A Response to “Why Most Unit Testing is Waste”

henrikwarne.com/2014/09/04/a-response-to-why-most-unit-testing-is-waste

September 4, 2014

A few months ago I came across the article Why Most Unit Testing is Waste by James O Coplien. The title is an accurate description of the contents – James considers most unit tests to be useless. He expands his arguments in the follow-up article. I was quite intrigued, since I get a lot of value from unit tests. How come we have such different views of them? Had I missed something? As it turns out, I was not persuaded by his arguments, and here is my response to the articles.

The main thesis in the articles is that integration tests are better than unit tests, to the point where integration tests should replace unit tests. I agree that integration tests are effective, but I think a combination of both is even better.

When to Unit Test

In my experience, unit tests are most valuable when you use them for algorithmic logic. They are not particularly useful for code that is more coordinating in its nature. Coordinating code often requires a lot of (mocked) context for unit testing, but the tests themselves are not very interesting. This type of code does not benefit a great deal from unit testing, and is instead better tested in integration testing. For more on the different types of code, see Selective Unit Testing – Costs and Benefits.

An example of algorithmic logic is a pool of temporary numbers in a mobile phone system. The telephone numbers in the pool are used for routing calls in the network. Typically a request is made to get a temporary number, it is used for a few hundred milliseconds, and is then released again. While it is used, it is marked as busy in the pool, and when it is released, it is marked as free in the pool (and can then be handed out for another call). Because there is no guarantee that the release request will be received (due to network errors), the pool needs a time-out mechanism as well. If a number has been marked busy for more than say 10 seconds, it will time-out and be marked as free, despite not receiving a release request. The pool functionality is well suited for unit testing. One way to handle the time aspect is to make it external, see TDD, Unit Tests and the Passage of Time.

Why Unit Test?

Well-tested parts. When testing the complete application, it is an advantage to use well-tested parts. It is the classic bottom-up approach of composing functionality from simpler parts. In the example above, if the pool functionality has been unit tested, I can concentrate on making sure it works well with the rest of the system when testing the complete system. If it was not unit tested, I might still find problems with how the pool works, but it could take more effort to find where in the code the problem is, since it could be anywhere.

Decoupled design. When you design the building blocks of your system to be unit tested, you automatically separate the different pieces as much as possible. If you don't, unit testing becomes quite difficult, often requiring the set-up of a complex environment. I was quite surprised at how much better separated my code become when I started writing unit testing as I was developing. This is also why retrofitting unit tests is so hard – the parts of the pre-existing system are usually all tangled together.

Rapid feedback. Sometimes the complete feature you are working on may take a while to finish, or it requires other parts to be done first. Thus it is not possible to integration test it at each step of the way. With unit tests, you get feedback immediately if you make a mistake. For example, I never make off-by-one errors anymore. As soon as I write the code, I have tests that check the values in the critical range to make sure it works correctly.

Context. Sometimes it is easier to set up the context you want in a unit test than it is to set it up in the complete system. In the example with the pool, it is easy to create a small pool, fill it up with requests, and then check the behavior when there are no free numbers in the pool. Creating a similar situation in the complete system is harder, especially when the time the temporary numbers are in use is small.

Flawed Arguments

I think James misunderstands or misrepresents unit tests in several ways:

Delete tests that haven't failed in a year. James argues that unit tests that haven't failed in a year provide no information, and can be thrown out. But if a unit test fails, it fails as you are developing the code. It is similar to a compilation failure. You fix it immediately. You never check in code where the unit tests are failing. So the tests fail, but the failures are transient.

Complete testing is not possible. In both the original and follow-up article, James talks about how it is impossible to completely test the code. The state-space as defined by {P*rogram Counter, System State*} is enormous. This is true, but it applies equally to integration testing, and is thus not an argument against unit testing.

We don't know what parts are used, and how. In the example of the *map* in the follow-up article, James points out that maybe the map will never hold more than five items. That may be true, but when we do integration testing we are still only testing.

Maybe we will encounter more than five items in production. In any case, it is prudent to make it work for larger values anyway. It is a trade-off I am willing to make: the cost is low, and a lot of the logic is the same, whether the maximum usage size is low or high.

What is correct behavior? James argues that the only tests that have business value are those directly derived from business requirements. Since unit tests are only testing building blocks, not the complete function, they cannot be trusted. They are based on *programmers' fantasies about how the function should work*. But programmers break down requirements into smaller components all the time – this is how you program. Sometimes there are misunderstandings, but that is the exception, not the rule, in my opinion.

Refactoring breaks tests. Sometimes when you refactor code, you break tests. But my experience is that this is not a big problem. For example, a method signature changes, so you have to go through and add an extra parameter in all tests where it is called. This can often be done very quickly, and it doesn't happen very often. This sounds like a big problem in theory, but in practice it isn't.

Asserts. James recommends turning unit tests into asserts. Asserts can be useful, but they are not a substitute for unit tests. If an assert fails, there is still a failure in the production system. If it assert on something that can also be unit tested, it is better to find the problem when testing, not in production.

Conclusion

Despite not agreeing with James on the value of unit tests, I enjoyed reading his articles. I agree with many things, for example the importance of integration testing. And where we don't agree, he made me articulate more clearly what I think, and why.

Unit tests are not a goal in themselves. They are not useful in every situation. But my experience is that they are very useful a lot of the time, and I consider unit testing one of my key software development techniques.