Is Unit Testing worth the effort? [closed]

Asked 16 years, 3 months ago Modified 9 years, 11 months ago

Viewed 315k times

572

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 12 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 am working to integrate unit testing into the development process on the team I work on and there are some sceptics. What are some good ways to convince the sceptical developers on the team of the value of Unit Testing? In my specific case we would be adding Unit Tests as we add functionality or fixed bugs. Unfortunately our code base does not lend itself to easy testing.

unit-testing

Share

edited Apr 10, 2013 at 19:43

community wiki
7 revs, 5 users 67%
George Stocker

25 How can you ask such a thing.. Unit testing is so hip :) ? Seriously... the general idea is the same... do unit testing if you feel that benefits outweigh the cost... if you have reasons to believe otherwise.. find something else that helps. – Gishu Sep 29, 2008 at 6:32

14 (regular asserts with debug\release build + clean interface to classes + dedicated dummy tester) > unit testing – Viktor Sehr Sep 30, 2010 at 19:34

110 Having been on a few teams with the same attitude, my experience is that you're wasting your time. The skeptics are just giving you the run-around, and will sabotage and stonewall you until you get frustrated and move to an organization that shares your values. Which is probably what you should do. If the "lead" of the team is one of the skeptics, think seriously about getting out. I've seen this sort of resistance to unit testing as just the tip of the iceberg of bad development practices. – sea-rob Feb 5, 2013 at 18:21

7 @ViktorSehr: unit tests are not in opposition to clean interfaces, in fact they are the opposite when applying TDD. Either that, or driver seat + steering wheel > garage. – Jonas Byström Feb 19, 2013 at 21:05

5 Unit testing is a massive waste of time that rarely ever discovers bugs but eats away 30% of bug-fixing and development time and budget – nanobar Nov 5, 2013 at 9:24

Comments disabled on deleted / locked posts / reviews |

44 Answers

Sorted by: Highest score (default)

722 votes

Every day in our office there is an exchange which goes something like this:

> "Man, I just love unit tests, I've just been able to make a bunch of changes to the way something works, and then was able to confirm I hadn't broken anything by running the test over it again..."

The details change daily, but the sentiment doesn't. Unit tests and test-driven development (TDD) have so many hidden and personal benefits as well as the obvious ones that you just can't really explain to somebody until they're doing it themselves.

But, ignoring that, here's my attempt!

1. Unit Tests allows you to make big changes to code quickly. You know it works now because you've run the tests, when you make the changes you need to

make, you need to get the tests working again. This saves hours.

2. TDD helps you to realise when to stop coding. Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing.
3. The tests and the code work together to achieve better code. Your code could be bad / buggy. Your TEST could be bad / buggy. In TDD you are banking on the chances of **both** being bad / buggy being low. Often it's the test that needs fixing but that's still a good outcome.
4. TDD helps with coding constipation. When faced with a large and daunting piece of work ahead writing the tests will get you moving quickly.
5. Unit Tests help you really understand the design of the code you are working on. Instead of writing code to do something, you are starting by outlining all the conditions you are subjecting the code to and what outputs you'd expect from that.
6. Unit Tests give you instant visual feedback, we all like the feeling of all those green lights when we've done. It's very satisfying. It's also much easier to pick up where you left off after an interruption because you can see where you got to - that next red light that needs fixing.
7. Contrary to popular belief unit testing does not mean writing twice as much code, or coding slower. It's

faster and more robust than coding without tests once you've got the hang of it. Test code itself is usually relatively trivial and doesn't add a big overhead to what you're doing. This is one you'll only believe when you're doing it :)

8. I think it was Fowler who said: "Imperfect tests, run frequently, are much better than perfect tests that are never written at all". I interpret this as giving me permission to write tests where I think they'll be most useful even if the rest of my code coverage is woefully incomplete.
9. Good unit tests can help document and define what something is supposed to do
10. Unit tests help with code re-use. Migrate both your code **and** your tests to your new project. Tweak the code till the tests run again.

