# Unit Testing 101: Harder Better Faster Stronger

# What it is

There are three generally accepted steps to testing a software application.

## Unit testing

Unit testing is a process in which the smallest testable parts of an application are individually and independently scrutinized for proper operation.

## Integration testing

Integration testing is a process in which individual software modules are combined and tested as a group.

## Validation testing

Verification and validation is the process of checking that a software system meets specifications and that it fulfills its intended purpose. It may also be referred to as software quality control.

If your team skips one or two of these steps you likely to experience preventable failures. That costs money.

Unit testing is a way to streamline the creation of accurate and maintainable software. It reduces the difficulty of reproducing problematic scenarios and encourages the use of best practices. Because these tests can be run quickly over large data sets or scenarios; they can provide a level of confidence and assurance in the software that is unattainable with manual testing alone.

The goals behind unit testing are to:

1. meet the expectations of our clients
2. avoid working under emergency situations
3. avoid losing face

These goals serve the higher goal of providing value to our clients, which supports our life style.

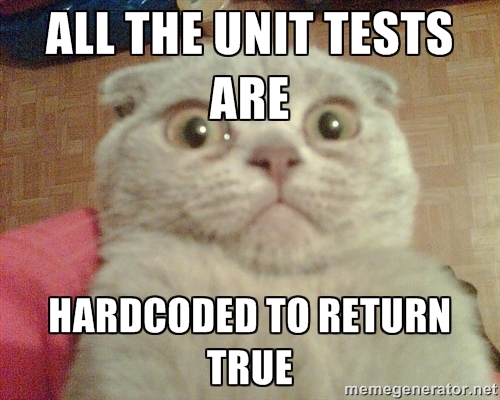
# What to expect

Good tests come from quality scenarios and the definition of edge cases. This takes teamwork because the code will only be as good as the tests and the tests are only as good as both the requirements and the use case scenarios.

The entire job of deciding what should and shouldn’t get tested shouldn’t set on the shoulders of a single developer at design time. If this is the way your team works, your mileage may vary.

Unit testing is not a fail-safe methodology.

A test only tests what it tests



# Doesn’t it take too long?

How many times have testers found bugs that should have never made to the QA environement?

What if you were asked to prove that what you did works before you used up the time of someone doing validation testing?

Your code isn't finished until you have verified it works.

Imagine a website that had different content displayed for different user types. How would you test that? How many times did would you log in as a different type of user? Ten times, one hundred times? How long does it take to log in every time? How many times did you have to restart your browser or your development environment because of this?

That’s slow. That costs money. It is not repeatable. If you change something the company has to buy that testing experience again.

And I bet you weren’t writing down what you did methodically and making sure that you tested everything. So, you probably missed something.

When we do manual testing in a browser, we generally don’t cover edge cases, though sometimes we cover only the edge cases.

We don’t have enough time in the day to rigorously test code manually. How long does it take a computer to count to a million? How long does it take you? Now think about having to do that every time you deploy the code.

Have you ever had some code that you weren’t sure worked? How did you decide that it was trustworthy?

Many web developers test the output of their algorithms in a web browser. Why are we doing this the hard way? Wouldn’t it be more efficient to write a test for this? This way we create a record of the fact that this bit was tested and this could be shown to a client as proof of due diligence.

# Won’t I have to update my tests as the project grows?

Yes, and that’s ok.

Did you plan out the project before you wrote the code? Did you imagine the project as you went, making constant schema changes? There is something to be said for planning schema ahead of time. The Code First mentality can contribute to a lack of planning that can make Unit Testing more cumbersome.

Schema changes will be made under any methodology and the unit tests will break. That’s the point. If the test stops working, then you may very well have introduced a bug and that needs to be addressed by updating the tests.

# Accuracy and Maintainability

Accuracy: Purposefully set out to create a solution that can be proven to be accurate. Do your clients expect accurate code? Do you owe it to them?

Do your clients ask for proof of code accuracy? Can a manual quality assurance process achieve the burden of proof that we’re given? This is about accurate business logic, rather than markup.

The quality assurance process should take into account the repeatable and non-manual exercise of code. That job currently sits with the developers of the code and not with an outside department.

A unit test is a concrete physical location where current and future issues with the chosen solution can be addressed.

# Keep it simple

There is no way achieve full code coverage.

Don’t bother with full code coverage. It won’t happen, so don’t try. Instead, attempt to identify those areas that are most likely to be complex or difficult to properly test manually.

Takes too long

What are you testing? Don’t write tests to test your frameworks. If you really think you need to do this, maybe you need better frameworks.

Debugging issues in production with users freaking out is way more disruptive and time consuming than a little test.

# Good Practices in Relation to Testing

MVC is a good pattern, but by itself it is not sufficient for an entire website that does something non-trivial. We should not have to run a website in order to exercise standard CRUD operations. Controllers are not nice little places to write business logic. Controllers are for tying together dependencies and routing requests to the proper calls into the application.

Use layers. We separate concerns because the isolation of concepts makes finding, maintaining and testing code easier. Using interfaces and Dependency Injection are great for flexibility and mocking. This has become rather common and relatively simple to implement in the .Net Framework, especially with MVC. Some developers, however still find them cumbersome or overly complex.

Think of the business layer of the application as an API and only expose what is necessary. The need to testing might sometimes create the desire to make every property public, but this is not the right spirit. A better option is to have the primary endpoints return objects that more thoroughly describe what happened when the code ran. This way a client, whether a user or an application can react to whatever the situation at hand may be.

Tight coupling is bad. It makes it hard to grow. It makes it hard to refactor. It makes it hard to abstract or generalize. Specifically, ORM should not be tightly coupled with website code or even the business layer. This creates leaky and unmanageable code. It should be in its own layer. When testing you should be able to configure the option of whether or not to even engage with the database based on the needs of the test.

Use a build server. This can be great because the developer does not have to be present, or even involved in deployment of the application. Tests can be run prior to deployment and deployment can be halted if the tests fail. But, if you are using a build server to test and deploy your code, you should be careful about what these tests do, because all of the tests that you are going to run have the potential to have an effect on the production database.

Test-driven development is often misunderstood as highly dogmatic and impractical. This is a misunderstanding of the discipline. One must not write tests prior to writing any code. The idea is to conceive of a test before you write a feature. This helps a developer think about the business logic as an API.

# Naming

It would be really confusing if you named all the tests Test1, Test2, etc. You knew that, but what is a good name for a test?

I often group tests into a test file, and from there make each test exercise a particular aspect of the feature being tested.

This has been suggested by many. I’ve found it to be pretty clear and usable.

UnitOfWork\_StateUnderTest\_ExpectedBehavior

1. Divide\_ZeroAs1stParam\_ExceptionThrown()

2. Divide\_ZeroAs2ndParam\_ExceptionThrown ()

3. Divide\_StandardEvenFractionalValues\_Calculated ()

# Assertions

Plan the feature.

Conceive the use case scenario.

Conceive the test.

Write the test?

The whole point is to exercise the code, examine the results and make assertions based on the expected result of the method call.

# References

There are too many to call out, but here’s a good blog post.

<http://codesheriff.blogspot.com/2011/03/excuse-1-writing-tests-takes-too-much.html>