Write More Tests

SATURDAY, 24 SEPTEMBER 2011

Test-Driven Development? Give me a break...

Update: At the bottom of this post, I've linked to two large and quite different discussions of this post, both of which are worth reading...

Update 2: If the contents of this post make you angry, okay. It was written somewhat brashly. **But**, if the title alone makes you angry, and you decide this is an article about "*Why Testing Code Sucks*" without having read it, you've missed the point. Or I explained it badly :-)

Some things programmers say can be massive red flags. When I hear someone start advocating Test-Driven Development as the One True Programming Methodology, that's a red flag, and I start to assume you're either a shitty (or inexperienced) programmer, or some kind of Agile Testing Consultant (which normally implies the former).

Testing is a tool for **helping you**, not for using to engage in a "more pious than thou" dick-swinging my *Cucumber* is bigger than yours idiocy. Testing is about giving **you the developer** useful and quick feedback about if you're on the right path, and if you've broken something, and for warning people who come after you if they've broken something. It's not an arcane methodology that somehow has some magical "making your code better" side-effect...

The whole concept of Test-Driven Development is hocus, and embracing it as your philosophy, criminal. Instead: Developer-Driven Testing. Give yourself and your coworkers useful tools for solving problems and supporting yourselves, rather than disappearing in to some testing hell where you're doing it a certain way because you're supposed to.

Have I had experience (and much value) out of sometimes writing tests for certain problem classes before writing any code? Yes. Changes to existing functionality are often a good candidate. Small and well-defined pieces of work, or little add-ons to already tested code are another.

But the demand that you should **always** write your tests first? Give me a break.

This is idiocy during a design or hacking or greenfield phase of development. Allowing your tests to dictate your code (rather than influence the design of modular code) and to dictate your design because you wrote over-invasive test is a massive fail.

Writing tests before code works pretty well in some situations. Test Driven Development, as handed down to us mortals by Agile Testing Experts and other assorted shills, is hocus.

Labouring under the idea that Tests Must Come First (and everything I've seen, and everything I *do* see now suggests that that is the central idea in TDD - you write a test, then you write the code to pass it) without pivoting to see that testing is a useful practice in so much as it helps developers is the wrong approach.

Even if you write only *some* tests first, if you want to do it meaningfully, then you either need to zoom down in to tiny bits of functionality first in order to be able to write those tests, or you write a test that requires most of the software to be finished, or you cheat and fudge it. The former is the right approach in a small number of

situations - tests around bugs, or small, very well-defined pieces of functionality).

Making tests a central part of the process because they're useful to developers? Awesome. Dictating a workflow to developers that works in some cases as the One True Way: ridiculous.

Testing is about helping developers, and recognizing that automated testing is about benefit to developers, rather than cargo-culting a workflow and decreeing that one size fits all.

Writing tests first as a tool to be deployed where it works is "Developer Driven Testing" - focusing on making the developer more productive by choosing the right tool for the job. Generalizing a bunch of testing rules and saying This Is The One True Way Even When It Isn't - that's not right.

Discussion and thoughts (posted a few hours later)...

I wrote this a few short hours ago, and it's already generated quite the discussion.

On Hacker News, there's a discussion that I think asks a lot of good questions, and there's a real set of well-reasoned opinions. I have been responding on there quite a bit with the username *peteretep*.

On Reddit, the debate is a little more ... uh ... robust. There are a lot of people defending writing automated tests. As this blog is largely meant to move forward as being a testing advocacy and practical advice resource, I've clearly miscommunicated my thoughts, and not made it clear enough that I think software testing is *pretty darn awesome*, but I'm put off by slavish adherence to a particular methodology!

If you've posted a comment on the blog and it's not there yet, sorry. Some are getting caught in the spam folder. I'm not censoring anyone, and I'm not planning to, so please be patient!

Anyway, the whole thing serves me right for putting together my first blog post by copy-pasting from a bunch of HN comments I'd made. The next article is a walk-through of retro-fitting functional testing to large web-apps that don't already have it, and in such a way as the whole dev team starts using it.

Posted by Peter Sergeant at 02:30

33 people +1'd this

35 comments:

Jason Hanley said...

Nicely said. Clearly the voice of experience.

Tests are only valuable if they're well thought out, and actually give more benefit than their cost.

I've seen way too many examples of automated tests that cause more problems than they solve.

24 September 2011 08:08

Dawid Loubser said...

