What Makes a Good Unit Test? [closed]

Asked 16 years, 3 months ago Modified 7 years, 11 months ago

Viewed 60k times

97

votes

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 13 years ago.

Locked. This question and its answers are locked because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

I'm sure most of you are writing lots of automated tests and that you also have run into some common pitfalls when unit testing.

My question is do you follow any rules of conduct for writing tests in order to avoid problems in the future? To be more specific: What are the **properties of good unit tests** or how do you write your tests?

Language agnostic suggestions are encouraged.

unit-testing language-agnostic tdd integration-testing testing-strategies

Share

edited Jan 1, 2009 at 23:44

asked Sep 14, 2008 at 15:20

Spoike

122k ● 45 ● 142 ● 158

Comments disabled on deleted / locked posts / reviews

18 Answers

Sorted by: Highest score (default)

93

votes

Let me begin by plugging sources - Pragmatic Unit Testing in Java with JUnit (There's a version with C#-Nunit too.. but I have this one.. its agnostic for the most part. Recommended.)

Good Tests should be **A TRIP** (The acronymn isn't sticky enough - I have a printout of the cheatsheet in the book that I had to pull out to make sure I got this right..)

- **Automatic** : Invoking of tests as well as checking results for PASS/FAIL should be automatic
- **Thorough**: Coverage; Although bugs tend to cluster around certain regions in the code, ensure that you

test all key paths and scenarios.. Use tools if you must to know untested regions

- **Repeatable**: Tests should produce the same results each time.. every time. Tests should not rely on uncontrollable params.
- **Independent**: Very important.
    - Tests should **test only one thing** at a time. Multiple assertions are okay as long as they are all testing one feature/behavior. When a test fails, it should pinpoint the location of the problem.
    - Tests **should not rely on each other** - Isolated. No assumptions about order of test execution. Ensure 'clean slate' before each test by using setup/teardown appropriately
- **Professional**: In the long run you'll have as much test code as production (if not more), therefore follow the same standard of good-design for your test code. Well factored methods-classes with intention-revealing names, No duplication, tests with good names, etc.
- Good tests also run **Fast**. any test that takes over half a second to run.. needs to be worked upon. The longer the test suite takes for a run.. the less frequently it will be run. The more changes the dev will try to sneak between runs.. if anything breaks.. it will take longer to figure out which change was the culprit.

Update 2010-08:

- **Readable** : This can be considered part of Professional - however it can't be stressed enough. An acid test would be to find someone who isn't part of your team and asking him/her to figure out the behavior under test within a couple of minutes. Tests need to be maintained just like production code - so make it easy to read even if it takes more effort. Tests should be symmetric (follow a pattern) and concise (test one behavior at a time). Use a consistent naming convention (e.g. the TestDox style). Avoid cluttering the test with "incidental details".. become a minimalist.

Apart from these, most of the others are guidelines that cut down on low-benefit work: e.g. 'Don't test code that you don't own' (e.g. third-party DLLs). Don't go about testing getters and setters. Keep an eye on cost-to-benefit ratio or defect probability.

Share

edited Jan 4, 2017 at 16:39

George Stocker

57.9k ● 29 ● 180 ● 238

answered Sep 15, 2008 at 4:50

Gishu

137k ● 47 ● 226 ● 311

We may disagree on the use of Mocks, but this was a very nice writeup of unit testing best practices. – Justin Standard Sep 15, 2008 at 4:54

I'll bump this one up as an answer then because I find the "A TRIP" acronym useful. – Spoike Sep 15, 2008 at 9:19

3 I agree for the most part, but would like to point out that there is a benefit to testing code that you don't own... You're testing that it meets your requirements. How else can you be confident that an upgrade isn't going to break your systems? (But of course, do keep the cost/benefit ratios in mind when doing so.) – Disillusioned Jan 6, 2010 at 11:22

@Craig - I believe you're referring to (interface level) regression tests (or learner tests in some cases), which document behavior on which you depend on. I'd not write 'unit' tests for third party code because a. the vendor knows more about that code than me b. The vendor is not bound to preserve any specific implementation. I do not control change to that codebase and I do not want to spend my time fixing broken tests with an upgrade. So I'd rather code up some high level regression tests for behavior that I use (and want to be notified when broken) – Gishu Jan 10, 2010 at 6:15

