How do you unit test a unit test? [closed]

Asked 16 years, 1 month ago Modified 14 years, 7 months ago

Viewed 5k times

89

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 7 years ago.

Improve this question

I was watching Rob Connerys webcasts on the MVCStoreFront App, and I noticed he was unit testing even the most mundane things, things like:

```
public Decimal DiscountPrice
{
   get
   {
       return this.Price - this.Discount;
   }
}
```

Would have a test like:

```
[TestMethod]
public void Test_DiscountPrice
{
    Product p = new Product();
    p.Price = 100;
    p.Discount = 20;
    Assert.IsEqual(p.DiscountPrice,80);
}
```

While, I am all for unit testing, I sometimes wonder if this form of test first development is really beneficial, for example, in a real process, you have 3-4 layers above your code (Business Request, Requirements Document, Architecture Document), where the actual defined business rule (Discount Price is Price - Discount) could be misdefined.

If that's the situation, your unit test means nothing to you.

Additionally, your unit test is another point of failure:

```
[TestMethod]
public void Test_DiscountPrice
{
    Product p = new Product();
    p.Price = 100;
    p.Discount = 20;
    Assert.IsEqual(p.DiscountPrice,90);
}
```

Now the test is flawed. Obviously in a simple test, it's no big deal, but say we were testing a complicated business rule. What do we gain here?

Fast forward two years into the application's life, when maintenance developers are maintaining it. Now the business changes its rule, and the test breaks again, some rookie developer then fixes the test incorrectly...we now have another point of failure.

All I see is more possible points of failure, with no real beneficial return, if the discount price is wrong, the test team will still find the issue, how did unit testing save any work?

What am I missing here? Please teach me to love TDD, as I'm having a hard time accepting it as useful so far. I want too, because I want to stay progressive, but it just doesn't make sense to me.

EDIT: A couple people keep mentioned that testing helps enforce the spec. It has been my experience that the spec has been wrong as well, more often than not, but maybe I'm doomed to work in an organization where the specs are written by people who shouldn't be writing specs.

tdd agile test-first

edited Oct 15, 2009 at 10:42

rafek

5,484 ● 13 ● 60 ● 72

asked Oct 28, 2008 at 18:37

FlySwat
175k ● 75 ● 248 ● 314

5 in many cases the unit test *is* the spec, and the documentation, too! – Steven A. Lowe Oct 28, 2008 at 18:59

32 ...and then unit test the unit test of the unit test... but what about the unit^4 test and the unit^5 test... aaaaaaaaahhhhhhhhh! – dacracot Oct 28, 2008 at 19:06

14 No amount of any kind of testing will save you from a wrong specification. – Bill the Lizard Oct 28, 2008 at 22:52

4 Did anyone else just hear what sounded like one hand clapping? – gnovice Jun 2, 2009 at 17:13

9 I think the appropriate quote is "It's just turtles all the way down." – Quinn Taylor Aug 4, 2009 at 6:53

17 Answers

Sorted by: Highest score (default)

63

First, testing is like security -- you can never be 100% sure you've got it, but each layer adds more confidence and a framework for more easily fixing the problems that remain.

Second, you can break tests into subroutines which themselves can then be tested. When you have 20 similar tests, making a (tested) subroutine means your main test is 20 simple invocations of the subroutine which is much more likely to be correct.

Third, some would argue that TDD addresses this concern. That is, if you just write 20 tests and they pass,

you're not completely confident that they are actually testing anything. But if each test you wrote initially *failed*, and then you fixed it, then you're much more confident that it's really testing your code. IMHO this back-and-forth takes more time than it's worth, but it is a process that tries to address your concern.

Share Improve this answer

Follow

answered Oct 28, 2008 at 18:38

Jason Cohen

83k ● 26 ● 110 ● 114

2 To play devils advocate, I see additional layers as more possible points of failure, that doesn't increase confidence for me. In my real job, I work with many teams over a highly distributed SOA enterprise. Each of those teams could jeopardize the project if their layer fails. – FlySwat Oct 28, 2008 at 18:41

2 That is why you are using mock objects to test each layer separately. – Toon Krijthe Oct 28, 2008 at 18:48

10 Great, so my test will pass using IMockedComplexObject but when I actually use a ComplexObject in the real world, it fails...I've gained nothing again. – FlySwat Oct 28, 2008 at 18:49

16 @Jonathan -- no, you've gained confidence that your code works -- assuming that you developed to the interface of ComplexObject and adequately tested against that interface. At worst, you've gained the knowledge that your understanding of ComplexObject wasn't what you anticipated. – tvanfosson Oct 28, 2008 at 18:53

9 @FlySwat: In response to your comment about IMockedComplexObject, I quote Cwash's comment on another answer: "You don't mock what you're trying to test,

you mock what you're not trying to test." – Brian Sep 23, 2009 at 13:39

39

A test being wrong is unlikely to break your production code. At least, not any worse than having no test at all. So it's not a "point of failure": the tests don't have to be correct in order for the product to actually work. They might have to be correct before it's signed off as working, but the process of fixing any broken tests does not endanger your implementation code.

You can think of tests, even trivial tests like these, as being a second opinion what the code is supposed to do. One opinion is the test, the other is the implementation. If they don't agree, then you know you have a problem and you look closer.

It's also useful if someone in future wants to implement the same interface from scratch. They shouldn't have to read the first implementation in order to know what Discount means, and the tests act as an unambiguous back-up to any written description of the interface you may have.

That said, you're trading off time. If there are other tests you could be writing using the time you save skipping these trivial tests, maybe they would be more valuable. It depends on your test setup and the nature of the application, really. If the Discount is important to the app, then you're going to catch any bugs in this method in functional testing anyway. All unit testing does is let you

catch them at the point you're testing this unit, when the location of the error will be immediately obvious, instead of waiting until the app is integrated together and the location of the error *might* be less obvious.

By the way, personally I wouldn't use 100 as the price in the test case (or rather, if I did then I'd add another test with another price). The reason is that someone in future might think that Discount is supposed to be a percentage. One purpose of trivial tests like this is to ensure that mistakes in reading the specification are corrected.

[Concerning the edit: I think it's inevitable that an incorrect specification is a point of failure. If you don't know what the app is supposed to do, then chances are it won't do it. But writing tests to reflect the spec doesn't magnify this problem, it merely fails to solve it. So you aren't adding new points of failure, you're just representing the existing faults in code instead of ~~waffle~~ documentation.]

Share Improve this answer Follow

edited Oct 28, 2008 at 19:14

answered Oct 28, 2008 at 18:55

Steve Jessop

279k ● 40 ● 469 ● 709

4 A wrong test will let broken code into the wild. Thats where the failure is introduced. It provides a false sense of confidence. – FlySwat Oct 28, 2008 at 18:56

9 That's true, but having no test also lets broken code out. The mistake is thinking that if code passes unit testing it must be correct - I was cured of that pretty early in my career. So a broken unit test doesn't let broken code out into the wild, it only lets it out into integration testing. – Steve Jessop Oct 28, 2008 at 19:06

7 Also, even a wrong test can catch broken code, so long as it contains different mistakes from the implementation. That's my point that tests don't absolutely have to be correct, they're there to draw your attention to areas of concern.
– Steve Jessop Oct 28, 2008 at 19:09

22

> All I see is more possible points of failure, with no real beneficial return, if the discount price is wrong, the test team will still find the issue, how did unit testing save any work?

Unit testing isn't really supposed to save work, it's supposed to help you find and prevent bugs. It's *more* work, but it's the right kind of work. It's thinking about your code at the lowest levels of granularity and writing test cases that prove that it works under *expected* conditions, for a given set of inputs. It's isolating variables so you can save *time* by looking in the right place when a bug does present itself. It's *saving* that suite of tests so that you can use them again and again when you have to make a change down the road.

I personally think that most methodologies are not many steps removed from cargo cult software engineering, TDD included, but you don't have to adhere to strict TDD to reap the benefits of unit testing. Keep the good parts and throw out the parts that yield little benefit.

Finally, the answer to your titular question "**How do you unit test a unit test?**" is that you shouldn't have to. Each unit test should be brain-dead simple. Call a method with a specific input and compare it to its expected output. If the specification for a method changes then you can expect that some of the unit tests for that method will need to change as well. That's one of the reasons that you do unit testing at such a low level of granularity, so only *some* of the unit tests have to change. If you find that tests for many different methods are changing for one change in a requirement, then you may not be testing at a fine enough level of granularity.

Share Improve this answer Follow

edited Oct 28, 2008 at 22:51

answered Oct 28, 2008 at 22:36

Bill the Lizard

405k ● 211 ● 572 ● 889

"Call a method with a specific input and compare it to its expected output." but what if the output is a complex type... like an XML document. You can't just "==", you'll have to write specific code compare, and then maybe your compare method could be buggy?? – andy Apr 12, 2010 at 0:28

@andy: You have to test your compare method separately. Once you've thoroughly tested it, you can rely on it working in other tests. – Bill the Lizard Apr 12, 2010 at 0:50

cool, thanks Bill. I'v started working at a new place, and it's my 1st time with Unit Testing. I think in principle it works, and we're using Cruise Control where it's really useful, but large sets of tests seem to suffer the same fate as legacy code... I'm just not sure about it.... – andy Apr 12, 2010 at 5:42

11

Unit tests are there so that your units (methods) do what you expect. Writing the test first forces you to think about what you expect **before** you write the code. Thinking before doing is always a good idea.

Unit tests should reflect the business rules. Granted, there can be errors in the code, but writing the test first allows you to write it from the perspective of the business rule before any code has been written. Writing the test afterwards, I think, is more likely to lead to the error you describe because you **know** how the code implements it and are tempted just to make sure that the implementation is correct -- not that the intent is correct.

Also, unit tests are only one form -- and the lowest, at that -- of tests that you should be writing. Integration tests and acceptance tests should also be written, the latter by the customer, if possible, to make sure that the system operates the way it is expected. If you find errors during this testing, go back and write unit tests (that fail) to test the change in functionality to make it work correctly, then

change your code to make the test pass. Now you have regression tests that capture your bug fixes.

[EDIT]

Another thing that I have found with doing TDD. It almost forces good design by default. This is because highly coupled designs are nearly impossible to unit test in isolation. It doesn't take very long using TDD to figure out that using interfaces, inversion of control, and dependency injection -- all patterns that will improve your design and reduce coupling -- are really important for testable code.

edited Oct 28, 2008 at 19:02

answered Oct 28, 2008 at 18:48

tvanfosson

532k ● 102 ● 699 ● 798

Perhaps this is where my problem lies. I can visual the algorithm for a business rule a low easier than I can visualize the result, so I have no problem implementing the code itself, but see mocking the rule as redundant. Maybe its just how I think. – FlySwat Oct 28, 2008 at 18:52

That's exactly what you do in a unit test. Break that algorithm down into pieces and check each piece. Typically, I find that my code writes itself because I've already written the expectation in my unit test. – tvanfosson Oct 28, 2008 at 18:56

Mock is an overloaded term in the testing space. You don't mock what you're trying to test, you mock what you're not trying to test... When you're writing the test for your business rule you create code that invokes it - it's not mocking it at all. – cwash Jun 3, 2009 at 17:35

@cwash -- I'm not sure how your comment applies to my answer. I didn't mention mocking... and I agree with your observation. – tvanfosson Jun 3, 2009 at 17:41

@tvanfosson - my last comment was in response to @FlySwat "...mocking the rule as redundant." Sorry I forgot specify. – cwash Jun 3, 2009 at 19:51

10

How does one test a test? Mutation testing is a valuable technique that I have personally used to surprisingly good effect. Read the linked article for more details, and links to even more academic references, but in general it "tests your tests" by modifying your source code (changing "x += 1" to "x -= 1" for example) and then rerunning your tests, ensuring that at least one test fails. Any mutations that don't cause test failures are flagged for later investigation.

You'd be surprised at how you can have 100% line and branch coverage with a set of tests that look comprehensive, and yet you can fundamentally change or even comment out a line in your source without any of the tests complaining. Often this comes down to not testing with the right inputs to cover all boundary cases, sometimes it's more subtle, but in all cases I was impressed with how much came out of it.

Share Improve this answer

Follow

answered Jan 8, 2009 at 15:22

Andrzej Doyle

104k ● 33 ● 191 ● 231

1 +1 interesting concept that I had not yet heard about – Wim Coenen May 10, 2009 at 23:31

9

When applying Test-Driven Development (TDD), one begins with a **failing** test. This step, that might seem unecessary, actually is here to verify the unit test is testing something. Indeed, if the test never fails, it brings no value and worse, leads to wrong confidence as you'll rely on a positive result that is not proving anything.

