GENERAL

test driven development by example

Published: September 3, 2026 | Index ID: #-

Introduction

Software development has evolved dramatically over the past few decades, with new methodologies, tools, and best practices emerging to address the ever-growing complexity of modern applications. Among these, **test driven development** (TDD) has become a cornerstone for many teams seeking to build reliable, maintainable codebases. But what does it truly mean to write code *by example*? This article will walk you through the principles of TDD, demonstrate a step by step example, and explain how this approach can be integrated into everyday development workflows.

What Is Test Driven Development?

TDD is a software development technique in which developers write unit tests before they write the actual code that satisfies those tests. The process follows a simple, repeatable cycle: write a failing test, write the minimal code to make the test pass, then refactor the code while keeping all tests green. This “red green refactor” loop ensures that code is continuously validated against expected behavior and that design decisions are driven by test specifications.

The Core Principles

- Tests First: Tests are written before the production code.
- Small, Incremental Steps: Each iteration focuses on a tiny piece of functionality.
- Continuous Validation: The test suite runs frequently to catch regressions early.
- Refactoring Safely: Refactoring is only done when all tests pass, guaranteeing that changes do not break existing behavior.

Why TDD Matters

Adopting TDD offers several tangible benefits:

- Higher Code Quality: Tests act as a safety net, reducing bugs and ensuring correct behavior.
- Better Design: Writing tests first forces developers to think about interfaces, responsibilities, and dependencies before implementation.
- Documentation: Tests serve as living documentation, illustrating how the code is intended to be used.
- Faster Feedback: Developers receive immediate feedback on the impact of their changes.

Setting Up the Environment

Before diving into the example, let’s set up a minimal project. We’ll use **Python**with the pytest framework, but the concepts apply across languages.

Prerequisites

1. Python 3.8+ installed.
2. Virtual environment tool (e.g., venv or virtualenv).
3. Basic knowledge of command-line operations.

Project Structure

Here’s a simple layout that keeps tests and source code separate:

```
my_tdd_project/
% % %  src/
%    % % %  calculator.py
% % %  tests/
%    % % %  test_calculator.py
% % %  requirements.txt
```

Installing Dependencies

In the project root, create a requirements.txt with the following content:

```
pytest
```

Then run:

```
python -m venv venv
source venv/bin/activate   # On Windows: venv\Scripts\activate
pip install -r requirements.txt
```

Test Driven Development by Example: Building a Simple Calculator

We'll build a small calculator that supports addition, subtraction, multiplication, and division. By following TDD, we'll see how tests guide the implementation and refactoring process.

Step 1: Write the First Test (Red)

Create tests/test_calculator.py and add a test for the addition operation.

```
from src.calculator import Calculator

def test_addition():
    calc = Calculator()
    result = calc.add(2, 3)
    assert result == 5
```

Run the test suite:

```
pytest tests/test_calculator.py
```

It fails with an AttributeError because Calculator and its add method don't exist yet. This is the expected "red" state.

Step 2: Implement the Minimal Code (Green)

Now create src/calculator.py with a skeleton class and the add method.

```
class Calculator:
    def add(self, a, b):
        return a + b
```

Run the tests again. They pass, marking the “green” state.

Step 3: Refactor (Optional at This Stage)

At this point, the code is already clean. In more complex scenarios, you would refactor here, ensuring no tests fail.

Step 4: Extend the Test Suite (Red)

Next, add tests for subtraction, multiplication, and division.

```
def test_subtraction():
    calc = Calculator()
    assert calc.subtract(5, 2) == 3

def test_multiplication():
    calc = Calculator()
    assert calc.multiply(4, 3) == 12

def test_division():
    calc = Calculator()
    assert calc.divide(10, 2) == 5
```

Run the tests; they all fail because the methods are missing.

Step 5: Implement the Remaining Methods (Green)

Update Calculator to include the missing operations.

```
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Division by zero")
```

```
        return a / b
```

All tests now pass. The code is functional and well-tested.

Step 6: Handle Edge Cases (Red)

Write a test to ensure division by zero raises an exception.

```
def test_division_by_zero():
    calc = Calculator()
    with pytest.raises(ValueError):
        calc.divide(5, 0)
```

Run the tests; they fail because we haven't implemented the error handling yet.