@Gishu: Yes, absolutely! The tests must only be done at the interface level; and in fact, you should at most test the features you actually use. Furthermore, when choosing with what to write these tests; I've found the simple straight-forward 'unit' testing frameworks usually fit the bill perfectly. – Disillusioned Jan 11, 2010 at 8:10

42

votes

1. **Don't write ginormous tests.** As the 'unit' in 'unit test' suggests, make each one as *atomic* and *isolated* as possible. If you must, create preconditions using mock objects, rather than recreating too much of the typical user environment manually.
2. **Don't test things that obviously work.** Avoid testing the classes from a third-party vendor, especially the

one supplying the core APIs of the framework you code in. E.g., don't test adding an item to the vendor's Hashtable class.

3. **Consider using a code coverage tool** such as NCover to help discover edge cases you have yet to test.
4. **Try writing the test *before* the implementation.** Think of the test as more of a specification that your implementation will adhere to. Cf. also behavior-driven development, a more specific branch of test-driven development.
5. **Be consistent.** If you only write tests for some of your code, it's hardly useful. If you work in a team, and some or all of the others don't write tests, it's not very useful either. Convince yourself and everyone else of the importance (and *time-saving* properties) of testing, or don't bother.

Share

answered Sep 14, 2008 at 15:36

Sören Kuklau

19.9k ● 8 ● 55 ● 90

1 Good answer. But it isn't that bad if you don't unit test for everything in a delivery. Sure it's preferable, but there needs to be a balance and pragmatism. Re: getting your colleagues on board; sometimes you just need to do it to demonstrate value and as a reference point. – Martin Clarke Sep 14, 2008 at 16:33

1 I agree. However, in the long run, you need to be able to rely on tests being there, i.e. able to assume that common pitfalls

will be caught by them. Otherwise, the benefits are massively diminished. – Sören Kuklau Sep 14, 2008 at 16:48

2 "If you only write tests for some of your code, it's hardly useful." Is this really the case? I've got projects with 20% code coverage (crucial / prone to fail areas) and they helped me massively, and projects are just fine as well. – dr. evil Dec 15, 2008 at 9:30

1 I agree with Slough. Even if there are only a few tests, given they are well written and isolated enough, they'll help out tremendously. – Spoike Jan 1, 2009 at 23:40

---

41 votes

Most of the answers here seem to address unit testing best practices in general (when, where, why and what), rather than actually writing the tests themselves (how). Since the question seemed pretty specific on the "how" part, I thought I'd post this, taken from a "brown bag" presentation that I conducted at my company.

Womp's 5 Laws of Writing Tests:

---

1. Use long, descriptive test method names.

```
    -
Map_DefaultConstructorShouldCreateEmptyGisMap()
    -
ShouldAlwaysDelegateXMLCorrectlyToTheCustomHandlers()
    -
Dog_Object_Should_Eat_Homework_Object_When_Hungry()
```

2. Write your tests in an Arrange/Act/Assert style.

- While this organizational strategy has been around for a while and called many things, the introduction of the "AAA" acronym recently has been a great way to get this across. Making all your tests consistent with AAA style makes them easy to read and maintain.

3. Always provide a failure message with your Asserts.

```
Assert.That(x == 2 && y == 2, "An incorrect number
of begin/end element
processing events was raised by the
XElementSerializer");
```

- A simple yet rewarding practice that makes it obvious in your runner application what has failed. If you don't provide a message, you'll usually get something like "Expected true, was false" in your failure output, which makes you have to actually go read the test to find out what's wrong.

4. Comment the reason for the test – what’s the business assumption?

```
  /// A layer cannot be constructed with a null
gisLayer, as every function
  /// in the Layer class assumes that a valid
gisLayer is present.
  [Test]
  public void
ShouldNotAllowConstructionWithANullGisLayer()
  {
  }
```

- This may seem obvious, but this practice will protect the integrity of your tests from people who don't understand the reason behind the test in the first place. I've seen many tests get removed or modified that were perfectly fine, simply because the person didn't understand the assumptions that the test was verifying.
- If the test is trivial or the method name is sufficiently descriptive, it can be permissible to leave the comment off.

5. Every test must always revert the state of any resource it touches

- Use mocks where possible to avoid dealing with real resources.
- Cleanup must be done at the test level. Tests must not have any reliance on order of execution.