When following this process strictly, all "units" are protected by the safety net the unit tests are making, even the most mundane.

```
Assert.IsEqual(p.DiscountPrice,90);
```

There is no reason the test evolves in that direction - or I'm missing something in your reasoning. When the price is 100 and the discount 20, the discount price is 80. This is like an invariant.

Now imagine your software needs to support another kind of discount based on percentage, perhaps depending on the volume bought, your Product::DiscountPrice() method may become more complicated. And it is possible that introducing those changes breaks the simple discount

rule we had initially. Then you'll see the value of this test which will detect the regression immediately.

Red - Green - Refactor - this is to remember the essence of the TDD process.

Red refers to JUnit red bar when a tests fails.

Green is the color of JUnit progress bar when all tests pass.

Refactor under green condition: remove any dupliation, improve readability.

Now to address your point about the "3-4 layers above the code", this is true in a traditional (waterfall-like) process, not when the development process is agile. And agile is the world where TDD is coming from ; TDD is the cornerstone of eXtreme Programming.

Agile is about direct communication rather than thrown-over-the-wall requirement documents.

Share Improve this answer Follow

edited Sep 8, 2009 at 6:11

answered Oct 30, 2008 at 8:33

philant

35.7k ● 11 ● 73 ● 113

8

Unit testing works very similar to double entry book keeping. You state the same thing (business rule) in two quite different ways (as programmed rules in your production code, and as simple, representative examples in your tests). It's very unlikely that you make *the same* mistake in both, so if they both agree with each other, it's rather unlikely that you got it wrong.

How is testing going to be worth the effort? In my experience in at least four ways, at least when doing test driven development:

- it helps you come up with a well decoupled design. You can only unit test code that is well decoupled;
- it helps you determine when you are done. Having to specify the needed behavior in tests helps to not build functionality that you don't actually need, and determine when the functionality is complete;
- it gives you a safety net for refactorings, which makes the code much more amenable to changes; and
- it saves you a lot of debugging time, which is horribly costly (I've heard estimates that traditionally, developers spend up to 80% of their time debugging).

Share Improve this answer

Follow

answered Oct 31, 2008 at 23:06

Ilja Preuß

2,421 ● 17 ● 15

> While, I am all for unit testing, I sometimes wonder if this form of test first development is really beneficial...

Small, trivial tests like this can be the "canary in the coalmine" for your codebase, alerting of danger before it's too late. The trivial tests are useful to keep around because they help you get the interactions right.

For example, think about a trivial test put in place to probe how to use an API you're unfamiliar with. If that test has any relevance to what you're doing in the code that uses the API "for real" it's useful to keep that test around. When the API releases a new version and you need to upgrade. You now have your assumptions about how you expect the API to behave recorded in an executable format that you can use to catch regressions.

> ...[I]n a real process, you have 3-4 layers above your code (Business Request, Requirements Document, Architecture Document), where the actual defined business rule (Discount Price is Price - Discount) could be misdefined. If that's the situation, your unit test means nothing to you.

If you've been coding for years without writing tests it may not be immediately obvious to you that there is any value. But if you are of the mindset that the best way to work is "release early, release often" or "agile" in that you want

the ability to deploy rapidly/continuously, then your test definitely means something. The only way to do this is by legitimizing every change you make to the code with a test. No matter how small the test, once you have a green test suite you're theoretically OK to deploy. See also "continuous production" and "perpetual beta."

You don't have to be "test first" to be of this mindset, either, but that generally is the most efficient way to get there. When you do TDD, you lock yourself into small two to three minute Red Green Refactor cycle. At no point are you not able to stop and leave and have a complete mess on your hands that will take an hour to debug and put back together.

> Additionally, your unit test is another point of failure...

A successful test is one that demonstrates a failure in the system. A failing test will alert you to an error in the logic of the test or in the logic of your system. The goal of your tests is to break your code or prove one scenario works.

If you're writing tests *after* the code, you run the risk of writing a test that is "bad" because in order to see that your test truly works, you need to see it both broken and working. When you're writing tests after the code, this means you have to "spring the trap" and introduce a bug into the code to see the test fail. Most developers are not only uneasy about this, but would argue it is a waste of time.

> What do we gain here?

There is definitely a benefit to doing things this way. Michael Feathers defines "legacy code" as "untested code." When you take this approach, you legitimize every change you make to your codebase. It's more rigorous than not using tests, but when it comes to maintaining a large codebase, it pays for itself.