You seem to live in a world of hacking together toy software. One day when you become a software ENGINEER who has to build complex, long-lived software as part of a team of people, following modern engineering practices (model-driven development, design-by-contract), upon which people's well-being depends, you will change your tune, I suspect. As son as you have DESIGNED a component (at any level of granularity) you can derive a good set of test cases for it using established techniques developed by the testing community over decades. But you don't design, do you? Your strongly-worded hissy-fit of a blog post suggests that you are far from being a professional, and I hope you're not writing any important software (for the good of mankind).

If you ally wanted to attack test-driven development, you could at least have taken the time to learn what it's all about, and could have tried to construct halfway-decent logical arguments against the (alleged, according to you) benefits. Instead, it seems you had a really bad day trying to write some complex code, and instead of kicking your dog, decided to add this giant fit of misunderstanding to the world.

Perhaps you should focus on the skills that good software engineers really are made of. (Hint: it's not programming.)

24 September 2011 08:09

chadastrophic said...

Great article, one of the greatest programmers Ive known is an advocated of TDD, but is also smart and balances reality with best practices. I really enjoyed the honesty in your article. Thanks! My colleague who is a strong advocate has made me consider this paradigm more seriously and its great to be knowledgeable when and how to apply certain approaches.

24 September 2011 08:09

Micha said...

I think in somepoint your right but
you can write TDD as Feature lists this i think is a good thing and iam using it often on web Project cause i first think what i wanne do before i do it. Often Programmers do something write shitty code and then they are to lazzy to fix it... this is not a good programming style

sry for my bad english

24 September 2011 08:39

Aleksey Korzun said...

Dawid Loubser,

Using TDD does not instantly make you a superior engineer and everybody else a toy software maker.

You are just backing up authors point by being full of your self just because you use specific development approach that works for you.

24 September 2011 10:06

Rafał Rusin said...

I agree with the idea of the article. We don't need TDD Palladins. It's usually a bad idea to start from a test case when you develop new piece of functionality and have little idea of how it will look like when it's done. On the other hand, I prefer one unit test, which is well thought and tests actual functionality, instead of 10 tests for getters and setters (which I've seen in some code). TDD if goes mad, is a monster.

24 September 2011 10:13

Lord almighty said...

TDD rocks, its just a matter of how its implemented and using it when it fits.

Small-team projects doing RAD are normally greatly benefited by TDD.

24 September 2011 10:24

foobardude said...

Dawid Loubser: a man of zero substance for an argument.

Dawid - experienced teams often don't write tests because we have to make money. We hate it, but we need to get a check. TDD is a fun exercise but isn't

really practiced often. Yet when attempted you have a lot of tests that are good to go.

Don't be such a douche, chill out and just read what everyone else is saying. And when you attempt a counter point, back it up with real facts and not sound like the agile consultant the author was ripping on.

24 September 2011 10:32

oliverclozoff said...

> because you wrote over-invasive test is a massive fail.

It's failURE.

24 September 2011 10:32

hacksoncode said...

You know, the biggest thing I've always wondered about "test driven development" is "what process do you use to develop your tests?".

24 September 2011 10:50

James said...

Aleksey,

Hating TDD with a passion doesn't make you one either, and advocating TDD doesn't make you a bad one. Tools and methodologies are nothing more than tools and methodologies. You can have a major preference for one, but if you're a good developer/engineer, you can adapt to whatever is being used.

Before I jump in here, I'll mention; my group doesn't advocate TDD, but it does require unit testing in some form. But let's see here:

Peter,

> Allowing your tests to dictate your code (rather than influence the design of modular code) and to dictate your design because you wrote over-invasive test is a massive fail.

Yep. Doing TDD also means that you actually have to be good about writing testable code, and writing -good- tests. You're applying TDD over the top of other software engineering best practices. If you don't, you're just going to wind up shooting yourself in the foot. Not using TDD but writing over-invasive tests is also a massive fail - it has nothing to do with TDD.

Let's say I'm writing a server which reads data from two sources, performs some complicated data munging, and returns some answer. Simple tests for your DAOs, write the DAOs. Nothing too invasive so far. Write tests for your data munging, and implement the munging algorithm. No over invasive tests, so far, and nothing has dictated my design. Each piece is logically going to do what it's going to do. Finally, the overall server tests, and the server itself.

If you take the other directional approach, you write your tests for the server, mocking out the algorithm (meaning you don't have to write the rest yet, so long as your mocks obey the contracts of the algorithm class), etc.

