Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di I livello in
Informatica

Test Driven Development for Embedded Systems

Relatore
Giuseppe Scanniello
Correlatore
Dott. Nome Cognome

Candidato
Michelangelo Esposito

Academic Year 2021-2022

Abstract

This thesis investigates the applicability of Test-Driven Development (TDD) for embedded systems development. Embedded systems are computer systems that perform a specific function and are integrated into larger products; these systems have unique characteristics, such as real-time constraints and resource limitations, which can make traditional software development and testing approaches difficult to apply. TDD is a software development methodology that emphasizes writing automated tests before writing code: the research in this thesis evaluates the feasibility and effectiveness of using TDD for embedded systems development through a case study involving .... The results of the study indicate that TDD can be successfully applied to embedded systems development, and can lead to more robust and maintainable code. The thesis also provides insights into the challenges that must be overcome when using TDD for embedded systems development and suggests best practices for addressing those challenges.

Contents

List of Figures

List of Acronyms and Abbreviations

Chapter 1

Introduction

This work of thesis is made up of five chapters:

Chapter one contains an overview on the software testing process, analyzing the main approaches, with a focus of Test-Driven Development.

Chapter two will provide information on the general embedded systems concepts, before discussing the techniques for testing such systems.

In chapter three, we conduct a

Chapter 2

Test Driven Development

2.1 Overview on software testing

Software testing is an essential part of the development process of a system, as it helps to ensure that a piece of software is reliable and performs as intended; it can be defined as the process of finding differences between the expected behavior specified by system's requirements and models, and the observed behavior of the implemented software. Unfortunately, it is impossible to completely test a nontrivial system. First, testing is not decidable. Second, testing must be performed under time and budget constraints [1], therefore 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, and they are often used in combination to ensure that a software product is thoroughly tested and conform to its specification.

- **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. Unit tests are typically written by the developers who wrote the 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 tests.

- **Integration Testing** is a method of testing how different units or components of a software product work together. The goal of inte-

gration 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

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. It includes the following phases:

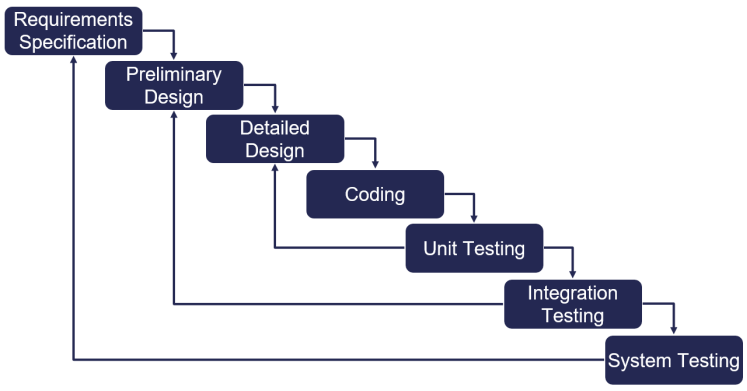


Figure 2.1: The Waterfall model

The Waterfall model is a linear, sequential approach to software development in which development is divided into distinct 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 if 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, which is often not the case in modern software development. 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.

Modern software development strays away from non-incremental models, as today's applications are continuously evolving and adapting; instead, iterative approaches are preferred ...

A key example is the Agile development process, 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:

Agile phases:

- **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 and stakeholders to ensure that the software meets their 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 rapid feedback on the quality of the code.

- **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.