A lot of work I'm involved with doesn't Unit Test well (web application user interactions etc.), but even so we're all test infected in this shop, and happiest when we've got our tests tied down. I can't recommend the approach highly enough.

edited Apr 25, 2014 at 6:49

community wiki
7 revs, 6 users 87%
reefnet_alex

3 The question was about unit testing. TDD is a whole different thing than unit testing. – ZunTzu Oct 16, 2014 at 12:12

420 votes

Unit testing is a lot like going to the gym. You know it is good for you, all the arguments make sense, so you start working out. There's an initial rush, which is great, but after a few days you start to wonder if it is worth the trouble. You're taking an hour out of your day to change your clothes and run on a hamster wheel and you're not sure you're really gaining anything other than sore legs and arms.

Then, after maybe one or two weeks, just as the soreness is going away, a Big Deadline begins approaching. You need to spend every waking hour trying to get "useful" work done, so you cut out extraneous stuff, like going to the gym. You fall out of the habit, and by the time Big Deadline is over, you're back to square one. If you manage to make it back to the gym at all, you feel just as sore as you were the first time you went.

You do some reading, to see if you're doing something wrong. You begin feel a little bit of irrational spite toward all the fit, happy people extolling the virtues of exercise. You realize that you don't have a lot in common. They don't have to drive 15 minutes out of the way to go to the gym; there is one in their building. They don't have to argue with anybody about the benefits of exercise; it is just something everybody does and accepts as important. When a Big

Deadline approaches, they aren't told that exercise is unnecessary any more than your boss would ask you to stop eating.

So, to answer your question, Unit Testing is usually worth the effort, but the amount of effort required isn't going to be the same for everybody. Unit Testing may require an enormous amount of effort if you are dealing with spaghetti code base in a company that doesn't *actually* value code quality. (A lot of managers will sing Unit Testing's praises, but that doesn't mean they will stick up for it when it matters.)

If you are trying to introduce Unit Testing into your work and are not seeing all the sunshine and rainbows that you have been led to expect, don't blame yourself. You might need to find a new job to really make Unit Testing work for you.

Share

answered Sep 16, 2008 at 3:53

community wiki
benzado

2 @user8863: Except its not just a good diet, but good exercise that you need as well. It's like reading all the good software engineering books but never applying the knowledge. In reality, we know things are different but its always good to strive for the best – user195488 Sep 30, 2010 at 18:50

1 idiots with ugly code will fail at unit testing even if they are not too lazy to try it. Pros will unit test and have lovely code, but

then be bored of it and move on to another project. The rest of us will unit test a little bit here and there when we really need it and we'll do 'Just Fine". – JDPeckham Mar 28, 2011 at 2:40

I used to work somewhere where we didn't unit-test. In fact people who tried unit-testing there soon went to work somewhere else. Now I'm working where I may and have to unit-test. Loving it a lot more. – Lauri Apr 5, 2012 at 11:51

1 Code is automated instructions to a computer. Tests are automated code to make sure said code is working. You even need tests for some of your tests depending on what they do. Most people are fine with testing on paper or after doing a simple tutorial. But once they try it on their own project, after it's been written, they realize it's very hard to do right -- especially with things that need Test Doubles. Thus the key is learn good testing engineering and to do it up front before your code is difficult test. Doing it all the time is required for being good at it anyway, just like anything. – rkulla Apr 16, 2014 at 17:18

136 votes

Best way to convince... find a bug, write a unit test for it, fix the bug.

That particular bug is unlikely to ever appear again, and you can prove it with your test.

If you do this enough, others will catch on quickly.

Share

answered Sep 15, 2008 at 21:48

community wiki

Ed Guiness

56 This only works if your code is built in such a way that facilitates unit testing (or you use TypeMock). Otherwise you'll spend eons building the test. Most code without unit tests is built with hard dependencies (i.e.'s new's all over the place) or static methods. This makes it almost impossible to just throw a quick unit test in place. – Andrew Jan 16, 2010 at 5:02

7 @Rei - The provable bug report is great, but you're only going to reproduce the bug ONCE. 2 years later, your unit test is still going to be checking the code, long after the bug report has been closed and forgotten. – Orion Edwards Sep 30, 2010 at 19:29

