```
import {
  Learning,
  JavaScript,
  AngularJS
} from 'Brad on Code.com'
```


Developer Testing - Why should developers write tests?

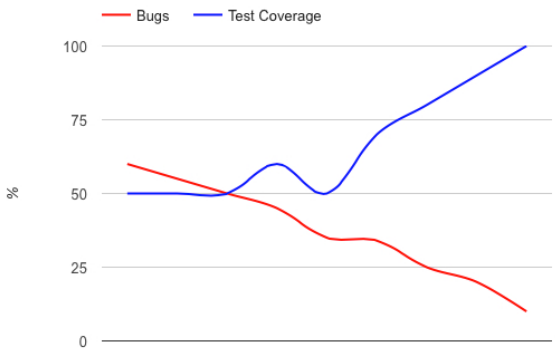
What are the different types of tests for software, and why should developers write them?

Bradley Braithwaite on May 10, 2015 on *musings, testing*

The section about testing from the AngularJS documentation has the following sentence which caught my attention (with emphasis mine):

> *Angular is written with testability in mind,* ***but it still requires that you do the right thing****. We tried to make the right things easy, but if you ignore these guidelines you may end up with an untestable application.*

But, what's the right thing? Those few words are packed full of subtle nuances and trade-offs that can lead to high-profile nerd wars and heated twitter exchanges. I plan to write a lot more about testing with AngularJS on this blog (subscribe to the RSS feed if you would like to keep up-to-date), but before I start writing technical posts I want to address why I think tests are important. I personally think that you have nothing to lose in getting started… perhaps just the bugs in your code!



The effect of test coverage?

Different Types of Tests

The common types of tests that developers can write for applications are:

1. Unit Tests
2. Integration Tests
3. Regression Tests
4. System Tests

1. Unit Tests

Unit Testing is the execution of a section of code or small program which is tested in isolation from the more complete system. Tested in isolation means not calling the implementation of code not under test e.g. database, web service calls or other code dependencies. The concept of isolation is why mocking frameworks are commonly used for unit tests. These would be written by developers during the development of a feature.

2. Integration Tests

Integration Testing is the combined execution sections of code. In these types of tests you would hit the database, make web service calls or call other code dependencies. These would be written by developers. These would be written by developers during the development of a feature.

3. Regression Tests

Regression testing is the repetition of previously executed test cases for the purpose of finding defects in software that

previously passed the same set of tests. Such tests would commonly be used before shipping code to a new environment or as part of a build process. It's common to see tools such a selenium used to write these types of tests, where a web browser would be launched and user input automated. Human testers can also perform regression tests by using an application directly.

4. System Tests

System testing is the execution of the software in its final configuration, including integration with other software and systems. It tests for security, performance, resource loss, timing problems, and other issues that can't be tested at lower levels of integration. As with regression testing, automated tools such as selenium can be used for this process as well as human testers.

Different Approaches to Writing Developer Tests

Developer tests are written whilst the developer is writing the actual functionality. This process is tightly integrated so that the developer can work in a flow of writing tests and the logic. There are two widely adopted approaches for writing developer tests:

1. Test Driven Development (TDD)
2. Behaviour Driven Development (BDD)

These different approaches are often the main point of contention amongst developers when setting out a testing strategy. Note that developer tests do not consider Regression or System Tests. Developers can also write code to automate Regression and/or System Tests, but that would come at a later stage in the development cycle when the application is in a more complete state. Regression or System Tests are sometimes even written by a different developer who specialises in writing regression tests and views the system as a black-box i.e. is not aware of the internal workings.

1. Test Driven Development - TDD

Using **TDD you write Unit Tests** that are driven by the implementation detail of the code. With unit tests, we must test things in isolation. If following the style as intended, the developer must code in a *"Red, Green, Refactor"* style, where they first write a test, watch the test fail (since the logic is not yet implemented), then write enough code to make the test pass, before refactoring the code and ensuring that the test still passes. Once a developer has finished working through a feature they will be left with a suite of passing tests verifying the correctness of their implementation.

