How do you know what to test when writing unit tests? [closed]

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

Viewed 88k times

128

Closed. This question needs to be more focused. It is not currently accepting answers.

Want to improve this question? Update the question so it focuses on one problem only by editing this post.

Closed 8 years ago.

Improve this question

Using C#, I need a class called `User` that has a username, password, active flag, first name, last name, full name, etc.

There should be methods to *authenticate* and *save* a user. Do I just write a test for the methods? And do I even need to worry about testing the properties since they are .Net's getter and setters?

c# unit-testing tdd

Share

Improve this question

Follow

edited Jan 2, 2017 at 20:49

community wiki
4 revs, 4 users 60%
nbro

This post will help with the narrowing the wider question: earnestengineer.blogspot.com/2018/03/… You can take these guidelines to focus your question – Lindsay Morsillo Jun 12, 2018 at 19:36

Keep in mind passwords should not be stored as plaintext. – Mr Anderson Apr 16, 2019 at 15:26

36 Answers

Sorted by: Highest score (default)

1 2 Next

134

Many great responses to this are also on my question: "Beginning TDD - Challenges? Solutions? Recommendations?"

May I also recommend taking a look at my blog post (which was partly inspired by my question), I have got some good feedback on that. Namely:

> **I Don't Know Where to Start?**
>
> - Start afresh. Only think about writing tests when you are writing new code. This can be

re-working of old code, or a completely new feature.

- Start simple. Don’t go running off and trying to get your head round a testing framework as well as being TDD-esque. Debug.Assert works fine. Use it as a starting point. It doesn’t mess with your project or create dependencies.
- Start positive. You are trying to improve your craft, feel good about it. I have seen plenty of developers out there that are happy to stagnate and not try new things to better themselves. You are doing the right thing, remember this and it will help stop you from giving up.
- Start ready for a challenge. It is quite hard to start getting into testing. Expect a challenge, but remember – challenges can be overcome.

Only Test For What You Expect

I had real problems when I first started because I was constantly sat there trying to figure out every possible problem that could occur and then trying to test for it and fix. This is a quick way to a headache. Testing should be a real YAGNI process. If you know there is a problem, then write a test for it. Otherwise, don’t bother.

> **Only Test One Thing**
>
> Each test case should only ever test one thing. If you ever find yourself putting “and” in the test case name, you’re doing something wrong.

I hope this means we can move on from "getters and setters" :)

Share Improve this answer Follow

edited May 23, 2017 at 12:10

community wiki
2 revs
Rob Cooper

2 "If you know there is a problem, then write a test for it. Otherwise, don’t bother." I would disagree with the way this is worded. I was under the impression that unit tests should cover all possible execution paths. – Corin Blaikie Sep 15, 2008 at 15:50

3 While some may advocate such things, I personally don't. A good 90% of my headache came from simply trying to do "everything". I say test for what you expect to happen (so you know you are getting the right values back) but don't try and figure it all out. YAGNI. – Rob Cooper Sep 15, 2008 at 16:17

4 I, too, advocate the "test your bugs" approach. If we all had infinite time and patience, we'd test every possible execution path. But we don't, so you have to spend your effort where it's going to have the greatest effect. – Schwern Oct 18, 2008 at 7:05

64

Test your code, not the language.

A unit test like:

```
Integer i = new Integer(7);
assert (i.instanceOf(integer));
```

is only useful if you are writing a compiler and there is a non-zero chance that your `instanceof` method is not working.

Don't test stuff that you can rely on the language to enforce. In your case, I'd focus on your authenticate and save methods - and I'd write tests that made sure they could handle null values in any or all of those fields gracefully.

Share Improve this answer Follow

edited Jan 30, 2014 at 19:46

community wiki
2 revs, 2 users 90%
Tim Howland

1 Good Point on "Dont test the framework" - Something I got on too when new to this. +1'ed :) – Rob Cooper Sep 15, 2008 at 13:44

This got me into unit testing and it made me very happy

38

We just started to do unit testing. For a long time I knew it would be good to start doing it but I had no idea how to start and more importantly what to test.

Then we had to rewrite an important piece of code in our accounting program. This part was very complex as it involved a lot of different scenarios. The part I'm talking about is a method to pay sales and/or purchase invoices already entered into the accounting system.

I just didn't know how to start coding it, as there were so many different payment options. An invoice could be $100 but the customer only transferred $99. Maybe you have sent sales invoices to a customer but you have also purchased from that customer. So you sold him for $300 but you bought for $100. You can expect your customer to pay you $200 to settle the balance. And what if you sold for $500 but the customer pays you only $250?

So I had a very complex problem to solve with many possibilities that one scenario would work perfectly but would be wrong on an other type of invocie/payment combination.

This is where unit testing came to the rescue.

I started to write (inside the test code) a method to create a list of invoices, both for sales and purchases. Then I wrote a second method to create the actual payment. Normally a user would enter that information through a user interface.

Then I created the first TestMethod, testing a very simple payment of a single invoice without any payment discounts. All the action in the system would happen when a bankpayment would be saved to the database. As you can see I created an invoice, created a payment (a bank transaction) and saved the transaction to disk. In my asserts I put what should be the correct numbers ending up in the Bank transaction and in the linked Invoice. I check for the number of payments, the payment amounts, the discount amount and the balance of the invoice after the transaction.

After the test ran I would go to the database and double check if what I expected was there.

After I wrote the test, I started coding the payment method (part of the BankHeader class). In the coding I only bothered with code to make the first test pass. I did not yet think about the other, more complex, scenarios.

I ran the first test, fixed a small bug until my test would pass.

Then I started to write the second test, this time working with a payment discount. After I wrote the test I modified the payment method to support discounts.

While testing for correctness with a payment discount, I also tested the simple payment. Both tests should pass of course.

Then I worked my way down to the more complex scenarios.

1) Think of a new scenario

2) Write a test for that scenario

3) Run that single test to see if it would pass

4) If it didn't I'd debug and modify the code until it would pass.

5) While modifying code I kept on running all tests

This is how I managed to create my very complex payment method. Without unit testing I did not know how to start coding, the problem seemed overwhelming. With testing I could start with a simple method and extend it step by step with the assurance that the simpler scenarios would still work.

I'm sure that using unit testing saved me a few days (or weeks) of coding and is more or less guaranteeing the correctness of my method.

If I later think of a new scenario, I can just add it to the tests to see if it is working or not. If not I can modify the code but still be sure the other scenarios are still working correctly. This will save days and days in the maintenance and bug fixing phase.

Yes, even tested code can still have bugs if a user does things you did not think of or prevented him

