

Kinds of Unit Tests

**TDD – Test Driven Development**

red – green – refactor. Test a little – code a little.

**DDT – Defect Driven Testing**

Write a unit test that reproduces the defect – Fix code – Test will succeed – Defect will never return.

**POUting – Plain Old Unit Testing**

aka test after. Write unit tests to check existing code. You cannot and probably do not want to test drive everything. Use POUT to increase sanity. Use to add additional tests after TDDing (e.g. boundary cases).

Design for Testability

**Constructor – Simplicity**

Objects have to be easily creatable. Otherwise, easy and fast testing is not possible.

**Constructor – Lifetime**

Pass dependencies and configuration/parameters into the constructor that have a lifetime equal or longer than the created object. For other values use methods or properties.

**Clean Code**

Follow the guidelines from Clean Code to get a design that is easy testable. If it is not easy testable then the design has to be improved.

**Abstraction Layers at System Boundary**

Use abstraction layers at system boundaries (database, file system, web services, COM interfaces ...) that simplify unit testing by enabling the usage of mocks.

Structure

**Arrange – Act – Assert**

Structure the tests always by AAA.

**Test Assemblies**

Create a test assembly for each production assembly and name it as the production assembly + „Test“.

**Test Namespace**

Put the tests in the same namespace as their associated testee.

**Unit Test Methods show whole truth**

Unit test methods show all parts needed for the test. Do not use SetUp method or base classes to perform actions on testee or dependencies.

**SetUp / TearDown for infrastructure only**

Use the SetUp method only for infrastructure that your unit test needs. Do not use it for anything that is under test.

**Test Method Naming**

Names reflect what is tested, e.g. FeatureWhenScenarioThenBehaviour

Mocking (Stubs, Fakes, Spies, Mocks, ...)

**Isolation from environment**

Use mocks to simulate all dependencies of the testee.

**Mocking framework**

Use a dynamic mock framework for mocks that show different behaviour in different test scenarios (little behaviour reuse).

**Manually written mocks**

Use manually written mocks when they can be used in several tests and they have only little changed behaviour in these scenarios (behaviour reuse).

**Mixing Stubbing and Expectation Declaration**

Make sure that you follow the AAA (arrange, act, assert) syntax when using mocks. Don't mix setting up stubs (so that the testee can run) with expectations (on what the testee should do) in the same code block.

**Checking mocks instead of testee**

Tests that check not the testee but values returned by mocks. Normally, due to excessive mock usage.

**Excessive mock usage**

If your test needs a lot of mocks or mock setup then consider splitting the testee into several classes or provide an additional abstraction between your testee and its dependencies.

Unit Test Principles

**Fast**

Unit tests have to be fast in order to be executed often. Fast means much smaller than seconds.

**Isolated**

Clear where the failure happened. No dependency between tests (random order)

**Repeatable**

No assumed initial state, nothing left behind. No dependency on external services that might be unavailable (databases, file system, ...)

**Self Validating**

No manual test interpretation or intervention. Red or green!

**Timely**

Tests are written at the right time (TDD, DDT, POUting)

Unit Test Smells

**Test not testing anything**

Passing test that at first sight appears valid, but does not test the testee.

**Test needing excessive setup**

A test that needs dozens of lines of code to setup its environment. This noise makes it difficult to see what is really tested.

**Too large test**

A valid test that is however too large. Reasons can be that this test checks for more than one feature or the testee does more than one thing (violation of Single Responsibility Principle).

**Checking internals**

A test that accesses internals of the testee (private/protected members). This is a refactoring killer.

**Test only running on the developer's machine**

A test that is dependent on the development environment and fails elsewhere. Use Continuous Integration to catch them as soon as possible.

**Test checking more than necessary**

A test that checks more than it is dedicated to. These tests fails whenever something changes that it unnecessarily checks. Especially probable when mocks are involved or checking for item order in unordered collections.

**Missing assertions**

Tests that do not have any assertions.

**Chatty test**

A test that fills the console with text – probably used once to check for something manually.

**Test swallowing exceptions**

A test that catches exceptions and let the test pass.

**Test not belonging in host test fixture**

A test that tests a completely different testee than all other tests in the fixture.

**Obsolete test**

A test that checks something no longer required in the system. May even prevent clean up of production code because it is still referenced.

**Hidden test functionality**

Test functionality hidden in either the SetUp method, base class or helper class. The test should be clear by looking at the test method only – no initialization or asserts somewhere else.

**Mixing Act and Assert**

Assert statements that execute code on the testee. First, execute operation on testee and store result in a local variable. Afterwards, make assertions. Especially, if several assertions in a single test.

**Unclear Fail Reason**

Split test or use assertion messages.

**Conditional Test Logic**

Tests should not have any conditional test logic because it's hard to read.

**Test Logic in Production Code**

Tests depend on special logic in production code.