Share

answered May 6, 2009 at 20:18

womp

117k ● 26 ● 238 ● 271

2 +1 because of point 1, 2 and 5 are important. 3 and 4 seem rather excessive for unit tests, if you are already using descriptive test method names, but I do recommend documentation of tests if they are large in scope (functional or acceptance testing). – Spoike May 7, 2009 at 6:35

17 votes

Keep these goals in mind (adapted from the book xUnit Test Patterns by Meszaros)

- Tests should reduce risk, not introduce it.
- Tests should be easy to run.
- Tests should be easy to maintain as the system evolves around them

Some things to make this easier:

- Tests should only fail because of one reason.
- Tests should only test one thing
- Minimize test dependencies (no dependencies on databases, files, ui etc.)

Don't forget that you can do intergration testing with your xUnit framework too **but keep intergration tests and unit tests separate**

Share

edited Sep 14, 2008 at 16:40

answered Sep 14, 2008 at 15:31

Mendelt

37.5k ● 6 ● 75 ● 97

I guess you meant you've adapted from the book "xUnit Test Patterns" by Gerard Meszaros. xunitpatterns.com – Spoike Sep 14, 2008 at 16:22

Excellent points. Unit tests can be very useful but it's very important to avoid falling into the trap of having complex, interdependent unit tests which create a huge tax for any attempts to change the system. – Wedge Sep 14, 2008 at 20:25

9 votes

Tests should be isolated. One test should not depend on another. Even further, a test should not rely on external systems. In other words, test *your* code, not the code your code depends on.You can test those interactions as part of your integration or functional tests.

Share

answered Sep 14, 2008 at 16:51

Haacked

59k ● 14 ● 91 ● 115

9 votes

Some properties of great unit tests:

- When a test fails, it should be immediately obvious where the problem lies. If you have to use the debugger to track down the problem, then your tests aren't granular enough. Having exactly one assertion per test helps here.
- When you refactor, no tests should fail.
- Tests should run so fast that you never hesitate to run them.
- All tests should pass always; no non-deterministic results.

- Unit tests should be well-factored, just like your production code.

@Alotor: If you're suggesting that a library should only have unit tests at its external API, I disagree. I want unit tests for each class, including classes that I don't expose to external callers. (However, if I feel the need to write tests for private methods, then I need to refactor.)

EDIT: There was a comment about duplication caused by "one assertion per test". Specifically, if you have some code to set up a scenario, and then want to make multiple assertions about it, but only have one assertion per test, you might duplication the setup across multiple tests.

I don't take that approach. Instead, I use test fixtures *per scenario*. Here's a rough example:

```
[TestFixture]
public class StackTests
{
    [TestFixture]
    public class EmptyTests
    {
        Stack<int> _stack;

        [TestSetup]
        public void TestSetup()
        {
            _stack = new Stack<int>();
        }

        [TestMethod]
        [ExpectedException (typeof(Exception))]
        public void PopFails()
        {
```

```
            _stack.Pop();
        }

        [TestMethod]
        public void IsEmpty()
        {
            Assert(_stack.IsEmpty());
        }
    }

    [TestFixture]
    public class PushedOneTests
    {
        Stack<int> _stack;

        [TestSetup]
        public void TestSetup()
        {
            _stack = new Stack<int>();
```

Share

edited May 23, 2017 at 12:17

Community Bot

1 ● 1

answered Sep 14, 2008 at 16:22

Jay Bazuzi

46.4k ● 16 ● 115 ● 170

I disagree about only one assertion per test. The more assertions you have in a test the less cut and paste test cases you will have. I believe a test case should focus on a scenario or code path and the assertions should stem from all the assumptions and requirements to fulfill that scenario. – Lucas B Oct 1, 2009 at 14:12

I think we agree that DRY applies to unit tests. As I said, "Unit tests should be well-factored". However, there are multiple ways to resolve the duplication. One, as you mention, is to have a unit test that first invokes the code under test, and then asserts multiple times. An alternative is to create a new "test

fixture" for the scenario, which invokes the code under test during an Initialize/Setup step, and then has a series of unit tests which simply assert. – Jay Bazuzi Oct 1, 2009 at 15:52

My rule of thumb is, if you're using copy-paste, you're doing something wrong. One of my favorite sayings is "Copy-paste is not a design pattern." I also agree one assertion per unit test is generally a good idea, but I don't always insist on it. I like the more general "test one thing per unit test". Though that usually does translate to one assert per unit test. – Jon Turner Aug 24, 2011 at 20:08

---

7 votes

What you're after is delineation of the behaviours of the class under test.

