Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di II livello in
Informatica

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

Supervisors
Giuseppe Scanniello
Simone Romano

Candidate
Michelangelo Esposito

Academic Year 2021-2022

Abstract

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A computer hardware and software combination created for a particular purpose is an Embedded System (ES). In many cases, ESs operate as part of a bigger system (*e.g.,* agricultural and processing sector equipment, automobiles, medical equipment, airplanes, and so on) and within tight resource constrains (*e.g.,* small battery capacity, limited memory and CPU speed, and so on). The global ESs market is expected to witness notable growth. A recent report evaluated the global ESs market 89.1 billion dollars in 2021, and this market is projected to reach 163.2 billion dollars by 2031; with a compound annual growth rate of 6.5% [1]. This growth is mostly related to an increase in the demand for advanced driver-assistance system (in electric and hybrid vehicles) and in the number of ESs-related research and development projects. To date, there has yet to be shown which approach is to be preferred when developing ESs. 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 development cycle made up of three phases: *red*, *green*, and *blue/refactor* [3]. During the red phase, the developer writes a unit test for a small chunk of a functionality not yet implemented and watches the test fail. In the green phase, the developer implements the chunk of functionalities as quickly as possible and watches all the unit tests pass. During the refactoring phase, the developer changes the internal structure of the code while paying attention to keep the functionality intact—accordingly, all unit tests should pass. TDD has been conceived to develop "regular" software, and it is claimed to improve software quality as well as developers' productivity [4]. ESs have all the same challenges of non-embedded systems (NoESs), such as poor quality, but add challenges of

their own [6]. For example, one of the most cited differences between embedded and NoESs is that embedded code depends on the hardware. While in principle, there is no difference between a dependency on a hardware device and one on a NoESs [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. Over the years, a huge amount of empirical investigations has been conducted to study the claimed effects of TDD on the development of NoESs (*e.g.,*, [5]). So far no investigations have been conducted to assess possible benefits concerned the application of TDD on the development of ESs although some authors, like Greening [2], believes that embedded developers can benefit from the application of TDD in the development of ESs.

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

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

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

experiments (task and the experimental procedure) were introduced to validate the results of the baseline experiment and to generalize these results to a more real setting. The results obtained by an aggregated analysis of the data from both the experiments suggest that there are no significant differences between TDD and NO-TDD on: productivity, number of written tests, a..., respectively. On the other hand, we observed a significant difference on: ... When analyzing experiments individually, we observed that ...

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

Chapter 2

Test Driven Development

2.1 Overview on software testing

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

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

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

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

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

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

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

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

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

2.1.1 Software Development Lifecycle

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

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

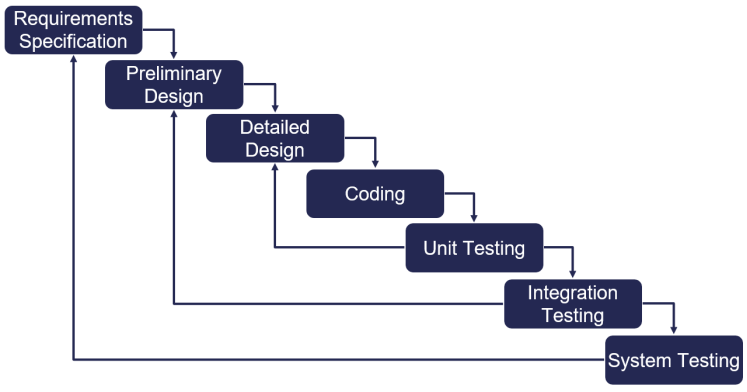


Figure 2.1: The Waterfall model

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

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

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

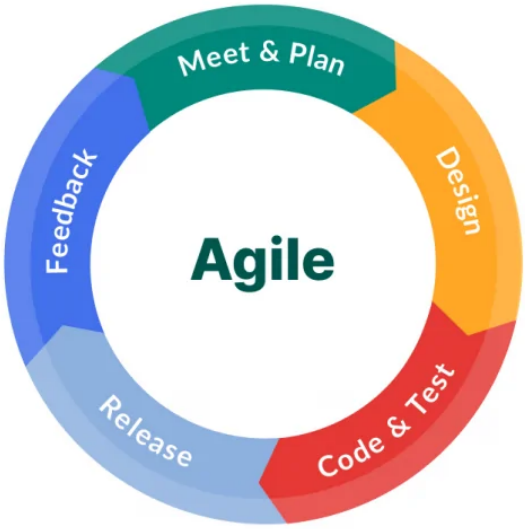


Figure 2.2: The Agile model

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

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

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

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

2.2 Test-Driven Development

2.2.1 Overview

Unit testing is arguably the most used testing technique since by itself it can already provide a general assessment of the quality and reliability of a software solution. TDD is a software development approach that builds on top of the concept of unit testing; it was firstly introduced in 2003 by Kent Back in the book "Test-Driven Development By Example" [3]; while there is no formal definition of the process, as the author states, the goal is to "write clean code that works". With TDD, test cases are written before any production code is written; these tests are used to define the requirements for the system and to guide the development process. The end goal of this practice is for all 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 SDLC approaches, TDD is an extremely short, incremental, and repetitive process, and is related to **test-first programming** concepts in agile development and extreme programming; this advocates for frequent

updates/releases for the software, in short cycles, while encouraging code reviews, unit testing and incremental addition of features.

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

- In the ***Red*** phase, a test case is written for the chunk of functionality to be implemented; since the corresponding logic does not exist yet, the test will obviously fail, often not even compiling.
- In the ***Green*** phase, only the code that is strictly required to make the test pass is written.
- Finally, in the Blue phase, the implemented code, as well as the respective test cases, is refactored and improved. It is important to perform regression testing after the refactoring to ensure that the changes didn't result in any unexpected behaviors in other components.

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

Figure 2.3 provides a visual representation of the TDD cycle:

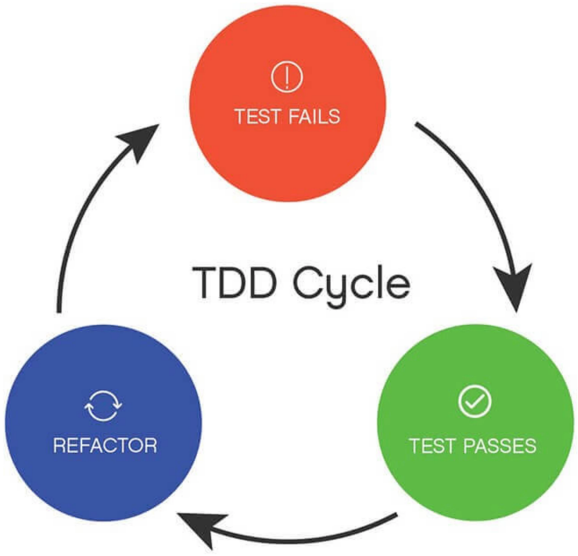


Figure 2.3: The Test Driven Development cycle

As previously stated, each TDD iteration should be extremely short, usually spanning from 10 to 15 minutes at most; this is possible thanks

to a meticulous decomposition of the system's requirements into a set of **User Stories**, each detailing a small chunk of a functionality specified in the requirements. These stories can then be prioritized and implemented iteratively.

User stories can vary in granularity: when using a fine-grained structure when describing the task, this can be broken up into a set of sub-tasks, each corresponding to a small feature; on the other hand, with coarser-grained tasks, this division is less pronounced [8]. Even when the same task is considered, the outcome of the TDD process will change depending on the level of granularity employed when describing it; there is no overall right or wrong approach, rather it is something that comes from the experience of the developer to break tasks into small work items [8].