4 It saves the fix bug 1...test...good. Users find bug 2...fix...test...good. Users find bug 1 again. Lather, rinse, repeat. This happens because your fix for one bug, causes the other, and vice-versa. – CaffGeek Sep 30, 2010 at 19:33

1 @Rei Miyasaka: "Maybe if you have several people maintaining the project" - I'd say almost all non-trivial projects do. As to "just leave a comment": The problem is that there may (i.e. *will*) be interactions with other code which might cause the bug to resurface. That's why you actually have to test for it. – sleske Oct 3, 2010 at 15:28

5 Note, however, that tests to prevent regressions are usually integrations tests, and not unit tests (because in my experience most bugs are not in one part of the code only, but arise from the combination of several pieces of code, so you can only reproduce them by combining all these pieces). – sleske Oct 3, 2010 at 15:29

69

votes

thetalkingwalnut asks:

> What are some good ways to convince the skeptical developers on the team of the value of Unit Testing?

Everyone here is going to pile on lots of reasons out of the blue why unit testing is good. However, I find that often the best way to convince someone of something is to *listen* to their argument and address it point by point. If you listen and help them verbalize their concerns, you can address each one and perhaps convert them to your point of view (or at the very least, leave them without a leg to stand on). *Who knows? Perhaps they will convince you why unit tests aren't appropriate for your situation.* Not likely, but possible. Perhaps if you post their arguments against unit tests, we can help identify the counterarguments.

It's important to listen to and understand both sides of the argument. If you try to adopt unit tests too zealously without regard to people's concerns, you'll end up with a religious war (and probably really worthless unit tests). If you adopt it slowly and start by applying it where you will see the most benefit for the least cost, you might be able to demonstrate the value of unit tests and have a better chance of convincing people. I realize this isn't as easy as it sounds - it usually requires some time and careful metrics to craft a convincing argument.

Unit tests are a tool, like any other, and should be applied in such a way that the benefits (catching bugs) outweigh the costs (the effort writing them). Don't use them if/where

they don't make sense and remember that they are only part of your arsenal of tools (e.g. inspections, assertions, code analyzers, formal methods, etc). What I tell my developers is this:

1. They can skip writing a test for a method if they have a good argument why it isn't necessary (e.g. too simple to be worth it or too difficult to be worth it) and how the method will be otherwise verified (e.g. inspection, assertions, formal methods, interactive/integration tests). They need to consider that some verifications like inspections and formal proofs are done at a point in time and then need to be repeated every time the production code changes, whereas unit tests and assertions can be used as regression tests (written once and executed repeatedly thereafter). Sometimes I agree with them, but more often I will debate about whether a method is really too simple or too difficult to unit test.
    - If a developer argues that a method seems too simple to fail, isn't it worth taking the 60 seconds necessary to write up a simple 5-line unit test for it? These 5 lines of code will run every night (you do nightly builds, right?) for the next year or more and will be worth the effort if even just once it happens to catch a problem that may have taken 15 minutes or longer to identify and debug. Besides, writing the easy unit tests drives up the count of unit tests, which makes the developer look good.

    - On the other hand, if a developer argues that a method seems too difficult to unit test (not worth the significant effort required), perhaps that is a good indication that the method needs to be divided up or refactored to test the easy parts. Usually, these are methods that rely on unusual resources like singletons, the current time, or external resources like a database result set. These methods usually need to be refactored into a method that gets the resource (e.g. calls getTime()) and a method that takes the resource as a argument (e.g. takes the timestamp as a parameter). I let them skip testing the method that retrieves the resource and they instead write a unit test for the method that now takes the resource as a argument. Usually, this makes writing the unit test much simpler and therefore worthwhile to write.
2. The developer needs to draw a "line in the sand" in how comprehensive their unit tests should be. Later in development, whenever we find a bug, they should determine if more comprehensive unit tests would have caught the problem. If so and if such bugs crop up repeatedly, they need to move the "line" toward writing more comprehensive unit tests in the future (starting with adding or expanding the unit test for the current bug). They need to find the right balance.