1. Verification of expected behaviours.
2. Verification of error cases.
3. Coverage of all code paths within the class.
4. Exercising all member functions within the class.

The basic intent is increase your confidence in the behaviour of the class.

This is especially useful when looking at refactoring your code. Martin Fowler has an interesting article regarding testing over at his web site.

HTH.

cheers,

Rob

Share

edited Jun 20, 2020 at 9:12

Community Bot

1 • 1

answered Sep 14, 2008 at 15:37

Rob Wells

37k • 13 • 84 • 147

Rob - mechanical this is good, but it misses the intent. Why did you do all this? Thinking this way may help others down the path of TDD. – Mark Levison Feb 24, 2011 at 16:50

7

votes

Test should originally fail. Then you should write the code that makes them pass, otherwise you run the risk of writing a test that is bugged and always passes.

Share

answered Sep 14, 2008 at 15:38

Quibblesome

25.4k • 10 • 62 • 104

@Rismo Not exclusive per se. By definition what Quarrelsome wrote here is exclusive to "Test First" methodology, which is part of TDD. TDD also takes refactoring into account. The most "smarty pants" definition I've read is that TDD = Test First + Refactor. – Spoike Sep 14, 2008 at 18:05

Yeah it doesn't have to be TDD, just make sure your test fails first off. Then wire in the rest afterwards. This does most commonly occur when doing TDD but you can apply it when not using TDD as well. – Quibblesome Sep 15, 2008 at 12:02

6

votes

I like the Right BICEP acronym from the aforementioned Pragmatic Unit Testing book:

- **Right**: Are the results **right**?
- **B**: Are all the **b**oundary conditions correct?
- **I**: Can we check **i**nverse relationships?
- **C**: Can we **c**ross-check results using other means?
- **E**: Can we force **e**rror conditions to happen?
- **P**: Are **p**erformance characteristics within bounds?

Personally I feel that you can get pretty far by checking that you get the right results (1+1 should return 2 in a addition function), trying out all the boundary conditions you can think of (such as using two numbers of which the sum is greater than the integer max value in the add function) and forcing error conditions such as network failures.

Share

answered Sep 15, 2008 at 11:23

Peter Evjan

2,433 ● 3 ● 32 ● 50

6

votes

Good tests need to be maintainable.

I haven't quite figured out how to do this for complex environments.

All the textbooks start to come unglued as your code base starts reaching into the hundreds of 1000's or millions of lines of code.

- Team interactions explode
- number of test cases explode
- interactions between components explodes.
- time to build all the unittests becomes a significant part of the build time
- an API change can ripple to hundreds of test cases. Even though the production code change was easy.
- the number of events required to sequence processes into the right state increases which in turn increases test execution time.

Good architecture can control some of interaction explosion, but inevitably as systems become more complex the automated testing system grows with it.

This is where you start having to deal with trade-offs:

- only test external API otherwise refactoring internals results in significant test case rework.
- setup and teardown of each test gets more complicated as an encapsulated subsystem retains more state.
- nightly compilation and automated test execution grows to hours.
- increased compilation and execution times means designers don't or won't run all the tests
- to reduce test execution times you consider sequencing tests to take reduce set up and teardown

You also need to decide:

where do you store test cases in your code base?

- how do you document your test cases?
- can test fixtures be re-used to save test case maintenance?
- what happens when a nightly test case execution fails? Who does the triage?
- How do you maintain the mock objects? If you have 20 modules all using their own flavor of a mock logging API, changing the API ripples quickly. Not only do the test cases change but the 20 mock objects change. Those 20 modules were written over several years by many different teams. Its a classic re-use problem.
- individuals and their teams understand the value of automated tests they just don't like how the other team is doing it. :-)

I could go on forever, but my point is that:

Tests need to be maintainable.

Share

answered Apr 24, 2009 at 18:46

DanM

2,391 ● 2 ● 18 ● 14

5

votes

I covered these principles a while back in [This MSDN Magazine article] which I think is important for any

developer to read.

