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 2

Test Driven Development

2.1 Introduction

Testing 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.

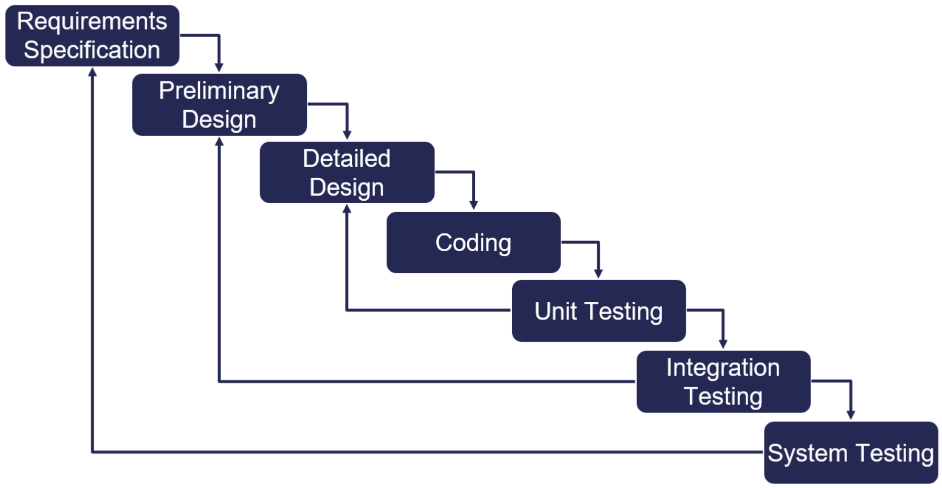

Figure 2.1: The Waterfall model

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

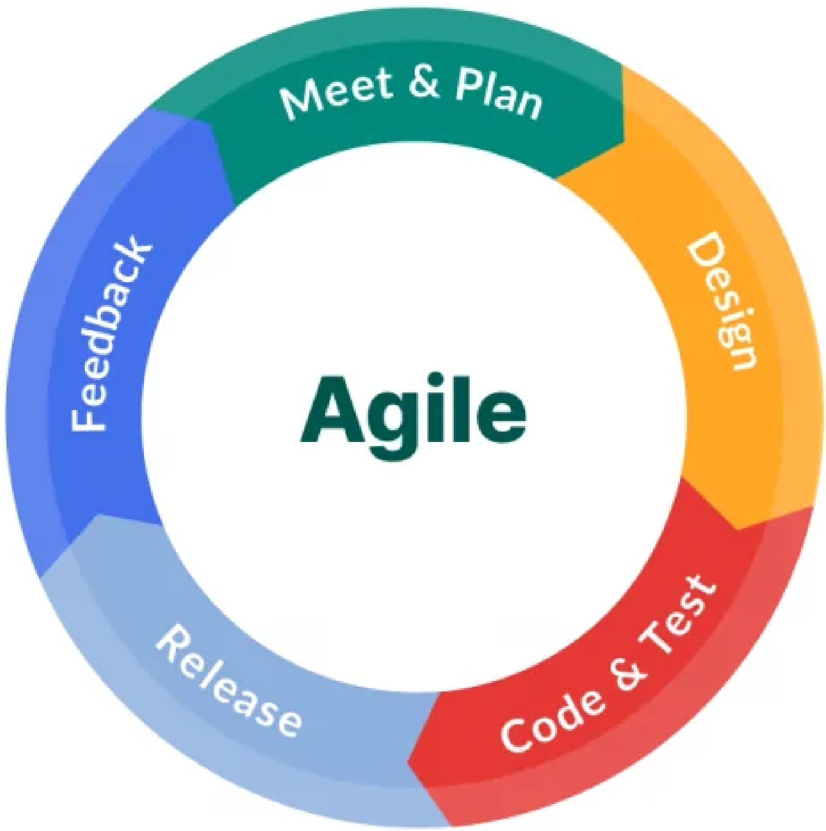

Figure 2.2: The Agile model

2.2 Test Driven Development

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 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

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

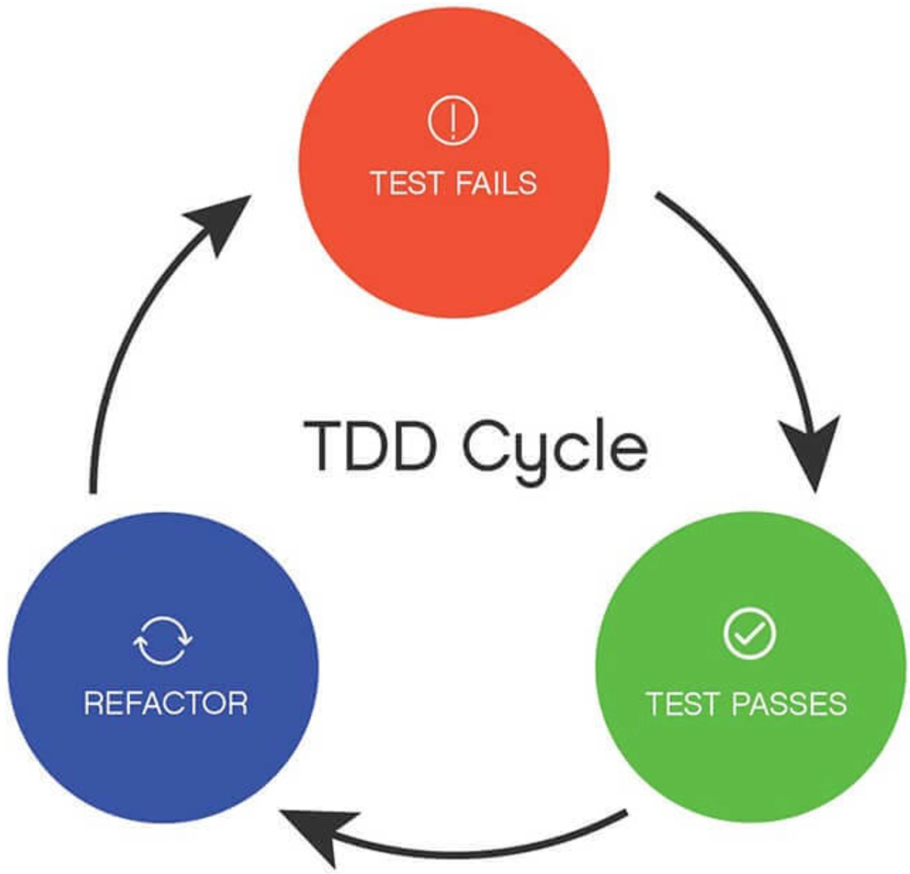

Figure 2.3: The Test Driven Development cycle

- **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.

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.