./

TDLC -Learning Report

ggGgg

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver.Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **To be**  **Approved By** | **Remarks/Revision Details** |
| 1 | 02/11/2020 | Mohammed Sadiq Afreed Meeran |  | Pagala Prithvi Sekhar |  |

**Document History**

**Contents**

[List of Tables 3](#__RefHeading___Toc5600_4123987002)

[Activity 1: TDLC, TDD and Application of TDLC & TDD on Miniproject 4](#__RefHeading___Toc5786_4123987002)

**Table of Figures**

Individual Activity 1

[Figure 1: Flowchart diagram](#_Toc52171470) 8

Figure 2: Class Diagram……………………………………………………………………………………………...9

[Figure 3: Low level requirement use case diagram](#_Toc52171471) 10

[Figure 4: High level requirement use case diagram](#_Toc52171472) 11

[Figure 5: Low level sequence diagram 1](#_Toc52171473)2

[Figure 6: High level sequence diagram 1](#_Toc52171474)3

Individual Activity 2

[Figure](#_Toc52171470) 7[:](#_Toc52171470) Class Diagram18

Figure 8: Component Diagram…..……………………………………………………………………….…...…...19

[Figure 9:](#_Toc52171471) High level Use case diagram19

[Figure](#_Toc52171472) 10[: Low level Use case diagram](#_Toc52171472) 20

[Figure](#_Toc52171473) 11[: Low level Sequence diagram](#_Toc52171473) 21

[Figure](#_Toc52171474) 12[:](#_Toc52171474) CI/CD workflows for C++ Mini Project22

Figure 13: Opening an issue……………………………………………………………………………………………...……22

Figure 14: Making Change,Commit and Closing issue………………………………………………...…………...23

Team Activity 2

Figure 15: Structural Diagram………………………………………………………………………………………...……….31

Figure 16: Behavioral Diagram………………...………………………………………………………………..…………...33

# List of Tables

Individual Acrtivity 1

[Table 1: Costing table](#_Toc52134959) 6

[Table 2: High level requirement test plan](#_Toc52134960) 7

[Table 3: Low level requirement test plan](#_Toc52134961) 8

[Table 4: High level requirement test plan](#_Toc52134962) 15

[Table 5: Low level Requirements](#_Toc52134963) 16

Individual Activty 2

[Table](#_Toc52134960) 6[: High level requirement test plan](#_Toc52134960) 18

[Table](#_Toc52134961) 7[: Low level requirement test plan](#_Toc52134961) 18

[Table 8:](#_Toc52134962) Unit Testing25

[Table 9:](#_Toc52134963) Integration Testing25

# **Team Activity** 1: **TDLC, TDD and Application of TDLC & TDD on Miniproject**

**Test Driven Development:**

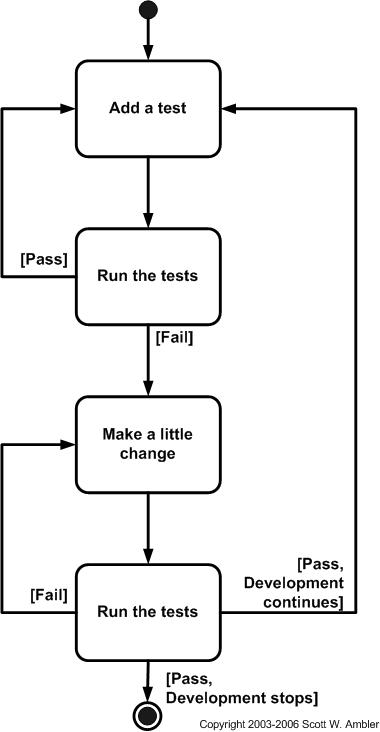
**What is TDD?**

* Test-driven development (TDD), is an evolutionary approach to development

which combines test-first development where you write a test before you write just enough production

code to fulfill that test and [refactoring](http://www.agiledata.org/essays/databaseRefactoring.html).

* Test driven development is an iterative development process. In TDD, developers write a test before they write just enough production code to fulfill that test and the subsequent refactoring. Developers use the specifications and first write test describing how the code should behave. It is a rapid cycle of testing, coding, and refactoring.
* The steps of test first development (TFD) are overviewed in the UML Activity Diagram of [Figure 1](http://agiledata.org/essays/tdd.html" \l "Figure1StepsOfTDD). The first step is to quickly add a test, basically just enough code to fail. Next you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may first need to refactor any duplication out of your design as needed, turning TFD into TDD).



**Why Test Driven Development?**

A significant advantage of TDD is that it enables you to take small steps when writing software. It is far more productive than attempting to code in large steps. For example, assume you add some new functional code, compile, and test it. Chances are pretty good that your tests will be broken by defects that exist in the new code. It is much easier to find, and then fix, those defects if you've written two new lines of code than two thousand. The implication is that the faster your compiler and regression test suite, the more attractive it is to proceed in smaller and smaller steps. I generally prefer to add a few new lines of functional code, typically less than ten, before I recompile and rerun my tests.

TDD is a must in order to avoid discovering the problem during build stage, instead discover it at the initial stage of the process.

* Improves productivity by

- Minimizing time spent debugging

- educes the need for manual (monkey) checking by developers and tester

- helping developers to maintain focus

- reduce wastage: hand overs

* Improves communication
* Creating living, up-to-date specification
* Communicate design decisions
* Learning: listen to your code
* Baby steps: slow down and think
* Improves confidence
* Testable code by design + safety net
* Loosely-coupled design
* Refactoring

**How do you use TDD?**

1. Write a Test. Since development is driven by tests, the obvious first step is to create a new test.
2. Confirm the Test Fails. Once the test is created, the next step is to confirm that the test fails.
3. Write Code to Pass Test.
4. Confirm the Test Passes.
5. Refactor.
6. Repeat All Steps.

