experience that I will treasure forever.

Abstract

In this thesis, we design and conduct two experiments (*i.e.,* a baseline and its replication), with the participation of final year Master's degree students in Computer Sceince, and present 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 testing 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). During the baseline experiment we provided the participants with two ESs to implement and test in a host development environment, while mocking the underlying hardware platform. As for the replication study, its main goal is to validate the results obtained from the baseline experiment and to generalize them to a different and more real setting. In particular, during the replication experiment, we asked participants to implement another ES, which this time had to be effectively deployed and tested on a hardware platform (*i.e.,* a *Raspberry Pi* model 4). Given the small number of participants to this study, we consider this assessment the first exploratory step to the research on the topic of TDD for ESs, given the fact that there are no similar studies available yet. As a result, the gathered data cannot prove a statistically significant difference between the two approaches; however, this data provided interesting cues to determine where to head with a future, more extensive, research on the topic. From the two experiments we gathered quantitative data; moreover, qualitative data was also gathered to explain the quantitative measurements obtained, and to provide a better understanding of the phenomenon under study. After the analysis, data suggests that the external quality of the developed implementation and number of written test cases increase when using TDD to develop ESs, while there is not a substantial difference with respect to the developers' productivity. Finally, developers' perspectives highlight how TDD is perceived as a more difficult approach to apply compared to NO-TDD.

Keywords: Embedded Systems, Empirical Assessment, Test-Driven Development.

Table of Contents

List of Figures

List of Tables

1. Introduction

A computer hardware and software combination designed 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, or airplanes); moreover, such systems are often subject to tight resource constrains (*e.g.,* small battery capacity or limited memory and CPU speeds), a characteristic that has made ES development prone to more challenges compared to traditional software systems. Nonetheless, the global ESs market is expected to witness notable growth: a recent report evaluated it at $89.1 billion in 2021, an amount that is projected to reach $163.2 billion 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 systems (in electric and hybrid vehicles) and in the number of ESs-related research and development projects.

Today, there has yet to be shown which approach is more effective for ESs development; for example, Greening [2] in his book asserted that embedded developers can benefit from the application of Test-Driven Development (TDD), an incremental approach to software development in which a developer repeats a short cycle made up of three phases: *Red*, *Green* and *Blue* (or *Refactor*) [3]. During the *Red* phase, the developer writes a test case for the chunk of functionality to be implemented; since there is no corresponding production code just yet, the test will fail. In the *Green* phase, only the code that is strictly required to make the test pass is written; at this point the developer re-runs the test and watches it pass. Finally, during the *Blue* phase, the implemented code, as well as the respective test cases, is refactored and improved; the developer will then run the tests again, as well as all the previous tests, in order to ensure that the functionality has been kept intact.

TDD has been conceived to develop "regular" software, and it is claimed to improve software quality as well as developers' productivity [4]. ESs have all the same challenges of non-embedded systems (NoESs), such as poor quality, but add challenges of their own [2]: one of the most cited differences between embedded and NoESs is that embedded code depends on the hardware platform it is/will be deployed to. While in principle there is no difference between a dependency on a hardware

device and one on a NoESs [2], 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; for example, this 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.

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, however, no investigations have been conducted to assess possible benefits concerning the application of TDD on the development of ESs. In order to improve our body of knowledge on the matter, and analyze the possible benefits deriving from the application of TDD in this growing field, as the focus of this thesis we investigate the following primary research question (RQ):

> **RQ.** To what extent does the use of TDD impact the external quality and developers' productivity of the developed ES?

To answer this RQ, we present the results of an empirical assessment made up of two experiments, a controlled one that acts as the baseline for our evaluation, and its replication, conducted to study the impact of the TDD approach on the implementation of ESs, with the goal of increasing the body of knowledge on the benefit (if any) of this development approach. The study was conducted with the participation of final year Master's degree students in Computer Science enrolled in the *Embedded Systems* course at the University of Salerno, in Italy.

In our study, TDD has been analyzed 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, for the baseline experiment, and for part of its replication, participants were asked to employ a mocked implementation for the target hardware platform while they developed an ES; these mocked components would intercept commands to and from the device simulating a given usage scenario; this included receiving signals from other sensors, triggering actuators, and overall adopting the implementation details of the hardware platform. 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 it on the actual hardware platform they had mocked up to that point, and test their implementation by running a small test suite in real time. The logic for all three

ESs (two in the baseline and one in the replication study) was deployed targeting a *Raspberry Pi* model 4 board; as for the test cases, for the first two experimental tasks, the acceptance test suites were executed on the mocked implementation on a host developing environment; the same thing happened for the final task, before running a subset of the test suite in the real hardware environment.

Variations in the replication experiment (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. We gathered qualitative data in both experiments to back up the quantitative measurements and to have a better picture of the phenomenon under study. Gathered data suggest that the external quality of the developed ESs and the number of written test cases increase when using TDD with respect to NO-TDD, while there is not a substantial difference with respect to the developers' productivity. Finally, as highlighted by the participants' interviews after the deployment of the final implemented ES, TDD is considered more difficult to apply in this context.

To end this introduction, we provide the structure for the remaining six chapters of this thesis. Chapters 2 and 3 act as the foundational knowledge concepts that provide a general overview on the main topics concerning the TDD methodology and ESs concepts. More in detail, Chapter 2 contains an overview on the software testing process, analyzing the main approaches, with a focus on TDD. Similarly, Chapter 3 will provide information on the most characteristic ESs topics, discussing the enabling technologies and implementation challenges, before examining the techniques for testing such systems, including a pipeline proposed to introduce the TDD cycle into this field. In Chapter 4, we conduct a review of the literature by surveying relevant previous empirical studies on the application of TDD, and literature mappings on the current testing methodologies for ESs. Chapter 5 will contain the detailed explanation of our approach for the definition of the two experiments, and the analysis of the results, while Chapter 6 will focus on our answers to the research questions, the possible implications of this study for both lecturers and researchers, as well as the identified threats to its validity. Chapter 7 will close our empirical assessment by providing the conclusions to the thesis, along with a discussion on the possible directions the research could take when moving forward in the analysis of TDD for ES development. This chapter will also provide a section where my personal contribution to this research, as the author of this thesis, will be discussed.

Finally, the appendix of this thesis serves as an extension of the main text and provides additional information on our experimental approach, from the details concerning the experimental tasks, to the questionnaires and interviews conducted

with the participation of the students. Moreover, the last document provided in the appendix is the article version of this thesis, which we submitted to the 24th International Conference on Agile Software Development [6].

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 the 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, its activities must be performed under time and budget constraints [7], therefore testing every possible configuration of the parameters of a system is unfeasible and impractical. 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, each having has its own 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, different techniques are often used in combination with each other to guarantee that a software product is thoroughly tested and conform to its specification. More in detail, the main testing techniques are:

- **Unit Testing**: a method of testing individual units or components of a software product in isolation; its goal is to verify that each of these units is working correctly and meets its expected behavior. 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 intelligently searching among the input space for the program.
- **Integration Testing**: two or more modules of an application are logically grouped together and tested as a whole. The focus of this type of testing is to find the defect on interface, communication, and data flow among modules.

The top-down or bottom-up approach is used while integrating modules into the whole system.

- **System Testing**: a method of testing a complete software product against the specified requirements; can involve different activities, such as end-to-end testing - where it is ensured that the software meets the expected flow of operations when running in a real-world environment, on different hardware combinations, operating systems or web browsers, and with different input configurations - or smoke testing, where the goal is to verify that basic and critical functionality of the system under test is working fine at a very high level.
- **Acceptance Testing**: a method of testing a software product to ensure that it complies with the needs and expectations of the end user. The goal here 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.
- **Performance Testing**: one type of non-functional testing technique. It revolves around 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**: a kind of security testing, where a system, network, or application is exercised with the objective of identifying vulnerabilities that an attacker could exploit; this evaluation happens by simulating an attack and identifying any weaknesses that could be leveraged 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. While it has been employed for long, especially in monolithic applications, it has a few weaknesses, most notably not allowing for much iteration or flexibility during the development process: 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.

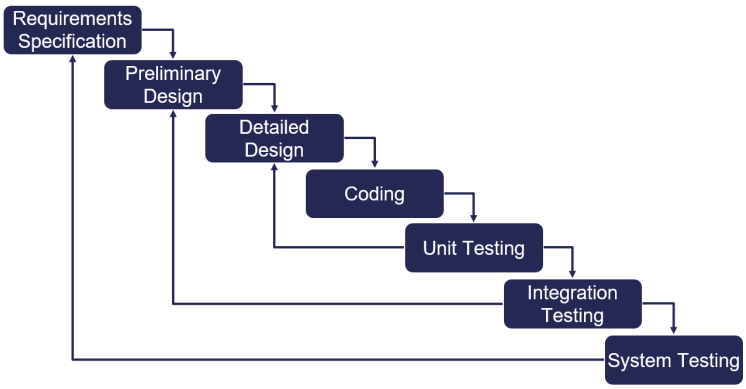


Figure 2.1.: The Waterfall model.

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 SDLC 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 iterations, or "sprints": 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:

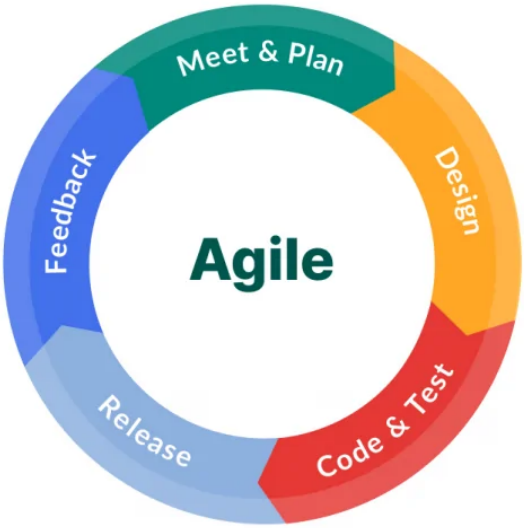


Figure 2.2.: The Agile model.

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**: during 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; this allows developers to gather feedback on the software before the final release.
- **Feedback**: the team reviews the feedback received from customers and stakeholders and makes any necessary changes to the software. Since this feedback is also incorporated into the development process in an iterative manner, it allows 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 one, 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 practiced 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, firstly introduced in 2003 by Kent Back in the book *"Test-Driven Development By Example"* [3]; while there is no formal definition of the process, the goal is to *"write clean code that works"*, as the author states. With TDD, test cases are written before any production code: these tests are used to define the requirements for the system and to guide the development process, with the objective of this practice being for all these tests to pass before the development is complete, by continuously running the entire test suite as more features are tested and built; this can help to ensure the quality and reliability of software. Moreover, TDD encourages software developers to write small, isolated units of code that are easy to test, maintain and understand, and will ultimately act themselves as a form of documentation for the developers. Compared to traditional testing and SDLC approaches, TDD is an extremely short, incremental, and repetitive process, which make it strongly 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, with the source code 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 did not result in any unexpected behaviors in other components.

Each new unit of code requires a repetition of this cycle [8]. Figure 2.3 provides a visual representation of the TDD process.

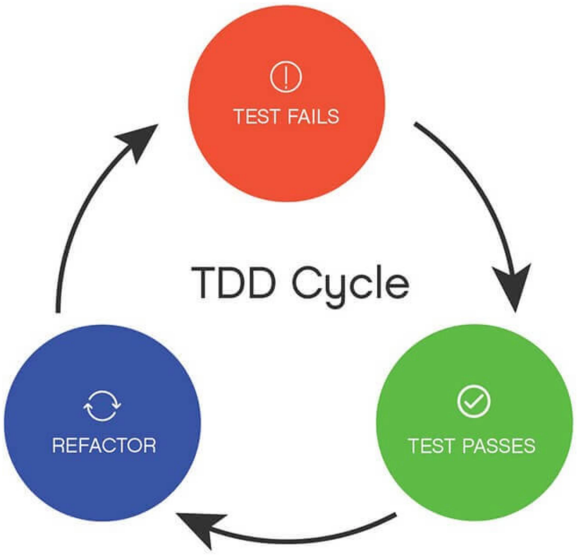


Figure 2.3.: The Test-Driven Development cycle.

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 scheduled to achieve an iterative implementation of the system. The specification of user stories can also vary in granularity: when using a fine-grained structure to describe a 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 [9]. 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 [9].

2.2.2. TDD advantages and challenges

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

- **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 called 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 and maintainability**: 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 easier to understand, without any extra pieces of functionalities that may not be needed, leading to a more maintainable architecture in the long run.
- **Regression testing**: by incrementally building a test suite as the different iterations of TDD are performed, we allow the system to always be ready for this suite to run whenever new changes or functionalities are pushed to its codebase.
- **Improved code readability and documentation**: well-written tests serve as documentation for the code, making it easier for new contributors to understand it.
- **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.

Similarly, however, the TDD approach can result in a series of pitfalls and challenges:

- **Learning curve**: TDD requires a different way of thinking about software development, which can be challenging for some developers to learn.
- **Overhead of maintaining tests**: maintaining a comprehensive suite of tests requires time and effort, and can become a burden for developers as the codebase grows.

- **False sense of security**: as with traditional testing, if tests are not comprehensive or not properly written/maintained, they can give a false sense of security about the code's quality.
- **Rigidity**: TDD can lead to a rigid and inflexible code structure, as the tests dictate the core design of the application; this can limit creativity and experimentation, as developers may be reluctant to change the code if they know it will break the tests.
- **Difficulty in testing certain types of code**: user interfaces or performance-critical code can be challenging to test using TDD; developers may still need to use additional testing methods or tools to ensure adequate coverage for these types of code.
- **Slower initial development**: writing tests before code can slow down the initial development process, as it requires additional time and effort. However, as we saw, this investment should generally pay off in the long run by reducing the need for reworks and debugging.

Keep in mind that the benefits and challenges of TDD discussed in this section are often not based on empirical evidence.

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 characteristic 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 guarantee 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, 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 there is a need for the system 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; the global ESs market, in fact, 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.

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 Internet of Things (IoT) purposes, is the Arduino, an open-source platform that is based on the Atmel AVR microcontroller and is widely used in hobbyist and educational projects because of its simplicity and low cost. Many Arduino versions and revisions exist, each with a different form factor, amounts of on-board resources, and I/O pins; as a result, they are 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, with the highest amount of analog/digital pins and resources.

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 also comes in different variants, from an Arduino Nano-sized - but equipped with much more resources - Raspberry Pi Zero and Zero W, to the Pi 3 and Pi 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 their high specialization, these microcontrollers often offer limited customization capabilities, and may not be easily programmable, if at all, by the user. Some examples 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 are usually 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 optimized for low power consumption. Popular EOSs include TinyOS and Embedded Linux. Moreover, the real-time requirements of some ESs call 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 ESs 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, meaning 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 any given 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 fast and can support multiple devices on the same bus, however, it requires more wires and pins compared to I2C and as a result 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 [10]; 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 nowadays 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, this protocol 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 protocols designed for IoT and machine-to-machine (M2M) applications; a typical use case is the periodical 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 means are impractical. For network communications, the Internet Protocol (IP), and it's low energy variant 6LoWPAN, are widely employed 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), or the Extensible Messaging and Presence Protocol (XMPP) are a common choice to implement queues of message for the applications to query; moreover, other general purpose protocols for limited devices (*e.g.,* the Constrained Application Protocol, CoAP) are adopted by applications to communicate.

3.3. Design 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, for both the hardware and the software, to ensure that the system meets all of its requirements, functional and, especially in this case, non-functional. In view of this, the main challenge resides in balancing the trade-offs between performance, power consumption, and cost. For example, increasing the performance of an ES in order to accommodate a certain functionality's needs may require more power-hungry components, thus increasing the cost of the system and potentially conflicting with another requirement, according to which the device must be able to operate on a battery for long periods of time. Similarly, reducing power consumption to reach a power target may come at the expense of performance. Going through multiple hardware revisions in order to meet the requirements and fine-tune power and computational behavior iteratively can be extremely expensive. For this reason, as we discussed, general-purpose microprocessors should be considered in the initial design phase; in most real systems however, there is the need for custom-made hardware, tailored according to the system's requirements, for security and reliability reasons. Regardless of the microcontroller, optimization steps can be performed in most cases and involve using specialized programming languages and techniques, such as RTOSs and low-level hardware access, implementing power-saving modes and using low-power components to minimize consumption.

Handling failures in ESs is another critical aspect to consider during their design phase: any failure should always be evident and identifiable quickly (a heart monitor should not fail quietly) [11]. Given the high criticality of many such systems, ensuring their dependability over the course of their lifespan is essential: ESs can be deployed in extreme conditions (*e.g.,* weather monitoring in remote 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. ESs should implement robust error-detection and -correction mechanisms, as a way to improve reliability.

- **Availability**: the degree to which an ES is operational and accessible to its users; important for systems that are used in critical applications, such as transportation or medical equipment. To improve availability, ESs should be designed with multiple levels of redundancy and with robust fault-tolerance mechanisms; additionally, it is important to ensure that these systems are able to handle the challenges of limited bandwidth, high latency, and unreliable connectivity.
- **Security**: the ability of a system to protect against unauthorized access, modification, or destruction of data; crucial for systems that handle sensitive information, such as financial transactions or personal data. In order to improve security, ESs should be designed with robust encryption, security protocols, and authentication mechanisms, and should be thoroughly tested for vulnerabilities.

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. Moreover, an ES can be unreliable due to its data being corrupted by an external attack or due to poor implementation. As usual, ensuring the validity of these dependability attributes in a real system, with respect to ES constraints, requires trade-offs and compromises.

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, performing field-testing of these systems can be impractical, unfeasible, or even dangerous due to the 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 [12], 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 [13]. As the next step, the controller module is created, and it is verified that it can manage the model, as per the requirements. Commonly, after the testers establish the correct behavior of the controller, its inputs and outputs are recorded, 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 [14]. 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 the Controller Area Network (CAN) protocol and the User Datagram Protocol (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 platform. A common environment for performing the simulation steps discussed above is Simulink [15]; it is a graphical modeling and simulation environment for dynamic systems based on blocks to represent different parts of a system, where 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 displays part of a Simulink model built to represent an autonomous vehicle system.

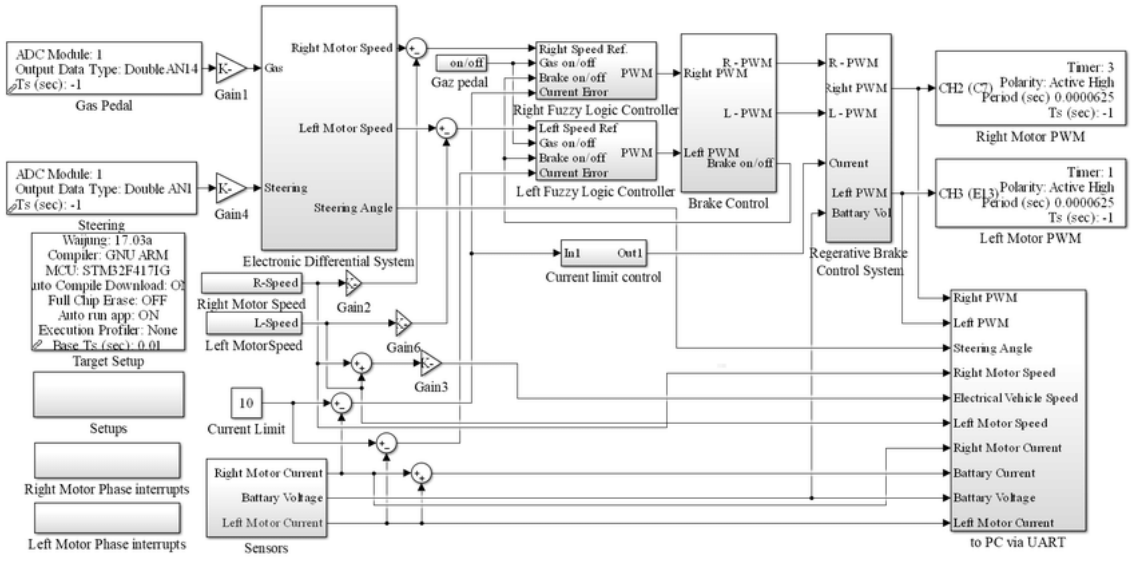


Figure 3.1.: Some components of a Simulink model for an autonomous driving system.

3.4.2. A TDD pipeline for Embedded Systems

Given the potential benefits of TDD for NoESs, asking whether this technique could be applied to the development of ESs comes natural. By approaching ES with TDD in mind, developers would write automated tests that would be used to verify the functionality and performance of the ES under test: as a reminder, these tests would be written before any production code is written, and they would be used to define the requirements for the system. Given the higher costs of fixing issues in ESs, applying TDD could allow developers to identify and fix problems earlier in the development process, before they become more difficult and costly to fix; it could also help to ensure that the system meets some of the requirements proper of ESs and that it is reliable, available and secure.

For the same difficulties that are proper of ES testing, TDD can be a challenging approach when practiced on ESs. However, with the right tools, techniques, and experience, it is possible to use it with and to achieve its benefits; often in fact, quality in embedded software is tied to platform-specific testing tools geared towards debugging [16], so employing a test-first approach should reveal beneficial. The benefits that would result from the usage of TDD include:

- Reduction in risk by verifying production code, independent of hardware, before hardware is ready or when hardware is expensive and scarce.
- Reduction in the number of long target compile, link, and upload cycles that are executed by removing bugs on the development system.
- Shorter debug times on the target hardware where problems are more difficult to find and fix.
- Isolation of hardware/software interaction issues by modeling hardware interactions in the tests (*e.g.,* by mocking the hardware components).
- Improvement in software design through the decoupling of modules from each other and the hardware; testable code is by necessity, modular.