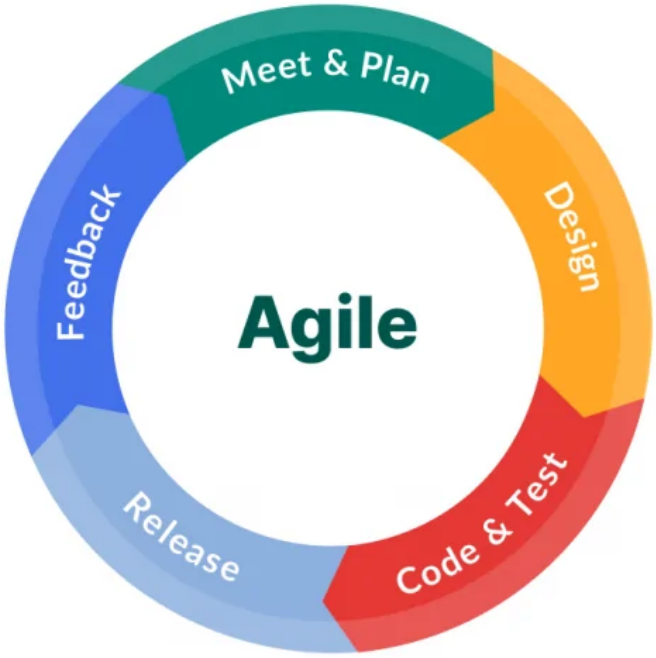


Figure 2.2: The Agile model

- **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

The concept of Test Driven Development (TDD) was firstly introduced in 2003 by Kent Back in the book "Test Driven Development By Example"

[2]. While there is no formal definition of the process, as the author states, the goal is to "write clean code that works". Compared to traditional SDL processes, TDD is an extremely short, incremental, and repetitive process, and is related to **test-first programming** concepts in agile development and extreme programming; this advocates for frequent updates/releases for the software, in short cycles, while encouraging code reviews, unit testing and incremental addition of features.

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

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

Each new unit of code requires a repetition of this cycle [3].

The figure below provides a representation of the TDD 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 [4]. 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 [4].