Its important to realize the unit tests are not a silver bullet and there *is* such a thing as too much unit testing. At my workplace, whenever we do a lessons learned, I inevitably

hear "we need to write more unit tests". Management nods in agreement because its been banged into their heads that "unit tests" == "good".

However, we need to understand the impact of "more unit tests". A developer can only write ~N lines of code a week and you need to figure out what percentage of that code should be unit test code vs production code. A lax workplace might have 10% of the code as unit tests and 90% of the code as production code, resulting in product with a lot of (albeit very buggy) features (think MS Word). On the other hand, a strict shop with 90% unit tests and 10% production code will have a rock solid product with very few features (think "vi"). You may never hear reports about the latter product crashing, but that likely has as much to do with the product not selling very well as much as it has to do with the quality of the code.

Worse yet, perhaps the only certainty in software development is that "change is inevitable". Assume the strict shop (90% unit tests/10% production code) creates a product that has exactly 2 features (assuming 5% of production code == 1 feature). If the customer comes along and changes 1 of the features, then that change trashes 50% of the code (45% of unit tests and 5% of the production code). The lax shop (10% unit tests/90% production code) has a product with 18 features, none of which work very well. Their customer completely revamps the requirements for 4 of their features. Even though the change is 4 times as large, only half as much of the code

base gets trashed (~25% = ~4.4% unit tests + 20% of production code).

My point is that you have to communicate that you understand that balance between too little and too much unit testing - essentially that you've thought through both sides of the issue. If you can convince your peers and/or your management of that, you gain credibility and perhaps have a better chance of winning them over.

Share

edited Jan 5, 2015 at 18:05

community wiki
2 revs, 2 users 91%
Bert F

47 If Vi has very few features, you're not using it. ;) – Stefan Mai Jun 26, 2009 at 18:14

1 Testivus on Test Coverage - googletesting.blogspot.com/2010/07/… – Bert F Jul 20, 2010 at 19:39

A good analysis, though two things I'd say. You seem to assume that changing a feature requires rewriting all the code that it relies on and it's rather an assumption that there's a linear relationship between change and 'amount of code' change. Also, a product with low testing is more likely to have a worse design, hence the 'lots-of-features' product will almost certainly take much longer per feature (as there are almost no tests and an implicitly bad design). – nicodemus13 Oct 18, 2013 at 0:14

writing test case for simple code block serves as a regression mechanism - only negative I had seen is adding more stack frames... – bgs Oct 19, 2013 at 19:23

38

votes

I have toyed with unit testing a number of times, and I am still to be convinced that it is worth the effort given my situation.

I develop websites, where much of the logic involves creating, retrieving or updating data in the database. When I have tried to "mock" the database for unit testing purposes, it has got very messy and seemed a bit pointless.

When I have written unit tests around business logic, it has never really helped me in the long run. Because I largely work on projects alone, I tend to know intuitively which areas of code may be affected by something I am working on, and I test these areas manually. I want to deliver a solution to my client as quickly as possible, and unit testing often seems a waste of time. I list manual tests and walk through them myself, ticking them off as I go.

I can see that it may be beneficial when a team of developers are working on a project and updating each other's code, but even then I think that if the developers are of a high quality, good communication and well-written code should often be enough.

Share

answered Jul 8, 2009 at 11:19

community wiki
Ronnie

8 I know this is very old but I am compelled to comment. I personally have found success writing tests for my persistence layer. Simply start a transaction before the test & roll it back afterwards. No mocking needed. Sometimes I have one class to pull an array of data, I then pass that array of data to a second class that does "things" to the data. This second class does not touch the database. In this case I group my tests, the ones that test the first class & touch the database are 'functional tests' and run infrequently, the tests for the latter class are unit tests & run frequently. – Josh Ribakoff Apr 25, 2013 at 4:06

4 Testing is all about happiness. I've got a 3900-line package (PL/SQL) that handles automated postings to the general ledger system. Now, I personally *hate* dealing with accountants - they're so picky about little things like fractional pennies and stuff. Buncha anal-retentive arithmetic geeks...I dunno. My unit test package for the accounting package is about 22000 lines, and runs just under 18000 tests. This keeps the Head Honcho in Accounting all happy, and as long as the propeller on his little beanie-counting-cap is spinning happily, I'm happy... :-)
– Bob Jarvis - Слава Україні Oct 2, 2013 at 18:20

31 votes

One great thing about unit tests is that they serve as documentation for how your code is meant to behave. Good tests are kind of like a reference implementation, and teammates can look at them to see how to integrate their code with yours.

