TDD Terminology Simplified

Denis Sokolov on Mar 12th 2013

The core idea of Test-Driven Development (TDD) is writing tests before writing any functional code, and then writing only the least possible amount of code required to make the tests pass. It may sound strange to develop in this fashion, but it's actually quite useful, as the test base doubles as a partial specification of the main code.

Given such a simple premise, however, there is an amazing amount of terminology and techniques. In this article, I gather the most important terms and buzzwords that you might hear, and define them.

Acceptance Testing

The highest level of testing validates that the software meets the customer's requirements. Acceptance testing is commonly run in environments as close to production as possible. See functional testing and system testing.

Assertion

Assertions are statements that perform an actual check on the software's output. In general, a single function called assertis enough to express any check. In practice, test libraries often have many assert functions to meet specific needs (such asassertFalse, assertEqual and more) to offer better analysis and friendlier output.

Behavior Testing

A testing technique that incorporates test doubles to the software, asserting that it calls correct methods in a correct order. See mock for an example. Also see state testing.

Behavior-Driven Development (BDD)

A subset of TDD driven by the need of clearer communication and proper documentation. BDD is perhaps the biggest recent development in TDD.

Its core idea is to replace confusing and developer-centric terminology (tests, suites, assertions etc) with ubiquitous language that all participating stakeholders (including non-technical staff, and, possibly, clients) can understand.

See user story.

Black-Box Testing

A general principle in testing where the person writing tests does not know or avoids the internals of the software, choosing instead to test the public interface of the software strictly by its interface or specification. See white-box testing.

Boundary-Value (Edge Case) Testing

A strategy for writing tests to catch off-by-one and other similar types of errors. To perform boundary-value testing, test the inputs around certain possibly problematic boundaries. In case of integers, this might be 0,-1, MIN_INT, MAX_INT and other similar values.

Dummy

A dummy is a type of test double that is never used by the actual software, but is only used in testing to fill required parameters.

Fake

Fakes are test doubles that implement the required functionality in a way that is useful in testing, but which also effectively disqualifies it from being used in production environment. For example, a key-value database that stores all values in memory and loses them after every execution potentially allows tests to run faster, but its tendency to destroy data would not allow it to be used in production.

Fixture

A particular environment that must be set up before a test can be run. It generally consists of setting up all test doubles and other dependencies for the software under test: such as inserting predefined data into afake database, setting up certain directory structure in the fake file system, setting up properties on the dependencies of software under test.

Functional Testing

A high level testing activity verifying that all business requirements of the product are met. Functional testing commonly involves using user stories to focus on a higher level of requirements to cover as many usage scenarios as possible. See acceptance testing and system testing.

Green

A colloquialism for a passing collection of tests, or sometimes a particular passing test. See red.

Integration Testing

A mid-level testing activity that verifies a certain set of modules work correctly together. Integration tests are like unit tests without using test doubles for a certain subset of dependencies, essentially testing the interactions between the software its dependencies.

Mock

A type of test double created for a particular test or test case. It expects to be called a specific number of times and gives a predefined answer. At the end of the test, a mock raises an error if it was not called as many times as expected. A mock with strict expectations is part of the assertion framework.

Red

A colloquialism for a failing collection of tests or sometimes a particular failing test. See green.

Refactoring

The process of improving implementation details of code without changing its functionality.
Refactoring without tests is a very brittle process, as the developer doing the refactoring can never be sure that his improvements are not breaking some parts of functionality.
If the code was written using test-driven development, the developer can be sure that his refactoring was successful as soon as all tests pass, as all the required functionality of the code is still correct.

Regression

A software defect which appears in a particular feature after some event (usually a change in the code).

Scenario Testing

See functional testing.

Setup

A process of preparing a fixture. See teardown. Example:

State Testing

A form of unit testing when the testing code provides test doubles to and asserts that the state of these doubles has been modified in a correct fashion. See behavior testing.

Stub

A type of test double that can reply to the software being tested with predefined answers. Unlike mocks, however, stubs do not usually check if they have been called properly, but rather only make sure that the software can call its dependencies.

System Testing

A high level testing activity when the entirety of the software is tested top to bottom. This includes functional testing, as well as checking other characteristics (such as performance and stability).

SUT

An abbreviation for *software under test*. Used to distinguish the software under test from its dependencies.

Teardown

A process of cleaning up a fixture. In garbage-collected languages, this functionality is mostly handled automatically. See setup.

Test

The smallest possible check for correctness. For example, a single test for a web form could be a check that, when given an invalid email address, the form warns the user and suggests a fix. See test case.

Test Case

A collection of tests grouped by an attribute. For example, a test case for a web form could be a collection of tests checking the behavior of the form for different valid and invalid inputs.

Test Coverage

Any kind of metric that attempts to estimate the likelihood of important behavior of the SUT still not covered by tests. Most popular techniques include different kinds of *code coverage*: techniques that make sure that all possible code statements (or functions, or logical branches in the code) have been executed during testing.

Test Cycle

A process of TDD development. Given that TDD development starts with writing a few tests, it is clear that the test suite starts red. As soon as the developer implements all newly tested functionality, tests turn green. Now the developer can safely refactor his implementation without the risk of introducing new bugs, as he has a test suite to rely on. Once refactoring is complete, the developer can start the cycle again by writing more tests for more new functionality. Thus, the *red-green-refactor test cycle*.

Test Double

Test doubles are objects the test code creates and passes to the SUT to replace real dependencies. For example, unit tests should be very fast and only test a particular piece of software.

For these reasons, its dependencies, such as a database or file system interaction libraries, are usually replaced by objects that act in memory instead of talking to a real database or file system.

There are four main categories of test doubles: dummies, fakes, stubs, and mocks.

Test Suite

A collection of test cases that test a large portion of software. Alternatively, all test cases for a particular software.

Test-First Programming

Test-first programming is a slightly broader term for test-driven development. While TDD promotes tight coupling between writing tests (usually unit tests) and writing the code, test-first programming allows for high level functional tests instead. However, the distinction in general usage is rarely noted, and two the terms are usually used interchangeably.

Unit Testing

The lowest level testing technique consisting of test cases for the smallest possible units of code. A single unit test usually checks only a particular small behavior, and a unit test caseusually covers all functionality of a particular single function or class.

User Story

A single description of a particular group of people willing to perform a particular task using the SUT to achieve a particular goal. User stories are usually defined in human languages, using simple, user-centric terms to avoid considering implementation details and to focus on user experience instead. For example: