# Standardized Naming Convention

“Readability is perhaps even more important in unit tests than it is in production code” – Bob Martin

There are **3** main elements to well-named unit tests:

1. What is being tested
2. Under what circumstances (uses “With” prefix)
3. What is the expected result (uses “Should” prefix)

UnitOfWork\_StateUnderTest\_ExpectedBehavior

*Example:*

* Public void Sum\_WithNegativeNumberAs2ndParam\_ShouldThrowException ()
* Public void CheckMemberPend\_WithNormalPendValues\_ShouldSetPendCodeAndStatus()

*Goal:* When a unit test fails, we should understand what just broke without having to read the test code.

# Standardized Structure

AAA pattern

1. **Arrange** – initialize the objects and values that will be tested.
2. **Act** – invoke the tested method with the arranged values.
3. **Assert** – verifies the method functions as expected.

*Example:*

*Example of a bad unit test:*

***Code smell*** *- a surface indication that usually corresponds to a deeper problem in the system (that likely increases technical debt).*

*Example of a code smell:*

public void BadPushPopTest()   
{    
        **// Arrange**   
        var stack = new Stack<int>();   
        Assert.That(stack.Count, Is.EqualTo(0));  **// Warning: asserting in the Arrange phase**!

**// Act**        stack.Push(1);

**// Assert**   
        Assert.That(stack.Count, Is.EqualTo(1));    
        int popped = stack.Pop();                      **// Warning: action in the Assert phase**!   
        Assert.That(popped, Is.EqualTo(1));       **// Warning: why are we asserting again?**        Assert.That(stack.Count, Is.EqualTo(0)); **// Warning: and again…**}

If it is very difficult to structure your test into AAA, it is almost always a code smell which indicates the test and/or code being tested should be refactored.

Benefits of AAA include:

* Separation of concerns lead to more modular tests
* Code smells are more visible – (assertions mixed with act, test methods that have too many responsibilities, ect.)

# Standardized Terminology

## General terms

**Test double / fake** – generic term for any kind of fake object used in place of a realistic object for testing. Test double is derived from the idea of a stunt double in movies.

\*Note there are different definitions for variations of test doubles such as dummies, fakes, spies, and others, but stubs and mocks are the two most popular.

**Brittle tests** – Not flexible, require overwhelming effort to maintain

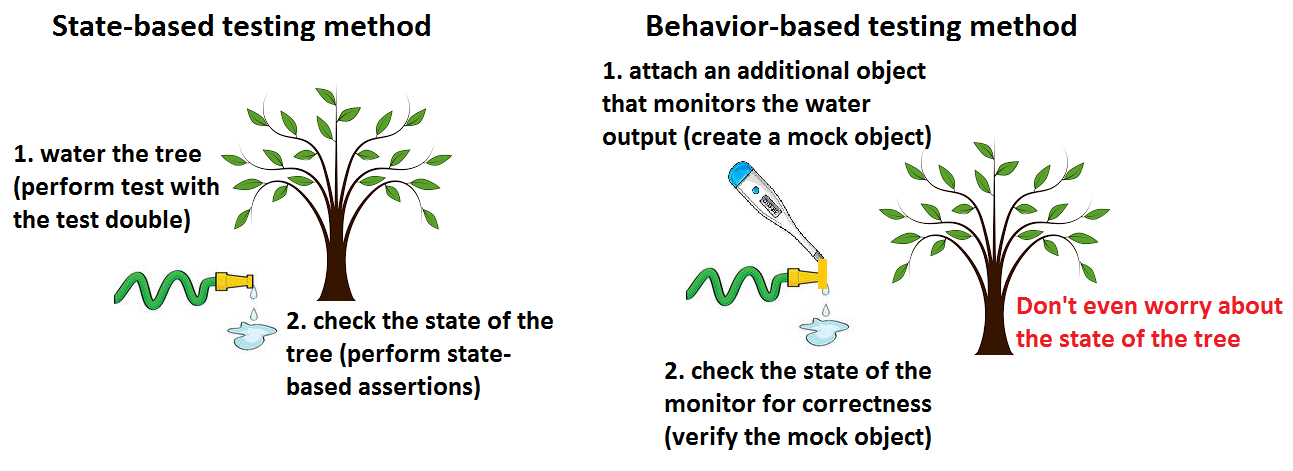
## Stubs vs Mocks

Real-life analogy:

*Imagine you have a tree that you would like to water with a hose for a certain amount of time (2 hours) to get the tree to grow correctly.*

**State-based stub testing**: Run the water for a certain amount (2 hours), then check the tree to see if it is growing well.