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

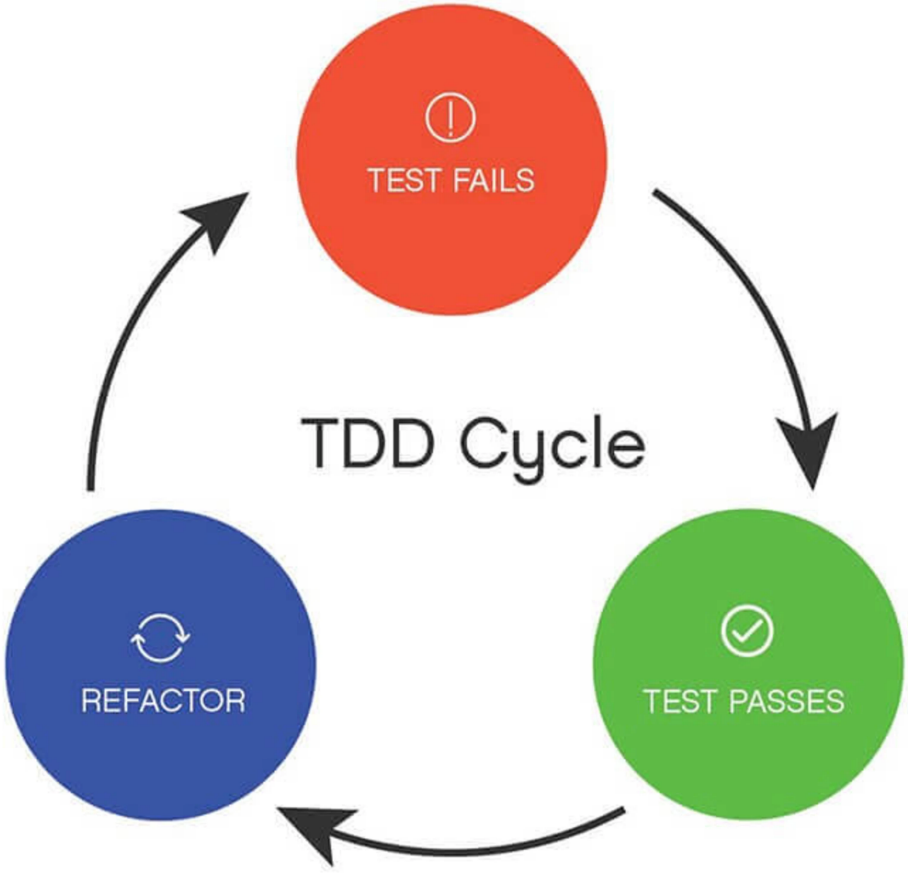


Figure 2.3: The Test Driven Development cycle

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

Testing Embedded Systems

3.1 Overview on Embedded Systems

Embedded Systems (ES) can be defined as a combination of hardware components and software systems that seamlessly work together to achieve a specific purpose; such systems can be dynamically programmed or have a fixed functionality set, and are often engineered to achieve their goal within a larger system. They are often found in 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 often required to work continuously without the need for human intervention.

Often enough, ES must operate in a stand-alone manner: many of these systems are used in environments where there is no access to a network or the internet, so the ability to function independently is essential; this requires the use of embedded software, which is specifically designed to run on the limited hardware of the system.

In recent years, such system have seen a steep surge in popularity, and have driven innovation forward in their respective areas of deployment: everywhere, spanning from the agricultural field, to the medical and energy ones, ES 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 ES will only become more prominent.

There are many different types of embedded systems, 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.

General purpose hardware, Arduino, Raspberry, etc...

3.1.1 Challenges

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

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

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

3.1.2 Enabling technologies

One of the key enabling technologies for ES is their microcontroller, which is a small, single-chip computer that is used to manage the functions of the larger system. Microcontrollers are often used in embedded systems because they are cheap, low-power. While some applications require their own custom-made microcontroller and hardware, there are a variety of general purpose boards that are far easier to program and can also be customized for a variety of applications, which makes them highly versatile.

One type of general-purpose microcontroller that is commonly used for ES development, as well as many other IoT applications, 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. Another popular general-purpose microcontroller platform is the Raspberry Pi, which is a small single-board computer that is based on the ARM architecture, which is also what many mobile processors are based on.

In addition to these general-purpose microcontrollers, there are also many proprietary microcontrollers that are designed for specific applications: these microcontrollers are often customized for a particular task and may not be easily programmable 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 the design of these microcontrollers may be kept confidential to protect against tampering or reverse engineering.

Overall, the choice of microcontroller for an ES will depend on the specific requirements of the application. 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.

Software-wise, more complex system can be equipped with their own **Embedded Operating System**, or more specifically, their **Real-Time Operating System** (RTOS); these are specialized operating systems that are designed to provide a predictable response time to events, even when there are many tasks running concurrently. RTOS are essential for embedded systems that require fast and reliable performance, such as in aircraft control systems or medical devices.

Often, multiple devices are deployed as part of the same larger system and must be able to efficiently communicate between one another to achieve their purpose; therefore, it is essential for ES to ... a robust suite of protocols. As always, determine 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 that is designed for low-power, low-data-rate applications [5]; 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. One of its key advantages is the low power consumption, which makes it well-suited for use in devices that need to operate for long periods of time without access to a power source.

Bluetooth is another wireless protocol that is commonly used in embedded systems 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.

Sigfox on the other hand, is a low-power, wide-area (LPWA) communication protocol that is designed for Internet of Things (IoT) applications; it is used to transmit small amounts of data over long distances, making it 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. It is used to transmit data between devices and is a key component of the internet. IP is often used in embedded systems that need to communicate with other devices or with the internet, such as in smart home systems or industrial control systems.

Overall, each of these protocols has its own strengths and is well-suited for different types of applications. Zigbee is ideal for low-power, low-data-rate applications, while Bluetooth is well-suited for medium-range communication. Sigfox is ideal for long-range communication in IoT applications, and IP is widely used for communication over networks.

3.1.3 Design

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

Failures in ES should always be evident and identifiable quickly (a heart monitor should not fail quietly [6]). Given the high criticality of such systems, ensuring their dependability over the course of their lifespan is essential; ES can be deployed in extreme conditions (i.e., weather monitoring in extreme locations of the planet, devices inside the human body, or ...), where maintenance operations cannot be performed regularly, and high availability is expected.

