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

...

The purpose of this thesis is to analyze the benefits and/or drawbacks derived from the application of Test Driven Development (TDD) as part of the software development lifecycle of Embedded Systems.

Contents

List of Figures

List of Acronyms and Abbreviations

Chapter 1

Introduction

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:

The main weakness of the Waterfall model is the very long feedback cycle between requirements specification and system testing; during this phase, the client is absent and there is no deliverable product before the end of the process

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

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,

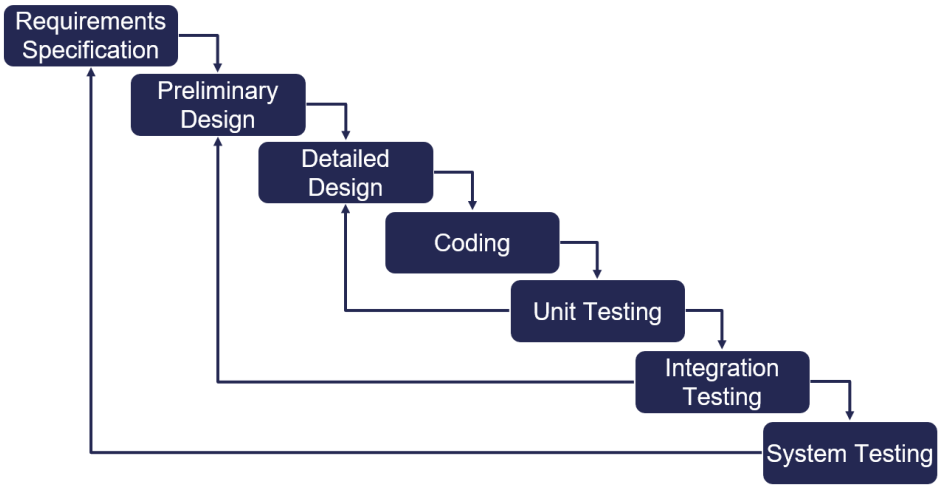


Figure 2.1: The Waterfall model

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 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