Step 7: Implement Edge Case Handling (Green)

Update the divide method to raise an exception on zero division. This was already included in the previous step, but if it wasn't, you'd add it now.

```
def divide(self, a, b):
    if b == 0:
        raise ValueError("Division by zero")
    return a / b
```

Run the tests again; all pass.

Step 8: Refactor for Readability (Optional)

Consider extracting common logic or simplifying the API. For instance, you might want to support operations via a single calculate method that accepts an operator string:

```
def calculate(self, operator, a, b):
    if operator == '+':
        return self.add(a, b)
    elif operator == '-':
        return self.subtract(a, b)
    elif operator == '*':
        return self.multiply(a, b)
    elif operator == '/':
        return self.divide(a, b)
    else:
        raise ValueError(f"Unsupported operator: {operator}")
```

Add corresponding tests and refactor accordingly. Since the test suite covers all behavior, you can safely refactor without fear of introducing regressions.

Integrating TDD Into a Development Workflow

While the example above demonstrates TDD in isolation, real-world projects require more structure. Below are key practices for scaling TDD across teams.

Continuous Integration (CI)

Automate test runs on every commit or pull request. Tools like GitHub Actions, GitLab CI, or Jenkins can run pytest and report failures before code merges.

Feature Branches and Pull Requests

Develop features in isolated branches. Each pull request should include a suite of tests that fully exercise the new code. Peer reviewers should verify that tests are comprehensive and that the code adheres to project standards.

Test Coverage Metrics

Measure coverage with tools such as coverage.py. Aim for high coverage, but remember that 100% coverage does not guarantee bug free code. Focus on meaningful tests that validate behavior rather than just hitting every line.

Test-First Design Patterns

- Mocking and Stubbing: Replace external dependencies with mocks to isolate unit tests.
- Factory Functions: Use factories to create test data, keeping test setup DRY.
- Parameterized Tests: Run the same test logic with multiple inputs to reduce duplication.

Behavior-Driven Development (BDD) as an Extension

BDD extends TDD by writing tests in a more natural language format, often using frameworks like behave (Python) or Cucumber (Java). This can improve collaboration with non technical stakeholders by expressing requirements as executable scenarios.

Common Pitfalls and How to Avoid Them

Writing Tests After the Code

Some developers still write tests last. This defeats the purpose of TDD. Always commit to the test first mindset.

Over Mocking

Excessive use of mocks can create brittle tests that are tightly coupled to implementation details. Mock only what is necessary.

Neglecting Test Maintenance

As the codebase evolves, tests may become outdated. Regularly review and refactor tests to keep them relevant.

Ignoring Test Performance

Slow tests can discourage frequent execution. Optimize by keeping tests lightweight and using fixtures wisely.

Beyond Unit Tests: Integration and System Testing

TDD focuses on unit tests, but comprehensive quality assurance also requires integration and system tests. These tests validate interactions between components and overall system behavior. In a TDD centric workflow, you can write integration tests after unit tests pass, ensuring that the entire stack behaves as expected.

Example: End-to-End Test for a Calculator API

Suppose you expose the calculator as a REST API. You could write an integration test that sends HTTP requests and verifies responses:

```
def test_calculator_api_addition():
    response = client.post("/calculate", json={"operator": "+", "a": 2, "b": 3})
    assert response.status_code == 200
    assert response.json()["result"] == 5
```

Case Studies: Companies That Thrive with TDD

- Google: Uses rigorous testing pipelines and encourages TDD for new features.
- Netflix: Employs extensive unit tests and automated refactoring to maintain a massive codebase.
- Facebook: Relies on unit tests and code reviews to catch regressions early.

These organizations demonstrate that TDD, when combined with automation and culture, can scale to large, complex systems.

Getting Started with TDD in Your Team

1. Educate: Provide workshops or pair programming sessions to demonstrate the red green refactor cycle.
2. Define Standards: Establish guidelines for test naming, structure, and coverage thresholds.
3. Automate: Integrate test runs into CI pipelines to enforce quality gates.