The dependability of a system can be expressed in terms of:

- **System maintainability**: the extent to which a system can be

adapted/modified to accommodate new change requests. As ES become 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.

- **System reliability**: the extent to which a system is reliable with respect to the expected behavior.
- **System availability**: the extent to which a system remains available for its users.
- **System security**: the extent to which a system can keep data of its users safe and ensures the safety of their users.

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

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

3.2 Testing Embedded Systems

Testing ES poses a series of challenges compared to traditional testing: first of all, in the case of ES that are highly integrated with a physical environment (such as with CPSs), replicating the exact conditions in which the hardware will be deployed may be challenging. Secondly, field-testing of these systems can be unfeasible to dangerous or impractical environmental conditions (i.e., a nuclear power plant, a deep-ocean station, or the human body). Furthermore, given the absence of a user interface in most cases, testing such systems can be particularly challenging, given the lack of immediate feedback. 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; similarly, system with tight hardware constraints, such as memory, limited processor power, or power consumption are fifficult to design and test.

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

3.2.1 X-in-the-loop

For these reasons and more, the general testing process of ES follows the X-in-the-loop paradigm [7] where the system goes through a series of step that simulate its behavior with an increased level of detail before being actually deployed; subcategories in this area include Model-in-the-Loop, Software-in-the-Loop, Processor-in-the-Loop, Hardware-in-the-Loop, and System-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 [8]. 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 sued in the later stages of verification.

- 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, 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 into the acceptance threshold, we can move to the next phase.

- The next step is **Processor-in-the-Loop (PIL)**; here, an embedded processor will be simulated in detail and used to run the controller code in a closed-loop simulation. This help 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,

- Finally, **Hardware-in-the-loop** is the last step performed before deploying the ES to the actual hardware. Here, we can run the simu-

lated system on a real-time environment, such as SpeedGoat [9]. The real-time system performs deterministic simulations and has physical connections to the embedded processor, i.e., analog inputs and outpus, 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, and it is required by automotive and aerospace validation standards.

After these steps, the system can finally be deployed on real hardware.

A common environment for performing the simulation steps discussed above is Simulink [10]; 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.

Chapter 4

Literature Review

4.1 Studies on Test Driven Development

The Empirical Software Engineering community has taken interest into the investigation of the effects of TDD on several efforts, including testing efforts, external software quality and developers productivity [11] [12] [13].

However, the empirical evidence so far has been mixed regarding the effects of TDD:

4.2 TDD for Embedded Systems

Often, quality in embedded software is generally tied to platform-specific testing tools geared towards debugging [14]

TDD benefits for ES: Embedded software has all the challenges of "regular" software, such as poor quality and unreliable schedules, but adds challenges of its own. But this doesn't mean that TDD can't work for embedded. Report erratum this copy is (P1.0 printing, April, 2011) B ENEFITS FOR E MBEDDED 34 The problem most cited by embedded developers is that embedded code depends on the hardware. Dependencies are a huge problem for nonembedded code too. Thankfully, there are solutions for managing dependencies. In principle, there is no difference between a dependency on a hardware device and one on a database. There are challenges that embedded developers face, and we'll explore how to use TDD to your advantage. The embedded developer can expect the same benefits described in the previous section that nonembedded developers enjoy, plus a few bonus benefits specific to embedded:

- 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 [15], 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**: there is a risk that the compiled code will 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 [15], 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 the same as stage 3 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 5

Case Study

In this chapter we will present in detail the planning and approach we followed to establish our study.

the two experimental tasks on which this study is based on, as well as the analysis of the gathered results.

The study was conducted with the participation of 9 undergraduate master's degree students enrolled in the "Embedded Systems" course at the University of Salerno in Italy. The participation voluntarily accepted to take part in this study.

