What’s the difference between Unit Testing, TDD and BDD?

TAGS: TESTING (HTTPS://CODEUTOPIA.NET/BLOG/TAG/TESTING/)

When you’re just getting started with automating your JavaScript testing, there’s a lot of questions. You’ll probably see people talk about unit testing, TDD or Test-Driven Development, and BDD or Behavior-Driven Development. But which one of them is the best approach? Can you use all of them?

I’ve talked to a number of JavaScript developers, and there seems to be some confusion about all this. So, let’s take a look at Unit testing, TDD and BDD, and fix some of the common misconceptions about them out there.

A korean translation of this article can be found here (https://medium.com/@sryu99/%EB%8B%A8%EC%9C%84-%ED%85%8C%EC%8A%A4%ED%8A%B8-tdd-bdd%EC%9D%98-%EC%B0%A8%EC%9D%B4%EC%A0%90-3d25fab5ccb2)

Unit testing

A unit test focuses on a single “unit of code” – usually a function in an object or module. By making the test specific to a single function, the test should be simple, quick to write, and quick to run. This means you can have many unit tests, and more unit tests means more bugs caught. They are especially helpful if you need to change your code: When you have a set of unit tests verifying your code works, you can safely change the code and trust that other parts of your program will not break.

A unit test should be isolated from dependencies – for example, no network access and no database access. There are tools that can replace these dependencies with fakes you can control. This makes it trivial to test all kinds of scenarios that would otherwise require a lot of setup. For example, imagine having to set up an entire database just to run a test. Ugh, no thanks.

There is a misconception that unit tests require a specific syntax to write. This so-called "xUnit style" syntax is common in many slightly older testing tools. Below is an example of the "xUnit style", using Mocha:

```
suite('My test suite name', function() {
  setup(function() {
    //do setup before tests
  });

  teardown(function() {
    //clean up after tests
  });

  test('x should do y', function() {
    //test something
  });
});
```

But this is just an example of what a tool looks like. You don't have to use any specific syntax for unit tests – in fact, you can even write unit tests with plain JavaScript:

```
//suite: User

//test: Name should start empty
var user = new User();
if(user.getName() !== '') {
  throw new Error('User name should start as empty');
}

//test: Password should be hashed
var user = new user();
user.setPassword('hello');
if(user.getPassword() != bcrypt('hello')) {
  throw new Error('User password should be hashed with bcrypt');
}
```

The basic pieces of a unit test are there: Individual tests, which test one thing, and they are isolated from each other. You could use scripts like this to build rudimentary unit tests for your code. But using an actual unit testing tool such as Mocha or Jasmine will make it easier to write tests, and they have other helpful features such as better reporting when tests fail (which makes it easier to find out what went wrong)

Some think that any automated test is a unit test. This is not true. There are different types of automated tests, and each type has its own purpose.

Here are three of the most common types of automated tests:

- **Unit tests**: A single piece of code (usually an object or a function) is tested, isolated from other pieces
- **Integration tests**: Multiple pieces are tested together, for example testing database access code against a test database
- **Acceptance tests (also called Functional tests)**: Automatic testing of the entire application, for example using a tool like Selenium to automatically run a browser.

If you feel it's not easy to write a unit test, chances are it's not a unit test at all. Both integration tests and acceptance tests are more complex and usually run slower. They are also more difficult to maintain than unit tests, so if you're having problems, make sure you're writing the right kind of test.

Do you want the best tips on how to unit test your JavaScript code? Let me send you my JavaScript unit testing guide (https://codeutopia.net/h/subscribe/)

TDD

TDD or Test-Driven Development is a process for when you write and run your tests. Following it makes it possible to have a very high test-coverage. Test-coverage refers to the percentage of your code that is tested automatically, so a higher number is better. TDD also reduces the likelihood of having bugs in your tests, which can otherwise be difficult to track down.

The TDD process consists of the following steps:

1. Start by writing a test
2. Run the test and any other tests. At this point, your newly added test should fail. If it doesn't fail here, it might not be testing the right thing and thus has a bug in it.
3. Write the minimum amount of code required to make the test pass
4. Run the tests to check the new test passes
5. Optionally refactor your code
6. Repeat from 1

It can take some effort to learn well, but spending the time can pay off big. TDD projects often get a code-coverage of 90-100%, which means maintaining the code and adding new features is easy. This is because you have a large set of tests, so you can trust your code and changes work, and didn't break any other code either.

Some think you must use the "xUnit style" testing tools to use the TDD process. This is not the case – TDD works great with unit tests, but you can apply it to other testing methods as well. It also does not require any specific tool or syntax.

The most difficult thing about TDD for many developers is the fact you have to write your tests before writing code. Check here for my 5 step method to make TDD easy (https://codeutopia.net/blog/2016/10/10/5-step-method-to-make-test-driven-development-and-unit-testing-easy/)

BDD

BDD – Behavior-Driven Development – is perhaps the biggest source of confusion. When applied to automated testing, BDD is a set of best practices for writing great tests. BDD can, and should be, used together with TDD and unit testing methods.

One of the key things BDD addresses is implementation detail in unit tests. A common problem with poor unit tests is they rely too much on how the tested function is implemented. This means if you update the function, even without changing the inputs and outputs, you must also update the test. This is a problem because it makes doing changes tedious.

Behavior-Driven Development addresses this problem by showing you how to test. You should not test implementation, but instead behavior. Let me show you an example of what happens when you think of implementation, and when you think of behavior:

```
suite('Counter', function() {
  test('tick increases count to 1', function() {
    var counter = new Counter();

    counter.tick();

    assert.equal(counter.count, 1);
  });
});
```

This is a unit test of an imaginary counter object. We test that after calling tick, the value should be 1, which sounds like it makes sense. But there's a problem in the test. The test is completely dependent on the fact that the counter starts at 0. So in other words, this test is relying on two things.

1. Counter starts at 0
2. Ticking increments by 1

The fact the counter starts at 0 is an implementation detail that's irrelevant to the behavior of the tick() function. Therefore it should not have any bearing on the test. The only reason we wrote the test like this is because we were thinking of the implementation, not of the behavior.

BDD suggests to test behaviors, so instead of thinking of how the code is implemented, we spend a moment thinking of what the scenario is. Typically you phrase BDD tests in the form of "it should do something". So when ticking a counter, it should increment count by one.

The important part here is thinking of the scenario, rather than the implementation, can lead you to design a better test.

```
describe('Counter', function() {
  it('should increase count by 1 after calling tick', function() {
    var counter = new Counter();
    var expectedCount = counter.count + 1;

    counter.tick();

    assert.equal(counter.count, expectedCount);
  });
});
```

In this version of the test, which uses Mocha's BDD style functions, we removed the implementation detail. Instead of relying on the counter starting at 0, we are comparing against counter.count + 1 which makes much more sense in terms of testing against behavior.

Sometimes your requirements change. Let's imagine that for some reason the Counter has to start at some other value. Before, we would have to change the test to accommodate that, but with the BDD-variant there is no need to do so.

Conclusion

Unit Testing gives you the what. Test-Driven Development gives you the when. Behavior Driven-Development gives you the how. Although you can use each individually, you should combine them for best results as they complement each other very nicely.