from doing

Below are just some of tests I created to test my payment method.

```
public class TestPayments
{
    InvoiceDiaryHeader invoiceHeader = null;
    InvoiceDiaryDetail invoiceDetail = null;
    BankCashDiaryHeader bankHeader = null;
    BankCashDiaryDetail bankDetail = null;



    public InvoiceDiaryHeader CreateSales(string amoun
invoiceNumber, string date)
    {
        ......
        ......
    }

    public BankCashDiaryHeader CreateMultiplePayments(
invoices, int headerNumber, decimal amount, decimal di
    {
       ......
       ......
       ......
    }


    [TestMethod]
    public void TestSingleSalesPaymentNoDiscount()
    {
        IList<InvoiceDiaryHeader> list = new List<Invo
        list.Add(CreateSales("119", true, 1, "01-09-20
        bankHeader = CreateMultiplePayments(list, 1, 1
        bankHeader.Save();

        Assert.AreEqual(1, bankHeader.BankCashDetails.
        Assert.AreEqual(1, bankHeader.BankCashDetails[
        Assert.AreEqual(119M,
bankHeader.BankCashDetails[0].Payments[0].PaymentAmoun
```

```
        Assert.AreEqual(0M,
bankHeader.BankCashDetails[0].Payments[0].PaymentDisco
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[0].InvoiceHeade
    }

    [TestMethod]
    public void TestSingleSalesPaymentDiscount()
    {
        IList<InvoiceDiaryHeader> list = new List<Invo
        list.Add(CreateSales("119", true, 2, "01-09-20
        bankHeader = CreateMultiplePayments(list, 2, 1
        bankHeader.Save();

        Assert.AreEqual(1, bankHeader.BankCashDetails.
        Assert.AreEqual(1, bankHeader.BankCashDetails[
        Assert.AreEqual(118M,
bankHeader.BankCashDetails[0].Payments[0].PaymentAmoun
        Assert.AreEqual(1M,
bankHeader.BankCashDetails[0].Payments[0].PaymentDisco
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[0].InvoiceHeade
    }

    [TestMethod]
    [ExpectedException(typeof(ApplicationException))]
    public void TestDuplicateInvoiceNumber()
    {
        IList<InvoiceDiaryHeader> list = new List<Invo
        list.Add(CreateSales("100", true, 2, "01-09-20
        list.Add(CreateSales("200", true, 2, "01-09-20

        bankHeader = CreateMultiplePayments(list, 3, 3
        bankHeader.Save();
        Assert.Fail("expected an ApplicationException"
    }

    [TestMethod]
    public void TestMultipleSalesPaymentWithPaymentDis
    {
        IList<InvoiceDiaryHeader> list = new List<Invo
        list.Add(CreateSales("119", true, 11, "01-09-2
        list.Add(CreateSales("400", true, 12, "02-09-2
        list.Add(CreateSales("600", true, 13, "03-09-2
        list.Add(CreateSales("25,40", true, 14, "04-09

        bankHeader = CreateMultiplePayments(list, 5, 1
        bankHeader.Save();

        Assert.AreEqual(1, bankHeader.BankCashDetails.
        Assert.AreEqual(4, bankHeader.BankCashDetails[
        Assert.AreEqual(118.60M,
bankHeader.BankCashDetails[0].Payments[0].PaymentAmoun
        Assert.AreEqual(400,
bankHeader.BankCashDetails[0].Payments[1].PaymentAmoun
        Assert.AreEqual(600,
bankHeader.BankCashDetails[0].Payments[2].PaymentAmoun
        Assert.AreEqual(25.40M,
bankHeader.BankCashDetails[0].Payments[3].PaymentAmoun

        Assert.AreEqual(0.40M,
bankHeader.BankCashDetails[0].Payments[0].PaymentDisco
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[1].PaymentDisco
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[2].PaymentDisco
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[3].PaymentDisco

        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[0].InvoiceHeade
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[1].InvoiceHeade
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[2].InvoiceHeade
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[3].InvoiceHeade
    }

    [TestMethod]
    public void TestSettlement()
    {
        IList<InvoiceDiaryHeader> list = new List<Invo
        list.Add(CreateSales("300", true, 43, "01-09-2
        list.Add(CreateSales("100", false, 6453, "02-0

        bankHeader = CreateMultiplePayments(list, 22,
        bankHeader.Save();
        Assert.AreEqual(1, bankHeader.BankCashDetails.
        Assert.AreEqual(2, bankHeader.BankCashDetails[
        Assert.AreEqual(300,
bankHeader.BankCashDetails[0].Payments[0].PaymentAmoun
        Assert.AreEqual(-100,
bankHeader.BankCashDetails[0].Payments[1].PaymentAmoun
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[0].InvoiceHeade
        Assert.AreEqual(0,
bankHeader.BankCashDetails[0].Payments[1].InvoiceHeade
    }
```

Share Improve this answer Follow

edited Jan 6, 2017 at 0:46

community wiki
2 revs, 2 users 99%
eroijen

1 Found a bug in your unit test! You repeat this line instead of including a 2 in one of them: `Assert.AreEqual(0, bankHeader.BankCashDetails[0].Payments[3].InvoiceHeader.Balance);` – Ryan Peschel Oct 21, 2011 at 23:42

2 You say: "After the test ran I would go to the database and double check if what I expected was there." This is a good example of an integration test between components of your system - not an isolated unit test of a single piece of code. – JTech Mar 27, 2017 at 23:39

2 You also broke the rule of more than one Assert per test. – Steve Jan 4, 2018 at 17:07

13

If they really are trivial, then don't bother testing. Eg, if they are implemented like this;

```
public class User
{
    public string Username { get; set; }
    public string Password { get; set; }
}
```

If, on the other hand, you are doing something clever, (like encrypting and decrypting the password in the getter/setter) then give it a test.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:10

community wiki
Steve Cooper

10

The rule is that you have to test every piece of logic you write. If you implemented some specific functionality in the getters and setters I think they are worth testing. If they only assign values to some private fields, don't bother.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:07

community wiki
Slavo

6

This question seems to be a question of where does one draw the line on what methods get tested and which don't.

The setters and getters for value assignment have been created with consistency and future growth in mind, and foreseeing that some time down the road the setter/getter may evolve into more complex operations. It would make sense to put unit tests of those methods in place, also for the sake of consistency and future growth.

Code reliability, especially while undergoing change to add additional functionality, is the primary goal. I am not aware of anyone ever getting fired for including setters/getters in the testing methodology, but I am certain there exists people who wished they had tested methods which last they were aware or can recall were simple set/get wrappers but that was no longer the case.

Maybe another member of the team expanded the set/get methods to include logic that now needs tested but didn't then create the tests. But now your code is calling these methods and you aren't aware they changed and need in-depth testing, and the testing you do in development and

QA don't trigger the defect, but real business data on the first day of release does trigger it.

The two teammates will now debate over who dropped the ball and failed to put in unit tests when the set/gets morphed to include logic that can fail but isn't covered by a unit test. The teammate that originally wrote the set/gets will have an easier time coming out of this clean if the tests were implemented from day one on the simple set/gets.

My opinion is that a few minutes of "wasted" time covering ALL methods with unit tests, even trivial ones, might save days of headache down the road and loss of money/reputation of the business and loss of someone's job.

And the fact that you did wrap trivial methods with unit tests might be seen by that junior team mate when they change the trivial methods into non-trivial ones and prompt them to update the test, and now nobody is in trouble because the defect was contained from reaching production.

The way we code, and the discipline that can be seen from our code, can help others.

Share Improve this answer Follow

answered Mar 21, 2015 at 12:55

community wiki
Thomas Carlisle

4

Another canonical answer. This, I believe, from Ron Jeffries:

> Only test the code that you want to work.

edited Nov 17, 2016 at 10:53

community wiki
2 revs, 2 users 60%
Liam

3

Testing boilerplate code is a waste of time, but as Slavo says, if you add a side effect to your getters/setters, then you should write a test to accompany that functionality.

If you're doing test-driven development, you should write the contract (eg interface) first, then write the test(s) to exercise that interface which document the expected results/behaviour. **Then** write your methods themselves, without touching the code in your unit tests. Finally, grab a code coverage tool and make sure your tests exercise all the logic paths in your code.

answered Sep 15, 2008 at 13:12

community wiki
warren_s

3

Really trivial code like getters and setters that have no extra behaviour than setting a private field are overkill to test. In 3.0 C# even has some syntactic sugar where the compiler takes care of the private field so you don't have to program that.

I usually write lots of very simple tests verifying behaviour I expect from my classes. Even if it's simple stuff like adding two numbers. I switch a lot between writing a simple test and writing some lines of code. The reason for this is that I then can change around code without being afraid I broke things I didn't think about.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:14

community wiki
Mendelt

Glad you made a good point of the KISS principle.. I often have tests that are literally like 2-3 lines of code, real small, simple tests. Easy to grok and hard to break :) +1'ed
– Rob Cooper Sep 15, 2008 at 13:31

You should test everything. Right now you have getters and setters, but one day you might change them somewhat, maybe to do validation or something else. The

3

tests you write today will be used tomorrow to make sure everything keeps on working as usual. When you write test, you should forget considerations like "right now it's trivial". In an agile or test-driven context you should test assuming future refactoring. Also, did you try putting in really weird values like extremely long strings, or other "bad" content? Well you should... never assume how badly your code can be abused in the future.

Generally I find that writing extensive user tests is on one side, exhausting. On the other side, though it always gives you invaluable insight on how your application should work and helps you throw away easy (and false) assumptions (like: the user name will always be less than 1000 characters in length).

Share Improve this answer Follow

answered Sep 15, 2008 at 14:01

community wiki
Sklivvz

3

For simple modules that may end up in a toolkit, or in an open source type of project, you should test as much as possible including the trivial getters and setters. The thing you want to keep in mind is that generating a unit test as you write a particular module is fairly simple and straight forward. Adding getters and setters is minimal code and can be handled without much thought. However, once your code is placed in a larger system, this extra effort can protect you against changes in the underlying system, such as type changes in a base class. Testing everthing is the best way to have a regression that is complete.

Share Improve this answer Follow

answered Sep 15, 2008 at 14:10

community wiki
Dirigible

2

It doesn't hurt to write unit tests for your getters and setters. Right now, they may just be doing field get/sets under the hood, but in the future you may have validation logic, or inter-property dependencies that need to be tested. It's easier to write it now while you're thinking about it then remembering to retrofit it if that time ever comes.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:08

community wiki
Bob King

well, if your getters/setters need unit tests, there must be some logic associated with them, so that means logic must be written inside them, if they don't have any logic, no unit tests need to be written. – Pop Catalin Sep 15, 2008 at 13:20

2 His point is that logic may be added to them later.
– LegendLength Mar 23, 2009 at 2:36

2

in general, when a method is only defined for certain values, test for values on *and over* the border of what is acceptable. In other words, make sure your method does what it's supposed to do, *but nothing more*. This is important, because when you're going to fail, you want to fail early.

In inheritance hierarchies, make sure to test for LSP compliance.

