**MEKELLE UNIVERSITY**



**Ethiopian institute of technology – Mekelle**

**Department of computing**

**Software engineering**

**Software tools and practices**

**Test Driven Development**

Group Members

|  |  |  |
| --- | --- | --- |
| No. | Name | ID |
| 1 | LEWAM YEMANE | EIMT/UR158539/11 |
| 2 | MEAREG TEFERI | EITM/UR158176/11 |
| 3 | NAHOM GIDEY | EITM/UR159379/11 |
| 4 | TEDROS G/HER | EITM/UR157777 /11 |
| 5 | ZEHARA KASSAW | EITM/UR158553/11 |

**Test-Driven Development (TDD)**

**Introduction**

Test-driven development (TDD) is a way of writing code that involves the following steps:

1. Writing an automated unit-level test case that fails.

2. Writing just enough code to make the test pass.

3. Refactoring both the test code and the production code.

4. Repeating the process with another new test case.

This is in contrast with alternative approaches, such as writing all the production code before starting on the test code, or writing all the test code before starting on the production code. With TDD, both are written together.

TDD is related to the test-first programming concepts of extreme programming, which began in 1999, but has more recently gained more general interest in its own right. Programmers also apply the concept to improving and debugging legacy code developed with older techniques.

**History**

Software engineer Kent Beck, who is credited with having developed or "rediscovered" the technique, stated in 2003 that TDD encourages simple designs and inspires confidence. Beck explains that the original description of TDD was in an ancient book about programming, where you would manually type in the expected output tape and then program until the actual output matched the expected output. After Beck had written the first xUnit framework in Smalltalk, he remembered reading this and tried it out, which was the origin of TDD for him.

**Coding Cycle**

The TDD steps are generally as follows:

1. List scenarios for the new feature.

2. Write a test for an item on the list.

3. Run all tests. The new test should fail for expected reasons.

4. Write the simplest code that passes the new test.

5. All tests should now pass.

6. Refactor as needed while ensuring all tests continue to pass.

7. Repeat the process, starting at step 2, with each test on the list until all tests are implemented and passing.

**Test-Driven Work**

TDD has been adopted outside of software development, in both product and service teams, as test-driven work. The six steps of the TDD sequence are applied with minor semantic changes:

1. "Add a check" replaces "Add a test".

2. "Run all checks" replaces "Run all tests".

3. "Do the work" replaces "Write some code".

4. "Run all checks" replaces "Run tests".

5. "Clean up the work" replaces "Refactor code".

6. "Repeat".

**Development Style**

There are various aspects to using test-driven development, such as the principles of "keep it simple, stupid" (KISS) and "You aren't gonna need it" (YAGNI). Writing the tests first, ensuring that each test case fails initially, and maintaining code visibility are also important practices in TDD.

**Fakes, Mocks, and Integration Tests**

Unit tests should never cross process boundaries or introduce dependencies on external modules or data. Whenever external access is needed in the final design, an interface should be defined, and the interface should be implemented in two ways: one that really accesses the external process, and one that is a fake or mock. Integration tests that alter any persistent store or database should be designed carefully with consideration of the initial and final state.

**Best Practices**

Best practices for TDD include:

1. Effective layout of a test case (setup, execution, validation, cleanup).

2. Separating common setup and teardown logic into test support services.

3. Keeping each test oracle focused on only the necessary results.

4. Designing time-related tests to allow tolerance for execution in non-real time operating systems.

Practices to avoid, or "anti-patterns", include:

1. Depending on system state manipulated from previously executed test cases.

2. Dependencies between test cases.

3. Interdependent tests.

4. Testing precise execution, behavior, timing, or performance.

5. Building "all-knowing oracles".

6. Testing implementation details.

7. Slow running tests.

**Comparison and Demarcation**

TDD is different from acceptance test–driven development (ATDD) in that TDD is primarily a developer's tool to help create well-written units of code, while ATDD is a communication tool between the customer, developer, and tester to ensure that the requirements are well-defined.

BDD (behavior-driven development) combines practices from TDD and ATDD, focusing on tests that describe behavior rather than tests that test a unit of implementation.

**Software for TDD**

There are many testing frameworks and tools that are useful in TDD, such as xUnit frameworks and the language-agnostic Test Anything Protocol (TAP) for test results.

**Advantages and Disadvantages of TDD**

**Advantages**:

- Comprehensive Test Coverage

- Enhanced Confidence in Code

- Well-Documented Code

- Requirement Clarity

- Facilitates Continuous Integration

- Boosts Productivity

- Reinforces Code Mental Model

- Emphasis on Design and Functionality

- Reduces Need for Debugging

- System Stability

**Disadvantages**:

- Increased Code Volume

- False Security from Tests

- Maintenance Overheads

- Time-Consuming Test Processes

- Testing Environment Set-Up

- Learning Curve

- Overcomplication

- Neglect of Overall Design

- Increased Costs

**Benefits**

TDD offers benefits such as increased programmer confidence, reduced fear of change and stress, improved focus, and a sense of achievement and job satisfaction. It can also lead to more modularized, flexible, and extensible code, as well as lower coupling between objects and more thorough unit tests.

**Limitations**

Limitations of TDD include its inability to perform sufficient testing in situations where full functional tests are required, the need for management support, the risk of blind spots shared between tests and code, the potential for a false sense of security from passing unit tests, and the maintenance overhead of tests.