> Even if you write only some tests first, if you want to do it meaningfully, then you either need to zoom down in to tiny bits of functionality

That's not really a bad thing. It's sort of the point of unit testing in general - you don't know that your higher level components are working unless you know the lower level ones are.

> write a test that requires most of the software to be finished, or you cheat and fudge it.
If you have software with well defined APIs, "fudging it" is fine. I can drop out my ExpensiveBullshitAlgorithm with a mock which returns 3 when the inputs are 1 and 2. Those are the expectations of the system, and we'll prove that

ExpensiveBullshitAlgorithm actually returns 3 for inputs of 1 and 2 when we write the tests there. This is not "cheating", this is "mocking", and it's only something you can get away with if you're actually writing solid tests for your components, and they obey defined APIs.

> Generalizing a bunch of testing rules and saying This Is The One True Way Even When It Isn't - that's not right.
Consistency is important in large group projects. If half the team is working in one way, and half the team is working in another, you ARE going to clash. It's not necessarily the One True Way - nothing is. But in the real world there is quite often the One Way We Decided On For Consistency and You're a Big Boy/Girl So You Can Adapt, Right? And once you get used to it, you might even like it.

24 September 2011 11:11

Joe Magly said...

Everything has a balance, the problem I see in some shops is they tend to take the chosen practice to one extreme or the other without balancing actual need or value.

Testing helps, writing tests first or last I feel is inconsequential to the ultimate goal of the tests if you really are doing true unit testing (some shops only think they are unit testing). Writing do nothing tests that just improve your coverage ratio are not worth the time. Spending an extra hour on that method with the complex object as an input parameter to cover more negative cases would be more valuable.

24 September 2011 12:33

Ben Smith said...

Use the right tool for the job! If you have a large, complex codebase, TDD is the ONLY sensible way to keep things manageable while you change fundamentals. TDD makes little sense for a 200 line project knocked off in an afternoon, or anybody using the waterfall method.

24 September 2011 12:46

Rush said...

We'll build it, get it working, then we'll design it. <<<---- 85% of the projects I've ever worked on.

24 September 2011 13:31

Cody said...

Great article. I don't care whether you like TDD or not, just don't be one of those douches who regurgitates all kinds of bullshit about how great some method is.

I, personally, like to test after I'm fairly sure of the design. When I've devised my overall algorithm and determined the interfaces and implementations, then I'll write tests as I go so I can safely refactor as I work.

Also, TDD for interfaces like a web page is just plain bullshit. When you do TDD for integration tests on a web application, you're just wasting your time. Test the hell out of your unit tests, write functional tests as you understand the problem more and you become more sure of the solution, and if write interface tests once you're done and you're concerned with them breaking during deployment.

24 September 2011 13:48

Vic said...

TDD is for http://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

Integration tests help, ex: browser to socket server to db during CRUD.
The rest are just a religion.

24 September 2011 14:02

WildThought said...

Every methodology can be taken to an extreme. RUP, Waterfall, and I think TDD is especially prone to it.

It reminds me of the relational database modeling vs. OOP debates during RUP's heyday. What do you do first, data model or class model. I would argue it doesn't matter as long as you get the same result. Their are rules that ORM's have figured out how to reverse and forward engineer between each other. If those rules exist, then why can't we model either way first.

Similarly whether we write tests first or code first should be the purview of the person writing the program. Do I really want to write a test first for every stored procedure I write. Of course, I do not. I am sick of seeing articles on how to do TDD in the database layer. I am equally sick of having to create my own mocks to mimic a layer I can get to in ms.

I think it has its place, but it does promote coding bloat. Now, RUP promotes documentation bloat. I truly believe, that quality software engineers (aka master craftsmen) can choose what methodologies to draw from as needed. To say, I have the way and its the only way is telling me that you are afraid to think outside your own box.

24 September 2011 15:08

Maht said...

Generalising is fun. We call it "having an opinion".

24 September 2011 15:13

Dave Thompson said...

This post has been removed by the author.

24 September 2011 16:03

Dave Thompson said...

"writing tests first or last I feel is inconsequential"

The point of TDD is that writing your tests first forces you to use your code before writing it, which in theory leads to better designed, simpler interfaces. The tests then serve as the formal specification for your interface, which often leads to easier and quicker implementation of your interface. Since your code's specification is now being tested, it is very easy to prove to stakeholders that your code works as intended, and is often easier to change when stakeholders change their minds. If you write your implementation first, you may not realize until later down the road that your interface is awkward or difficult to use, and by then it takes more time to fix it.