In [2], the author proposes the *"Embedded TDD Cycle"*, as a pipeline made of the following steps to responsibly and safely apply TDD to ESs:

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.

Moreover, 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 [2], the risk is real. If it's late in the development cycle, and a reliable target hardware is already available, this stage may reveal 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, results in a more complicated build automation process.
5. **Run acceptance tests in the target hardware**: Finally, in order to make sure that the product's 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 cannot be fully tested automatically is tested manually.

Would this pipeline reveal beneficial, it could provide interesting cues for more research of TDD applied to ESs.

4. Literature review

4.1. Empirical studies on Test-Driven Development

In the past, the Empirical Software Engineering community has taken interest into the investigation of the effects of TDD on several outcomes [17] [18] [19], including the ones of interest for this study, *i.e.,* software/functional quality, productivity, and (more rarely) number of written test cases. These studies are summarized in Systematic Literature Reviews (SLRs) and meta-analyses (*e.g.,* [20, 21, 22, 23]).

The SLR by Turhan *et al.* [23] 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.* [20] conducted another SLR that includes 27 primary studies ranging from 1999 to 2014, with the aim of locating, selecting, and evaluating available empirical studies on the application of TDD in software development, targeting the effects that this practice produces on productivity and internal and external software quality. Similarly to Turhan *et al.* [23], the authors observed an improvement of functional quality due to TDD, while results are inconclusive for productivity. Moreover, one of the opportunities identified by this SLR for future research on the topic concerns the expansion of the TDD practice to other paradigms and software development technologies, as most research is focused on simple examples using the *Java* programming language.

Rafique and Misic [22] 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.* [21] 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.* [24], where the authors conducted a three-year-long case study about the use of TDD at Nokia-Siemens Network, during which 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; furthermore, productivity increased due to the team's 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.* [25] 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.* [26] conducted a retrospective analysis of a number 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 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 [27] studied the capability of 30 professional software developers of different seniority levels (junior, intermediate, and expert) to develop a complex 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 training session on TDD. After this session, the participants were able to correctly apply TDD (*e.g.,* following the three-steps micro-cycle): they followed the TDD cycle between 80% and 90% of the time, but initially, their performance depended on experience, as the seniors developers only needed a 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 [27] 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.

Similarly, Romano *et al.* [28] researched the retainment of TDD knowledge and the effect on developers' productivity and external quality of software in a longitudinal study with the participation of Computer Science students; while their findings suggests that using TDD does not result in a significant statistical difference on the latter constructs, employing this development practice allowed participants to write more test cases. Furthermore, this ability of novice developers to produce more tests using TDD compared to NO-TDD was retained over time (6 months).

Although the above-mentioned studies 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. Related works on Embedded Systems testings

Garousi *et al.* [29] provided a Systematic Literature Mapping (SLM) for ES testing by reviewing 312 papers, concerning the types of testing activity, types of test artifacts generated, and the types of industries in which studies have focused, with the goal of identifying the state-of-the-art and -practice for ESs testing. Topics such as model-based and automated/automatic testing, test-case generation, and control systems were among the most popular and discussed ones in the articles they examined. Most of the review papers (137 of 312, around 43.9%) presented solution proposals without rigorous empirical studies; 98 (31.4%) papers were weak empirical studies (validation research). 36 (11.5%) were experience papers. 34 were strong empirical studies (evaluation research). 2 and 5 papers, respectively, were 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; among the ones focused on unit testing, only two of the sources

([30], [31]) applied TDD to embedded software. Several practical examples of automated unit test code were provided. The most popular technique for deriving test artifacts was requirements-based testing: 159 papers (58.4%) discussed it, and most of them (142, 52.2%) used model-based testing, which falls into the X-in-the-loop development paradigm. Model-based testing employs forward or backward engineering to develop models; once these models are validated, they can be used for test-case design (*e.g.,* Finite-State Machines, FSMs, and their extensions are frequently used to derive test-case sequences, employing coverage criteria such as all-transitions coverage.) As for the type of test activities, there was 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). Finally, as the authors' state, there is a need for future research to conduct empirical studies providing industrial evidence on the effectiveness and efficiency of embedded software testing approaches (including TDD) in specific contexts to further improve decision support on the selection of embedded software testing approaches beyond their SLM.

5. Experimentation

5.1. Overview

In this chapter we will present in detail the planning and approach we followed to establish the experiments on the application of TDD for ESs, and provide an initial analysis of the results.

A controlled experiment, the baseline (*Exp1*), followed by its replication (*Exp2*), was conducted with the participation of 9 undergraduate final-year Master's degree students enrolled in the *Embedded Systems* course at the University of Salerno, in Italy. Participation in the experiments was agreed upon by the students and was voluntary, with the outcome not directly affecting their final mark for the exam; students were however awarded 2 bonus points, in order to encourage their participation. Before taking part in the first experiment, a set of lectures and training sessions was held with the objective of providing the participants with a common body of knowledge on the topics tackled by the experiments, namely unit testing, test scaffolding, and the TDD approach.

Afterwards, the participants were partitioned into two groups and tackled the first task of *Exp1*, one group using TDD, the other using NO-TDD; this task, *IntelligentOffice* (*IO*) required the implementation of a system responsible for handling various parameters and functionalities inside a smart office, including the detection of workers, handling the lights and monitoring the CO2 levels inside the office. For the second task, *CleaningRobot* (*CR*), the approach followed by the two groups was inverted, namely the group that implemented the first task using TDD switched to NO-TDD and vice versa. This task concerned the development of an ES to manage a small cleaning robot which had to move in a room to clean dust, while avoiding obstacles along the way. After each task of *Exp1* the participants were asked to fill a questionnaire to express their feelings about the experience.

For (*Exp2*), the participants had to develop an additional ES, this time at home on their own (*i.e.,* not under our supervision in a laboratory of the university), before submitting their implementation and deploying it on a real hardware platform, a

Raspberry Pi Model 4, and using real sensors and actuators. This task, *SmartHome* (*SH*), was focused on the implementation of an intelligent system which allowed for handling light, temperature, and gas levels inside a room. Since *Exp2* was only made up of this task, the students were randomly assigned one of the two approaches (*i.e.,* TDD or NO-TDD) upon receiving the instructions for implementing this ES. The students were given a deadline to implement the task, and were asked to book a time slot, during which they would have deployed their implementation on hardware and would have tested it in real time by observing the behavior of the sensors and actuators; moreover, each participant was individually interviewed after the hardware deployment step, in order to gather qualitative data regarding their feedback on the experiments.

Following the three tasks, we extracted the statistical values for a set of predefined dependent variables by running the acceptance test suite we prepared for each task on the implementations delivered by the participants. These values, as well as the submitted post-questionnaires and final interview, were our primary subject of analysis to answer the established research questions on the impact of TDD on the external quality, productivity, and number of written test cases of the developers' implementations when tackling ESs.

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. Furthermore, it has both research and educational goals: on one hand, we conceived the study to answer our RQs; on the other, the study allowed the participants (students) to gain practical experience with TDD applied to ESs. As for the educational goal, TDD is a development approach widely used in several contexts [32], and it seems to be promising in the development of ESs too [2].

Although the preliminary and exploratory nature of our investigation, it has the merit to study for the first time the application of TDD in a new development context, namely that of ESs; therefore, our results can have several practical implications, for both lecturers and researchers. For example, they could provide initial evidence on the application of TDD to the development of ESs as to justify future research on this matter and/or promote or discourage its adoption in an industrial setting.

5.2. Research questions

Following the main research question presented in the introduction chapter of the thesis, we defined the main goal of this study by applying the Goal/Question/Metrics (GQM) template [33]. According to this template, 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. We identify our goal as follows:

> **Analyze** the use of TDD **for the purpose of** evaluating its effects in the development of ESs **with respect to** the external quality of the implemented solution, the developers' productivity, and the number of written test cases **from the point of** view of the researcher and lecturer **in the context of** an ESs course involving second year Master's degree students in Computer Science.

According to this objective, the following research questions were defined:

- **RQ1.** Does the use of TDD impact the quality of the developed ES? And if so, to what extent?

  Aim: The answer to this RQ has practical implications, especially within the Agile community: for example, in case the use of TDD positively affects the quality of ESs developed within implementation tasks, it would mean that it is the case to teach this development approach in academic contexts, with the ultimate goal of facilitating the adoption of this Agile technique in the industry. That is, if the newcomers of the working market are familiar with TDD (because they learned and experienced it in the academic context), and it is shown that it produces ESs with improved quality, then the software industry could be encouraged to migrate their development from NO-TDD to TDD.

- **RQ2.** Does the use of TDD increase developers' productivity when developing ESs? And if so, to what extent?

  Aim: A positive answer to this RQ has practical implications, since it would help us to further improve our body of knowledge in the context of TDD applied to the development of ESs. For example, some software companies operating in the context of the development of ESs could be encouraged to use TDD in case: *(i)* there is 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).

- **RQ3.** Does the use of TDD increase the number of test cases written by developers when developing ESs? And if so, to what extent?

  Aim: If the adoption of TDD were to result in a higher number of written test cases when developing ESs, using this approach would be beneficial to the industry. Not only more test cases would potentially cover more of the implemented source code, thus increasing reliability; additionally, the fact that tests are written as the first thing when using TDD, a higher number of them would provide more documentation for developers and new contributors to the system, especially for those aspects of ESs that concern lower-level details, such as hardware interaction, and are generally harder to test. Finally, more tests would open up more refactoring opportunities when using TDD; this could help with the optimization steps commonly performed in many ESs.

5.3. Participants

The participants for both *Exp1* and *Exp2* were Computer Science 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. Both groups were enrolled in the *Embedded Systems* course at the University of Salerno, however not all students had a strong Computer Science background, and among those taking the course, 9 participated.

Participating in the experiments was voluntary: the students were informed that *(i)* any gathered data would be treated anonymously and shared for research purposes only; *(ii)* they could drop out of the study at any time if they wished to do so, and *(iii)* they could achieve the highest mark in the course even if they did not participate. Finally, as an incentive to encourage participation, those who took part in the study were rewarded with 2 bonus points in their final mark for the course, in line with Carver *et al.*'s advice [34].

Before *Exp1* took place, we collected some information on the participants' knowledge and experience with general programming and testing concepts, in order to get a general idea of their personal preparation before the studies. To this end, the participants were asked to fill out a form in which they had to rate their experience using an interval-scale questions system (where 1 means "very inexperienced" and 5 means "very experienced"). All the participants had programming experience and most of them rated such an experience as 3 out of 5 on the scale. As for testing, the

participants were mostly not experienced with unit testing, since most of them chose 1 or 2 on the scale to represent their unit-testing level of experience; finally, three students had heard about TDD, from another university course, but none had had a practical experience with this technique before the study.

The participants for the experiments 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 in.

5.4. Experimental tasks design

The experimental objects designed for the study are three code katas, *i.e.,* programming exercises aimed at practicing a technique or a programming language: in this case the design of a small ES, which would take around 2 hours to implement. All three tasks were designed according to a target platform, a *Raspberry Pi Model 4*, and using the *Python* programming language. As for why this environment was chosen, compared to other programming languages and platforms typically used for ESs or IoT projects (*e.g., Arduino* with *C++*), in our opinion designing the tasks with a higher level language such as *Python* - which is still employed for ESs (either in its standard form or in its *MicroPython* variant), as well as for Web and Enterprise applications, as reported by IEEE Spectrum [35] - allowed us to focus more on the implementation and logic details than we would have done by designing the tasks orbiting around lower level language features and mechanisms. Furthermore, if we also take into account that some participants had a limited, mostly theoretical, knowledge of ESs prior to the course's end, we think that introducing these practical concepts with *Python* made it easier for them to get a grasp on the main implementation concepts for ESs. Finally, participants relied on the *PyCharm IDE* to implement all three tasks, with the native `unittest` package as the testing library.

The target platform was considered even during the design of the two tasks of *Exp1*, which in the end did not have to be effectively deployed on hardware; this was done in order to make the mocked implementations developed by the participants resemble as closely as possible the embedded implementations, in line with the approach proposed by Grenning [2]. The main mocked component utilized was a facade for the *Raspberry Pi*'s General Purpose Input/Output (GPIO) library, available as open source software on GitHub [36]. Other mocked components included third party libraries for some individual sensors and actuators, like for the *DHT11* temperature and humidity sensor. For *Exp2*, another factor that influenced the design of the code

kata was the availability of the hardware at the university, including sensors and actuators, as well as how hard/practical these would have been to test in real time; for example, testing a gas sensor by actually releasing gas close to it for extended periods of times could be dangerous. For some sensors, on the other hand, this was not a concern since it was possible to manually trigger them by rotating an on-board potentiometer, effectively lowering or increasing the detection threshold for their respective measurements. The task developed for *Exp2* was later deployed and tested inside a laboratory at the University of Salerno: each participant had their project uploaded on a *Raspberry Pi model 4* board and assisted as we ran a small acceptance test suite in real time and logged the results (*i.e.,* which test cases passed and which failed).

For *Exp1*, the experimental tasks were:

- ***IntelligentOffice*** (***IO***): task revolving around the implementation of a smart system to manage an office, by using different sensors and actuators to handle various aspects inside it. The system was responsible for detecting the presence of workers in four quadrants of the office by using infrared (IR) distance sensors, manage the light level in the room with a smart light bulb, a photoresistor (or Light Dependent Resistor, LDR), and a set of blinds controlled by a servo motor; the motor opened/closed the blinds based on the time of day and day of week measured by a Real Time Clock (RTC) module. Finally, the system would monitor the CO2 levels inside the office, with the possibility of triggering an air vent system to balance them by expelling the air outside.
- ***CleaningRobot*** (***CR***): as the name suggests, this task required participants to implement an ES to manage a small cleaning robot. This system received command strings from an external management unit, and had to move/rotate the robot inside the room accordingly, using two Direct Current (DC) motors, one for the wheels, and one for the robot itself. Besides this, the robot had to be able to detect obstacles in front of it by using an IR distance sensor, and had to communicate its position and the last encountered obstacle back to the remote management system after performing an action. Finally, an Intelligent Battery Sensor (IBS) was used to check the charge left of the internal battery of the robot, and a LED was turned on to signal the need for a recharge; similarly, the cleaning system of the robot would be turned on/off according to the measured remaining battery level.

As for *Exp2*, the replication experiment, the ES to implement was the following:

- ***SmartHome*** (***SH***): this final task concerned the development of a system which allowed the handling of light, temperature, and gas levels inside a room.

> By using an IR distance sensor, a photoresistor and a smart light bulb, the system managed the light level inside the room to avoid wastefully turning the light on when the user was not in the room or when it was already bright enough due to natural light. Furthermore, two temperature sensors, one on the inside and one on the outside of the room were used to collect measurements, based on which a servo motor opened or closed a window in the room. Finally, the system was also equipped with an air quality sensor to measure the gas levels inside the room; based on these measurements, in case of high amounts of gas particles detected in the air, an active buzzer would trigger an alarm to notify the user.

Further details on the experimental objects, including the list of user stories, sensors and actuators employed, and other information, are provided as an appendix to this thesis (A, B, C).

The material provided to the participants for each task included: *(i)* a document containing the description of the task, made up of a high level description of the purpose of the system, instructions on the development approach (*i.e.,* TDD or NO-TDD), and the list of user stories to implement, with each detailing the corresponding class/method to modify in the source code; and *(ii)* a link to project template for the *PyCharm IDE* that the participants had to import in their environment (*i.e.,* by either forking/cloning the corresponding GitHub repository or by downloading a zip file); this template contained method stubs (*i.e.,* empty methods exposing the expected API signatures), utility functions, mocked libraries for the GPIO features and other components, as well as an example empty `unittest` test class.

For each of these tasks, we prepared a corresponding acceptance test suite, as a mean to evaluate the implementations delivered by participants, and extract the metrics on which to base our empirical assessment.

5.5. Study design

The *Embedded Systems* course, during which the study was conducted, started in September 2022 and 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 ESs testing techniques.

As the survey pre-*Exp1* highlighted, few students had unit testing experience, and almost no participants ha dealt with TDD up to that point: for this reason, both

topics were covered through a series of frontal lectures and exercise/homework sessions in the weeks preceding the *Exp1*. All the participants attended these lectures and training sessions during which they had some theoretical and hands-on experiences with the main topics they would encounter during the future experimental tasks. The overall schedule for training sessions and experiments was as follows:

- **Day1.** Frontal lecture - Introduction on unit testing and its guidelines - Interactive exercise on unit testing.
- **Day2.** Frontal lecture - Test scaffolding and *Raspberry Pi*'s GPIO library - Interactive exercise on test scaffolding.
- **Day3.** Frontal lecture - Introduction to TDD- Interactive exercise on TDD.
- **Day4.** Training task - TDD exercise and homework with the same structure as the future experimental task (*i.e.,* project template, a set of user stories, and 2 hours deadline).
- **Day6.** First task for *Exp1* (*IO*).
- **Day7.** Second task for *Exp1* (*CR*).
- **Day8.** Start date for the task relative to *Exp2* (*SH*).

The first task, *IO* took place on Tuesday, December 6^{th} 2022, while the second, *CR* took place on Tuesday, December 13^{th} 2022; both tasks required around 2 hours to be implemented by the participants. Finally, the start date for *Exp2* (*i.e.,* the date when we sent out the documentation to the students) was December 19^{th} 2022, and the participants had to deliver their implementation by January 8^{th} 2023.

The design of *Exp1* is an ABBA crossover [37]; it is a kind of *within-participants* design where each participant receives both treatments (*i.e., A* and *B* or, in our case, TDD and NO-TDD). In ABBA crossover designs, there are two sequences (*i.e., G1* and *G2*), defined as the order with which the treatments are administered to the participants, and two periods (*i.e., P1* and *P2*), defined as the times at which each treatment is administered; the experimental groups correspond to the sequences. Also, to mitigate learning effects, each period is paired with a different experimental task.

For the first period *P1*, the group *G1* was assigned the TDD version of the first task, *IO*, 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, *CR*, while the group *G2* was assigned the TDD version. Therefore, at the end of *Exp1*, every participant had tackled each experimental object only once. As for *Exp2*, 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, *SH*; specifically, 5 students ended up in the TDD group, while 4 formed the NO-TDD group. As a result, the design of this study is *one-factor-with-two-treatments* [38], a kind of *between-participants* design.

After each period of *Exp1*, participants were asked to fill out an online questionnaire, with the purpose of describing their general experience with the implementation of the task, focusing on their perceived complexity and 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 group, with the latter having an additional question, as the first open-ended one, asking to provide information about the chosen approach for testing. Furthermore, the post-questionnaire presented at the end of period *P2* for *Exp1* contained an additional open-ended question: here, participants had to provide their feelings towards both testing practices, TDD and NO-TDD, and compare them based on the two encountered tasks. 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 this testing approach (*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, these were:

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

Finally, no formal questionnaire was provided for the replication study; however, after the hardware deployment and testing steps, each participant was individually interviewed about their overall experience with the study; the structure of the final

interview had a predefined script, as suggested by King [39]. The topics covered by the interview were:

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 your implementation 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?
   - Provide a discussion on TDD vs. NO-TDD in the development of ESs.

The answer for both the post-questionnaires and the final interviews, as well as their thematic analyses, are available as an appendix to this thesis (D, E).

5.6. Independent and dependent variables

The participants were asked to carry out each task by using either TDD or the approach they preferred (NO-TDD), therefore one of the independent variables considered is ***Approach***, a nominal variable assuming two values, TDD and NO-TDD. The data was collected over two periods for the controlled baseline study (*Exp1*), and over an additional period for the non-controlled replication study (*Exp2*), so a second independent variable is ***Period***, assuming the values $P1$, $P2$, and $P3$. During the three periods both approaches (TDD or NO-TDD) were applied. Finally, since the participants were split into two groups, the last independent variable is ***Group***, which can assume the values *G1* and *G2*.

As for the dependent variables considered in the experiments, these are: ***QLTY***, ***PROD***, ***TEST***, ***CYC***, ***COG***, ***LOC***. The variables $QLTY$, $PROD$ and $TEST$ have been used in previous empirical studies on NoESs [18], [40], [28], [41]. As for the

other variables (*i.e.,* CYC, COG, and LOC), while they could just be used to assess constructs regarding code complexity, they can also be leveraged to further assess the quality of a software solution, since they have an impact on external attributes, such as efficiency, reliability, and maintainability.

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 tackled 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 $[0, 100]$.

The PROD variable estimates the productivity of the solution implemented by 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 in their implementation. 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, these were computed using *SonarQube*, an open-source platform for continuous inspection and static analysis of code [42]; these variables provide additional insight regarding the quality in the implemented software solution, mostly in terms of how hard the production code is to comprehend and therefore maintain.

They can be defined as follows: CYC refers to the cyclomatic complexity metric 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 code module; a program with a lower cyclomatic complexity is generally easier to understand and less "risky" to modify; the variable can also 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 [43]:

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 files and the test code source files.

For both CYC and COG we only considered the production code, since there was no noticeable difference, besides one outlier, between the same metrics in the test files of the participants.

Generally speaking, the higher the values of CYC, COG, and LOC, the worse it is: software with high cyclomatic and cognitive complexity is more difficult to understand, maintain, and extend, and is more prone to errors and bugs. It is good practice to keep these complexities (as well as the number of lines of code, although to a lesser extent) as low as possible for easier maintenance and extensibility.