Speaking of Feathers, there are two great resources you should check out in regard to this:

- Working Effectively with Legacy Code
- Brownfield Application Development in .NET

Both of these explain how to work these types of practices and disciplines into projects that aren't "Greenfield." They provide techniques for writing tests around tightly coupled components, hard wired dependencies, and things that you don't necessarily have control over. It's all about finding "seams" and testing around those.

> [I]f the discount price is wrong, the test team will still find the issue, how did unit testing save any work?

Habits like these are like an investment. Returns aren't immediate; they build up over time. The alternative to not testing is essentially taking on debt of not being able to

catch regressions, introduce code without fear of integration errors, or drive design decisions. The beauty is you legitimize every change introduced into your codebase.

> What am I missing here? Please teach me to love TDD, as I'm having a hard time accepting it as useful so far. I want too, because I want to stay progressive, but it just doesn't make sense to me.

I look at it as a professional responsibility. It's an ideal to strive toward. But it is very hard to follow and tedious. If you care about it, and feel you shouldn't produce code that is not tested, you'll be able to find the will power to learn good testing habits. One thing that I do a lot now (as do others) is timebox myself an hour to write code without any tests at all, then have the discipline to throw it away. This may seem wasteful, but it's not really. It's not like that exercise cost a company physical materials. It helped me to understand the problem and how to write code in such a way that it is both of higher quality and testable.

My advice would ultimately be that if you really don't have a desire to be good at it, then don't do it at all. Poor tests that aren't maintained, don't perform well, etc. can be worse than not having any tests. It's hard to learn on your own, and you probably won't love it, but it is going to be next to impossible to learn if you don't have a desire to do

it, or can't see enough value in it to warrant the time investment.

> A couple people keep mentioned that testing helps enforce the spec. It has been my experience that the spec has been wrong as well, more often than not...

A developer's keyboard is where the rubber meets the road. If the spec is wrong and you don't raise the flag on it, then it's highly probable you'll get blamed for it. Or at least your code will. The discipline and rigor involved in testing is difficult to adhere to. It's not at all easy. It takes practice, a lot of learning and a lot of mistakes. But eventually it does pay off. On a fast-paced, quickly changing project, it's the only way you can sleep at night, no matter if it slows you down.

Another thing to think about here is that techniques that are fundamentally the same as testing have been proven to work in the past: "clean room" and "design by contract" both tend to produce the same types of "meta"-code constructs that tests do, and enforce those at different points. None of these techniques are silver bullets, and rigor is going to cost you ultimately in the scope of features you can deliver in terms of time to market. But that's not what it's about. It's about being able to maintain what you do deliver. And that's very important for most projects.

Share Improve this answer Follow

edited Oct 13, 2009 at 18:52

answered Jun 3, 2009 at 5:15

cwash
4,245 ● 5 ● 46 ● 53

5

Most unit tests, test assumptions. In this case, the discount price should be the price minus the discount. If your assumptions are wrong I bet your code is also wrong. And if you make a silly mistake, the test will fail and you will correct it.

If the rules change, the test will fail and that is a good thing. So you have to change the test too in this case.

As a general rule, if a test fails right away (and you don't use test first design), either the test or the code is wrong (or both if you are having a bad day). You use common sense (and possilby the specs) to correct the offending code and rerun the test.

Like Jason said, testing is security. And yes, sometimes they introduce extra work because of faulty tests. But most of the time they are huge time savers. (And you have the perfect opportunity to punish the guy who breaks the test (we are talking rubber chicken)).

Share Improve this answer

Follow

answered Oct 28, 2008 at 18:46

Toon Krijthe

53.4k ● 38 ● 149 ● 202

4

Test everything you can. Even trivial mistakes, like forgetting to convert meters to feet can have very expensive side effects. Write a test, write the code for it to check, get it to pass, move on. Who knows at some point in the future, someone may change the discount code. A test can detect the problem.

Share Improve this answer

Follow

answered Oct 28, 2008 at 18:42

Jim C

4,981 ● 23 ● 25

> That doesn't address any of my thoughts. I understand the basic mantra of TDD...I don't see the benefit. – FlySwat Oct 28, 2008 at 18:43

4

I see unit tests and production code as having a symbiotic relationship. Simply put: one tests the other. And both test the developer.

