Clean TDD/ATDD Cheat Sheet

Kinds of Automated Tests

ATDD – Acceptance Test Driven Development +
Specify a feature first with a test, then implement.

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 to or longer than the created object. For other values use methods or properties.

Abstraction Layers at System Boundary +
Use abstraction layers at system boundaries (database, file system, web services, ...) that simplify unit testing by enabling the usage of fakes.

Structure

Arrange – Act – Assert +
Structure the tests always by AAA. Never mix these three blocks.

Test Assemblies (.Net) +
Create a test assembly for each production assembly and name it as the production assembly + ".Test"/".Facts"/... .

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 / TearDown methods only for infrastructure that your unit test needs. Do not use it for anything that is under test.

Test Method Naming +
Use a pattern that reflects behaviour of tested code, e.g. *Behaviour*[_On*Trigger*][_When*Scenario*] with [] as optional parts.

Resource Files +
Test and resource are together: FooTest.cs, FooTest.resx

Naming

Naming SUT Test Variables +
Give the variable holding the System Under Test always the same name (e.g. testee or sut). Clearly identifies the SUT, robust against refactoring.

Naming Result Values +
Give the variable holding the result of the tested method always the same name (e.g. result).

Anonymous Variables +
Always use the same name for variables holding uninteresting arguments to tested methods (e.g. anonymousText, anyText).

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.

Faking (Stubs, Fakes, Spies, Mocks, Test Doubles ...)

Isolation from environment +
Use fakes to simulate all dependencies of the testee.

Faking Framework +
Use a dynamic fake framework for fakes that show different behaviour in different test scenarios (little behaviour reuse).

Manually Written Fakes +
Use manually written fakes 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 fakes. 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 Fakes instead of Testee –
Tests that do not check the testee but values returned by fakes. Normally due to excessive fake usage.

Excessive Fake Usage –
If your test needs a lot of fakes or fake 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 +
Isolated testee: Clear where the failure happened.
Isolated test: 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 set up its environment. This noise makes it difficult to see what is really tested.

Too Large Test / Assertions for Multiple Scenarios –
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 (private/protected members) of the testee directly (Reflection). This is a refactoring killer.

Test Only Running on 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. The test fails whenever something changes that it checks unnecessarily. Especially probable when fakes are involved or checking for item order in unordered collections.

Irrelevant Information –
Test contains information that is not relevant to understand it.

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

Test Swallowing Exceptions –
A test that catches exceptions and lets 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 initialisation or asserts somewhere else.

Bloated Construction –
The construction of dependencies and arguments used in calls to testee makes test hardly readable. Extract to helper methods that can be reused.

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 required production code. Then repeat. Add only one Assert per step.

Keep Tests Simple +
Whenever a test 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. Refactoring is easier due to less coupling to implementation.

Test Domain Specific Language +
Use test DSLs to simplify reading tests: builders to create test data using fluent APIs, assertion helpers for concise assertions.

Red Bar Patterns

One Step Test +
Pick a test you are confident you can implement and which maximises learning effect (e.g. impact on design).

Partial Test +
Write a test that does not fully check the required behaviour, but brings you a step closer to it. Then use Extend Test below.

Extend Test +
Extend an existing test to better match real-world scenarios.

Another Test +
If you think of new tests, then write them on the TO DO list and don't lose 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 (and no element).

TDD Process Smells

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.

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 surprisingly does not fail then make sure the test is correct.

Not Spending Enough Time on Refactoring –
Refactoring is an investment in 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.

Organising Tests around Methods, Not Behaviour –
These tests are brittle and refactoring killers. Test complete "mini" use cases in a way which reflects how the feature will be used in the real world. Do not test setters and getters in isolation, test the scenario they are used in.

Acceptance Test Driven Development

Use Acceptance Tests to Drive Your TDD tests +
Acceptance tests check for the required 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.

Component Acceptance Tests +
Write acceptance tests for individual components or subsystems so that these parts can be combined freely without losing test coverage.