5.7. Analysis methods

5.7.1. Individual analysis

As a first way to examine the distributions of the dependent variables, we computed some descriptive statistics, for each of them. We organize this information inside a set of tables for each experimental task. Moreover, in order to provide a graphical representation of the data and to summarize the distributions, we employ box plot charts.

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.

5.7.2. Aggregate analysis

Meta-analysis is a statistical method used to combine the results of multiple studies in order to obtain a more precise estimate of the effect of the treatment/intervention; its goal is to increase the power of the analysis and to provide a more robust estimate of

the treatment's effects. There are different methods used to conduct a meta-analysis, but generally, the process involves identifying the relevant studies, extracting data from them, and then analyzing this data by employing statistical techniques. In our case, after considering the mean, standard deviation, and number of participants for each development approach in *Exp1* and *Exp2*, we made use of the Standard Mean Difference (SMD) as the measure of the effect size for the meta-analysis: this value represents the difference between the two groups' mean values, standardized by dividing the difference by the standard deviation; SMD can be used to determine the overall effect size when comparing the results of the different studies. More formally, it is computed as follows:

$$\theta = \frac{\mu_1 - \mu_2}{\sigma}$$

Where μ_1 and μ_2 represent the two means, and σ refers to the standard deviation, used to measure the amount of variation or dispersion of a set of values. We computed the SMD as *Hedges' (adjusted) g.*; to obtain joint SMDs (one for each dependent variable), we leveraged random-effects meta-analysis models. Based on the guidelines proposed by Cohen [44], the SMD can be interpreted as: *negligible*, if $|\text{SMD}| < 0.2$; *small*, if $0.2 \leq |\text{SMD}| < 0.5$; *medium*, if $0.5 \leq |\text{SMD}| < 0.8$; or *large*, otherwise [32].

A forest plot is a common way to visually present the results of a meta-analysis: it is a representation of the effect estimates and their corresponding confidence intervals for each study included in the meta-analysis. The forest plot is divided into two parts: the left side shows the individual study results and the right side shows the overall effect estimate and its corresponding confidence interval; more specifically, the box in the middle of each horizontal line (*i.e.,* the confidence interval, CI) represents the point estimate of the effect for a single study; the size of the box is proportional to the weight of the study in relation to the pooled estimate. The diamond represents the overall effect estimate of the meta-analysis; the placement of the center of the diamond on the x-axis represents the point estimate, and the width of the diamond represents the 95% CI around the point estimate of the pooled effect.

5.7.3. Thematic analysis

Thematic analysis is an approach widely used in qualitative psychology research to analyze data from interviews, focus groups, and other forms of open-ended data; this kind of analysis involves reading through the gathered unstructured data multiple times to identify recurrent patterns, concepts, or themes, and then coding these patterns using a set of codes or labels. At this point researchers would organize the

coded data into themes and sub-themes, which would in turn be used to construct a narrative or a report of the findings.

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 [45]. In template analysis, it is common practice 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.

In our study, we used thematic analysis to analyze the open-ended questions of the post-questionnaires for the two experimental tasks of the baseline experiment, as well as the individual participant interviews of the replication study.

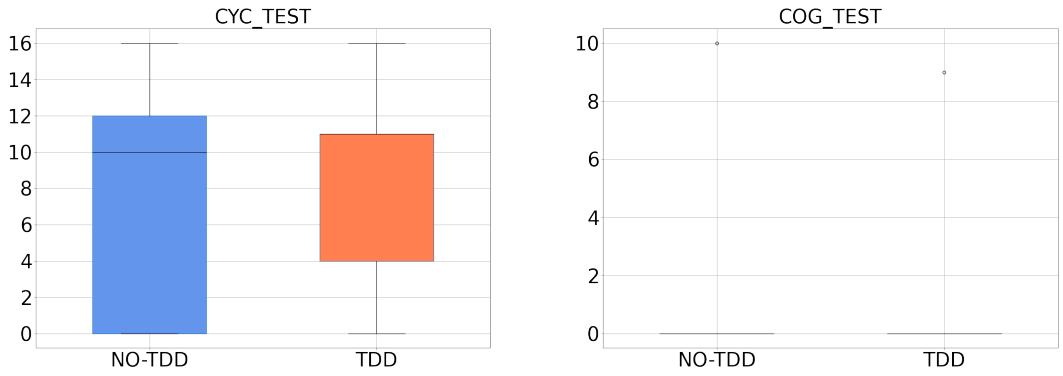
5.8. Results

5.8.1. Dependent variable 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 the considered dependent variables, for the two separate approaches (*i.e.,* TDD and NO-TDD), before aggregating the values of the variables for the first two tasks simultaneously, in order to provide an overview of the differences between the two testing approaches in *Exp1*. Finally, we will compare the results for both experiments, in order to provide the foundation on which to answer our research questions. Besides the tables, we show the box plot charts to visualize the values for the statistics extracted from the dependent variables.

First, as we discussed in the dependent variables' definition section, in the end we did not take into account the cyclomatic and cognitive complexities of the test code written by the participants during the experimental tasks. Figure 5.1 shows the box plot chart for these metrics, relative to the experimental tasks for the baseline study:

Furthermore, we originally intended to also analyze the number of code smells in both the production and test code, however, as figure 5.2 highlights, there was no significant difference between the two approaches.



(a) Cyclomatic complexity (b) Cognitive complexity

Figure 5.1.: Box plots for the cyclomatic and cognitive complexities of test code for the first two tasks

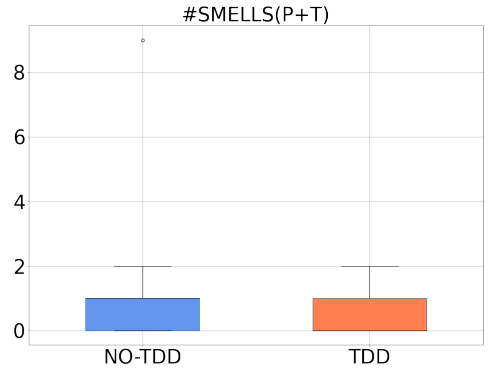


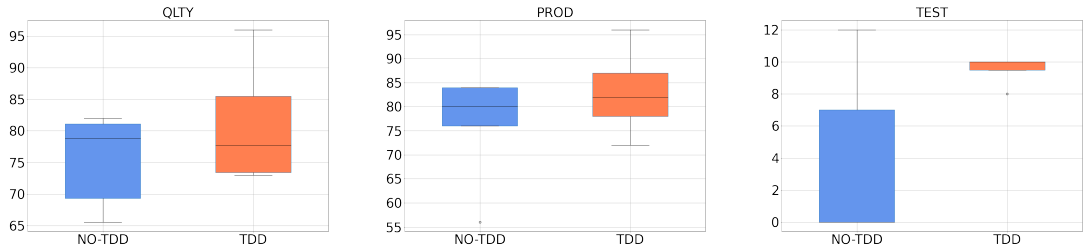
Figure 5.2.: Number of code smells for the two tasks of *Exp1*.

Starting with the first experimental task of *Exp1*, table 5.1, as well as figure 5.3 show the tables containing the extracted statistical measures and the box plot charts for the dependent variables, respectively.

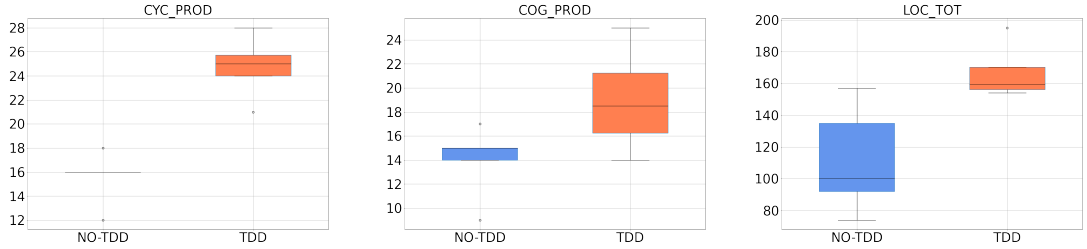
Exp1 - *IO* - TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	73	96	81.12	77.77	10.73
PROD	72	96	83	82	10
TEST	8	10	9.5	10	1
CYC	21	28	24.75	25	2.87
COG	14	25	19	18.5	4.69
LOC	154	195	167	159.5	18.95
Exp1 - *IO* - NO-TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	65.55	82	75.35	78.77	7.43
PROD	56	84	76	80	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

Table 5.1.: Dependent variables' statistics for task 1 of *Exp1* - *IO*.

Here, *QLTY* and *PROD* paint the same picture, with both having a similar median and distribution of values among the two groups, with the group employing TDD reporting slightly higher average and maximum values. As for the number of written tests, every participant in *G1* (TDD), except for an outlier, wrote a similar number of tests (*i.e.,* 9 or 10), with a standard deviation of only 1, compared to *G2*, where the number of written test cases fluctuates a lot more; still, however, the median for *G2* is 0, since a few participants did not write any tests at all. Interestingly, the maximum value for *TEST* in the NO-TDD group is higher than the maximum value for TDD. The other code quality metrics, *CYC*, *COG*, and *LOC* are on average higher for the TDD implementations, but still not significant (keep in mind that each function has a minimum cyclomatic complexity of 1, and the values here are considering the entire program). A higher value of *LOC* is generally expected for the TDD implementation since this metric aggregates the lines of code in both the production code and the test files; however, even by considering the individual values for *LOC* in the production and test files (not reported here), there is still not a significant difference in lines of code with respect to the implemented user stories, for both groups. We will however keep displaying the value of *LOC* as an aggregated measurement, in order to highlight how the number of test cases impacts it.



(a) QLTY (b) PROD (c) TEST



(d) CYC (e) COG (f) LOC

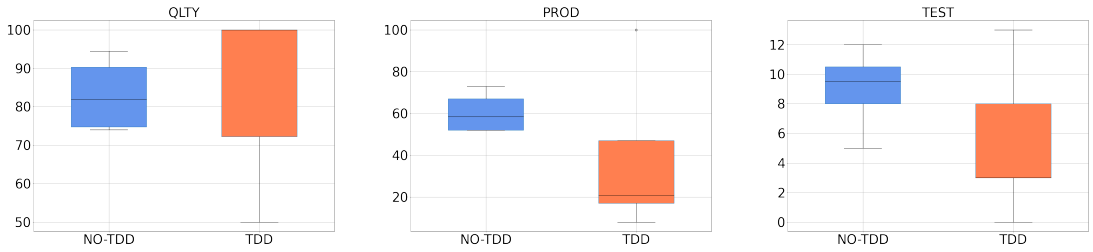
Figure 5.3.: Box plots for task 1 of *Exp1 - IO*.

In task 2 (which, as emerged from the post-questionnaires, many participants found harder compared to the previous one), the extracted statistics highlight the opposite situation (see table 5.2 and figure 5.4): while the values for $QLTY$ are still higher for the TDD group (*G2* in this case), the same thing is not true for $PROD$; this suggests that among the participants that tackled this task using TDD, there are some that only tackled the first user stories, and did so flawlessly, thus resulting in a high value for $QLTY$ (100% median, and 84% mean), and a much lower value for $PROD$ (21% median and 38.6% mean). For the same reason, the average number of tests written is lower (5.4 average for *G2* compared to the 9 average for *G1*); the maximum value, however, is still higher for *G2*, possibly resulting from a participant more extensively applying TDD to all user stories of the task. Finally, this time the values of CYC and COG are higher for the NO-TDD group; this is just related to the latter implementing overall more stories. LOC displays a similar behavior.

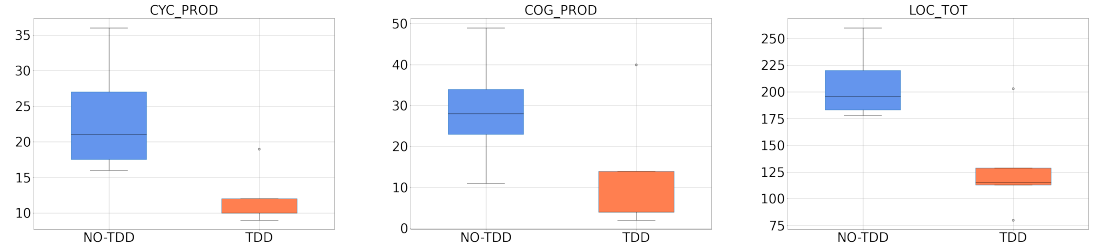
Exp1 - *CR* - TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	50	100	84.44	100	22.70
PROD	8	100	38.6	21	37.35
TEST	0	13	5.4	3	5.12
CYC	9	19	12	10	4.06
COG	2	40	12.8	4.0	15.91
LOC	80	203	128	115	45.61
Exp1 - *CR* - NO-TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	74	94.43	83.07	81.94	10.17
PROD	52	73	60.5	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

Table 5.2.: Dependent variables' statistics for task 2 of *Exp1* - *CR*.

In the task developed for *Exp2*, on the other hand, (see table 5.3 and figure 5.6), we see something much more in line with the first experimental task, with results in favor of TDD. First, *QLTY* and *PROD* have a similar distribution of values for both approaches, suggesting that all stories that have been tackled are "balanced" (*i.e.,* there is no hard user story like there was in task 2); the TDD group produced results that are around 10% better on average. For the number of tests, we see again a result in line with the first task, with the TDD group producing on average many more tests, and with the minimum value for this group still well above the median of 3.5 for the NO-TDD group. Regarding the three last variables associated with code complexity and quality, once again these are higher for the TDD approach, even though, in the case of cyclomatic and cognitive complexity, the result are still not concerning from a maintainability point of view.



(a) QLTY (b) PROD (c) TEST

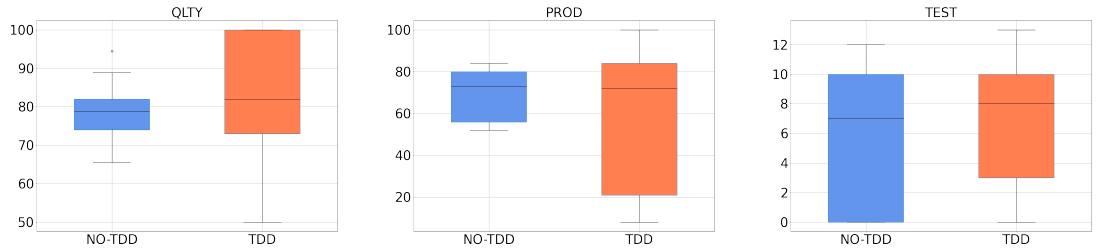


(d) CYC (e) COG (f) LOC

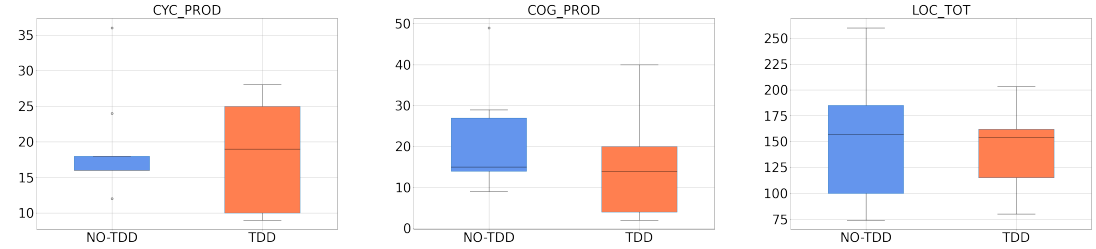
Figure 5.4.: Box plots for task 2 of *Exp1* - *CR*.

Exp2 - 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
Exp2 - 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

Table 5.3.: Dependent variables' statistics for *Exp2*.

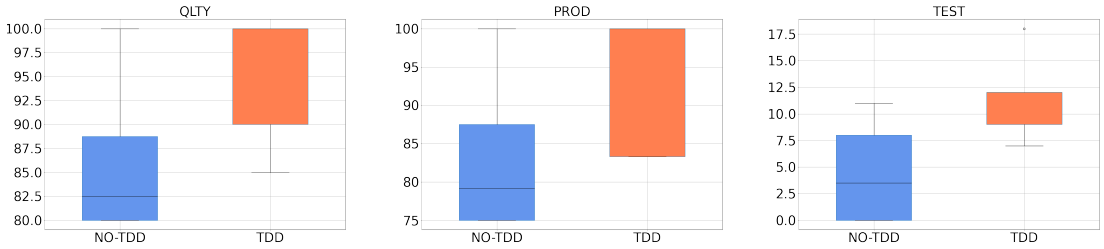


(a) QLTY (b) PROD (c) TEST

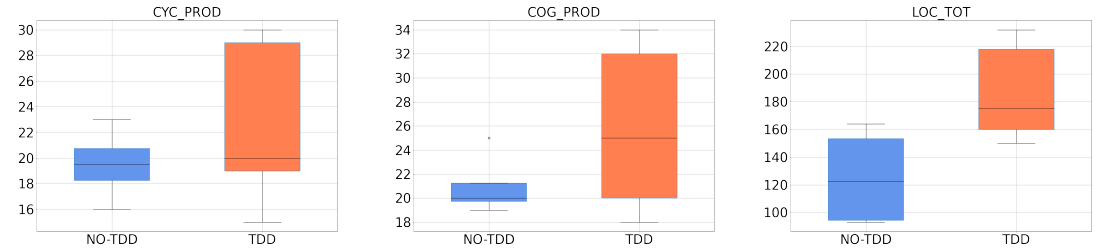


(d) CYC (e) COG (f) LOC

Figure 5.5.: Aggregated box plot charts for *Exp1*.



(a) QLTY (b) PROD (c) TEST



(d) CYC (e) COG (f) LOC

Figure 5.6.: Box plot charts for *Exp2*.

Let's now consider both *Exp1* and *Exp2* simultaneously. As for $QLTY$, the comparison between TDD and NO-TDD seems in favor of the former for any experimental task; indeed, by looking at table 5.4 and figures 5.5.a - which aggregate the results for both tasks in *Exp1* - and 5.6.a, we can notice how the boxes for TDD in the charts are higher than, or comparable to the ones for NO-TDD. The same trend is confirmed by the mean and median values (see table 5.4). However, we can notice that the effect of TDD on $QLTY$ is not uniform across the experimental tasks: the gap between TDD and NO-TDD is wider for the experimental task in *Exp2*, while it is more limited for the two experimental tasks in *Exp1*.

As for $PROD$, it seems that the effect of TDD is contrasting across the experimental tasks; in particular, when considering the first task in *Exp1* and the one in *Exp2*, the boxes for TDD are higher than the boxes for NO-TDD (figures 5.3.b and 5.6.b). On the contrary, the box for TDD is lower than the one for NO-TDD in the second experimental task of *Exp1* (figures 5.4.b and 5.6.b). Such a contrasting trend is confirmed when looking at the mean and median values (see table 5.4).

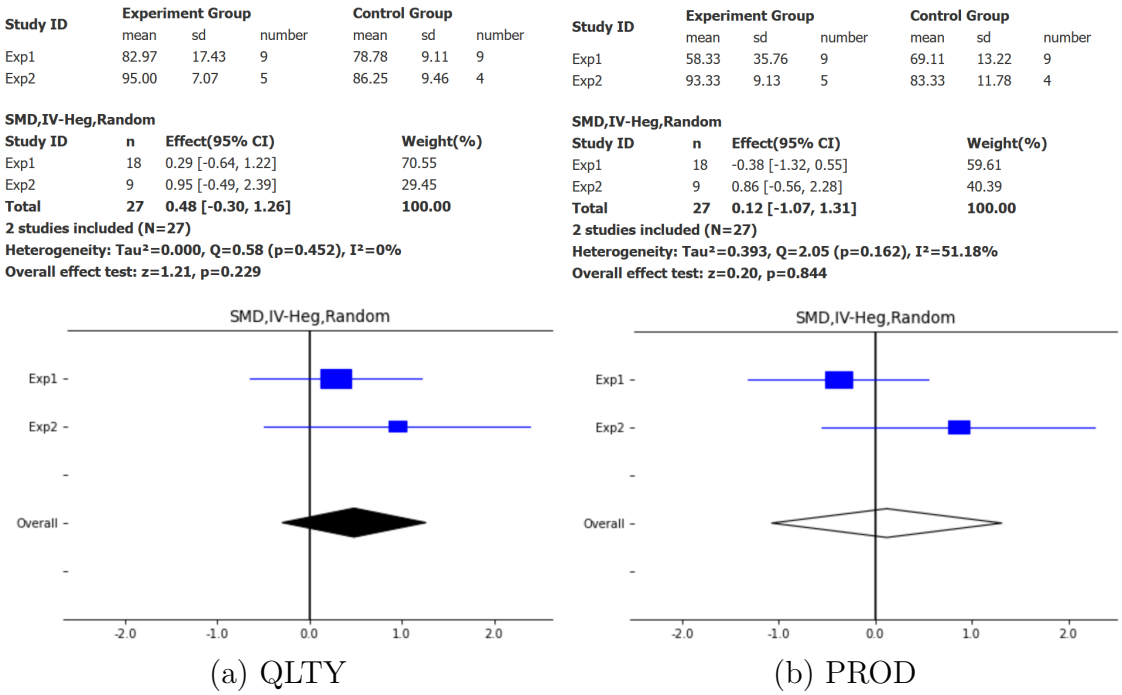
Finally, if we consider the number of test cases written by the participants (*i.e.,* the $TEST$ dependent variable), we can see how - similarly to the $QLTY$ variable (see figures 5.5.c and 5.6.c) - the boxes for TDD are either higher than or comparable to the ones for NO-TDD. Moreover, the median values for both experiments are still higher for TDD, while the majority of values in the boxes are close together, especially in *Exp2*.