5.1 Research Questions

The study we performed aimed at answering the following Research Questions (RQ):

- **RQ1**:
- **RQ2**: Are there differences between TDD and NO-TDD in the external quality of the implemented solutions, developers' productivity, and number of tests written?
- **RQ3**: Are there differences between TDD and NO-TDD in (cyclomatic, cognitive and code smells)

5.2 Experimental Units

The participants for this study were students at the University of Salerno, in Italy, they were a mix of second-year master's degree students at the

University of Salerno and students visiting with the Erasmus program; this last group was further made up of master's degree student and third-year bachelor's degree students. Both groups were enrolled in the Embedded Systems course at the University of Salerno.

he course covered the following topics, software quality, unit testing, integration testing, SOLID principles, refactor- ing, iterative test-last development, big-bang testing, and TDD. The course included frontal lectures, laboratory sessions, and home- work. During the laboratory sessions, the students improved their knowledge about how to develop unit tests in Java by using the Eclipse IDE and JUnit, and the refactoring functionality available in Eclipse. During laboratory sessions and by developing homework, the students practiced unit testing, iterative test-last development, big-bang testing, and TDD

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

The participants for the tasks were asked to carry out their assignment by either using TDD or no-TDD (i.e., any approach they preferred, except for TDD) depending on the group they were partitioned in. and on the period the task took place.

5.3 Experimental Materials

The experimental objects were three code katas, programming exercises aimed at practicing a technique or a programming language:

- **IntelligentOffice**:
- **CleaningRobot**:
- **SmartHome**:

5.4 Tasks

5.5 Variables

Independent variables:

- **Technique**: a nominal variable assuming two values, TDD and no-TDD.
- **Period**: since the study is longitudinal - i.e., the data was collected over time - we took into account the period during which each treatment (TDD or no-TDD) was applied.
- **Group**: variable representing the two groups. It can assume the values G1 and G2.

Dependent variables:

- **Quality (QLTY)**: quantifies the external quality of the solution a participant implemented. This variable is 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, we checked the asserts in the test suite corresponding to the story. Namely, if at least one assert in the test suite for the story passes, than the story was considered as tackled. $\#TUS$ is formally defined as follows:

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

On the other hand, 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:

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

- **Productivity (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: a value close to 0 indicates low productivity in the implemented solution, while a value close to 1 refers to high productivity.

- **Number of tests (TEST)**: 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 ∞.

Furthermore, we considered four additional dependent variables regarding the general code quality in the submitted projects were collected:

- **Number of smells (SMELL)**:
- **Cyclomatic Complexity (CYC)**:
- **Cognitive Complexity (COG)**:
- **Lines of code (LOC)**: the number of lines of code written by the participant in both the production code and the test code.

5.6 Study design

The participants were randomly split into two groups, G1 and G2, having 4 and 5 participants respectively. For the first period P1, the group G1 was assigned the TDD task, while the group G2 was assigned the no-TDD task; on the other hand, during period P2, the group G1 was assigned the no-TDD task, while the group G2 was assigned the TDD task. Therefore, the design of our study can be classified as a repeated-measures, within-subjects design. In each period, the participants in G1 and G2 dealt with different experimental objects. For instance, in P1, the participants in G1 dealt with BSK, while those in G2 with MRA. At the end of the study, every participant had tackled each experimental object only once.

5.7 Procedure

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

The Embedded Systems course, during which the study was conducted, started in September 2022. The first task, P1, took place on Tuesday,

December 6th 2022, while the second, P2, took place on Tuesday, December 13th 2022.

Between the start of the course and P1, the participants had never dealt with TDD, while they were somewhat familiar with unit testing and iterative test-last development. In the weeks before P1, TDD was introduced to the participants via some lectures; also had taken part in ... training sessions on TDD and completed some homework by using this development practice.

5.8 Analysis Procedure

The gathered experimental data were analyzed according to the following procedure:

1. **Descriptive Statistics**: in order to provide an overview of the distributions of the dependent variables, we calculated a set of summary statistics, including the mean, median, and standard deviation (SD). Additionally, we employed box plots to graphically summarize these distributions.

