How test driven development helps keep you in the driver's seat

by Stefan Riedmann - Wednesday, July 19, 2017

http://blog.belatrixsf.com/how-test-driven-development-helps-keep-you-in-the-drivers-seat/

Stefan Riedmann is a senior-software developer at Belatrix. This blog was originally published on his personal blog.

I think there's an expression that fits pretty well to what I have to say…

What goes around, comes around!

One of my basic principles in development is the circle, and that's also where the name of my personal blog comes from. I don't say that this is something new that we should apply to development. Truth is, we all develop and program in circles! The problem in many cases is just that people are not aware of it, or that the circles are out of control. And this leads to nightmares regarding planning and quality. Let me explain this based on the most recent task in which I played a part. If you don't want to get too much into the details of unit testing, just skip the *Solution* part and read directly my *Conclusion*.

The challenge

Stage 1

Many programmers see a task or get a ticket, think a while about it, and start programming. When this is a little bit more than writing a simple calculator, it happens just so often that he or she is hacking like hell for days or even weeks, before even being able to really test and debug any of it. And why is that? Because the solution is a big stack with two ends – not necessarily a complete stack with an UI and a database, it can also be just a part of a bigger stack. Anyway, it often has a lot of interfaces that it depends on. An example of my recent experience:

- Data and user input coming from a web front end
- Long and complex validation logic of the data
- Generation of a complex response object
- Saving a document containing the original data and the validation results in a blob storage
- Functionality for the user to retrieve that document in some other moment

Stage 2

When finally reaching the point of filling the stack until all the interfaces (upper and lower end), the next huge challenge starts: Debugging and fixing all the little pieces that one wrote in many hours of coding. For this, one has to set up the whole stack usually (from the UI until the database and the blob), and in a

debuggable DEV environment of course. Then, every test needs manual data input, user interaction and result check. And when it finally works, you can be sure that you're still not done – by far…

Every piece of software that receives input and produces output usually has many different combinations and execution paths. Usually that's one or a few 'happy paths', and many, many 'sad paths'. I saw it happen often, and it also happened to me often enough, that a programmer (after going through all this work) tests the happy path that the ticket was all about, and deploys the whole thing for testing.

Stage 3

A good tester then, after performing one or two test runs, already sees the whole thing breaking apart, starts cursing about the programmers, and starts throwing tickets (bugs) back to the source of the evil piece of software. The reason is the many different ways a real user interacts with the software. And the many different combinations of data inputs. With any luck, the tester was in a good mood and provides precise test data, to reproduce the error.

Based on that, the programmer returns to stage 2, sets up the DEV environment again (which is a lot of fun if he or she has already switched over to another construction site), fixes the software based on the provided test case, and gives it back to stage 3. You can imagine, this circle between 2 and 3 can go on forever. And if the tester didn't see a fairly common sad path, the whole (unstable) thing goes to production and to real users…

Stage 4

But still, even after successfully stabilizing the thing in this time-consuming circle between 2 and 3, we still didn't think at all about testing it automatically. The necessity of it should not have to be discussed anymore nowadays… not just because it's a best practice, but also from a regression standpoint (does it stay stable when changing some details or versions of underlying libraries?). So, at some point, the programmer (or their team) decides that a certain test coverage should be fulfilled for this piece. And just after starting to write unit tests, their stress levels rise at an alarming rate. This happens when they realize that the different pieces of the software are absolutely not testable!

Symptoms of non-testable code

What does that mean? When looking at a class separately from the rest, you should be able to understand its functionality without studying the classes and interface implementations that it uses. It should be self-explaining. In a modern implementation, one of the most important principles is to use interfaces and Dependency Injection. This allows you to easily replace (and understand) the surrounding pieces of a class with mocked objects.

Also, when you're actually writing a unit test, and end up needing dozens of lines of code just to prepare ('arrange') unit tests, you can also see that your code must be improved. This preparation code can be

- Mocking behaviors of interfaces
- Writing and filling huge data objects with test data

But still, just following these principles and avoid these symptoms doesn't automatically lead to testable code. There are some cases I saw in real life that can also give you a really hard time:

- Calling an interface function with data from another interface function
- Having a data object that ended up having functions which use yet another interface internally (however those end up being referenced by this data object)
- Leaving the path of injecting dependent interfaces, but setting them at some point during runtime

I said some cases that can give you a really hard time! Of course, there can be many more examples, and some of these cases are not necessarily bad. The problem is to know on implementation time, how to write the piece of software so it's testable (and still maintainable) later on.

Now you could order 10 different books on unit testing, patterns, design style guides and best practices, read them, get completely confused, and end up making the same mess again (which I have done often enough!).

Or you do it the better way – the TEST-DRIVEN way.