Testing default getters and setters doesn't seem very useful to me, unless you're planning to do some validation later on.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:14

community wiki
Rik

2

well if you think it can break, write a test for it. I usually don't test setter/getter, but lets says you make one for User.Name, which concatenate first and last name, I would write a test so if someone change the order for last and first name, at least he would know he changed something that was tested.

Share Improve this answer

Follow

answered Sep 15, 2008 at 13:49

community wiki
pmlarocque

2

The canonical answer is "test anything that can possibly break." If you are sure the properties won't break, don't test them.

And once something is found to have broken (you find a bug), obviously it means you need to test it. Write a test to reproduce the bug, watch it fail, then fix the bug, then watch the test pass.

Share Improve this answer

Follow

answered Sep 15, 2008 at 13:51

community wiki
Eric Normand

1

As I understand unit tests in the context of agile development, Mike, yes, you need to test the getters and setters (assuming they're publicly visible). The whole concept of unit testing is to test the software unit, which is a class in this case, as a black box. Since the getters and setters are externally visible you need to test them along with Authenticate and Save.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:08

community wiki
Onorio Catenacci

1

If the Authenticate and Save methods use the properties, then your tests will indirectly touch the properties. As long as the properties are just providing access to data, then explicit testing should not be necessary (unless you are going for 100% coverage).

Share Improve this answer Follow

answered Sep 15, 2008 at 13:08

community wiki
Tom Walker

I would test your getters and setters. Depending on who's writing the code, some people change the meaning of the

1

getter/setter methods. I've seen variable initialization and other validation as part of getter methods. In order to test this sort of thing, you'd want unit tests covering that code explicitly.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:09

community wiki
Peter Bernier

1

Personally I would "test anything that can break" and simple getter (or even better auto properties) will not break. I have never had a simple return statement fail and therefor never have test for them. If the getters have calculation within them or some other form of statements, I would certainly add tests for them.

Personally I use Moq as a mock object framework and then verify that my object calls the surrounding objects the way it should.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:09

community wiki
tronda

1

You have to cover the execution of every method of the class with UT and check the method return value. This includes getters and setters, especially in case the members(properties) are complex classes, which requires large memory allocation during their initialization. Call the setter with some very large string for example (or something with greek symbols) and check the result is correct (not truncated, encoding is good e.t.c.)

In case of simple integers that also applies - what happens if you pass long instead of integer? That's the reason you write UT for :)

Share Improve this answer Follow

answered Sep 15, 2008 at 13:10

community wiki
m_pGladiator

1

I wouldn't test the actual setting of properties. I would be more concerned about how those properties get populated by the consumer, and what they populate them with. With any testing, you have to weigh the risks with the time/cost of testing.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:11

community wiki

Mike Becatti

1

You should test "every non-trivial block of code" using unit tests as far as possible.

If your properties are trivial and its unlikely that someone will introduce a bug in it, then it should be safe to not unit test them.

Your Authenticate() and Save() methods look like good candidates for testing.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:11

community wiki
user7015

1

Ideally, you would have done your unit tests as you were writing the class. This is how you're meant to do it when using Test Driven Development. You add the tests as you implement each function point, making sure that you cover the edge-cases with test too.

Writing the tests afterwards is much more painful, but doable.

Here's what I'd do in your position:

1. Write a basic set of tests that test the core function.

2. Get NCover and run it on your tests. Your test coverage will probably be around 50% at this point.
3. Keep adding tests that cover your edge-cases until you get coverage of around 80%-90%

This should give you a nice working set of unit tests that will act as a good buffer against regressions.

The only problem with this approach is that code has to be *designed* to be testable in this fashion. If you made any coupling mistakes early on, you won't be able to get high coverage very easily.

This is why it is really important to write the tests before you write the code. It forces you to write code that is loosely coupled.

answered Sep 15, 2008 at 13:13

community wiki
Simon Johnson

1

Don't test obviously working (boilerplate) code. So if your setters and getters are just "propertyvalue = value" and "return propertyvalue" it makes no sense to test it.

answered Sep 15, 2008 at 13:13

community wiki
erlando

1

Even get / set can have odd consequences, depending upon how they have been implemented, so they should be treated as methods.

Each test of these will need to specify sets of parameters for the properties, defining both acceptable and unacceptable properties to ensure the calls return / fail in the expected manner.

You also need to be aware of security gotchas, as an example SQL injection, and test for these.