TDD is not always necessary or even the best way to do things. TDD is probably overkill if you're working on a simple CRUD form with no logic outside of validation and persistence. TDD's advantages show themselves quickly when working with a technology or business domain that you're not experienced with, when you're working with complex systems, and when you're creating public apis. In these cases, TDD helps get your design correct the 1st try, and saves a lot of time. In addition TDD has many advantages when working with a large team. Any time 'wasted' writing tests is more than made up for by elimination of technical debt and time spent refactoring or fix bugs.

24 September 2011 16:07

Darren said...

Something that should be kept in mind about TDD is that nobody expects you to do it all the time -- even its most staunch promoters. Full test-first code is an ideal, as something to be worked for. Whether you will reach 100% is dependent on a lot of factors, like skill, time, understanding, tools, your framework and language, coworkers, etc.

But if you don't hit 100%, you don't throw up your hands, curse the method and

write an angry blog post about how TDD upsets you.

Instead, you say: "Next time, I'll do better." And you do.

That's the difference between test driven development and your "developer driven development." TDD is a method of producing tested, working code, it takes a long time to master (I'm not even there yet), and it's an ideal that its practitioners work towards. Your DDD is a method that says that whatever "works" today is fine, whoever you are and whatever you do today, and testing is nice so long as it's in some form before or after the code is written. Kinda vague...

But since you mock TDD as the "one try way," but I have a question for you: If you're not able to write simple test cases for all of the code you write, even before that code, how can you be satisfied with yourself?

24 September 2011 19:23

Dawid Loubser said...

Let's say you make a statement about your component or system, such as "under circumstances X, given input Y, it will produce Z". One of only two truths apply: Option A: you make the statement based on the belief that your code (which other members of your team have perhaps modified) is sound, or based on experience. Let's call this "faith". Option B: You make the statement because there is a unit test that proves it ("proof"). In other fields of engineering, things are not built based on faith.

Unit tests, at every level of granularity, are the only way to prove that your system works. Anything less fosters a self-important, "code ownership", hacking culture, and virtually proves that you are coding without having performed any real design.

Anybody is free to follow this style of work, but in the 21st century, this is thoroughly amateur, in my opinion, and suited only to toy software. Are you really willing to bet your job, and the experience of your clients, on faith?

25 September 2011 03:46

Peter Sergeant said...

Dawid,

Your comment is on the money when you point out that testing as a developer is hugely important. That's really what this blog is/will be about.

I'm not sure what your comment has to do with the methodology of Test-Driven Development, which is the specific idea that you must write a test for the piece of code you're working on BEFORE you do anything else.

25 September 2011 03:50

MononcQc said...

I think you make a mistake by thinking that prototyping and testing are mutually exclusive. Of course it's useless to write tests when you don't know the specs of your software and what it should do. But then again, why should you write any production software without knowing this?

Prototyping is a[nother] tool to help find out the specs of programs you end up writing. Tests are a way to write those specifications, force you to think as the user of the code rather than its writer. Then this is turned into runnable code to be used as a guideline. That you have tests is more or less accidental in the process.

To me, the best argument about writing tests first is that writing tests last is absolutely boring. Most of the time, it's a half-assed, useless job. Writing tests first is the only way to make it somewhat fun.

25 September 2011 06:38

Javin Paul said...

I agree with your first statement no matter how good a practice ,process or technology is its not ultimate solution.Test driven approach has its own advantage but its not perfect for every scenario. My experience says its flexibility and hybrid nature which gives you option to use agile, waterfall or test driven based on needs and suitability of situation , resources and environment.

25 September 2011 07:48

Dawid Loubser said...

Earlier, I presented the argument that we require unit tests at each level of granularity in our system, to ensure quality and consistency (which is, after all, what we strive for, right?). Nobody has presented a counter-argument, so let's assume this for the moment, and discuss Test-Driven Development (upfront unit tests):

First of all, I don't view TDD as a development methodology in itself, but rather a "technique" (not unlike, say, design-by-contract) which can be used in many development process methodologies, together with other techniques.

There are several overwhelmingly compelling reasons to write one's tests first:

- It enforces a deep understanding of the contract ("requirements") of the component to be written by the developer, which in itself enforces that requirements analysis / design actually be done. How many teams jump right ahead and start coding, only to have to refactor later? Put the overheads where it belongs - requirements analysis.

- It greatly speeds up and simplifies the development process - for the developer now knows precisely when he is "done" (there is no uncertainty, no unnecessary work is done, but nothing is left out). Of course, this depends on having "good" tests (sufficient coverage) - which is a separate and complex topic itself.

