When/how frequently should I test?

Asked 16 years, 4 months ago Modified 6 years, 8 months ago

Viewed 828 times

4

As a novice developer who is getting into the rhythm of my first professional project, I'm trying to develop good habits as soon as possible. However, I've found that I often forget to test, put it off, or do a whole bunch of tests at the end of a build instead of one at a time.

My question is what rhythm do you like to get into when working on large projects, and where testing fits into it.

testing

Share

Improve this question

Follow

edited Apr 6, 2018 at 23:33

Nathan Smith

681 ● 1 ● 10 ● 24

asked Aug 22, 2008 at 17:54

btw

7,156 ● 9 ● 41 ● 40

12 Answers

Sorted by: Highest score (default)

Well, if you want to follow the TDD guys, **before you start to code** ;)

6

I am very much in the same position as you. I want to get more into testing, but I am currently in a position where we are working to "get the code out" rather than "get the code out right" which scares the crap out of me. So I am slowly trying to integrate testing processes in my development cycle.

Currently, **I test as I code, trying to bust the code as I write it**. I do find it hard to get into the TDD mindset.. Its taking time, but that is the way I would *want* to work..

EDIT:

I thought I should probably expand on this, this is my basic "working process"...

1. Plan what I want from the code, possible object design, whatever.
2. Create my first class, add a huge comment to the top outlining what my "vision" for the class is.
3. Outline the basic test scenarios.. These will basically become the unit tests.
4. Create my first method.. Also writing a short comment explaining how it is *expected* to work.
5. Write an automated test to see if it does what I expect.
6. Repeat steps 4-6 for each method (note the automated tests are in a huge list that runs on F5).

7. I then create some beefy tests to emulate the class in the working environment, obviously fixing any issues.
8. If any new bugs come to light following this, I then go back and write the new test in, make sure it fails (this also serves as proof-of-concept for the bug) then fix it..

I hope that helps.. Open to comments on how to improve this, as I said it is a concern of mine..

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 • 1

answered Aug 22, 2008 at 17:57

Rob Cooper

28.9k • 26 • 105 • 142

2

Before you check the code in.

Share Improve this answer

Follow

answered Aug 22, 2008 at 17:56

N8g

636 • 2 • 8 • 19

First and often. If I'm creating some new functionality for the system I'll be looking to initially define the interfaces

1

and then write unit tests for those interfaces. To work out what tests to write consider the API of the interface and the functionality it provides, get out a pen and paper and think for a while about potential error conditions or ways to prove that it is doing the correct job. If this is too difficult then it's likely that your API isn't good enough. In regards to the tests, see if you can avoid writing "integration" tests that test more than one specific object and keep them as "unit" test.

Then create a default implementation of your interface (that does nothing, returns rubbish values but doesn't throw exceptions), plug it into the tests to make sure that the tests fail (this tests that your tests work! :)). Then write in the functionality and re-run the tests. This mechanism isn't perfect but will cover a lot of simple coding mistakes and provide you with an opportunity to run your new feature without having to plug it into the entire application.

Following this you then need to test it in the main application with the combination of existing features. This is where testing is more difficult and if possible should be partially outsourced to good QA tester as they'll have the knack of breaking things. Although it helps if you have these skills too. Getting testing right is a knack that you have to pick up to be honest. My own experience comes from my own naive deployments and the subsequent bugs that were reported by the users when they used it in anger.

At first when this happened to me I found it irritating that the user was intentionally trying to break my software and I wanted to mark all the "bugs" down as "training issues". However after reflecting on it I realised that it is our role (as developers) to make the application as simple and reliable to use as possible even by idiots. It is our role to empower idiots and thats why we get paid the dollar. Idiot handling.

To effectively test like this you have to get into the mindset of trying to break everything. Assume the mantle of a user that bashes the buttons and generally attempts to destroy your application in weird and wonderful ways. Assume that if you don't find flaws then they will be discovered in production to your companies serious loss of face. Take full responsibility for all of these issues and curse yourself when a bug you are responsible (or even part responsible) for is discovered in production.

If you do most of the above then you should start to produce much more robust code, however it is a bit of an art form and requires a lot of experience to be good at.

Share Improve this answer

Follow

answered Aug 22, 2008 at 18:09

Quibblesome

25.4k ● 10 ● 62 ● 104

A good key to remember is

1

> "Test early, test often and test again, when you think you are done"

Share Improve this answer

Follow

answered Aug 22, 2008 at 18:46

Pascal

4,127 ● 8 ● 34 ● 29

1

When to test? When it's important that the code works correctly!

Share Improve this answer

Follow

answered Sep 20, 2008 at 5:22

Andy Lester

93.5k ● 15 ● 104 ● 159

0

When hacking something together for myself, I test at the end. Bad practice, but these are usually small things that I'll use a few times and that's it.