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

2.2.2 TDD advantages

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

- **Regression testing**: by incrementally building a test suite as the different iterations of TDD are performed, we ensure that the system
- **Very high code coverage**: coverage is a metric used to determine how much of the code is being tested; it can be expressed according to different criteria such as statement coverage, i.e., how many statements in the code are reached by the test cases, branch coverage, i.e., how many conditional branches are executed during testing, or function coverage, i.e., how many functions are executed when running the test suite. While different coverage criteria result in different benefits, by employing TDD we ensure that any segment of code written has at least one associated test case.
- **Improved code quality**: as we are specifically writing code to pass the tests in place, and refactoring it after the "Green" phase, we ensure that the code is cleaner and overall more optimized, without any extra pieces of functionalities that may not be needed.
- **Improved code readability and documentation**: test act as documentation...

- **Simplified debugging and early fault detection**: Whenever a test fails it becomes obvious which component has caused the fault: by adopting this incremental approach and performing regression testing, if a test fail we will be certain that the newly written code will be responsible. For this reason, faults are detected extremely early during the testing process, rather than potentially remaining hidden until the whole test suite has been built and executed.

Chapter 3

Embedded Systems

3.1 Overview

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

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

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

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

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

3.2 Enabling technologies

3.2.1 Microcontrollers

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

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

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

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

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

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

3.2.2 Embedded software and communication protocols

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

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

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

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

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

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

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

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

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

3.3 Design 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 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 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 power 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) [10]. 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 (*i.e.,* weather monitoring in extreme locations of the planet, satellite managements systems, devices inside the human body, and so on), where maintenance operations cannot be performed regularly, and high availability is expected. The dependability

of an ES can be expressed in terms of:

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

- **Reliability**: the extent to which a system is reliable with respect to the expected behavior. To improve reliability, ES should be designed with robust error-detection and correction mechanisms.

- **Availability** is 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 the system is able to handle the challenges of limited bandwidth, high latency, and unreliable connectivity.

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

3.4.1 X-in-the-loop

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

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

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

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

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

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

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

Figure 3.1 ...aaa

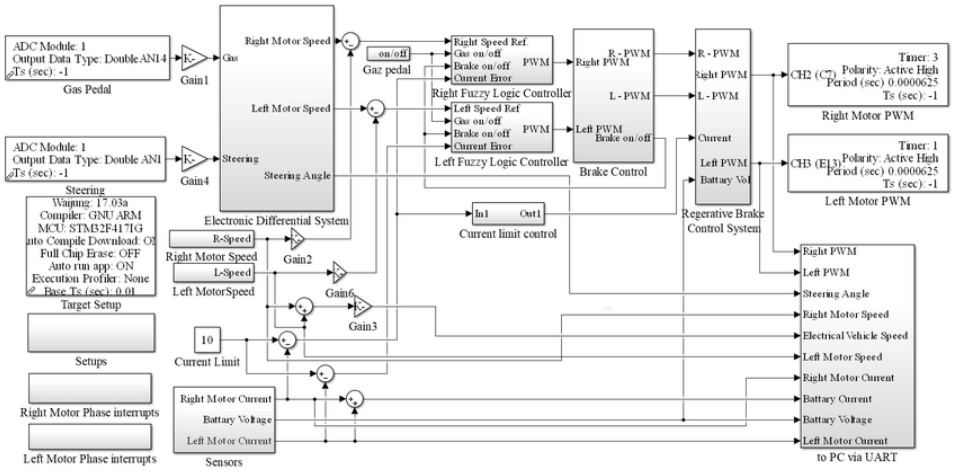


Figure 3.1: Some components of a Simulink model for an autonomous driving system

3.4.2 A TDD pipeline for Embedded Systems

Given the potential benefits of TDD for NoESs, it comes natural to ask whether this technique could be applied to the development of ESs. By applying TDD for ESs, developers would write automated tests that would be used to verify the functionality and performance of the ES. As a reminder, these tests would be written before any 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 reasons we discussed above, TDD can be a challenging approach when applied to ESs. However, with the right tools, techniques, and experience, it is possible to apply it to ESs and to achieve the benefits of this approach. Often in fact, quality in embedded software is tied to platform-specific testing tools geared towards debugging and testing [15]. Applying TDD practices to ESs could, in theory, result in a series of benefits:

- Reduce risk by verifying production code, independent of hardware, before hardware is ready or when hardware is expensive and scarce.
- Reduce the number of long target compile, link, and upload cycles that are executed by removing bugs on the development system.

- Reduce debug time on the target hardware where problems are more difficult to find and fix.
- Isolate hardware/software interaction issues by modeling hardware interactions in the tests.
- Improve software design through the decoupling of modules from each other and the hardware. Testable code is by necessity, modular.

In [2], the author proposes the "Embedded TDD Cycle", as a pipeline made of the following steps:

1. **TDD micro-cycle**: this first stage is the one run most frequently, usually every few minutes. During this stage, a bulk of code is written in TDD fashion, and compiled to run on the host development system: doing so gives the developer fast feedback, not encumbered by the constraints of hardware reliability and/or availability, since there are no target compilers or lengthy upload processes. Furthermore, the development system should be a proven and stable execution environment, and usually has a richer debugging environment compared to the target platform. Running the code on the development system, when it is eventually going to run in a foreign environment can be risky, so it's best to confront that risk regularly.
2. **Compiler Compatibility Check**: periodically compile for the target environment, using the cross-compiler expected to be used for production compilations; this stage can be seen as an early warning system for any compiler incompatibilities, since it warns the developer of any porting issue, such as unavailable header files, incompatible language support, and missing language features. As a result, the written code only uses facilities available in both development environments. A potential issue at this stage is that in early ES development, the tool chain may not yet be decided, and this compatibility check cannot be performed: in this case, developers should take their best guess on the tool chain and compile against that compiler. Finally, this stage should not run with every code change; instead, a target cross-compile should take place whenever a new language feature is used, a new header file is included or a new library call is performed.
3. **Run unit tests in an evaluation board**: compiled code could potentially run differently in the host development system and the target embedded processor. In order to mitigate this risk, developers

can run the unit tests on an evaluation board; with this, any behavior differences between environments would emerge, ad since runtime libraries are usually prone to bugs [2], the risk is real. If it's late in the development cycle, and a reliable target hardware is available, this stage may appear unnecessary.

4. **Run unit tests in the target hardware**: the objective here is again to run the test suite, however this time doing so while exercising the real hardware. One additional aspect to this stage is that developers could also run target hardware-specific tests. These tests allow developers to characterize or learn how the target hardware behaves. An additional challenge in this stage is limited memory in the target. The entire unit test suite may not fit into the target. In that case, the tests can be reorganized into separate test suites, where each suite fits in memory. This, however, does result in more complicated build automation process.

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

Chapter 4

Literature Review

4.1 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 [16] [17] [18], including the ones of interest for this study, *i.e.,* software/functional quality and productivity. These studies are summarized in Systematic Literature Reviews (SLRs) and meta-analyses (*e.g.,* [19, 20, 21, 22]).

The SLR by Turhan *et al.* [22] includes 32 primary studies (2000-2009). The gathered evidence shows a moderate effect in favor of TDD on quality, while results on productivity is inconclusive, namely there is no decisive advantage on productivity for employing TDD.

Bissi *et al.* [19] conducted 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.* [22], 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 [21] conducted a meta-analysis of 25 controlled experiments (2000-2011). The authors observed a small effect in favor of TDD on functional quality, while again for productivity the results are inconclusive. Finally, Munir et al. [20] in their SLR classifies 41 primary studies (2000-

2011) according to the combination of their rigor and relevance. The authors found different conclusions for both functional quality and productivity in each classification.