Exp1 - TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	50	100	82.97	82	17.43
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
Exp1 - 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
Exp2 - 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
Exp2 - 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

Table 5.4.: Dependent variables' statistics for *Exp1* and *Exp2*.

5.8.2. Meta-analysis

Figure 5.7 displays the results of the aggregate analysis on the variables for *Exp1* and *Exp2* through means of forest plot charts, focusing on the *QLTY* and *PROD* variables. The SMDs relative to the *QLTY* variable for the single experiments (see figure 5.7.a) are both in favor of TDD; in particular, the SMD is considered small (0.290) in *Exp1* and large (0.950) in *Exp2*, highlighting how the effect of TDD on *QLTY* is not uniform and may depend on the tackled task; the overall SMD is still in favor of TDD and small (0.480).



(a) QLTY (b) PROD

Figure 5.7.: Forest plot charts for the *QLTY* and *PROD* variables.

Figure 5.7.b displays the forest plot for the *PROD* variable. Here, the SMD for *Exp1* is in favor of NO-TDD and small (-0.380); this is due to the second experimental task, as expected from looking at the box plots charts in figure 5.4. For *Exp2*, on the other hand, the SMD is in favor of TDD and large (0.860). The joint SMD is still in favor of TDD but is negligible (0.120); the hollow joint SMD indicates a higher heterogeneity (I^2 is more than 50%).

Finally, figure 5.8 plots the results for the meta-analysis relative to the *TEST* variable. Here, similarly to what happens with the *QLTY* variable, both experiments

are in favor of TDD: the SMD for *Exp1* is small (0.230), which is again expected given the contrast between the first two experimental tasks, while the SMD for *Exp2* is large (1.330). The joint SMD is still in favor of TDD and medium (0.600).

Study ID	Experiment Group			Control Group		
	mean	sd	number	mean	sd	number
Exp1	7.22	4.26	9	6.11	5.08	9
Exp2	11.60	4.15	5	4.50	5.44	4

SMD,IV-Heg,Random

Study ID	n	Effect(95% CI)	Weight(%)
Exp1	18	0.23 [-0.70, 1.15]	66.43
Exp2	9	1.33 [-0.22, 2.88]	33.57
Total	**27**	**0.60 [-0.43, 1.62]**	**100.00**

2 studies included (N=27)
Heterogeneity: Tau²=0.185, Q=1.44 (p=0.236), I²=30.4%
Overall effect test: z=1.14, p=0.256

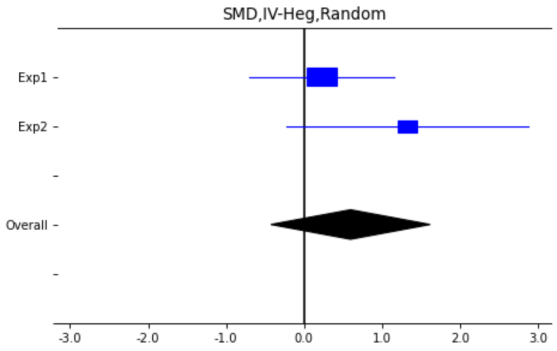


Figure 5.8.: Forest plot chart for the $TEST$ variable.

Given that for all three variables, the 95% CI interval contains the 0, and especially considering the small size of this study, the meta-analysis alone does not indicate a statistical significance in favor of any of the two approaches for any of these variables. A non-significant result does not mean that the treatment has no effect or that the difference is not meaningful; it simply means that there is not enough evidence to support a claim of a significant effect; therefore, we are in need of further replication to reach a statistical conclusion on the effects of TDD on external quality, productivity, and number of tests.

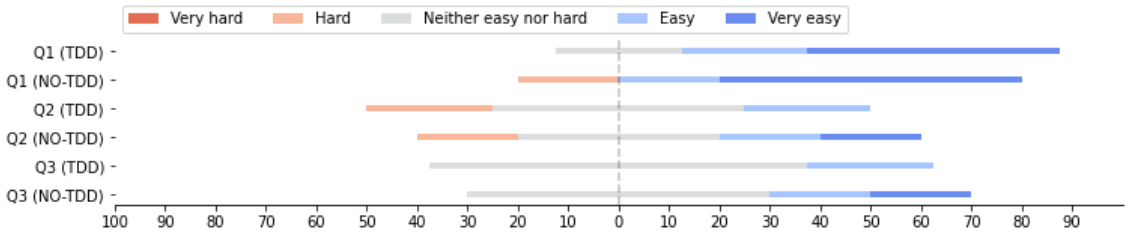
5.8.3. Post-questionnaire analysis

Figure 5.9 summarizes the answers provided by the participants in the post-experiment questionnaires submitted at the end of each task for *Exp1*, comparing the responses by the employed approach (TDD or NO-TDD). Regarding user story comprehensibility, in the first experimental task, *IO*, the participants had a similar level of agreement on the matter, with the majority finding the provided user stories easy or very easy. In the second task, *CR*, there is a more substantial difference, with only 40% of the TDD group having a positive perception of the user stories' clarity, compared to the 87% of the NO-TDD group.

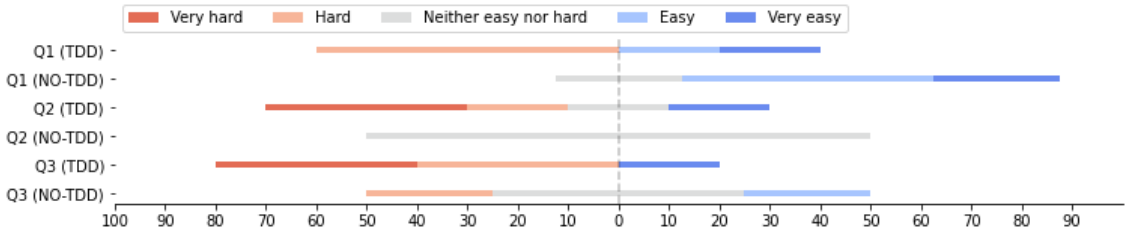
As for the second question, regarding the feelings of the participants on the general task difficulty, in the first experimental task the participants somewhat agree, with

50% of the TDD group and 40% of the NO-TDD group feeling neutral about the difficulty of the task; no one in the TDD group found the task very easy, however, compared to a 20% in the NO-TDD group. In the second task, the answers are quite in favor of NO-TDD, with 60% of the TDD group having some difficulties with the implementation of the provided user stories; 20% however found the task very easy. On the other hand, all of the participants in the NO-TDD group agreed on feeling neutral about their ease in developing the task.

Finally, question three was focused on the difficulty of the participants in applying their reference condition: for task 1, the TDD group did not have a strong opinion on applying this technique, with 75% feeling neutral about it, and the other 25% finding the application of TDD easy but not too easy; as for the NO-TDD group, this 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 them being overall more accustomed to the approach. This conveys the idea that TDD was considered less easy to apply compared to NO-TDD; such a trend is even clearer when considering the second experimental task, where 80% of the TDD group had trouble with their approach(40% answered "Very Hard" and 40% answered "Hard"), compared to only 25% for the participants that used NO-TDD.



(a) First task for *Exp1*, *IO*.



(b) Second task for *Exp1*, *CR*.

Figure 5.9.: Diverging stacked bar charts for the post-questionnaires.

Moving to the open-ended questions, we performed thematic analysis in order to identify a base template on which to parse the answers provided by the participants in the same post-questionnaires. The themes we identified are:

1. General feelings on the development task.
2. Feelings on TDD and NO-TDD to accomplish the task.
3. NO-TDD testing approach.
4. Personal comparison and thoughts between TDD and NO-TDD.

Most of the participants found the first development task easy, with 6 out of 9 expressing a positive feeling about it; 2 out of the remaining 3 found some challenges and would have liked a bit more time, while the last participant did not answer. For the second task, 2 out of 9 students had no issues with it, while 6 found the task to be harder than the previous one: out of the latter, 3 reported struggling with implementing some user stories, while the other 3 simply stated that the task was more challenging, without voicing in detail their struggles with it.

Employing TDD to tackle the tasks was still not fully clear at the end of *Exp1*: for task 1, 2 participants expressed how getting into the TDD mindset for a new problem was not so easy for them, while another student, participant 2, found this practice very useful and expressed their good feelings for having learned it:

> *"I think TDD is very helpful, and I am glad that I can learn it."*

In the second task, which as we have seen in the interval-scale questions was considered overall harder, 3 out of 5 participants in the TDD group expressed having trouble with it. For example, participants 5 stated:

> *"I think the CleaningRobot was harder than the IntelligentOffice, but maybe only because of TDD."*

While expressing a similar feeling, participant 8 also said how more experience with TDD would be beneficial to them:

> *"It is hard to think the other way around as a software developer, but I think in my opinion TDD is very useful and if you are used to it, it can really help you a lot to improve your programming."*

Out of the NO-TDD group, on the other hand, the majority had no particular issue with the approach in both tasks. However, in the post-questionnaire for *CR*

2 participants expressed how in their opinion, since this task was harder, maybe employing TDD would have made it easier, since it would have allowed them to tackle the user stories in a more incremental manner; for example, participant 3 said:

> *"With TDD I wouldn't have fallen into some pitfalls; I had to rewrite some code because I didn't know I broke some components with features that I added later. With TDD I would have known way earlier."*

As for the chosen NO-TDD testing approach, in the first task, 3 out of 5 participants dealing with this testing approach still wrote test cases after the production code, while the other 2 chose not to; for the second task, all 4 participants in the NO-TDD group tested their implementation, which is in line with them finding *CR* overall harder.

At the end of the second task, 4 participants expressed their preference with TDD, while 3 were still more comfortable with NO-TDD. Finally, one participant stated how in their opinion the best approach to use depends on how refined the requirements for the task to implement are, stating how they would prefer TDD, but only if the functional requirements are clear, while for discovering new technologies, they wouldn't use this approach:

> *"TDD reduces bugs, if you break something with a refactoring you know it instantly, and the tests are like documentation. With NO-TDD bugs and functional errors are only known at the end of the testing phase, which for me is terrible. I would now prefer TDD, but only if the functional requirements are clear. For discovering technology, I wouldn't use TDD for sure."*

5.8.4. Final interview analysis

Thematic analysis was also used to extract patterns from the final individual participant interviews; we identified the following main topics:

1. Thoughts on the replication experiment.
2. Refactoring with TDD.
3. NO-TDD testing approach.
4. Applying TDD for ESs development.
5. Prior participant testing experience.

6. Thoughts on the overall experience with the study, from the lectures and homework, to the last development task.

For the first point, 5 out of 9 participants found the task straight forward and had no issues with its development, while the other 4 found it balanced, with some user stories requiring more thought before being implemented, according to them. Furthermore, 3 participants stated how they were already familiar with some of the sensors and actuators used for the task, either from previous university courses, or from personal experience.

As a reminder, for *Exp2*, 5 out of 9 participants were (randomly) assigned the TDD version, while the other 4 had to develop the ES using NO-TDD; out of the 5 that used TDD, 4 of them performed some minor refactoring, either in the test cases or in the production code, while 1 did not. As for the NO-TDD group, half still wrote tests (although many less compared to the TDD group, as we have seen from the gathered data) after implementing the user stories, while the other half did not.

When discussing the employment of TDD and NO-TDD in the context of ESs and their strengths and weaknesses, some participants expressed how they were not comfortable providing a definitive answer on the matter, mostly given their still limited experience with TDD. Others conveyed how the best approach depends on the task they are working on, similarly to what happened in the post-questionnaire for *CR*; for example, participant 7 said:

> *"The task (SR) was pretty easy for me; as a result I think that TDD was not really needed [...], it was a bit overkill. On the second task (CR), however, which I found much more complex, I feel like TDD was much more beneficial."*

Participants also provided information about their testing experience prior to the study: 2 of them had no testing experience at all; on the other hand, 4 students already had some unit testing experience, while 3 had limited and mostly theoretical knowledge of TDD, obtained in a previous university course. Concerning the learning experience with TDD, while some participants expressed how at the end of the study they were still more comfortable using NO-TDD, one of the students among those with no prior testing, experience stated:

> *"I had no prior testing experience at all, so it was very nice to learn about it in the lectures, especially TDD; I would use TDD for sure going forward, especially with even more experience. The main challenge was getting into it in the beginning."*

Considering the last point, most students appreciated the overall experience and expressed how they enjoyed the mixture of theory and practice on TDD and the other concepts explored. For example, participant 4 (TDD) stated:

> *"I enjoyed the overall experience made up of a mix of theory and practical stuff."*

Something very interesting that came up during the thematic analysis of these interviews, especially from the lecturer's point of view, is that participants expressed a high interest into the hardware deployment step: 5 out of the 9 students that have been interviewed, in fact, demonstrated genuine interest in seeing their developed ES in action, making use of real sensors and actuators and operating on a real hardware platform. Participants 2 and 3, for example, stated:

> *"Nice to see it run and test by yourself with hardware and with the sensors."*

> *"I knew most of the sensors so developing this task was not hard for me, but still really cool to implement because I wanted to see it in action on real hardware."*

Keep in mind that there was no reference to this aspect in the script for the interview, so each of the participants that showed interest in the topic, did so out of their initiative while discussing their thoughts on this task; also, each student was alone during their interview, so it is unlikely that the answers were influenced by each other.

6. Discussion

In this last chapter before the conclusions we will provide an answer to the defined research questions and discuss the implications of this study; finally, we will analyze potential threats to its validity.

6.1. Answers to the research questions

After examining the data emerged from the dependent variables' evaluation, the meta-analysis, and the thematic analysis, we can provide an answer to the research questions defined in Chapter 5.

6.1.1. Research question 1

When we introduced our dependent variables, besides $QLTY$, we defined three additional variables to estimate the external quality of software: CYC, COG, and LOC; however, as we have seen in the previous chapter, the values for these variables did not display any significant difference between the two approaches. As a result, to answer our first research question, we only considered the $QLTY$ variable, which highlighted more interesting results.

For $QLTY$, the comparison between TDD and NO-TDD seems in favor of the former for any of the experimental tasks. However, the effects of TDD are not uniform across the tasks: by looking at *Exp1* the values for $QLTY$ represented by the width of the boxes in the plot charts are very similar across the two tasks (see figures 5.3.a and 5.4.a); this is further highlighted by the mean and median values. On the other hand, by looking at the box plot chart for $QLTY$ in *Exp2* (see figure 5.6), we can notice how the difference in width between the boxes is wider in this case. This suggests a moderating role of the experimental task. As for the meta-analysis (see figure 5.7.a), the joint SMD for $QLTY$ is still in favor of TDD and small (0.480). Finally, considering how there might be cases, such as *CR*, where a more complex ES, as expressed by the participants, still made the TDD group produce higher values

for *QLTY* with respect to the NO-TDD group, while struggling in other areas, such as productivity, further strengthens the idea that using TDD improves the external quality of the implemented ES.

6.1.2. Research question 2

Considering *PROD*, the effects of TDD are contrasting across the experimental tasks: while the situation is similar for the first and final tasks, where the results for the box plot charts relative to the TDD group are either higher than or comparable to the NO-TDD results (see figures 5.3.b and 5.6.b), for the second task in *Exp1* the situation is the opposite, suggesting a moderating role of the experimental task, and highlighting how a harder task impacts developers' productivity. The overall SMD for *PROD* in the meta-analysis is in favor of TDD (0.120), but it is negligible, suggesting an insignificant impact of TDD on developers' productivity.

Given the fact that participants expressed how *CR* was much harder compared to the first task (*i.e., IO*), and considering that *G2* applied TDD for the first time during this period, the obtained result is less surprising. By looking at the task for *Exp2*, on the other hand we can notice how the gap between TDD and NO-TDD (in favor of the former) grew compared to the first task. Keep in mind that by the point participants tackled *SH*, they had all applied TDD exactly once, in a previous task. Consequently, we can speculate how gaining more experience with TDD might balance the results, or even show an edge for this approach.

6.1.3. Research question 3

As for the number of written tests, reported by the *TEST* variable, by looking at the box plot charts for the first and final experimental tasks (*i.e., IO* and *SH*), we can see how the boxes for the TDD group are not only much higher, but much shorter than the ones for the NO-TDD group; this is due to the fact that all participants that applied TDD wrote a similar number of test cases compared to the NO-TDD group (for which some participants did not write any tests). This is further highlighted by the mean values, which are 9.5 for TDD and 3.8 for NO-TDD in *IO*, and 11.6 for TDD and 4.5 for NO-TDD in *SH*, respectively. On the other hand, in task2 of *Exp1*, *CR*, it is shown how a harder task, as voiced by the participants, impacted once again the TDD group, resulting in a lower number of written test cases on average, with a mean of 5.4 for TDD versus a mean of 9 for NO-TDD. However, the results for *CR* also produced a higher maximum value for the *TEST* variable; this

may have resulted from a participant being more familiar with TDD, who applied this approach more extensively. By aggregating both tasks in *Exp1* we can however see how on average TDD still produced a higher number of tests. Additionally, the meta-analysis for this variable produced an overall SMD in favor of TDD and medium (0.600). This suggests that employing TDD may result in a higher number of test cases written when developing ESs.

6.2. Implications

In this section, we outline implications for lecturers and researchers based on the main findings from our assessment.

6.2.1. Implications for lecturers

The data, which seems to indicate that overall external quality and number of test cases written improve when using TDD and that there is not a substantial difference between TDD and NO-TDD in terms of productivity, suggests properly teaching TDD in the context of ES development. Furthermore, as for the first task of *Exp1* and the one in *Exp2*, we also observed by means of the descriptive statistics that TDD improves the productivity of the developers for their produced solution.

These results also allow us to speculate that TDD and the used development pipeline [2], which suggests writing simple simulations of hardware components (*i.e.,* mock objects) to let tests pass and to continue writing first tests and then production code for features of a given ES, can be taught in academic ESs courses instead of a more traditional approach such as NO-TDD.

To some extent, this would also have practical implications from the professional developers' perspective: that is, if a newcomer to the working market is familiar with TDD (and the referenced development pipeline for ESs), the software industry could be encouraged to migrate their development from NO-TDD to TDD. In fact, software companies could be encouraged to use TDD because *(i)* there is initial evidence that it improves productivity and number of written test cases, and *(ii)* developers could be familiar with it before being hired, thus reducing the amount of training required to gain a higher level of proficiency with this technique, as well as its related costs.

Finally, participants in our study appreciated their overall experience in studying TDD, especially the deployment and testing of the developed solutions in a real

target environment of their ES. We can postulate that this positive experience has affected the participants' intentions to use TDD in the development of future ESs: this is practically relevant for lecturers that should plan courses on the design and the implementation of ESs that make a mix of theory and practical experiences for the students.

6.2.2. Implications for researchers

The post-experimental data we gathered highlights how both the execution of the implementation tasks and the application of TDD were perceived as more difficult by the participants. To some extent, this outcome is coherent with past studies (*e.g.,* [46, 32]), with a plausible explanation being that participants were experienced with unit testing in a test-last manner and therefore were less prone to learn a new approach which expected them to write test cases before production code.

This postulation suggests two possible future research directions: *(i)* replicating our experiments with the participation of more experienced developers to ascertain that the greater the experience with unit testing in a test-last manner, the more negative their perspective with TDD is and *(ii)* conducting a longitudinal study with a cohort of developers to investigate how developers perception and retainment of TDD applied to ES development changes over time, as well as determining how this changes affect the constructs we analyze (*i.e.,* external quality, developers' productivity, and number of test cases).

Quantitative and qualitative results suggest that the development pipeline proposed by Greening[2] is a valuable alternative to a traditional way of coding, where production code is written before tests; researchers could be interested in further studying this pipeline (*i.e.,* "*The Embedded TDD Cycle*") by exploiting field studies. Our preliminary results have the merit to justify such kinds of studies; moreover, we make our raw data publicly available online in order to allow, in the future, researchers to conduct secondary studies (*e.g.,* meta-analyses) on TDD applied to ESs development. We also made our replication package, including the raw data, available online to allow researchers to replicate our study and ease the execution of secondary studies (*e.g.,* meta-analyses) on TDD applied to ESs development [47].

6.3. Threats to validity

According to Campbell and Stanley [48], threats to validity are either internal or external: internal validity concerns "controlling" the aspects of the experiment's setting to ensure that the outcomes are caused only by the introduced techniques [49], while external validity refers to showing a real-world effect, but without knowing which factors actually caused the observed difference [49]. Cook and Campbell extended the list of validity categories to: conclusion, internal, construct, and external validity [50]. The latter classification has often been adopted in past empirical software engineering studies [38].

In order to determine the potential threats to validity that may affect our study, we referenced Wohlin *et al.*'s guidelines [38]. Completely avoiding/mitigating threats is often unfeasible, given the dependency between some of them: avoiding/mitigating a kind of threat (*i.e.,* in internal validity) might intensify or even introduce another kind of threat [38]. As a result, there are inherent trade-offs between validities: with internal and external validities for example, the more we control extraneous factors in our study, the less we can generalize our findings to a broader context. As for our baseline and replication experiments, in the next subsections we will consider the different kinds of threat individually.

6.3.1. Threats to internal validity

Internal validity refers to 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): *(i)*: the treatment and response variables change together; *(ii)*: the treatment precedes changes in the response variables, and *(iii)*: no confounding or extraneous factors can explain the results of the study.

The main threat in this category is perhaps related to the monitoring of the participants during the replication study; since they accomplished the implementation of the final task at home and alone, before deploying it on hardware under our supervision, we cannot be sure of the means employed by the participants to accomplish the task. Given the fact that the final score of the *Embedded Systems* course was not

influenced in any way by the outcome of the task, we can reasonably assume that the participants would have no reason to maliciously interfere with the task (*i.e.,* by cheating or sharing information). On the other hand, and entering the domain of the *maturation* and *motivation* threats, towards the end of the study, some participants might have grown bored with the procedure or might have received a less desirable treatment, and thus decided to not perform the same way they would have if they were more engaged. The threat of *motivation* holds in all three experimental tasks, since it is related to the nature of the adopted experimental design; as for the threat of *maturation* we see it holding especially in the replication study, since participants were not in a controlled environment.