2. Behaviour Driven Development - BDD

Using **BDD you write Integration Tests** that are driven by application behaviour which is in contrast to TDD which concerns itself with the implementation detail. You are also not restricted to testing in isolation as with TDD. In fact, BDD has emerged from TDD which has been around for longer. As with TDD, you would write tests before implementing the logic and make each test pass when moving through the implementation. Once a feature is completed the developer is left with a suite of passing tests verifying the correctness of the behaviour of the code.

Why I like to write Developer Tests

What I like to get from developer tests, regardless of approach (TDD or BDD) is to:

1. Write Fewer Bugs
2. Avoid Manual Testing
3. Get a Faster Feedback Loop
4. Increased Confidence when Changing Code

1. Write Fewer Bugs

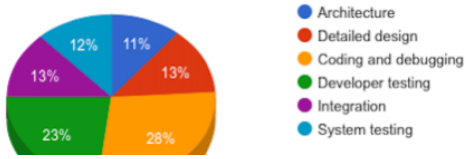
This seems like an obvious point, but I've had discussions with developers who do not write tests as in their view, developers should check their work anyway and that they have testers who can catch regressions, therefore the tests are unnecessary additional code. A study by Glenford Myers in 1978 had a group of experienced programmers test a program with 15 known defects. **The average programmer only found 5 of the bugs**. In the study the main source of the errors that went unnoticed was erroneous output not being looked at closely enough. The conclusion of this? Humans are error-prone.

It's important to recognise that writing tests does not guarantee zero bugs. There is still plenty that can go wrong such as database problems, mistakes with functionality, integration problems and so on, so there will always be a need for testers.

2. Avoid Manual Testing

The process of testing code after making changes can be very time consuming, and as we've already established, is prone to human error. The book Code Complete indicates that **developer testing takes 8 to 25 percent of the total project time** dependant on project size and complexity. That's 16.5% on average, or around an hour per 8 hour working day. This means that we developers will spend a lot of time testing the code we write, so any approach to streamline this process is worth serious consideration.

The following pie chart is the approximate percentage breakdown of developer time spent on a project with 2,000 lines of code:



Percentage of Development Time for 2k lines of code

3. Get a Faster Feedback Loop

A feedback loop is the cycle of adding some code, getting feedback about the change i.e. is ok/not ok, then repeating. For me, the faster this cycle, the more productive I am. Increasing productivity and reducing the number of bugs we write sounds like an idealistic programmer utopia, but once a programmer cultivates a flow for writing tests I believe that this is possible.

4. Increased Confidence when Changing Code

Have you worked on a feature and spotted something you thought could be improved with the existing code, but chose not to correct it because you didn't want to risk creating more work for yourself by potentially breaking something? Good test coverage provides a safety net for making such changes, making you and the other developers in your team more likely to refactor that code smell. The net effect of this is a better code-base for you all.

Tests by themselves do not improve software quality. The process of writing code that is testable, and the safety net of being able to refactor code and (almost) be sure that it does not result in a regression, is where the improvements occur.

Summary

This post has introduced the main concepts of writing developer tests, and how the types of tests integrate with the approaches for writing tests. The high-level differences between TDD and BDD have been discussed and all that's left now is to start looking at code examples. The code examples will come in future posts where I will be covering testing using AngularJS extensively.

My thoughts on writing tests are aligned with this quote from Growing Object-Oriented Software, Guided by Tests:

> *If we write tests all the way through the development process, we can build up a safety net of automated regression tests that give us the confidence to make changes.*

Subscribe to the RSS feed if you want to learn about developer testing with AngularJS.

CONNECT WITH BRADLEY

Bradley Braithwaite is a software engineer who works for search engine start-ups. He is a published author at pluralsight.com. He writes about software development practices, JavaScript, AngularJS and Node.js via his website bradoncode.com. Find out more about Brad. Find him via:

You might also like:

ALSO ON BRADONCODE

6 years ago · 2 comments

How to use the ngMock Dump function for debugging with Unit Tests ...

6 years ago · 2 comments

Part 1 of a two part series taking a deeper dive into the two key concepts of ...

6 years ago · 9 comments

How to unit test AngularJS code that uses the $http service with ngMock. Part ...

6 years ago · 1 comment

Estimating Software Development Tasks Made (a little bit) Easier

6 years

A quic unit te contro

Rudeus Greyrat • 6 years ago

Nice explanation. I hope I'll be able to fully integrate testing (automatic tests that is) into my development cycle