- In technology-neutral metalanguages like URDAD or UML, we can express the "dynamics" of requirements sufficiently, but few programming languages (other than WS-BPEL Abstract, which is a bit dead in the water) have artifacts that can express such requirements. Take, for example, a Java interface or WSDL contract: They express only a small part of the requirements. The test suite becomes an essential artifact to express the dynamics (interactions) of the requirements. We should express the requirements *before* implementing them, surely?

- One's framework is then already in place for test-driven bug fixing. Got a bug? Prove it with a unit test. Once proven, fix it (which you know, once your test passes), with the assurance that the other 20 unit tests prove you haven't broken anything else. Nobody is going to start putting those 20 tests in place when under pressure to fix a bug. Luckily they can already be there, and your world does not spiral out of control in a frantic mess of complexity.

Test-driven development introduces a degree of precision, control, and simplicity to the development team that is profound. Of course it's more work, and requires much more insight from the developer.

Two things have held true in the decade or so that I have been teaching this to developers though:

Firstly, absolutely everybody is opposed to the supposed "overheads" of this process: "But we have deadlines!" "We don't have time!"

Secondly, every last developer that adopts the test-driven development process (not because they "have to", but in their hearts) rises to a new plateau of understanding - they speak a different language when it comes the coding, and approach problem solving differently. All of the sudden, it's obvious why other engineering disciplines do things this way as a matter of course. And us software engineers have infinitely better tools than our distant relations in mechanical, chemical, civil engineering - We can automatically build and break components at zero cost!

They never go back to this uncontrolled hacking that most people call "software development".

25 September 2011 12:14

Attila Magyar said...

TDD is not "The One True Programming Methodology", but clearly one of the best if you are doing OO design.

At my previous workplace we were doing a lot of TDD, and at first I didn't like it at all. Later I recognized that we completely misused the whole methodology. The problem was that we didn't allow the test to influence our code, which resulted both unmaintainable code and tests (lots of hacks in the test because the code was not unit testable (and not reusable, not flexible). So insisting on unit testing, but not allowing to change the code because of the test, is two incompatible mindset, which will result lots of stress in the test.

Later I started to get into in TDD more deeply and I learned how can I "listen to the test" and alter my design because of it. Overall, it helped me to understand OO in a better way. (At that time I did not consider myself as an unexperienced programmer, and I though I already know everything about OO, but it was not the case).
So, now I consider TDD as a design process which helps me to develop good OO design.
(I emphasising the OOP intentionally, because I started studying functional programming recently, and I still not sure whether TDD offers the same benefit in that world or not).

Regarding TDD, I recommend a book called: Growing object oriented software guided by tests

25 September 2011 14:41

Mojo said...

This article isn't written "brashly", rather its written with not much merit.

First three paragraphs the author is just trolling.

Fourth paragraph you admit to doing TDD is good sometimes.

Fifth paragraph and you are just taking things too literal. Not all development should be TDD. API Discovery, testing the waters, is okay to not being TDD. There is also other things that cannot be TDD like GUI related work. Also this is not a huge portion of coding so don't hold on the the 20% of development and edge cases and claim TDD sucks based on those.

Sixth paragraph, you again say TDD is good sometimes, then you troll again.

Seventh paragraph, you don't back this claim up.

Eigtht Paragraph....oh fcuk it, I give up...this is useless.

25 September 2011 22:54

Antwan "@ADubb_DC" Wimberly said...

Great post. For those that are caught up in the hype and want to imply that you some how are not a good coder or have poor design skills because you don't strictly follow TDD, you're delusional and you're in severe need of help. Building software is all about trade offs. There is no ONE AND ONLY way to program that beats all. It's all about what you think is best for you and or your team. For example, you can justify all your decisions by starting each sentence off with "But Martin Fowler said_____"....or you can grow some balls and make a decision for yourself. Stop letting people control your lives as a developer and go against the grain sometimes!! You guys crack me up. Calling this man out because he has the balls to admit that sometimes he doesn't think adhering to the process to the T is worth all that it's portrayed to be. What a buncha chumps!! Even Jeffrey Palermo says he and his crew at Headspring don't always write their tests first. They just make sure their tests are committed at the same time their code is. It's good to do initially, but after a while you just know what the hell you're doing. I'm not saying don't write tests...they're helpful as hell...no doubt. But writing them FIRST EACH AND EVERY TIME...naaaaaah!! And that word "trolling" cracks me up. It's become synonymous with..."this guy has the balls to provide a counter argument for a widely supported practice". Get real

people!!