An example of long-term investigation is the one by Marchenko et al. [23]. The authors conducted a three-year-long case study about the use of TDD at Nokia-Siemens Network. They observed and interviewed eight participants (one Scrum master, one product owner, and six developers) and then ran qualitative data analyses. The participants perceived TDD as important for the improvement of their code from a structural and functional perspective. 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.* [24] executed a long-term study in-the-wild covering 594 open-source projects over the course of 2.5 years. They found that only 16 developers use TDD more than 20% of the time when making changes to their source code. Moreover, TDD was used in only 12% of the projects claiming to do so, and for the majority by experienced developers.

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

Latorre [26] studied the capability of 30 professional software developers of different seniority levels (junior, intermediate, and expert) to develop a complex 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 only a short practice session, the participants were able to correctly apply TDD (*e.g.,* following the prescribed steps). They followed the TDD cycle between 80% and 90% of the time, but initially, their performance depended on experience, 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 [26] 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.* [27] 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.* [28] provided a systematic literature mapping 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 to identify 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 are among the most popular ones. Most of the review papers (137 of 312, around 43.9%) present solution proposals without rigorous empirical studies; 98 (31.4%) papers are weak empirical studies (validation research). 36 (11.5%) are experience papers. 34 are strong empirical studies (evaluation research). 2 and 5 papers, respectively, are philosophical and opinion papers.

In terms of level of testing considered in the papers, most of them (233

papers) considered system testing. 89 and 36 papers, respectively, focused on unit and integration testing; among the ones focused on unit testing, only two of the sources ([29], [30]) 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. Most of these papers (142, 52.2 percent) 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 is a good mix of papers proposing techniques and tools for each of the test activities, with a major focus on test execution, automation and criteria-based test case design (*e.g.,* based on code coverage). 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.

Chapter 5

Experimental Approach

5.1 Overview

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

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

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

Overall, this experimental study aims to contribute to the body of knowledge in the field of empirical software engineering by providing new insights and understanding into the application of TDD for ES development. Although the preliminary 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 the ESs; therefore, our results can have several practical implications. For example, they could provide initial evidence on the application of TDD to the development of ESs so justifying future research on this matter and/or promoting or discouraging its adoption in an industrial setting.

5.2 Research Questions

As for the baseline experiment, and following the main research question presented in the introduction section of this thesis, we defined the main goal of this study by applying the Goal/Question/Metrics (GQM) template [31]. According to the GQM 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.

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 ESs and the developers' productivity **from the point of** view of the researcher and lecturer **in the context of** an ES course involving second year Master's degree students in Computer Science.

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

- **RQ1.** Does the use of TDD impact the external quality of the developed ESs? 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 external 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 development approach 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 better quality ESs, 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: 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).

5.3 Participants

The participants for the two experimental studies 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; this last group was further made up of Master's degree students and third-year Bachelor's degree students. Both groups were enrolled in the Master's degree course of *Embedded Systems* at the University of Salerno. As for the Master's students, not all of them had a computer science background (Bachelor's degree). Among the students taking the course, 9 participated.

The 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 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].

Participating in the studies 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 didn't participate. Finally, as an incentive to encourage participation, those who took part in the studies were rewarded with 2 bonus points in their final mark for the course, in line with Carver *et al.*'s advice [33].

Before the studies took place, we collected some information on the participants' knowledge and experience programming and testing, 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 a Likert scale (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 their scale to represent their unit-testing experience; finally, three students had heard about TDD, 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 studies 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. All three tasks were designed according to a target platform, a *Raspberry Pi Model 4*, and around the *Python* programming language. As for why this environment was chosen, compared to other programming languages and platforms typically used for ES or IoT projects (*e.g., Arduino* with *C++*), in our opinion focusing on a higher level language such as *Python* (which is still employed for ESs, either in its standard form or in its *MicroPython* variant) 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 to get a grasp of the whole picture. Finally, participants relied on the *PyCharm IDE* to implement all three tasks.

The target platform was considered even for the two tasks of the baseline study, which in the end did not have to be effectively deployed on the real hardware; this was done in order to make the mocked implementation of the participants resemble as close as possible the embedded implementation. The main mocked component utilized was a facade for the *Raspberry Pi*'s General Purpose Input/Output (GPIO) library, available as open source software [34]. Other mocked components include third party libraries for some individual sensors and actuators. For the replication study, another factor that influenced the design of the kata was the availability of the hardware, including sensors and actuators, as well as how hard these would have been to test in real time; for example, testing .... would 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 replication project was tested inside a laboratory at the University of Salerno. Each participant had their project uploaded on the *Raspberry Pi* board and ...

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

5.4.1 IntelligentOffice

This task required developing an intelligent office system, which allowed for handling light and CO2 levels inside an office. To handle the light, the system relied on the information gathered from different sensors—*i.e.,* Infra-Red (IR) distance sensors, a real-time clock, and a photoresistor—and then controlled a servo motor (to open/close the blinds) and a light bulb. A carbon dioxide sensor was used to monitor the CO2 levels inside the office and then control the switch of an exhaust fan.

5.4.2 CleaningRobot

The second task for the baseline experiment required participants to develop a cleaning robot that, while moving in a room based on the commands received from an external system, cleaned the dust on the floor along the way.

To clean the dust, the robot controlled a cleaning system. The robot also detected potential obstacles through an IR distance sensor. An intelligent battery sensor 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.

5.4.3 SmartHome

This task required developing a smart room system, which allowed for handling light, temperature, and gas levels inside a room. To handle the light, the system relied on the information gathered from an IR distance sensor and a photoresistor, and then turned on/off a light bulb. The information from a pair of temperature sensors was used to control a servo motor of a window. The system was also equipped with an air quality sensor to measure the gas levels inside the room and, if necessary, trigger an alarm through a buzzer.

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 ES testing techniques.

As the pre-questionnaire for the study highlighted, few students had unit testing experience, and almost no participants ha dealt with TDD up to that point. For this reason, these topics were covered through a series of frontal lectures and exercise/homework sessions in the weeks preceding the studies.

All the participants attended these lectures and training sessions during which they were trained on the main topics they would encounter during the future experimental tasks. The overall schedule for training sessions and studies was subdivided in the following days:

- **D1.** Frontal lecture - Introduction on unit testing and its guidelines. Interactive exercise on unit testing.
- **D2.** Frontal lecture - Test scaffolding and *Raspberry Pi*'s GPIO library. Interactive exercise on test scaffolding.
- **D3.** Frontal lecture - Introduction to Test-Driven Development. Interactive exercise on TDD.

- **D4.** Training task - TDD exercise and homework.
- **D6.** First experimental task for the baseline study - *IntelligentOffice* (*IO*).
- **D7.** Second experimental task for the baseline study - *CleaningRobot* (*CR*).
- **D8.** Start date for the replication study's task - *SmartHome* (*SH*).

The first task, *IO* took place on Tuesday, December 6th 2022, while the second, *CR* took place on Tuesday, December 13th 2022.

The experimental design of our baseline study is an ABBA crossover [35]; 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., 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, *IntelligentOffice*, while the group *G2* was assigned the NO-TDD version; on the other hand, during period *P2*, the group *G1* was assigned the NO-TDD version of the second task, *CleaningRobot*, while the group *G2* was assigned the TDD version. Therefore, at the end of the baseline study, every participant had tackled each experimental object only once. As for the replication study, the group structure remained the same, however each participant was randomly assigned the TDD or NO-TDD version of the third and final experimental task, *SmartHome*. As a result, the design of this study is *one-factor-with-two-treatments* [36], a kind of *between-participants* design. The assignment of the participants for done randomly.