On a larger project, I write tests before I write a class and I run the tests after every change to that class.

Share Improve this answer

Follow

answered Aug 22, 2008 at 17:56

Thomas Owens

116k ● 99 ● 317 ● 436

0

I test constantly. After I finish even a loop inside of a function, I run the program and hit a breakpoint at the top of the loop, then run through it. This is all just to make sure that the process is doing exactly what I want it to.

Then, once a function is finished, you test it in it's entirety. You probably want to set a breakpoint just before the function is called, and check your debugger to make sure that it works perfectly.

I guess I would say: "Test often."

Share Improve this answer

Follow

answered Aug 22, 2008 at 18:03

EndangeredMassa

17.5k ● 8 ● 56 ● 80

0

I've only recently added unit testing to my regular work flow but I write unit tests:

- to express the requirements for each new code module (right after I write the interface but before writing the implementation)
- every time I think "it had better ... by the time I'm done"
- when something breaks, to quantify the bug and prove that I've fixed it
- when I write code which explicitly allocates or deallocates memory---I loath hunting for memory leaks...

I run the tests on most builds, and always before running the code.

Share Improve this answer

Follow

answered Aug 22, 2008 at 18:03

dmckee --- ex-moderator kitten

101k ● 25 ● 146 ● 235

0

Start with unit testing. Specifically, check out TDD, Test Driven Development. The concept behind TDD is you write the unit tests first, then write your code. If the test fails, you go back and re-work your code. If it passes, you move on to the next one.

I take a hybrid approach to TDD. I don't like to write tests against nothing, so I usually write some of the code first, then put the unit tests in. It's an iterative process, one which you're never really done with. You change the code, you run your tests. If there's any failures, fix and repeat.

The other sort of testing is integration testing, which comes along later in the process, and might typically be done by a QA testing team. In any case, integration testing addresses the need to test the pieces as a whole. It's the working product you're concerned with testing. This one is more difficult to deal with b/c it usually involves having automated testing tools (like Robot, for ex.).

Also, take a look at a product like CruiseControl.NET to do continuous builds. CC.NET is nice b/c it will run your unit tests with each build, notifying you immediately of any failures.

Share Improve this answer

Follow

answered Aug 22, 2008 at 18:05

Scott Marlowe

8,025 ● 12 ● 46 ● 52

0

We don't do TDD here (though some have advocated it), but our rule is that you're supposed to check your unit tests in with your changes. It doesn't always happen, but it's easy to go back and look at a specific changeset and see whether or not tests were written.

Share Improve this answer

Follow

answered Aug 22, 2008 at 19:00

Adam V

6,336 ● 3 ● 41 ● 52

0

I find that if I wait until the end of writing some new feature to test, I forget many of the edge cases that I thought might break the feature. This is ok if you are doing things to learn for yourself, but in a professional environment, I find my flow to be the classic form of: Red, Green, Refactor.

Red: Write your test so that it fails. That way you know the test is asserting against the correct variable.

Green: Make your new test pass in the easiest way possible. If that means hard-coding it, that's ok. This is great for those that just want something to work right away.

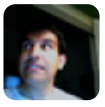
Refactor: Now that your test passes, you can go back and change your code with confidence. Your new change broke your test? Great, your change had an implication you didn't realize, now your test is telling you.

This rhythm has made me speed my development over time because I basically have a history compiler for all the things I thought that needed to be checked in order for a feature to work! This, in turn, leads to many other benefits, that I won't get to here...

Share Improve this answer

Follow

answered Sep 10, 2008 at 15:42



casademora

69.5k ● 18 ● 71 ● 78

0

Lots of great answers here!

I try to test at the lowest level that makes sense:

- If a single computation or conditional is difficult or complex, add test code while you're writing it and ensure each piece works. Comment out the test code when you're done, but leave it there to document how you tested the algorithm.
- Test each function.

    - Exercise each branch at least once.
    - Exercise the *boundary conditions* -- input values at which the code changes its behavior -- to catch "off by one" errors.
    - Test various combinations of valid and invalid inputs.
    - Look for situations that might break the code, and test them.
- Test each module with the same strategy as above.
- Test the body of code as a whole, to ensure the components interact properly. If you've been diligent about lower-level testing, this is essentially a "confidence test" to ensure nothing broke during assembly.

Since most of my code is for embedded devices, I pay particular attention to robustness, interaction between various threads, tasks, and components, and unexpected use of resources: memory, CPU, filesystem space, *etc*.

In general, the earlier you encounter an error, the easier it is to isolate, identify, and fix it--and the more time you get to spend creating, rather than chasing your tail.*

**I know, -1 for the gratuitous buffer-pointer reference!*

answered Oct 29, 2008 at 1:14

Adam Liss
48.3k ● 13 ● 113 ● 152