Share

edited Feb 10, 2009 at 0:12

community wiki
2 revs, 2 users 67%
pkaeding

26

votes

Unit-testing is well worth the initial investment. Since starting to use unit-testing a couple of years ago, I've found some real benefits:

- **regression testing** removes the fear of making changes to code (there's nothing like the warm glow of seeing code work or explode every time a change is made)
- **executable code examples** for other team members (and yourself in six months time..)
- ***merciless* refactoring** - this is incredibly rewarding, try it!

Code snippets can be a great help in reducing the overhead of creating tests. It isn't difficult to create snippets that enable the creation of a class outline and an associated unit-test fixture in seconds.

Share

edited Sep 15, 2008 at 22:33

community wiki

2 revs
Jonathan Webb

25

votes

You should test as little as possible!

meaning, you should write just enough unit tests to reveal intent. This often gets glossed over. Unit testing costs you. If you make changes and you have to change tests you will be less agile. Keep unit tests short and sweet. Then they have a lot of value.

Too often I see lots of tests that will never break, are big and clumsy and don't offer a lot of value, they just end up slowing you down.

Share

answered Sep 15, 2008 at 23:32

community wiki
Keith Nicholas

23

votes

I didn't see this in any of the other answers, but one thing I noticed is that **I could debug so much faster**. You don't need to drill down through your app with just the right sequence of steps to get to the code your fixing, only to find you've made a boolean error and need to do it all again. With a unit test, you can just step directly into the code you're debugging.

Share

answered Feb 10, 2009 at 0:21

community wiki
John MacIntyre

21

votes

[I have a point to make that I cant see above]

"Everyone unit tests, they don't necessarily realise it - FACT"

Think about it, you write a function to maybe parse a string and remove new line characters. As a newbie developer you either run a few cases through it from the command line by implementing it in Main() or you whack together a visual front end with a button, tie up your function to a couple of text boxes and a button and see what happens.

That is unit testing - basic and badly put together but you test the piece of code for a few cases.

You write something more complex. It throws errors when you throw a few cases through (unit testing) and you debug into the code and trace though. You look at values as you go through and decide if they are right or wrong. This is unit testing to some degree.

Unit testing here is really taking that behaviour, formalising it into a structured pattern and saving it so that you can easily re-run those tests. If you write a "proper" unit test case rather than manually testing, it takes the same

amount of time, or maybe less as you get experienced, and you have it available to repeat again and again

Share

answered Aug 19, 2009 at 15:12

community wiki
Chris Gill

---

16 votes

For years, I've tried to convince people that they needed to write unit test for their code. Whether they wrote the tests first (as in TDD) or after they coded the functionality, I always tried to explain them all the benefits of having unit tests for code. Hardly anyone disagreed with me. You cannot disagree with something that is obvious, and any smart person will see the benefits of unit test and TDD.

The problem with unit testing is that it requires a behavioral change, and it is very hard to change people's behavior. With words, you will get a lot of people to agree with you, but you won't see many changes in the way they do things.

You have to convince people by doing. Your personal success will atract more people than all the arguments you may have. If they see you are not just talking about unit test or TDD, but you are doing what you preach, and you are successful, people will try to imitate you.

You should also take on a lead role because no one writes unit test right the first time, so you may need to coach them on how to do it, show them the way, and the tools available

to them. Help them while they write their first tests, review the tests they write on their own, and show them the tricks, idioms and patterns you've learned through your own experiences. After a while, they will start seeing the benefits on their own, and they will change their behavior to incorporate unit tests or TDD into their toolbox.

Changes won't happen over night, but with a little of patience, you may achieve your goal.

Share

answered Sep 16, 2008 at 1:32

community wiki
Curro

12

votes

A major part of test-driven development that is often glossed over is the writing of testable code. It seems like some kind of a compromise at first, but you'll discover that testable code is also ultimately modular, maintainable and readable. If you still need help convincing people this is a nice simple presentation about the advantages of unit testing.

