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

Problem formulation

2.1 Unit Testing

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" [1]. 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 [2].

The figure below provides a representation of the TDD cycle:

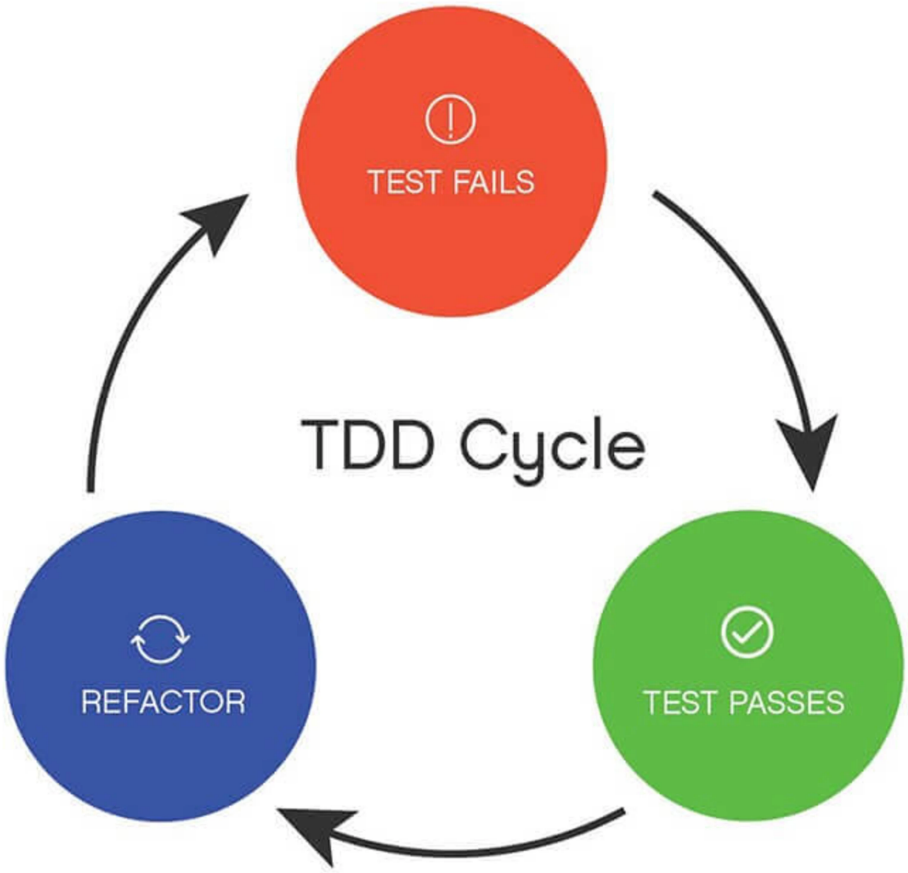

Figure 2.1: The Test Driven Development cycle

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

User stories can vary in granularity

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:

- **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.

2.3 Testing 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 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 [3]). 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.

Furthermore, given the absence of a user interface in most cases, testing such systems can be particularly challenging, given the lack of immediate feedback.

2.4 Test Driven Development for Embedded Systems

Generally, the testing process of ES follows the X-in-the-loop paradigm [4]; 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. Reference the old survey papers (i.e. X in the loop) that provide a summary of the main techniques

Chapter 3

Literature

Chapter 4

Conclusions

Bibliography

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

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

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

[4] 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.

[5] Ross B. Girshick et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 2014, pp. 580–587. DOI: 10.1109/CVPR.2014.81. URL: https://doi.org/10.1109/CVPR.2014.81.

[6] Jindi Zhang et al. "Evaluating Adversarial Attacks on Driving Safety in Vision-Based Autonomous Vehicles". In: *IEEE Internet Things J.* 9.5 (2022), pp. 3443–3456. DOI: 10.1109/JIOT.2021.3099164. URL: https://doi.org/10.1109/JIOT.2021.3099164.

[7] Edmund K. Burke and Graham Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer US, 2014, pp. 403–449.

[8] *Check: framework for unit testing in C*. URL: https://libcheck.github.io/check/.

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

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

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

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