Unit Testing Guidelines

Asked 16 years, 3 months ago Modified 12 years, 8 months ago

Viewed 5k times

23

Does anyone know of where to find unit testing guidelines and recommendations? I'd like to have something which addresses the following types of topics (for example):

- Should tests be in the same project as application logic?
- Should I have test classes to mirror my logic classes or should I have only as many test classes as I feel I need to have?
- How should I name my test classes, methods, and projects (if they go in different projects)
- Should private, protected, and internal methods be tested, or just those that are publicly accessible?
- Should unit and integration tests be separated?
- Is there a **good** reason not to have 100% test coverage?

What am I not asking about that I should be?

An online resource would be best.

unit-testing tdd

Share

Improve this question

Follow

edited Mar 17, 2009 at 6:56

asked Sep 20, 2008 at 2:19

Ian Suttle

3,402 ● 2 ● 25 ● 28

7 Answers

Sorted by: Highest score (default)

21

I would recommend Kent Beck's book on TDD.

Also, you need to go to Martin Fowler's site. He has a lot of good information about testing as well.

We are pretty big on TDD so I will answer the questions in that light.

> Should tests be in the same project as application logic?

Typically we keep our tests in the same solution, but we break tests into seperate DLL's/Projects that mirror the DLL's/Projects they are testing, but maintain namespaces with the tests being in a sub namespace. Example: Common / Common.Tests

> Should I have test classes to mirror my logic classes or should I have only as many test

> classes as I feel I need to have?

Yes, your tests should be created before any classes are created, and by definition you should only test a single unit in isolation. Therefore you should have a test class for each class in your solution.

> How should I name my test classes, methods, and projects (if they go in different projects)

I like to emphasize that behavior is what is being tested so I typically name test classes after the SUT. For example if I had a User class I would name the test class like so:

```
public class UserBehavior
```

Methods should be named to describe the behavior that you expect.

```
public void ShouldBeAbleToSetUserFirstName()
```

Projects can be named however you want but usually you want it to be fairly obvious which project it is testing. See previous answer about project organization.

> Should private, protected, and internal methods be tested, or just those that are publicly

> accessible?

Again you want tests to assert expected behavior as if you were a 3rd party consumer of the objects being tested. If you test internal implementation details then your tests will be brittle. You want your test to give you the freedom to refactor without worrying about breaking existing functionality. If your test know about implementation details then you will have to change your tests if those details change.

> Should unit and integration tests be separated?

Yes, unit tests need to be isolated from acceptance and integration tests. Separation of concerns applies to tests as well.

> Is there a good reason not to have 100% test coverage?

I wouldn't get to hung up on the 100% code coverage thing. 100% code coverage tends to imply some level of quality in the tests, but that is a myth. You can have terrible tests and still get 100% coverage. I would instead rely on a good Test First mentality. If you always write a test before you write a line of code then you will ensure 100% coverage so it becomes a moot point.

In general if you focus on describing the full behavioral scope of the class then you will have nothing to worry

about. If you make code coverage a metric then lazy programmers will simply do just enough to meet that mark and you will still have crappy tests. Instead rely heavily on peer reviews where the tests are reviewed as well.

Share Improve this answer Follow

edited Nov 12, 2010 at 14:44

answered Sep 20, 2008 at 2:26

Josh

44.9k ● 7 ● 104 ● 124

3

It's a good question. We grew our own organically, and I suspect the best way is just that. There's a bit "It depends..." in there.

We put out tests in the same project, in a sub-namespace called "UnitTes"

Our test classes mirror the logic class, in order to simplify keeping track of where the tests are in relation to what they are testing

Classes are named like the logic class they are testing, methods are named for the scenario they are testing.

We only write tests for the public and internal methods (test are in the same project), and aim for 95% coverage of the class.

I prefer not to distinguish between "unit" and "intergation". To much time will be spent trying to figure out which is which...bag that! A test is a test.

100% is too difficult to achieve all the time. We aim for 95%. There's also diminishing returns on how much time it will take to get that final 5% and what it will actually catch.

That's us and what suited out environment and pace. You're milage may vary. Think about your envivonment nad the personalities that are involved.

I look forward to seeing what others have to say on this one!

Share Improve this answer

Follow

answered Sep 20, 2008 at 2:29

Tom Carr

1,439 ● 8 ● 11

3

Josh's answer is right on - just one point of clarification:

The reason I separate unit tests from integration and acceptance tests is speed. I use TDD. I need close to instant feedback about the line of code I just created/modified. I cannot get that if I'm running full suites of integration and/or acceptance tests - tests that hit real disks, real networks, and really slow and unpredictable external systems.

Don't cross the beams. Bad things will happen if you do.