The way I define "good" unit tests, is if they posses the following three properties:

- They are readable (naming, asserts, variables, length, complexity..)
- They are Maintainable (no logic, not over specified, state-based, refactored..)
- They are trust-worthy (test the right thing, isolated, not integration tests..)

Share

answered Sep 28, 2008 at 19:35

RoyOsherove

3,269 ● 2 ● 31 ● 26

> Roy, I wholeheartedly agree. These things are so much more important than edge case coverage. – Matt Hinze Feb 4, 2009 at 3:11

4

votes

- Unit Testing just tests the external API of your Unit, you shouldn't test internal behaviour.
- Each test of a TestCase should test one (and only one) method inside this API.
    - Aditional Test Cases should be included for failure cases.
- Test the coverage of your tests: Once a unit it's tested, the 100% of the lines inside this unit should had been executed.

Share

answered Sep 14, 2008 at 15:40

Alotor

7,477 ● 12 ● 39 ● 36

2

votes

Jay Fields has a lot of good advices about writing unit tests and there is a post where he summarize the most important advices. There you will read that you should critically think about your context and judge if the advice is worth to you. You get a ton of amazing answers here, but is up to you decide which is best for your context. Try them and just refactoring if it smells bad to you.

Kind Regards

Share

answered Sep 15, 2008 at 3:52

marcospereira

12.2k ● 3 ● 48 ● 53

1

vote

Never assume that a trivial 2 line method will work. Writing a quick unit test is the only way to prevent the missing null test, misplaced minus sign and/or subtle scoping error from biting you, inevitably when you have even less time to deal with it than now.

Share

answered Sep 14, 2008 at 15:52

Joel in Gö

7,530 ● 9 ● 50 ● 77

1

vote

I second the "A TRIP" answer, except that **tests SHOULD rely on each other!!!**

Why?

DRY - Dont Repeat Yourself - applies to testing as well! Test dependencies can help to 1) save setup time, 2) save fixture resources, and 3) pinpoint to failures. Of course, only given that your testing framework supports first-class dependencies. Otherwise, I admit, they are bad.

Follow up
http://www.iam.unibe.ch/~scg/Research/JExample/

Share

answered Oct 7, 2008 at 21:44

akuhn
27.8k ● 2 ● 78 ● 92

I do agree with you. TestNG is another framework in which dependencies are allowed easily. – Davide Oct 15, 2008 at 16:25

0

votes

Often unit tests are based on mock object or mock data. I like to write three kind of unit tests:

- "transient" unit tests: they create their own mock objects/data and test their function with it, but destroy everything and leave no trace (like no data in a test database)

- "persistent" unit test: they test functions within your code creating objects/data that will be needed by more advanced function later on for their own unit test (avoiding for those advanced function to recreate every time their own set of mock objects/data)
- "persistent-based" unit tests: unit tests using mock objects/data that are already there (because created in another unit test session) by the persistent unit tests.

The point is to avoid to replay *everything* in order to be able to test every functions.

- I run the third kind very often because all mock objects/data are already there.
- I run the second kind whenever my model change.
- I run the first one to check the very basic functions once in a while, to check to basic regressions.

Share

answered Sep 14, 2008 at 21:04

VonC

1.3m ● 558 ● 4.7k ● 5.6k

0

votes

Think about the 2 types of testing and treat them differently - functional testing and performance testing.

Use different inputs and metrics for each. You may need to use different software for each type of test.

Share

answered Sep 14, 2008 at 21:34

Techboy
4,404 ● 5 ● 37 ● 46

0
votes

I use a consistent test naming convention described by Roy Osherove's Unit Test Naming standards Each method in a given test case class has the following naming style MethodUnderTest_Scenario_ExpectedResult.

> The first test name section is the name of the method in the system under test.
>
> Next is the specific scenario that is being tested.
>
> Finally is the results of that scenario.

Each section uses Upper Camel Case and is delimited by a under score.

I have found this useful when I run the test the test are grouped by the name of the method under test. And have a convention allows other developers to understand the test intent.

I also append parameters to the Method name if the method under test have been overloaded.

Share

answered Aug 14, 2010 at 17:41

Yack
1,410 ● 1 ● 11 ● 13