26 September 2011 08:35

Dawid Loubser said...

Antwan, the original author does NOT present a counter-argument - that is the point! If anybody can present an argument (i.e. a conclusion based on logical premises) that test-driven development *reduces* quality or productivity under *any* circumstances, let him come forward.

As it stands, it reads a little like "sometimes I just don't feel like producing good quality work, because it's too much effort. I think striving for good quality sucks."

Do you also sometimes say: "Designing things before I build them EACH AND EVERY TIME... naaaahhhh!!" ? Where does it stop? "Satisfying the customer's requirements EVERY TIME? Naaahhhh!!. Writing code that compiles EVERY TIME? Naaahhhh!!"

Good heavens, man, what kind of software do you build? You are the one that has to "get real".

Did you even read my prvious post? Can you present a counter-argument to any of my statements?

26 September 2011 13:03

Mauricio Aniche said...

I study, practice and research in academia about TDD for a long time. I can say that I am a TDD evangelist. I do believe that TDD makes such a difference in my development environment.

However, you can't "Always" and "Never" in any software engineering context. If someone still thinks that TDD, or any other practice should be done 100% of the time, s/he is wrong.

TDD is a tool as many others that we have. You should use whenever you need it. It is up to the developer to identify moments that he needs to use TDD and moments that he does not need to use. This is what I expect from an experienced developer.

27 September 2011 08:08

Dawid Loubser said...

Quote (Mauricio Aniche):
> However, you can't "Always" and "Never" in
> any software engineering context. If someone
> still thinks that TDD, or any other practice
>should be done 100% of the time, s/he is wrong.

Of course you can! One can logically argue that Test-Driven Development will *always* produce higher-quality output. Of course, higher-quality means slower, more expensive, and requires a stronger class of developers.

One should thus not say "one must always follow test-driven development" just like one should not say "one must always pursue the highest quality".

But if the decision is indeed made to pursue quality - and with very complex software projects and small teams, this is a good idea - nobody has yet presented an argument that test-driven development will not always result in higher quality.

28 September 2011 01:58

P@blo said...

Great article! The article's point is quite clear IMHO, but the way it points out what we are doing wrong is a bit harsh.

I can't agree more with the purpose of automated testing, that of helping you to develop good software. It is of no avail following a cook recipe in the wrong context.

Also, as Hanley, chadastrophic and Magly say you should always strike a balance between benefits it provides and costs, for example when doing a webapp, it is not the same as doing software for a pacemaker.

28 September 2011 07:10

---

reality-analyst said...

Dawid Loubser,

There are plenty of counterarguments possible (and necessary).

First, there are many ways of writing correct code and insuring that already written code is correct. Only a fraction of those involve unit tests. Claiming that TDD is the best practice without comparing (or even knowing) about other practices is pure arrogance.

Second, it is perfectly possible to write horrible code while having 100% unit test coverage. Your code can be unreadable, hard to navigate and overly complex, for example. Another common illness of TDD practitioners is the tendency to sweep bugs under the carpet, moving them to config files and the database. Yes, yes, your code is perfect and super-configurable, but if your application fails to reliably work in real life, I don't really care.

Needless to say, these are not theoretical issues. I'm speaking from experience.

Finally, it's absurd to pretend that "real" engineers always deliver or even want to deliver 100% correct code. If your hardware fails in 1% of all transactions, is it reasonable to attempt to fix some software issue that affects 0.01% of all transactions? What's the cost of fixing hardware? What's the costs of fixing software? What's the cost of failures? That's the kind of reasoning I would expect from a real engineer.

1 October 2011 10:45

Eternally Lost said...

Frankly, most programmers are scared of writing tests, and unfortunately TDD many times has been presented as a rigorous discipline or worse, dogma. This just scares them off even more. While I'm actually an advocate of pragmatic TDD (or even TFD where it makes sense), the real value in TDD is the rapid feedback you get. It reduces the mental load on a developer (and in my professional opinion can produce better designs). For a moment, don't think of the unit test as a test. Instead, think of it as a "mini prototyping environment" where you can quickly write a piece of code, immediately execute it, and see the results. If viewed this way, pragmatic TDD can become a very liberating exercise as you feel less constrained and more willing to explore.

3 October 2011 08:11

Post a Comment

Comment as: Select profile...

Post Comment Preview

Newer Post Home

Subscribe to: Post Comments (Atom)