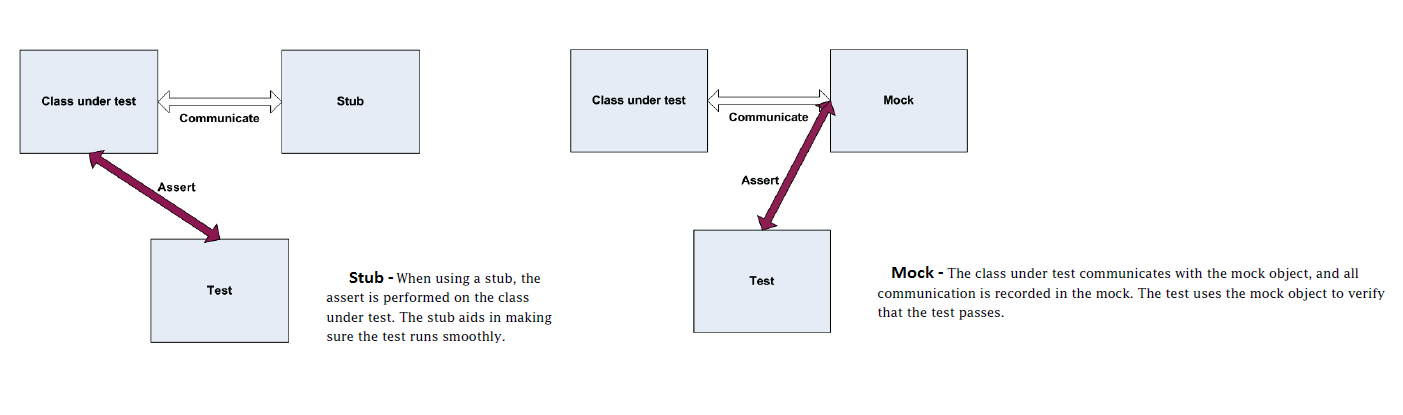
**Behavior-based mock testing**: Attach a device to the end of your hose that monitors how long water flows through the device. Check the monitoring device to assure it had been watering the tree for 2 hours.

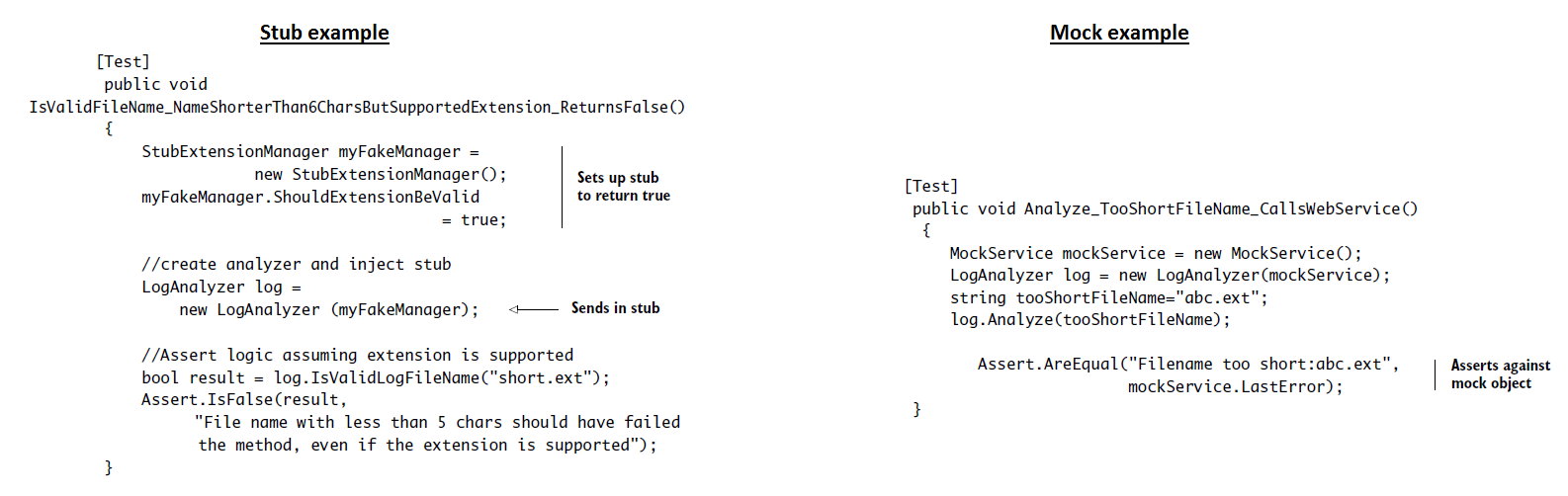


**Stub objects** – implement the dependency as a concrete class which only mimics a small part of the overall behavior of the stubbed object