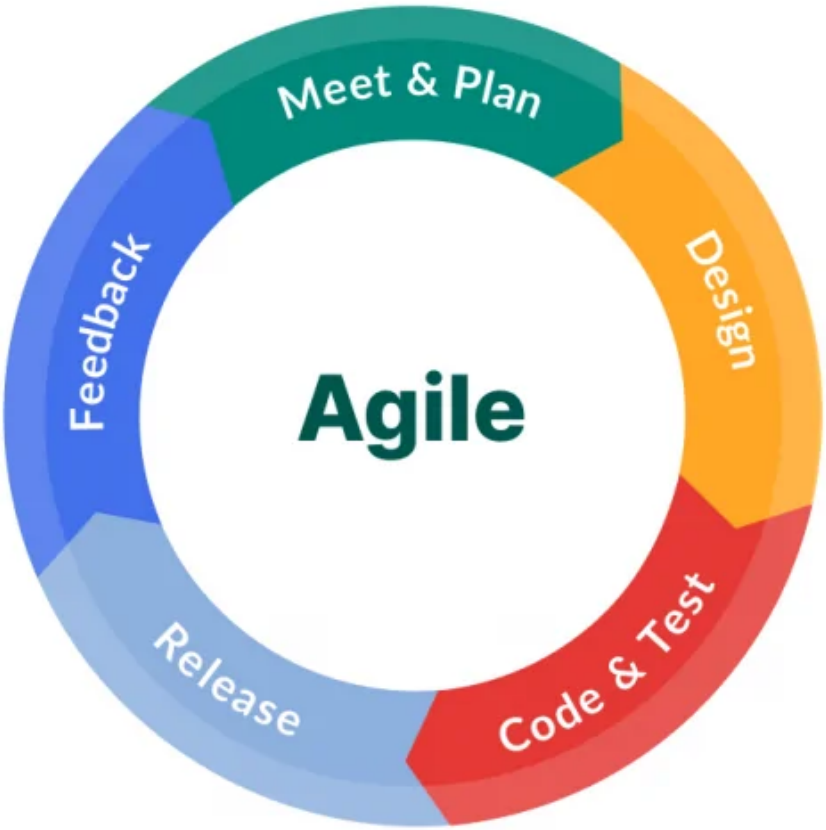


Figure 2.2: The Agile model

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

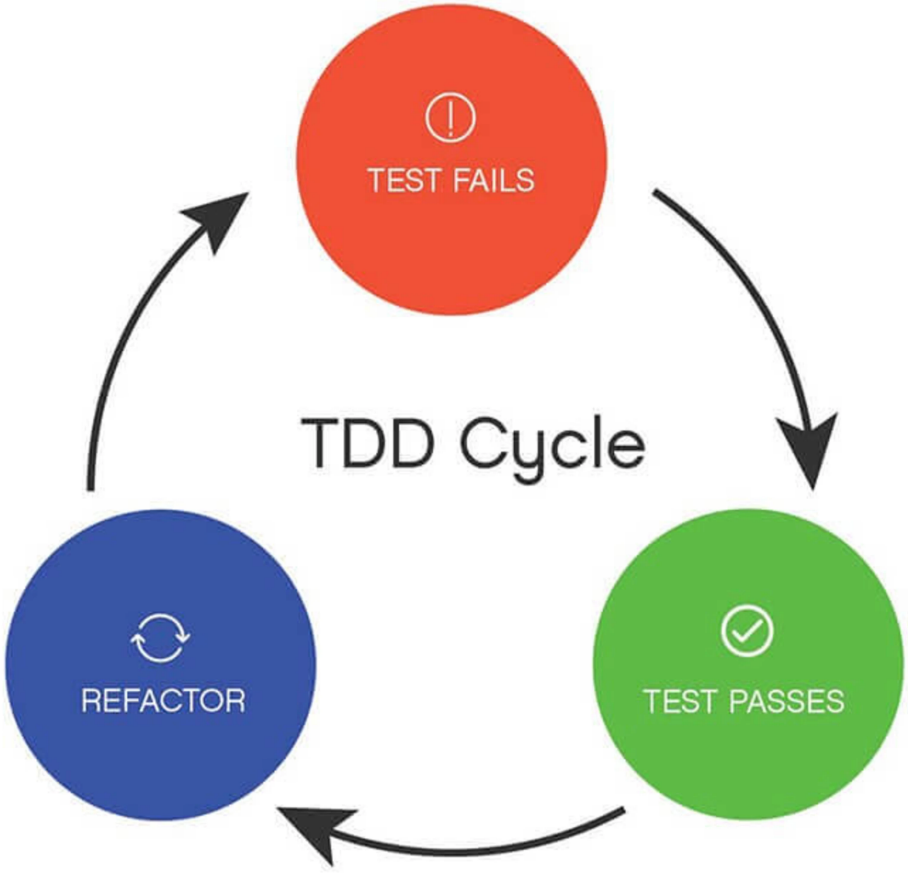


Figure 2.3: The Test Driven Development cycle

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

2.2.2 TDD advantages

The employment of TDD can result in a series of benefits during the development process, sych 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 Embedded Systems Overview

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 a domain-specific, often critical, goal. In recent years, such system have seen a 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.
Due to their high specialization, ES often deal with time and resource constraints, both in hardware and in software; often these systems are battery-powered and therefore 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.
Failures in ES should always be evident and identifiable quickly (a heart monitor should not fail quietly [5]). 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.

- **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.

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 [6] 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 [7]. 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 simulated system on a real-time environment, such as SpeedGoat [8]. 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 [9]; 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

Chapter 5

Case Study

5.1 Overview

In this section we will present in detail the approach we followed to establish 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.

5.2 Research Questions

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

- **RQ1**:
- **RQ2**:
- **RQ3**:

5.3 Experimental tasks

The following table contains a summary of the user stories for the two experimental tasks on which the study was conducted:

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

[6] 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.

[7] *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.

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

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

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

[11] A. Author. "Article title". In: *Journal name* (2099).

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

[13] 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.

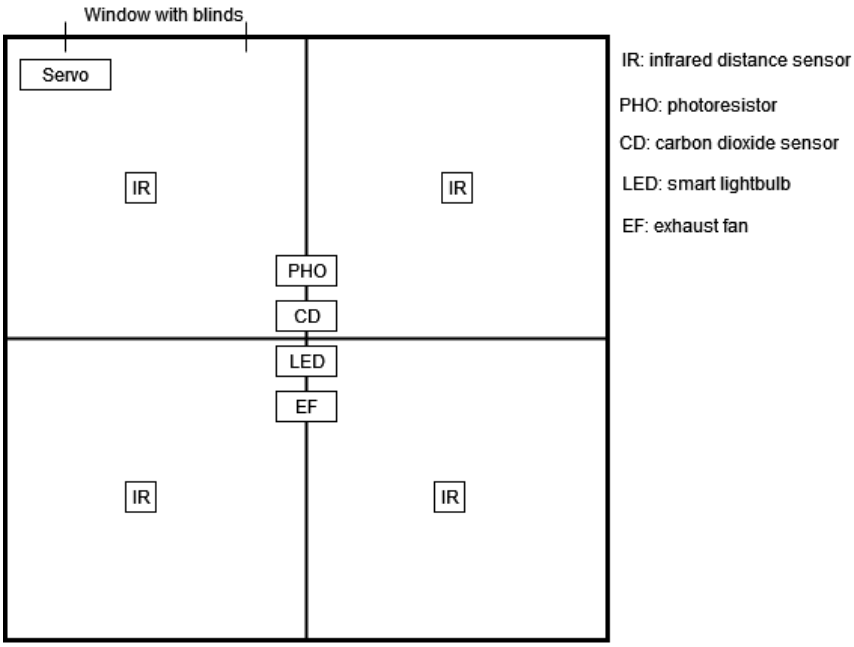
To recap, the following sensors and actuators are present:

- 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.