After each period of the controlled baseline study, 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 question, asking to provide information about the chosen approach for testing. Furthermore, the post-questionnaire presented at the end of period *P2* contained an additional open-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 (*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 (*i.e.,* TDD or NO-TDD).
- Provide your feelings (both positive and negative) about the development task.
- (**Task 2 only**) After applying (*i.e.,* TDD or NO-TDD) in the last exercise, do you have any thoughts on the differences between the two approaches and your preference for using one over the other?

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

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
   - Please provide a discussion on TDD vs. NO-TDD in the development of ESs

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 ***Condition***, a nominal variable assuming two values, TDD and NO-TDD. The data was collected over two periods for the controlled study, and over an additional period for the non-controlled study, so a second independent variable is ***Period***, assuming the values $P1, P2, and P3$. During the three periods both treatments (TDD or NO-TDD) were applied. Finally, since the participants were split into two groups, the last independent variable is **Group**, which can assume the values *G1* and *G2*.

As for the dependent variables considered in the studies, these are: ***QLTY***, ***PROD***, ***TEST***, ***CYC***, ***COG***, ***LOC***. The variables $QLTY$, $PROD$ and $TEST$ have been used in previous empirical studies on NoESs [17], [37], [27], [38]. As for the others variables

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

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

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

passed, than the story was considered as tackled. $\#TUS$ is formally defined as follows:

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

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

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

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

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

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

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

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

As for the additional three dependent variables, these were computed using SonarQube and provide insight regarding the internal quality in the implemented software solution, in terms of how hard the production code is to comprehend and 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 program module. A program with a lower Cyclomatic Complexity is generally easier to understand and less "risky" to modify; it can be used as an estimate on how difficult the code will be to cover/test.

(COG) is the Cognitive Complexity of the solution; it is a measurement of how difficult a program module is to intuitively understand. A method's

Cognitive Complexity is based on a few rules [39]:

1. Code is not considered more complex when it uses shorthand syntax that the language provides for collapsing multiple statements into one.

2. Code is considered more complex for each "break in the linear flow of the code".

3. Code is considered more complex when "flow breaking structures are nested"

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

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 easy maintenance and extendability.

5.7 Analysis Methods

5.7.1 Individual analysis

As a first way to examine the distribution of the dependent variables, we compute the descriptive statistics, for each of them (*i.e.,* $QLTY$, $PROD$, $TEST$, CYC, COG, and LOC). 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.

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

...

Some observations that can be made include:

- The box plot is comparatively short (see box plot (2)). This suggests that overall the student have a high level of agreement and therefore the values are very similar between each other.
- The box plot is comparatively tall (see box plots (1) and (3)). This, on the other hand, suggests students hold quite different opinions about this aspect or sub-aspect.
- Obvious differences between box plots (see box plots (1) and (2), (1) and (3), or (2) and (4)). Any obvious difference between box plots for comparative groups is worthy of further investigation.
- The 4 sections of the box plot are uneven in size (see box plot (1)). This reveals that many students share a similar view at certain parts of the scale, but in other parts of the scale students are more variable in their views. A long upper whisker in the means that students views

are varied among the most positive quartile group, and very similar for the least positive quartile group.

- Same median, different distribution (see box plots (1), (2), and (3)). The medians, which generally tend to be close to the average, are at the same level. However, the box plots in these examples show very different distributions of views. It's always important to consider the pattern of the whole distribution of responses in a box plot.

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 effect. There are different methods used to conduct a meta-analysis, but generally, the process involves identifying relevant studies, extracting data from them, and then analyzing the data using statistical techniques. The most common method of meta-analysis is the random-effects model, which accounts for between-study variation in addition to within-study variation.

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. To read a forest plot chart, you need to understand the following elements:

- Study name: This is the name of the study that is represented in the plot.
- Study size: This is the number of participants
- Effect estimate: This is the measure of the treatment effect for each study, typically shown as a point estimate (e.g. odds ratio, mean difference, relative risk)
- Confidence interval: This is the range of values within which the true effect is likely to lie, typically shown as a horizontal line.

In our case, we considered the baseline and replication as the two studies for the meta-analysis.

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. The researcher then organizes the coded data into themes and sub-themes, which are then used to construct a narrative or 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 [40]. In Template Analysis, it is permissible (though not obligatory) to start with some a priori themes, identified in advance as likely to be helpful and relevant to the analysis. These are always tentative, and may be redefined or removed if they do not prove to be useful for the analysis at hand.

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 variables analysis

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

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

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

each of the two conditions.

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

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

(a) QLTY

(b) PROD

(c) TEST

(d) CYC

(e) COG

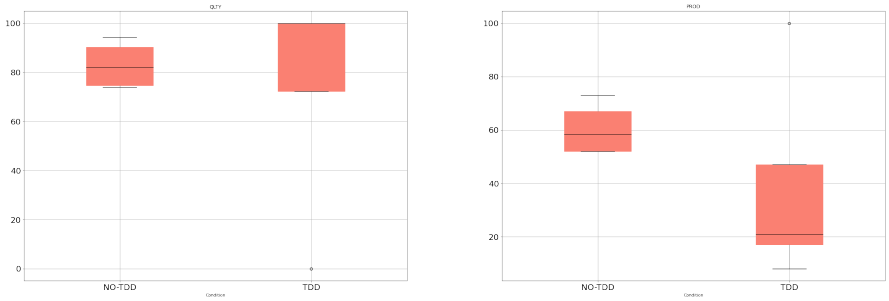
(f) LOC

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

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

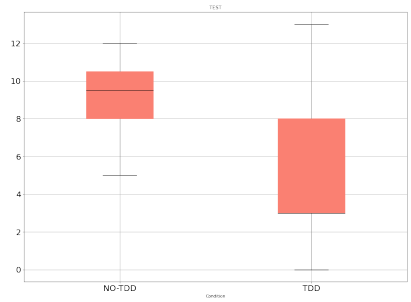
applying each of the two conditions.

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

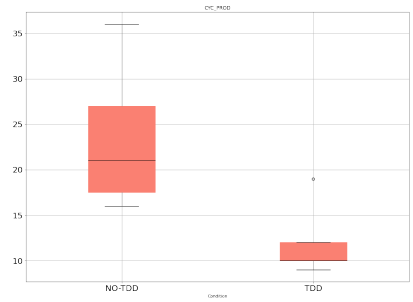


(a) QLTY

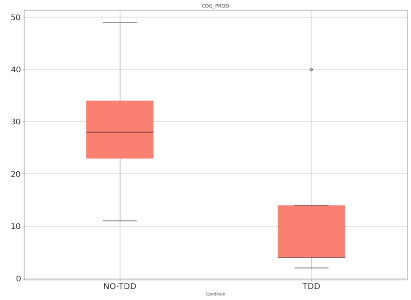
(b) PROD



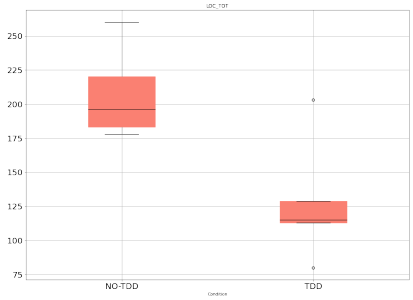
(c) TEST



(d) CYC



(e) COG



(f) LOC

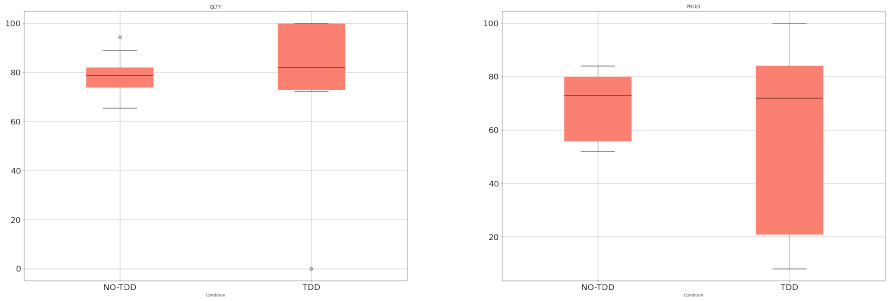
Figure 5.2: Box plots for task 2, *CleaningRobot*