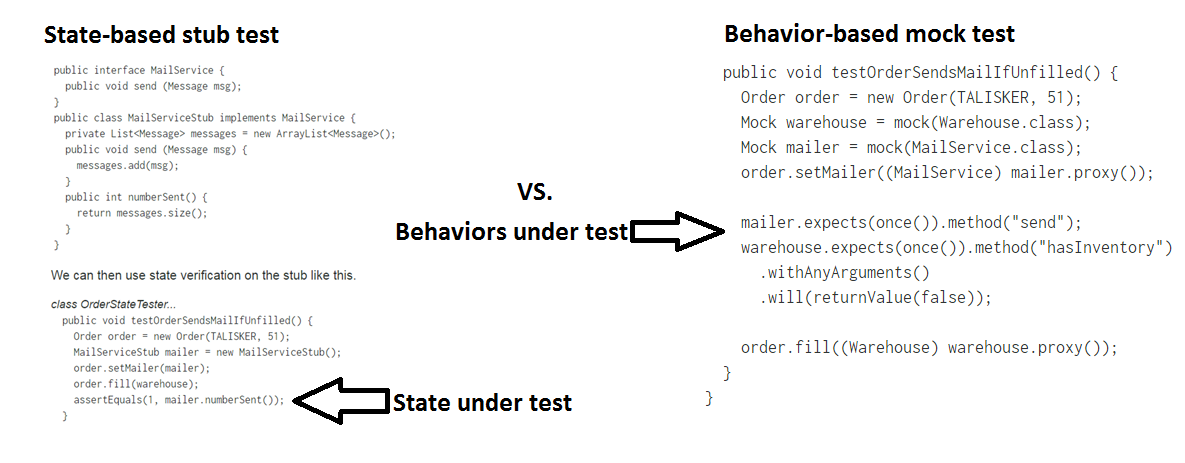
* Stubs emphasize state-based verification
* Stubs often mimic dependency behavior using significant shortcuts

**Mock objects** – an object that we can set expectations on, and which will verify that the expected actions have indeed occurred.

* Mocks emphasize behavior-based verification
* Mocks expect some method Foo() to be called with certain parameters, then it records and verifies those expectations (stubs only replace the behavior)
* A mock is a more detailed stub

*Example 1:*

*Example 2:*

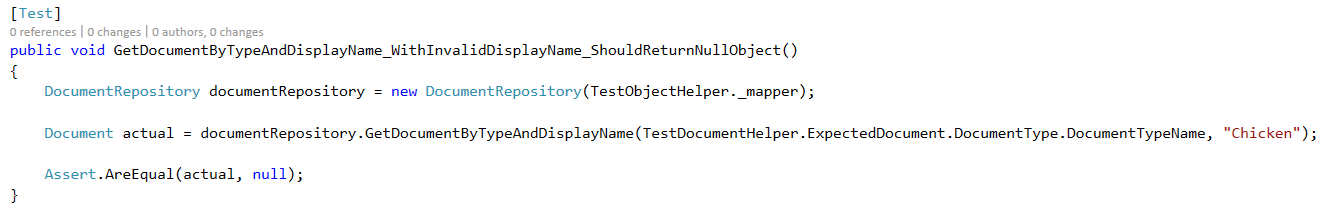


**When to use which?**

Use either a mock or stub based on what suits your test case and creates the simplest code to read and maintain. Our current unit testing codebase is heavily biased towards stubbing and stubbing tends to require less setup and maintenance than mocks.

Source: (“The Art of Unit Testing with Examples in .NET”, Manning 2009)

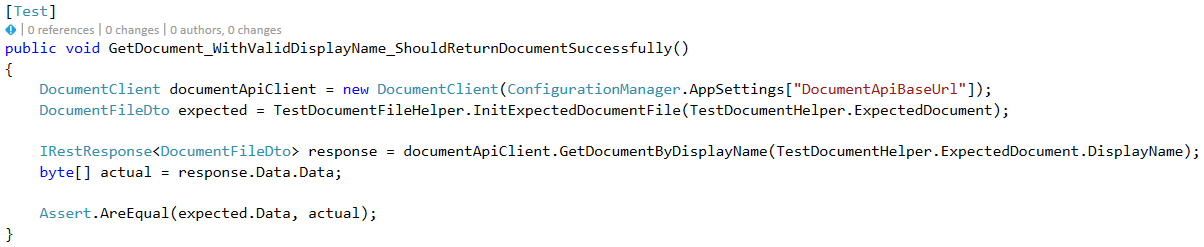
## Unit test vs integration test

**Unit test** – verifies the behavior of a small part of the software in isolated test scenarios

The “act” part of the test stays within the class it is testing i.e. an isolated unit test

**Integration test** – verifies that behavior parts work together in realistic scenarios. Integration tests typically require more effort in setup, teardown, and overall maintenance.