5.9 Results

In this section we will report the values observed for each dependent variable. 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.

Box plot definitions:

- **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 shows that many students have similar views at certain parts of the scale, but in other parts of the scale students are more variable in their views. The long upper whisker in the example 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 will be close to the average) are all at the same level. However, the box plots in these examples show very different distributions of views. It's always important to consider the pattern of the whole distribution of responses in a box plot.

Task 1: The following table display the values for the variables measured for the first experimental task, IntelligentOffice, for the two groups, in 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

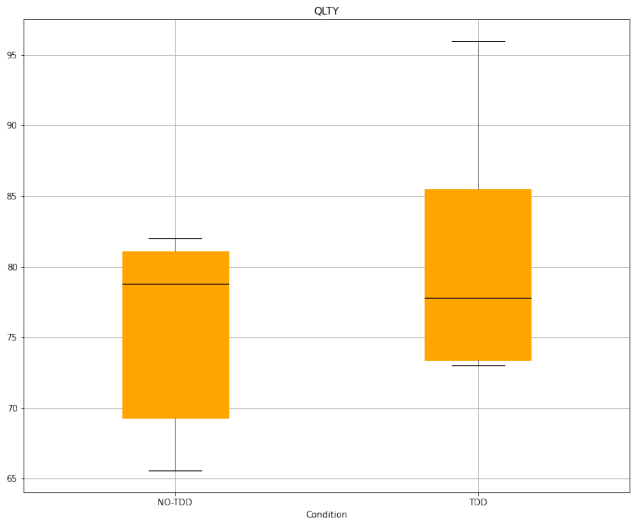


Figure 5.1: Box plot for $QLTY$ in task 1

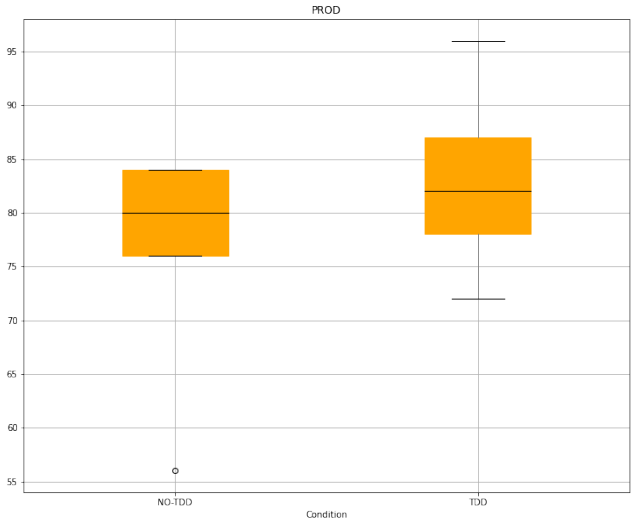


Figure 5.2: Box plot for *PROD* in task 1

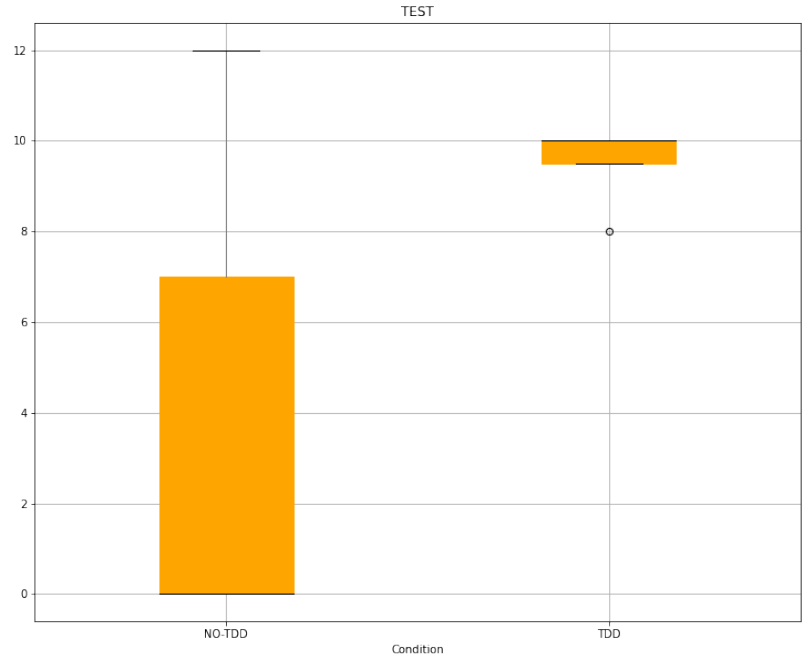


Figure 5.3: Box plot for $TEST$ in task 1

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

Tasks 1 & 2:

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

5.10 Discussion

Chapter 6

Conclusions

Bibliography

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

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

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