Share Improve this answer

Follow

answered Sep 20, 2008 at 6:55

aridlehoover

3,567 ● 1 ● 29 ● 31

1

I insistently recommend you to read Test Driven Development: By Example and Test-Driven Development: A Practical Guide It's too much questions for single topic

Share Improve this answer

Follow

answered Sep 20, 2008 at 2:23

aku

124k ● 33 ● 176 ● 203

> Thanks. I'm not asking for answers but pointers to known, good resources for such info. I appreciate your feedback.
> – Ian Suttle Sep 20, 2008 at 2:24

1

With respect to your last question, in my experience, the "good" reason for not insisting upon 100% test coverage is that it takes a disproportionate amount of effort to get the last few percentage points, particularly in larger code bases. As such, it's a matter of deciding whether or not it's worth your time once you reach that point of diminishing returns.

Share Improve this answer

Follow

answered Sep 20, 2008 at 2:33

Fhoxh

778 ● 3 ● 17

1

In order:

- No, it's usually best to include them in a seperate project; unless you want to be able to run diagnostics at runtime.
- The ideal is 100% code coverage, which means every line of code in every routine in every class.
- I go with ClassnameTest, ClassnameTest.MethodNameTestnumber
- Everything.
- I'd say yes, as integration tests don't need to be run if unit tests fail.
- Simple properties that just set and get a field don't need to be tested.

Share Improve this answer Follow

edited Sep 20, 2008 at 2:38

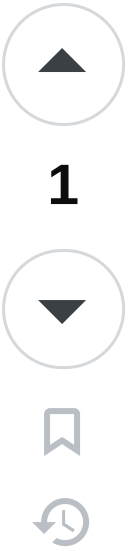
answered Sep 20, 2008 at 2:30

TraumaPony

10.8k ● 13 ● 57 ● 75

Thanks for the info. Funny thing about not testing properties, is I've actually written a test which caught a bad setter. It was setting itself instead of a module level private variable. Seems that might be an infinite loop but it didn't seem to cause an issue. My unit test caught it right away

– Ian Suttle Sep 20, 2008 at 2:33



Should tests be in the same project as application logic?

It depends. There are trade-offs either way.

Keeping it in one project requires extra bandwidth to distribute your project, extra build time and increases the installation footprint, and makes it easier to make the mistake of having production logic that depends on test code.

On the other hand, keeping separate projects can make it harder to write tests involving private methods/classes (depending on programming language), and causes minor administration hassles, such as making setting up a new development environment (e.g. when a new developer joins the project) harder.

How much these different costs matter varies by project, so there's no universal answer.

Should I have test classes to mirror my logic classes or should I have only as many test classes as I feel I need to have?

No.

You should have test classes that allow for well-factored test code (i.e. minimal duplication, clear intent, etc).

The obvious advantage of directly mirroring the logic classes in your test classes is that it makes it easy to find

the tests corresponding to a particular piece of code. There are other ways solve this problem without restricting the flexibility of the test code. Simple naming conventions for test modules and classes is usually enough.

How should I name my test classes, methods, and projects (if they go in different projects)

You should name them so that:

- each test class and test method has a clear purpose, and
- so that someone looking for a particular test (or for tests about a particular unit) can find it easily.

Should private, protected, and internal methods be tested, or just those that are publicly accessible?

Often non-public methods should be tested. It depends on if you get enough confidence from just testing the public interface, or if the unit you really want to be testing is not publically accessible.

Should unit and integration tests be separated?

This depends on your choice of testing framework(s). Do whichever works best with your testing framework(s) and makes it so that:

- both the unit and integration tests relating to a piece of code are easy to find,

- it is easy to run just the unit tests,
- it is easy to run just the integration tests,
- it is easy to run all tests.

Is there a good reason not to have 100% test coverage?

Yes, there is a good reason. Strictly speaking “100% test coverage” means every possible situation in your code is exercised and tested. This is simply impractical for almost any project to achieve.

If you simply take “100% test coverage” to mean that every line of source code is exercised by the test suite at some point, then this is a good goal, but sometimes there are just a couple of lines in awkward places that are hard to reach with automated tests. If the cost of manually verifying that functionality periodically is less than the cost of going through contortions to reach those last five lines, then that is a good reason not to have 100% line coverage.

Rather than a simple rule that you should have 100% line coverage, encourage your developers to discover *any* gaps in your testing, and find ways to fix those gaps, whether or the number of lines “covered” improves. In other words, if you measure lines covered, then you will improve your line coverge — but what you actually want is improved quality. So don't forget that line coverage is just a very crude approximation of quality.

Share Improve this answer

Follow

answered Sep 27, 2008 at 15:55

spiv

3,242 1 20 8

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.