* Most suggest that integration tests remain separate from unit tests because they take longer to run and are more dependent
* Integration tests do not need to run every build, but unit tests should



The “act” part of the test reaches out to the document API and tests the system’s ability to integrate with that external API.

# Standardized Guidelines

## F.I.R.S.T

* Runtime Efficiency
  + Unit tests can be written with less efficient runtimes than production code
  + However, the cleanliness should remain on par with the production standard
* Assertions per test – multiple or one per unit test?
  + When multiple assertions seem appropriate, we can refactor our tests into redundant, smaller pieces or we could use the Template design pattern
  + However, the authors of Clean Code deem this unnecessary and support multiple assertions per test when appropriate, but minimize this amount (pg. 131, “Clean Code”)

Follow **F.I.R.S.T**:

**Fast** – Test should run fast (test suite should finish within about 10 seconds)

If not followed: tests will not get run as often as they should

**violation example:** A test that very slowly retrieves a resource used in the test

**Independent** – Tests should be able to run independently of all other tests

If not followed: tests downstream of a failed test will affect others and not be accurate

**violation example:** A test the relies on a state-change provided by a preceding test

**Repeatable** – Tests should be repeatable against any environment

If not followed: you’ll always have an excuse for their failure and never amend them

**violation example:** The test behaves different when run on a Tuesday vs. any other day

**Self-validating** – tests should return a Boolean pass or fail

If not followed: tests are subjective and require manual checks

**violation example:** A test that returns an object

**Timely** – Tests should be written just before the production code

If not followed: the production may be designed to be difficult to test

**violation example:** The production code was written in a very untestable way, thus testing will be difficult without re-factoring

Source: (pg. 133, “Clean Code”)

## Evolving our designs

* By writing tests, we force ourselves to design the program to be modular and testable
* Always keep “How will future refactoring affect my unit tests” in mind
* We are immediately concerned with the client-facing side of the program because we now have the perspective of the caller of our program
* Our unit tests serve as an invaluable form of documentation – it is compliable, executable, current, and it **cannot lie**
* Write tests to reproduce bugs. Bugs are usually found in groups, thus bug fixes should have extra coverage of tests
* Use explicit asserts for more accurate error information from failing tests. For example, use “*assertEquals(a, b)”*, not “*assertTrue(a == b)”* because the equals method will return more detail regarding object inequality while the latter expression will simply return an invalid expression error.

Source: (ch. 4, “Agile Principles, Patterns, and Practices”)

## Unit Testing Code Smells

1. Having Insufficient tests

How to resolve: Strive to have everything that could possibly break covered by tests

1. Not using a coverage tool

How to resolve: Most IDEs will mark lines covered by unit testing in green

1. Skipping trivial tests

How to resolve: Spend time writing easy tests as it is well worth the investment

1. Ignoring or commenting out random tests

How to resolve: Only ignore tests that test ambiguous requirements. When the requirement details are known, remove or add the test into the suite

1. Not testing boundary conditions

How to resolve: Thoroughly test all possible conditions of the algorithm under test

1. Not exhaustively testing around bugs

How to resolve: Extensively test code surrounding spotted bugs to ensure no further issues

1. Groups of tests are unrelated

How to resolve: Group related tests to run at once because patterns of failed, related tests can reveal patterns to help with debugging

1. Slow running tests

How to resolve: Write test groupings that run under 10 seconds

Source: (chapter 17 – pg. 313-314, “Clean Code”)

# Standardized Suite

3 main options (ordered from most preferred to least preferred):

1. Testing suite in the same solution, but a different project

Pros:

* Good separation of tests from production code
* Tests in the same solution make them easily available to developers

Cons:

* Developers will need to assure the unit test projects are managed and maintained in a way that easily relates to the production code

1. Testing suite in the same solution, and same project as the portion under test

Pros:

* Untested class files are obvious to the developer (they will be the only files without a similar file name post-fixed with “test”)

Cons:

* Unit tests are highly coupled with production code

1. Testing suite in a different solution

Pros:

* Separation of unit tests from production assembly

Cons:

* Difficult for developers to constantly switch between solutions

# Testing private methods

Do not test all private methods as they will cause brittle unit tests