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

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

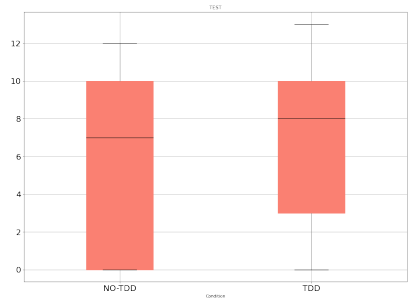
Task 1 & 2 - TDD					
Metric	Min	Max	Mean	Median	Std
QLTY	0	100	77.41	82	31.52
PROD	8	100	58.33	72	35.76
TEST	0	13	7.22	8	4.26
CYC	9	28	17.66	19	7.51
COG	2	40	15.55	14	12.06
LOC	80	203	145.33	154	39.97

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

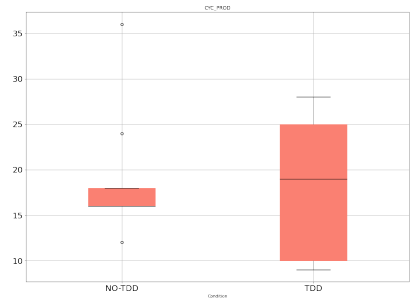


(a) QLTY

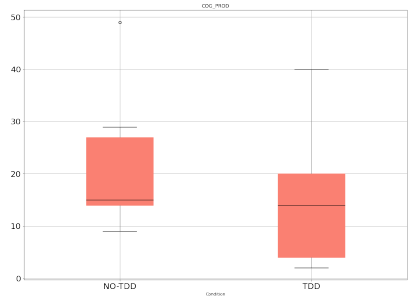
(b) PROD



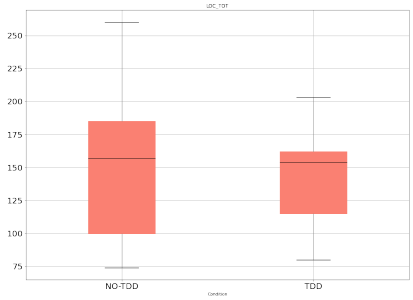
(c) TEST



(d) CYC



(e) COG



(f) LOC

Figure 5.3: Aggregated box plots for tasks 1 and 2

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

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

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

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

(a) QLTY

(b) PROD

(c) TEST

(d) CYC

(e) COG

(f) LOC

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

5.8.2 Meta-analysis

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

Forest plots are ...

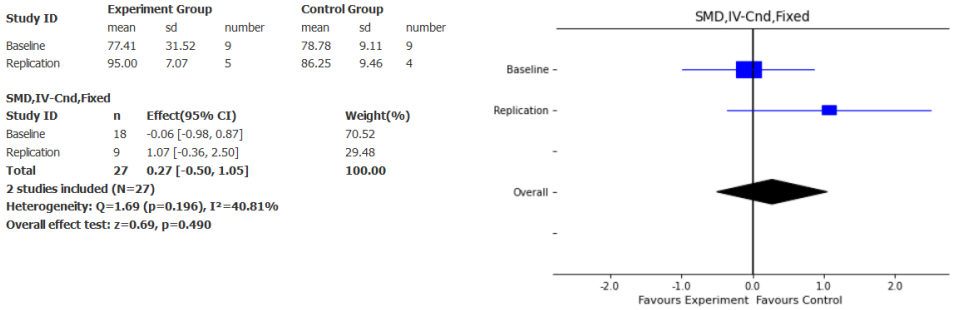


Figure 5.5: Forest plot chart for the QLTY dependent variable

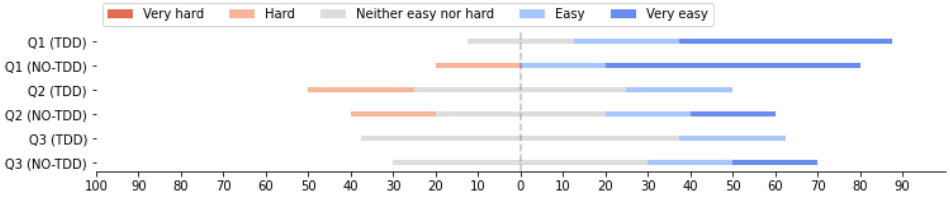
5.8.3 Post-questionnaire Analysis

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

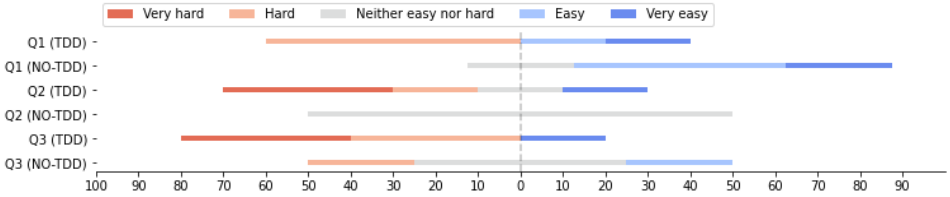
A similar trend can be noted for the second question, regarding the general task feasibility; here, in the first experimental task, the participants somewhat agree, with 50% of the TDD group and 40% of the NO-TDD group feeling neutral about the difficulty of the task. In the second task, the answers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

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

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



(a) First experimental task, *IntelligentOffice*



(b) Second experimental task, *CleaningRobot*

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

Template analysis of the interview for the replication study

Chapter 6

Discussion

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

6.1 Answers to Research Questions

6.2 Implications

6.3 Threats to Validity

According to Campbell and Stanley [41], 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 [42]. External validity refers to showing a real-world effect, but without knowing which factors actually caused the observed difference [42]. Cook and Campbell extended the list of validity categories to: conclusion, internal, construct, and external validity [43]. The latter classification has often been adopted in past empirical software engineering studies [36].

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

1. **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. Threats to internal validity include:

- **Deficiency of treatment setup**: the treatment setup is sometimes not appropriate, which may impact the results. For example, noise and tool performance could impact the results of a study, when they are not related to the treatment of the study.
- **Ignoring relevant factors**: factors not considered in the experiment setup sometimes impact the study results, such as the usability of the tools used in the experiment and their performance.
- **History**: A study composed of a set of treatments applied at different occasions may be impacted by the history threat. Treatments may be given to the same object at several occasions, each of which is associated with specific circumstances, such as time and location. The change in circumstances may impact the results.
- **Maturation:** the subjects may react differently as time passes while they perform the treatment: some may become bored and others may become motivated.
- **Testing**: the subjects may behave differently towards the treatment if they do it several times: they may learn the results and adapt their responses accordingly.
- **Treatment design**: the artifacts used in the treatment, such as the data collection form and the documents used as information source, could affect the results if not well-designed and tested.
- **Subject selection**: subjects for studies are selected to represent a population. The selection method affects the results and their interpretation. The group of subjects that participate in a study is always heterogeneous. The difference between individuals should not be the dominant factor for the study results: the treatment should be the dominant factor.
- **Sample selection**: data is usually collected from data sources that represent the context of the study, such as NVD database,

or open-source logs and artifacts. The data sample should be representative of the studied type of data.