The opposite, however, might also be true: given the fact that a volunteer population is generally more motivated and engaged compared to the average population [38], a *subject selection* threat might have affected the overall results of the experiments.

6.3.2. Threats to external validity

External validity represents 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?

The main external validity threat could be the one of *interaction of selection and treatment*: since only students were involved in the study, some with no prior testing experience or even without a strong Computer Science background, the results could potentially not be applicable to professional developers. However, the use of students has the advantage that they have more homogeneous backgrounds and skills and allow obtaining initial empirical evidence [34, 51]. As for the threat of *representation of the setting*, while one might be concerned that the implementation tasks and tools used in the experiments are not really representative of the technologies that make up the majority of ESs, they are quite popular implementation tools for these systems. However, we opted for ESs (*i.e., IO*, *CR*, and *SH*) that can be considered representative of the domain of interest for our study. Furthermore, to deal with external validity, we also asked the participants to base the ESs implementation on a recognized development pipeline [2]. Finally, we asked the participants in *Exp2* to deploy and test the ES (*i.e., SR*) in a real hardware environment.

6.3.3. Threats to construct validity

Construct validity refers to the extent to which a study's measures are related to the theoretical construct being studied.

In the end, we measured each construct with a single dependent variable (*e.g.,* external quality with $QLTY$). As so, in case of measurement bias, this might affect the obtained results with a threat of *mono-method bias*. 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. From a *theory definition* thread standpoint, the used dependent variables have been employed in a number of empirical studies of this kind in the past, thus we can safely assume that they are consolidated by this point. Finally, since no participant had a practical experience with TDD, a threat of *treatment testing* does not concern our study.

6.3.4. Threats to conclusion validity

Conclusion validity refers to the extent to which the conclusions drawn from a study are supported by the data.

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 in the end and can thus be ignored.

7. Conclusions and final remarks

In this thesis we presented an empirical assessment constituted of two experiments, a baseline (*Exp1*) and its replication (*Exp2*), with the objective of investigating how TDD impacts the development of ESs and how it compares to traditional test-last approaches, especially in terms of the external quality of the implemented ES, developers' productivity, and number of written test cases. Participants to this study accomplished three implementation tasks in *Python* and targeting the *Raspberry Pi model 4* hardware platform, while using either TDD or NO-TDD to test their developed ESs.

For the first and second tasks, participants made use of mock objects to model the interaction with the underlying hardware components (according to the pipeline proposed by Greening [2]), while developing and testing their implementation in a host development environment. As for the final tasks, around which the replication study was designed, participants used the same approach, before deploying and testing their solution on the real hardware.

The overall results of the experiments suggest that external quality of the developed ESs and number of written test cases increase when applying TDD, while there is not a substantial difference with respect to the developers'productivity. Furthermore, participants expressed how learning TDD was harder, especially those who were strongly used to a test-last approach; we can see this reflected in the implementation of the second experimental task, which saw on average better performances for NO-TDD. However, some members of the TDD group, which we speculate had a stronger reception of this approach during the training sessions, still performed better than the NO-TDD group.

Based on the gathered data, we delineated possible implications from the perspectives of lecturers and researchers. For example, our results suggest lecturers teach TDD along with a development pipeline where hardware components are mocked before actually deploying the ES in the environment for which it has been developed.

Finally, despite we gathered evidence that TDD can be successfully applied to the development of ESs, we foster replications of our experiments, especially by involving

professional developers and the software industry. Our investigation has the merit to justify such replications, as it is easier to recruit professional developers when initial evidence is available.

7.1. Personal contribution to the experimentation

This section concerns the personal contribution that I, as the author of this thesis, provided towards the planning, execution, and analysis of this study. Specifically, the steps I performed are:

- **Participation to the planning of the study**. I participated in a series of meeting with my supervisors during which the overall planning of the study was determined, including both experiments and the interviews.
- **Development of the experimental material**. I assembled the teaching material that was used for the initial lectures and training/homework sessions that took place before the beginning of the study. This material was made up of presentations, exercises, and project templates. Similarly, I designed the experimental tasks, by defining a document for each of them, detailing the goal of the task, development instructions, and the list of user stories to implement. Furthermore, for each task, I created a GitHub repository containing the project template that was later shared with the participants. Besides designing the experimental tasks, I also defined the respective acceptance test suite for each of them; these suites were not delivered to the students, as they were part of the tools used to later extract the data from the delivered implementations. Finally, I assembled the hardware necessary for the deployment of the final task; this included plugging the various sensors and actuators in a breadboard and wiring them to the main board (a *Raspberry Pi*), as well as making sure that they worked and they measured their respective values according to their specifications.
- **Execution of the study**. I held the lectures and interactive training sessions to provide the students with the knowledge necessary to tackle the experiments. Moreover, I oversaw the students during the execution of the first two experimental tasks, and provided assistance and clarification when necessary. For the final task, I individually met each participant and oversaw them during the deployment and testing phases of their solution on the real hardware platform. Once the hardware implementation was tested, I sat down with each participant and interviewed them according to a prepared script, in order to

gather their feedback on the whole study, from the lectures, to the last task they implemented.

- **Data extraction**. Once the participants to the study had submitted their developed systems and the final task was deployed, I implemented a program in *Python* to automatically scan their projects and run the acceptance test suites on their source code, with the goal of extracting the metrics on which the analysis was later performed; these metrics include the number of test cases written by the participants, the number of user stories tackled in each task, as well as the number of test that passed in each acceptance suite. Moreover, as for the code complexity metrics, these were extracted using the *SonarQube* tool. Based on these initial metrics, the values for each dependent variable were extracted from the submitted software artifacts: for example, the number of tests that passed in the acceptance suite were used to compute the values for the $PROD$ variable. These results were later organized in a spreadsheet, where I also reported the answers to the interval-scale questions for the post-questionnaires. As for the final individual interviews, originally recorded with a microphone, they were transcribed and organized in a text file.
- **Data analysis**. For the last step, I built a notebook using *Python* and the `pandas` data analysis library, in which I performed the analysis steps reported in this thesis. First, for each dependent variable, I computed their minimum and maximum values, their mean, their median, and their standard deviation, and organized them all in a set of tables. Secondly, I built the box plot charts for each dependent variable for each of the tasks, to get a visual representation of their distributions. In order to compare both experiments, using meta-analysis, I plotted the effect estimates and their corresponding confidence intervals for both experiments using forest plot charts. As a way to analyze the responses provided in the interval-scale questions in the post-questionnaires, I created the bar charts in which these results are summarized, for the first two tasks. Finally, I used thematic analysis to analyze the responses provided during the final interviews, and highlight any interesting cues provided by the students.

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, 2022.

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

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

[4] 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.

[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] *24th International Conference on Agile Software Development*. URL: `https://www.agilealliance.org/xp2023/`.

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

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

[9] 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.

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

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

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

[13] *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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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, pp. 13–22.

[25] Moritz Beller et al. "Developer Testing in the IDE: Patterns, Beliefs, and Behavior". In: *IEEE Trans. Software Eng.* 45.3 (2019), pp. 261–284.

[26] Neil C. Borle et al. "Analyzing the effects of test driven development in GitHub". In: *Empir. Softw. Eng.* 23.4 (2018), pp. 1931–1958.

[27] Roberto Latorre. "Effects of Developer Experience on Learning and Applying Unit Test-Driven Development". In: *IEEE Trans. Software Eng.* 40.4 (2014), pp. 381–395.

[28] Davide Fucci et al. "A longitudinal cohort study on the retainment of test-driven development". In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*. Ed. by Markku Oivo, Daniel Méndez Fernández, and Audris Mockus. ACM, 2018, 18:1–18:10.

[29] Vahid Garousi et al. "Testing embedded software: A survey of the literature". In: *Inf. Softw. Technol.* 104 (2018), pp. 14–45.

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

[31] 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.

[32] Simone Romano et al. "Do Static Analysis Tools Affect Software Quality when Using Test-driven Development?" In: *ESEM '22: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki Finland, September 19 - 23, 2022*. Ed. by Fernanda Madeiral et al. ACM, 2022, pp. 80–91.

[33] Rini Solingen et al. "Goal Question Metric (GQM) Approach". In: Jan. 2002. ISBN: 9780471028956.

[34] Jeffrey C. Carver et al. "Issues in Using Students in Empirical Studies in Software Engineering Education". In: *9th IEEE International Software Metrics Symposium (METRICS 2003), 3-5 September 2003, Sydney, Australia*. IEEE Computer Society, 2003, p. 239.

[35] *IEEE Spectrum*. URL: `https://spectrum.ieee.org/top-programming-languages-2022`.

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

[37] Sira Vegas, Cecilia Apa, and Natalia Juristo Juzgado. "Crossover Designs in Software Engineering Experiments: Benefits and Perils". In: *IEEE Trans. Software Eng.* 42.2 (2016), pp. 120–135.

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

[39] Nigel King. "Using templates in the thematic analysis of text". In: *Essential Guide to Qualitative Methods in Organizational Research*. Ed. by Catherine Cassell and Gillian Symon. Sage, 2004, pp. 256–270.

[40] 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.

[41] 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.

[42] *SonarQube*. URL: `https://www.sonarsource.com/products/sonarqube/`.

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

[44] Jacob Cohen. "A power primer." In: *Psychological bulletin* 112 (1 1992), pp. 155–9.

[45] Nigel King. "Using Templates in the Thematic Analysis of Text". In: Jan. 2004, pp. 257–270. ISBN: 9780761948889.

[46] Simone Romano et al. "Results from a Replicated Experiment on the Affective Reactions of Novice Developers When Applying Test-Driven Development". In: *Agile Processes in Software Engineering and Extreme Programming - 21st International Conference on Agile Software Development, XP 2020, Copenhagen, Denmark, June 8-12, 2020, Proceedings*. Ed. by Viktoria Stray et al. Vol. 383. Lecture Notes in Business Information Processing. Springer, 2020, pp. 223–239.

[47] Simone Romano. *Replication Package of "Test-Driven Development and Embedded Systems: an Exploratory Investigation"*. 2023. URL: `https://figshare.com/s/6322beb3a23373149862`.

[48] D.T. Campbell and J.C. Stanley. *Handbook of Research on Teaching*. 5th Edition. American Educational Research Association, 2016. ISBN: 9780935302509. (Visited on 01/27/2023).

[49] Janet Siegmund, Norbert Siegmund, and Sven Apel. "Views on Internal and External Validity in Empirical Software Engineering". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. Ed. by Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum. IEEE Computer Society, 2015, pp. 9–19.

[50] "Quasi-Experimentation - Design and Analysis Issues for Field Settings". In: *Psychologica Belgica* (1979).

[51] Martin Höst, Björn Regnell, and Claes Wohlin. "Using Students as Subjects-A Comparative Study of Students and Professionals in Lead-Time Impact Assessment". In: *Empir. Softw. Eng.* 5.3 (2000), pp. 201–214.

Appendices

A. Experimental task 1 - IntelligentOffice - TDD group

A.1. Constraints

Your project must be free of syntactic errors. Please make sure that your code actually runs in PyCharm.

A.2. Task goal

The goal of this task is to develop an intelligent office system, which allows the user to manage the light, blinds, 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 (IR) distance sensor used to detect the presence of a worker inside 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 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.
- RTC to handle time operations.
- Servo motor to open/close the blinds of the office window.
- Photoresistor to measure the light level inside the office.
- Smart light bulb.

- Carbon dioxide sensor, used to measure the CO2 levels inside the office.
- 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` file in the source code. Handle any error situation that you may encounter by throwing the `IntelligentOfficeError` exception. Figure A.1 recaps the layout of the sensors and actuators in the office.

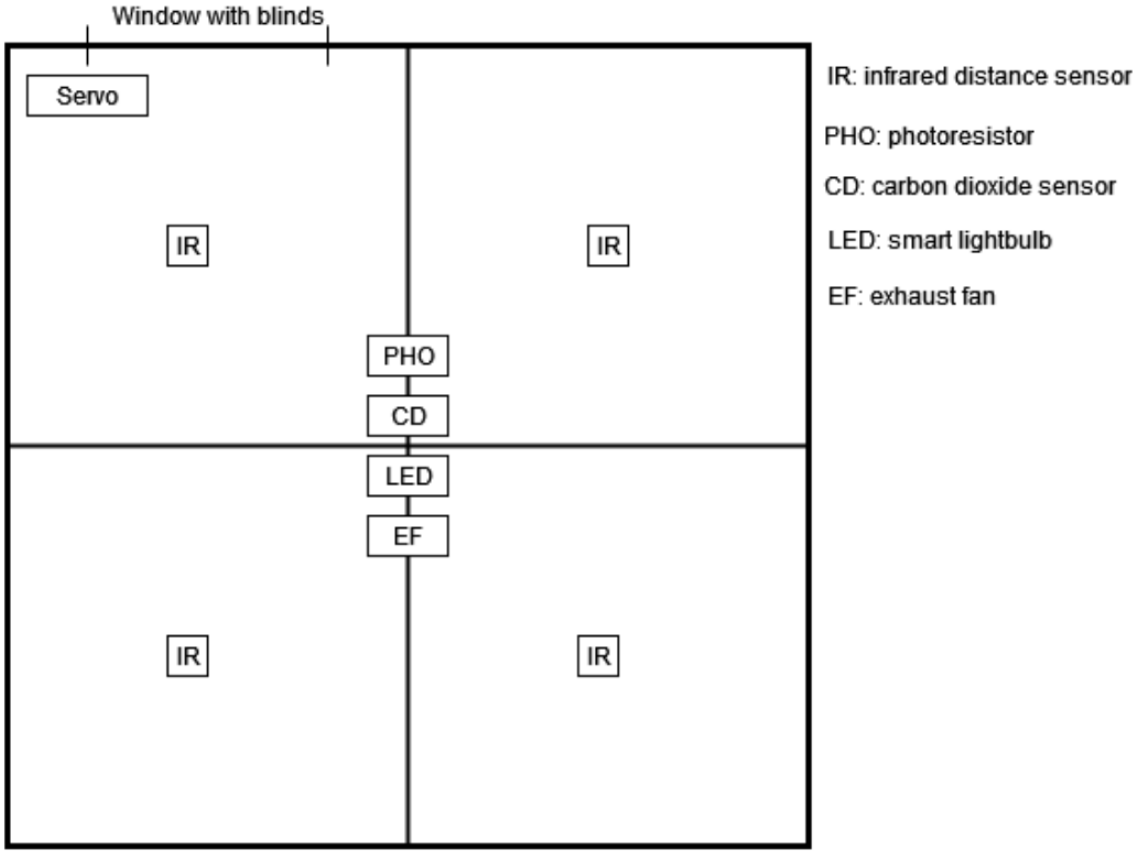


Figure A.1.: Office layout.

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

A.3. 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 public link to a zip file). The post-questionnaire is available at: `https://forms.gle/H4eNDDR6CjLjUofY6`.

A.4. User Stories

A.4.1. 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 it.

A.4.2. 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°. We can control it by sending a Pulse-Width Modulation (PWM) 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° rotation of the servo motor.
- **Fully open**, corresponding to a 180° 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.

A.4.3. 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.

A.4.4. 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:

- Update the implementation of `IntelligentOffice.manage_light_level()` → `None` to control the behavior of the smart light bulb.

A.4.5. 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.

B. Experimental task 2 - CleaningRobot - NO-TDD group

B.1. Constraints

Your project must be free of syntactic errors. Please make sure that your code actually runs in PyCharm.

B.2. Task goal

The goal of this task is to develop a cleaning robot; the robot moves in a room and cleans the dust on the floor along the way. To clean the dust, the robot is equipped with a cleaning system placed underneath it. When the robot is turned on, it turns on the cleaning system. The robot moves into the room, thanks to two DC motors, one that controls its wheels and another that controls its rotation, by executing a command, which a Route Management System (RMS) sends to the robot. While moving in the room, the robot can encounter obstacles: these can be detected thanks to an infrared (IR) distance sensor placed in the front. The robot can check the charge left in its internal battery: to do so, it is equipped with an Intelligent Battery Sensor (IBS); furthermore, a recharging LED is mounted on the top of the robot to signal that it needs to be recharged.

The room, where the robot moves, is represented as a rectangular grid with x and y coordinates; the origin cell of the grid - *i.e.,* $(0, 0)$ - is located in the bottom-left corner. A cell of the grid may or may not contain an obstacle. The RMS keeps track of the room layout, including the position of the last obstacle encountered in the room. To recap, the following sensors, actuators, and systems are present:

- DC motor to control the wheels in order to move the robot forward.

- DC motor to control the rotation of the body of the robot, in order to make it rotate left or right.
- RMS, sending commands to the robot.
- Infrared distance sensor used to detect obstacles.
- IBS to determine the battery charge left.
- Recharge LED.
- Cleaning system.

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` file in the source code. Handle any error situation that you may encounter by throwing the `CleaningRobotError` exception. For now, you don't need to know more; further details will be provided in the user stories below.

B.3. Instructions

Depending on your preference, either clone the repository at `https://github.com/Espher5/cleaning_robot` 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:

- `CleaningRobot`: you will implement your methods here.
- `CleaningRobotError`: exception that you will raise to handle errors.
- `CleaningRobotTest`: you can write your tests here.
- `mock.GPIO`: contains the mocked methods for GPIO 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 NO-TDD to implement this software system (*i.e.,* use the development approach you prefer but not TDD). 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. 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 public link to a zip file). The post-questionnaire is available at: `https://forms.gle/GEXHybbDJujZJz2V6`.

B.4. User Stories

B.4.1. Robot deployment

The robot has a status, namely a string (without white spaces) formatted as follows: (x, y, h). The pair (x, y) represents the position of the robot in the room (in terms of x and y coordinates) while h is the heading - *i.e.,* the direction the robot is pointing towards; the direction can be: $N(North)$, $S(South)$, $E(East)$, or $W(West)$. The robot assumes the $North$ is parallel to the y-axis. The robot starts its duty from the origin position - *i.e.,*, the position with coordinates $(0, 0)$, facing $North$ (see figure B.1 below).

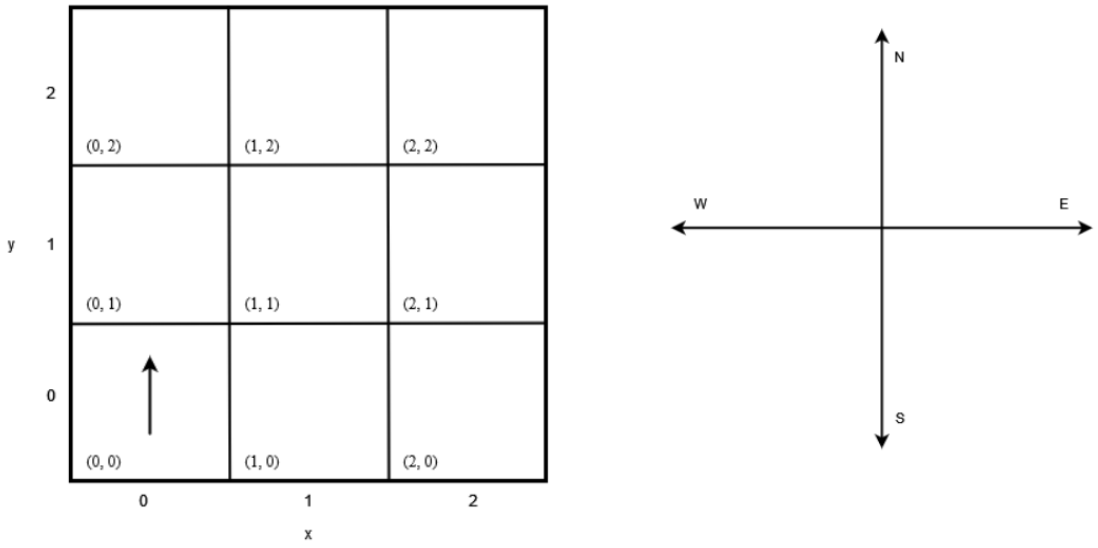


Figure B.1.: Room layout and robot compass.

Requirement:

- Implement `CleaningRobot.initialize_robot` → `None` to set the status of the robot to "$(0, 0, N)$".
- Implement `CleaningRobot.robot_status()` → `str` to retrieve the current status of the robot.

Example:

- The robot status "$(0, 0, N)$" indicates that the robot lies in the cell with x and y coordinates both equal to 0. The heading of the robot is N - *i.e.,* it is pointing $North$.

B.4.2. Robot startup

When the robot is turned on, it first checks how much battery is left by querying the IBS. If the capacity returned by the IBS is equal to or less than 10%, the robot turns on the recharging led. Otherwise, the robot turns on the cleaning system and sends its status to the RMS. In any case, the robot stands still.

The IBS is connected on pin 11 (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 a percentage indicating the amount of battery charge left. The recharge LED is connected on pin 12 (BOARD mode). The communication with the sensor happens via the GPIO `output` function. Finally, the cleaning system is connected on pin 13 (BOARD mode). Assume this component to be an independent system; the communication with it happens via the GPIO `output` function. The pin for the IBS, the one for the recharge LED and the one for the cleaning system have already been set up in the constructor of the `CleaningRobot` class. Since at this stage of development there is no way to determine the state of the physical LED and cleaning system, use the `self.battery_led_on` and `self.cleaning_system_on` instance variables to keep track of their states.

