# 

**EIT-M SCHOOL OF COMPUTING**

**DEPARTMENT OF SOFTWARE ENGINEERING**

Course Title: Software Tools and Practices

Project title: Test Driven Development Banking Application

**PROPOSED BY**

|  |  |
| --- | --- |
| NAME | ID NO |
| 1.HABEN ETSAY | EITM/UR158087/11 |
| 2.ABEL FERADE | EITM/UR156042/11 |
| 3.EPHREM KAHSAY | EITM/UR156039/11 |
| 4.HAILU SHUMYE | EITM/UR156010/11 |
| 5.EKRAM MOHAMMEDABDU | EITM/TUR181518/16 |

**Test-Driven Development (TDD)**

Test-Driven Development (TDD) is a software development methodology where tests are written before the actual code. This approach emphasizes creating small, incremental tests that define the desired functionality of the software. The development process follows a strict cycle of writing tests, implementing code, and then refactoring the code. This cycle is commonly summarized as **"Red-Green-Refactor."**

**Detailed Explanation**

1. **Red Phase: Write a Test**
   * In this phase, a developer writes a test for the next bit of functionality they want to add. This test is written based on the requirements and specifications. Initially, the test will fail because the corresponding functionality has not yet been implemented. This is why it's referred to as the "Red" phase, indicating a failing test.
2. **Green Phase: Write the Code**
   * Once the test is written and verified to fail, the next step is to write the minimum amount of code necessary to make the test pass. This phase focuses on getting the test to pass as quickly as possible, without worrying about the quality or efficiency of the code. The "Green" phase signifies that the test is now passing.
3. **Refactor Phase: Refactor the Code**
   * After making the test pass, the developer then refactors the code. Refactoring involves cleaning up the code, improving its structure, and enhancing its readability without changing its external behavior. This step is crucial for maintaining code quality and ensuring that the system remains robust and easy to maintain. The tests ensure that the refactoring process does not introduce new bugs.

**Principles of TDD**

* **Write Tests First**: Before writing any functional code, write a test that defines a specific requirement or functionality.
* **Incremental Development**: Develop software in small, manageable increments. Each increment is a test-case-driven step.
* **Refactor Continuously**: Regularly refactor the code to improve its structure and remove duplication, ensuring the codebase remains clean and maintainable.
* **Keep Tests Running**: Tests should be automated and run frequently to ensure that new changes do not break existing functionality.

**Benefits of TDD**

* **Higher Code Quality**: Writing tests first leads to better-designed, more modular, and less coupled code. It ensures that the code meets the specified requirements from the outset.
* **Reduced Defects**: Continuous testing helps catch bugs early in the development cycle, reducing the number of defects in the final product.
* **Improved Design**: The need to write tests influences the design of the code, often leading to more thoughtful and cleaner design choices.
* **Documentation**: Tests serve as a form of documentation, providing examples of how the code is expected to behave.
* **Confidence in Refactoring**: With a comprehensive suite of tests, developers can refactor code with confidence, knowing that any changes will be validated against the tests.

**Challenges of TDD**

* **Initial Learning Curve**: TDD can be challenging to adopt initially, requiring a shift in mindset and familiarity with writing tests.
* **Time Investment**: Writing tests can initially slow down the development process, although this is often offset by the reduced time spent on debugging and maintenance.
* **Test Maintenance**: As the codebase evolves, tests need to be maintained and updated, which can add overhead.

**TDD Workflow**

1. **Add a Test**: Write a test for the next piece of functionality.
2. **Run All Tests**: Run all tests and see if the new test fails (it should fail).
3. **Write Code**: Write the simplest code to pass the new test.
4. **Run Tests**: Run all tests again and ensure they pass.
5. **Refactor Code**: Refactor the code for readability and maintainability.
6. **Repeat**: Repeat the cycle for the next piece of functionality.

**Example**

Consider a simple example where a developer wants to implement a function that adds two numbers:

1. **Write a Test**:

//test create Account with intial balance

@Test

public void testCreateAccountWithInitialBalance() throws Exception {

// Test creating an account with an initial balance

bankingApp.createAccount("1234567890", 1000.0);

double balance = bankingApp.checkBalance("1234567890");

assertEquals(1000.0, balance, 0.01);

}

1. **Run the Test**:
   * The test will fail because the  **testCreateAccountWithInitialBalance** function does not exist yet.
2. **Write the Code**:

public void createAccount(String accountNumber, double initialBalance) {

if (!accounts.containsKey(accountNumber)) {

accounts.put(accountNumber, initialBalance);

}

}

1. **Run the Test**:
   * The test will pass because the **testCreateAccountWithInitialBalance** function now meets the requirement.
2. **Refactor**:
   * Since the code is simple, no refactoring may be needed in this case. For more complex code, ensure it is clean and efficient.

By following TDD, developers ensure that every piece of functionality is tested and verified, leading to robust and reliable software development.

**Conclusion**

Test-Driven Development is a powerful methodology that helps developers produce high-quality, maintainable code by writing tests before the implementation. It enforces a disciplined approach to software development, emphasizing testing, refactoring, and incremental progress. Despite its challenges, the benefits of TDD in terms of code quality, defect reduction, and documentation make it a valuable practice in modern software development.