Simulate System Boundaries +
Simulate system boundaries like the user interface, databases, file system and external services to speed up your acceptance tests and to be able to check exceptional cases (e.g. a full hard disk). Use system tests to check the boundaries.

Acceptance Test Spree –
Do not write acceptance tests for every possibility. Write acceptance tests only for real scenarios. The exceptional and theoretical cases can be covered more easily with unit tests.

Legend:

DO

DON'T

Clean TDD/ATDD Cheat Sheet

Continuous Integration

Pre-Commit Check +

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

Post-Commit Check +

Run all unit and acceptance tests on every commit to the version control system on the continuous integration server.

Communicate Failed Integration to Whole Team +

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

Build Staging +

Split the complete continuous integration workflow into individual stages to reduce feedback time.

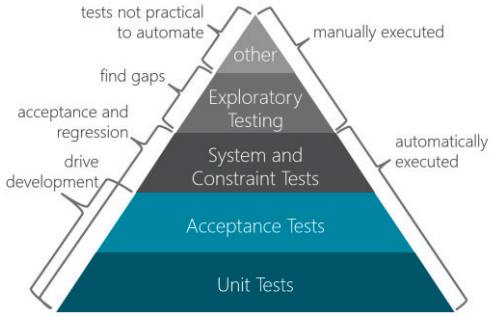
Automatically Build an Installer for Test System +

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

Continuous Deployment +

Install the system to a test environment on every commit/push and on manual request. Deployment to production environment is automated to prevent manual mistakes, too.

Test Pyramid



Constraint Test = Test for non-functional requirements.

Bibliography

Test Driven Development: By Example by Kent Beck

ATDD by Example: A Practical Guide to Acceptance Test-Driven Development by Markus Gärtner

The Art of Unit testing by Roy Osherove

xUnit Test Patterns: Refactoring Test Code by Gerard Meszaros

ATDD, TDD cycle

Write acceptance criteria for user story
The whole team defines acceptance criteria for user stories.

Define examples
The whole team defines examples for acceptance criteria used to show that code works.

Write acceptance test skeleton
Map the examples into an empty specification/test in your acceptance test framework (Gherkin, MSpec/xBehave classes and statements …)

You need to build up knowledge to implement the acceptance test.

You have enough knowledge to implement the acceptance test.

Explore design
Implement a Spike to gather enough knowledge so you can design a possible solution.

Make an initial design
Roughly design how you want to implement the new functionality, especially the interface for your acceptance test (how to call and verify functionality).

Refactor
Refactor existing code to simplify introduction of new functionality. Run all tests to keep code working.

Write an acceptance test
Add arrange, act and assert parts to the acceptance test skeleton (Given, When, Then or Establish, Because, It …).

Succeeded, not all acceptance tests implemented yet

Run acceptance test

Failed

Succeeded

Run all acceptance tests

Succeeded and all examples tested

Failed

Make error reason obvious
The failing test should state what went wrong so you don't have to debug the code.

You have no class design idea

You have a class design idea

Succeeded, code clean, TO DO list empty

Spike a solution
Implement a Spike to get the acceptance test running so that you get an initial design.

Make initial or update class design
Design how you want to implement the new functionality.

Do per class

TO DO list
- Add missing test when you think of one
- Remove test when written

We write the TO DO list into the same file as the unit test with // TODO:

Write a test
Add a minimal test or make a minimal change to an existing test (< 10 minutes).

Pick test with the following priority:
1) Prove that the code is making a hard coded assumption.
2) Prove that something is wrong.
3) Prove that something is missing.

Run test

Succeeded, code clean, TO DO list not empty

Failed

Succeeded

Make error reason obvious
The failing test should state what went wrong so you don't have to debug the code.

Failed

Run all tests

Write code
Write as little code as possible to make the test pass.

Succeeded, code not clean

Clean up code
Apply clean code guidelines. Redesign classes as needed. (< 10 minutes).

ATDD

TDD

Legend:

DO

DON'T

Urs Enzler www.bbv.ch February 2015 V2.5