- **Incompleteness of data**: researchers often use heuristics or keyword searches to select records from data sources that represent the data required for the given study. These techniques may fail to identify all the expected records from the data sources.
- **Mortality**: some subjects selected for a given treatment may drop out of the treatment. This should be considered when evaluating the impact of the given treatment on the subjects. Drop-out subjects should be removed from the treatment.
- **Imitation of treatment**: this applies to studies that require different subjects/groups to apply different methods/techniques and use the responses to compare the methods and techniques. The subjects/groups may provide responses influenced by their experience and knowledge about the evaluated methods if they learn that these methods/techniques are being applied by other subjects/groups.
- **Motivation**: a subject may be motivated or resistant to use a new approach/method/technique. This may affect their response/performance in applying either the old or the new approach/method-/technique

2. **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? Threats to external validity include:

   - **Representation of the population**: the selected subjects/-groups should represent the population that the study applies to.
   - **Representation of the setting**: the setting of the study should be representative of the study goal. For example, tools used in the study should represent a real setting, not old ones.
   - **Context of the study**: The time and location of the study impacts the ability to generalize its results.

3. **Construct validity** refers to the extent to which a study's measures are related to the theoretical construct being studied. Threats to construct validity include:

- **Theory definition**: the measured variables may not actually measure the conceptual variable. An experiment derived from an insufficiently defined theory does not represent the theory.
- **Mono-operation bias**: the study should include more than one independent variable, one treatment, and one subject. Discovering a phenomenon from one variable, case, or subject implies that a theory may exist but may not confirm the theory.
- **Mono.method bias**: using only one metric to measure a variable results in a measurement bias that can mislead the experiment.
- **Appropriateness of data**: researchers often use heuristics or keyword searches to select records from data sources. These techniques may result in the extraction of records that are not related to the given study.
- **Experimenter bias**: this happens when a researcher classifies artifacts /data based on his/her own perception or understanding rather than an objective metric. The perception may not be correct.
- **Measurement metrics**: the measurement method and the details of the measurement impact the study results.
- **Interaction with different treatments**: a subject that participates in a set of treatments may provide biased responses; his/her responses could be impacted by the interactions of the treatments of the study.
- **Treatment testing**: a study construction needs to be tested for quality assurance. However, the responses of subjects participating in the study test are affected by their experience with the treatment.
- **Hypothesis guessing**: some subjects try to figure out the intended outcomes of studies they are involved in and adapt their responses based on their guesses.
- **Evaluation apprehension**: subjects may behave differently when evaluated, *e.g.,* review their code more thoroughly. This impacts the truth of the evaluated responses.
- **Experimenter expectations**: the subjects may have expectations of the experiment and may provide answers accordingly. The study should formulate the treatment to mitigate that, such as asking the questions in different ways.

4. **Conclusion validity** refers to the extent to which the conclusions drawn from a study are supported by the data. Threats to conclusion validity include:

    - **Statistical validity**: statistical tests have confidence and power, which indicate the ability of the test to assert a true pattern. Low confidence (or power) implies that the results are not conclusive and don't permit deriving conclusions.
    - **Statistical assumptions**: some statistical tests and methods (*e.g.,* prediction and forecasting) use assumptions, such as normality and independence of the data, or independence of the variables. Violations or absence of tests for the assumptions for a given test/method threaten the ability to use the given test/algorithm.
    - **Lack of expert evaluation**: interpreting the results often requires having deep knowledge about the context of the collected data. The results may also include critical hidden facts, which only experts can point out.
    - **Fishing for results**: fishing for specific results (often results that conform to the researcher hypotheses) impacts the study setup and design. The researcher could "unintentionally" draw conclusions that are not correct for the study setup and design.
    - **Reliability of the measures**: measurements of independent variables should be reliable: measuring the concept twice should provide the same result. Questionnaire wording is an example of causes of this threat.
    - **Reliability of treatment implementation**: The implementation of the treatment should follow a standard, and it should be the same for all subjects.
    - **Lack of data preprocessing**: The quality of raw data is often not excellent. Researchers need to explore them to identify problems, such as missing data, outliers, and wrong data values, e.g., values that do not follow the codification rules.

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

As mentioned above, there are inherent trade-offs between validities: with internal and external validities for example, the more we control extraneous

factors in your study, the less we can generalize our findings to a broader context.

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

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

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

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

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

Chapter 7

Conclusions

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

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

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] 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.

[27] 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.

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

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

[30] 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.

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

[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] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

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

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

[41] 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).

[42] 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.

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

Appendices

Intelligent Office - TDD Group

Goal

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

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

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

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

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

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

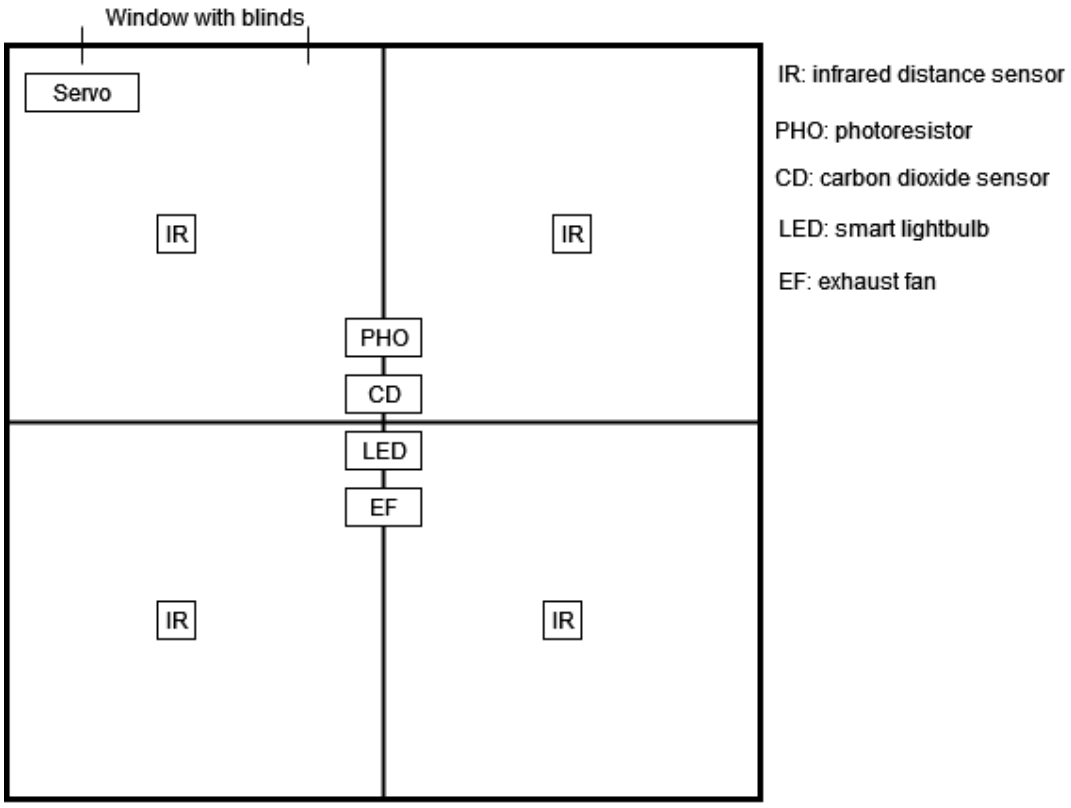
To recap, the following sensors and actuators are present:

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

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

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

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



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

Instructions

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

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

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

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

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

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

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

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

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

User Stories

Remember to use TDD to implement the following user stories.