So yes, you do need to worry about testing the properties.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:14

community wiki
CestLaGalere

1

I believe it's silly to test getters & setters when they only make a simple operation. Personally I don't write complex unit tests to cover any usage pattern. I try to write enough tests to ensure I have handled the normal execution behavior and as much error cases I can think of. I will write more unit tests as a response to bug reports. I use unit test to ensure the code meets the requirements and to make future modification easier. I feel a lot more willing to change code when I know that if I break something a test will fail.

Share Improve this answer Follow

answered Sep 15, 2008 at 13:16

community wiki
Andrei Savu

1

I would write a test for anything that you are writing code for that is testable outside of the GUI interface.

Typically, any logic that I write that has any business logic I place inside another tier or business logic layer.

Then writing tests for anything that does something is easy to do.

First pass, write a unit test for each public method in your "Business Logic Layer".

If I had a class like this:

```
public class AccountService
 {
     public void DebitAccount(int accountNumber, do
     {

     }

     public void CreditAccount(int accountNumber, d
        {

        }

        public void CloseAccount(int accountNumber)
        {

        }
    }
```

The first thing I would do before I wrote any code knowing that I had these actions to perform would be to start writing unit tests.

```
    [TestFixture]
     public class AccountServiceTests
     {
         [Test]
         public void DebitAccountTest()
         {

         }

         [Test]
         public void CreditAccountTest()
         {

         }

         [Test]
         public void CloseAccountTest()
         {

         }
     }
```

Write your tests to validate the code you've written to do something. If you iterating over a collection of things, and changing something about each of them, write a test that does the same thing and Assert that actually happened.

There's a lot of other approaches you can take, namely Behavoir Driven Development (BDD), that's more involved and not a great place to start with your unit testing skills.

So, the moral of the story is, test anything that does anything you might be worried about, keep the unit tests testing specific things that are small in size, a lot of tests are good.

Keep your business logic outside of the User Interface layer so that you can easily write tests for them, and you'll be good.

I recommend [TestDriven.Net]() or [ReSharper]() as both easily integrate into Visual Studio.

Share Improve this answer Follow

edited Sep 15, 2008 at 13:25

community wiki
2 revs, 2 users 99%
Code Monkey

1

I would recommend writing multiple tests for your Authenticate and Save methods. In addition to the success case (where all parameters are provided, everything is correctly spelled, etc), it's good to have tests for various failure cases (incorrect or missing parameters, unavailable database connections if applicable, etc). I recommend Pragmatic Unit Testing in C# with NUnit as a reference.

As others have stated, unit tests for getters and setters are overkill, unless there's conditional logic in your getters and setters.

Share Improve this answer Follow

answered Sep 15, 2008 at 14:09

community wiki
Scott Lawrence

1

Whilst it is possible to correctly guess where your code needs testing, I generally think you need metrics to back up this guess. Unit testing in my view goes hand in hand with code-coverage metrics.

Code with lots of tests but a small coverage hasn't been well tested. That said, code with 100% coverage but not testing the boundry and error cases is also not great.

You want a balance between high coverage (90% minimum) and variable input data.

Remember to test for "garbage in"!

Also, a unit-test is not a unit-test unless it checks for a failure. Unit-tests that don't have asserts or are marked with known exceptions will simply test that the code doesn't die when run!

You need to design your tests so that they always report failures or unexpected/unwanted data!

Share Improve this answer Follow

answered Sep 15, 2008 at 14:51

community wiki
Ray Hayes

1

It makes our code better... period!

One thing us software developers forget about when doing test driven development is the purpose behind our actions. If a unit test is being written after the production code is already in place, the value of the test goes way down (but is not completely lost).

In the true spirit for unit testing, these tests are **not** primarily there to "test" more of our code; or to get 90%-100% better code coverage. These are all **fringe benefits** of writing the tests first. The big payoff is that our production code ends be be written much better due to the natural process of TDD.

To help better communicate this idea, the following may be helpful in reading:

The Flawed Theory of Unit Tests
Purposeful Software Development

If we feel that the act of writing **more unit tests** is what helps us gain a higher quality product, then we may be suffering from a Cargo Cult of Test Driven Development.

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12

community wiki
2 revs
Scott Saad

I disagree with the assertion that unit tests don't have value after production code is already in place. Such assertions don't account for their utility in replicating error conditions found in production, or in the understanding of code inherited from a previous developer or team. – Scott Lawrence Sep 15, 2008 at 15:25

I may have come across incorrectly. I didn't mean that unit tests don't have value after production code is in place. However, their value goes down. The biggest benefit to unit testing comes from the inherent magic that occurs when we let them drive our production development. – Scott Saad Sep 16, 2008 at 4:20