The solution

First, you should stop thinking of tests as a painful and boring task you have to do after writing your code. Instead, you should embrace testing and learn that they can actually help you solve most of the evil circle between stage 2 and 3 which I described above, and reduce the stress in stage 4.

I'll stick to the syntax of VisualStudioTest and Moq for C# at this point, but this is transferable to other languages and test frameworks as well.

So, when you're writing a class *A.cs*, always write a test class *ATests.cs*. Usually that is in a separate test project.

- Add a function 'Setup' with the *[TestInitialize]* attribute. Here you create a Mock object for each interface that class A needs.
- Add a function 'Cleanup' with the *[TestCleanup]* attribute. Here you verify that all test setup has been executed like expected. I'm aware that this is something specific for Moq, but it's recommendable to do this additional test assertion. Just to be sure that the expected outcome didn't happen by chance.

Now, for every function X that you write in your class, create a *[TestMethod]* called *X_Success* in your test class. This will be the first happy path test for the new function. Every test method consists of three parts:

1. Arrange: preparing data objects for input, and mock the behavior of dependent objects
2. Act: Actually call the test function
3. Assert: Check that the returned value is like expect. This part can be codeless, when it's about an expected exception, or when the verification in the cleanup method are enough for the test case

As a next step (before or after finishing X), think about as many execution paths as possible:

- Look at every parameter that is being passed to X, and write tests with different combinations
- Look at the return values of the dependent interfaces, and write tests with different combinations
- Look at 'catch' and 'throw' in X, and make sure you have tests that run into these guys
- Look which exceptions can be thrown from the dependent interfaces, and write tests in which the mocked objects throw these exceptions.

So for each test case, you can define a test method called

- 'X_NegativeInput_ReturnsExpectedValue OR
- X_InvalidInput_ThrowsExpectedException.

When it's about different combinations of parameters for X or mocked behavior – many test frameworks allow you to use one test function for many test cases – like in NUnit for C#:

TestCase-Attribute. When it's about expecting exceptions, you don't have to catch and assert them in the test function, just use ExpectedException.

To think about all these combinations and paths, you will automatically end up covering a lot of sad paths. And long before they can happen in test or production! A very helpful metric is the test coverage (percentage of code which your unit tests cover). In VisualStudio Enterprise, or when you're using ReSharper, there are nice tools for it. But there are a lot of tools, and for most of the languages out there.

Writing these tests will also lead you to the point where you have to define behaviors that you usually wouldn't think so much about at this point. For example:

- Should the function catch an underlying exception or not?
- Should you return an empty object/list, null or throw an Exception?
- Should members of a data structure be nullable or not?

Defining these details, and backing these definitions with unit tests, can help you a lot when writing the next piece of software, which is using the one you just wrote and tested!

Conclusion

Now you may say 'Ok, shut up, I've heard that a million times!' This would make me happy actually. It's not a big deal to write unit tests, I agree! But the big deal is to do this parallel to your programming work. Especially when it's not clear how to solve a problem. Or when a problem consists of many different software steps to solve (like in the example I showed above).

Like in all agile development, you should also accept on this level of work that it's completely OK to throw away stuff that you've written before. When you don't know yet if a class or an interface will be exactly like that in the end, and if it's worth unit testing it already – I say yes, it is! Always unit test it! **Unit testing is not a post-dev-quality-thing**. It is a tool for you to verify your code structure, to write

pieces independently from others, and to think of as many sad paths as possible! You would be surprised of how many times unit tests helped me to avoid error reports coming back from test or production. Because **unit testing is a prophylactic measure** – like brushing your teeth everyday instead of waiting for them to rot, to let the dentist fix them!

Closing the circle for this post, and coming back to my 'Circle' concept of development…

In the problem I described, looking at the different stages, you could see that work is always happening in circles. There is the big circle between starting to work on a feature and releasing it to production. In between, there are smaller circles between programming and testing. In some project management models, this is referred to as waterfalls, iterations or sequences. For me, they are different names for the same thing.

The problem at hand is to make these circles plannable, and keep them under control. And in the end, it always comes down to the people really doing the work – the programmers! Of course, there are a lot of factors and dependencies involved to finish a project in time. But in my experience, when deadlines can't be fulfilled and quality can't be reached, it's because the software is written in huge circles – which means between implementation and test, there has been too much time!

To be able to create complex software in time, and with good quality, **we have to keep the circles small, and always close them**. Please also see my former blog post about Developing in small circles for that. Because this here was mainly about unit-test-driven development. But the whole subject is much bigger.

I hope you enjoyed this post, and I'm looking forward to receiving feedback!

Related content

Understand performance testing and why it is becoming important

8 top mobile testing tools you should use