1. Office worker detection

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

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

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

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

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

Requirement:

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

2. Open/close blinds based on time

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

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

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

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

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

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

duty cycle = (angle / 18) + 2

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

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

Requirement:

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

3. Light level management

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

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

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

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

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

Requirement:

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

4. Manage smart light bulb based on occupancy

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

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

Requirement:

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

5. Monitor air quality level

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

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

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

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

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

Requirement:

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

Cleaning Robot - no-TDD Group

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 below 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 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 at the bottom-left corner. A cell of the grid may contain or not an obstacle. The RMS keeps track of the room layout, including the last known positions of the obstacles in the room.

To recap, the following sensors, actuators, and systems are present:

- A DC motor to control the wheels in order to move the robot forward.
- A DC motor to control the rotation of the body of the robot, in order to make it rotate left or right.
- An RMS, sending commands to the robot.
- An infrared distance sensor used to detect obstacles.
- An IBS to determine the battery charge left.
- A recharge LED.
- A 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** class in the source code.

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

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.

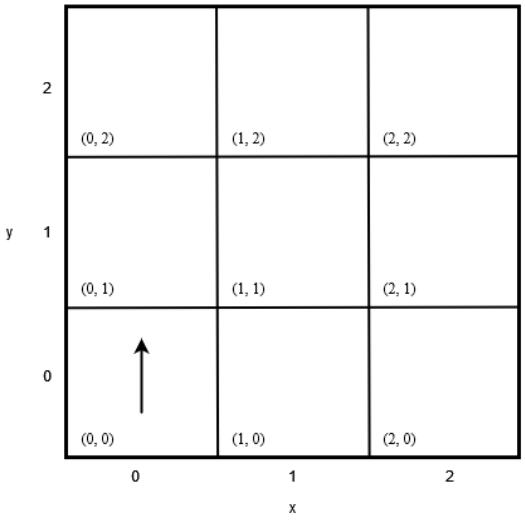
At the end of the task, fill out the post-questionnaire where, among other things, you will be asked to share your project (either as a link to a GitHub repository or as a sharing link to a ZIP file). The post-questionnaire is available at: https://forms.gle/GEXHybbDJujZJz2V6.

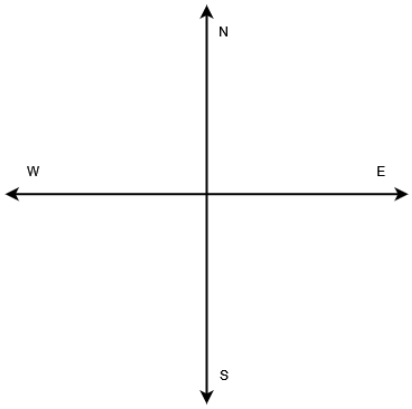
User Stories

Remember to use **no-TDD** to implement the following user stories.

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 the figure below).





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.

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.

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. 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`)

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 that obstacle. If an obstacle is detected, the robot cannot move beyond the obstacle; 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)(xo,yo)”. The triple “(x,y,h)” represents the usual position and heading of the robot while the pair “(xo,yo)” 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:

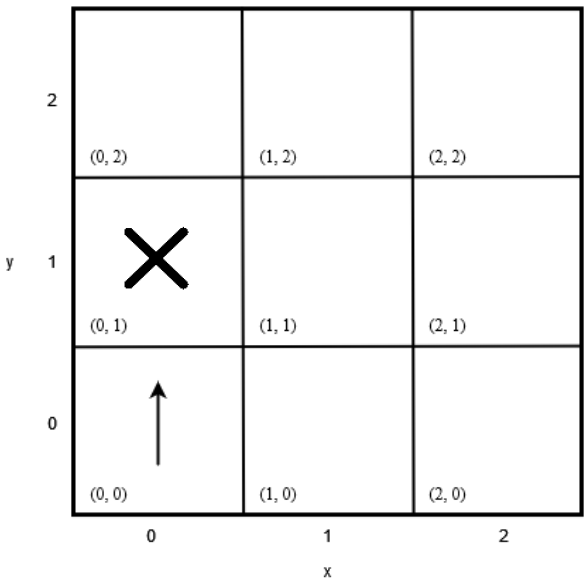
- Non-zero value: it indicates that an object is present in front of the sensor (i.e., an obstacle).
- Zero value: nothing is detected in front of the sensor.

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 below). The robot with initial status "(0,0,N)", after executing the command string "f", returns the following status: "(0,0,N)(0,1)".



Hint:

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

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) -> str` to control the behavior associated with the battery level.
- Update the implementation of `CleaningRobot.manage_battery() -> None` to control the behavior associated with the battery level.

Smart Home - TDD Group

Constraints

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

Goal

The goal of this task is to develop an intelligent system to manage a smart room in a house. First of all, an **infrared 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:

- An infrared distance sensor located on the ceiling to the room.
- A smart light bulb.
- A photoresistor sensor to measure the light level inside the room.
- A 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.
- An air quality sensor, used to measure the gas level inside the room.
- An 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** class 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.

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.

How to Deliver Your Project

1. Reserve a slot for the exam 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) to: gscanniello@unisa.it, siromano@unisa.it, and m.esposito253@studenti.unisa.it.

   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.

User Stories

Remember to use TDD to implement the following user stories.

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 this sensor is the opposite of 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 of it by using the infrared distance sensor.

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 smart home system turns off the light bulb 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 a LED, connected to the main board via pin GPIO26 (BCM mode). Communication with the LED 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.

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:

- Implementat `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.

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 to 30 degrees celsius. 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 to 180 degrees. We can control it by sending a PWM (Pulse-Width Modulation) signal to its signal pin; this means sending a HIGH signal for a certain period of time (called duty cycle), followed by a LOW signal period. The duty cycle determines the angle the servo motor will rotate to.

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

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

duty cycle = (angle / 18) + 2

The servo motor has already been configured and can be controlled by calling the `ChangeDutyCycle(duty_cycle: float)` 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.

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.

Thematic Analysis – Baseline

Template

1. **Feelings on the development task**
    a) **Task1**
        - Easy / went smoothly **(1)**
        - Some difficulties / Needed more time **(2)**
    b) **Task 2**
        - No problem with the task (2)
        - Task was harder than last time and I struggled **(3)**
        - Task was harder than last time but no problem **(2)**
2. **Feelings on TDD to accomplish the task**
    a) **Task1**
        - Good experience **(1)**
        - Still not sure **(2)**
    b) **Task 2**
        - Good experience **(1)**
        - Mixed opinion **(1)**
        - Having troubles **(3)**
3. **Feelings on NO-TDD to accomplish the task**
    a) **Task1**
        - Good experience (I'm used to NO-TDD) **(2)**
        - Concerns on not testing enough some components **(1)**
        - **NO-TDD approach applied**
            - Wrote tests after the production code **(3)**
            - No tests **(2)**
    b) **Task 2**
        - Easier to use NO-TDD **(2)**
        - Not using TDD was harder **(2)**
        - **NO-TDD approach applied**
            - Wrote tests after the production code **(4)**
4. **Comparison between TDD and NO-TDD**
    a) TDD preference **(4)**
    b) NO-TDD preference **(3)**
    c) Depends on the task **(1)**

Task 1:

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 it’s not as error prone as regular development. It increases software quality. And it assures that tests exist. But I’m honest, TDD is sometimes hard. Because you have to think different as usual methods.
- **Q2**: Task 4 - i really had to think deeply into this. Because implementing this breaks Task 3 tests

Student_04 (TDD):

- **Q1**: I have yet to fully understand how it works.
- **Q2**: The task I find it very useful to get a good understanding of the various steps to be done.