[4] 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. DOI: 10.1109/TSE.2019.2920377. URL: https://doi.org/10.1109/TSE.2019.2920377.

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

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

[7] Vahid Garousi et al. "What We Know about Testing Embedded Software". In: *IEEE Softw.* 35.4 (2018), pp. 62–69. DOI: 10.1109/MS.2018.2801541. URL: https://doi.org/10.1109/MS.2018.2801541.

[8] *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.

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

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

[11] 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. DOI: 10.1145/2961111.2962592. URL: https://doi.org/10.1145/2961111.2962592.

[12] 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. DOI: 10.1109/TSE.2005.37. URL: https://doi.org/10.1109/TSE.2005.37.

[13] 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. DOI: 10.1016/j.infsof.2009.08.007. URL: https://doi.org/10.1016/j.infsof.2009.08.007.

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

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

[16] A. Author and A. Author. *Book reference example*. Publisher, 2099.

[17] A. Author. “Article title”. In: *Journal name* (2099).

[18] *Example*. URL: https://www.isislab.it.

[19] A. Author. “Tesi di esempio ISISLab”. 2099.

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.

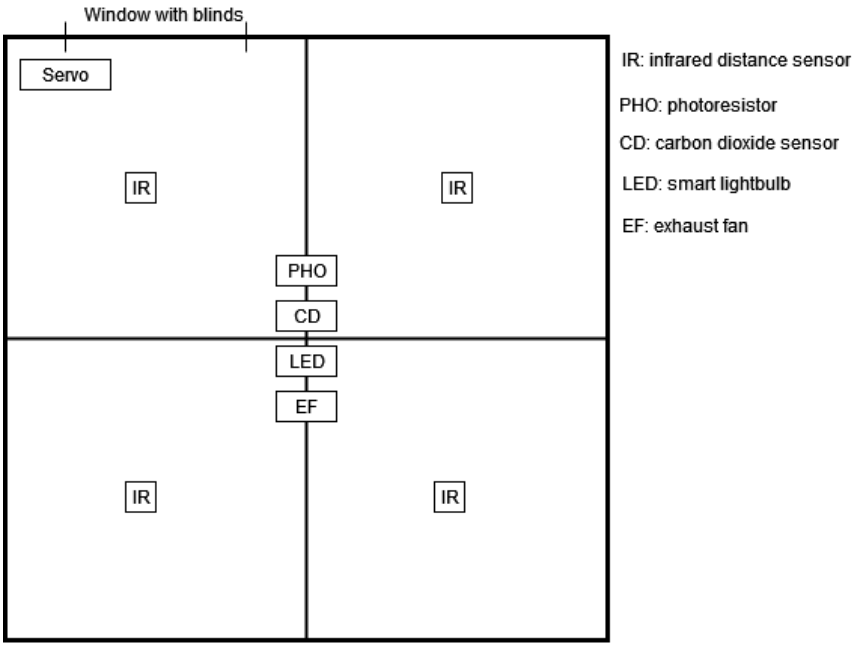
To recap, the following sensors and actuators are present:

- 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.