Share

answered Sep 15, 2008 at 21:50

community wiki
Tom Martin

11

votes

If your existing code base doesn't lend itself to unit testing, and it's already in production, you might create more problems than you solve by trying to refactor all of your code so that it is unit-testable.

You may be better off putting efforts towards improving your integration testing instead. There's lots of code out there that's just simpler to write without a unit test, and if a QA can validate the functionality against a requirements document, then you're done. Ship it.

The classic example of this in my mind is a SqlDataReader embedded in an ASPX page linked to a GridView. The code is all in the ASPX file. The SQL is in a stored procedure. What do you unit test? If the page does what it's supposed to do, should you really redesign it into several layers so you have something to automate?

Share

answered Sep 15, 2008 at 21:59

community wiki
Eric Z Beard

10

votes

One of the best things about unit testing is that your code will become easier to test as you do it. Preexisting code created without tests is always a challenge because since they weren't meant to be unit-tested, it's not rare to have a high level of coupling between classes, hard-to-configure objects inside your class - like an e-mail sending service

reference - and so on. But don't let this bring you down! You'll see that your overall code design will become better as you start to write unit-tests, and the more you test, the more confident you'll become on making even more changes to it without fear of breaking you application or introducing bugs.

There are several reasons to unit-test your code, but as time progresses, you'll find out that the time you save on testing is one of the best reasons to do it. In a system I've just delivered, I insisted on doing automated unit-testing in spite of the claims that I'd spend way more time doing the tests than I would by testing the system manually. With all my unit tests done, I run more than 400 test cases in less than 10 minutes, and every time I had to do a small change in the code, all it took me to be sure the code was still working without bugs was ten minutes. Can you imagine the time one would spend to run those 400+ test cases by hand?

When it comes to automated testing - be it unit testing or acceptance testing - everyone thinks it's a wasted effort to code what you can do manually, and sometimes it's true - if you plan to run your tests only once. The best part of automated testing is that you can run them several times without effort, and after the second or third run, the time and effort you've *wasted* is already paid for.

One last piece of advice would be to not only unit test your code, but start doing test first (see TDD and BDD for more)

Share

answered Sep 15, 2008 at 21:59

community wiki
Alexandre Brasil

---

8 votes

Unit tests are also especially useful when it comes to refactoring or re-writing a piece a code. If you have good unit tests coverage, you can refactor with confidence. Without unit tests, it is often hard to ensure the you didn't break anything.

Share

answered Sep 15, 2008 at 21:59

community wiki
killdash10

---

7 votes

In short - yes. They are worth every ounce of effort... to a point. Tests are, at the end of the day, still code, and much like typical code growth, your tests will eventually need to be refactored in order to be maintainable and sustainable. There's a tonne of GOTCHAS! when it comes to unit testing, but man oh man oh man, nothing, and I mean NOTHING empowers a developer to make changes more confidently than a rich set of unit tests.

I'm working on a project right now.... it's somewhat TDD, and we have the majority of our business rules

encapuslated as tests... we have about 500 or so unit tests right now. This past iteration I had to revamp our datasource and how our desktop application interfaces with that datasource. Took me a couple days, the whole time I just kept running unit tests to see what I broke and fixed it. Make a change; Build and run your tests; fix what you broke. Wash, Rinse, Repeat as necessary. What would have traditionally taken days of QA and boat loads of stress was instead a short and enjoyable experience.

Prep up front, a little bit of extra effort, and it pays 10-fold later on when you have to start dicking around with core features/functionality.

I bought this book - it's a Bible of xUnit Testing knowledge - tis probably one of the most referenced books on my shelf, and I consult it daily: link text

Share

answered Sep 15, 2008 at 22:27

community wiki
cranley

+1: *but man oh man oh man, nothing, and I mean NOTHING empowers a developer to make changes more confidently than a rich set of unit tests.* - So true. I wish I figured this out sooner. – Jim G. Sep 30, 2010 at 19:12

7

votes

Occasionally either myself or one of my co-workers will spend a couple of hours getting to the bottom of slightly obscure bug and once the cause of the bug is found 90% of the time that code isn't unit tested. The unit test doesn't exist because the dev is cutting corners to save time, but then looses this and more debugging.