Student_05 (NO-TDD)

- **Q1**: I just programmed one task after the other, like we learned in the lessons ago.
- **Q2**: I think for me it’s easier to program a NO-TDD software, because I never did a 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.
- **Q3**: I liked that there were similarities with the previous tasks.

Student_06 (NO-TDD)

- **Q1**: I still have some difficult during the test phase, maybe is simpler doing it with TDD approach.
- **Q2**: For me is simpler because i can focus on the code.
- **Q3**: Are clear and not difficult to implement.

Student_07 (NO-TDD)

- **Q1**: 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.
- **Q2**: NO-TDD may be faster when you are doing easy staff but following this kind of approach may incur in same difficult debugging session, there are the possibility to implement some code that may be never tested or not tested well.
- **Q3**: this development task has been useful to understand the difference with using TDD and its advantages in respect to use NO-TDD

Student_08 (NO-TDD)

- **Q1**: The knowledge from the further lectures helped me to accomplish the development.
- **Q2**: I’m not used to python, so the syntax was a bit complicated. But NO-TDD is classical programming and implementing of code, so i felt familiar with it.
- **Q3**: easy to understand, very well structured and in knew exactly what to do.

Student_09 (NO-TDD)

- **Q1**: NO ANSWER
- **Q2**: Thats nice activities but maybe we can discuss about that because that’s not really clear and there are some unknown places in my mind.
- **Q3**: These activities help and improves us.

Task 2:

Student_01 (NO-TDD)

- **Q1**: I have coded first and then wrote the test.
- **Q2**: It was ok.
- **Q3**: It was not so hard because based on the exercises we have done before.
- **Q4**: TDD is the best one to avoid any errors in the future.

Student_02 (NO-TDD)

- **Q1**: No certain approach
- **Q2**: It was hard not to use TDD.
- **Q3**: It was okay.
- **Q4**: I prefer using TDD.

Student_03 (NO-TDD)

- **Q1:** Plain write code, test later.
- **Q2:** Actually, with TDD i wouldn't have run into some pitfalls. I had to rewrite some code because i didn't know i broke some legacy ones. With TDD i would have known way earlier.
- **Q3:** It was a brain full task, but still really good. Wouldn't it be funny to run the code on an actual roomba?
- **Q4:** TDD: It reduces bugs, refactoring breaking changes are known like instantly, the tests are like documentation. Non-TDD: Bugs and functional errors are only known at the end testing state, really bad. I would now prefer TDD. But only if the functional features 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.

Student_04 (NO-TDD)

- **Q1**: I implemented all the functions first and then wrote the various tests.
- **Q2**: I feel quite confident.
- **Q3**: I feel quite confident.
- **Q4**: With the NO-TDD approach, I seem to be able to better test the functions I implement.

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. Because for me it was impossible to do both at the same time, I tried to copy the code and so I don't know how to program well the TDD style. So today it was really hard for me.
- **Q2**: I think the Cleaning Robot was harder than the intelligent office, but maybe only because of TDD. Also, when you write TDD, you have to writ like two codes, the actual code and then the whole test code, so it takes more time.
- **Q3**: I think that I have a few mistakes in the non TDD code, but still I got everything, and the mistakes should be easy to fix. So, I prefer the non TDD programming, but maybe also because I'm used to it.

Student_06 (TDD)

- **Q1**: For me is more difficult to use this methodology. I prefer to create the code before testing it
- **Q2**: I had some difficult to understand the task, more complex.
- **Q3**: NO-TDD was simpler to apply, also the tasks were easier to implement. Anyway, i prefer a 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 bit harder than the last one and it required me more time.
- **Q3**: Applying TDD force me to think in a way I'm not used to do, 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 it is very useful and if you are used to it, it really helps you a lot to improve your programming
- **Q2**: NO ANSWER
- **Q3**: As i said in a further answer the thinking is upside down and it's kind of learning a new logical thinking. I still prefer the no-TDD because i like it more to write the logic.

Student_09 (TDD)

- **Q1**: I think I didn't get it, but I tried so hard.
- **Q2**: If i am still alive probably I am developing my skills.
- **Q3**: NO ANSWER

Thematic analysis – Replication

Template

1. **Replicated experiment**
    a) **General difficulty**
        - **Easy task (5)**
        - **Balanced/some difficulties (4)**
    b) **Prior knowledge of the sensors/actuators (3)**
    c) **Positive thoughts on hardware deployment (5)**
2. **Refactorings in TDD**
    a) **Some refactoring (4)**
    b) **No refactoring (1)**
3. **Test cases in NO-TDD**
    a) **No tests (2)**
    b) **Tested anyways (2)**
4. **Application of TDD for ESs**
    a) **Would use TDD going forward/ imo TDD is more suitable for ESs (5)**
    b) **Would use NO-TDD going forward/ imo NO-TDD is more suitable for ESs (2)**
    c) **Not sure/depends on the complexity (2)**
5. **Prior testing experience**
    a) **No prior testing experience (2)**
    b) **Low testing experience (only unit testing) (4)**
    c) **Some familiarity with TDD (3)**
6. **Seminars and exercises**
    a) **Easy to follow/overall good (6)**
    b) **Other**
        - **Too fast for some things / some difficulties (2)**
        - **Would have liked more explanation on TDD (1)**

Student_01 (NO-TDD)

1. For me development of this task was not very hard after the previous tasks. I also feel like at this point I am familiar with some of the libraries.
2. Still wrote tests afterwards. However, in my opinion there are downsides since you're not sure you're testing everything. Writing tests before would be better but also harder.
3. 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 of the 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)

1. Good feeling because it was in line the first one. Nice to see it run and test by yourself with hardware with the sensors.

2. TDD was very useful because it helps you understand what you are programming, and it will work for sure cause you have written the tests before. Only little refactoring performed for the temperature and motor story (4th) because I wanted to clean it a bit after the tests.
3. 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 ES development in my opinion.

Student_03 (TDD)

1. I knew most of the sensors so developing this task was not hard for me. But really cool to develop because I wanted to see it in action on real hardware.
2. Helpful to use TDD cause the test acted as documentation. You don't need to have a "master plan" for testing the whole system. You can rather focus on each phase at a time. Could help in TDD. Performed refactoring after implementing one story (3) broke something in a previous user story (2). Also, other minor refactorings to improve overall code readability.
3. Overall good experience with the lectures and seminars. It was good to see some testing approaches in practice besides theory. Going forward it 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 (NO-TDD)

1. I liked the task overall, but it was simple for me because I already used the sensors (Lab of IoT).
2. I had no issues using NO-TDD for this task, cause 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.
3. I liked the experience overall with a mixture of theory and practical stuff. I knew some TDD concepts from another course (Software Dependability). As for ES 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 (NO-TDD)

1. 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.
2. 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.
3. 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 ES TDD could be useful if you know more about what you're doing, but generally I would use NO-TDD going forward.

Student_06 (TDD)

1. Had some trouble with the implementation of the 4th task with the servo motor, but the task was manageable overall and clear to understand.

2. 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.
3. 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)

1. 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.
2. 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.
3. 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)

1. I felt quite comfortable about developing this task. It was nice to deploy it on hardware, afterwards. Overall, a pleasant experience.
2. Using TDD helped me with this task especially in user stories 2 and 3. For the others the was no big difference in my opinion. Did not perform any refactoring as they were not necessary in my opinion.
3. 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 (NO-TDD)

1. 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.
2. 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.
3. 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. with no-TDD it can be a bit more complex since adding the tests after can be more challenging. When I applied TDD I didn't find any particular difficulty, however it was a bit more challenging then no-TDD, but i think that if I learned more how to use TDD, it would be better.