Requirement:

- Implement `CleaningRobot.manage_battery()` → `None` to control the behavior associated with the battery level.

B.4.3. Robot movement

To move into the room, the robot must receive a command from the RMS; when this happens, the robot controls the wheel motor and the rotation motor in order to execute the command and finally returns its new status to the RMS. The robot moves one cell forward when it receives the command "f". If the robot receives the command "r" or "l", it turns right or left respectively - *i.e.,* it rotates clockwise or counterclockwise 90° around itself. The robot cannot move backwards. The two motors that control the wheels of the robot and the rotation of the body are DC motors, and the pins to connect them to the main board have already been set up in the constructor of the `CleaningRobot` class. In order to control them, please use the following two methods in the `CleaningRobot` class:

- `activate_wheel_motor()` → `None` activates the wheel motor to make the robot move forward.
- `activate_rotation_motor(direction: str)` → `None` activates the rotation motor to make the robot turn left or right, based on the direction parameter ("l" to turn left or "r" to turn right)

Requirement:

- Implement `CleaningRobot.execute_command(command: str)` → `str` to execute the command corresponding to the string given by the RMS and return the status of the robot.

Example:

- If the robot status is "$(0, 0, N)$" and it receives the command "f", the robot moves one cell forward - *i.e.,* the robot controls the wheel motor in order to move one cell forward - and returns the new status "$(0, 1, N)$".
- If the status of the robot is "$(0, 0, N)$" and it receives the command "r", the robot rotates clockwise 90° - *i.e.,* the robot controls the rotation motor in order to rotate 90° - and returns the new status "$(0, 0, E)$". Similarly, if it receives the command string "l", the robot rotates counterclockwise 90° and returns the new status "$(0, 0, W)$".
- If the status of the robot is "$(1, 1, N)$" and it receives the command "f", the robot moves forward and returns the new status "$(1, 2, N)$". If the robot receives the command "l" afterwards, it turns left and returns the new status "$(1, 2, W)$".

Hint:

- Remember that you can use the `robot_status() → str` method in the `CleaningRobot` class to retrieve the current status of the robot.
- Use the class variables provided in the `CleaningRobot` class to update the rotation of the robot (N, E, S, W).

B.4.4. Obstacle detection

While executing a command, the robot can encounter an obstacle in a cell; the robot uses the infrared distance sensor to determine if there is an obstacle in front of it and thus avoid bumping into it. If an obstacle is detected, the robot cannot move beyond it; it will instead return its new status, including the positions of the encountered obstacle. In this case, the robot status is a string formatted as follows: "$(x, y, h)(x_o, y_o)$". The triple "(x, y, h)" represents the usual position and heading of the robot while the pair "(x_o, y_o)" represents the position of the encountered obstacle. In case multiple commands are executed in sequence, and multiple obstacles are found, the system only keeps track of the last one encountered.

The infrared sensor is connected to the main board on pin 15 (BOARD mode); communication with the sensor happens via the GPIO `input` function. The pin has already been set up in the constructor of the `CleaningRobot class`. Finally, 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.,* an obstacle).
- **Zero value**: nothing is detected in front of the sensor.

Requirement:

- Implement `CleaningRobot.obstacle_found() → bool` to determine whether an obstacle is in front of the robot (as detected by the infrared distance sensor).
- Update the implementation of `CleaningRobot.execute_command(command: str) → str` to include the obstacle processing steps.
- Update the implementation of `CleaningRobot.robot_status() → str` to retrieve the current status of the robot, including the possible encountered

obstacle.

Example:

- Let us suppose that there is an obstacle in the room at coordinates $(0, 1)$ (see figure B.2 below). The robot with initial status "$(0, 0, N)$", after executing the command string "f", returns the following status: "$(0, 0, N)(0, 1)$".

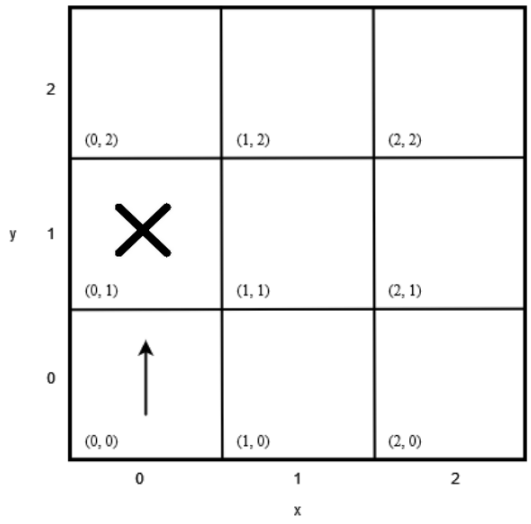


Figure B.2.: Room layout with an obstacle.

Hint:

- To keep track of the position of the obstacle after detecting it, use the `self.obstacle` instance variable in the `CleaningRobot` class.

B.4.5. Robot recharge

Before making any kind of movement/rotation, the robot checks how much battery is left by querying the IBS. When the capacity returned by the IBS is equal to or less than 10%, the robot shuts down the cleaning system, turns on the recharging led, and stands still (*i.e.,* it doesn't update its position in any way).

Requirement:

- Update the implementation of `CleaningRobot.execute_command(command: str)` $\rightarrow$ `str` to control the behavior associated with the battery level.
- Update the implementation of `CleaningRobot.manage_battery()` $\rightarrow$ `None`

to control the behavior associated with the battery level.

C. Experimental task 3 - SmartHome - TDD group

C.1. Constraints

Your project must be free of syntactic errors. Please make sure that your code actually runs in PyCharm.

C.2. Task goal

The goal of this task is to develop an intelligent system to manage a room in a house. First of all, an infrared (IR) distance sensor is placed on the ceiling of the room and is used to determine whether the user is inside the room or not. Based on the occupancy of the room and on the light measurements obtained by a photoresistor, the smart home system turns on/off a smart light bulb. Furthermore, the room has a window on one side, equipped with a servo motor to open/close it based on the delta between the temperatures measured by two temperature sensors, one indoor and one outdoor (*i.e.,* inside and outside the room). Finally, the smart home system also monitors the gas level in the air inside the room through an air quality sensor and then triggers an active buzzer when too many gas particles are detected. To recap, the following sensors and actuators are present:

- Infrared distance sensor located on the ceiling to the room.
- Smart light bulb.
- Photoresistor to measure the light level inside the room.
- Servo motor to open/close the window.
- Two temperature sensors, one indoor and one outdoor, used in combination with the servo motor to open/close the window.
- Air quality sensor, used to measure the gas level inside the room.

- Active buzzer that is triggered when a certain level of gas particles in the air is detected.

The communication between the main board and the other components happens with GPIO pins; GPIO communication is configured in BCM mode. For further details on how to use the GPIO library refer to the `mock.GPIO` file in the source code. Handle any error situation that you may encounter by throwing the `SmartHomeError` exception. For now, you don't need to know more; further details will be provided in the user stories below.

C.3. Instructions

Depending on your preference, either clone the repository at `https://github.com/Espher5/smart_home` 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:

- `SmartHome`: you will implement your methods here.
- `SmartHomeError`: exception that you will raise to handle errors.
- `SmartHomeTest`: you will write your tests here.
- `mock.GPIO`: contains the mocked methods for GPIO functionalities.
- `mock.adafruit_dht`: mocked library to control the temperature sensors.

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.

C.4. How to deliver your project

1. Reserve a slot for the final project discussion in the following calendar: `https://doodle.com/meeting/participate/id/aQn47p0b/vote`. Each student must reserve a single slot among those still available. In other words, please reserve a single slot among those still available, and make sure to not select a slot already reserved by someone else.
2. Deliver your project by sending an email containing a link to your project (either to a GitHub repository or to a sharing site). Keep in mind that the last possible date to deliver your project is January 8th at 23:59.
3. Please wait for a confirmation email after we receive your project.

C.5. User Stories

C.5.1. User detection

An infrared distance sensor, placed on the ceiling of the room is used to determine whether or not a user is inside the room. This sensor has a data pin connected to the board and used by the system in order to receive the measurements; more specifically, the data pin is connected to pin GPIO25 (BCM mode). The communication with the sensors happens via the GPIO `input` function. The pins have already been set up in the constructor of the `SmartHome` class.

The output of the infrared sensor is an analog signal which increases intensity according to the distance between the sensor and the object (*i.e.,* 0V when an object is very close to the sensor and >3V when the object is out of the max range of the sensor). Please note that this behavior of the sensor is the opposite compared to the one seen in the previous exercises. For this exercise, let's assume the input can be classified into these two categories:

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

Requirement:

- Implement `SmartHome.check_room_occupancy()` → `bool` to verify whether the room has someone inside it by using the infrared distance sensor.

C.5.2. Manage smart light bulb based on occupancy

When the user is inside the room, the smart home system turns on the smart light bulb; on the other hand, the system turns it off when the user leaves the room. The infrared distance sensor is used to determine whether someone is inside the room (see the previous user story).

The smart light bulb is represented by an LED, connected to the main board via pin GPIO26 (BCM mode)a, and Communication with it happens via the GPIO `output` function. The pin has already been set up in the constructor of the `SmartHome` class. Finally, since at the first 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 `SmartHome` class, to keep track of the state of the light bulb.

Requirement:

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

C.5.3. Manage smart light bulb based on light level

Before turning on the smart light bulb, the system checks how much light is inside the room (by querying the photoresistor). If the measured light level inside the room is above or equal to the threshold of 500 lux, the smart home system does not turn on the smart light bulb even if the user is in the room (or turns the light off if it was already on); on the other hand, if the light level is below the threshold of 500 lux and the user is in the room, the system turns on the smart light bulb as usual. The behavior when the user is not inside the room does not change (*i.e.,* the light stays off).

The light level is measured by the photoresistor connected on pin GPIO27 (BCM 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.

Requirement:

- Implement `SmartHome.measure_lux()` → `float` to query the photoresistor and measure the amount of lux in the room.

- Update the implementation of `SmartHome.manage_light_level() → None` to handle the additional constraint of the photoresistor measurements.

C.5.4. Open/close window based on temperature

Two temperature sensors, one indoor and one outdoor are used to manage the window: whenever the indoor temperature is lower than the outdoor temperature minus two degrees, the system opens the window by using the servo motor; on the other hand, when the indoor temperature is greater than the outdoor temperature plus two degrees, the system closes the window. The above behavior is only triggered when both of the sensors measure temperature in the range of $18°C$ to $30°C$; otherwise, the window stays closed.

The two temperature sensors are *DHT11* sensors, and are connected to pins GPIO23 and GPIO24 respectively (BCM mode); they can be controlled with the `mock.adafruit_dht` library. Both of the sensors have been initialized in the constructor of the `SmartHome` class and can be accessed by using the `self.dht_indoor` and `self.dht_outdoor` objects. In order to retrieve the temperature use the `mock.adafruit_dht.DHT11.temperature property` (*i.e.,* by calling `self.dht_indoor.temperature`) on the appropriate sensor object. This property returns a float value.

To open/close the window, 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 $0°C$ to $180°C$. We can control it by sending a Pulse-Width Modulation (PWM) 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 motor is an *SG90* model, is connected on pin GPIO6 (BCM mode), and operates at 50Hz frequency. Furthermore, let's assume the window can be in the following states:

- **Fully closed**, corresponding to a 0° rotation of the servo motor.
- **Fully open**, corresponding to a 180° 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 calling the `ChangeDutyCycle(duty_cycle: float) → None` method on the `self.servo` object and passing it the appropriate duty cycle. Finally, since at the first stage of

development there is no way to determine the state of the physical servo motor, use the `self.window_open` instance variable to keep track of its state.

Requirement:

- Implement `SmartHome.manage_window()` → `None` to control the behavior of the window based on the temperature values. In particular, please write your code inside the try block provided inside the method.

Hint:

- Mocking the `mock.adafruit_dht.DHT11.temperature` property is a bit more unusual compared to mocking a method. In particular you have to add the argument `new_callable=PropertyMock` to the patch decorator before your test method when you mock a property, like so:

  ```
  @patch('mock.adafruit_dht.DHT11.temperature', new_callable=PropertyMock)
  ```

- For the return values you can use `return_value` and `side_effect` (for multiple return values) on the mock object as usual.

C.5.5. Gas leak detection

An air quality sensor is used to check for any gas leaks inside the room; the sensor is configured in a way such that if the amount of gas particles detected is below 500 PPM, the sensor returns a constant reading of 1. As soon as the gas measurement goes to 500 PPM or above, the sensor switches state and starts returning readings of 0. To notify the user of any gas leak an active buzzer is employed; if the amount of detected gas is greater than or equal to 500 PPM, the system turns on the buzzer; otherwise, when the gas level goes below the threshold of 500 PPM, the buzzer is turned off. The air quality sensor is connected on pin GPIO5 (BCM mode). The communication with the sensor happens via the GPIO `input` function. The active buzzer is connected on pin GPIO6 (BCM mode). The communication with the buzzer happens via the GPIO `output` function. Both the pin for the air quality sensor and the one for the buzzer have already been set up in the constructor of the `SmartHome` class. Finally, since at the first stage of development there is no way to determine the state of the physical active buzzer, use the boolean instance variable `self.buzzer_on`, defined in the constructor of the `SmartHome` class, to keep track of the state of the buzzer.

Requirement:

- Implement `SmartHome.monitor_air_quality()` → `None` to control the behavior of the air quality sensor and the buzzer.

D. Thematic Analysis - Baseline study

D.1. Template

1. **Feelings on the development task**:
   - Task 1:
     - Easy / went smoothly. (**6**)
     - Some difficulties / needed more time. (**2**)
   - Task 2:
     - No problem with the task. (**2**)
     - Harder task, found some difficulties. (**3**)
     - Harder task, but no problem. (**3**)
2. **Feelings on TDD to accomplish the task**:
   - Task 1:
     - Good experience. (**1**)
     - Still not sure. (**2**)
   - Task 2:
     - Good experience. (**1**)
     - Still not sure. (**1**)
     - Having troubles. (**3**)
3. **Feelings on NO-TDD to accomplish the task**:
   - Task 1:
     - Good experience. (**2**)

    - Concerns on not testing enough some components. (**1**)
    - **NO-TDD approach**
        * Wrote tests after the production code. (**3**)
        * No tests written. (**2**)
  - Task 2:
    - Good experience. (**2**)
    - Not using TDD was harder. (**2**)
    - **NO-TDD approach**
        * Wrote tests after the production code. (**4**)
4. **Comparison between TDD and NO-TDD**:
   - TDD preference. (**4**)
   - NO-TDD preference. (**3**)
   - Depends on the task. (**1**)

D.2. Answers to the questionnaires

D.2.1. Task 1 - Intelligent Office

- **Student_01 (TDD)**:
  - **Q1.** NO ANSWER.
  - **Q2.** NO ANSWER.
- **Student_02 (TDD)**:
  - **Q1.** I think TDD is very helpful, and I am glad that I can learn it.
  - **Q2.** I just needed more time and more exercise.
- **Student_03 (TDD)**:
  - **Q1.** I honestly think TDD is not as error-prone as regular development. It increases software quality, and it assures that tests exist. But if I'm honest, TDD is sometimes hard, because you have to think different compared to the usual methods.
  - **Q2.** Task 4 - I really had to think deeply into this, because implementing

this story broke the tests for the third story.

- **Student_04 (TDD)**:
  - **Q1.** I still have to fully understand how it works.
  - **Q2.** I found the task very clear for understanding the various steps to be done.
- **Student_05 (NO-TDD)**:
  - **Q1.** I think that for me it's easier to program a NO-TDD software, because I never did TDD before. So, this is my "normal" way to code, because we didn't need to write a test file it was quicker and less work.
  - **Q2.** I liked that there were similarities with the previous tasks.
  - **Q3 (NO-TDD testing approach).** I just programmed one task after the other, like we learned in the previous lessons.
- **Student_06 (NO-TDD)**:
  - **Q1.** For me it is simpler because I can focus on the code.
  - **Q2.** User stories were clear and not difficult to implement.
  - **Q3.** I still have some difficulties during the test phase, maybe is simpler doing it with TDD approach.
- **Student_07 (NO-TDD)**:
  - **Q1.** NO-TDD may be faster when you are doing easy tasks but following this kind of approach may incur in some difficult debugging session, since there are possibilities to implement some code that may be never tested or not tested well.
  - **Q2.** This development task has been useful to understand the difference with using TDD and its advantages in respect to use NO-TDD
  - **Q3.** Long story short, I've implemented all the function following the user story and only after I've finished all of them I started writing tests.
- **Student_08 (NO-TDD)**:
  - **Q1.** I'm not used to Python, so the syntax was a bit complicated for me, but NO-TDD is the classical way of coding and testing for me, so I felt familiar with it.
  - **Q2.** Easy to understand, very well-structured, and I knew exactly what to do.

  - **Q3.** The knowledge from the previous lectures helped me to accomplish this task.

- **Student_09 (NO-TDD)**:
  - **Q1.** This is a nice activity, but maybe we have to discuss it more, because it is not really clear and there are some unknown in my mind.
  - **Q2.** These activities help and improves us.
  - **Q3.** NO ANSWER.

D.2.2. Task 2 - Cleaning Robot

- **Student_01 (NO-TDD)**:
  - **Q1.** It was ok.
  - **Q2.** It was not so hard because based on the exercises we have done before.
  - **Q3 (TDD and NO-TDD comparison).** TDD is the best one to avoid any errors in the future.
  - **Q4 (NO-TDD testing approach).** I have coded first and then wrote the test.
- **Student_02 (NO-TDD)**:
  - **Q1.** It was hard not to use TDD.
  - **Q2.** It was okay.
  - **Q3 (TDD and NO-TDD comparison).** I prefer using TDD.
  - **Q4 (NO-TDD testing approach).** No certain approach.
- **Student_03 (NO-TDD)**:
  - **Q1.** With TDD I wouldn't have fallen into some pitfalls; I had to rewrite some code because I didn't know I broke some components with future modifications. With TDD I would have known way earlier.
  - **Q2.** It was a brainful task, but still excellent. Wouldn't it be funny to run the code on an actual Roomba?
  - **Q3 (TDD and NO-TDD comparison).** TDD: it reduces bugs, if you break something with a refactoring you know it instantly, and the

tests are like documentation. NO-TDD: bugs and functional errors are only known at the end of the testing phase, which for me is terrible. I would now prefer TDD, but only if the functional requirements are clear; for discovering technology, I wouldn't use TDD for sure. But I guess it makes sense that you can't write good tests before the implementation for something like a quick throw away prototype.

  - **Q4 (NO-TDD testing approach).** Write all the production code, and then I tested it.

- **Student_04 (NO-TDD)**:
  - **Q1.** I feel quite confident.
  - **Q2.** I feel quite confident.
  - **Q3 (TDD and NO-TDD comparison).** With the NO-TDD approach, I seem to be able to better test the functions I implement.
  - **Q4 (NO-TDD testing approach).** I implemented all the functions first and then wrote the various tests.
- **Student_05 (TDD)**:
  - **Q1.** During the programming lessons of TDD, I could only listen to the explanation or try to write down the code from the projector, since for me it was impossible to do both at the same time; I tried to copy the code, but I don't know how to program well the TDD style. So today it was really hard for me.
  - **Q2.** I think the CleaningRobot was harder than the IntelligentOffice, but maybe only because of TDD. Also, when you write TDD, you have to write two codes, the actual code and then the whole test code, so it takes more time.
  - **Q3 (TDD and NO-TDD comparison).** I think that I have made a few mistakes in the NO-TDD code, but still I got everything, and the mistakes should be easy to fix. So, I prefer NO-TDD, but maybe also because I'm used to it.
- **Student_06 (TDD)**:
  - **Q1.** For me it's more difficult to use this methodology; I prefer to create the code before testing it
  - **Q2.** I had some difficulties to understand the task, it was more complex.

  - **Q3 (TDD and NO-TDD comparison).** NO-TDD was simpler to apply, also the tasks were easier to implement. Anyway, I prefer the NO-TDD approach.