Taking the small amount of time to write a unit test can save hours of future debugging.

Share

answered Sep 16, 2008 at 5:11

community wiki
Jon Cahill

+1: *Taking the small amount of time to write a unit test can save hours of future debugging.* - Yup! – Jim G. Sep 30, 2010 at 19:15

7

votes

I'm working as a maintenance-engineer of a poorly documented, awful and big code base. I wish the people who wrote the code had written the unit tests for it.
Each time I make a change and update the production code I'm scared that I might introduce a bug for not having considered some condition.
If they wrote the test making changes to the code base would be easier and faster.(at the same time the code base would be in a better state)..

I think unit tests prove a lot useful when writing api or frameworks that have to last for many years and to be used/modified/evolved by people other than the original coders.

Share

edited Sep 30, 2010 at 19:21

community wiki
2 revs, 2 users 89%
mic.sca

Do you add unit tests for new code or fixes that you make?
– ɔɔɯəɹ Jan 22, 2015 at 22:32

6 votes

Unit Testing is definitely worth the effort. Unfortunately you've chosen a difficult (but unfortunately common) scenario into which to implement it.

The best benefit from unit testing you'll get is when using it from the ground up - on a few, select, small projects I've been fortunate enough to write my unit tests before implementing my classes (the interface was already complete at this point). With proper unit tests, you will find and fix bugs in your classes while they're still in their infancy and not anywhere near the complex system that they'll undoubtedly become integrated in in the future.

If your software is solidly object oriented, you should be able to add unit testing at the class level without too much

effort. If you aren't that fortunate, you should still try to incorporate unit testing wherever you can. Make sure when you add new functionality the new pieces are well defined with clear interfaces and you'll find unit testing makes your life much easier.

Share

answered Sep 15, 2008 at 21:59

community wiki
Matt

---

6 votes

When you said, "our code base does not lend itself to easy testing" is the first sign of a code smell. Writing Unit Tests means you typically write code differently in order to make the code more testable. This is a good thing in my opinion as what I've seen over the years in writing code that I had to write tests for, it forced me to put forth a better design.

Share

answered Sep 16, 2008 at 0:55

community wiki
Keith Elder

---

6 votes

I do not know. A lot of places do not do unit test, but the quality of the code is good. Microsoft does unit test, but Bill Gates gave a blue screen at his presentation.

Share

answered May 21, 2010 at 0:25

community wiki
BigTiger

> Note that was almost 15 years ago, when unit testing wasn't as popular as it is today. – cthulhu Oct 19, 2010 at 14:33

5
votes

I wrote a very large blog post about the topic. I've found that unit testing alone isn't worth the work and usually gets cut when deadlines get closer.

Instead of talking about unit testing from the "test-after" verification point of view, we should look at the true value found when you set out to write a spec/test/idea before the implementation.

This simple idea has changed the way I write software and I wouldn't go back to the "old" way.

How test first development changed my life

Share

edited Sep 2, 2013 at 18:39

community wiki
2 revs, 2 users 82%
Toran Billups

1 +1 - and I must say it's mind-boggling how many trolls that blog entry attracted (prior to you posting here, if I'm not mistaken).
– Jeff Sternal Oct 1, 2010 at 15:56

haha agreed :) </ reddit front-page does that it seems>
– Toran Billups Oct 2, 2010 at 21:11

3
votes

Yes - Unit Testing is definitely worth the effort but you should know it's not a silver bullet. Unit Testing is work and you will have to work to keep the test updated and relevant as code changes but the value offered is worth the effort you have to put in. The ability to refactor with impunity is a huge benefit as you can always validate functionality by running your tests after any change code. The trick is to not get too hung up on exactly the unit-of-work you're testing or how you are scaffolding test requirements and when a unit-test is really a functional test, etc. People will argue about this stuff for hours on end and the reality is that any testing you do as your write code is better than not doing it. The other axiom is about quality and not quantity - I have seen code-bases with 1000's of test that are essentially meaningless as the rest don't really test anything useful or anything domain specific like business rules, etc of the particular domain. I've also seen codebases with 30% code coverage but the tests were relevant, meaningful and really awesome as they tested the core functionality of the code it was written for and expressed how the code should be used.