**Erratic Test**

Sometimes passes, sometimes fails due to left overs or environment.

TDD Principles

**A test checks one feature**

A test checks exactly one feature of the testee. That means that it tests all things included in this feature but not more. This includes probably more than one call to the testee. This way, the tests serve as samples and documentation of the usage of the testee.

**Tiny steps**

Make tiny little steps. Add only a little code in test before writing the needed production code. Then repeat. Add only one Assert per step.

**Keep tests simple**

Whenever a tests gets complicated, check whether you can split the testee into several classes (Single Responsibility Principle)

**Prefer State Verification to Behaviour Verification**

Use behaviour verification only if there is no state to verify.

**Test Domain Specific Language**

Use test DSLs to simplify reading tests: helper method, classes, ...

Don't assume

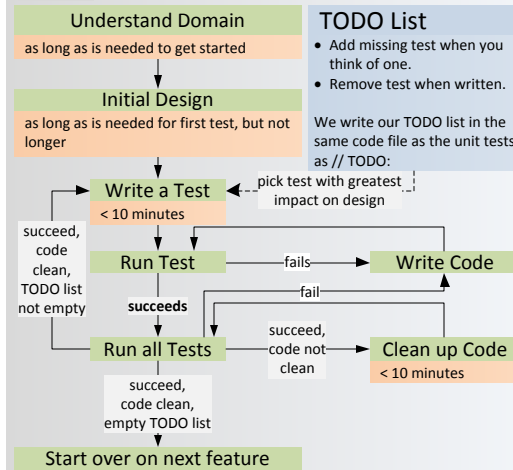
**Understand The Algorithm**

Just working is not enough, make sure you understand why it works.

**Incorrect Behaviour At Boundaries**

Always unit test boundaries. Do not assume behaviour.

TDD Cycle



TDD Process Smells

**No Green Bar in the last ~10 Minutes**

Make small steps to get feedback as fast and frequent as possible.

**Not running test before writing production code**

Only if the test fails then new code is required. Additionally, if the test does surprisingly not fail then make sure the test is correct.

**Not spending enough time on refactoring**

Refactoring is an investment into the future. Readability, changeability and extensibility will pay back.

**Skipping something too easy to test**

Don't assume, check it. If it is easy then the test is even easier.

**Skipping something too hard to test**

Make it simpler, otherwise bugs will hide in there and maintainability will suffer.

**Organizing tests around methods, not behavior.**

These tests are brittle and refactoring killers. Test complete „mini“ use cases in a way the feature will be used in the real world.

**Using Code Coverage as a Goal**

Use code coverage to find missing tests but don't use it as a driving tool. Otherwise, the result could be tests that increase code coverage but not certainty.

Red Bar Patterns

**One Step Test**

Pick a test you are confident you can implement and maximizes learning effect (e.g. impact on design).

**Another Test**

If you think of new tests then write them on the TODO list and don't loose focus on current test.

**Learning Test**

Write tests against external components to make sure they behave as expected.

Green Bar Patterns

**Fake It (~Til You Make It)**

Return a constant to get first test running. Refactor later.

**Triangulate – Drive Abstraction**

Write test with at least two sets of sample data. Abstract implementation on these.

**Obvious Implementation**

If the implementation is obvious then just implement it and see if test runs. If not then step back and just get test running and refactor then.

**One to Many – Drive Collection Operations**

First, implement operation for a single element. Then, step to several elements.

Refactoring Patterns

**Reconcile Differences – Unify Similar Code**

Stepwise change both pieces of code until they are identical.

**Isolate Change**

First, isolate the code to be refactored from the rest. Then refactor. Finally, undo isolation.

**Migrate Data**

Moving from one representation to another by temporary duplication.

**Temporary Parallel Implementation**

Refactoring test per test by introducing a temporary parallel implementation. Remove old solution when all tests are refactored.

Acceptance Test Driven Development

**Use Acceptance Tests to drive your TDD tests**

Acceptance tests check for the needed functionality. Let them guide your TDD.

**User Feature Test**

An Acceptance test is a test for a complete user feature from top to bottom that provides business value.

**Automated ATDD**

Use automated Acceptance Test Driven Development for regression testing and executable specifications.

Continuous Integration

**Commit Check**

Run all unit tests covering currently worked on code prior to committing to the source code repository.

**Integration Check**

Run all automated integration and unit tests on every commit to the source code repository.

**Automated Acceptance Tests**

Run all automated acceptance tests as often as possible on the integration server.

**Communicate Failed Integration to Whole Team**

Whenever a stage on the continuous integration server fails then notify whole team in order to get blocking situation resolved as soon as possible.

**Automatically build an Installer for Test System**

Automatically build an installer as often as possible to test software on a test system (for manual tests, or tests with real hardware).

Practices

Smells

V 1.2

Author: UrsENZler