- **Student_07 (TDD)**:
  - **Q1.** At first TDD may seem trickier, but in a few exercises it became more and more easy to apply.
  - **Q2.** The exercise seems a little harder than the last one ,and it required me more time.
  - **Q3 (TDD and NO-TDD comparison).** Applying TDD forces me to think in a way I'm not used to: the idea to implement only the code necessary to pass the code it's something I'm not into yet, but I still prefer TDD in respect to NO-TDD.
- **Student_08 (TDD)**:
  - **Q1.** It is hard to think the other way around as a software developer, but I think in my opinion TDD is very useful and if you are used to it, it really helps you a lot to improve your programming.
  - **Q2.** NO ANSWER.
  - **Q3 (TDD and NO-TDD comparison).** As I said, the thinking is upside down, and it's kind of like learning a new logical thinking. I still prefer the NO-TDD because I like it more to write the logic first.
- **Student_09 (TDD)**:
  - **Q1.** I think I didn't get it, but I tried so hard.
  - **Q2.** If I am still alive I am probably developing my skills.
  - **Q3 (TDD and NO-TDD comparison).** NO ANSWER,

E. Thematic Analysis - Replication study

E.1. Template

1. **Replication experiment**:
    - **General difficulty of the task**:
        - Easy. (**5**)
        - Balanced / some difficulties. (**4**)
    - Prior knowledge of the sensors / actuators. (**3**)
    - Positive thoughts on hardware deployment. (**5**)
2. **Refactorings in TDD**:
    - Some refactoring. (**4**)
    - No refactoring. (**1**)
3. **Test cases in NO-TDD**:
    - No tests. (**2**)
    - Tested anyway. (**2**)
4. **Application of TDD for ESs**:
    - Would use TDD going forward for ESs. (**5**)
    - Would use NO-TDD going forward for ESs. (**2**)
    - Not sure/depends on the complexity. (**2**)
5. **Testing experience**:
    - No prior testing experience. (**2**)
    - Low testing experience (only unit testing). (**4**)

- Some familiarity with TDD. (**3**)

6. **Seminars and exercises**:
   - Easy to follow / overall good. (**6**)
   - **Other**
     - Too fast for some things / some difficulties. (**2**)
     - Would have liked more explanation on TDD. (**1**)

E.2. Answers to the questionnaires

- **Student_01 (TDD):**
  - **Q1.** For me the development of this task was not very hard after the previous tasks; I also feel like at this point I am familiar with some libraries.
  - **Q2.** Still wrote tests afterwards, however, in my opinion there are downsides since you're not sure that you're testing everything. Writing tests before would be better but also harder.
  - **Q3.** Lectures and seminars were clear, but some more explanation would have been better on TDD. In my opinion TDD is better for ESs because some requirements may be harder to understand all at once, but I feel more comfortable with NO-TDD going forward, just because I am more used to it, regardless of ES or not. If I was more experienced with TDD I would for sure use it in the future. Also, a great experience for my future internship; I will have an advantage when going back home.
- **Student_02 (TDD):**
  - **Q1.** Good feeling because it was in line the first one. Nice to see it run and test by yourself with hardware and with the sensors.
  - **Q2.** TDD was very useful because it helps you understand what you are programming, and you know that it will work for sure since you have written the tests before. Only little refactoring performed for the temperature and motor story (4^{th} one) because I wanted to clean it a bit after the tests.
  - **Q3.** No prior testing experience at all, so it was very nice to learn about this in the lectures, especially TDD; would use TDD for sure going forward,

especially with more experience. Main challenge was getting into it in the beginning. It is suitable for ESs development in my opinion.

- **Student_03 (TDD):**
  - **Q1.** I knew most of the sensors so developing this task was not hard for me, but still really cool to implement because I wanted to see it in action on real hardware.
  - **Q2.** Helpful to use TDD cause the test acted as documentation. You don't need to have a "master plan" for testing the whole system, and you can rather focus on each phase at a time. Performed refactoring after implementing one story (3) broke something in a previous user story (2). Also, other minor refactorings to improve overall code readability.
  - **Q3.** Overall good experience with the lectures and seminars. It was good to see some testing approaches in practice besides theory. The approach that I'll use going forward depends for me: for quick prototyping or if we do not know all the requirements (like Agile) I would use TDD for sure. As I said you don't need a master plan, and you can do a little bit at a time. But if the requirements are already clear I wouldn't use TDD because I feel like it wouldn't really make much sense. Also, I work with ESs but not much with the actual hardware, so I can't really say how to really improve TDD for ESs cause for the software part I think it's fine with the tools that are available.
- **Student_04 (TDD):**
  - **Q1.** I liked the task overall, but it was simple for me because I already used the sensors (*Lab of IoT* course).
  - **Q2.** I had no issues using NO-TDD for this task, because it is the approach I used the most and I also wrote test cases; I feel like I wrote the same number of tests I would have written by using TDD.
  - **Q3.** Enjoyed the experience overall with a mixture of theory and practical stuff. I knew some TDD concepts from another course (*Software Dependability*). As for ESs going forward and using TDD or NO-TDD, I would say it depends on the complexity of the task. For ESs, using TDD in a much more complex task would be more helpful to determine the difference with NO-TDD.
- **Student_05 (TDD):**
  - **Q1.** Not too easy, not too hard. Interesting to see it like this (deployed

on hardware). Liked how the user stories are "incremental" and you keep adding stuff on top.

- **Q2.** Did not write any test cases with NO-TDD. I didn't really feel like I needed to do it, so I preferred to just modify the source code until everything worked. Also don't have much testing experience overall.
- **Q3.** Lectures were a bit fast for me. Would improve my overall experience with testing before saying for sure which to use in the future. With TDD of course you have less errors, so I feel like for ESs it could be useful if you know more about what you're doing, but generally I would use NO-TDD going forward.

- **Student_06 (TDD):**
  - **Q1.** Had some trouble with the implementation of the 4^{th} user story with the servo motor, but the task was manageable overall and clear to understand.
  - **Q2.** I feel like for this task using TDD was helpful; I performed some refactoring, mostly for the test cases, very little or maybe nothing for the production code, I don't remember.
  - **Q3.** No testing experience at all and very little experience with Python. It was hard to get into the mechanics even for the exercises during the lectures. In the beginning I preferred NO-TDD because it was more natural to test the code after; however, after applying TDD in the last tasks I liked using it. I still need a lot more practice with, but I feel like TDD would be useful for ESs going forward.
- **Student_07 (TDD):**
  - **Q1.** I already knew these sensors from other courses (*Lab Of IoT*) and knew their interface and how they were used, so the task was pretty easy for me.
  - **Q2.** As a result, TDD for me was not really needed for this task, it was a bit overkill. Minor refactoring. On the second task however, which I found much more complex, I feel like TDD was much more beneficial.
  - **Q3.** I already knew unit testing, and TDD was introduced in another course (*Software Dependability*), even if mostly from a theory point of view; it has been pretty natural for me to apply it in these tasks. I am not experienced with TDD for ESs specifically, so I can't really say if there is a big impact with it, but probably it can be very helpful with more

complex ESs.

- **Student_08 (TDD):**
  - **Q1.** I felt quite comfortable about developing this task. It was nice to deploy it on hardware, afterwards. Overall, a pleasant experience.
  - **Q2.** Using TDD helped me with this task especially in user stories 2 and 3. For the others there was no big difference in my opinion. Did not perform any refactoring as they were not necessary in my opinion.
  - **Q3.** Comfortable about the lectures and seminars. Learned why TDD could be helpful. For sure testing ESs is important regardless of the approach; however, TDD gives you immediate feedback which I feel is important for ES development.
- **Student_09 (TDD):**
  - **Q1.** At first was a big tough, but I really studied and gave it all; it was fun to see the project deployed and tested on real hardware.
  - **Q2.** I did not write any test cases. I really tried to do it but couldn't finish it, so I just decided to focus on the implementation. In the future I would love to write more test cases.
  - **Q3.** Didn't have much experience with testing before the lectures and seminars. I think TDD is more fun, and it could be useful for ES development. When I applied TDD I didn't find any particular difficulty, however it was a bit more challenging than NO-TDD, but I think that if I learned more how to use TDD, it would be better.

F. Paper

Test-Driven Development and Embedded Systems: An Exploratory Investigation

Michelangelo Esposito, Simone Romano, and Giuseppe Scanniello

University of Salerno, Fisciano, Italy,
{m.esposito253@studenti.unisa.it} and {siromano, gscanniello}@unisa.it

Abstract. We present the results of an exploratory investigation to obtain preliminary evidence on the use of *Test-Driven Development (TDD)*, an incremental approach to software development where tests are written before production code, to develop *Embedded Systems* (*ESs*). Specifically, we conducted two experiments in which we compared TDD with a non-TDD approach in terms of the external quality of ESs and developers' productivity. In the experiments, we also gathered qualitative data to better understand the investigated phenomenon. We found that the external quality of the implemented solutions increases when using TDD as compared to a non-TDD approach, while there is not a substantial difference with respect to developers' productivity. However, TDD is perceived as more difficult to apply than a non-TDD approach, and the development task is deemed more challenging with TDD.

Keywords: Embedded Systems, Experiment, Replication, TDD

1 Introduction

An *Embedded System* (*ES*) is a combination of hardware and software created for a particular purpose. In many cases, ESs operate as part of a larger system (*e.g.,* agricultural-sector equipment, automobiles, medical equipment, airplanes, *etc.*). The global market of ESs is expected to witness notable growth. A recent report evaluated the global market of ESs $89.1 billion in 2021, and this market is projected to reach $163.2 billion by 2031; with a compound annual growth rate of 6.5% [1]. This growth is related to an increase in the number of ESs-related research and advanced development projects (*e.g.,* driver-assistance systems) [1].

To date, there has yet to be shown which approach is to be preferred when developing ESs. In his book, Greening [12] asserted that developers can benefit from the application of *Test-Driven Development* (*TDD*), an incremental approach to software development in which the developer repeats a short cycle, to incrementally implement functionality, made up of three phases: *Red*, *Green*, and *Refactor* [5]. In the Red phase, the developer writes a unit test for a small chunk of functionality not yet implemented and watches the test fail. In the Green phase, the developer implements that chunk of functionality as quickly as possible and watches all unit tests pass. In the Refactor phase, the developer changes the internal structure of the code while paying attention to keeping the

functionality intact (*i.e.,* all unit tests have to pass). The Red-Green-Refactor cycle is thus repeated until the functionality is completely implemented. When this happens, the developer can tackle new functionality.

TDD has been conceived to develop "traditional" software and it is claimed to improve software quality as well as developers' productivity [14]. ESs have all challenges of non-Embedded Systems (NoESs), such as poor quality, but add challenges of their own [12]. For example, one of the most cited differences between ESs and NoESs is that the code of ESs depends on the hardware.

A huge amount of empirical investigations has been conducted to study the claimed effects of TDD when developing NoESs [14]. On the other hand, no investigation has been conducted yet to assess the benefits that TDD could bring to the development of ESs although some authors, like Greening [12], claim that the application of this approach produces benefits in the context of ESs development (*e.g.,* high-quality ESs).

To increase the body of knowledge on the application of TDD to ESs development, we investigated the following high-level Research Question (RQ):

> *Does the use of TDD affect the external quality of ESs, as well as developers' productivity?*

To answer this RQ, we conducted an exploratory study made up of two experiments (*i.e.,* a baseline experiment and a replication) on TDD applied to the development of ESs. The participants in the study were final-year Master's students in Computer Science (CS) taking an ESs course at the University of Salerno (Italy). To assess the benefits (if any) due to the application of TDD in terms of external (*i.e.,* functional) quality of developed solutions and developers' productivity, we compared TDD with a non-TDD approach named *YW* (*YourWay*)—*i.e.,* the approach developers would normally use to produce software when no restriction is imposed to them (except for not using TDD, of course) [2, 3, 11]. In both experiments, whichever the used approach (TDD or YW) was to develop a given ES, the participants were asked to use *mocks* to model and confirm the interactions between the device driver and the hardware. The mocks thus intercepted the commands to and from the device by simulating specific usage scenarios [12]. In the replication, we also asked the participants to replace the mocks with the actual interactions with sensors and actuators by deploying and testing their ES in the target hardware. In both experiments, we also gathered qualitative data to better understand the phenomenon under investigation.

Paper Structure. In Section 2, we outline related work. The design of our investigation is described in Section 3. The results are presented and discussed in Section 4 and Section 5, respectively. We highlight the threats to the validity of our investigation in Section 6. Conclusion and final remarks ends the paper.

2 Related Work

Despite different authors (*e.g.,* [9, 12, 15, 27]) have promoted the use of TDD in the context of ESs development, nobody has ever investigated the benefits that

TDD could bring to the development of ESs. Conversely, in the context of NoESs development, the claimed benefits of TDD—including increased external quality and developers' productivity, which we investigated in our study—have been the subject of different primary studies (*e.g.,* [10, 23]), whose results have been also gathered and combined in secondary studies (*e.g.,* [6, 19, 20, 24]). For example, in their Systematic Literature Review (SLR), Turhan *et al.* [24] included 32 primary studies (2000-2009) and found a moderate effect in favor of TDD on external quality while the results about developers' productivity are inconclusive (*i.e.,* the results do not lead to a firm conclusion in favor or against TDD). Bissi *et al.* [6] conducted an SLR that included 27 primary studies (1999-2014). The authors observed, similar to Turhan *et al.* [24], an improvement of external quality due to TDD, while the results about productivity are inconclusive. Rafique and Misic [20] conducted a meta-analysis of 25 controlled experiments (2000-2011) and observed a small effect in favor of TDD on external quality, while again the results about productivity are inconclusive. Finally, Munir *et al.* [19], in their SLR, classified 41 primary studies (2000-2011) into four categories based on rigor and relevance. They found that in each category different conclusions could be drawn for both external quality and productivity.

Our investigation, although it is preliminary, has the merit to study for the first time the application of TDD in a new development context, the one of ESs, in which different authors have promoted its application (*e.g.,* [9, 12, 15, 27]). Given the preliminary nature of our study, the obtained results are not intended to be conclusive; rather, we aim to gather initial evidence on the application of TDD when developing ESs and pave the way for future research on this matter.

3 Baseline Experiment and Replication

In the following of this section, we detail the design of our exploratory study, which comprises a baseline experiment (**Exp1**) and a replication (**Exp2**) with the same group of participants. To design our study, we took into account the guidelines for experimentation in Software Engineering by Wohlin *et al.* [26].

3.1 Research Questions

Consistent with our high-level RQ, the goal of our study (formulated according to the *GQM* template [4]) is the following:

> **Analyze** the use of TDD **for the purpose of** evaluating its effects in the development of ESs **with respect to** the external quality of the developed solutions and developers' productivity **from the point of view of** researchers and lecturers **in the context of** an ESs course involving final-year Master's students in CS who develop ESs in Python.

Accordingly, we defined and investigated the following (low-level) RQs:

RQ1. Does the use of TDD improve the external quality of ESs whose software is written in Python?

RQ2. Does the use of TDD increase developers' productivity when coding in Python?

We focused our study on Python because it (including its variants like MicroPython) in IEEE Spectrum[1] has ranked as the first programming languages for ESs, Web and Enterprise development.

3.2 Participants

The participants in our study (*i.e.,* Exp1 and Exp2) were final-year Master's students in CS at the University of Salerno. They were sampled by convenience among the students taking an ES course. Nine students, out of ten, accepted to participate in the study on a voluntary basis. In line with Carver *et al.*'s advice [7], we rewarded the students for their participation with two bonus points on the final mark of the course. The students were aware that: *(i)* they would receive the bonus points independently of their performance in the study; *(ii)* they could drop out of the study at any time, without being negatively judged; *(iii)* they could achieve the highest mark of the course even without participating in the study; and *(iv)* all the gathered data would be confidentially treated and anonymously shared for research purposes only. Our study had both research and educational goals: on one hand, we conceived the study to answer our RQs; on the other hand, the study allowed the students to gain experience with TDD applied to ESs development. As for the educational goal, TDD has attracted great attention from developers [11, 22] and it is claimed to bring benefits to the development of ESs [12].

The participants had all programming experience, and most of them rated such an experience 3 on a 5-point Likert scale (where 1 means "very inexperienced" and 5 means "very experienced"). The participants were mostly not experienced with unit testing since most of them rated 1 or 2 their unit-testing experience (again on a 5-point Likert scale). Three students had heard about TDD but nobody had a practical experience with it.

3.3 Experimental Tasks

The participants had to fulfill three experimental tasks: two in Exp1 and one in Exp2. Each experimental task consisted of developing an ES for *Raspberry Pi* by coding in Python and using *unittest* and *Mock.GPIO* as testing and mocking frameworks, respectively. We opted for Raspberry Pi because it supports Python and is a well-known general-purpose microprocessor. In Exp1, the experimental tasks were: **Intelligent Office** (**IO**) and **Cleaning Robot** (**CB**). As for Exp2, the experimental task was **Smart Room** (**SR**). A brief description of the experimental tasks is reported in Table 1 (further details are available in our online replication package [21].

[1] https://spectrum.ieee.org/top-programming-languages-2022

Table 1: Experimental task description.

Name	Description
IO	Developing an intelligent office system (referred to as IO), which allows handling light and CO2 levels inside an office. To handle the light, IO relies on the information gathered from different sensors—*i.e.,* Infra-Red (IR) distance sensors, a real-time clock, and a photoresistor—and then controls a servo motor (to open/close the blinds) and a light bulb. A carbon dioxide sensor is used to monitor the CO2 levels inside the office and then control the switch of an exhaust fan.
CB	Developing a cleaning robot (referred to as CB) that, while it moves in a room based on the commands it receives from an external system, cleans the dust on the floor along the way (by using rotating brushes). The robot also detects potential obstacles through an IR distance sensor. An intelligent battery sensor is used to check the charge left of the internal battery of the robot, and a LED is turned on to signal the need for a recharge.
SR	Developing a smart room system (referred to as SR), which handles light, temperature, and gas levels inside a room. To handle the light, the system relies on the information gathered from an IR distance sensor and photoresistor, and then turned on/off a light bulb. The information from a pair of temperature sensors is used to control the servo motor of a window. SR is also equipped with an air quality sensor to measure the gas levels inside the room and, if necessary, it triggers an alarm through a buzzer.

For each task, the experimental material provided to the participants included: *(i)* its description, which consisted of an abstract of the ES to be developed, a list of instructions, and a set of user's stories describing the ES's functionality to be implemented; and *(ii)* a project template (for *PyCharm* the used IDE), which contained function stubs (*i.e.,* empty functions exposing the expected API signatures), utility functions, and an example unittest test class.

Independently from the used development approach (TDD or YW) and experiment (Exp1 or Exp2), the participants when implementing an ES had to use mocks to model and confirm the interactions between the device driver and the hardware—this is similar to what Grenning suggests in his book [12]. In other words, the mocks allowed the unit tests written by the participants to simulate specific scenarios without depending on the actual sensors and actuators. We deliberately introduced a difference in Exp2 to take into account further steps of the development pipeline for ESs. In particular, we asked the participants in Exp2 (whatever the used development approach was) to apply two additional steps in the development pipeline: *(i)* replace the mocks with the actual interactions with sensors and actuators and then deploy their code in a Raspberry Pi; and *(ii)* test the ES on the field.

3.4 Independent and Dependent Variables

Regardless of the experiment, the participants were asked to fulfill each experimental task by using either TDD or YW as a development approach. Therefore, the independent variable of both Exp1 and Exp2 is **Approach**. It is a nominal variable assuming two possible values: *TDD* and *YW*. The choice of using YW as a baseline for comparison is based on past studies on TDD in the context of NoESs development (*e.g.,* [2, 3, 11]).

In both Exp1 and Exp2, the dependent variables are **QLTY** and **PROD**. The definitions of these variables are borrowed from past studies on TDD, but applied to NoESs development (*e.g.,* [2, 10, 23]). QLTY measures the external quality and, similar to the past studies mentioned above, it is defined as follows:

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

where #TUS is the number of user stories a participant tackled, while $QLTY_i$ is the external quality of the i-th user story tacked. A user story is tackled if at least one assert in the acceptance test suite of that user story passes. The acceptance test suites (one per user story) were pre-defined by the authors of this paper and not delivered to the participants. As for $QLTY_i$, it is computed as the number of asserts passed for the i-th tackled user story out of the total number of asserts for that user story:

$$QLTY_i = \frac{\#ASSERT_i(PASS)}{\#ASSERT_i(ALL)} \quad (2)$$

QLTY assumes values between 0 and 100, where a value close to 100 indicates high external quality of the developed solutions.

As for the PROD variable, it measures the productivity of a participant when developing an ES. Similar to past studies [2, 10, 23], PROD is defined as follows:

$$PROD = \frac{\#ASSERT(PASS)}{\#ASSERT(ALL)} * 100 \quad (3)$$

where #ASSERT(PASS) is the number of asserts passed in the acceptance test suites, while #ASSERT(ALL) is the total number of asserts in the acceptance test suites. PROD assumes values between 0 and 100, where a value close to 100 means high productivity of developers.

3.5 Experimental Designs

The experimental design of Exp1 is *ABBA crossover* [25]. It is a kind of *within-participants* design where each participant receives both treatments (*i.e., A* and *B*). In ABBA crossover designs, there are two *sequences* (*i.e., AB* and *BA*), defined as the order with which the treatments are administered to the participants, and two *periods* (*i.e., Period1* and *Period2*), defined as the times at which each treatment is administered. The experimental groups correspond to the sequences. Also, to mitigate learning effects, each period is paired with a different experimental task. In our case, the experimental groups are *TDD-YW* and *YW-TDD*. As shown in Table 2, the former first experimented with TDD to fulfill IO and then YW to fulfill CR, while the latter first applied YW and then TDD to fulfill IO and CR, respectively. The assignment of the participants to the experimental groups was done randomly.

As for Exp2, the experimental design is *one-factor-two-treatments* [26], a kind of *between-participants* design. In such a design, each participant receives

Table 2: Assignment of the participants.

Approach	Exp1		Exp2 (SR)
	Period1 (IO)	Period2 (CR)	
TDD	P1-P4	P5-P9	P2-P3, P6-P8
YW	P5-P9	P1-P4	P1, P4-P5, P9

only one treatment (in our case, either TDD or YW) while dealing with the same experimental task (in our case, SR). The experimental groups are paired with the treatments. The assignment of the participants to the groups was done randomly: five participants were assigned to TDD and four to YW (see Table 2). We would like to mention that the variation in the design of Exp2, as compared with the design of Exp1, was introduced because of the teaching schedule—*i.e.,* we exhausted the class hours reserved for the practical part of the ESs course.

3.6 Experimental Procedure

The experimental procedure of our study consisted of the following steps.

1. Recruitment. We gathered the adhesion of the students to participate in the experiments through an online questionnaire. It was also used to gather demographic information on the participants (*e.g.,* their programming experience).
2. Training. All the participants attended a lesson about unit testing in Python with unittest as a testing framework. The first part of the lesson was theoretical, while the second part was practical—in particular, under the lecturer's supervision, the students performed the unit testing of a NoES. Then, the participants attended a lesson about TDD, mocking (including the Mock.GPIO framework), and how to program Raspberry Pi. Again, the first part of the lesson was theoretical, while the second one was practical—*i.e.,* the students, with the guide of the lecturer, applied TDD to develop an ES for managing a greenhouse. Finally, as a laboratory activity, the students took part in a *warm-up task* where they all used TDD to develop an ES for managing a parking garage.
3. Exp1 Execution. We (randomly) split the participants into the TDD-YW and YW-TDD groups. As shown in Section 3.5, each participant applied alone each approach only once to fulfill IO and CR. The participants tackled the experimental tasks during two laboratory sessions, each lasting two hours, on two different days. At the beginning of each laboratory session, the participants received the experimental material (see Section 3.3), imported the project template into the PyCharm IDE, and then started implementing the user stories of the experimental task once at a time by using the requested approach. We asked the participants to use mocks. At the end of each laboratory session, the participants filled out an online post-questionnaire by which they delivered their project and shared feedback about both the experimental task and the used approach.
4. Exp2 Execution. We (randomly) split the participants into the TDD and YW groups. The participants tackled alone the experimental task at home and each participant applied either TDD or YW to fulfill SR (see Section 3.5). The experimental material was delivered via email; once received, the participants

could import the project template into the PyCharm IDE and then start implementing the user stories of the experimental task once at a time by using the requested approach. To fulfill the experimental task and similar to Exp1, the participants used mocks without interacting with the actual sensors/actuators. By the delivery date, the participants had to fill out an online post-questionnaire to deliver their PyCharm project and then book a slot for deploying and testing SR in the target hardware (pre-assembled in a research laboratory at the University of Salerno). The deployment of SR required each participant to replace the mocks with the actual interactions with sensors and actuators (see Section 3.3). The participants deployed and tested SR once at a time; in the end, they were (individually) interviewed by one of the authors of this paper. We conducted the interviews to gather feedback from the participants about the entire study. We asked the participants for their consent to audio-record the interviews.

3.7 Data Analysis

We used boxplots to depict the distributions of the values of each dependent variable across the experimental tasks of Exp1 and Exp2 (we also plotted the mean values in these boxplots). Later, we aggregated the data from both experiments by performing a meta-analysis—to perform it, we took into account the guideline by Kitchenham *et al.* [17]. For each dependent variable and experiment, we computed the Standardized Mean Difference (SMD) as *Hedges' (adjusted) g*.[2] To obtain joint SMDs (one for each dependent variable), we leveraged random-effects meta-analysis models. Finally, we show the results of the meta-analysis through forest plots.[3] We did not perform any statistical inference given the number of participants and exploratory nature of our study.

As for the post-questionnaires, we used diverging stacked bar plots to summarize the answers to the closed questions (reported as statements to be rated on 5-point Likert scales) by approach. To analyze the interview data instead, we first transcribed the interview audio recordings and then identified common themes in the interview transcripts by applying a thematic analysis. We took into account the guidelines by King [16] to identify such common themes. Briefly, we exploited the interview script (we used to guide the interviews) to develop some initial themes, and then we revised these themes as the analysis of the interview transcripts progressed until reaching a consensus on the final themes.

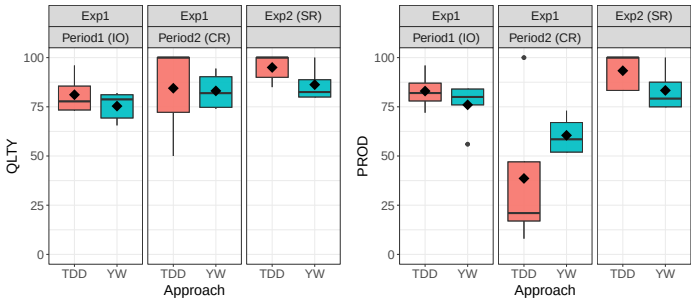
Analysis scripts, raw data, and more are available online [21].

4 Results

In this section, we first show the results regarding RQ1 and RQ2, and then those from the post-questionnaires and interviews, respectively.

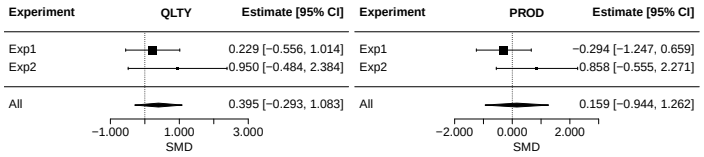
[2] We computed any SMD from Exp1 as suggested by Madeyski and Kitchenham [18]; this is to make the SMD from a crossover experiment (Exp1) comparable with that from a between-participants experiment (Exp2).

[3] The values of the descriptive statistics are available online [21], as well as the values of the I^2 statistic giving an indication of the between-experiment heterogeneity.



(a) QLTY (b) PROD

Fig. 1: Boxplots with respect to QLTY and PROD.



(a) QLTY (b) PROD

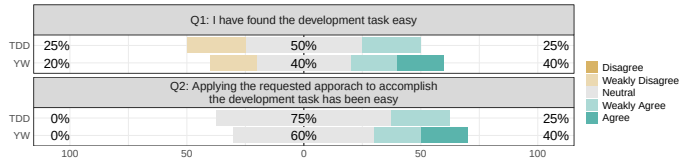
Fig. 2: Forest plots with respect to QLTY and PROD.

4.1 Results on RQ1 and RQ2

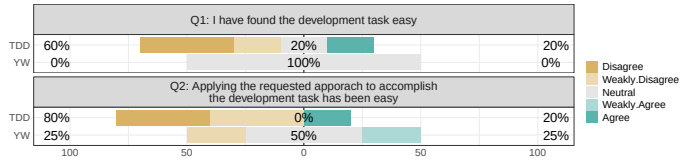
In Figure 1, we show the boxplots depicting the distributions of the values of QLTY and PROD for TDD and YW across the experimental tasks (*i.e.,* IO and CR in Exp1, and SR in Exp2). The forest plots depicting, for QLTY and PROD, the SMDs across the experiments and the joint SMD are shown in Figure 2.

QLTY. The comparison with respect to QLTY between TDD and YW seems in favor of the former for any experimental task. Indeed, by looking at Figure 1.a, we can notice that the boxes for TDD are either higher than or comparable to the ones for YW. This trend is confirmed by the mean and median values (represented in the boxplots as diamonds and thick horizontal lines, respectively). We can also notice that the effect of TDD on QLTY is not uniform across the experimental tasks: the difference between TDD and YW is wider for the experimental task in Exp2, while it is more limited for the two experimental tasks in Exp1. This suggests a moderating role of the experimental task. As for the SMDs, we can notice in Figure 2.a that, in the single experiments, the SMDs are both in favor of TDD. In particular, the SMD is small[4] (0.229) in Exp1 and large (0.95) in Exp2. The joint SMD is in favor of TDD and it is small (0.395).

[4] Based on Cohen's guidelines [8], the SMD can be interpreted as: *negligible*, if $|\text{SMD}| < 0.2$; *small*, if $0.2 \leq |\text{SMD}| < 0.5$; *medium*, if $0.5 \leq |\text{SMD}| < 0.8$; or *large*, otherwise.



(a) First experimental task



(b) Second experimental task

Fig. 3: Diverging stacked bar plots summarizing the answers to the closed questions of the post-questionnaires in Exp1 for first and second experimental tasks.

PROD. The effect of TDD on PROD is contrasting as shown in Figure 1.b. In particular, when considering the first experimental task in Exp1 and the one in Exp2, the boxes for TDD are either higher than the boxes for YW. On the contrary, the box for TDD is lower than the one for YW in the second experimental task of Exp1. Such a contrasting trend is confirmed when looking at the mean and median values, so suggesting a moderating role of the experimental task. As for the SMDs depicted in Figure 2.b, in the single experiments, the SMDs (with respect to PROD) are one in favor of YW and one in favor of TDD (Exp2). In particular, the SMD of Exp1 is in favor of YW and small (-0.294)—this is due to the second experimental task, as shown in Figure 2.b. On the contrary, the SMD of Exp2 is in favor of TDD and it is large (0.858). The joint SMD is still in favor of TDD, but negligible (0.159).

4.2 Results from Post-questionnaires

In Figure 3, we show the diverging stacked bar plots summarizing the answers to the closed questions of the post-questionnaires the participants filled out at the end of the two experimental tasks in Exp1. It seems that the participants who applied TDD found the experimental tasks less easy than those who applied YW. Indeed, the percentages of agreement in the first experimental task are equal to 25% for TDD and 40% for YW (see Figure 3.a). Such a difference seems stronger when considering the second experimental task (see Figure 3.b): the percentage of disagreement for TDD is equal to 60% while the one for YW is equal to 0%.

TDD seems to be considered less easy to be applied than YW in the first experimental task (see Figure 3.a): the percentages of the agreement for TDD and YW are equal to 25% and 40%, respectively. This pattern seems to be

much stronger in the second task (see Figure 3.b), where the percentage of disagreements for TDD and YW are equal to 80% and 25%, respectively.

4.3 Results from Interviews

In the following, we report some of the main themes resulting from the thematic analysis by highlighting them in bold. To bring credibility to our themes, we present them together with some excerpts from the interview transcripts.

1. Appreciations for Overall TDD Experience. The students appreciated the overall experience consisting of theory and practice on TDD and related topics. For example, P4 stated:

> *"I liked the overall experience made of a mix of theory and practical stuff."*

2. Appreciations for Deployment and Testing in Target Hardware. The students found it interesting and/or pleasant to see their ES running in the target hardware. For example, R2 said:

> *"Nice to see it [RS] run, and test by yourself, with hardware and sensors."*

3. Intention to Use TDD for ESs Development. The students considered TDD useful to develop ESs. Therefore, they declared they would take into account TDD to develop their ESs in the future; although some of them declared to certainly use TDD, while others on the basis of some factors (*e.g.,* the ES to be developed). An excerpt from P1's interview follows:

> *"In my opinion, TDD is better [than YW]...I would for sure use it in the future."*

4. More Practice with TDD. The students found it easier to write their tests after the production code; therefore, they stated to need more time to be comfortable with TDD (where the opposite happens). In this respect, P6 said:

> *"In the beginning, I preferred YW because it is more natural to test the code after; however, after applying TDD in the last tasks I liked using it. However, I still need more practice with TDD."*

5 Discussion

Below, We highlight the implications of our results for lecturers and researchers.

5.1 Implications for Lecturers

The results suggest that, when using TDD as compared to YW, the external quality of developed solutions increases, while there is not a substantial difference with respect to developers' productivity (despite in some tasks we observed better productivity with TDD). These initial results should encourage lectures to teach TDD in the context of ESs development. If this happened, companies that deal with the development of ESs would be encouraged to use TDD, rather than other approaches, for the following reasons: *(i)* there would be initial evidence that TDD brings benefits to the development of ESs; and *(ii)* if developers, before

being hired, were already familiar with TDD applied to ESs development, the training costs incurred by software companies would be reduced or even null.

The participants also appreciated the overall experience with TDD applied to the ESs development, especially the deployment and testing in the target hardware. We can postulate that these positive experiences affected participants' intention to use TDD to develop their future ESs. Therefore, we suggest lectures that plan ESs courses with a well-thought mix of theory and practice on TDD applied to ESs development, giving space to deployment and testing in the target hardware.

5.2 Implications for Researchers

Our results (*i.e.,* improved external quality and no difference with respect to productivity) are similar to those of past studies on TDD applied to NoESs development (*e.g.,* [6, 20, 24]). Nevertheless, we advise further studies on TDD in the context of ESs development because: *(i)* the challenges developers tackle to develop ESs are different from those they tackle to develop NoESs [12]; and *(ii)* our results, despite being promising and consistent with those of past studies on TDD in the context of NoESs development, are preliminary. Also, our results suggest that future studies on TDD applied to ESs development should take into account a possible moderating role of the ESs to be implemented.

As for the post-questionnaire results, it emerged that both the development tasks with TDD and application of TDD were perceived as more difficult as compared to YW. To some extent, this outcome is coherent with past studies (*e.g.,* [3]), and a plausible explanation—also supported by the interview results—is that the participants found it more natural to write their tests after the production code rather than doing the opposite as it happens in TDD. Researchers could be interested in conducting longitudinal studies to investigate how developers' perception of TDD applied to ESs development changes over time, and whether changes in such a perception affect external quality and productivity.

Finally, we believe that our results could encourage other researchers to contribute to increasing the body of knowledge on TDD applied to ESs development, especially by conducting laboratory or field studies with software professionals– after all, it is easier to involve software companies and professionals when promising preliminary results are available. We also made our replication package, including the raw data, available online to allow researchers to replicate our study and ease the execution of secondary studies (*e.g.,* meta-analyses) on TDD applied to ESs development [21].

6 Threats to Validity

To determine the threats that might affect our results, we followed Wohlin *et al.*'s guidelines [26]. Although we tried to mitigate/avoid as many threats to validity as possible, some of them are unavoidable. This is because mitigating/avoiding a kind of threat (*e.g.,* internal validity) might intensify/introduce another kind of

threat (*e.g.,* external validity) [26]. Since we conducted the first study (comprising two experiments where both quantitative and qualitative data are gathered) on the application of TDD in the development of ESs, we preferred to reduce threats to internal validity (*i.e.,* making sure that the cause–effect relationships were correctly identified), rather than being in favor of external validity.

Construct validity. We measured each construct with a single dependent variable (*e.g.,* external quality with QLTY). Thus, in the case of measurement bias, the study results would be affected (threat of *mono-method bias*). Although we did not disclose the research goals of our study to the participants, they might have guessed them and changed their behavior based on their guess (threat of *hypotheses guessing*). To mitigate a threat of *evaluation apprehension*, we informed the participants that they would obtain the bonus points on the final exam mark regardless of their performance in both Exp1 and Exp2. Finally, there might be a threat of *restricted generalizability across constructs*: *i.e.,* TDD might have influenced some non-measured constructs.

Conclusion validity. We mitigated a threat of *random heterogeneity of participants* through two countermeasures: *(i)* we involved students taking the same course, allowing us to have participants with a similar background, skills, and experience; *(ii)* the participants underwent a training period to make them as more homogeneous as possible within each experimental group. A threat of *reliability of treatment implementation* might have occurred (*i.e.,* some participants might have followed TDD more strictly than others). To mitigate this threat, during the experiment, we reminded the participants to use the development approach we assigned them. Finally, we did not perform any statistical inference given the number of participants and exploratory nature of our study. That is, if we had performed statistical inference, we would have suffered from a threat of *low statistical power* (*i.e.,* even if there had been a true effect, we would have been probably unable to reject the corresponding null hypothesis).

Internal validity. The participation in the experiments was voluntary. Since volunteers are generally more motivated to carry out new tasks than the whole population, this might affect the obtained results (*selection* threat). Another potential threat is *resentful demoralization–i.e.,* participants receiving a less desirable treatment might not behave as they normally would. To mitigate a possible threat of *diffusion or imitation of treatments*, we monitored the participants during the execution of the experimental tasks in Exp1 and checked any delivered project for plagiarism—no hint of plagiarism emerged.

External validity. The participants were Master's students, this might pose a threat of *interaction of selection and treatment—i.e.,* the results might not be generalized to other populations (*e.g.,* software professionals). However, the use of students has the advantage that they have more homogeneous backgrounds and skills, and allow obtaining initial empirical evidence [7, 13]. The implementation tasks, along with the used language and microprocessor, might represent another threat: *interaction of setting and treatment*. However, we opted for ESs that can be considered representative of the domain of interest for our study. Similar considerations can be done for Python and Raspberry Pi.

7 Conclusion and Final Remarks

In this paper, we present the results of an exploratory investigation, comprising two experiments, to study the external quality of ESs when applying TDD, as well as developers' productivity. Specifically, the participants in the two experiments carried out ESs development tasks in Python by applying TDD or a non-TDD approach (YW). Our results suggest that the external quality of the implemented solutions increases when using TDD as compared to YW, while there is not a substantial difference with respect to developers' productivity. However, TDD is perceived as more difficult to apply than YW, and the development task is deemed more challenging with TDD. These results have implications for lecturers and researchers. For example, it is worth teaching TDD in the context of ESs development, with a well-thought mix of theory and practice, including the deployment and testing in the target hardware. Also, despite we gather preliminary evidence that TDD can be successfully applied to the development of ESs, we foster replications of our experiments, especially by involving software companies and professionals. Our exploratory investigation has the merit to justify these replications since it is easier to involve software companies and professionals when promising preliminary results are available.

References

1. Embedded systems market by component, by application: global opportunity analysis and industry forecast, 2021-2031. Tech. rep., Allied Market Research (2022), `https://www.alliedmarketresearch.com/embedded-systems-market-A08516`
2. Baldassarre, M.T., Caivano, D., Fucci, D., Juristo, N., Romano, S., Scanniello, G., Turhan, B.: Studying test-driven development and its retainment over a six-month time span. J. Syst. Softw. 176, 110937 (2021)
3. Baldassarre, M.T., Caivano, D., Fucci, D., Romano, S., Scanniello, G.: Affective reactions and test-driven development: results from three experiments and a survey. J. Syst. Softw. 185, 111154 (2022)
4. Basili, V., Caldiera, G., Rombach, H.D.: Goal Question Metric (GQM) Approach, pp. 528–532. John Wiley & Sons, Ltd (2002)
5. Beck, K.: Test-driven development: by example. Addison-Wesley (2003)
6. Bissi, W., Neto, A.G.S.S., Emer, M.C.F.P.: The effects of test driven development on internal quality, external quality and productivity: a systematic review. Inf. Softw. Technol. 74, 45–54 (2016)
7. Carver, J., Jaccheri, L., Morasca, S., Shull, F.: Issues in using students in empirical studies in software engineering education. In: Proc. of International Symposium on Software Metrics. pp. 239–249. IEEE (2003)
8. Cohen, J.: A power primer. Psychol. Bull. 112, 155–9 (1992)
9. Cordemans, P., Van Landschoot, S., Boydens, J., Steegmans, E.: Test-Driven Development as a Reliable Embedded Software Engineering Practice, pp. 91–130. Springer (2014)
10. Fucci, D., Scanniello, G., Romano, S., Shepperd, M., Sigweni, B., Uyaguari, F., Turhan, B., Juristo, N., Oivo, M.: An external replication on the effects of test-driven development using a multi-site blind analysis approach. In: Proc. of International Symposium on Empirical Software Engineering and Measurement. pp. 3:1–3:10. ACM (2016)

11. Ghafari, M., Gross, T., Fucci, D., Felderer, M.: Why research on test-driven development is inconclusive? In: Proc. of International Symposium on Empirical Software Engineering and Measurement. pp. 25:1–25:10. ACM (2020)
12. Grenning, J.: Test Driven Development for Embedded C. Pragmatic Bookshelf Series, Pragmatic Bookshelf (2011)
13. Höst, M., Regnell, B., Wohlin, C.: Using students as subjects–a comparative study of students and professionals in lead-time impact assessment. Empir. Softw. Eng. 5(3), 201–214 (2000)
14. Karac, I., Turhan, B.: What do we (really) know about test-driven development? IEEE Software 35(4), 81–85 (2018)
15. Karlesky, M.J., Bereza, W.I., Erickson, C.B.: Effective test driven development for embedded software. In: Proc. of International Conference on Electro/Information Technology. pp. 382–387. IEEE (2006)
16. King, N.: Using templates in the thematic analysis of text. In: Cassell, C., Symon, G. (eds.) Essential Guide to Qualitative Methods in Organizational Research, pp. 256–270. Sage (2004)
17. Kitchenham, B., Madeyski, L., Brereton, P.: Meta-analysis for families of experiments in software engineering: a systematic review and reproducibility and validity assessment. Empir. Softw. Eng. 25(1), 353–401 (2020)
18. Madeyski, L., Kitchenham, B.: Effect sizes and their variance for ab/ba crossover design studies. Empir. Softw. Eng. 23(4), 1982–2017 (2018)
19. Munir, H., Moayyed, M., Petersen, K.: Considering rigor and relevance when evaluating test driven development: a systematic review. Inf. Softw. Technol. 56(4), 375–394 (2014)
20. Rafique, Y., Mišić, V.B.: The effects of test-driven development on external quality and productivity: A meta-analysis. IEEE Trans. Softw. Eng 39(6), 835–856 (2013)
21. Romano, S.: Replication package of "test-driven development and embedded systems: an exploratory investigation" (2023), `https://t.ly/aGNY`
22. Romano, S., Zampetti, F., Baldassarre, M.T., Di Penta, M., Scanniello, G.: Do static analysis tools affect software quality when using test-driven development? In: Proc. of International Symposium on Empirical Software Engineering and Measurement. pp. 80–91. ACM (2022)
23. Tosun, A., Dieste, O., Fucci, D., Vegas, S., Turhan, B., Erdogmus, H., Santos, A., Oivo, M., Toro, K., Jarvinen, J., Juristo, N.: An industry experiment on the effects of test-driven development on external quality and productivity. Empir. Softw. Eng. 22(6), 2763–2805 (2017)
24. Turhan, B., Layman, L., Diep, M., Erdogmus, H., Shull, F.: How effective is test-driven development. In: Making Software: What Really Works, and Why We Believe It, pp. 207–217. O'Reilly Media (2010)
25. Vegas, S., Apa, C., Juristo, N.: Crossover designs in software engineering experiments: Benefits and perils. IEEE Trans. Softw. Eng. 42(2), 120–135 (2016)
26. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer (2012)
27. Xiao, J.: Application of agile methods on embedded system development. In: Proc. of International Conference on Mechatronics, Materials, Biotechnology and Environment. pp. 667–670. Atlantis Press (2016)