Share Improve this answer

Follow

answered May 17, 2010 at 11:17

johnsyweb

141k ● 26 ● 194 ● 251

3

Remember that the cost of fixing defects increases (exponentially) as the defects live through the development cycle. Yes, the testing team might catch the defect, but it will (usually) take more work to isolate and

fix the defect from that point than if a unit test had failed, and it will be easier to introduce other defects while fixing it if you don't have unit tests to run.

That's usually easier to see with something more than a trivial example ... and with trivial examples, well, if you somehow mess up the unit test, the person reviewing it will catch the error in the test or the error in the code, or both. (They are being reviewed, right?) As tvanfosson points out, unit testing is just one part of an SQA plan.

In a sense, unit tests are insurance. They're no guarantee that you'll catch every defect, and it may seem at times like you're spending a lot of resources on them, but when they do catch defects that you can fix, you'll be spending a lot less than if you'd had no tests at all and had to fix all defects downstream.

edited May 23, 2017 at 11:52

Community Bot

1 ● 1

answered Oct 28, 2008 at 19:24

Dave DuPlantis

6,543 ● 3 ● 28 ● 30

3

I see your point, but it's clearly overstated.

Your argument is basically: Tests introduce failure. Therefore tests are bad/waste of time.

While that may be true in some cases, it's hardly the majority.

TDD assumes: More Tests = Less Failure.

Tests are **more likely** to catch points of failure than introduce them.

Share Improve this answer

Follow

answered Jul 2, 2009 at 22:04

Ray

3,109 ● 23 ● 29

1

Even more automation can help here ! Yes, writing unit tests can be a lot of work, so use some tools to help you out. Have a look at something like Pex, from Microsoft, if you're using .Net It will automatically create suites of unit tests for you by examining your code. It will come up with tests which give good coverage, trying to cover all paths through your code.

Of course, just by looking at your code it can't know what you were actually trying to do, so it doesn't know if it's correct or not. But, it will generate interesting tests cases for you, and you can then examine them and see if it is behaving as you expect.

If you then go further and write parameterized unit tests (you can think of these as contracts, really) it will generate specific tests cases from these, and this time it can know if something's wrong, because your assertions in your tests will fail.

Share Improve this answer Follow

answered Mar 16, 2009 at 8:08

Glurk

1

I've thought a bit about a good way to respond to this question, and would like to draw a parallel to the scientific method. IMO, you could rephrase this question, "How do you experiment an experiment?"

Experiments verify empirical assumptions (hypotheses) about the physical universe. Unit tests will test assumptions about the state or behavior of the code they call. We can talk about the validity of an experiment, but that's because we know, through numerous other experiments, that something doesn't fit. It doesn't have both *convergent validity* and *empirical evidence*. We don't design a new experiment to test or verify the validity of *an experiment*, but we may design a *completely new experiment*.

So **like experiments**, we don't describe the validity of a unit test based on whether or not it passes a unit test itself. Along with other unit tests, it describes the assumptions we make about the system it is testing. Also, like experiments, we try to remove as much complexity as

we can from what we are testing. *"As simple as possible, but no simpler."*

Unlike experiments, we have a trick up our sleeve to verify our tests are valid other than just convergent validity. We can cleverly introduce a bug we know should be caught by the test, and see if the test does indeed fail. (If only we could do that in the real world, we'd depend much less on this convergent validity thing!) A more efficient way to do this is watch your test fail before implementing it (the red step in *Red, Green, Refactor*).

Share Improve this answer Follow

edited Jul 2, 2009 at 19:45

answered Jul 2, 2009 at 19:11

cwash

4,245 ● 5 ● 46 ● 53

1

You need to use the correct paradigm when writing tests.

1. Start by first writing your tests.
2. Make sure they fail to start off with.
3. Get them to pass.
4. Code review before you checkin your code (make sure the tests are reviewed.)

You cant always be sure but they improve overall tests.

Share Improve this answer

Follow

answered May 17, 2010 at 11:08

Jonathan

2,358 ● 7 ● 25 ● 45

0

Even if you do not test your code, it will surely be tested in production by your users. Users are very creative in trying to crash your soft and finding even non-critical errors.

Fixing bugs in production is much more costly than resolving issues in development phase. As a side-effect, you will lose income because of an exodus of customers. You can count on 11 lost or not gained customers for 1 angry customer.

Share Improve this answer

Follow

answered Aug 4, 2009 at 6:47

Luc Moerman