One of my favorite tricks when exploring new frameworks or codebases is to write unit-tests for 'it' to discover how things work. It's a great way to learn more about something new instead of reading a dry doc :)

Share

answered Sep 15, 2008 at 23:46

community wiki
Vinny Carpenter

3
votes

I recently went through the exact same experience in my workplace and found most of them knew the theoretical benefits but had to be sold on the benefits to them specifically, so here were the points I used (successfully):

- They save time when performing negative testing, where you handle unexpected inputs (null pointers, out of bounds values, etc), as you can do all these in a single process.
- They provide immediate feedback at compile time regarding the standard of the changes.
- They are useful for testing internal data representations that may not be exposed during normal runtime.

and the big one...

- You might not need unit testing, but when someone else comes in and modifies the code without a full

understanding it can catch a lot of the silly mistakes they might make.

Share

edited Sep 29, 2008 at 6:20

community wiki
2 revs
dlanod

+1: •*You might not need unit testing, but when someone else comes in and modifies the code without a full understanding it can catch a lot of the silly mistakes they might make.* - Yup. And given the amount of turnover in the software industry, this is a huge benefit. – Jim G. Sep 30, 2010 at 19:16

3

votes

I discovered TDD a couple of years ago, and have since written all my pet projects using it. I have estimated that it takes roughly the same time to TDD a project as it takes to cowboy it together, but I have such increased confidence in the end product that I can't help a feeling of accomplishment.

I also feel that it improves my design style (much more interface-oriented in case I need to mock things together) and, as the green post at the top writes, it helps with "coding constipation": when you don't know what to write next, or you have a daunting task in front of you, you can write small.

Finally, I find that by far the most useful application of TDD is in the debugging, if only because you've already developed an interrogatory framework with which you can prod the project into producing the bug in a repeatable fashion.

Share

answered Jul 20, 2009 at 16:02

community wiki
Kaz Dragon

3
votes

One thing no-one has mentioned yet is getting the commitment of all developers to actually run and update any existing automated test. Automated tests that you get back to and find broken because of new development looses a lot of the value and make automated testing really painful. Most of those tests will not be indicating bugs since the developer has tested the code manually, so the time spent updating them is just waste.

Convincing the skeptics to not destroy the work the others are doing on unit-tests is a lot more important for getting value from the testing and might be easier.

Spending hours updating tests that has broken because of new features each time you update from the repository is neither productive nor fun.

Share

answered Oct 1, 2011 at 8:42

community wiki
Samuel Åslund

2
votes

If you are using NUnit one simple but effective demo is to run NUnit's own test suite(s) in front of them. Seeing a real test suite giving a codebase a workout is worth a thousand words...

Share

answered Sep 15, 2008 at 21:51

community wiki
Garth Gilmour

2
votes

Unit testing helps a lot in projects that are larger than any one developer can hold in their head. They allow you to run the unit test suite before checkin and discover if you broke something. This cuts down a *lot* on instances of having to sit and twiddle your thumbs while waiting for someone else to fix a bug they checked in, or going to the hassle of reverting their change so you can get some work done. It's also immensely valuable in refactoring, so you can be sure that the refactored code passes all the tests that the original code did.

Share

answered Sep 15, 2008 at 21:59

community wiki
Max Rible Kaehn

---

2 votes

With unit test suite one can make changes to code while leaving rest of the features intact. Its a great advantage. Do you use Unit test sutie and regression test suite when ever you finish coding new feature.

Share

answered Sep 15, 2008 at 23:37

community wiki
Shinwari

---

2 votes

I'm agree with the point of view opposite to the majority here: It's OK Not to Write Unit Tests Especially prototype-heavy programming (AI for example) is difficult to combine with unit testing.

Share

edited Feb 25, 2011 at 12:14

community wiki
2 revs
DSblizzard

---

1 2 Next